



The CRM Digital FTE Factory Final Hackathon 5

Build Your First 24/7 AI Employee: From Incubation to Production

Duration: 48-72 Development Hours | **Team Size:** 1 Student | **Difficulty:** Advanced

Executive Summary

In this final and fifth hackathon, you will implement the complete [Agent Maturity Model](#) by building a real Digital FTE (Full-Time Equivalent) - an AI employee that works 24/7 without breaks, sick days, or vacations.

You'll experience the full evolutionary arc:

1. **Stage 1 - Incubation:** Use Claude Code to explore, prototype, and discover requirements
2. **Stage 2 - Specialization:** Transform your prototype into a production-grade Custom Agent using OpenAI Agents SDK, FastAPI, PostgreSQL, Kafka, and Kubernetes

By the end, you'll have a production-deployed AI employee handling a real business function autonomously across **multiple communication channels**.

Reference: Agent Maturity Model

<https://agentfactory.panaversity.org/docs/General-Agents-Foundations/agent-factory-paradigm/the-2025-inflection-point#the-agent-maturity-model>

The Business Problem: Customer Success FTE

Your client is a growing SaaS company drowning in customer inquiries. They need a **Customer Success FTE** that can:

- Handle customer questions about their product 24/7
- **Accept inquiries from multiple channels:** Email (Gmail), WhatsApp, and Web Form
- Triage and escalate complex issues appropriately
- Track all interactions in a ticket management system (PostgreSQL-based - you will build this)
- Generate daily reports on customer sentiment
- Learn from resolved tickets to improve responses

Note on CRM/Ticket System: For this hackathon, you will build your own ticket management and customer tracking system using PostgreSQL. This serves as your CRM. You are NOT required to integrate with external CRMs like Salesforce or

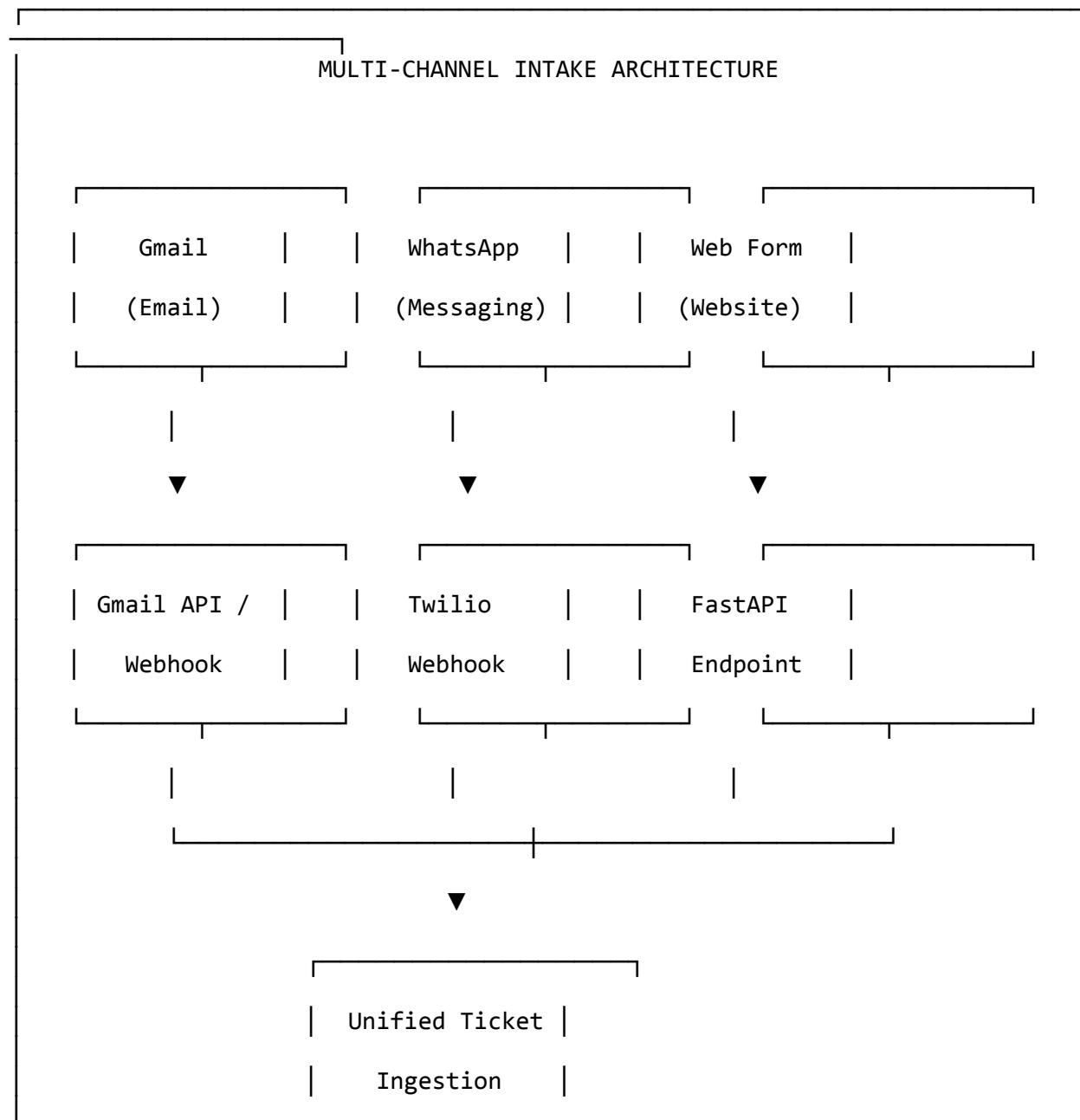
HubSpot. The database schema you create (customers, conversations, tickets, messages tables) IS your CRM system.

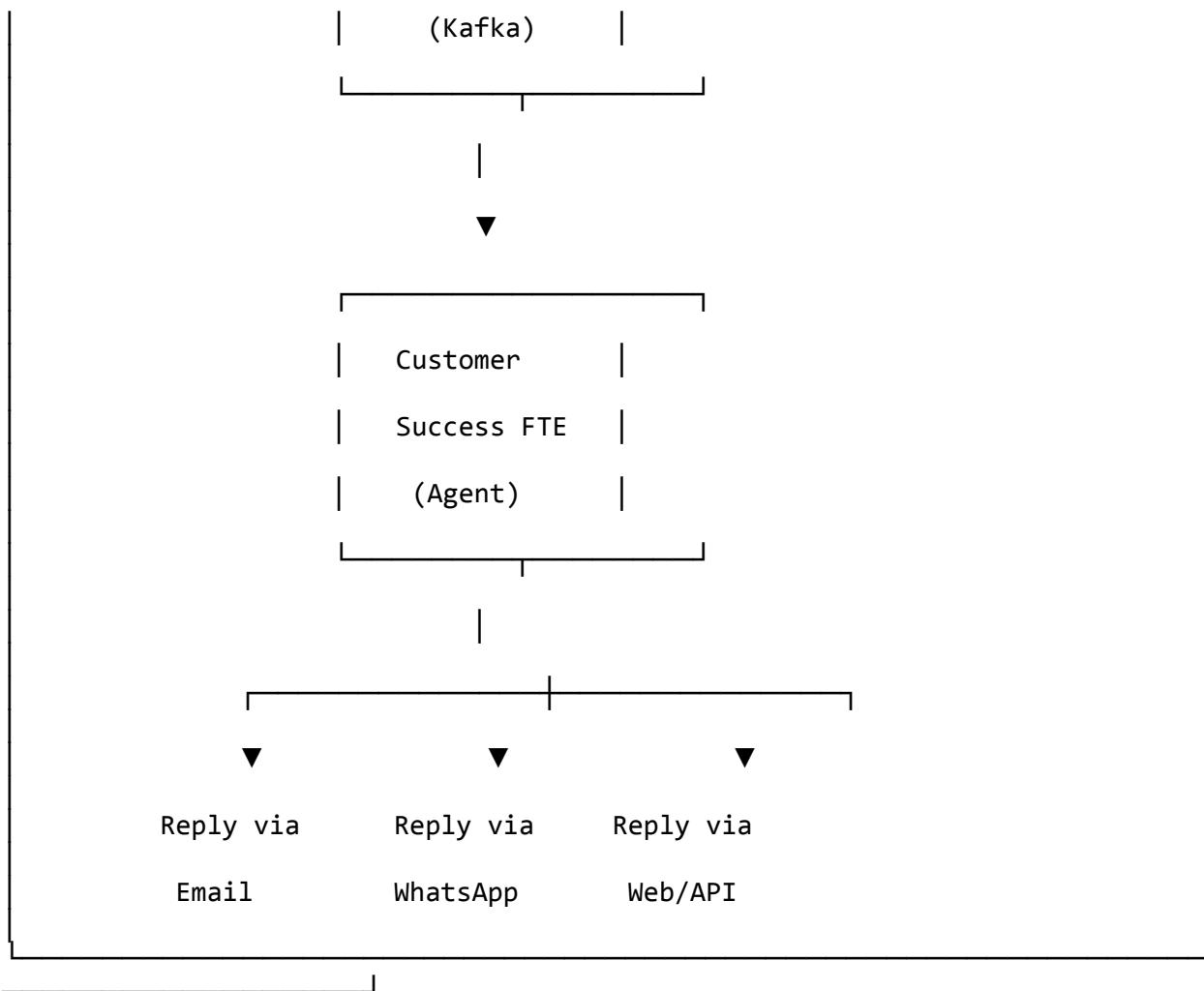
Current cost of human FTE: \$75,000/year + benefits + training + management overhead

Your target: Build a Digital FTE that operates at <\$1,000/year with 24/7 availability

Multi-Channel Architecture Overview

Your FTE will receive support tickets from three channels:





Channel Requirements

Channel	Integration Method	Student Builds	Response Method
Gmail	Gmail API + Pub/Sub or Polling	Webhook handler	Send via Gmail API
WhatsApp	Twilio WhatsApp API	Webhook handler	Reply via Twilio
Web Form	Next.js/HTML Form	Complete form UI	API response + Email

Important: Students must build the complete **Web Support Form** (not the entire website). The form should be a standalone, embeddable component.

Part 1: The Incubation Phase (Hours 1-16)

Objective

Use Claude Code as your **Agent Factory** to explore the problem space, discover hidden requirements, and build a working prototype.

Your Role: Director

You are NOT writing code line-by-line. You are directing an intelligent system toward a goal.

Setup: The Development Dossier

Before starting, prepare your “dossier” - the context Claude Code needs:

```
project-root/
  └── context/
    ├── company-profile.md      # Fake SaaS company details
    ├── product-docs.md         # Product documentation to answer from
    └── sample-tickets.json     # 50+ sample customer inquiries
  (multi-channel)
  └── escalation-rules.md     # When to involve humans
    └── brand-voice.md          # How the company communicates
  └── src/
    ├── channels/               # Channel integrations
    ├── agent/                  # Core agent logic
    └── web-form/               # Support form frontend
  └── tests/                   # Test cases discovered during incubation
  └── specs/                  # Crystallized requirements (output)
```

Exercise 1.1: Initial Exploration (2-3 hours)

Prompt Claude Code with your initial intent:

I need to build a Customer Success AI agent for a SaaS company.

The agent should:

- Answer customer questions from product documentation
- Accept tickets from THREE channels: Gmail, WhatsApp, and a Web Form
- Know when to escalate to humans
- Track all interactions with channel source metadata

I've provided company context in the /context folder.

Help me explore what this system should look like.

Start by analyzing the sample tickets and identifying patterns across channels.

What to observe: - How does Claude Code plan the exploration? - What patterns does it discover in the sample tickets? - Are there channel-specific patterns (email tends to be longer, WhatsApp is more conversational)? - What questions does it ask you for clarification?

Document your discoveries in specs/discovery-log.md

Exercise 1.2: Prototype the Core Loop (4-5 hours)

Direct Claude Code to build the basic interaction:

Based on our analysis, let's prototype the core customer interaction loop.
Build a simple version that:

1. Takes a customer message as input (with channel metadata)
2. Normalizes the message regardless of source channel
3. Searches the product docs for relevant information
4. Generates a helpful response
5. Formats response appropriately for the channel (email vs chat style)
6. Decides if escalation is needed

Use Python. Start simple - we'll iterate.

Iteration prompts to use:

```
# After first version works:  
"This crashes when the customer asks about pricing.  
Add handling for pricing-related queries."  
  
# Channel-specific iteration:  
"WhatsApp messages are much shorter and more casual.  
Adjust response style based on channel."  
  
# Email-specific iteration:  
"Email responses need proper greeting and signature.  
Add channel-aware formatting."  
  
# After testing with real scenarios:  
"The responses are too long for WhatsApp. Customers want concise answers.  
Optimize for brevity on chat channels while keeping detail on email."
```

Exercise 1.3: Add Memory and State (3-4 hours)

Extend the prototype:

Our agent needs to remember context across a conversation.
If a customer asks follow-up questions, the agent should understand
they're continuing the same topic - even if they switch channels!

