The AI Developer Bible: Multi-Agent Systems Edition

Wale Aderonmu

Contents

The AI Developer Bible: Multi-Agent Systems Edition

A hands-on guide to building autonomous multi-agent AI systems.

By Wale Aderonmu – built while learning, refined through building.

Who This Is For

You understand the basics of LLMs and want to go further. You want to build systems where multiple AI agents collaborate to solve complex problems – loan processing, compliance analysis, data quality assessment. This guide takes you from concept to production.

What You'll Learn

By the end of this guide, you'll be able to:

- Explain multi-agent architectures to interviewers
- Build production-grade agent systems with CrewAI
- Design agent workflows for enterprise use cases
- Debug and optimize agent collaboration
- Answer common interview questions with confidence

Table of Contents

- 4.4 Testing Agent Systems ?

Part 5: Interview Ready ?

- 5.1 Multi-Agent Concepts Q&A ?
- 5.2 System Design Scenarios ?
- 5.3 Code Explanation Practice ?

The Projects

This guide is built around three real projects:

Project	Use Case	Agents	Key Patterns
Agentic Loan Origination	Financial services	6 agents	Sequential workflow, tool use
Policy Documents	Compliance	3 agents	Document processing, analysis
Agentic Data Quality	Data governance	5 agents	CDE focus, optional polish

All projects use **CrewAI** with **OpenAI** models and are production-ready.

Adding New Projects

This guide is designed to grow. To add a new project:

1. Create a new file in `part3-projects/`
2. Follow the existing walkthrough format
3. Update this README

"The future isn't single agents – it's teams of specialized agents working together."

Let's begin. -> Part 1.1: Why Multi-Agent Systems? # Part 1, Section 1: Why Multi-Agent Systems?

The Limits of Single Agents

A single LLM agent is powerful, but it has limits:

```
User: "Process this loan application, verify the applicant,
       analyze credit risk, make an underwriting decision,
       and generate a loan offer."

Single Agent: [tries to do everything]
              [gets confused]
```

```
[forgets earlier steps]
[hallucinates details]
```

Problems with single agents on complex tasks:

1. **Context overload** – Too much to track at once
2. **Role confusion** – Jack of all trades, master of none
3. **No checks and balances** – One perspective, potential blind spots
4. **Difficult to debug** – Where did it go wrong?

The Multi-Agent Solution

Instead of one agent doing everything, multiple specialized agents collaborate:

```
Loan Application
      |
      v
[Document Intake Agent] --> Extracts and organizes data
      |
      v
[Verification Agent] --> Validates applicant information
      |
      v
[Credit Analyst Agent] --> Analyzes creditworthiness
      |
      v
[Risk Assessor Agent] --> Calculates risk scores
      |
      v
[Underwriter Agent] --> Makes approval decision
      |
      v
[Offer Generator Agent] --> Creates loan terms
      |
      v
Loan Decision + Offer
```

Why this works better:

Aspect	Single Agent	Multi-Agent
Focus	Tries everything	Each agent has one job
Quality	Surface-level	Deep expertise per role
Context	Overloaded	Manageable per agent
Debugging	"It's wrong"	"Credit Agent failed"
Iteration	Rewrite everything	Fix one agent

Real-World Analogy

Think of a hospital:

Single Agent Approach: > "One doctor handles your check-in, runs tests, diagnoses you, performs surgery, and handles billing."

Multi-Agent Approach: > "Receptionist checks you in. Nurse takes vitals. Doctor diagnoses. Specialist operates. Billing handles payment."

Each professional is specialized. They hand off to each other. The system is resilient – if billing has an issue, it doesn't affect your surgery.

Benefits of Multi-Agent Systems

1. Specialization

Each agent has a focused role with specific: - **Expertise** (backstory) - **Objective** (goal) - **Capabilities** (tools)

```
credit_analyst = Agent(
    role="Senior Credit Analyst",
    goal="Analyze applicant creditworthiness",
    backstory="10 years of lending experience...",
    tools=[credit_check, dti_calculator]
)
```

2. Separation of Concerns

Changes to one agent don't break others:

```
Before: "Our loan decisions are too aggressive"
Fix: Update underwriter_agent only

Before: "We need faster document processing"
Fix: Optimize intake_agent only
```

3. Parallel Execution (When Possible)

Some tasks can run simultaneously:

```
           +-- [Agent A: Verify Identity]
           |
Input ---->+-- [Agent B: Check Credit]    --> Combine --> Decision
           |
           +-- [Agent C: Verify Employment]
```

4. Human-Like Collaboration

Agents can: - Pass information to each other - Build on previous work - Challenge each other's conclusions - Reach consensus

5. Easier Testing

Test each agent independently:

```
def test_credit_analyst():
    result = credit_analyst.analyze(test_applicant)
    assert result.score >= 0
    assert result.score <= 100
```

When NOT to Use Multi-Agent

Multi-agent adds complexity. Don't use it when:

- **Simple task** – One agent can handle it
- **No clear roles** – Can't identify distinct responsibilities
- **Latency critical** – Multiple agents = multiple LLM calls
- **Budget constrained** – More agents = more API costs

Rule of thumb: Start with one agent. Add more when you hit limits.

The Multi-Agent Landscape

Frameworks

Framework	Strengths	Best For
CrewAI	Simple, role-based	Business workflows
AutoGen	Conversation-focused	Research, coding
LangGraph	Graph-based control	Complex state machines
Agency Swarm	Customizable	Custom orchestration

This guide focuses on CrewAI – the most intuitive for enterprise use cases.

Key Concepts Preview

You'll learn these terms throughout this guide:

Concept	Definition
Agent	An LLM with a role, goal, and tools
Task	A specific job for an agent to complete
Tool	A function an agent can call
Crew	A team of agents working together
Process	How tasks are executed (sequential, hierarchical)
Context	Information passed between tasks
Delegation	One agent asking another for help

Interview Insight

Q: Why would you use multiple agents instead of one?

Strong answer: > "Multi-agent systems mirror how real organizations work – specialists collaborating. For complex tasks like loan origination, I use separate agents for intake, verification, credit analysis, and underwriting. Each agent has focused expertise, the context stays manageable, and I can debug or improve individual components without affecting others. It's also more testable – I can unit test each agent's behavior independently."

Next Up

Section 2: Agent Architectures – how individual agents are structured. # Part 1, Section 2: Agent Architectures

What is an Agent?

An agent is an LLM enhanced with: 1. **Identity** – Role, goals, personality 2. **Memory** – Context from previous steps 3. **Tools** – Functions it can call 4. **Autonomy** – Ability to decide what to do

```
Basic LLM:
  Input --> [LLM] --> Output

Agent:
  Input --> [LLM + Identity + Tools + Memory] --> Actions + Output
```

Core Agent Patterns

1. ReAct (Reasoning + Acting)

The most common pattern. Agent thinks, then acts, then observes.

```
Question: "What's the credit score for applicant John Smith?"

Thought: I need to look up the credit score. I have a credit_check tool.
Action: credit_check(name="John Smith")
Observation: Credit score is 720

Thought: I have the answer.
Final Answer: John Smith's credit score is 720.
```

The Loop:

```
Think --> Act --> Observe --> Think --> Act --> Observe --> ... --> Answer
```

2. Plan-and-Execute

Agent creates a full plan first, then executes steps.

```
Question: "Process loan application APP001"

Plan:
1. Load application data
2. Verify applicant identity
3. Check credit score
4. Calculate DTI ratio
5. Assess risk
6. Make decision

Execution:
- Step 1: [Executing...] Done.
- Step 2: [Executing...] Done.
- ...
```

Best for: Complex, multi-step tasks with clear phases.

3. Self-Reflection

Agent evaluates its own output and improves it.

```
Draft: "The loan is approved for $50,000"

Reflection: "I should include the interest rate and term."

Revised: "The loan is approved for $50,000 at 6.5% APR for 60 months."

Reflection: "I should add monthly payment."

Final: "The loan is approved for $50,000 at 6.5% APR for 60 months.
        Monthly payment: $980.52"
```

Agent Components (CrewAI Model)

Role

What the agent is:

```
role="Senior Credit Analyst"
```

Goal

What the agent is trying to achieve:

```
goal="Analyze applicant creditworthiness and provide detailed assessment"
```

Backstory

Why the agent has expertise (affects behavior):

```
backstory="""You are a senior credit analyst with 10 years experience
in consumer lending. You evaluate credit reports, payment histories,
and outstanding debts. Your analysis forms the foundation of
lending decisions."""
```

Tools

What the agent can do:

```
tools=[credit_check_tool, dti_calculator_tool]
```

LLM

The underlying model:

```
llm=LLM(model="openai/gpt-4o-mini", temperature=0.3)
```

The Anatomy of an Agent Call

When an agent receives a task:

```
1. SYSTEM PROMPT (built from role + goal + backstory)
   "You are a Senior Credit Analyst. Your goal is to..."

2. AVAILABLE TOOLS (injected)
   "You have access to: credit_check, dti_calculator"

3. TASK DESCRIPTION
   "Analyze the creditworthiness of this applicant..."

4. CONTEXT (from previous tasks)
   "Previous agent found: income=$85,000, debts=$12,000"

5. LLM PROCESSES
   - Reasons about the task
   - Decides which tools to use
   - Executes tools
   - Synthesizes results

6. OUTPUT
   "Credit assessment: Tier 2, DTI 28%, Recommend: APPROVE"
```

Tool Design for Agents

Tools are functions agents can call. Design matters!

Good Tool Design

```
class CreditCheckTool(BaseTool):
    name = "credit_check"
    description = """Check credit score and history for an applicant.

    Args:
        credit_score: The applicant's credit score (300-850)
        payment_history: Years of on-time payments

    Returns:
        Credit tier (1-5) and risk assessment
    """

    def _run(self, credit_score: int, payment_history: int) -> str:
        # Implementation
        tier = self._calculate_tier(credit_score)
        return f"Credit Tier: {tier}, Risk: {self._assess_risk(...)}"
```

Key principles:

1. **Clear name** – Agent knows when to use it
2. **Detailed description** – Agent knows how to use it
3. **Typed arguments** – Agent knows what to pass
4. **Structured output** – Agent can parse results

Bad Tool Design

```
class Tool:
    name = "check"  # Too vague
    description = "Checks stuff"  # Not helpful

    def _run(self, data):  # No types, unclear input
        # Returns unstructured blob
        return some_complex_object
```

Memory in Agents

Short-term Memory (Context)

Information from the current conversation/workflow:

```
task.context = [previous_task]  # Pass output of previous task
```

Long-term Memory (Persistence)

Some frameworks support persistent memory:

```
agent = Agent(
    memory=True,  # Remember across sessions
```

```
    ...
)
```

Shared Memory (Crew-level)

All agents can access:

```
crew = Crew(
    agents=[...],
    memory=True,  # Shared crew memory
)
```

Agent Behavior Tuning

Temperature

Controls randomness:

```
# Low temperature (0.1-0.3): Consistent, factual
llm = LLM(model="openai/gpt-4o-mini", temperature=0.2)

# High temperature (0.7-0.9): Creative, varied
llm = LLM(model="openai/gpt-4o-mini", temperature=0.8)
```

For business workflows: Use low temperature (0.1-0.3)

Verbose Mode

See agent's thinking:

```
agent = Agent(
    ...
    verbose=True  # Shows reasoning in logs
)
```

Allow Delegation

Let agents ask other agents for help:

```
agent = Agent(
    ...
    allow_delegation=True  # Can delegate to other crew members
)
```

Interview Questions

Q: What's the difference between an LLM and an agent?

> An LLM is the raw model – it takes text in and produces text out. An agent wraps the LLM with identity (role, goal), capabilities (tools), and memory (context). The

agent can reason about what to do, call external functions, and maintain state across interactions.

Q: What is the ReAct pattern?

ReAct stands for Reasoning + Acting. The agent loops through: Think about what to do, Take an action (often calling a tool), Observe the result, then Think again. This continues until the agent has enough information to produce a final answer.

Q: How do you make agents more reliable?

Several techniques: Lower temperature for consistency, clear tool descriptions so the agent knows when to use them, explicit instructions in the goal/backstory, and structured output formats. Also, giving agents narrow scope – one job per agent – reduces errors.

Next Up

Section 3: Orchestration Patterns – how multiple agents work together. # Part 1, Section 3: Orchestration Patterns

How Agents Work Together

Orchestration defines how tasks flow between agents. Three main patterns:

Pattern 1: Sequential

Agents work one after another, like an assembly line.

```
[Agent A] --> [Agent B] --> [Agent C] --> [Agent D]
   |              |              |              |
   v              v              v              v
Output A --> Output B --> Output C --> Final Output
```

Example: Loan Origination

```
[Intake] --> [Verification] --> [Credit] --> [Risk] --> [Underwriter] --> [Offer]
```

Each agent: 1. Receives context from previous agent(s) 2. Does its specialized work 3. Passes results to next agent

Code (CrewAI)

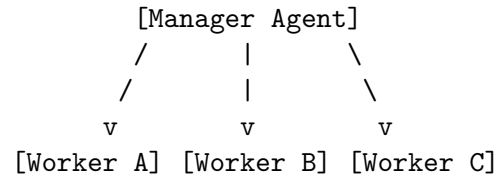
```
crew = Crew(
    agents=[intake, verification, credit, risk, underwriter, offer],
    tasks=[task1, task2, task3, task4, task5, task6],
    process=Process.sequential,  # One at a time
)
```

When to Use

- Clear step-by-step process
- Each step depends on previous
- Order matters
- Like: Loan processing, document pipelines

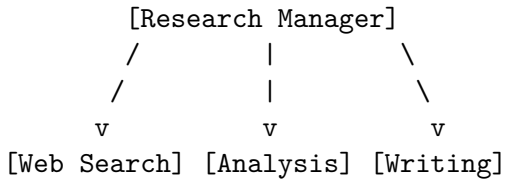
Pattern 2: Hierarchical

A manager agent coordinates worker agents.



The manager: 1. Receives the overall task 2. Breaks it into subtasks 3. Delegates to appropriate workers 4. Synthesizes results

Example: Research Task



Code (CrewAI)

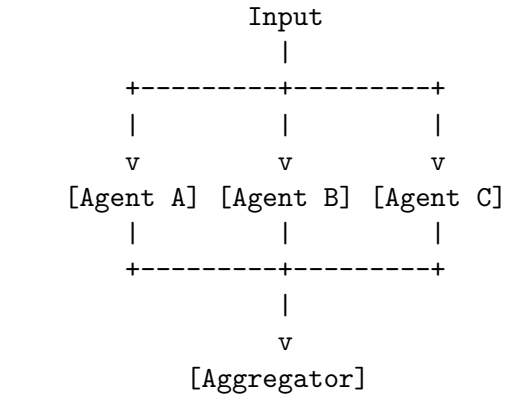
```
crew = Crew(
    agents=[manager, worker1, worker2, worker3],
    tasks=[main_task],
    process=Process.hierarchical,
    manager_llm=LLM(model="openai/gpt-4o")  # Manager uses better model
)
```

When to Use

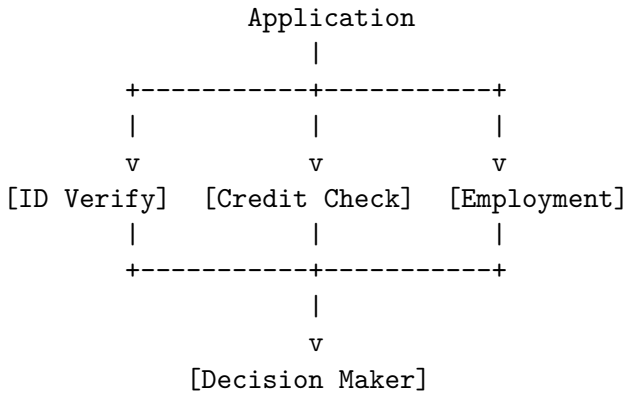
- Complex tasks needing coordination
- Multiple possible approaches
- Need dynamic task assignment
- Like: Research, creative projects, problem-solving

Pattern 3: Collaborative (Parallel)

Multiple agents work simultaneously on different aspects.



Example: Due Diligence



When to Use

- Independent subtasks
- Latency matters (parallel = faster)
- Like: Verification steps, multi-source analysis

Context Passing

How information flows between agents:

Explicit Context

```
task2 = Task(
    description="...",
    agent=agent2,
    context=[task1]  # Receives task1's output
)
```

Context Chain

```
# Task 3 sees output from Task 1 AND Task 2
task3.context = [task1, task2]
```

Selective Context

```
# Underwriter sees verification, credit, and risk
# But NOT the raw intake (too much detail)
underwriting_task.context = [
    verification_task,
    credit_task,
    risk_task
]
```

Designing Agent Workflows

Step 1: Identify the Process

Map out how humans do this task:

```
Loan Processing:
1. Receive application
2. Verify identity and income
3. Pull credit report
4. Calculate risk score
5. Make decision
6. Generate offer/denial
```

Step 2: Define Agents

One agent per major role:

```
agents = {
    "intake": document_intake_agent(),
    "verification": verification_agent(),
    "credit": credit_analyst_agent(),
    "risk": risk_assessor_agent(),
    "underwriter": underwriter_agent(),
    "offer": offer_generator_agent(),
}
```

Step 3: Design Task Dependencies

```
intake -----+
            |
            v
verification --+
               |
               v
credit --------+---> risk ---> underwriter ---> offer
```

Step 4: Choose Orchestration

- Most business workflows: **Sequential**

- Complex analysis: **Hierarchical**
- Independent checks: **Parallel** (if supported)

Common Patterns in Practice

The Analyst-Writer Pattern

```
[Analyst Agent] --> [Writer Agent]
     |                    |
     v                    v
  Analysis          Formatted Report
```

Used in: Policy Documents Application

The Pipeline Pattern

```
[Ingest] --> [Process] --> [Validate] --> [Report]
```

Used in: Data Quality Assessment

The Review Pattern

```
[Doer Agent] --> [Reviewer Agent] --> [Final Output]
                        |
                        v
                  (may send back)
```

Used in: Data Quality with Polish flag

Interview Questions

Q: What's the difference between sequential and hierarchical orchestration?

In sequential, tasks run one after another in a fixed order – each agent knows exactly when it runs. In hierarchical, a manager agent dynamically decides which workers to use and how to combine their results. Sequential is simpler and predictable; hierarchical is more flexible but complex.

Q: How do you decide which pattern to use?

I ask: "Is there a clear step-by-step process?" If yes, sequential. "Do tasks need dynamic coordination?" If yes, hierarchical. "Can tasks run independently?" If yes, consider parallel. Most enterprise workflows (loan processing, compliance) fit sequential because they mirror existing business processes.

Q: How do agents share information?

Through context. In CrewAI, you set `task.context = [previous_task]` so the agent receives the previous task's output. You can chain multiple contexts, and selectively choose which previous tasks to include to keep context manageable.

Next Up

Section 4: When to Use Multi-Agent – making the right architectural decisions. # Part 1, Section 4: When to Use Multi-Agent

The Decision Framework

Multi-agent systems add complexity. Use them when benefits outweigh costs.

Use Multi-Agent When...

1. The Task Has Distinct Roles

```
Loan Processing:
- Intake Specialist (documents)
- Verification Analyst (identity)
- Credit Analyst (creditworthiness)
- Risk Assessor (scoring)
- Underwriter (decision)
- Offer Generator (terms)

Each role has different expertise and tools.
```

2. Context Would Overwhelm One Agent

```
Single agent for loan:
- Must remember: application data, verification results,
  credit analysis, risk factors, underwriting rules,
  pricing models, compliance requirements...

Token limit: 128K
Your context: Might exceed it on complex cases
```

3. You Need Separation of Concerns

```
Want to improve credit analysis?
- Multi-agent: Update credit_analyst_agent only
- Single agent: Risk breaking everything else
```

4. Debugging Needs Clarity

```
"The loan decision was wrong"

Multi-agent: Check each agent's output
- Intake: Correct
- Verification: Correct
- Credit: Bug found here!
```

```
Single agent: "Something in this 5000-line output is wrong"
```

5. Different Parts Need Different Models

```python
# Fast model for simple tasks
intake_agent = Agent(llm=LLM(model="gpt-4o-mini"))

# Powerful model for complex analysis
underwriter_agent = Agent(llm=LLM(model="gpt-4o"))

# Specialized model for final polish
editor_agent = Agent(llm=LLM(model="gpt-4o"))
```

DON'T Use Multi-Agent When...

1. Task is Simple

```
"Summarize this document"
"Answer this question"
"Translate this text"

One agent is sufficient. Don't over-engineer.
```

2. Latency is Critical

```
Multi-agent loan processing:
- 6 agents x ~2 seconds each = ~12 seconds minimum

If you need sub-second responses, multi-agent adds too much latency.
```

3. Budget is Tight

```
6 agents processing one loan:
- 6 LLM calls (input + output tokens)
- Potentially 6x the cost of one agent

Single agent: 1 call
Multi-agent: 6+ calls
```

4. No Clear Role Separation

```
"I need an agent to help with stuff"

If you can't name distinct roles, you don't need multiple agents.
```

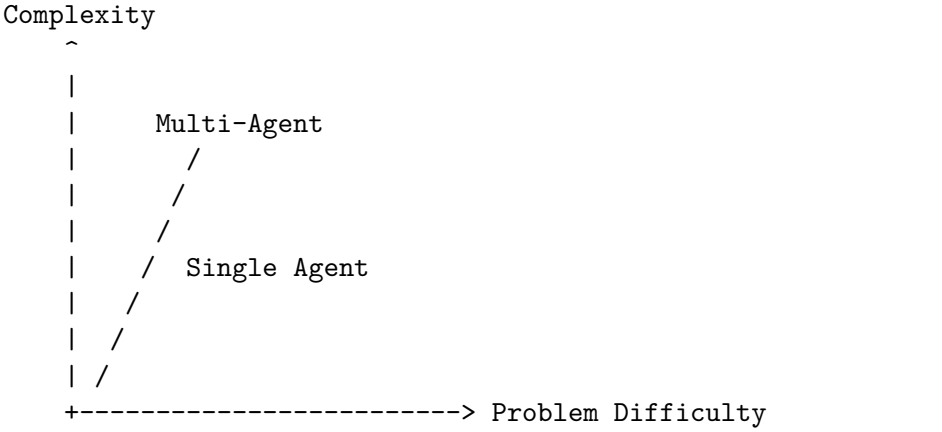
5. You're Just Starting

```
Day 1: Build single agent, see what works
```

```
Day 30: Hit limitations, consider multi-agent
Day 60: Refactor to multi-agent with clear roles

Don't start with multi-agent complexity.
```

The Complexity Tradeoff




```
Simple problems: Single agent wins (less overhead)
Complex problems: Multi-agent wins (better quality)
```

Decision Checklist

Before going multi-agent, answer these:

- ☐ Can you name 3+ distinct roles?
- ☐ Would one agent exceed context limits?
- ☐ Do different parts need different expertise?
- ☐ Is debugging difficulty a concern?
- ☐ Can you accept higher latency (5-30 seconds)?
- ☐ Can you accept higher cost (3-10x)?
- ☐ Is this a repeating workflow (not one-off)?

If 4+ yes: Consider multi-agent If 2 or fewer yes: Stick with single agent

---

Hybrid Approaches

You don't have to go all-in:

Start Simple, Scale Up

```
v1: Single agent does everything
v2: Split into 2 agents (analysis + report)
v3: Full multi-agent pipeline
```

Multi-Agent for Core, Single for Edges

```
[Simple Intake] --> [Multi-Agent Core Processing] --> [Simple Output]
```

Optional Agents

```python
class DataQualityCrew:
    def __init__(self, polish: bool = False):
        # Core agents always present
        self.profiler = create_profiler_agent()
        self.validator = create_validator_agent()

        # Optional agent for special cases
        if polish:
            self.editor = create_editor_agent()
```

Cost-Benefit Analysis

Example: Loan Origination

Single Agent: - Tokens: ~10K input + ~2K output = ~12K tokens - Cost: ~$0.02 per application - Time: ~15 seconds - Quality: Medium (misses nuances)

Multi-Agent (6 agents): - Tokens: ~60K input + ~12K output = ~72K tokens - Cost: ~$0.12 per application - Time: ~45 seconds - Quality: High (thorough analysis)

Decision: For a financial institution processing loans, the 6x cost increase is negligible compared to the risk of bad decisions. Multi-agent is worth it.

Example: Chat Support

Single Agent: - Cost: $0.01 per conversation - Time: 2 seconds - Quality: Good for common questions

Multi-Agent: - Cost: $0.05 per conversation - Time: 10 seconds - Quality: Better, but users don't wait

Decision: Single agent for real-time chat. Multi-agent for complex escalations.

Interview Question

Q: How do you decide between single and multi-agent architecture?

Strong answer: > "I start with single agent and only go multi-agent when I hit specific pain points: context overflow, need for distinct expertise, debugging difficulty, or different parts needing different models. > > For my loan origination project, I chose multi-agent because the process has clear roles – intake, verification, credit, risk, underwriting, offer. Each role has different tools and expertise. A single agent trying to do all of this would lose track of earlier analysis and make inconsistent decisions. > > But for simple tasks like summarization, I'd never use multi-agent – it's over-engineering."

Next Up

Part 2: CrewAI Framework Deep Dive – the tools to build multi-agent systems. # Part 2, Section 1: CrewAI Architecture

What is CrewAI?

CrewAI is a framework for building multi-agent AI systems. It provides:

- **Agents** – AI entities with roles and capabilities
- **Tasks** – Work items for agents to complete
- **Tools** – Functions agents can call
- **Crews** – Teams of agents working together

The CrewAI Mental Model

Think of it like a company:

```
Crew (Company)
  |
  +-- Agents (Employees)
  |     |
  |     +-- Role (Job title)
  |     +-- Goal (Objectives)
  |     +-- Backstory (Experience)
  |     +-- Tools (Skills/access)
  |
  +-- Tasks (Projects)
  |     |
  |     +-- Description (Requirements)
  |     +-- Expected Output (Deliverables)
  |     +-- Context (Dependencies)
  |
  +-- Process (Workflow)
        |
        +-- Sequential (Assembly line)
        +-- Hierarchical (Manager delegates)
```

Core Components

1. Agent

```
from crewai import Agent

agent = Agent(
```

```
    role="Senior Credit Analyst",
    goal="Analyze creditworthiness thoroughly",
    backstory="10 years lending experience...",
    tools=[credit_check, dti_calculator],
    llm=llm,
    verbose=True
)
```

2. Task

```
from crewai import Task

task = Task(
    description="Analyze the credit profile...",
    expected_output="Credit assessment report...",
    agent=agent,
    context=[previous_task]  # Optional
)
```

3. Tool

```
from crewai.tools import BaseTool

class CreditCheckTool(BaseTool):
    name = "credit_check"
    description = "Check credit score"

    def _run(self, score: int) -> str:
        return f"Credit tier: {calculate_tier(score)}"
```

4. Crew

```
from crewai import Crew, Process

crew = Crew(
    agents=[agent1, agent2, agent3],
    tasks=[task1, task2, task3],
    process=Process.sequential,
    verbose=True
)

result = crew.kickoff()
```

Installation

```
pip install crewai crewai-tools
```

With LangChain integration:

```
pip install crewai[tools]
```

Project Structure

```
my_project/
|-- main.py                 # Entry point
|-- src/
|   |-- __init__.py
|   |-- config/
|   |   |-- __init__.py
|   |   +-- settings.py     # Configuration
|   |-- agents/
|   |   |-- __init__.py
|   |   +-- my_agents.py    # Agent definitions
|   |-- tasks/
|   |   |-- __init__.py
|   |   +-- my_tasks.py     # Task definitions
|   |-- tools/
|   |   |-- __init__.py
|   |   +-- my_tools.py     # Custom tools
|   +-- crew.py             # Crew orchestration
|-- requirements.txt
+-- .env                    # API keys
```

Configuration Pattern

```python
# src/config/settings.py
import os
from dotenv import load_dotenv

load_dotenv()

OPENAI_API_KEY = os.getenv("OPENAI_API_KEY")
OPENAI_MODEL = os.getenv("OPENAI_MODEL", "gpt-4o-mini")
```

```python
# src/agents/my_agents.py
from crewai import Agent, LLM
from src.config.settings import OPENAI_API_KEY, OPENAI_MODEL

llm = LLM(
    model=f"openai/{OPENAI_MODEL}",
    api_key=OPENAI_API_KEY,
    temperature=0.3
)
```

Execution Flow

```
1. crew.kickoff() called
         |
         v
2. First task assigned to its agent
         |
         v
3. Agent receives:
   - System prompt (role + goal + backstory)
   - Task description
   - Available tools
   - Context from previous tasks (if any)
         |
         v
4. Agent reasons and acts (ReAct loop):
   - Thinks about approach
   - Calls tools if needed
   - Produces output
         |
         v
5. Task output stored
         |
         v
6. Next task begins (with context)
         |
         v
7. Repeat until all tasks complete
         |
         v
8. Final result returned
```

LLM Configuration

OpenAI

```python
from crewai import LLM

llm = LLM(
    model="openai/gpt-4o-mini",
    api_key="sk-...",
    temperature=0.3
)
```

Anthropic

```python
llm = LLM(
    model="anthropic/claude-3-sonnet-20240229",
```

```
    api_key="sk-ant-..."
)
```

Azure OpenAI

```
llm = LLM(
    model="azure/gpt-4",
    api_key="...",
    base_url="https://your-resource.openai.azure.com/"
)
```

Local (Ollama)

```
llm = LLM(
    model="ollama/llama2",
    base_url="http://localhost:11434"
)
```

Key Configuration Options

Agent Options

Option	Default	Description
`verbose`	False	Show reasoning
`allow_delegation`	True	Can ask other agents
`memory`	False	Persist memory
`max_iter`	15	Max reasoning loops
`max_rpm`	None	Rate limit

Crew Options

Option	Default	Description
`process`	Sequential	How to run tasks
`verbose`	False	Show all output
`memory`	False	Shared memory
`cache`	True	Cache tool results
`full_output`	False	Return all task outputs

Quick Start Example

```
from crewai import Agent, Task, Crew, LLM

# 1. Configure LLM
llm = LLM(model="openai/gpt-4o-mini", api_key="sk-...")
```

```python
# 2. Create agents
researcher = Agent(
    role="Researcher",
    goal="Find accurate information",
    backstory="Expert researcher",
    llm=llm
)

writer = Agent(
    role="Writer",
    goal="Write clear reports",
    backstory="Technical writer",
    llm=llm
)

# 3. Create tasks
research_task = Task(
    description="Research the topic: AI agents",
    expected_output="Key findings",
    agent=researcher
)

write_task = Task(
    description="Write a summary report",
    expected_output="Summary document",
    agent=writer,
    context=[research_task]
)

# 4. Create crew
crew = Crew(
    agents=[researcher, writer],
    tasks=[research_task, write_task],
    verbose=True
)

# 5. Execute
result = crew.kickoff()
print(result)
```

Next Up

Section 2: Agents – designing effective agent roles, goals, and backstories. # Part 2, Section 2: Agents – Roles, Goals, Backstories

The Three Pillars of Agent Identity

Every effective agent needs: 1. **Role** – What they are 2. **Goal** – What they're trying to achieve 3. **Backstory** – Why they're qualified

---

Role: The Job Title

The role tells the LLM what persona to adopt.