Add conversation memory. Also track:

- Customer sentiment (is this interaction going well?)
- Topics discussed (for reporting)
- Resolution status (solved/pending/escalated)
- Original channel and any channel switches
- Customer identifier (email address as primary key)

Exercise 1.4: Build the MCP Server (3-4 hours)

Model Context Protocol (MCP) is how your agent will connect to external tools. Build an MCP server that exposes your prototype's capabilities:

Let's expose our customer success agent as an MCP server.

Create tools for:

- search_knowledge_base(query) -> relevant_docs
- create_ticket(customer_id, issue, priority, channel) -> ticket_id
- get_customer_history(customer_id) -> past interactions across ALL channels
- escalate_to_human(ticket_id, reason) -> escalation_id
- send_response(ticket_id, message, channel) -> delivery_status

Follow the MCP specification for tool definitions.

MCP Server Template to Start:

```
# mcp_server.py
from mcp.server import Server
from mcp.types import Tool, TextContent
from enum import Enum

class Channel(str, Enum):
    EMAIL = "email"
    WHATSAPP = "whatsapp"
    WEB_FORM = "web_form"

server = Server("customer-success-fte")

@server.tool("search_knowledge_base")
async def search_kb(query: str) -> str:
    """Search product documentation for relevant information."""
    # Your implementation from incubation
    pass

@server.tool("create_ticket")
async def create_ticket(
    customer_id: str,
    issue: str,
    priority: str,
    channel: Channel
) -> str:
    """Create a support ticket in the system with channel tracking."""
    pass

@server.tool("get_customer_history")
async def get_customer_history(customer_id: str) -> str:
    """Get customer's interaction history across ALL channels."""
    pass
```

```

@server.tool("send_response")
async def send_response(
    ticket_id: str,
    message: str,
    channel: Channel
) -> str:
    """Send response via the appropriate channel."""
    pass

# Add more tools...

if __name__ == "__main__":
    server.run()

```

Exercise 1.5: Define Agent Skills (2-3 hours)

Agent Skills are reusable capabilities your FTE can invoke. Create skill definitions:

Based on what we've built, let's formalize the agent's skills.
Create a skills manifest that defines:

1. Knowledge Retrieval Skill
 - When to use: Customer asks product questions
 - Inputs: query text
 - Outputs: relevant documentation snippets
2. Sentiment Analysis Skill
 - When to use: Every customer message
 - Inputs: message text
 - Outputs: sentiment score, confidence
3. Escalation Decision Skill
 - When to use: After generating response
 - Inputs: conversation context, sentiment trend
 - Outputs: should_escalate (bool), reason
4. Channel Adaptation Skill
 - When to use: Before sending any response
 - Inputs: response text, target channel
 - Outputs: formatted response appropriate for channel
5. Customer Identification Skill
 - When to use: On every incoming message
 - Inputs: message metadata (email, phone, etc.)
 - Outputs: unified customer_id, merged history

Create the skill definitions in a reusable format.

Incubation Deliverables Checklist

Before moving to Stage 2, ensure you have:

- **Working prototype** that handles customer queries from any channel
- **Discovery log** documenting requirements found during exploration
- **MCP server** with 5+ tools exposed (including channel-aware tools)
- **Agent skills** defined and tested
- **Edge cases** documented with handling strategies
- **Escalation rules** crystallized from testing
- **Channel-specific response templates** discovered
- **Performance baseline** (response time, accuracy on test set)

Crystallization Document

Create specs/customer-success-fte-spec.md:

```
# Customer Success FTE Specification

## Purpose
Handle routine customer support queries with speed and consistency across
multiple channels.

## Supported Channels
| Channel | Identifier | Response Style | Max Length |
|-----|-----|-----|-----|
| Email (Gmail) | Email address | Formal, detailed | 500 words |
| WhatsApp | Phone number | Conversational, concise | 160 chars preferred |
| Web Form | Email address | Semi-formal | 300 words |

## Scope
### In Scope
- Product feature questions
- How-to guidance
- Bug report intake
- Feedback collection
- Cross-channel conversation continuity

### Out of Scope (Escalate)
- Pricing negotiations
- Refund requests
- Legal/compliance questions
- Angry customers (sentiment < 0.3)

## Tools
| Tool | Purpose | Constraints |
|-----|-----|-----|
| search_knowledge_base | Find relevant docs | Max 5 results |
| create_ticket | Log interactions | Required for all chats; include channel
```

```
| escalate_to_human | Hand off complex issues | Include full context |
| send_response | Reply to customer | Channel-appropriate formatting |
```

Performance Requirements

- Response time: <3 seconds (processing), <30 seconds (delivery)
- Accuracy: >85% on test set
- Escalation rate: <20%
- Cross-channel identification: >95% accuracy

Guardrails

- NEVER discuss competitor products
 - NEVER promise features not in docs
 - ALWAYS create ticket before responding
 - ALWAYS check sentiment before closing
 - ALWAYS use channel-appropriate tone
-

The Transition: From General Agent to Custom Agent (Hours 15-18)

This is the most critical phase of the hackathon. You're transforming exploratory code into production-ready systems. Many teams struggle here because they don't have a clear methodology. Follow this guide step-by-step.

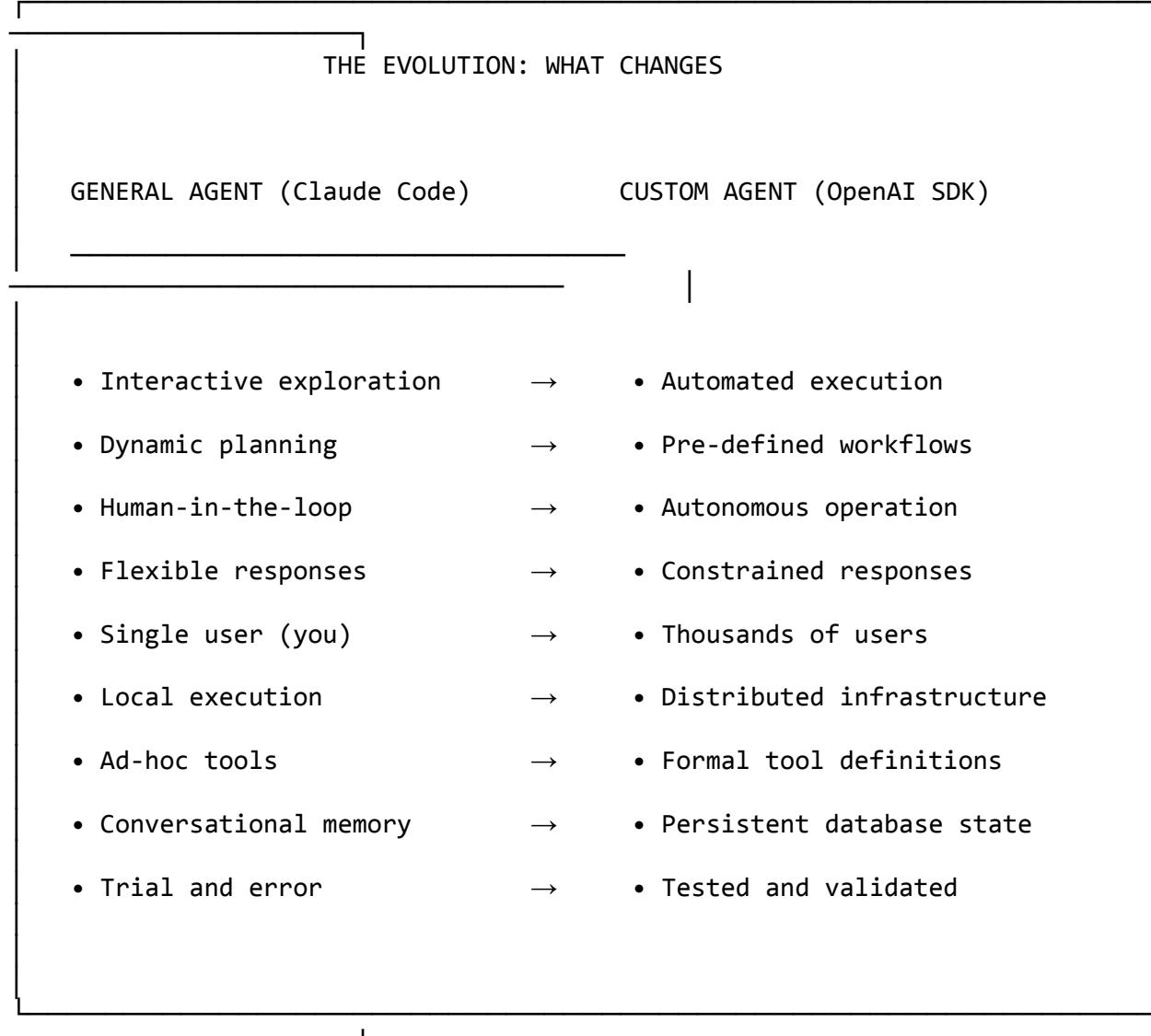
Important: Claude Code Remains Your Development Partner

A common misconception is that you stop using Claude Code (the General Agent) once you transition to building the Custom Agent. **This is incorrect.** Claude Code remains your primary development tool throughout the entire hackathon. During the Specialization Phase, you will use Claude Code to:

- Write the OpenAI Agents SDK implementation code
- Generate the FastAPI endpoints and channel handlers
- Create the PostgreSQL schema and database queries
- Build Kubernetes manifests and Docker configurations
- Debug issues and iterate on your production code

Think of it this way: **Claude Code is the factory that builds the Custom Agent.** The General Agent (Claude Code) doesn't get replaced—it becomes the tool you use to construct, test, and refine your Specialist. This is the essence of the Agent Factory paradigm: General Agents build Custom Agents. You're not coding alone; you're directing Claude Code to help you engineer a production-ready system.

Understanding the Transition



Step 1: Extract Your Discoveries (1 hour)

Before writing any production code, document everything you learned during incubation.

Create `specs/transition-checklist.md`:

```
# Transition Checklist: General → Custom Agent
```

1. Discovered Requirements

List every requirement you discovered during incubation:

- [] Requirement 1: _____
- [] Requirement 2: _____
- [] (Add all requirements)

2. Working Prompts

Copy the exact prompts that worked well:

System Prompt That Worked:

[Paste your working system prompt from Claude Code here]

Tool Descriptions That Worked:

[Paste tool descriptions that gave good results]

3. Edge Cases Found

Edge Case	How It Was Handled	Test Case Needed
Example: Empty message	Return helpful prompt	Yes

4. Response Patterns

What response styles worked best?

- Email: [describe]
- WhatsApp: [describe]
- Web: [describe]

5. Escalation Rules (Finalized)

When did escalation work correctly?

- Trigger 1: _____
- Trigger 2: _____

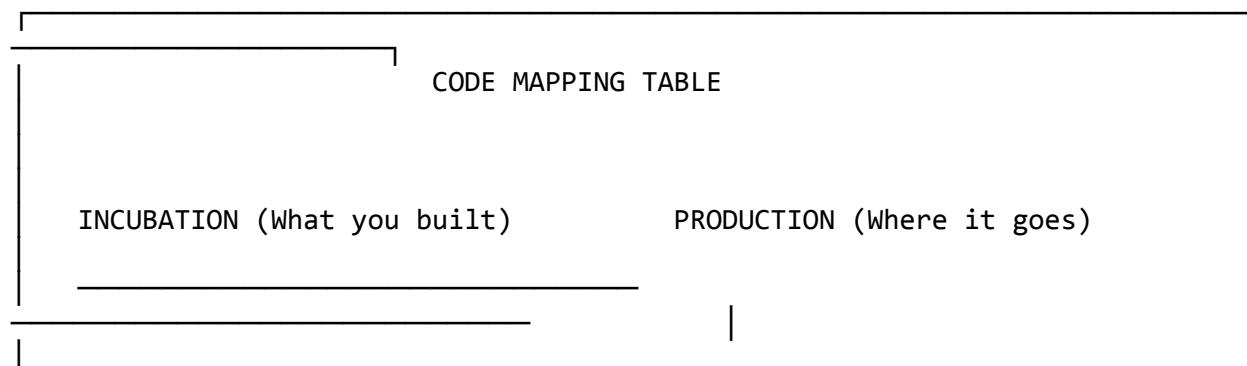
6. Performance Baseline

From your prototype testing:

- Average response time: ____ seconds
- Accuracy on test set: ____%
- Escalation rate: ____%

Step 2: Map Prototype Code to Production Components (1 hour)

Your incubation code needs to be restructured. Here's how each piece maps:



Prototype Python script	→	agent/customer_success_agent.py
MCP server tools	→	@function_tool decorated functions
In-memory conversation	→	PostgreSQL messages table
Print statements	→	Structured logging + Kafka events
Manual testing	→	pytest test suite
Local file storage	→	PostgreSQL + S3/MinIO
Single-threaded	→	Async workers on Kubernetes
Hardcoded config	→	Environment variables + ConfigMaps
Direct API calls	→	Channel handlers with retry logic

Action: Create this file structure for production:

```

production/
  └── agent/
    ├── __init__.py
    ├── customer_success_agent.py      # Your agent definition
    ├── tools.py                      # All @function_tool definitions
    └── prompts.py                   # System prompts (extracted from
prototype)
    └── formatters.py                # Channel-specific response formatting
  └── channels/
    ├── __init__.py
    ├── gmail_handler.py            # Gmail integration
    ├── whatsapp_handler.py        # Twilio/WhatsApp integration
    └── web_form_handler.py        # Web form API
  └── workers/
    ├── __init__.py
    ├── message_processor.py       # Kafka consumer + agent runner
    └── metrics_collector.py       # Background metrics
  └── api/
    ├── __init__.py
    └── main.py                    # FastAPI application
  └── database/
    ├── schema.sql
    ├── migrations/               # PostgreSQL schema
    └── queries.py                # Database migrations

```

```

└── tests/
    ├── test_agent.py
    ├── test_channels.py
    └── test_e2e.py
└── k8s/                      # Kubernetes manifests
└── Dockerfile
└── docker-compose.yml          # Local development
└── requirements.txt

```

Step 3: Transform Your MCP Tools to Production Tools (1 hour)

Your MCP server tools need to become OpenAI Agents SDK `@function_tool` functions.

Before (MCP Server - Incubation):

```

# What you built during incubation
from mcp.server import Server

server = Server("customer-success-fte")

@server.tool("search_knowledge_base")
async def search_kb(query: str) -> str:
    """Search product documentation."""
    # Your prototype implementation
    results = simple_search(query) # Maybe just string matching
    return str(results)

```

After (OpenAI Agents SDK - Production):

```

# production/agent/tools.py

from agents import function_tool
from pydantic import BaseModel
from typing import Optional
import asyncpg

# 1. Define strict input schemas
class KnowledgeSearchInput(BaseModel):
    """Input schema for knowledge base search."""
    query: str
    max_results: int = 5
    category: Optional[str] = None # Optional filter

# 2. Create production tool with proper typing and error handling
@function_tool
async def search_knowledge_base(input: KnowledgeSearchInput) -> str:
    """Search product documentation for relevant information.

```

Use this when the customer asks questions about product features, how to use something, or needs technical information.

```

Args:
    input: Search parameters including query and optional filters

Returns:
    Formatted search results with relevance scores
"""

try:
    # Production: Use database with vector search
    pool = await get_db_pool()
    async with pool.acquire() as conn:
        # Generate embedding for semantic search
        embedding = await generate_embedding(input.query)

        # Query with vector similarity
        results = await conn.fetch("""
            SELECT title, content, category,
                   1 - (embedding <=> $1::vector) as similarity
            FROM knowledge_base
            WHERE ($2::text IS NULL OR category = $2)
            ORDER BY embedding <=> $1::vector
            LIMIT $3
        """, embedding, input.category, input.max_results)

        if not results:
            return "No relevant documentation found. Consider escalating
to human support."

        # Format results for the agent
        formatted = []
        for r in results:
            formatted.append(f"**{r['title']}** (relevance:
{r['similarity']:.2f})\n{r['content'][:500]}")

    return "\n\n---\n\n".join(formatted)

except Exception as e:
    # Log error but return graceful message to agent
    logger.error(f"Knowledge base search failed: {e}")
    return "Knowledge base temporarily unavailable. Please try again or
escalate."

```

Key Differences:

Aspect	MCP (Incubation)	OpenAI SDK (Production)
Input validation	Loose/none	Pydantic BaseModel
Error handling	Crashes	Try/catch with fallbacks

Aspect	MCP (Incubation)	OpenAI SDK (Production)
Database	In-memory/file	PostgreSQL with connection pool
Search	String matching	Vector similarity (pgvector)
Logging	Print statements	Structured logging
Documentation	Basic docstring	Detailed docstring for LLM

Step 4: Transform Your System Prompt (30 minutes)

Your working prompt from incubation needs to be formalized with explicit constraints.

Before (Incubation - Conversational):

You're a helpful customer support agent. Answer questions about our product. Be nice and escalate if needed.

After (Production - Explicit Constraints):

```
# production/agent/prompts.py

CUSTOMER_SUCCESS_SYSTEM_PROMPT = """You are a Customer Success agent for TechCorp SaaS.

## Your Purpose
Handle routine customer support queries with speed, accuracy, and empathy across multiple channels.

## Channel Awareness
You receive messages from three channels. Adapt your communication style:
- **Email**: Formal, detailed responses. Include proper greeting and signature.
- **WhatsApp**: Concise, conversational. Keep responses under 300 characters when possible.
- **Web Form**: Semi-formal, helpful. Balance detail with readability.

## Required Workflow (ALWAYS follow this order)
1. FIRST: Call `create_ticket` to log the interaction
2. THEN: Call `get_customer_history` to check for prior context
3. THEN: Call `search_knowledge_base` if product questions arise
4. FINALLY: Call `send_response` to reply (NEVER respond without this tool)

## Hard Constraints (NEVER violate)
- NEVER discuss pricing → escalate immediately with reason "pricing_inquiry"
- NEVER promise features not in documentation
- NEVER process refunds → escalate with reason "refund_request"
```

- NEVER share internal processes or system details
- NEVER respond without using send_response tool
- NEVER exceed response limits: Email=500 words, WhatsApp=300 chars, Web=300 words

```
## Escalation Triggers (MUST escalate when detected)
- Customer mentions "lawyer", "legal", "sue", or "attorney"
- Customer uses profanity or aggressive language (sentiment < 0.3)
- Cannot find relevant information after 2 search attempts
- Customer explicitly requests human help
- Customer on WhatsApp sends "human", "agent", or "representative"
```

```
## Response Quality Standards
- Be concise: Answer the question directly, then offer additional help
- Be accurate: Only state facts from knowledge base or verified customer data
- Be empathetic: Acknowledge frustration before solving problems
- Be actionable: End with clear next step or question
```

```
## Context Variables Available
- {{customer_id}}: Unique customer identifier
- {{conversation_id}}: Current conversation thread
- {{channel}}: Current channel (email/whatsapp/web_form)
- {{ticket_subject}}: Original subject/topic
"""
```

Step 5: Create the Transition Test Suite (1 hour)

Before building production infrastructure, ensure your agent logic still works.

```
# production/tests/test_transition.py
"""
Transition Tests: Verify agent behavior matches incubation discoveries.
Run these BEFORE deploying to production.
"""

import pytest
from agent.customer_success_agent import customer_success_agent
from agent.tools import search_knowledge_base, create_ticket

class TestTransitionFromIncubation:
    """Tests based on edge cases discovered during incubation."""

    @pytest.mark.asyncio
    async def test_edge_case_empty_message(self):
        """Edge case #1 from incubation: Empty messages."""
        result = await customer_success_agent.run(
            messages=[{"role": "user", "content": ""}],
            context={"channel": "web_form", "customer_id": "test-1"}
        )
        # Should ask for clarification, not crash
```

```

        assert "help" in result.output.lower() or "question" in
result.output.lower()

@pytest.mark.asyncio
async def test_edge_case_pricing_escalation(self):
    """Edge case #2 from incubation: Pricing questions must escalate."""
    result = await customer_success_agent.run(
        messages=[{"role": "user", "content": "How much does the
enterprise plan cost?"}],
        context={"channel": "email", "customer_id": "test-2"})
    )
    # Must escalate, never answer
    assert result.escalated == True
    assert "pricing" in result.escalation_reason.lower()

@pytest.mark.asyncio
async def test_edge_case_angry_customer(self):
    """Edge case #3 from incubation: Angry customers need care."""
    result = await customer_success_agent.run(
        messages=[{"role": "user", "content": "This is RIDICULOUS! Your
product is BROKEN!"}],
        context={"channel": "whatsapp", "customer_id": "test-3"})
    )
    # Should show empathy or escalate
    assert result.escalated == True or "understand" in
result.output.lower()

@pytest.mark.asyncio
async def test_channel_response_length_email(self):
    """Verify email responses are appropriately detailed."""
    result = await customer_success_agent.run(
        messages=[{"role": "user", "content": "How do I reset my
password?"}],
        context={"channel": "email", "customer_id": "test-4"})
    )
    # Email should have greeting and signature
    assert "dear" in result.output.lower() or "hello" in
result.output.lower()

@pytest.mark.asyncio
async def test_channel_response_length_whatsapp(self):
    """Verify WhatsApp responses are concise."""
    result = await customer_success_agent.run(
        messages=[{"role": "user", "content": "How do I reset my
password?"}],
        context={"channel": "whatsapp", "customer_id": "test-5"})
    )
    # WhatsApp should be short
    assert len(result.output) < 500 # Much shorter than email

```

```

@pytest.mark.asyncio
async def test_tool_execution_order(self):
    """Verify tools are called in correct order."""
    result = await customer_success_agent.run(
        messages=[{"role": "user", "content": "I need help with the
API"}],
        context={"channel": "web_form", "customer_id": "test-6"}
    )

    # Extract tool call order
    tool_names = [tc.tool_name for tc in result.tool_calls]

    # create_ticket should be first
    assert tool_names[0] == "create_ticket"
    # send_response should be last
    assert tool_names[-1] == "send_response"

class TestToolMigration:
    """Verify tools work the same as MCP versions."""

    @pytest.mark.asyncio
    async def test_knowledge_search_returns_results(self):
        """Knowledge search should return formatted results."""
        from agent.tools import KnowledgeSearchInput

        result = await search_knowledge_base(
            KnowledgeSearchInput(query="password reset", max_results=3)
        )

        assert result is not None
        assert len(result) > 0
        assert "password" in result.lower()

    @pytest.mark.asyncio
    async def test_knowledge_search_handles_no_results(self):
        """Knowledge search should handle no results gracefully."""
        from agent.tools import KnowledgeSearchInput

        result = await search_knowledge_base(
            KnowledgeSearchInput(query="xyznonexistentquery123",
max_results=3)
        )

        # Should return helpful message, not crash
        assert "no" in result.lower() or "not found" in result.lower()

```

Step 6: The Transition Checklist

Use this checklist to ensure you haven't missed anything:

```

## Pre-Transition Checklist

### From Incubation (Must Have Before Proceeding)
- [ ] Working prototype that handles basic queries
- [ ] Documented edge cases (minimum 10)
- [ ] Working system prompt
- [ ] MCP tools defined and tested
- [ ] Channel-specific response patterns identified
- [ ] Escalation rules finalized
- [ ] Performance baseline measured

```

```

### Transition Steps
- [ ] Created production folder structure
- [ ] Extracted prompts to prompts.py
- [ ] Converted MCP tools to @function_tool
- [ ] Added Pydantic input validation to all tools
- [ ] Added error handling to all tools
- [ ] Created transition test suite
- [ ] All transition tests passing

```

```

### Ready for Production Build
- [ ] Database schema designed
- [ ] Kafka topics defined
- [ ] Channel handlers outlined
- [ ] Kubernetes resource requirements estimated
- [ ] API endpoints listed

```

Common Transition Mistakes (Avoid These!)