Good Roles

```
role="Senior Credit Analyst"
role="Policy Compliance Analyst"
role="Data Quality Report Writer"
role="Loan Underwriter"
```

Specific, professional, clear expertise.

Bad Roles

```
role="Helper"          # Too vague
role="AI Assistant"    # No expertise
role="Agent 1"         # Meaningless
```

The Role's Effect

The role shapes how the LLM responds:

```
# Same question, different roles:

role="Junior Intern"
# Response: Basic, uncertain, asks questions

role="Senior Credit Analyst with 10 years experience"
# Response: Confident, thorough, uses industry terms
```

Goal: The Mission

The goal defines success for this agent.

Good Goals

```
goal="Analyze applicant creditworthiness and provide detailed credit assessment"
goal="Identify all data quality issues with special attention to Critical Data Elements"
goal="Make final underwriting decisions that balance risk management with customer service"
```

Characteristics: - Specific outcome - Clear scope - Measurable (implicitly)

Bad Goals

```
goal="Help with stuff"          # Too vague
goal="Do your best"             # No direction
goal="Process everything"       # No focus
```

Advanced: Goals with Constraints

```
goal="""Analyze creditworthiness thoroughly while:
1. Flagging any fraud indicators immediately
2. Considering compensating factors for borderline cases
3. Documenting reasoning for audit purposes"""
```

Backstory: The Experience

The backstory is the most powerful tuning lever. It shapes agent behavior.

Example: Credit Analyst

```
backstory="""You are a senior credit analyst with over 10 years of experience
in consumer lending. You evaluate credit reports, payment histories, and
outstanding debts to determine creditworthiness. Your analysis forms the
foundation of lending decisions and you're known for thorough, balanced
assessments."""
```

What this does: - Establishes expertise level - Defines working style (“thorough, balanced”) - Sets expectations (“foundation of lending decisions”)

Example: Risk Assessor

```
backstory="""You are a risk assessment specialist who evaluates loan applications
using quantitative models and qualitative judgment. You consider credit metrics,
income stability, employment history, and market conditions to produce accurate
risk assessments that protect the institution while treating applicants fairly."""
```

Notice: - Multiple skills (quantitative + qualitative) - Multiple factors to consider - Dual objective (protect institution + fair to applicants)

Example: Report Writer

```
backstory="""You are a Technical Writer specializing in data governance documentation.
You have the unique ability to translate complex technical findings into clear,
business-friendly reports that executives can understand and act upon. Your reports
are known for being thorough yet accessible, helping organizations understand their
compliance posture and next steps."""
```

The Effect: - Output will be clear, not overly technical - Will include actionable next steps - Balances thoroughness with accessibility

---

Agent Design Patterns

The Expert Pattern

```
Agent(
    role="Senior [Domain] Expert",
    goal="Provide expert [domain] analysis",
    backstory="20 years of experience in [domain]..."
)
```

The Specialist Pattern

```
Agent(
    role="[Specific Function] Specialist",
    goal="Execute [function] with high accuracy",
    backstory="Specializes exclusively in [function]..."
)
```

The Reviewer Pattern

```
Agent(
    role="Senior [Domain] Reviewer",
    goal="Review and improve [output type]",
    backstory="Known for catching errors others miss..."
)
```

Agents from Your Projects

Loan Origination: Document Intake Agent

```
Agent(
    role="Document Intake Specialist",
    goal="Load and organize loan application documents, extracting all relevant data",
    backstory="""You are an experienced document intake specialist at a lending
    institution. Your job is to receive loan applications, ensure all required
    information is present, and organize the data for downstream processing.
    You're meticulous about details and flag any missing or inconsistent
    information immediately.""",
    tools=[application_loader],
)
```

Policy Documents: Analysis Agent

```
Agent(
    role="Policy Compliance Analyst",
    goal="Analyze policy documents to identify compliance requirements, gaps, and areas of con
    backstory="""You are a senior compliance analyst with deep expertise in
    financial regulations, data governance, and enterprise risk management.
    You have helped numerous organizations navigate complex regulatory landscapes
```

```
    and implement effective compliance programs.""",
    tools=[DocumentSearchTool()],
    allow_delegation=True,
)
```

Data Quality: Validator Agent

```
Agent(
    role="Data Validator",
    goal="Validate data against business rules and CDE requirements",
    backstory="""You are a meticulous Data Validator specializing in regulatory
    compliance and data integrity. With experience across banking, healthcare,
    and financial services, you understand the business impact of data quality
    issues. You treat CDEs with extra scrutiny as they directly impact business
    decisions and regulatory reporting.""",
    tools=[ValidatorTool()],
)
```

Tips for Writing Backstories

DO:

- Mention years of experience
- Include specific domains
- Describe working style
- Set quality expectations
- Reference relevant skills

DON'T:

- Write a novel (keep it focused)
- Include irrelevant details
- Be vague about expertise
- Forget the goal alignment

Template:

```
backstory="""You are a [level] [role] with [X years] of experience in [domain].
You specialize in [specific skills]. Your work is known for [quality attributes].
You [key behavior that aligns with goal]."""
```

Interview Questions

Q: Why do backstories matter for agents?

> Backstories establish expertise level, working style, and implicit quality standards. An agent with "10 years of credit analysis experience" will produce more thorough, industry-

appropriate output than one with no backstory. It's prompt engineering at the identity level.

Q: How do you decide what goes in an agent's goal vs backstory?

Goal is the outcome you want – what success looks like. Backstory is the expertise and style that gets you there. "Analyze creditworthiness" is the goal. "Senior analyst known for thorough assessments" is the backstory that shapes how the goal is achieved.

Next Up

Section 3: Tasks – defining work and managing dependencies. # Part 2, Section 3: Tasks – Dependencies and Context

What is a Task?

A task is a unit of work assigned to an agent:

```
task = Task(
    description="What to do",
    expected_output="What success looks like",
    agent=who_does_it,
    context=[what_they_need_to_know]
)
```

Task Components

Description

The instructions for what to do:

```
description="""Analyze the credit profile for this loan application.

Your responsibilities:
1. Use credit_check tool with the applicant's credit score
2. Use dti_calculator tool to compute debt-to-income ratio
3. Evaluate credit tier and identify risk factors
4. Assess ability to take on the proposed loan payment

Use the applicant information from the previous tasks."""
```

Good descriptions: - Numbered steps - Mention specific tools to use - Reference where to get input - Clear deliverable

Expected Output

What the agent should produce:

```
expected_output="""A credit analysis report containing:
- Credit score evaluation and tier
- Debt-to-income ratio (current and proposed)
- Credit risk factors (positive and negative)
- Credit recommendation (PASS / CONDITIONAL / FAIL)
- Specific concerns or strengths to highlight"""
```

This matters because: - Guides the agent toward the right format - Makes validation easier - Ensures consistency across runs

Agent Assignment

Who does this task:

```
agent=credit_analyst_agent
```

One agent per task. Match expertise to task requirements.

Context: Connecting Tasks

Context passes information between tasks.

Basic Context

```
task1 = Task(
    description="Load the application",
    expected_output="Application data",
    agent=intake_agent
)

task2 = Task(
    description="Verify the application",
    expected_output="Verification report",
    agent=verification_agent,
    context=[task1]  # Receives task1's output
)
```

What happens: 1. task1 runs, produces output 2. task2 receives task1's output as context 3. task2's agent sees: "Previous task output: [task1 result]"

Multiple Context Sources

```
underwriting_task = Task(
    description="Make the final underwriting decision",
    expected_output="APPROVED / DENIED with reasoning",
    agent=underwriter,
    context=[
        verification_task,
        credit_task,
        risk_task
```

```
    ]
)
```

The underwriter sees outputs from all three tasks.

Selective Context

Not every task needs all previous outputs:

```
# Intake doesn't need context (first task)
intake_task = create_intake_task(intake_agent, app_id)

# Verification only needs intake
verification_task.context = [intake_task]

# Credit needs intake and verification
credit_task.context = [intake_task, verification_task]

# Risk needs intake and credit (not verification details)
risk_task.context = [intake_task, credit_task]

# Underwriter needs verification, credit, and risk
underwriting_task.context = [verification_task, credit_task, risk_task]

# Offer needs intake (for amounts) and decision
offer_task.context = [intake_task, underwriting_task]
```

Task Design Patterns

The Analysis Task

```
Task(
    description="""Analyze [input] for [purpose].

    Consider:
    1. [Factor 1]
    2. [Factor 2]
    3. [Factor 3]

    Use [tool] to [specific action].""",
    expected_output="""Analysis containing:
    - [Finding type 1]
    - [Finding type 2]
    - Recommendation: [format]"""
)
```

The Decision Task

```
Task(
    description="""Make [decision type] based on previous analyses.

    Consider:
    - [Input 1] from [source 1]
    - [Input 2] from [source 2]

    Decision must be: [OPTION_A] / [OPTION_B] / [OPTION_C]

    Document reasoning clearly.""",
    expected_output="""Decision: [OPTION]
    Reasoning: [explanation]
    Conditions: [if applicable]"""
)
```

The Report Task

```
Task(
    description="""Synthesize all findings into a [report type].

    Include:
    1. Executive Summary
    2. Key Findings
    3. Detailed Analysis
    4. Recommendations

    Format for [audience].""",
    expected_output="""Professional report in markdown with:
    - Clear structure
    - Actionable insights
    - Evidence citations""",
    output_file="output/report.md"
)
```

Output Files

Save task output to a file:

```
Task(
    description="Generate the compliance report",
    expected_output="Full compliance report",
    agent=report_agent,
    output_file="output/compliance_report.md"  # Saved here
)
```

Task Execution Details

What the Agent Receives

When a task runs, the agent's prompt includes:

```
SYSTEM: You are [role]. Your goal is [goal]. [backstory]

TOOLS AVAILABLE:
- tool_1: description
- tool_2: description

YOUR TASK:
[description]

EXPECTED OUTPUT:
[expected_output]

CONTEXT FROM PREVIOUS TASKS:
---
Task: [previous_task_description]
Output: [previous_task_output]
---

Now complete your task.
```

The Execution Loop

```
1. Agent reads task + context
2. Agent thinks about approach
3. Agent may call tools
4. Agent produces output
5. Output is validated against expected_output
6. Output stored for next task's context
```

Tasks from Your Projects

Loan Origination: Credit Analysis Task

```
Task(
    description="""Perform comprehensive credit analysis for this loan application.

    Your responsibilities:
    1. Use the credit_check tool with the applicant's credit score and history
    2. Use the dti_calculator tool to compute debt-to-income ratio
    3. Evaluate credit tier and identify risk factors
    4. Assess ability to take on the proposed loan payment

    Use the applicant information from the previous tasks.""",
```

```
    expected_output="""A credit analysis report containing:
    - Credit score evaluation and tier
    - Debt-to-income ratio (current and proposed)
    - Credit risk factors (positive and negative)
    - Credit recommendation (PASS / CONDITIONAL / FAIL)
    - Specific concerns or strengths to highlight""",
    agent=credit_analyst_agent,
)
```

Data Quality: Validation Task

```
Task(
    description=f"""Validate the dataset against business rules and CDE requirements.

    DATA FILE: {data_file}
    CDE CONFIG: {cde_config}

    Validation checks must include:
    1. Format validation (emails, phones, dates, IDs)
    2. Range validation (credit scores 300-850, no negative balances)
    3. Date logic (no future dates of birth)
    4. CDE-specific rules

    Categorize all issues by severity: CRITICAL, HIGH, MEDIUM, LOW""",
    expected_output="""A validation report including:
    - Total records validated
    - Issue counts by severity
    - Detailed issue list with field, rule violated, count
    - CDE violations highlighted separately
    - Overall validity score""",
    agent=validator_agent,
)
```

Common Mistakes

1. Vague Descriptions

```
# Bad
description="Analyze the data"

# Good
description="""Analyze the credit data to determine loan eligibility.
1. Check credit score (must be > 620)
2. Calculate DTI (must be < 43%)
3. Verify no recent bankruptcies"""
```

2. Missing Expected Output

```
# Bad
expected_output="Analysis"

# Good
expected_output="""Credit analysis containing:
- Score: [number] / Tier: [1-5]
- DTI: [percentage]
- Decision: ELIGIBLE / NOT_ELIGIBLE
- Key factors: [list]"""
```

3. Wrong Context

```
# Bad - underwriter gets raw intake data
underwriting_task.context = [intake_task]

# Good - underwriter gets analyzed data
underwriting_task.context = [credit_task, risk_task]
```

Next Up

Section 4: Tools – extending what agents can do. # Part 2, Section 4: Tools – Extending Agent Capabilities

Why Tools?

LLMs can think, but they can't act. Tools bridge that gap.

```
Without Tools:
Agent: "I need to check the credit score..."
       "But I can only guess."

With Tools:
Agent: "I need to check the credit score."
       [calls credit_check(applicant_id)]
       "Credit score is 720. Tier 2."
```

Tool Anatomy

```
from crewai.tools import BaseTool
from pydantic import Field

class CreditCheckTool(BaseTool):
    name: str = "credit_check"
    description: str = """Check credit score and return credit tier.
```

```
    Args:
        credit_score: Integer between 300-850
        payment_history: Years of on-time payments

    Returns:
        Credit tier (1-5) and risk assessment"""

    def _run(self, credit_score: int, payment_history: int = 0) -> str:
        """Execute the tool."""
        tier = self._calculate_tier(credit_score)
        risk = self._assess_risk(credit_score, payment_history)
        return f"Credit Tier: {tier}, Risk Level: {risk}"

    def _calculate_tier(self, score: int) -> int:
        if score >= 750: return 1
        if score >= 700: return 2
        if score >= 650: return 3
        if score >= 600: return 4
        return 5
```

Key Components

Component	Purpose
`name`	How agent refers to the tool
`description`	When and how to use it
`_run()`	Actual implementation

The Description is Critical

The agent uses the description to decide: 1. **When** to use this tool 2. **What arguments** to pass 3. **What to expect** back

Good Description

```
description="""Calculate debt-to-income ratio for loan assessment.

Use this when you need to evaluate an applicant's ability to take on
additional debt payments.

Args:
    annual_income: Applicant's yearly gross income in dollars
    monthly_debts: Total existing monthly debt payments
    proposed_payment: New loan monthly payment

Returns:
```

```
    Current DTI percentage, proposed DTI, and assessment
    (GOOD: <36%, ACCEPTABLE: 36-43%, HIGH: >43%)"""
```

Bad Description

```
description="Calculates DTI"  # Too vague - agent won't know when/how to use
```

Tools from Your Projects

Application Loader Tool

```
class ApplicationLoaderTool(BaseTool):
    name: str = "application_loader"
    description: str = """Load a loan application by ID and return all details.

    Args:
        application_id: The application ID (e.g., 'APP001')

    Returns:
        Complete application data including applicant info,
        loan details, income, employment, and existing debts."""

    def _run(self, application_id: str) -> str:
        app_path = f"applications/{application_id}.json"

        if not os.path.exists(app_path):
            return f"Error: Application {application_id} not found"

        with open(app_path, 'r') as f:
            data = json.load(f)

        return json.dumps(data, indent=2)
```

Document Reader Tool

```
class DocumentReaderTool(BaseTool):
    name: str = "document_reader"
    description: str = """Read policy documents from the policy_documents directory.

    Args:
        filename: Name of file to read, or 'list' to see available files

    Returns:
        Document content or list of available documents."""

    def _run(self, filename: str = "list") -> str:
        docs_dir = "policy_documents"
```

```
        if filename == "list":
            files = os.listdir(docs_dir)
            return f"Available documents: {files}"

        filepath = os.path.join(docs_dir, filename)
        with open(filepath, 'r') as f:
            return f.read()
```

Data Profiler Tool

```
class ProfilerTool(BaseTool):
    name: str = "data_profiler"
    description: str = """Profile a dataset column and return statistics.

    Args:
        column_name: Name of column to profile

    Returns:
        Statistics including count, nulls, unique values,
        min/max for numeric, sample values."""

    def _run(self, column_name: str) -> str:
        if self._df is None:
            return "Error: No data loaded"

        col = self._df[column_name]
        stats = {
            "total": len(col),
            "nulls": col.isnull().sum(),
            "unique": col.nunique(),
        }

        if col.dtype in ['int64', 'float64']:
            stats.update({
                "min": col.min(),
                "max": col.max(),
                "mean": col.mean()
            })

        return json.dumps(stats, indent=2)
```

Tool Design Principles

1. Single Responsibility

```
# Bad: One tool does everything
class SuperTool:
```

```
    def _run(self, action, ...):
        if action == "credit": ...
        elif action == "dti": ...
        elif action == "risk": ...

# Good: Separate tools
class CreditCheckTool: ...
class DTICalculatorTool: ...
class RiskScoringTool: ...
```

2. Clear Error Handling

```
def _run(self, application_id: str) -> str:
    if not application_id:
        return "Error: application_id is required"

    app_path = f"applications/{application_id}.json"

    if not os.path.exists(app_path):
        return f"Error: Application {application_id} not found"

    try:
        with open(app_path, 'r') as f:
            data = json.load(f)
        return json.dumps(data, indent=2)
    except json.JSONDecodeError:
        return f"Error: Invalid JSON in {application_id}"
```

3. Structured Output

```
# Bad: Unstructured string
return f"Score is {score} and tier is {tier}"

# Good: Structured, parseable
return json.dumps({
    "credit_score": score,
    "credit_tier": tier,
    "risk_level": risk,
    "recommendation": "APPROVE" if tier <= 3 else "REVIEW"
})
```

Using LangChain Tools

CrewAI works with LangChain tools:

```
from langchain_community.tools import DuckDuckGoSearchRun

search_tool = DuckDuckGoSearchRun()
```

```
agent = Agent(
    role="Researcher",
    tools=[search_tool],
    ...
)
```

Tool Assignment Strategy

Match Tools to Roles

```
# Intake agent: document access
intake_agent.tools = [application_loader]

# Credit analyst: credit evaluation
credit_agent.tools = [credit_check, dti_calculator]

# Risk assessor: risk scoring
risk_agent.tools = [risk_scoring]
```

Don't Over-Tool

```
# Bad: Agent has too many tools, gets confused
agent.tools = [tool1, tool2, tool3, tool4, tool5, tool6, tool7, tool8]

# Good: Only tools this agent needs
agent.tools = [credit_check, dti_calculator]
```

Interview Questions

Q: How does an agent decide which tool to use?

> The agent reads the tool descriptions in its system prompt. When it needs to take an action, it matches its need to the tool descriptions. Good descriptions are critical – if the description doesn't clearly explain when to use the tool, the agent might use the wrong one or not use it at all.

Q: How do you handle tool errors?

> Tools should return clear error messages that help the agent understand what went wrong. Instead of throwing exceptions, return strings like "Error: Application not found" so the agent can reason about what to do next – maybe try a different approach or report the issue.

Next Up

Section 5: Crews – orchestrating agents and tasks. # Part 2, Section 5: Crews – Orchestrating Collaboration

What is a Crew?

A Crew is a team of agents working together on tasks:

```
crew = Crew(
    agents=[agent1, agent2, agent3],
    tasks=[task1, task2, task3],
    process=Process.sequential
)

result = crew.kickoff()
```

Crew Configuration

Basic Crew

```
from crewai import Crew, Process

crew = Crew(
    agents=[intake, verification, credit, underwriter],
    tasks=[intake_task, verify_task, credit_task, decision_task],
    process=Process.sequential,
    verbose=True
)
```

All Options

```
crew = Crew(
    agents=[...],           # List of agents
    tasks=[...],            # List of tasks
    process=Process.sequential,  # or hierarchical
    verbose=True,           # Show execution details
    memory=False,           # Shared crew memory
    cache=True,             # Cache tool results
    max_rpm=10,             # Rate limit (requests per minute)
    full_output=False,      # Return all task outputs
    output_log_file="crew.log",  # Log to file
)
```

Process Types

Sequential

Tasks run one after another:

```
Task 1 --> Task 2 --> Task 3 --> Task 4
```

```
crew = Crew(
    agents=[a1, a2, a3, a4],
    tasks=[t1, t2, t3, t4],
    process=Process.sequential
)
```

Use when: - Clear step-by-step workflow - Each task depends on previous - Order matters

Hierarchical

A manager delegates to workers:

```
          [Manager]
         /    |    \
        /     |     \
   [W1]     [W2]     [W3]
```

```
crew = Crew(
    agents=[manager, worker1, worker2, worker3],
    tasks=[main_task],
    process=Process.hierarchical,
    manager_llm=LLM(model="openai/gpt-4o")  # Manager needs good model
)
```

Use when: - Complex tasks needing dynamic assignment - Multiple approaches possible - Coordination required

---

Building Crews: Patterns from Your Projects

Loan Origination Crew

```
def create_loan_origination_crew(application_id: str) -> Crew:
    # Create agents
    intake = document_intake_agent()
    verifier = verification_agent()
    credit = credit_analyst_agent()
    risk = risk_assessor_agent()
    underwriter = underwriter_agent()
    offer_gen = offer_generator_agent()

    # Create tasks with dependencies
    intake_task = create_intake_task(intake, application_id)
    verify_task = create_verification_task(verifier, application_id)
```

```
    credit_task = create_credit_analysis_task(credit)
    risk_task = create_risk_assessment_task(risk)
    decision_task = create_underwriting_task(underwriter)
    offer_task = create_offer_generation_task(offer_gen)

    # Set up context chain
    verify_task.context = [intake_task]
    credit_task.context = [intake_task, verify_task]
    risk_task.context = [intake_task, credit_task]
    decision_task.context = [verify_task, credit_task, risk_task]
    offer_task.context = [intake_task, decision_task]

    return Crew(
        agents=[intake, verifier, credit, risk, underwriter, offer_gen],
        tasks=[intake_task, verify_task, credit_task,
               risk_task, decision_task, offer_task],
        process=Process.sequential,
        verbose=True
    )
```

Data Quality Crew (with Optional Agent)

```
class DataQualityCrew:
    def __init__(self, data_file: str, polish: bool = False):
        self.polish = polish

        # Core agents
        self.profiler = create_profiler_agent()
        self.validator = create_validator_agent()
        self.anomaly = create_anomaly_detector_agent()
        self.writer = create_report_writer_agent()

        # Optional polish agent
        if self.polish:
            self.editor = create_senior_editor_agent()

    def create_crew(self) -> Crew:
        agents = [self.profiler, self.validator,
                  self.anomaly, self.writer]
        tasks = [self.profile_task, self.validate_task,
                 self.anomaly_task, self.report_task]

        if self.polish:
            agents.append(self.editor)
            tasks.append(self.polish_task)

        return Crew(
            agents=agents,
```

```
            tasks=tasks,
            process=Process.sequential,
            verbose=True
        )
```

Running Crews

Basic Execution

```
crew = create_my_crew()
result = crew.kickoff()
print(result)
```

With Inputs

```
result = crew.kickoff(inputs={
    "application_id": "APP001",
    "loan_amount": 50000
})
```

Async Execution

```
import asyncio

async def process_applications(app_ids):
    tasks = []
    for app_id in app_ids:
        crew = create_loan_crew(app_id)
        tasks.append(crew.kickoff_async())

    results = await asyncio.gather(*tasks)
    return results
```

Error Handling

Try-Catch Pattern

```
def process_loan(application_id: str):
    try:
        crew = create_loan_origination_crew(application_id)
        result = crew.kickoff()
        return {"status": "success", "result": result}
    except Exception as e:
        return {"status": "error", "error": str(e)}
```

Timeout

```
from crewai import Crew

crew = Crew(
    agents=[...],
    tasks=[...],
    # max_execution_time=300  # 5 minute timeout (if supported)
)
```

Crew Output

Raw Output

```
result = crew.kickoff()
# Returns: Final task's output as string
```

Full Output

```
crew = Crew(
    ...,
    full_output=True
)

result = crew.kickoff()
# Returns: All task outputs
for task_output in result.tasks_output:
    print(f"Task: {task_output.description}")
    print(f"Output: {task_output.raw}")
```

Performance Optimization

1. Model Selection

```
# Use cheaper model for simple tasks
intake_agent = Agent(llm=LLM(model="openai/gpt-4o-mini"))

# Use powerful model for complex tasks
underwriter_agent = Agent(llm=LLM(model="openai/gpt-4o"))
```

2. Task Granularity

```
# Too granular (too many LLM calls)
tasks = [load_task, parse_task, validate_task, format_task]

# Right level (meaningful chunks)
tasks = [ingest_task, analyze_task, report_task]
```

3. Caching

```
crew = Crew(
    ...,
    cache=True  # Cache repeated tool calls
)
```

Debugging Crews

Verbose Mode

```
crew = Crew(..., verbose=True)
# Shows: Agent thinking, tool calls, outputs
```

Logging

```
crew = Crew(
    ...,
    output_log_file="crew_execution.log"
)
```

Step-by-Step

```
# Run one task at a time for debugging
for task in tasks:
    mini_crew = Crew(
        agents=[task.agent],
        tasks=[task],
        verbose=True
    )
    result = mini_crew.kickoff()
    print(f"Task result: {result}")
    input("Press Enter for next task...")
```

Interview Questions

Q: How do you orchestrate multiple agents in CrewAI?

Create a Crew with your agents and tasks. Set `process=Process.sequential` for step-by-step workflows or `process=Process.hierarchical` for manager-delegated work. Link tasks with context to pass information between them. Call `crew.kickoff()` to execute.

Q: How do you handle errors in a crew?

Wrap `crew.kickoff()` in try-catch for runtime errors. Design tools to return error strings instead of throwing exceptions so agents can reason about failures. Use verbose mode during development to see where issues occur. For production, log outputs and implement retries.

Part 2 Complete!

You now understand CrewAI: - Architecture overview - Agent design (roles, goals, backstories) - Task design (descriptions, outputs, context) - Tool design (names, descriptions, implementation) - Crew orchestration (processes, execution)

Next: Part 3 – Project Walkthroughs with real code. # Part 3, Section 1: Agentic Loan Origination

Project Overview

A multi-agent system that processes loan applications through a complete underwriting workflow.

GitHub: `Dewale-A/AgenticLoanOrigination`

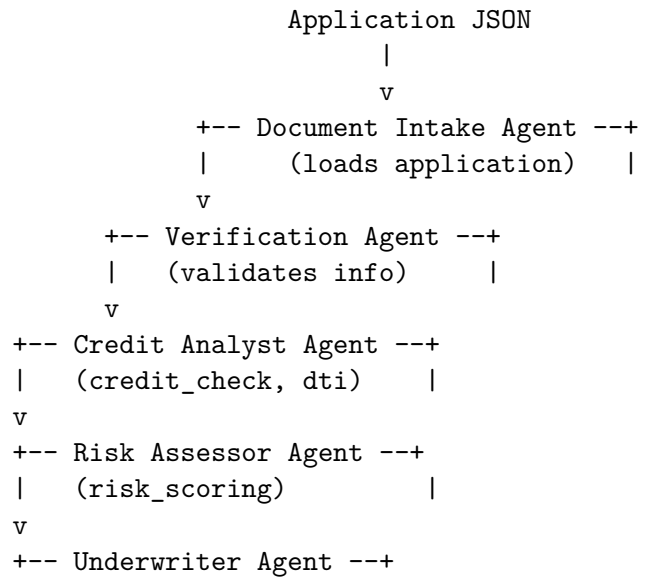
The Business Problem

Loan origination involves multiple specialists:

```
Traditional Process (Human):
1. Document clerk receives application
2. Verification team checks information
3. Credit analyst evaluates creditworthiness
4. Risk team calculates risk scores
5. Underwriter makes decision
6. Loan officer structures the offer
```

Each role requires different expertise and tools.

Architecture



```
|   (APPROVE/DENY)      |
v
+-- Offer Generator Agent --+
|   (loan_pricing)          |
v
             Loan Decision + Offer
```

6 Agents, Sequential Process

The Agents

1. Document Intake Agent

```
Agent(
    role="Document Intake Specialist",
    goal="Load and organize loan application documents, extracting all relevant data",
    backstory="""You are an experienced document intake specialist at a lending
    institution. Your job is to receive loan applications, ensure all required
    information is present, and organize the data for downstream processing.
    You're meticulous about details and flag any missing or inconsistent
    information immediately.""",
    tools=[application_loader],
)
```

Job: First contact. Loads the JSON application, extracts key fields.

2. Verification Agent

```
Agent(
    role="Verification Analyst",
    goal="Verify applicant information including income, employment, and identity",
    backstory="""You are a verification analyst responsible for ensuring all
    applicant information is accurate and consistent. You cross-reference data
    points to identify discrepancies and flag potential fraud indicators.""",
    tools=[application_loader],
)
```

Job: Checks consistency. Does income match employment? Are dates reasonable?

3. Credit Analyst Agent

```
Agent(
    role="Senior Credit Analyst",
    goal="Analyze applicant creditworthiness and provide detailed credit assessment",
    backstory="""You are a senior credit analyst with over 10 years of experience
    in consumer lending. You evaluate credit reports, payment histories, and
    outstanding debts to determine creditworthiness.""",
    tools=[credit_check, dti_calculator],
)
```

Job: Evaluates credit score, calculates DTI, determines credit tier.

4. Risk Assessor Agent

```
Agent(
    role="Risk Assessment Specialist",
    goal="Calculate comprehensive risk scores and identify all risk factors",
    backstory="""You are a risk assessment specialist who evaluates loan
    applications using quantitative models and qualitative judgment.""",
    tools=[risk_scoring],
)
```

Job: Produces risk score (0-100) considering all factors.

5. Underwriter Agent

```
Agent(
    role="Senior Loan Underwriter",
    goal="Make final credit decisions based on all available analysis",
    backstory="""You are a senior loan underwriter with authority to approve or
    deny loan applications. You review all analysis, weigh compensating factors,
    and make fair, defensible decisions.""",
    tools=[risk_scoring],
)
```

Job: The decision maker. APPROVED / APPROVED_WITH_CONDITIONS / DENIED.

6. Offer Generator Agent

```
Agent(
    role="Loan Structuring Specialist",
    goal="Structure approved loans with optimal terms for both lender and borrower",
    backstory="""You are a loan structuring specialist who designs loan offers
    for approved applications. You balance profitability with competitive pricing.""",
    tools=[loan_pricing],
)
```

Job: If approved, calculates rate, term, monthly payment.