Mistake	Why It Happens	How to Avoid
Skipping documentation	"I remember what worked"	Write it down immediately
Copying code directly	"It worked in prototype"	Refactor for production patterns
Ignoring edge cases	"We'll fix those later"	Test edge cases first
Hardcoding values	"Just for now"	Use config from day 1
No error handling	"It didn't crash before"	Everything can fail at scale
Forgetting channel differences	"One response fits all"	Test each channel separately

Transition Complete Criteria

You're ready to proceed to Part 2 (Specialization) when:

1. All transition tests pass
 2. Prompts are extracted and documented
 3. Tools have proper input validation
 4. Error handling exists for all tools
 5. Edge cases are documented with test cases
 6. Production folder structure is created
-

Part 2: The Specialization Phase (Hours 17-40)

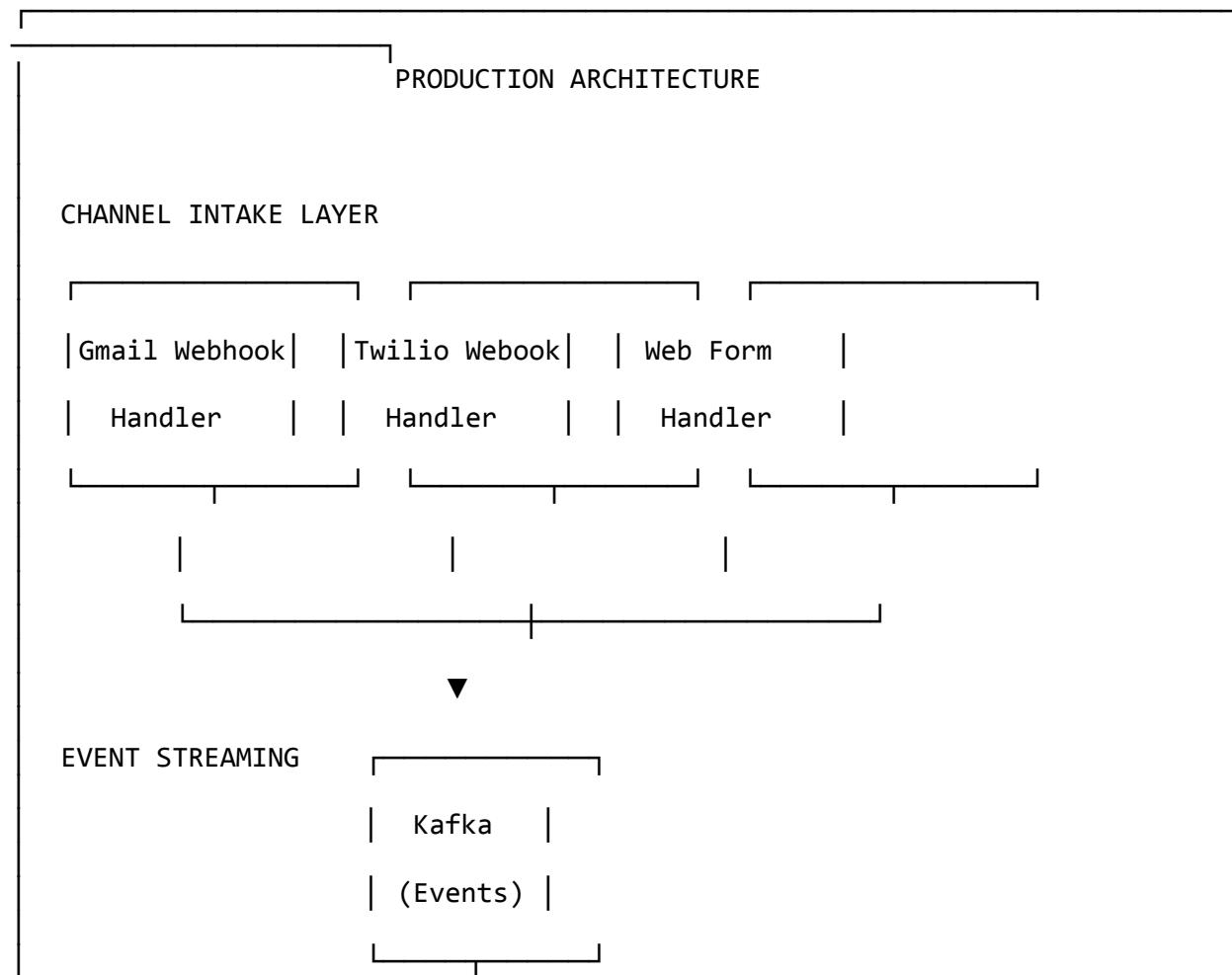
Objective

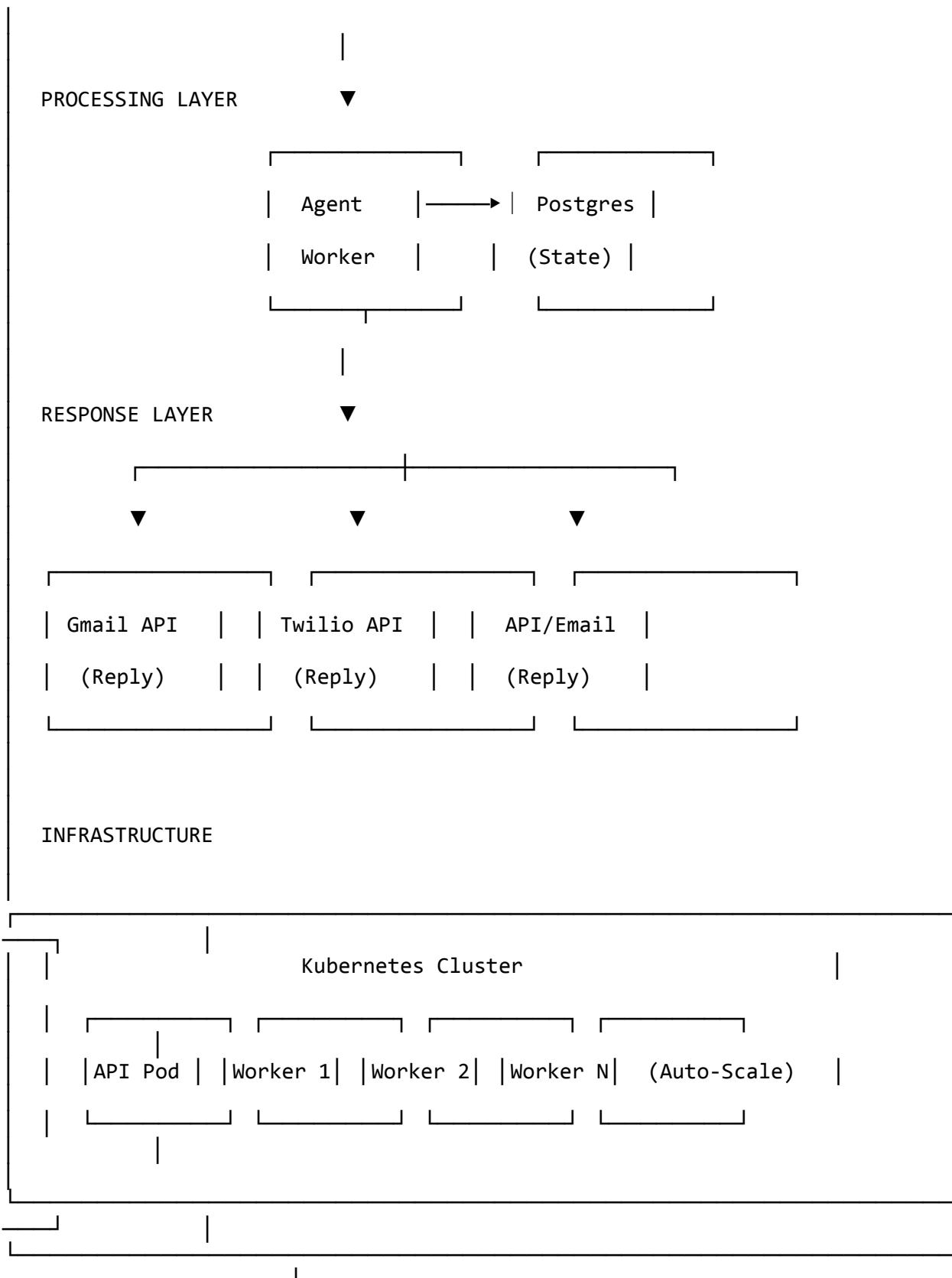
Transform your incubated prototype into a production-grade Custom Agent that runs 24/7 on Kubernetes with Kafka for event streaming and **multi-channel intake**.

Your Role: Builder

You are now engineering for reliability, scale, and governance.

Architecture Overview





Exercise 2.1: Database Schema - Your CRM System (2-3 hours)

Design your PostgreSQL schema for production state management with multi-channel support. **This database IS your CRM/ticket management system** - you are building a custom solution that tracks customers, conversations, tickets, and all interactions across channels.

Why build your own CRM? In production environments, companies often integrate with enterprise CRMs (Salesforce, HubSpot). However, for this hackathon, building your own system teaches you the fundamentals of customer data management, and your PostgreSQL-based solution provides all the functionality needed for a working Digital FTE.

```
-- schema.sql
-- =====
-- CUSTOMER SUCCESS FTE - CRM/TICKET MANAGEMENT SYSTEM
-- =====
-- This PostgreSQL schema serves as your complete CRM system for tracking:
-- - Customers (unified across all channels)
-- - Conversations and message history
-- - Support tickets and their lifecycle
-- - Knowledge base for AI responses
-- - Performance metrics and reporting
-- =====
-- Customers table (unified across channels) - YOUR CUSTOMER DATABASE
CREATE TABLE customers (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    email VARCHAR(255) UNIQUE,
    phone VARCHAR(50),
    name VARCHAR(255),
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    metadata JSONB DEFAULT '{}'
);
-- Customer identifiers (for cross-channel matching)
CREATE TABLE customer_identifiers (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    customer_id UUID REFERENCES customers(id),
    identifier_type VARCHAR(50) NOT NULL, -- 'email', 'phone', 'whatsapp'
    identifier_value VARCHAR(255) NOT NULL,
    verified BOOLEAN DEFAULT FALSE,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    UNIQUE(identifier_type, identifier_value)
);
-- Conversations table
```

```

CREATE TABLE conversations (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    customer_id UUID REFERENCES customers(id),
    initial_channel VARCHAR(50) NOT NULL, -- 'email', 'whatsapp', 'web_form'
    started_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    ended_at TIMESTAMP WITH TIME ZONE,
    status VARCHAR(50) DEFAULT 'active',
    sentiment_score DECIMAL(3,2),
    resolution_type VARCHAR(50),
    escalated_to VARCHAR(255),
    metadata JSONB DEFAULT '{}'
);

-- Messages table (with channel tracking)
CREATE TABLE messages (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    conversation_id UUID REFERENCES conversations(id),
    channel VARCHAR(50) NOT NULL, -- 'email', 'whatsapp', 'web_form'
    direction VARCHAR(20) NOT NULL, -- 'inbound', 'outbound'
    role VARCHAR(20) NOT NULL, -- 'customer', 'agent', 'system'
    content TEXT NOT NULL,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    tokens_used INTEGER,
    latency_ms INTEGER,
    tool_calls JSONB DEFAULT '[]',
    channel_message_id VARCHAR(255), -- External ID (Gmail message ID, Twilio
SID)
    delivery_status VARCHAR(50) DEFAULT 'pending' -- 'pending', 'sent',
'delivered', 'failed'
);

-- Tickets table
CREATE TABLE tickets (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    conversation_id UUID REFERENCES conversations(id),
    customer_id UUID REFERENCES customers(id),
    source_channel VARCHAR(50) NOT NULL,
    category VARCHAR(100),
    priority VARCHAR(20) DEFAULT 'medium',
    status VARCHAR(50) DEFAULT 'open',
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    resolved_at TIMESTAMP WITH TIME ZONE,
    resolution_notes TEXT
);

-- Knowledge base entries
CREATE TABLE knowledge_base (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    title VARCHAR(500) NOT NULL,
    content TEXT NOT NULL,

```

```

category VARCHAR(100),
embedding VECTOR(1536), -- For semantic search
created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
updated_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

-- Channel configurations
CREATE TABLE channel_configs (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    channel VARCHAR(50) UNIQUE NOT NULL,
    enabled BOOLEAN DEFAULT TRUE,
    config JSONB NOT NULL, -- API keys, webhook URLs, etc.
    response_template TEXT,
    max_response_length INTEGER,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

-- Agent performance metrics
CREATE TABLE agent_metrics (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    metric_name VARCHAR(100) NOT NULL,
    metric_value DECIMAL(10,4) NOT NULL,
    channel VARCHAR(50), -- Optional: channel-specific metrics
    dimensions JSONB DEFAULT '{}',
    recorded_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

-- Indexes for performance
CREATE INDEX idx_customers_email ON customers(email);
CREATE INDEX idx_customer_identifiers_value ON
customer_identifiers(identifier_value);
CREATE INDEX idx_conversations_customer ON conversations(customer_id);
CREATE INDEX idx_conversations_status ON conversations(status);
CREATE INDEX idx_conversations_channel ON conversations(initial_channel);
CREATE INDEX idx_messages_conversation ON messages(conversation_id);
CREATE INDEX idx_messages_channel ON messages(channel);
CREATE INDEX idx_tickets_status ON tickets(status);
CREATE INDEX idx_tickets_channel ON tickets(source_channel);
CREATE INDEX idx_knowledge_embedding ON knowledge_base USING ivfflat
(embedding vector_cosine_ops);

```

Exercise 2.2: Channel Integrations (4-5 hours)

Build the intake handlers for each channel:

Gmail Integration

```
# channels/gmail_handler.py
```

```
from google.oauth2.credentials import Credentials
from googleapiclient.discovery import build
```

```

from google.cloud import pubsub_v1
import base64
import email
from email.mime.text import MIMEText
from datetime import datetime
import json

class GmailHandler:
    def __init__(self, credentials_path: str):
        self.credentials =
            Credentials.from_authorized_user_file(credentials_path)
        self.service = build('gmail', 'v1', credentials=self.credentials)

    @async def setup_push_notifications(self, topic_name: str):
        """Set up Gmail push notifications via Pub/Sub."""
        request = {
            'labelIds': ['INBOX'],
            'topicName': topic_name,
            'labelFilterAction': 'include'
        }
        return self.service.users().watch(userId='me',
                                          body=request).execute()

    @async def process_notification(self, pubsub_message: dict) -> dict:
        """Process incoming Pub/Sub notification from Gmail."""
        history_id = pubsub_message.get('historyId')

        # Get new messages since last history ID
        history = self.service.users().history().list(
            userId='me',
            startHistoryId=history_id,
            historyTypes=['messageAdded']
        ).execute()

        messages = []
        for record in history.get('history', []):
            for msg_added in record.get('messagesAdded', []):
                msg_id = msg_added['message']['id']
                message = await self.get_message(msg_id)
                messages.append(message)

        return messages

    @async def get_message(self, message_id: str) -> dict:
        """Fetch and parse a Gmail message."""
        msg = self.service.users().messages().get(
            userId='me',
            id=message_id,
            format='full'
        ).execute()

```

```

headers = {h['name']: h['value'] for h in msg['payload']['headers']}

# Extract body
body = self._extract_body(msg['payload'])

return {
    'channel': 'email',
    'channel_message_id': message_id,
    'customer_email': self._extract_email(headers.get('From', '')),
    'subject': headers.get('Subject', ''),
    'content': body,
    'received_at': datetime.utcnow().isoformat(),
    'thread_id': msg.get('threadId'),
    'metadata': {
        'headers': headers,
        'labels': msg.get('labelIds', [])
    }
}

def _extract_body(self, payload: dict) -> str:
    """Extract text body from email payload."""
    if 'body' in payload and payload['body'].get('data'):
        return
base64.urlsafe_b64decode(payload['body']['data']).decode('utf-8')

    if 'parts' in payload:
        for part in payload['parts']:
            if part['mimeType'] == 'text/plain':
                return
base64.urlsafe_b64decode(part['body']['data']).decode('utf-8')

    return ''

def _extract_email(self, from_header: str) -> str:
    """Extract email address from From header."""
    import re
    match = re.search(r'<(.+?)>', from_header)
    return match.group(1) if match else from_header

async def send_reply(self, to_email: str, subject: str, body: str,
thread_id: str = None) -> dict:
    """Send email reply."""
    message = MIMEText(body)
    message['to'] = to_email
    message['subject'] = f'Re: {subject}' if not
subject.startswith('Re:') else subject

    raw = base64.urlsafe_b64encode(message.as_bytes()).decode('utf-8')

```

```

    send_request = {'raw': raw}
    if thread_id:
        send_request['threadId'] = thread_id

    result = self.service.users().messages().send(
        userId='me',
        body=send_request
    ).execute()

    return {
        'channel_message_id': result['id'],
        'delivery_status': 'sent'
    }
}

```

WhatsApp Integration (via Twilio)

channels/whatsapp_handler.py

```

from twilio.rest import Client
from twilio.request_validator import RequestValidator
from fastapi import Request, HTTPException
import os
from datetime import datetime

class WhatsAppHandler:
    def __init__(self):
        self.account_sid = os.getenv('TWILIO_ACCOUNT_SID')
        self.auth_token = os.getenv('TWILIO_AUTH_TOKEN')
        self.whatsapp_number = os.getenv('TWILIO_WHATSAPP_NUMBER') # e.g.,
'whatsapp:+14155238886'
        self.client = Client(self.account_sid, self.auth_token)
        self.validator = RequestValidator(self.auth_token)

    async def validate_webhook(self, request: Request) -> bool:
        """Validate incoming Twilio webhook signature."""
        signature = request.headers.get('X-Twilio-Signature', '')
        url = str(request.url)
        form_data = await request.form()
        params = dict(form_data)

        return self.validator.validate(url, params, signature)

    async def process_webhook(self, form_data: dict) -> dict:
        """Process incoming WhatsApp message from Twilio webhook."""
        return {
            'channel': 'whatsapp',
            'channel_message_id': form_data.get('MessageSid'),
            'customer_phone': form_data.get('From', '').replace('whatsapp:', ''),
            'content': form_data.get('Body', ''),
            'received_at': datetime.utcnow().isoformat(),
        }
}

```

```

        'metadata': {
            'num_media': form_data.get('NumMedia', '0'),
            'profile_name': form_data.get('ProfileName'),
            'wa_id': form_data.get('WaId'),
            'status': form_data.get('SmsStatus')
        }
    }

async def send_message(self, to_phone: str, body: str) -> dict:
    """Send WhatsApp message via Twilio."""
    # Ensure phone number is in WhatsApp format
    if not to_phone.startswith('whatsapp:'):
        to_phone = f'whatsapp:{to_phone}'

    message = self.client.messages.create(
        body=body,
        from_=self.whatsapp_number,
        to=to_phone
    )

    return {
        'channel_message_id': message.sid,
        'delivery_status': message.status # 'queued', 'sent',
        'delivered', 'failed'
    }

    def format_response(self, response: str, max_length: int = 1600) ->
list[str]:
    """Format and split response for WhatsApp (max 1600 chars per
message)."""
    if len(response) <= max_length:
        return [response]

    # Split into multiple messages
    messages = []
    while response:
        if len(response) <= max_length:
            messages.append(response)
            break

        # Find a good break point
        break_point = response.rfind('. ', 0, max_length)
        if break_point == -1:
            break_point = response.rfind(' ', 0, max_length)
        if break_point == -1:
            break_point = max_length

        messages.append(response[:break_point + 1].strip())
        response = response[break_point + 1:].strip()

```

```
    return messages
```

Web Support Form (Required Build)

This is a required deliverable - students must build the complete form UI.

```
# channels/web_form_handler.py

from fastapi import APIRouter, HTTPException
from pydantic import BaseModel, EmailStr, validator
from datetime import datetime
from typing import Optional
import uuid

router = APIRouter(prefix="/support", tags=["support-form"])

class SupportFormSubmission(BaseModel):
    """Support form submission model with validation."""
    name: str
    email: EmailStr
    subject: str
    category: str # 'general', 'technical', 'billing', 'feedback'
    message: str
    priority: Optional[str] = 'medium'
    attachments: Optional[list[str]] = [] # Base64 encoded files or URLs

    @validator('name')
    def name_must_not_be_empty(cls, v):
        if not v or len(v.strip()) < 2:
            raise ValueError('Name must be at least 2 characters')
        return v.strip()

    @validator('message')
    def message_must_have_content(cls, v):
        if not v or len(v.strip()) < 10:
            raise ValueError('Message must be at least 10 characters')
        return v.strip()

    @validator('category')
    def category_must_be_valid(cls, v):
        valid_categories = ['general', 'technical', 'billing', 'feedback',
'bug_report']
        if v not in valid_categories:
            raise ValueError(f'Category must be one of: {valid_categories}')
        return v

class SupportFormResponse(BaseModel):
    """Response model for form submission."""
    ticket_id: str
```

```

message: str
estimated_response_time: str

@router.post("/submit", response_model=SupportFormResponse)
async def submit_support_form(submission: SupportFormSubmission):
    """
    Handle support form submission.

    This endpoint:
    1. Validates the submission
    2. Creates a ticket in the system
    3. Publishes to Kafka for agent processing
    4. Returns confirmation to user
    """

    ticket_id = str(uuid.uuid4())

    # Create normalized message for agent
    message_data = {
        'channel': 'web_form',
        'channel_message_id': ticket_id,
        'customer_email': submission.email,
        'customer_name': submission.name,
        'subject': submission.subject,
        'content': submission.message,
        'category': submission.category,
        'priority': submission.priority,
        'received_at': datetime.utcnow().isoformat(),
        'metadata': {
            'form_version': '1.0',
            'attachments': submission.attachments
        }
    }

    # Publish to Kafka
    await publish_to_kafka('fte.tickets.incoming', message_data)

    # Store initial ticket
    await create_ticket_record(ticket_id, message_data)

    return SupportFormResponse(
        ticket_id=ticket_id,
        message="Thank you for contacting us! Our AI assistant will respond
shortly.",
        estimated_response_time="Usually within 5 minutes"
    )

@router.get("/ticket/{ticket_id}")
async def get_ticket_status(ticket_id: str):
    """
    Get status and conversation history for a ticket.
    """
    ticket = await get_ticket_by_id(ticket_id)

```

```

if not ticket:
    raise HTTPException(status_code=404, detail="Ticket not found")

return {
    'ticket_id': ticket_id,
    'status': ticket['status'],
    'messages': ticket['messages'],
    'created_at': ticket['created_at'],
    'last_updated': ticket['last_updated']
}

```

React/Next.js Web Support Form Component (Required):

```

// web-form/SupportForm.jsx

import React, { useState } from 'react';

const CATEGORIES = [
    { value: 'general', label: 'General Question' },
    { value: 'technical', label: 'Technical Support' },
    { value: 'billing', label: 'Billing Inquiry' },
    { value: 'bug_report', label: 'Bug Report' },
    { value: 'feedback', label: 'Feedback' }
];

const PRIORITIES = [
    { value: 'low', label: 'Low - Not urgent' },
    { value: 'medium', label: 'Medium - Need help soon' },
    { value: 'high', label: 'High - Urgent issue' }
];

export default function SupportForm({ apiEndpoint = '/api/support/submit' })
{
    const [formData, setFormData] = useState({
        name: '',
        email: '',
        subject: '',
        category: 'general',
        priority: 'medium',
        message: ''
    });

    const [status, setStatus] = useState('idle'); // 'idle', 'submitting',
    'success', 'error'
    const [ticketId, setTicketId] = useState(null);
    const [error, setError] = useState(null);

    const handleChange = (e) => {
        const { name, value } = e.target;
        setFormData(prev => ({ ...prev, [name]: value }));
    }
}

```

```

};

const validateForm = () => {
  if (formData.name.trim().length < 2) {
    setError('Please enter your name (at least 2 characters)');
    return false;
  }
  if (!/^[^@\s]+@[^\s@]+\.[^\s@]+$/ .test(formData.email)) {
    setError('Please enter a valid email address');
    return false;
  }
  if (formData.subject.trim().length < 5) {
    setError('Please enter a subject (at least 5 characters)');
    return false;
  }
  if (formData.message.trim().length < 10) {
    setError('Please describe your issue in more detail (at least 10
characters)');
    return false;
  }
  return true;
};

const handleSubmit = async (e) => {
  e.preventDefault();
  setError(null);

  if (!validateForm()) return;

  setStatus('submitting');

  try {
    const response = await fetch(apiEndpoint, {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify(formData)
    });

    if (!response.ok) {
      const errorData = await response.json();
      throw new Error(errorData.detail || 'Submission failed');
    }

    const data = await response.json();
    setTicketId(data.ticket_id);
    setStatus('success');
  } catch (err) {
    setError(err.message);
    setStatus('error');
  }
}

```

```

};

if (status === 'success') {
  return (
    <div className="max-w-2xl mx-auto p-6 bg-white rounded-lg shadow-md">
      <div className="text-center">
        <div className="w-16 h-16 bg-green-100 rounded-full flex items-center justify-center mx-auto mb-4">
          <svg className="w-8 h-8 text-green-500" fill="none" stroke="currentColor" viewBox="0 0 24 24">
            <path strokeLinecap="round" strokeLinejoin="round" strokeWidth={2} d="M5 13l4 4L19 7" />
          </svg>
        </div>
        <h2 className="text-2xl font-bold text-gray-900 mb-2">Thank You!</h2>
        <p className="text-gray-600 mb-4">Your support request has been submitted successfully.</p>
        <div className="bg-gray-50 rounded-lg p-4 mb-4">
          <p className="text-sm text-gray-500">Your Ticket ID</p>
          <p className="text-lg font-mono font-bold text-gray-900">{ticketId}</p>
        </div>
        <p className="text-sm text-gray-500">
          Our AI assistant will respond to your email within 5 minutes.
          For urgent issues, responses are prioritized automatically.
        </p>
        <button
          onClick={() => {
            setStatus('idle');
            setFormData({ name: '', email: '', subject: '', category: 'general', priority: 'medium', message: '' });
          }}
          className="mt-6 px-4 py-2 bg-blue-600 text-white rounded-lg hover:bg-blue-700 transition-colors"
        >
          Submit Another Request
        </button>
      </div>
    </div>
  );
}

return (
  <div className="max-w-2xl mx-auto p-6 bg-white rounded-lg shadow-md">
    <h2 className="text-2xl font-bold text-gray-900 mb-2">Contact Support</h2>
    <p className="text-gray-600 mb-6">
      Fill out the form below and our AI-powered support team will get back to you shortly.
    </p>
  </div>
);

```

```

</p>

{error && (
  <div className="mb-4 p-4 bg-red-50 border border-red-200 rounded-lg
text-red-700">
    {error}
  </div>
)}

<form onSubmit={handleSubmit} className="space-y-6">
  {/* Name Field */}
  <div>
    <label htmlFor="name" className="block text-sm font-medium
text-gray-700 mb-1">
      Your Name *
    </label>
    <input
      type="text"
      id="name"
      name="name"
      value={formData.name}
      onChange={handleChange}
      required
      className="w-full px-4 py-2 border border-gray-300 rounded-lg
focus:ring-2 focus:ring-blue-500 focus:border-transparent"
      placeholder="John Doe"
    />
  </div>

  {/* Email Field */}
  <div>
    <label htmlFor="email" className="block text-sm font-medium
text-gray-700 mb-1">
      Email Address *
    </label>
    <input
      type="email"
      id="email"
      name="email"
      value={formData.email}
      onChange={handleChange}
      required
      className="w-full px-4 py-2 border border-gray-300 rounded-lg
focus:ring-2 focus:ring-blue-500 focus:border-transparent"
      placeholder="john@example.com"
    />
  </div>

  {/* Subject Field */}
  <div>