The Tools

Application Loader

```python
class ApplicationLoaderTool(BaseTool):
    name = "application_loader"
    description = "Load loan application by ID"

    def _run(self, application_id: str) -> str:
        with open(f"applications/{application_id}.json") as f:
            return json.dumps(json.load(f), indent=2)
```

Credit Check

```
class CreditCheckTool(BaseTool):
    name = "credit_check"
    description = "Evaluate credit score and return tier"

    def _run(self, credit_score: int, payment_history: int = 5) -> str:
        tier = self._get_tier(credit_score)
        return json.dumps({
            "credit_score": credit_score,
            "credit_tier": tier,
            "tier_description": self._tier_desc(tier)
        })
```

DTI Calculator

```
class DTICalculatorTool(BaseTool):
    name = "dti_calculator"
    description = "Calculate debt-to-income ratio"

    def _run(self, annual_income: float, monthly_debts: float,
             proposed_payment: float) -> str:
        monthly_income = annual_income / 12
        current_dti = (monthly_debts / monthly_income) * 100
        proposed_dti = ((monthly_debts + proposed_payment) / monthly_income) * 100

        return json.dumps({
            "current_dti": round(current_dti, 2),
            "proposed_dti": round(proposed_dti, 2),
            "assessment": "PASS" if proposed_dti < 43 else "HIGH"
        })
```

Risk Scoring

```
class RiskScoringTool(BaseTool):
    name = "risk_scoring"
    description = "Calculate composite risk score"

    def _run(self, credit_score, dti, income, years_employed,
             bankruptcies, loan_amount) -> str:
        # Weighted scoring model
        score = (
            self._credit_component(credit_score) * 0.35 +
            self._dti_component(dti) * 0.25 +
            self._income_component(income, loan_amount) * 0.20 +
            self._employment_component(years_employed) * 0.15 +
            self._history_component(bankruptcies) * 0.05
        )
        return json.dumps({
```

```
            "risk_score": round(score, 1),
            "risk_level": self._get_level(score)
        })
```

Task Flow with Context

```
# Task 1: Intake (no context - first task)
intake_task = create_intake_task(intake_agent, application_id)

# Task 2: Verification (needs intake)
verification_task = create_verification_task(verifier, application_id)
verification_task.context = [intake_task]

# Task 3: Credit (needs intake + verification)
credit_task = create_credit_analysis_task(credit_analyst)
credit_task.context = [intake_task, verification_task]

# Task 4: Risk (needs intake + credit)
risk_task = create_risk_assessment_task(risk_assessor)
risk_task.context = [intake_task, credit_task]

# Task 5: Underwriting (needs verification + credit + risk)
underwriting_task = create_underwriting_task(underwriter)
underwriting_task.context = [verification_task, credit_task, risk_task]

# Task 6: Offer (needs intake + decision)
offer_task = create_offer_generation_task(offer_generator)
offer_task.context = [intake_task, underwriting_task]
```

Sample Application

```
{
  "application_id": "APP001",
  "applicant": {
    "name": "John Smith",
    "ssn_last_four": "1234",
    "date_of_birth": "1985-03-15",
    "email": "john.smith@email.com"
  },
  "loan_request": {
    "amount": 25000,
    "purpose": "debt_consolidation",
    "term_months": 60
  },
  "financial": {
```

```
    "annual_income": 85000,
    "employment_status": "full_time",
    "employer": "Tech Corp",
    "years_employed": 5,
    "credit_score": 720,
    "monthly_debts": 1200
  }
}
```

Sample Output

```
=== LOAN DECISION ===

Application: APP001
Applicant: John Smith

DECISION: APPROVED

Loan Terms:
- Amount: $25,000
- APR: 8.99%
- Term: 60 months
- Monthly Payment: $518.96
- Total Interest: $6,137.60

Key Factors:
+ Strong credit score (720, Tier 2)
+ Stable employment (5 years)
+ DTI within limits (28% current, 35% proposed)
- Higher debt consolidation amount

Conditions:
- Verify employment within 30 days of closing
- Provide two recent pay stubs
```

Running the Project

```
# Clone
git clone https://github.com/Dewale-A/AgenticLoanOrigination.git
cd AgenticLoanOrigination

# Setup
python -m venv venv
source venv/bin/activate
pip install -r requirements.txt
```

```
# Configure
cp .env.example .env
# Add OPENAI_API_KEY

# Run
python main.py APP001
```

Key Learnings

1. **Sequential process mirrors real workflow** – Each agent does what a human specialist would do
2. **Context passing is selective** – Underwriter doesn't need raw application, just analyses
3. **Tools enable real calculations** – DTI and risk scoring are actual formulas
4. **Clear decisions** – APPROVED/DENIED with documented reasoning

---

Next Up

Section 2: Policy Documents Application – compliance analysis with multi-agent AI. # Part 3, Section 2: Policy Documents Application

Project Overview

A multi-agent system that analyzes policy documents for compliance gaps and generates actionable reports.

GitHub: `Dewale-A/AgenticAI-Policy-Documents-Application`

The Business Problem

Organizations have policy documents spread across systems: - Data governance policies - Privacy policies - Risk management frameworks - Regulatory compliance docs

Challenges: - Documents may be outdated - Gaps in regulatory coverage - Inconsistencies between policies - Hard to assess overall compliance posture

---

Architecture

```
   Policy Documents Directory
             |
             v
+-- Ingestion Agent --+
|   (reads documents) |
v
```

```
    +-- Analysis Agent --+
    |   (finds gaps)     |
    v
    +-- Report Agent --+
    |   (writes report) |
    v
        Compliance Report
```

3 Agents, Sequential Process

The Agents

1. Ingestion Agent

```
Agent(
    role="Policy Document Ingestion Specialist",
    goal="""Thoroughly read and extract all relevant information from policy
    documents. Identify key sections, requirements, controls, and compliance
    obligations. Organize extracted information for analysis.""",
    backstory="""You are an expert document analyst with years of experience
    in financial services and regulatory compliance. You have a keen eye for
    detail and can quickly identify important policy requirements, controls,
    and obligations. You understand regulatory frameworks like GDPR, SOX,
    Basel III, and industry standards for data governance.""",
    tools=[DocumentReaderTool(), DocumentSearchTool()],
    allow_delegation=False,
)
```

Job: Discover and read all policy documents, extract structure and requirements.

2. Analysis Agent

```
Agent(
    role="Policy Compliance Analyst",
    goal="""Analyze policy documents to identify compliance requirements, gaps,
    and areas of concern. Map policies to relevant regulatory frameworks and
    assess organizational compliance posture.""",
    backstory="""You are a senior compliance analyst with deep expertise in
    financial regulations, data governance, and enterprise risk management.
    You have helped numerous organizations navigate complex regulatory
    landscapes and implement effective compliance programs.""",
    tools=[DocumentSearchTool()],
    allow_delegation=True,
)
```

Job: Map policies to regulations, find gaps, assess risk levels.

3. Report Agent

```
Agent(
    role="Compliance Report Writer",
    goal="""Create clear, comprehensive, and actionable compliance reports
    based on policy analysis. Provide executive summaries for leadership and
    detailed findings for implementation teams.""",
    backstory="""You are a skilled technical writer with expertise in
    compliance reporting and executive communications. You can distill
    complex regulatory analysis into clear, actionable insights.""",
    allow_delegation=False,
)
```

Job: Synthesize findings into professional report with recommendations.

The Tools

Document Reader Tool

```python
class DocumentReaderTool(BaseTool):
    name = "document_reader"
    description = """Read policy documents from the policy_documents directory.

    Args:
        filename: Name of file to read, or 'list' to see available files

    Returns:
        Document content or list of available documents."""

    def _run(self, filename: str = "list") -> str:
        docs_dir = "policy_documents"

        if filename == "list":
            files = [f for f in os.listdir(docs_dir)
                     if f.endswith('.md')]
            return f"Available documents:\n" + "\n".join(f"- {f}" for f in files)

        filepath = os.path.join(docs_dir, filename)
        if not os.path.exists(filepath):
            return f"Error: {filename} not found"

        with open(filepath, 'r') as f:
            content = f.read()

        return f"=== {filename} ===\n\n{content}"
```

Document Search Tool

```python
class DocumentSearchTool(BaseTool):
    name = "document_search"
    description = """Search across all policy documents for specific terms.

    Args:
        search_term: Term to search for

    Returns:
        Matching excerpts with document names."""

    def _run(self, search_term: str) -> str:
        results = []
        docs_dir = "policy_documents"

        for filename in os.listdir(docs_dir):
            filepath = os.path.join(docs_dir, filename)
            with open(filepath, 'r') as f:
                content = f.read()

            if search_term.lower() in content.lower():
                # Find relevant excerpt
                idx = content.lower().find(search_term.lower())
                excerpt = content[max(0, idx-100):idx+200]
                results.append(f"[{filename}]: ...{excerpt}...")

        if not results:
            return f"No matches found for '{search_term}'"

        return "\n\n".join(results)
```

Sample Policy Documents

Data Governance Policy

```
# Data Governance Policy

## Purpose
Establish framework for data management across the organization.

## Scope
All business units handling customer or financial data.

## Data Classification
- Public: Marketing materials
- Internal: Business operations
```

```
- Confidential: Customer PII
- Restricted: Financial records, credentials

## Roles and Responsibilities
- Data Owner: Business accountability
- Data Steward: Day-to-day management
- Data Custodian: Technical implementation

## Compliance Requirements
- GDPR Article 5: Data processing principles
- SOX Section 404: Internal controls
- Basel III: Operational risk data
```

Task Definitions

Ingestion Task

```
Task(
    description="""Perform comprehensive ingestion of all policy documents.

    1. List all available policy documents
    2. Read each document thoroughly
    3. For each document, identify:
       - Document title and purpose
       - Key policy statements
       - Compliance obligations
       - Referenced regulations
    4. Note cross-references between documents
    5. Flag unclear or incomplete areas""",
    expected_output="""Structured extraction containing:
    1. Document inventory with metadata
    2. Key requirements by theme
    3. Compliance controls
    4. Cross-reference map
    5. Initial observations"""
)
```

Analysis Task

```
Task(
    description="""Analyze extracted policy content for compliance posture.

    1. Regulatory Mapping
       - Map policies to GDPR, SOX, Basel, etc.
       - Identify coverage and gaps

    2. Gap Analysis
```

```
        - Missing policies
        - Inconsistencies
        - Outdated sections

    3. Risk Assessment
        - Prioritize gaps by impact
        - Consider enforcement trends

    4. Control Effectiveness
        - Are stated controls adequate?
        - Missing enforcement mechanisms?""",
    expected_output="""Analysis report containing:
    1. Regulatory mapping matrix
    2. Prioritized gap inventory
    3. Control assessment
    4. Risk heat map"""
)
```

Report Task

```
Task(
    description="""Generate professional compliance report.

    EXECUTIVE SUMMARY:
    - Overall compliance score
    - Key findings (top 5)
    - Strategic recommendations

    DETAILED FINDINGS:
    - Complete gap analysis
    - Regulatory mapping
    - Remediation roadmap

    APPENDICES:
    - Document inventory
    - Glossary""",
    expected_output="Professional compliance report in markdown",
    output_file="output/compliance_report.md"
)
```

Configurable Options

```
def create_policy_analysis_crew(
    document_focus: str = None,     # Focus on specific doc/topic
    focus_areas: list = None,       # e.g., ["GDPR", "data governance"]
    report_type: str = "full"       # "executive", "detailed", "full"
) -> Crew:
```

Sample Output

```
# Compliance Assessment Report

## Executive Summary

**Overall Compliance Score: 72/100**

### Key Findings
1. **CRITICAL**: No data retention policy defined
2. **HIGH**: Privacy policy missing CCPA requirements
3. **HIGH**: Risk framework lacks operational risk metrics
4. **MEDIUM**: Data classification inconsistent across docs
5. **LOW**: Document version control needs improvement

### Recommended Actions
1. Develop data retention policy (2 weeks)
2. Update privacy policy for CCPA (1 week)
3. Add operational risk metrics to framework (3 weeks)

## Regulatory Coverage Matrix

| Regulation | Coverage | Gaps |
|------------|----------|------|
| GDPR | 75% | Retention, DPIA |
| SOX | 80% | IT controls documentation |
| Basel III | 60% | Operational risk data |
| CCPA | 40% | Opt-out, disclosure |

## Detailed Findings
...
```

Running the Project

```bash
# Clone
git clone https://github.com/Dewale-A/AgenticAI-Policy-Documents-Application.git
cd AgenticAI-Policy-Documents-Application

# Setup
python -m venv venv
source venv/bin/activate
pip install -r requirements.txt

# Configure
cp .env.example .env
```

```
# Add OPENAI_API_KEY (or ANTHROPIC_API_KEY)

# Run
python main.py

# With options
python main.py --focus "data governance" --report-type executive
```

Key Learnings

1. **Document tools are simple** – Just file I/O, but critical for agent access
2. **Analysis agent does the heavy lifting** – Mapping, gaps, risk assessment
3. **Delegation enabled** – Analysis agent can ask for more document searches
4. **Output file** – Report saved directly to markdown file

Next Up

Section 3: Agentic Data Quality – automated data assessment with CDE focus. # Part 3, Section 3: Agentic Data Quality

Project Overview

A multi-agent system that assesses data quality with special focus on Critical Data Elements (CDEs).

GitHub: `Dewale-A/AgenticDataQuality`

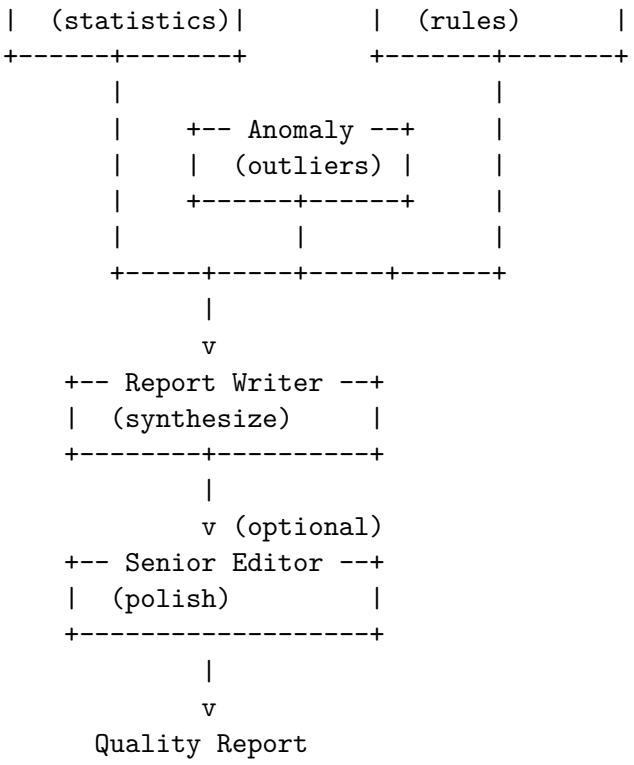
The Business Problem

Data quality issues cost organizations millions: - Bad decisions from bad data - Regulatory fines for data governance failures - Customer trust erosion - Operational inefficiencies

Critical Data Elements (CDEs) are the most important fields – customer IDs, account numbers, transaction amounts. These need extra scrutiny.

Architecture

```
            Data File (CSV)
                  |
           CDE Config (JSON)
                  |
     +------------------------+
     |                        |
     v                        v
+-- Profiler --+        +-- Validator --+
```



4-5 Agents (Editor optional), Sequential Process

The Agents

1. Data Profiler Agent

```
Agent(
    role="Data Profiler",
    goal="Thoroughly profile datasets with special attention to CDEs",
    backstory="""You are an expert Data Profiler with 15 years of experience
    in data governance. You have a keen eye for identifying data patterns,
    anomalies, and quality issues. You understand the critical importance
    of CDEs in enterprise data management.""",
    tools=[DataLoaderTool(), CDELoaderTool(), ProfilerTool()],
)
```

Job: Analyze data structure, completeness, distributions. Flag CDE issues.

2. Data Validator Agent

```
Agent(
    role="Data Validator",
    goal="Validate data against business rules and CDE requirements",
    backstory="""You are a meticulous Data Validator specializing in
    regulatory compliance and data integrity. You've developed validation
    frameworks used by Fortune 500 companies and prevented millions in
```

```
    regulatory fines. You treat CDEs with extra scrutiny.""",
    tools=[ValidatorTool()],
)
```

Job: Check formats, ranges, business rules. Categorize by severity.