```

```

        <label htmlFor="subject" className="block text-sm font-medium
text-gray-700 mb-1">
            Subject *
        </label>
        <input
            type="text"
            id="subject"
            name="subject"
            value={formData.subject}
            onChange={handleChange}
            required
            className="w-full px-4 py-2 border border-gray-300 rounded-lg
focus:ring-2 focus:ring-blue-500 focus:border-transparent"
            placeholder="Brief description of your issue"
        />
    </div>

    {/* Category and Priority Row */}
    <div className="grid grid-cols-1 md:grid-cols-2 gap-4">
        <div>
            <label htmlFor="category" className="block text-sm font-medium
text-gray-700 mb-1">
                Category *
            </label>
            <select
                id="category"
                name="category"
                value={formData.category}
                onChange={handleChange}
                className="w-full px-4 py-2 border border-gray-300 rounded-lg
focus:ring-2 focus:ring-blue-500 focus:border-transparent"
            >
                {CATEGORIES.map(cat => (
                    <option key={cat.value}
value={cat.value}>{cat.label}</option>
                )))
            </select>
        </div>

        <div>
            <label htmlFor="priority" className="block text-sm font-medium
text-gray-700 mb-1">
                Priority
            </label>
            <select
                id="priority"
                name="priority"
                value={formData.priority}
                onChange={handleChange}
                className="w-full px-4 py-2 border border-gray-300 rounded-lg
focus:ring-2 focus:ring-blue-500 focus:border-transparent"
            >

```

```

focus:ring-2 focus:ring-blue-500 focus:border-transparent"
    >
        {PRIORITIES.map(pri => (
            <option key={pri.value}
value={pri.value}>{pri.label}</option>
        )))
    </select>
</div>
</div>

/* Message Field */
<div>
    <label htmlFor="message" className="block text-sm font-medium
text-gray-700 mb-1">
        How can we help? *
    </label>
    <textarea
        id="message"
        name="message"
        value={formData.message}
        onChange={handleChange}
        required
        rows={6}
        className="w-full px-4 py-2 border border-gray-300 rounded-lg
focus:ring-2 focus:ring-blue-500 focus:border-transparent resize-none"
        placeholder="Please describe your issue or question in detail...">
    />
    <p className="mt-1 text-sm text-gray-500">
        {formData.message.length}/1000 characters
    </p>
</div>

/* Submit Button */
<button
    type="submit"
    disabled={status === 'submitting'}
    className={`${`w-full py-3 px-4 rounded-lg font-medium text-white
transition-colors ${`status === 'submitting'
    ? 'bg-gray-400 cursor-not-allowed'
    : 'bg-blue-600 hover:bg-blue-700'}`}`}
    >
    {status === 'submitting' ? (
        <span className="flex items-center justify-center">
            <svg className="animate-spin -ml-1 mr-3 h-5 w-5 text-white
fill="none" viewBox="0 0 24 24">
                <circle className="opacity-25" cx="12" cy="12" r="10"
stroke="currentColor" strokeWidth="4" />
                <path className="opacity-75" fill="currentColor" d="M4 12a8 8

```

```

0 018-8V0C5.373 0 0 5.373 0 12h4z" />
    </svg>
    Submitting...
</span>
) : (
    'Submit Support Request'
)
</button>

<p className="text-center text-sm text-gray-500">
    By submitting, you agree to our{' '}
    <a href="/privacy" className="text-blue-600
    hover:underline">Privacy Policy</a>
</p>
</form>
</div>
);
}

```

Exercise 2.3: OpenAI Agents SDK Implementation (4-5 hours)

Transform your prototype into a production agent using the OpenAI Agents SDK with multi-channel support:

```

# agent/customer_success_agent.py

from openai import OpenAI
from agents import Agent, Runner, function_tool
from pydantic import BaseModel
from typing import Optional
from enum import Enum
import asyncpg
from datetime import datetime

class Channel(str, Enum):
    EMAIL = "email"
    WHATSAPP = "whatsapp"
    WEB_FORM = "web_form"

# Tool definitions with strict typing
class KnowledgeSearchInput(BaseModel):
    query: str
    max_results: int = 5

class TicketInput(BaseModel):
    customer_id: str
    issue: str
    priority: str = "medium"
    category: Optional[str] = None
    channel: Channel

```

```

class EscalationInput(BaseModel):
    ticket_id: str
    reason: str
    urgency: str = "normal"

class ResponseInput(BaseModel):
    ticket_id: str
    message: str
    channel: Channel

# Define tools
@function_tool
async def search_knowledge_base(input: KnowledgeSearchInput) -> str:
    """Search product documentation for relevant information.

    Use this when the customer asks questions about product features,
    how to use something, or needs technical information.
    """
    pool = await get_db_pool()
    async with pool.acquire() as conn:
        # Generate embedding for query
        embedding = await generate_embedding(input.query)

        results = await conn.fetch("""
            SELECT title, content,
                   1 - (embedding <=> $1::vector) as similarity
            FROM knowledge_base
            ORDER BY embedding <=> $1::vector
            LIMIT $2
        """, embedding, input.max_results)

    return format_search_results(results)

@function_tool
async def create_ticket(input: TicketInput) -> str:
    """Create a support ticket for tracking.

    ALWAYS create a ticket at the start of every conversation.
    Include the source channel for proper tracking.
    """
    pool = await get_db_pool()
    async with pool.acquire() as conn:
        ticket_id = await conn.fetchval("""
            INSERT INTO tickets (customer_id, category, priority, status,
            source_channel)
            VALUES ($1, $2, $3, 'open', $4)
            RETURNING id
        """, input.customer_id, input.category, input.priority,

```

```

    input.channel.value)

    return f"Ticket created: {ticket_id}"

@function_tool
async def get_customer_history(customer_id: str) -> str:
    """Get customer's complete interaction history across ALL channels.

    Use this to understand context from previous conversations,
    even if they happened on a different channel.
    """
    pool = await get_db_pool()
    async with pool.acquire() as conn:
        history = await conn.fetch("""
            SELECT c.initial_channel, c.started_at, c.status,
                   m.content, m.role, m.channel, m.created_at
            FROM conversations c
            JOIN messages m ON m.conversation_id = c.id
            WHERE c.customer_id = $1
            ORDER BY m.created_at DESC
            LIMIT 20
        """, customer_id)

    return format_customer_history(history)

@function_tool
async def escalate_to_human(input: EscalationInput) -> str:
    """Escalate conversation to human support.

    Use this when:
    - Customer asks about pricing or refunds
    - Customer sentiment is negative
    - You cannot find relevant information
    - Customer explicitly requests human help
    """
    pool = await get_db_pool()
    async with pool.acquire() as conn:
        await conn.execute("""
            UPDATE tickets
            SET status = 'escalated',
                resolution_notes = $1
            WHERE id = $2
        """, f"Escalation reason: {input.reason}", input.ticket_id)

        # Publish to Kafka for human agents
        await publish_escalation_event(input.ticket_id, input.reason)

    return f"Escalated to human support. Reference: {input.ticket_id}"

@function_tool

```

```

async def send_response(input: ResponseInput) -> str:
    """Send response to customer via their preferred channel.

    The response will be automatically formatted for the channel.
    Email: Formal with greeting/signature
    WhatsApp: Concise and conversational
    Web: Semi-formal
    """

    # Get ticket and channel info
    ticket = await get_ticket(input.ticket_id)

    # Format response for channel
    formatted = await format_for_channel(input.message, input.channel)

    # Send via appropriate channel
    if input.channel == Channel.EMAIL:
        result = await gmail_handler.send_reply(
            to_email=ticket['customer_email'],
            subject=ticket['subject'],
            body=formatted,
            thread_id=ticket.get('thread_id')
        )
    elif input.channel == Channel.WHATSAPP:
        result = await whatsapp_handler.send_message(
            to_phone=ticket['customer_phone'],
            body=formatted
        )
    else: # web_form
        # Store response for API retrieval + send email notification
        result = await store_web_response(input.ticket_id, formatted)
        await send_email_notification(ticket['customer_email'], formatted)

    return f"Response sent via {input.channel.value}:
{result['delivery_status']}"

```

```

# Channel-aware response formatting
async def format_for_channel(response: str, channel: Channel) -> str:
    """Format response appropriately for the channel."""

    if channel == Channel.EMAIL:
        return f"""Dear Customer,

```

Thank you for reaching out to TechCorp Support.

{response}

If you have any further questions, please don't hesitate to reply to this email.

```

Best regards,
TechCorp AI Support Team
---
Ticket Reference: {{ticket_id}}
This response was generated by our AI assistant. For complex issues, you can
request human support.""""

elif channel == Channel.WHATSAPP:
    # Keep it short for WhatsApp
    if len(response) > 300:
        response = response[:297] + "..."
    return f"{response}\n\n💡 Reply for more help or type 'human' for
live support."

else: # web_form
    return f"""{response}

"""

Need more help? Reply to this message or visit our support portal.""""

# Define the Agent with channel awareness
customer_success_agent = Agent(
    name="Customer Success FTE",
    model="gpt-4o",
    instructions="""You are a Customer Success agent for TechCorp SaaS.

## Your Purpose
Handle routine customer support queries with speed, accuracy, and empathy
across multiple channels.

## Channel Awareness
You receive messages from three channels. Adapt your communication style:
- **Email**: Formal, detailed responses. Include proper greeting and
signature.
- **WhatsApp**: Concise, conversational. Keep responses under 300 characters
when possible.
- **Web Form**: Semi-formal, helpful. Balance detail with readability.

## Core Behaviors
1. ALWAYS create a ticket at conversation start (include channel!)
2. Check customer history ACROSS ALL CHANNELS before responding
3. Search knowledge base before answering product questions
4. Be concise on WhatsApp, detailed on email
5. Monitor sentiment - escalate if customer becomes frustrated

## Hard Constraints
- NEVER discuss pricing - escalate immediately
"""

```

- NEVER promise features not in documentation
- NEVER process refunds - escalate to billing
- NEVER share internal processes or systems
- ALWAYS use send_response tool to reply (ensures proper channel formatting)

```

## Escalation Triggers
- Customer mentions "lawyer", "legal", or "sue"
- Customer uses profanity or aggressive language
- You cannot find relevant information after 2 searches
- Customer explicitly requests human help
- WhatsApp customer sends 'human' or 'agent'

## Cross-Channel Continuity
If a customer has contacted us before (any channel), acknowledge it:
"I see you contacted us previously about X. Let me help you further..."
""",  

    tools=[  

        search_knowledge_base,  

        create_ticket,  

        get_customer_history,  

        escalate_to_human,  

        send_response  

    ],  

)

```

Exercise 2.4: Unified Message Processor (3-4 hours)

Build the worker that processes messages from all channels:

```
# workers/message_processor.py

import asyncio
from kafka_client import FTEKafkaConsumer, FTEKafkaProducer, TOPICS
from agent.customer_success_agent import customer_success_agent, Channel
from channels.gmail_handler import GmailHandler
from channels.whatsapp_handler import WhatsAppHandler
from datetime import datetime
import logging

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

class UnifiedMessageProcessor:
    """Process incoming messages from all channels through the FTE agent."""

    def __init__(self):
        self.gmail = GmailHandler()
        self.whatsapp = WhatsAppHandler()
        self.producer = FTEKafkaProducer()
```

```

async def start(self):
    """Start the message processor."""
    await self.producer.start()

    consumer = FTEKafkaConsumer(
        topics=[TOPICS['tickets_incoming']],
        group_id='fte-message-processor'
    )
    await consumer.start()

    logger.info("Message processor started, listening for tickets...")
    await consumer.consume(self.process_message)

async def process_message(self, topic: str, message: dict):
    """Process a single incoming message from any channel."""
    try:
        start_time = datetime.utcnow()

        # Extract channel
        channel = Channel(message['channel'])

        # Get or create customer
        customer_id = await self.resolve_customer(message)

        # Get or create conversation
        conversation_id = await self.get_or_create_conversation(
            customer_id=customer_id,
            channel=channel,
            message=message
        )

        # Store incoming message
        await self.store_message(
            conversation_id=conversation_id,
            channel=channel,
            direction='inbound',
            role='customer',
            content=message['content'],
            channel_message_id=message.get('channel_message_id')
        )

        # Load conversation history
        history = await self.load_conversation_history(conversation_id)

        # Run agent
        result = await customer_success_agent.run(
            messages=history,
            context={
                'customer_id': customer_id,
                'conversation_id': conversation_id,
            }
    )