3. Anomaly Detector Agent

```
Agent(
    role="Anomaly Detector",
    goal="Identify statistical outliers and unusual patterns",
    backstory="""You are a Statistical Analyst specializing in anomaly
    detection. With a PhD in Statistics, you've developed methods for
    identifying anomalies that traditional rules miss.""",
    tools=[AnomalyDetectorTool()],
)
```

Job: Find statistical outliers using Z-score and IQR methods.

4. Report Writer Agent

```
Agent(
    role="Data Quality Report Writer",
    goal="Synthesize findings into clear, actionable reports",
    backstory="""You are a Technical Writer specializing in data governance.
    You translate complex findings into business-friendly reports that
    executives can understand and act upon.""",
)
```

Job: Create comprehensive report with scores and recommendations.

5. Senior Editor Agent (Optional)

```
Agent(
    role="Senior Report Editor",
    goal="Polish reports to executive presentation standards",
    backstory="""You are a Senior Editor with 20 years in corporate
    communications. Your edited reports have been presented to boards
    of directors. You transform good reports into exceptional ones.""",
    llm="gpt-4o",  # More powerful model for polish
)
```

Job: Refine language, fix formatting, ensure C-suite ready.

The Tools

Data Loader Tool

```
class DataLoaderTool(BaseTool):
    name = "data_loader"
```

```
    description = "Load a CSV data file for analysis"

    def _run(self, filepath: str) -> str:
        df = pd.read_csv(filepath)
        return f"Loaded {len(df)} rows, {len(df.columns)} columns: {list(df.columns)}"
```

CDE Loader Tool

```
class CDELoaderTool(BaseTool):
    name = "cde_loader"
    description = "Load CDE configuration defining critical fields"

    def _run(self, config_path: str) -> str:
        with open(config_path) as f:
            config = json.load(f)
        return json.dumps(config, indent=2)
```

Profiler Tool

```
class ProfilerTool(BaseTool):
    name = "data_profiler"
    description = "Profile a column with statistics"

    def _run(self, column_name: str) -> str:
        col = self._df[column_name]
        stats = {
            "total_count": len(col),
            "null_count": col.isnull().sum(),
            "null_percent": round(col.isnull().sum() / len(col) * 100, 2),
            "unique_count": col.nunique(),
            "unique_percent": round(col.nunique() / len(col) * 100, 2),
        }
        if col.dtype in ['int64', 'float64']:
            stats.update({
                "min": col.min(),
                "max": col.max(),
                "mean": round(col.mean(), 2),
                "std": round(col.std(), 2)
            })
        return json.dumps(stats, indent=2)
```

Validator Tool

```
class ValidatorTool(BaseTool):
    name = "data_validator"
    description = "Validate data against business rules"

    def _run(self, validation_type: str, column: str) -> str:
        issues = []
```

```
        if validation_type == "email":
            invalid = self._df[~self._df[column].str.match(r'^[\w.-]+@[\w.-]+\.\w+$')]
            issues = [{"row": i, "value": v} for i, v in invalid[column].items()]

        elif validation_type == "credit_score":
            invalid = self._df[(self._df[column] < 300) | (self._df[column] > 850)]
            issues = [{"row": i, "value": v} for i, v in invalid[column].items()]

        return json.dumps({
            "validation": validation_type,
            "column": column,
            "issues_found": len(issues),
            "sample_issues": issues[:5]
        })
```

Anomaly Detector Tool

```
class AnomalyDetectorTool(BaseTool):
    name = "anomaly_detector"
    description = "Detect statistical outliers in numeric columns"

    def _run(self, column: str, method: str = "zscore") -> str:
        col = self._df[column].dropna()

        if method == "zscore":
            z_scores = np.abs((col - col.mean()) / col.std())
            outliers = col[z_scores > 3]
        else:  # IQR
            Q1, Q3 = col.quantile([0.25, 0.75])
            IQR = Q3 - Q1
            outliers = col[(col < Q1 - 1.5*IQR) | (col > Q3 + 1.5*IQR)]

        return json.dumps({
            "column": column,
            "method": method,
            "outlier_count": len(outliers),
            "outlier_percent": round(len(outliers) / len(col) * 100, 2),
            "sample_outliers": outliers.head(5).tolist()
        })
```

CDE Configuration

```
{
  "critical_data_elements": [
    {
      "field_name": "customer_id",
```

```
      "business_definition": "Unique identifier for each customer",
      "data_owner": "Customer Data Management",
      "nullable": false,
      "unique": true,
      "format": "CUS-\\d{6}"
    },
    {
      "field_name": "account_balance",
      "business_definition": "Current balance in customer account",
      "data_owner": "Finance",
      "nullable": false,
      "min_value": 0,
      "regulatory_requirement": "SOX Section 404"
    },
    {
      "field_name": "credit_score",
      "business_definition": "Customer credit rating",
      "data_owner": "Risk Management",
      "nullable": false,
      "min_value": 300,
      "max_value": 850
    }
  ]
}
```

The Polish Feature

The `--polish` flag adds a Senior Editor agent:

```python
class DataQualityCrew:
    def __init__(self, data_file: str, polish: bool = False):
        self.polish = polish

        # Core agents always created
        self.profiler = create_profiler_agent()
        self.validator = create_validator_agent()
        self.anomaly = create_anomaly_detector_agent()
        self.writer = create_report_writer_agent()

        # Optional editor for executive polish
        if self.polish:
            self.editor = create_senior_editor_agent()
```

Without polish: Good technical report **With polish:** Executive-ready, boardroom-quality document

---

Sample Output

```
# Data Quality Assessment Report

**Dataset:** customers.csv
**Assessment Date:** 2024-02-19
**Overall Score:** 78/100

---

## Executive Summary

The customer dataset shows **good overall quality** with specific
issues requiring attention in email validation and credit score ranges.

### Key Metrics
| Dimension | Score | Status |
|-----------|-------|--------|
| Completeness | 94% | GOOD |
| Validity | 72% | NEEDS ATTENTION |
| Uniqueness | 100% | EXCELLENT |
| CDE Compliance | 85% | GOOD |

### Critical Findings
1. **HIGH** - 127 invalid email addresses (8.5%)
2. **HIGH** - 23 credit scores outside valid range
3. **MEDIUM** - Missing phone numbers for 45 records

---

## CDE Analysis

### customer_id
- Status: COMPLIANT
- Completeness: 100%
- Uniqueness: 100%

### account_balance
- Status: WARNING
- Issue: 3 negative balances detected
- Impact: SOX 404 compliance risk

### credit_score
- Status: NON-COMPLIANT
- Issue: 23 values outside 300-850 range
- Action Required: Investigate data source

---
```

```
## Recommendations

### Immediate (This Week)
1. Correct invalid credit scores
2. Validate email addresses at entry

### Short-term (This Month)
1. Implement real-time validation
2. Add monitoring for CDE fields

### Long-term (This Quarter)
1. Establish data quality dashboard
2. Automate monthly assessments
```

Running the Project

```bash
# Clone
git clone https://github.com/Dewale-A/AgenticDataQuality.git
cd AgenticDataQuality

# Setup
python -m venv venv
source venv/bin/activate
pip install -r requirements.txt

# Configure
cp .env.example .env
# Add OPENAI_API_KEY

# Run basic assessment
python main.py --data sample_data/customers.csv

# Run with CDE config
python main.py --data sample_data/customers.csv --cde sample_data/cde_config.json

# Run with executive polish
python main.py --data sample_data/customers.csv --cde sample_data/cde_config.json --polish
```

Key Learnings

1. **CDE focus differentiates** – Not just profiling, but business-critical field emphasis
2. **Multiple quality dimensions** – Completeness, validity, uniqueness, anomalies
3. **Optional agents** – `--polish` flag shows how to add agents conditionally
4. **Different models per agent** – Editor uses GPT-4o for better quality

5. **Severity categorization** – CRITICAL/HIGH/MEDIUM/LOW helps prioritization

---

Part 3 Complete!

You've seen three production multi-agent projects: - **Loan Origination**: 6-agent sequential workflow with tools - **Policy Documents**: 3-agent document analysis pipeline - **Data Quality**: 4-5 agent assessment with optional polish

Next: Part 4 – Production Patterns for deploying multi-agent systems. # Part 4, Section 1: Error Handling in Agent Systems

Why Agent Errors Are Different

Multi-agent systems have compounding failure modes:

```
Single Agent:
  Error -> Task fails

Multi-Agent (6 agents):
  Agent 3 error -> Context corrupted -> Agents 4,5,6 produce garbage
```

Common Failure Modes

1. Tool Failures

```
# Agent tries to load non-existent application
[Tool Error] application_loader: File not found: APP999.json
```

Solution: Tools should return error strings, not raise exceptions:

```python
class ApplicationLoaderTool(BaseTool):
    def _run(self, application_id: str) -> str:
        filepath = f"applications/{application_id}.json"

        if not os.path.exists(filepath):
            return f"ERROR: Application {application_id} not found. Available: {self._list_apps

        try:
            with open(filepath) as f:
                return json.dumps(json.load(f), indent=2)
        except json.JSONDecodeError:
            return f"ERROR: {application_id} contains invalid JSON"
```

2. LLM Rate Limits

```
openai.RateLimitError: Rate limit exceeded
```

Solution: Retry with backoff:

```python
from tenacity import retry, stop_after_attempt, wait_exponential

@retry(
    stop=stop_after_attempt(3),
    wait=wait_exponential(multiplier=1, min=2, max=10)
)
def run_crew_with_retry(crew):
    return crew.kickoff()
```

3. Context Overflow

Agent receives too much context and gets confused.

Solution: Selective context passing:

```python
# Bad: Pass everything
task5.context = [task1, task2, task3, task4]

# Good: Pass only what's needed
task5.context = [task3, task4]  # Only analysis tasks
```

4. Agent Loops

Agent keeps calling tools without progressing.

Solution: Set max iterations:

```python
agent = Agent(
    ...,
    max_iter=15,  # Limit reasoning loops
)
```

5. Hallucinated Tool Calls

Agent invents tools that don't exist.

Solution: Clear tool descriptions:

```python
class CreditCheckTool(BaseTool):
    name = "credit_check"  # Exact name agent must use
    description = """ONLY use this to check credit scores.

    Required args: credit_score (int 300-850)
    Returns: Credit tier and risk assessment"""
```

Defensive Crew Design

Wrap Kickoff

```python
def process_application(app_id: str) -> dict:
    try:
```

```
        crew = create_loan_crew(app_id)
        result = crew.kickoff()
        return {"status": "success", "result": str(result)}

    except FileNotFoundError as e:
        return {"status": "error", "type": "missing_file", "message": str(e)}

    except Exception as e:
        logger.error(f"Crew failed for {app_id}: {e}")
        return {"status": "error", "type": "unknown", "message": str(e)}
```

Validate Inputs

```
def create_loan_crew(app_id: str) -> Crew:
    # Validate before creating crew
    if not app_id or not app_id.startswith("APP"):
        raise ValueError(f"Invalid application ID: {app_id}")

    app_path = f"applications/{app_id}.json"
    if not os.path.exists(app_path):
        raise FileNotFoundError(f"Application not found: {app_id}")

    # Now safe to create crew
    return _build_crew(app_id)
```

Timeout Protection

```
import signal

def timeout_handler(signum, frame):
    raise TimeoutError("Crew execution timed out")

def run_with_timeout(crew, timeout_seconds=300):
    signal.signal(signal.SIGALRM, timeout_handler)
    signal.alarm(timeout_seconds)

    try:
        result = crew.kickoff()
        signal.alarm(0)  # Cancel alarm
        return result
    except TimeoutError:
        logger.error("Crew timed out")
        raise
```

Graceful Degradation

When full workflow fails, provide partial value:

```python
def process_loan(app_id: str):
    try:
        # Full crew
        result = run_full_crew(app_id)
        return {"complete": True, "result": result}

    except CreditCheckFailure:
        # Partial crew without credit analysis
        result = run_partial_crew(app_id, skip=["credit"])
        return {
            "complete": False,
            "result": result,
            "warning": "Credit analysis unavailable - manual review required"
        }

    except Exception as e:
        # Minimum viable response
        return {
            "complete": False,
            "result": None,
            "error": str(e),
            "suggestion": "Please retry or contact support"
        }
```

Interview Questions

Q: How do you handle errors in multi-agent systems?

> I design tools to return error strings rather than throw exceptions, so agents can reason about failures. I wrap crew execution in try-catch blocks, validate inputs before starting, and implement graceful degradation – if credit analysis fails, I can still provide partial results and flag for manual review. I also set max iterations to prevent agent loops.

Next Up

Section 2: Observability – seeing inside your agent systems. # Part 4, Section 2: Observability & Debugging

The Visibility Problem

Multi-agent systems are black boxes without observability:

```
Input --> [???] --> Output

"Why did it deny the loan?"
"I don't know, it just did."
```

Verbose Mode

The simplest observability – see agent thinking:

```
crew = Crew(
    agents=[...],
    tasks=[...],
    verbose=True  # Shows everything
)
```

Output:

```
[Agent: Credit Analyst] Starting task...
[Agent: Credit Analyst] Thinking: I need to check the credit score first.
[Agent: Credit Analyst] Using tool: credit_check
[Tool: credit_check] Input: {"credit_score": 680}
[Tool: credit_check] Output: {"tier": 3, "risk": "moderate"}
[Agent: Credit Analyst] Thinking: Credit tier 3 indicates moderate risk...
[Agent: Credit Analyst] Task complete.
```

Structured Logging

Log key events for production monitoring:

```
import structlog

logger = structlog.get_logger()

def process_loan(app_id: str):
    logger.info("loan_processing_started", application_id=app_id)

    try:
        crew = create_loan_crew(app_id)

        start_time = time.time()
        result = crew.kickoff()
        duration = time.time() - start_time

        logger.info("loan_processing_complete",
            application_id=app_id,
            duration_seconds=round(duration, 2),
            decision=extract_decision(result)
        )

        return result

    except Exception as e:
```

```
        logger.error("loan_processing_failed",
            application_id=app_id,
            error=str(e),
            error_type=type(e).__name__
        )
        raise
```

Key Metrics to Track

Per-Crew Metrics

Metric	Description	Alert Threshold
`crew_duration_seconds`	Total execution time	> 120s
`tasks_completed`	Number of tasks finished	$<$ expected
`tool_calls_total`	Total tool invocations	> 50
`llm_tokens_used`	Token consumption	$>$ budget

Per-Agent Metrics

Metric	Description	Alert Threshold
`agent_iterations`	ReAct loops	> 10
`agent_tool_calls`	Tools used	> 15
`agent_duration`	Time per agent	> 30s

Business Metrics

Metric	Description
`loans_approved`	Approvals count
`loans_denied`	Denials count
`approval_rate`	Approved / Total
`avg_processing_time`	Mean duration

Log File Output

```
crew = Crew(
    agents=[...],
    tasks=[...],
    verbose=True,
    output_log_file="logs/crew_execution.log"
)
```

Custom Callbacks

Track events during execution:

```
class CrewObserver:
    def __init__(self):
        self.events = []

    def on_task_start(self, task):
        self.events.append({
            "event": "task_start",
            "task": task.description[:50],
            "agent": task.agent.role,
            "timestamp": time.time()
        })

    def on_task_complete(self, task, output):
        self.events.append({
            "event": "task_complete",
            "task": task.description[:50],
            "output_length": len(str(output)),
            "timestamp": time.time()
        })

    def on_tool_use(self, tool_name, inputs, output):
        self.events.append({
            "event": "tool_use",
            "tool": tool_name,
            "timestamp": time.time()
        })
```

Debugging Techniques

1. Isolate the Problem

Run one agent at a time:

```
# Instead of full crew
result = crew.kickoff()

# Test each agent individually
for task in tasks:
    mini_crew = Crew(agents=[task.agent], tasks=[task], verbose=True)
    print(f"\n=== Testing: {task.agent.role} ===")
    result = mini_crew.kickoff()
    print(f"Output: {result[:200]}...")
    input("Press Enter to continue...")
```

2. Check Context Flow

Print what each agent receives:

```
def debug_context(task):
    print(f"\nTask: {task.description[:50]}...")
    print(f"Agent: {task.agent.role}")
    if task.context:
        for ctx_task in task.context:
            print(f"  Context from: {ctx_task.agent.role}")
```

3. Tool Testing

Test tools outside of agents:

```
# Test credit check tool directly
tool = CreditCheckTool()
result = tool._run(credit_score=720, payment_history=5)
print(f"Tool output: {result}")

# Verify it's what agent expects
assert "tier" in result.lower()
```

4. Compare Runs

Log outputs and compare:

```
def compare_runs(app_id, run1_log, run2_log):
    """Find where two runs diverged."""
    for i, (e1, e2) in enumerate(zip(run1_log, run2_log)):
        if e1 != e2:
            print(f"Divergence at step {i}:")
            print(f"  Run 1: {e1}")
            print(f"  Run 2: {e2}")
            break
```

Production Dashboard Ideas

```
+-----------------------------------------+
|  Multi-Agent System Dashboard            |
+-----------------------------------------+
|                                         |
|  Active Crews: 3    Completed: 127      |
|  Avg Duration: 45s  Error Rate: 2.1%    |
|                                         |
|  Last 24 Hours:                         |
|  ?????????????????????? 85% success    |
|                                         |
|  By Agent:                              |
```

```
|  Credit Analyst  - 98% success, 12s avg  |
|  Underwriter     - 95% success, 18s avg  |
|  Risk Assessor   - 99% success, 8s avg   |
|                                          |
|  Recent Errors:                          |
|  - APP234: Tool timeout (credit_check)   |
|  - APP238: Context overflow              |
+------------------------------------------+
```

Interview Questions

Q: How do you debug a multi-agent system?