```

```

        'channel': channel.value,
        'ticket_subject': message.get('subject', 'Support
Request'),
        'metadata': message.get('metadata', {})
    }
)

# Calculate metrics
latency_ms = (datetime.utcnow() - start_time).total_seconds() *
1000

# Store agent response
await self.store_message(
    conversation_id=conversation_id,
    channel=channel,
    direction='outbound',
    role='agent',
    content=result.output,
    latency_ms=latency_ms,
    tool_calls=result.tool_calls
)

# Publish metrics
await self.producer.publish(TOPICS['metrics'], {
    'event_type': 'message_processed',
    'channel': channel.value,
    'latency_ms': latency_ms,
    'escalated': result.escalated,
    'tool_calls_count': len(result.tool_calls)
})

logger.info(f"Processed {channel.value} message in
{latency_ms:.0f}ms")

except Exception as e:
    logger.error(f"Error processing message: {e}")
    await self.handle_error(message, e)

async def resolve_customer(self, message: dict) -> str:
    """Resolve or create customer from message identifiers."""
    pool = await get_db_pool()
    async with pool.acquire() as conn:
        # Try to find by email first
        if email := message.get('customer_email'):
            customer = await conn.fetchrow(
                "SELECT id FROM customers WHERE email = $1", email
            )
            if customer:
                return str(customer['id'])

```

```

# Create new customer
customer_id = await conn.fetchval("""
    INSERT INTO customers (email, name)
    VALUES ($1, $2)
    RETURNING id
""", email, message.get('customer_name', ''))

return str(customer_id)

# Try phone for WhatsApp
if phone := message.get('customer_phone'):
    identifier = await conn.fetchrow("""
        SELECT customer_id FROM customer_identifiers
        WHERE identifier_type = 'whatsapp' AND identifier_value =
$1
""", phone)

    if identifier:
        return str(identifier['customer_id'])

# Create new customer with phone
customer_id = await conn.fetchval("""
    INSERT INTO customers (phone) VALUES ($1) RETURNING id
""", phone)

await conn.execute("""
    INSERT INTO customer_identifiers (customer_id,
identifier_type, identifier_value)
    VALUES ($1, 'whatsapp', $2)
""", customer_id, phone)

return str(customer_id)

raise ValueError("Could not resolve customer from message")

async def get_or_create_conversation(
    self,
    customer_id: str,
    channel: Channel,
    message: dict
) -> str:
    """Get active conversation or create new one."""
    pool = await get_db_pool()
    async with pool.acquire() as conn:
        # Check for active conversation (within last 24 hours)
        active = await conn.fetchrow("""
            SELECT id FROM conversations
            WHERE customer_id = $1
            AND status = 'active'
            AND started_at > NOW() - INTERVAL '24 hours'
        """)

```

```

        ORDER BY started_at DESC
        LIMIT 1
      """", customer_id)

    if active:
        return str(active['id'])

    # Create new conversation
    conversation_id = await conn.fetchval("""
        INSERT INTO conversations (customer_id, initial_channel,
status)
        VALUES ($1, $2, 'active')
        RETURNING id
      """", customer_id, channel.value)

    return str(conversation_id)

async def handle_error(self, message: dict, error: Exception):
    """Handle processing errors gracefully."""
    # Send apologetic response via appropriate channel
    channel = Channel(message['channel'])
    apology = "I'm sorry, I'm having trouble processing your request
right now. A human agent will follow up shortly."

    try:
        if channel == Channel.EMAIL:
            await self.gmail.send_reply(
                to_email=message['customer_email'],
                subject=message.get('subject', 'Support Request'),
                body=apology
            )
        elif channel == Channel.WHATSAPP:
            await self.whatsapp.send_message(
                to_phone=message['customer_phone'],
                body=apology
            )
    except Exception as e:
        logger.error(f"Failed to send error response: {e}")

    # Publish for human review
    await self.producer.publish(TOPICS['escalations'], {
        'event_type': 'processing_error',
        'original_message': message,
        'error': str(error),
        'requires_human': True
    })
}

async def main():
    processor = UnifiedMessageProcessor()

```

```

    await processor.start()

if __name__ == "__main__":
    asyncio.run(main())

```

Exercise 2.5: Kafka Event Streaming (2-3 hours)

Set up Kafka topics for multi-channel event processing:

```

# kafka_client.py

from aiokafka import AIOKafkaProducer, AIOKafkaConsumer
import json
from datetime import datetime
import os

KAFKA_BOOTSTRAP_SERVERS = os.getenv("KAFKA_BOOTSTRAP_SERVERS", "kafka:9092")

# Topic definitions for multi-channel FTE
TOPICS = {
    # Incoming tickets from all channels
    'tickets_incoming': 'fte.tickets.incoming',

    # Channel-specific inbound
    'email_inbound': 'fte.channels.email.inbound',
    'whatsapp_inbound': 'fte.channels.whatsapp.inbound',
    'webform_inbound': 'fte.channels.webform.inbound',

    # Channel-specific outbound
    'email_outbound': 'fte.channels.email.outbound',
    'whatsapp_outbound': 'fte.channels.whatsapp.outbound',
    # Escalations
    'escalations': 'fte.escalations',
    # Metrics and monitoring
    'metrics': 'fte.metrics',
    # Dead Letter queue for failed processing
    'dlq': 'fte.dlq'
}

class FTEKafkaProducer:
    def __init__(self):
        self.producer = None

    async def start(self):
        self.producer = AIOKafkaProducer(
            bootstrap_servers=KAFKA_BOOTSTRAP_SERVERS,

```

```

        value_serializer=lambda v: json.dumps(v).encode('utf-8')
    )
    await self.producer.start()

async def stop(self):
    await self.producer.stop()

async def publish(self, topic: str, event: dict):
    event["timestamp"] = datetime.utcnow().isoformat()
    await self.producer.send_and_wait(topic, event)

class FTEKafkaConsumer:
    def __init__(self, topics: list, group_id: str):
        self.consumer = AIOKafkaConsumer(
            *topics,
            bootstrap_servers=KAFKA_BOOTSTRAP_SERVERS,
            group_id=group_id,
            value_deserializer=lambda v: json.loads(v.decode('utf-8'))
        )

    async def start(self):
        await self.consumer.start()

    async def stop(self):
        await self.consumer.stop()

    async def consume(self, handler):
        async for msg in self.consumer:
            await handler(msg.topic, msg.value)

```

Exercise 2.6: FastAPI Service with Channel Endpoints (3-4 hours)

Build the API layer with endpoints for all channels:

```
# api/main.py
```

```

from fastapi import FastAPI, HTTPException, BackgroundTasks, Request
from fastapi.middleware.cors import CORSMiddleware
from pydantic import BaseModel
from typing import Optional, List
from datetime import datetime
import uuid

from channels.gmail_handler import GmailHandler
from channels.whatsapp_handler import WhatsAppHandler
from channels.web_form_handler import router as web_form_router
from kafka_client import FTEKafkaProducer, TOPICS

app = FastAPI(

```

```

        title="Customer Success FTE API",
        description="24/7 AI-powered customer support across Email, WhatsApp, and
Web",
        version="2.0.0"
)

# CORS for web form
app.add_middleware(
    CORSMiddleware,
    allow_origins=[ "*"], # Configure appropriately for production
    allow_credentials=True,
    allow_methods=[ "*"],
    allow_headers=[ "*"],
)

# Include web form router
app.include_router(web_form_router)

# Initialize handlers
gmail_handler = GmailHandler()
whatsapp_handler = WhatsAppHandler()
kafka_producer = FTEKafkaProducer()

@app.on_event("startup")
async def startup():
    await kafka_producer.start()

@app.on_event("shutdown")
async def shutdown():
    await kafka_producer.stop()

# Health check
@app.get("/health")
async def health_check():
    return {
        "status": "healthy",
        "timestamp": datetime.utcnow().isoformat(),
        "channels": {
            "email": "active",
            "whatsapp": "active",
            "web_form": "active"
        }
    }

# Gmail webhook endpoint
@app.post("/webhooks/gmail")
async def gmail_webhook(request: Request, background_tasks: BackgroundTasks):
    """
Handle Gmail push notifications via Pub/Sub.

```

```

"""
try:
    body = await request.json()
    messages = await gmail_handler.process_notification(body)

    for message in messages:
        # Publish to unified ticket queue
        background_tasks.add_task(
            kafka_producer.publish,
            TOPICS['tickets_incoming'],
            message
        )

    return {"status": "processed", "count": len(messages)}

except Exception as e:
    raise HTTPException(status_code=500, detail=str(e))

# WhatsApp webhook endpoint (Twilio)
@app.post("/webhooks/whatsapp")
async def whatsapp_webhook(request: Request, background_tasks: BackgroundTasks):
    """
    Handle incoming WhatsApp messages via Twilio webhook.
    """

    # Validate Twilio signature
    if not await whatsapp_handler.validate_webhook(request):
        raise HTTPException(status_code=403, detail="Invalid signature")

    form_data = await request.form()
    message = await whatsapp_handler.process_webhook(dict(form_data))

    # Publish to unified ticket queue
    background_tasks.add_task(
        kafka_producer.publish,
        TOPICS['tickets_incoming'],
        message
    )

    # Return TwiML response (empty = no immediate reply, agent will respond)
    return Response(
        content='<?xml version="1.0" encoding="UTF-8"?><Response></Response>',
        media_type="application/xml"
    )

# WhatsApp status callback
@app.post("/webhooks/whatsapp/status")
async def whatsapp_status_webhook(request: Request):
    """Handle WhatsApp message status updates (delivered, read, etc.)."""

```

```

form_data = await request.form()

# Update message delivery status
await update_delivery_status(
    channel_message_id=form_data.get('MessageSid'),
    status=form_data.get('MessageStatus')
)

return {"status": "received"}


# Conversation history endpoint
@app.get("/conversations/{conversation_id}")
async def get_conversation(conversation_id: str):
    """Get full conversation history with cross-channel context."""
    history = await load_conversation_history(conversation_id)
    if not history:
        raise HTTPException(status_code=404, detail="Conversation not found")
    return history


# Customer lookup endpoint
@app.get("/customers/lookup")
async def lookup_customer(email: str = None, phone: str = None):
    """Look up customer by email or phone across all channels."""
    if not email and not phone:
        raise HTTPException(status_code=400, detail="Provide email or phone")

    customer = await find_customer(email=email, phone=phone)
    if not customer:
        raise HTTPException(status_code=404, detail="Customer not found")

    return customer


# Channel metrics endpoint
@app.get("/metrics/channels")
async def get_channel_metrics():
    """Get performance metrics by channel."""
    pool = await get_db_pool()
    async with pool.acquire() as conn:
        metrics = await conn.fetch("""
            SELECT
                initial_channel AS channel,
                COUNT(*) AS total_conversations,
                AVG(sentiment_score) AS avg_sentiment,
                COUNT(*) FILTER (WHERE status = 'escalated') AS escalations
            FROM conversations
            WHERE started_at > NOW() - INTERVAL '24 hours'
            GROUP BY initial_channel
        """)

        return {row['channel']: dict(row) for row in metrics}

```

Exercise 2.7: Kubernetes Deployment (4-5 hours)

Deploy your multi-channel FTE to Kubernetes:

```
# k8s/namespace.yaml
apiVersion: v1
kind: Namespace
metadata:
  name: customer-success-fte
  labels:
    app: customer-success-fte
---

# k8s/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: fte-config
  namespace: customer-success-fte
data:
  ENVIRONMENT: "production"
  LOG_LEVEL: "INFO"
  KAFKA_BOOTSTRAP_SERVERS: "kafka.kafka.svc.cluster.local:9092"
  POSTGRES_HOST: "postgres.customer-success-fte.svc.cluster.local"
  POSTGRES_DB: "fte_db"
  # Channel configs
  GMAIL_ENABLED: "true"
  WHATSAPP_ENABLED: "true"
  WEBFORM_ENABLED: "true"
  # Response Limits
  MAX_EMAIL_LENGTH: "2000"
  MAX_WHATSAPP_LENGTH: "1600"
  MAX_WEBFORM_LENGTH: "1000"

---

# k8s/secrets.yaml
apiVersion: v1
kind: Secret
metadata:
  name: fte-secrets
  namespace: customer-success-fte
type: Opaque
stringData:
  OPENAI_API_KEY: "${OPENAI_API_KEY}"
  POSTGRES_PASSWORD: "${POSTGRES_PASSWORD}"
  # Gmail credentials
  GMAIL_CREDENTIALS: "${GMAIL_CREDENTIALS_JSON}"
  # Twilio credentials
  TWILIO_ACCOUNT_SID: "${TWILIO_ACCOUNT_SID}"
  TWILIO_AUTH_TOKEN: "${TWILIO_AUTH_TOKEN}"
  TWILIO_WHATSAPP_NUMBER: "${TWILIO_WHATSAPP_NUMBER}"
---
```

```

# k8s/deployment-api.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: fte-api
  namespace: customer-success-fte
spec:
  replicas: 3
  selector:
    matchLabels:
      app: customer-success-fte
      component: api
  template:
    metadata:
      labels:
        app: customer-success-fte
        component: api
    spec:
      containers:
        - name: fte-api
          image: your-registry/customer-success-fte:latest
          command: ["uvicorn", "api.main:app", "--host", "0.0.0.0", "--port",
                    "8000"]
          ports:
            - containerPort: 8000
          envFrom:
            - configMapRef:
                name: fte-config
            - secretRef:
                name: fte-secrets
      resources:
        requests:
          memory: "512Mi"
          cpu: "250m"
        limits:
          memory: "1Gi"
          cpu: "500m"
      livenessProbe:
        httpGet:
          path: /health
          port: 8000
        initialDelaySeconds: 10
        periodSeconds: 30
      readinessProbe:
        httpGet:
          path: /health
          port: 8000
        initialDelaySeconds: 5
        periodSeconds: 10
  ...