> First, enable verbose mode to see agent thinking. Then isolate – run one agent at a time to find which one fails. Check context flow to ensure agents receive the right information. Test tools independently to verify they work. Log key events and compare successful vs failed runs to find divergence points.

Next Up

Section 3: Cost Management – controlling your LLM spend. # Part 4, Section 3: Cost Management

The Cost Challenge

Multi-agent systems multiply LLM costs:

```
Single Agent:
  1 task x ~2K tokens = $0.01

Multi-Agent (6 agents):
  6 tasks x ~2K tokens = $0.06
  + tool calls
  + retries
  = $0.10-0.20 per execution
```

Scale to 1000 executions/day = $100-200/day

Cost Breakdown

Token Usage by Component

Component	Typical Tokens	Cost (GPT-4o-mini)
Agent system prompt	500-1000	$0.001-0.002
Task description	200-500	$0.0005-0.001
Context (per task)	1000-3000	$0.002-0.006

Component	Typical Tokens	Cost (GPT-4o-mini)
Agent reasoning	500-2000	$0.001-0.004
Tool outputs	200-1000	$0.0005-0.002
Final output	500-2000	$0.001-0.004

Per-Execution Estimate

```
6 agents x ~3000 tokens each = 18,000 tokens
At $0.15/1M input + $0.60/1M output (GPT-4o-mini):
? $0.01-0.05 per loan application
```

Optimization Strategies

1. Right-Size Models

Use cheaper models for simple tasks:

```python
# Simple tasks: cheap model
intake_agent = Agent(
    role="Document Intake",
    llm=LLM(model="openai/gpt-4o-mini", temperature=0.1)
)

# Complex decisions: better model
underwriter_agent = Agent(
    role="Senior Underwriter",
    llm=LLM(model="openai/gpt-4o", temperature=0.1)
)
```

Cost impact: - GPT-4o-mini: $0.15/$0.60 per 1M tokens (input/output) - GPT-4o: $2.50/$10 per 1M tokens - ~16x cheaper for simple tasks!

2. Minimize Context

Don't pass everything to every agent:

```python
# Expensive: Full context chain
task6.context = [task1, task2, task3, task4, task5]  # ~15K tokens

# Cheaper: Selective context
task6.context = [task4, task5]  # ~5K tokens, still has what it needs
```

3. Concise Prompts

```python
# Verbose (costs more)
backstory="""You are a highly experienced senior credit analyst with
over 15 years of experience in consumer lending, mortgage evaluation,
and commercial credit assessment. Throughout your distinguished career,
```

```
you have evaluated thousands of applications..."""  (200 tokens)

# Concise (same effect, fewer tokens)
backstory="""Senior credit analyst, 15 years consumer lending.
Expert at evaluating creditworthiness."""  (20 tokens)
```

4. Cache Tool Results

```
crew = Crew(
    agents=[...],
    tasks=[...],
    cache=True  # Don't re-run identical tool calls
)
```

5. Limit Agent Iterations

```
agent = Agent(
    role="...",
    max_iter=10,  # Prevent runaway reasoning (and cost)
)
```

6. Batch Processing

Process multiple items in one crew run:

```
# Expensive: One crew per application
for app_id in app_ids:
    crew = create_crew(app_id)
    crew.kickoff()  # Full LLM overhead each time

# Cheaper: Batch in single crew
crew = create_batch_crew(app_ids)  # One crew, multiple items
crew.kickoff()
```

Cost Tracking

Track Per Execution

```
import tiktoken

def count_tokens(text, model="gpt-4o-mini"):
    encoding = tiktoken.encoding_for_model(model)
    return len(encoding.encode(text))

class CostTracker:
    def __init__(self, input_rate=0.15, output_rate=0.60):
        self.input_tokens = 0
        self.output_tokens = 0
        self.input_rate = input_rate / 1_000_000
```

```
        self.output_rate = output_rate / 1_000_000

    def add_input(self, text):
        self.input_tokens += count_tokens(text)

    def add_output(self, text):
        self.output_tokens += count_tokens(text)

    @property
    def total_cost(self):
        return (self.input_tokens * self.input_rate +
                self.output_tokens * self.output_rate)

    def report(self):
        return {
            "input_tokens": self.input_tokens,
            "output_tokens": self.output_tokens,
            "estimated_cost": f"${self.total_cost:.4f}"
        }
```

Budget Alerts

```
DAILY_BUDGET = 50.00  # $50/day

def check_budget():
    today_cost = get_today_cost()  # From your tracking

    if today_cost > DAILY_BUDGET * 0.8:
        alert("80% of daily budget consumed")

    if today_cost > DAILY_BUDGET:
        alert("BUDGET EXCEEDED - pausing operations")
        pause_crews()
```

Model Selection Guide

Use Case	Recommended Model	Why
Document intake	GPT-4o-mini	Simple extraction
Verification	GPT-4o-mini	Rule-based checks
Credit analysis	GPT-4o-mini	Tool-heavy, models don't add much
Risk assessment	GPT-4o-mini	Calculations from tools
Underwriting decision	GPT-4o	Complex judgment needed
Report writing	GPT-4o-mini	Synthesis, not reasoning
Executive polish	GPT-4o	Quality matters most

Cost Comparison: Your Projects

Project	Agents	Est. Cost/Run
Loan Origination	6	$0.03-0.08
Policy Documents	3	$0.02-0.05
Data Quality	4-5	$0.03-0.10
Data Quality + Polish	5	$0.05-0.15

With GPT-4o for all agents: 5-10x higher

Interview Questions

Q: How do you manage costs in multi-agent systems?

> Several strategies: Use cheaper models (GPT-4o-mini) for simple tasks, reserve expensive models for complex decisions. Minimize context by only passing what each agent needs. Keep prompts concise. Enable caching for repeated tool calls. Track token usage and set budget alerts. The key is matching model capability to task complexity.

Next Up

Section 4: Testing Agent Systems – ensuring reliability. # Part 4, Section 4: Testing Agent Systems

The Testing Challenge

Multi-agent systems are non-deterministic:

```
Same input -> Different output each time
```

Traditional unit tests don't work well:

```python
# This will fail randomly
def test_loan_decision():
    result = process_loan("APP001")
    assert result == "APPROVED"  # Might say "APPROVED" differently
```

Testing Strategies

1. Tool Unit Tests

Test tools independently – they're deterministic:

```python
def test_credit_check_tool():
    tool = CreditCheckTool()

    # Test excellent credit
```

```
    result = tool._run(credit_score=780)
    assert "Tier 1" in result or "tier 1" in result.lower()

    # Test poor credit
    result = tool._run(credit_score=550)
    assert "Tier 5" in result or "tier 5" in result.lower()

def test_dti_calculator():
    tool = DTICalculatorTool()

    result = json.loads(tool._run(
        annual_income=100000,
        monthly_debts=2000,
        proposed_payment=1500
    ))

    assert result["current_dti"] == 24.0  # 2000/(100000/12)*100
    assert result["proposed_dti"] == 42.0
```

2. Input Validation Tests

Test that bad inputs are handled:

```
def test_application_loader_missing_file():
    tool = ApplicationLoaderTool()
    result = tool._run("NONEXISTENT")
    assert "error" in result.lower() or "not found" in result.lower()

def test_application_loader_invalid_json():
    # Create invalid JSON file
    with open("applications/BAD.json", "w") as f:
        f.write("not valid json")

    tool = ApplicationLoaderTool()
    result = tool._run("BAD")
    assert "error" in result.lower()
```

3. Output Structure Tests

Test that outputs have required fields:

```
def test_loan_output_structure():
    result = process_loan("APP001")

    # Should mention decision
    assert any(word in result.upper() for word in
               ["APPROVED", "DENIED", "CONDITIONAL"])

    # Should include key information
```

```
    result_lower = result.lower()
    assert "credit" in result_lower
    assert "risk" in result_lower

def test_report_has_sections():
    result = run_policy_analysis()

    assert "executive summary" in result.lower()
    assert "findings" in result.lower()
    assert "recommendations" in result.lower()
```

4. Boundary Tests

Test edge cases:

```
@pytest.mark.parametrize("credit_score,expected_tier", [
    (850, 1),  # Maximum
    (300, 5),  # Minimum
    (749, 2),  # Just below tier 1
    (750, 1),  # Exactly tier 1
])
def test_credit_tiers(credit_score, expected_tier):
    tool = CreditCheckTool()
    result = json.loads(tool._run(credit_score=credit_score))
    assert result["credit_tier"] == expected_tier
```

5. Smoke Tests

Run full crew and check it completes:

```
def test_loan_crew_completes():
    """Smoke test - full execution should complete without error."""
    crew = create_loan_origination_crew("APP001")

    try:
        result = crew.kickoff()
        assert result is not None
        assert len(str(result)) > 100  # Produced meaningful output
    except Exception as e:
        pytest.fail(f"Crew execution failed: {e}")
```

6. Regression Tests

Save known-good outputs, compare:

```
def test_regression_app001():
    result = process_loan("APP001")

    # Load expected output patterns
    with open("tests/fixtures/APP001_expected.txt") as f:
```

```
        expected_patterns = f.read().splitlines()

    # Check key phrases appear
    for pattern in expected_patterns:
        assert pattern.lower() in result.lower(), \
            f"Expected pattern not found: {pattern}"
```

7. Agent Isolation Tests

Test each agent individually:

```
def test_credit_analyst_agent():
    agent = credit_analyst_agent()
    task = Task(
        description="Analyze credit for: score=720, payment_history=5yrs",
        expected_output="Credit assessment",
        agent=agent
    )

    crew = Crew(agents=[agent], tasks=[task], verbose=True)
    result = crew.kickoff()

    assert "tier 2" in result.lower() or "720" in result
```

Test Fixtures

Create consistent test data:

```
# tests/fixtures/test_applications.py

GOOD_APPLICANT = {
    "application_id": "TEST001",
    "applicant": {"name": "Test Good"},
    "financial": {
        "annual_income": 100000,
        "credit_score": 750,
        "monthly_debts": 1000
    },
    "loan_request": {"amount": 20000}
}

BAD_APPLICANT = {
    "application_id": "TEST002",
    "applicant": {"name": "Test Bad"},
    "financial": {
        "annual_income": 30000,
        "credit_score": 520,
        "monthly_debts": 2000
```

```
    },
    "loan_request": {"amount": 50000}
}
```

Mocking LLMs for Speed

Full LLM calls are slow. Mock for unit tests:

```python
from unittest.mock import patch, MagicMock

def test_tool_usage_without_llm():
    # Mock LLM response
    mock_response = MagicMock()
    mock_response.content = "Credit tier: 2. Recommendation: APPROVE"

    with patch('crewai.LLM') as mock_llm:
        mock_llm.return_value.invoke.return_value = mock_response

        # Test runs without real LLM
        result = process_loan("APP001")

        assert mock_llm.return_value.invoke.called
```

Continuous Integration

```yaml
# .github/workflows/test.yml
name: Tests

on: [push, pull_request]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2

      - name: Set up Python
        uses: actions/setup-python@v2
        with:
          python-version: '3.11'

      - name: Install dependencies
        run: pip install -r requirements.txt

      - name: Run unit tests (no LLM)
        run: pytest tests/unit -v
```

```yaml
      - name: Run tool tests
        run: pytest tests/tools -v

      # Integration tests only on main (cost $)
      - name: Run integration tests
        if: github.ref == 'refs/heads/main'
        env:
          OPENAI_API_KEY: ${{ secrets.OPENAI_API_KEY }}
        run: pytest tests/integration -v --timeout=120
```

Test Organization

```
tests/
+-- unit/
|   +-- test_tools.py        # Tool unit tests (no LLM)
|   +-- test_utilities.py    # Helper function tests
+-- tools/
|   +-- test_credit_check.py
|   +-- test_dti_calculator.py
|   +-- test_risk_scoring.py
+-- integration/
|   +-- test_loan_crew.py    # Full crew tests (needs LLM)
|   +-- test_policy_crew.py
+-- fixtures/
|   +-- applications/
|   +-- expected_outputs/
+-- conftest.py              # Pytest configuration
```

Interview Questions

Q: How do you test multi-agent systems?

> I use layered testing. Tools get traditional unit tests since they're deterministic. For agents, I test output structure (does it have required fields?) rather than exact values. I use smoke tests to ensure crews complete without error. I save known-good outputs for regression testing. Integration tests run on CI but less frequently due to LLM costs.

Part 4 Complete!

You now understand production patterns: - Error handling and graceful degradation - Observability and debugging - Cost management - Testing strategies

Next: Part 5 – Interview Ready. # Part 5, Section 1: Multi-Agent Concepts Q&A

Core Concept Questions

Q: What is a multi-agent system and why use it?

Strong Answer: > A multi-agent system uses multiple specialized AI agents working together instead of one agent doing everything. Each agent has a specific role, goal, and tools. > > I use multi-agent when: > 1. The task has distinct roles (like loan processing with intake, credit, underwriting) > 2. One agent would get overwhelmed with context > 3. I need different expertise for different parts > 4. I want to debug and improve components independently > > For my loan origination project, I have 6 agents because each mirrors a real specialist – intake, verification, credit analyst, risk assessor, underwriter, and offer generator.

Q: Explain the difference between sequential and hierarchical orchestration.

Strong Answer: > In **sequential**, tasks run one after another in a fixed order. Agent A finishes, then Agent B starts with A's output. It's predictable and great for workflows with clear steps. > > In **hierarchical**, a manager agent receives the task and dynamically delegates to worker agents. The manager decides which workers to use and how to combine their results. > > I use sequential for business workflows like loan processing because they mirror existing processes. I'd use hierarchical for open-ended research where the approach isn't predetermined.

Q: How do agents communicate in CrewAI?