```

```

# k8s/deployment-worker.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: fte-message-processor
  namespace: customer-success-fte
spec:
  replicas: 3
  selector:
    matchLabels:
      app: customer-success-fte
      component: message-processor
  template:
    metadata:
      labels:
        app: customer-success-fte
        component: message-processor
    spec:
      containers:
        - name: message-processor
          image: your-registry/customer-success-fte:latest
          command: ["python", "workers/message_processor.py"]
          envFrom:
            - configMapRef:
                name: fte-config
            - secretRef:
                name: fte-secrets
      resources:
        requests:
          memory: "512Mi"
          cpu: "250m"
        limits:
          memory: "1Gi"
          cpu: "500m"
      ...
# k8s/service.yaml
apiVersion: v1
kind: Service
metadata:
  name: customer-success-fte
  namespace: customer-success-fte
spec:
  selector:
    app: customer-success-fte
    component: api
  ports:
    - port: 80
      targetPort: 8000
  ...
# k8s/ingress.yaml

```

```

apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: customer-success-fte
  namespace: customer-success-fte
  annotations:
    kubernetes.io/ingress.class: nginx
    cert-manager.io/cluster-issuer: letsencrypt-prod
spec:
  tls:
  - hosts:
    - support-api.yourdomain.com
    secretName: fte-tls
  rules:
  - host: support-api.yourdomain.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: customer-success-fte
            port:
              number: 80
---
# k8s/hpa.yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: fte-api-hpa
  namespace: customer-success-fte
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: fte-api
  minReplicas: 3
  maxReplicas: 20
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 70
---
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: fte-worker-hpa

```

```

namespace: customer-success-fte
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: fte-message-processor
  minReplicas: 3
  maxReplicas: 30
  metrics:
    - type: Resource
      resource:
        name: cpu
      target:
        type: Utilization
        averageUtilization: 70

```

Part 3: Integration & Testing (Hours 41-48)

Exercise 3.1: Multi-Channel E2E Testing (3-4 hours)

```
# tests/test_multichannel_e2e.py
```

```

import pytest
import asyncio
from httpx import AsyncClient
from datetime import datetime

BASE_URL = "http://localhost:8000"

@pytest.fixture
async def client():
    async with AsyncClient(base_url=BASE_URL) as ac:
        yield ac

class TestWebFormChannel:
    """Test the web support form (required build)."""

    @pytest.mark.asyncio
    async def test_form_submission(self, client):
        """Web form submission should create ticket and return ID."""
        response = await client.post("/support/submit", json={
            "name": "Test User",
            "email": "test@example.com",
            "subject": "Help with API",
            "category": "technical",
            "message": "I need help with the API authentication"
        })

        assert response.status_code == 200

```

```

data = response.json()
assert "ticket_id" in data
assert data["message"] is not None

@pytest.mark.asyncio
async def test_form_validation(self, client):
    """Form should validate required fields."""
    response = await client.post("/support/submit", json={
        "name": "A", # Too short
        "email": "invalid-email",
        "subject": "Hi",
        "category": "invalid",
        "message": "Short" # Too short
    })
    assert response.status_code == 422 # Validation error

@pytest.mark.asyncio
async def test_ticket_status_retrieval(self, client):
    """Should be able to check ticket status after submission."""
    # Submit form
    submit_response = await client.post("/support/submit", json={
        "name": "Test User",
        "email": "test@example.com",
        "subject": "Status Test",
        "category": "general",
        "message": "Testing ticket status retrieval"
    })
    ticket_id = submit_response.json()["ticket_id"]

    # Check status
    status_response = await client.get(f"/support/ticket/{ticket_id}")
    assert status_response.status_code == 200
    assert status_response.json()["status"] in ["open", "processing"]

class TestEmailChannel:
    """Test Gmail integration."""

    @pytest.mark.asyncio
    async def test_gmail_webhook_processing(self, client):
        """Gmail webhook should process incoming emails."""
        # Simulate Pub/Sub notification
        response = await client.post("/webhooks/gmail", json={
            "message": {
                "data": "base64_encoded_notification",
                "messageId": "test-123"
            },
            "subscription": "projects/test/subscriptions/gmail-push"
        })

```

```

        })

    assert response.status_code == 200

class TestWhatsAppChannel:
    """Test WhatsApp/Twilio integration."""

    @pytest.mark.asyncio
    async def test_whatsapp_webhook_processing(self, client):
        """WhatsApp webhook should process incoming messages."""
        # Note: Requires valid Twilio signature in production
        response = await client.post(
            "/webhooks/whatsapp",
            data={
                "MessageSid": "SM123",
                "From": "whatsapp:+1234567890",
                "Body": "Hello, I need help",
                "ProfileName": "Test User"
            }
        )

        # Will fail signature validation in test, that's expected
        assert response.status_code in [200, 403]

class TestCrossChannelContinuity:
    """Test that conversations persist across channels."""

    @pytest.mark.asyncio
    async def test_customer_history_across_channels(self, client):
        """Customer history should include all channel interactions."""
        # Create ticket via web form
        web_response = await client.post("/support/submit", json={
            "name": "Cross Channel User",
            "email": "crosschannel@example.com",
            "subject": "Initial Contact",
            "category": "general",
            "message": "First contact via web form"
        })

        ticket_id = web_response.json()["ticket_id"]

        # Look up customer
        customer_response = await client.get(
            "/customers/lookup",
            params={"email": "crosschannel@example.com"}
        )

```

```

    if customer_response.status_code == 200:
        customer = customer_response.json()
        # Should have web form interaction
        assert len(customer.get("conversations", [])) >= 1

class TestChannelMetrics:
    """Test channel-specific metrics."""

    @pytest.mark.asyncio
    async def test_metrics_by_channel(self, client):
        """Should return metrics broken down by channel."""
        response = await client.get("/metrics/channels")

        assert response.status_code == 200
        data = response.json()

        # Should have metrics for each enabled channel
        for channel in ["email", "whatsapp", "web_form"]:
            if channel in data:
                assert "total_conversations" in data[channel]

```

Exercise 3.2: Load Testing (2-3 hours)

tests/load_test.py

```

from locust import HttpUser, task, between
import random

class WebFormUser(HttpUser):
    """Simulate users submitting support forms."""
    wait_time = between(2, 10)
    weight = 3 # Web form is most common

    @task
    def submit_support_form(self):
        categories = ['general', 'technical', 'billing', 'feedback',
        'bug_report']

        self.client.post("/support/submit", json={
            "name": f"Load Test User {random.randint(1, 1000)}",
            "email": f"loadtest{random.randint(1, 1000)}@example.com",
            "subject": f"Load Test Query {random.randint(1, 100)}",
            "category": random.choice(categories),
            "message": "This is a load test message to verify system
performance under stress."
        })

```

```

class HealthCheckUser(HttpUser):

```

```

"""Monitor system health during Load test."""
wait_time = between(5, 15)
weight = 1

@task
def check_health(self):
    self.client.get("/health")

@task
def check_metrics(self):
    self.client.get("/metrics/channels")

```

Deliverables Checklist

Stage 1: Incubation Deliverables

- **Working prototype** handling customer queries from any channel
- **specs/discovery-log.md** - Requirements discovered during exploration
- **specs/customer-success-fte-spec.md** - Crystallized specification
- **MCP server** with 5+ tools (including channel-aware tools)
- **Agent skills manifest** defining capabilities
- **Channel-specific response templates**
- **Test dataset** of 20+ edge cases per channel

Stage 2: Specialization Deliverables

- **PostgreSQL schema** with multi-channel support
- **OpenAI Agents SDK implementation** with channel-aware tools
- **FastAPI service** with all channel endpoints
- **Gmail integration** (webhook handler + send)
- **WhatsApp/Twilio integration** (webhook handler + send)
- **Web Support Form (REQUIRED)** - Complete React component in Next.js
- **Kafka event streaming** with channel-specific topics
- **Kubernetes manifests** for deployment
- **Monitoring configuration**

Stage 3: Integration Deliverables

- **Multi-channel E2E test suite** passing
 - **Load test results** showing 24/7 readiness
 - **Documentation** for deployment and operations
 - **Runbook** for incident response
-

Scoring Rubric

Technical Implementation (50 points)

Criteria	Points	Requirements
Incubation Quality	10	Discovery log shows iterative exploration; multi-channel patterns found
Agent Implementation	10	All tools work; channel-aware responses; proper error handling
Web Support Form	10	Complete React/Next.js form with validation, submission, and status checking
Channel Integrations	10	Gmail + WhatsApp handlers work; proper webhook validation
Database & Kafka	5	Normalized schema; channel tracking; event streaming works
Kubernetes Deployment	5	All manifests work; multi-pod scaling; health checks passing

Operational Excellence (25 points)

Criteria	Points	Requirements
24/7 Readiness	10	Survives pod restarts; handles scaling; no single points of failure
Cross-Channel Continuity	10	Customer identified across channels; history preserved
Monitoring	5	Channel-specific metrics; alerts configured

Business Value (15 points)

Criteria	Points	Requirements
Customer Experience	10	Channel-appropriate responses; proper escalation; sentiment handling
Documentation	5	Clear deployment guide; API documentation; form integration guide

Innovation (10 points)

Criteria	Points	Requirements
Creative Solutions	5	Novel approaches; enhanced UX on web form
Evolution	5	Clear progression from incubation to specialization
Demonstration		

Resources

Required Reading

- [Agent Maturity Model](#) - Core framework
- [OpenAI Agents SDK Documentation](#)
- [Model Context Protocol Specification](#)
- [Gmail API Documentation](#)
- [Twilio WhatsApp API](#)

Recommended Tools

- **Development:** Claude Code, VS Code, Docker Desktop
- **Database/CRM:** PostgreSQL 16 with pgvector extension (this IS your CRM - no external CRM needed)
- **Streaming:** Apache Kafka (use Confluent Cloud for simplicity)
- **Kubernetes:** minikube (local) or any cloud provider
- **Email:** Gmail API with Pub/Sub
- **WhatsApp:** Twilio WhatsApp Sandbox (for development)

What You Don't Need

- External CRM (Salesforce, HubSpot, etc.) - PostgreSQL is your CRM
 - Full website - Only the support form component
 - Production WhatsApp Business account - Twilio Sandbox is sufficient
-

FAQ

Q: Do I need to integrate with Salesforce, HubSpot, or another CRM? A: No. The PostgreSQL database you build IS your CRM system. The schema includes tables for customers, conversations, tickets, and messages - this is all you need. In a real production environment, you might sync this data to enterprise CRMs, but that's outside the scope of this hackathon.

Q: Do I need real Gmail and WhatsApp accounts? A: For development, use Gmail API sandbox and Twilio WhatsApp Sandbox. These provide free testing capabilities without affecting real accounts.

Q: Is the entire website required? A: **No.** Only the Web Support Form component is required. It should be embeddable but doesn't need a surrounding website.

Q: How do I test WhatsApp without Twilio costs? A: Twilio provides a free WhatsApp Sandbox. Join it by sending a WhatsApp message to their sandbox number.

Q: Can I skip a channel? A: The Web Support Form is **required**. Gmail and WhatsApp integrations are expected but partial implementations are acceptable with documented limitations.

Final Challenge: The 24-Hour Multi-Channel Test

After deployment, your FTE must survive a **24-hour continuous operation test** across all channels:

1. **Web Form Traffic:** 100+ submissions over 24 hours
2. **Email Simulation:** 50+ Gmail messages processed
3. **WhatsApp Simulation:** 50+ WhatsApp messages processed
4. **Cross-Channel:** 10+ customers contact via multiple channels
5. **Chaos Testing:** Random pod kills every 2 hours

Metrics Validation: - Uptime > 99.9% - P95 latency < 3 seconds (all channels) - Escalation rate < 25% - Cross-channel customer identification > 95% - No message loss

Teams that pass the 24-hour multi-channel test have built a **true omnichannel Digital FTE**.

Welcome to the future of customer support. Now build it.