Strong Answer: > Through context passing. When I define a task, I set `task.context = [previous_task]`. The agent receives the previous task's output in its prompt. > > I'm selective about context – I don't pass everything to every agent. For example, my underwriter agent gets context from verification, credit, and risk tasks, but not the raw intake. This keeps context manageable and focused.

Q: What makes a good agent backstory?

Strong Answer: > A good backstory establishes expertise level, working style, and quality expectations. It shapes how the LLM approaches the task. > > For example, my credit analyst has: "10 years of consumer lending experience, known for thorough, balanced assessments." This tells the LLM to be experienced and balanced, not aggressive or superficial. > > Key elements: years of experience, specific domain, working style, quality standards. Keep it focused – a paragraph, not a page.

Q: How do tools extend agent capabilities?

Strong Answer: > LLMs can think but can't act. Tools bridge that gap – they're functions agents can call. > > For example, my credit analyst has a `credit_check` tool that calculates credit tiers from a score, and a `dti_calculator` that computes debt-to-income ratios. The agent reasons

about what to do, calls the tools, and interprets results. > > Good tools have clear names, detailed descriptions (so agents know when to use them), and return structured output the agent can parse.

Architecture Questions

Q: Walk me through how you'd design a multi-agent system.

Strong Answer: > I follow four steps: > > 1. **Map the process**: How do humans do this today? For loan processing, I mapped out intake -> verification -> credit -> risk -> decision -> offer. > > 2. **Define agents**: One agent per major role. Each gets a role, goal, backstory, and relevant tools. > > 3. **Design tasks**: What does each agent need to do? What inputs do they need? What outputs should they produce? > > 4. **Connect with context**: Which tasks depend on which? I set `task.context` to pass information between agents. > > Then I test iteratively – run with verbose mode, see what works, adjust.

Q: How do you decide how many agents to use?

Strong Answer: > I start with the minimum that makes sense. Each agent adds latency and cost. > > I ask: "Can this task be split into distinct roles with different expertise?" If yes, those become agents. If roles blur together, I keep them as one agent. > > For loan origination, I have 6 agents because each has genuinely different expertise and tools. For policy analysis, I only need 3 – ingestion, analysis, reporting. > > I wouldn't use 6 agents if 3 could do it. I also wouldn't force everything into 1 agent if it makes the task too complex.

Q: What's the role of context in task design?

Strong Answer: > Context is how information flows between agents. But more isn't always better. > > If I pass all previous outputs to every agent, context grows huge – slower, more expensive, and agents lose focus. Instead, I'm selective. > > My underwriter doesn't need the raw application data – they need the analyzed results from verification, credit, and risk. So I set: > `python > underwriting_task.context = [verification_task, credit_task, risk_task] > > >` Not `[intake_task, verification_task, credit_task, risk_task]`.

Practical Questions

Q: How do you handle errors in multi-agent systems?

Strong Answer: > At multiple levels: > > 1. **Tools** return error strings instead of throwing exceptions. So if a file isn't found, the agent sees "Error: not found" and can reason about it. > > 2. **Crew execution** is wrapped in try-catch. If something fails, I log it and can return partial results. > > 3. **Graceful degradation**: If credit analysis fails, I might still return verification results and flag for manual review. > > 4. **Max iterations**: I set limits so agents don't loop forever if confused.

Q: How do you manage costs?

Strong Answer: > Several strategies: > > 1. **Right-size models**: GPT-4o-mini for simple tasks, GPT-4o only for complex decisions. > > 2. **Minimize context**: Only pass what each agent needs. > > 3. **Concise prompts**: Keep backstories focused. > > 4. **Caching**: Don't re-run identical tool calls. > > 5. **Tracking**: Log token usage, set budget alerts. > > My loan origination crew costs about $0.03-0.08 per application with GPT-4o-mini. Using GPT-4o everywhere would be 5-10x more.

Q: How do you test multi-agent systems?

Strong Answer: > Layered approach: > > 1. **Tool unit tests**: Tools are deterministic, so I test them traditionally. > > 2. **Output structure tests**: Does the result contain required fields? I don't check exact values because LLMs vary. > > 3. **Smoke tests**: Does the crew complete without errors? > > 4. **Regression tests**: I save known-good outputs and check that key patterns still appear. > > 5. **Agent isolation**: Test each agent individually before testing the full crew. > > Integration tests with real LLMs run on CI but less frequently due to cost.

Q: When would you NOT use multi-agent?

Strong Answer: > When it's overkill: > > - Simple tasks that one agent handles fine > - Latency-critical applications (multi-agent adds seconds) > - Tight budgets (more agents = more LLM calls) > - No clear role separation > > I start with single agent and only go multi-agent when I hit specific pain points: context overflow, need for distinct expertise, or debugging difficulty.

Next Up

Section 2: System Design Scenarios – whiteboard problems for multi-agent. # Part 5, Section 2: System Design Scenarios

Scenario 1: Automated Insurance Claims Processing

Prompt: "Design a multi-agent system to process insurance claims."

Requirements Clarification

Ask: - What types of claims? (Auto, health, property) - What's the volume? (1000/day) - What decisions are needed? (Approve, deny, investigate) - Integration requirements? (Policy database, payment system)

Architecture

```
Claim Submission
      |
      v
[Document Intake Agent]
```

```
  - Extract claim details
  - Identify claim type
  - Check completeness
      |
      v
[Policy Verification Agent]
  - Verify coverage
  - Check policy status
  - Identify exclusions
      |
      v
[Fraud Detection Agent]
  - Pattern analysis
  - Red flag identification
  - Risk scoring
      |
      v
[Assessment Agent]
  - Damage evaluation
  - Coverage calculation
  - Deductible application
      |
      v
[Decision Agent]
  - Approve/Deny/Investigate
  - Calculate payout
  - Generate explanation
      |
      v
[Communication Agent]
  - Draft customer letter
  - Prepare internal report
```

Key Design Decisions

1. **Sequential process** – Each step depends on previous
2. **Fraud detection early** – Before spending time on assessment
3. **Separate communication** – Customer-facing vs internal needs differ
4. **Tools**: Policy lookup, fraud pattern database, payout calculator

---

Scenario 2: Regulatory Compliance Monitoring

Prompt: "Design a system to monitor regulatory changes and assess impact."

Architecture

```
                Regulatory Sources
```

```
    (SEC, Fed, State sites)
             |
             v
[Monitoring Agent] (scheduled)
  - Scan for new regulations
  - Extract key changes
  - Classify by domain
             |
             v
[Impact Analysis Agent]
  - Map to internal policies
  - Identify affected processes
  - Assess compliance gaps
             |
             v
[Prioritization Agent]
  - Risk scoring
  - Deadline tracking
  - Resource estimation
             |
             v
[Report Agent]
  - Executive summary
  - Detailed action items
  - Timeline recommendations
```

Key Decisions

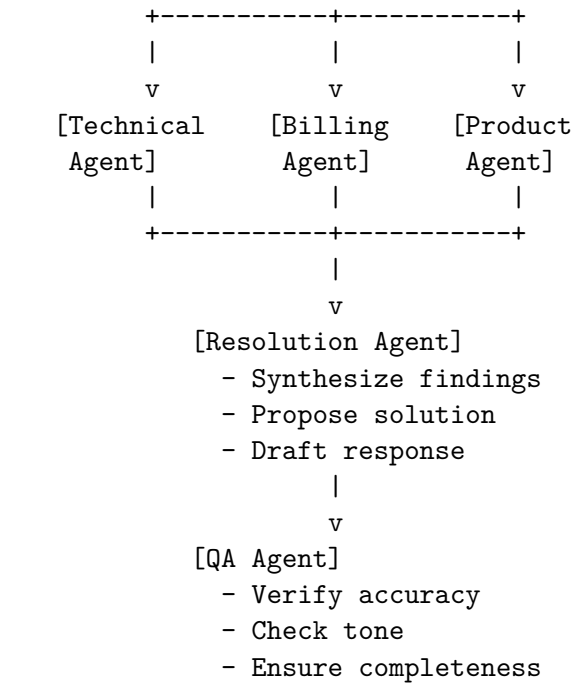
1. **Scheduled execution** – Run monitoring daily/weekly
2. **Hierarchical for analysis** – Different regs need different expertise
3. **Tools**: Web scraping, internal policy search, compliance database
4. **Output**: Actionable report with deadlines

Scenario 3: Customer Support Escalation

Prompt: "Design a multi-agent system for complex support tickets."

Architecture

```
    Escalated Ticket
          |
          v
  [Triage Agent]
    - Categorize issue
    - Assess urgency
    - Identify customer tier
          |
```



Key Decisions

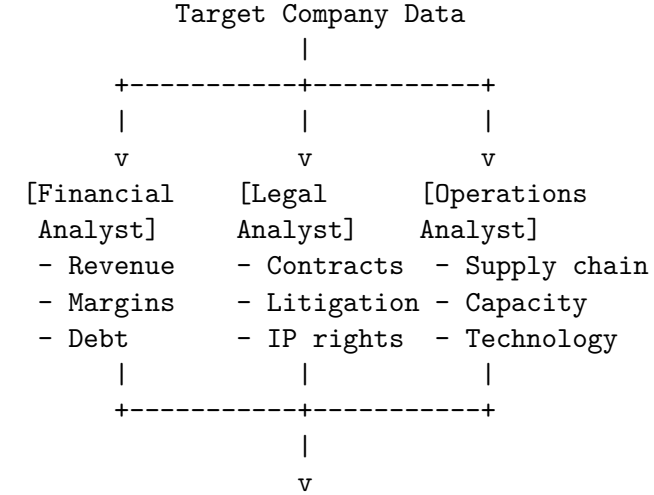
1. **Parallel specialists** – Technical, billing, product can work simultaneously
2. **QA at the end** – Catch errors before customer sees response
3. **Customer context** – Pass account history to all agents
4. **Escalation path** – If confidence low, flag for human review

---

Scenario 4: M&A Due Diligence

Prompt: "Design a system to analyze target companies for acquisition."

Architecture



```
[Risk Assessment Agent]
  - Consolidated risks
  - Deal breakers
  - Mitigation options
        |
        v
[Valuation Agent]
  - Financial models
  - Synergy estimates
  - Price recommendation
        |
        v
[Report Agent]
  - Executive summary
  - Go/No-go recommendation
  - Key findings
```

Key Decisions

1. **Parallel analysis** – Financial, legal, operations are independent
2. **Risk consolidation** – Separate agent to synthesize risks
3. **Valuation last** – Needs all inputs
4. **Tools**: Financial data APIs, legal database search, industry benchmarks

---

Design Framework

For any multi-agent system design:

1. Process Mapping (2 min)

- How is this done manually today?
- What are the distinct phases?

2. Agent Identification (3 min)

- What roles are needed?
- What expertise does each need?
- What tools does each need?

3. Flow Design (3 min)

- Sequential, parallel, or hierarchical?
- What context does each agent need?
- Where are the decision points?

4. Error Handling (2 min)

- What if an agent fails?

- Where are human checkpoints?
- How do you gracefully degrade?

5. Scale & Cost (2 min)

- How many executions per day?
- Which agents need powerful models?
- What's the cost per execution?

Common Follow-ups

"How would you handle errors?" > Tools return error strings. Crew wrapped in try-catch. Graceful degradation – return partial results. Human escalation for low-confidence decisions.

"How would you scale this?" > Run crews in parallel for independent cases. Cache repeated lookups. Use cheaper models for simple agents. Batch similar cases.

"How would you measure success?" > Accuracy vs human decisions. Processing time. Cost per case. Error rate. Customer satisfaction for support scenarios.

"What if latency is a concern?" > Reduce agents. Parallel where possible. Faster models. Pre-compute common patterns. Stream partial results.

Next Up

Section 3: Code Explanation Practice – walking through your implementations. # Part 5, Section 3: Code Explanation Practice

How to Explain Your Multi-Agent Code

When asked about your code in interviews: 1. Start with the business problem 2. Explain the architecture (agents, flow) 3. Walk through key code sections 4. Discuss design decisions 5. Mention what you'd improve

---

Loan Origination Walkthrough

Opening

> "I built a multi-agent loan origination system that processes loan applications through a complete underwriting workflow. It mirrors how real lending institutions operate – specialized roles working sequentially."

Architecture Overview

> "The system has 6 agents: 1. Document Intake – loads and validates applications 2. Verification – checks data consistency 3. Credit Analyst – evaluates creditworthiness

using tools 4. Risk Assessor – calculates risk scores 5. Underwriter – makes the approve/deny decision 6. Offer Generator – structures the loan terms

They run sequentially because each step depends on the previous. The output of one becomes context for the next."

Key Code: Agent Definition

```
def credit_analyst_agent() -> Agent:
    return Agent(
        role="Senior Credit Analyst",
        goal="Analyze applicant creditworthiness and provide detailed assessment",
        backstory="""You are a senior credit analyst with over 10 years of
        experience in consumer lending. You evaluate credit reports, payment
        histories, and outstanding debts. Your analysis forms the foundation
        of lending decisions.""",
        tools=[credit_check, dti_calculator],
        llm=llm,
        verbose=True,
    )
```

"The backstory establishes expertise – '10 years experience' shapes how the LLM approaches the task. I give this agent two tools: credit_check for tier evaluation and dti_calculator for debt ratios."

Key Code: Context Flow

```
verification_task.context = [intake_task]
credit_task.context = [intake_task, verification_task]
risk_task.context = [intake_task, credit_task]
underwriting_task.context = [verification_task, credit_task, risk_task]
offer_task.context = [intake_task, underwriting_task]
```

"Context is selective. The underwriter doesn't need raw application data – they need the analyses. But the offer generator needs both the original amounts and the decision. This keeps context focused and manageable."

Key Code: Tool Implementation

```
class DTICalculatorTool(BaseTool):
    name = "dti_calculator"
    description = """Calculate debt-to-income ratio.

    Args:
        annual_income: Yearly gross income
        monthly_debts: Existing monthly payments
        proposed_payment: New loan payment

    Returns: Current DTI, proposed DTI, assessment"""
```

```
    def _run(self, annual_income, monthly_debts, proposed_payment):
        monthly_income = annual_income / 12
        current_dti = (monthly_debts / monthly_income) * 100
        proposed_dti = ((monthly_debts + proposed_payment) / monthly_income) * 100

        return json.dumps({
            "current_dti": round(current_dti, 2),
            "proposed_dti": round(proposed_dti, 2),
            "assessment": "PASS" if proposed_dti < 43 else "HIGH"
        })
```

"Tools are deterministic – same input, same output. The description is detailed so the agent knows exactly when and how to use it. I return structured JSON so the agent can parse results."

What You'd Improve

"A few things I'd add in production: 1. Parallel verification – identity, income, employment could run simultaneously 2. More sophisticated risk scoring with ML models 3. Audit logging for compliance 4. Human-in-the-loop for edge cases 5. Caching for repeated applicant lookups"

Data Quality Walkthrough

Opening

"This is a multi-agent data quality assessment system. It profiles datasets, validates against business rules, detects anomalies, and generates reports – with special focus on Critical Data Elements."

Unique Feature: Optional Agent

```
class DataQualityCrew:
    def __init__(self, data_file: str, polish: bool = False):
        self.polish = polish

        # Core agents
        self.profiler = create_profiler_agent()
        self.validator = create_validator_agent()
        self.anomaly = create_anomaly_detector_agent()
        self.writer = create_report_writer_agent()

        # Optional editor
        if self.polish:
            self.editor = create_senior_editor_agent()
```

"The `--polish` flag adds a Senior Editor agent. For internal reports, the basic writer is fine. For executive presentations, I add an editor using GPT-4o for higher quality. This shows how to make agents optional based on use case."

Unique Feature: Different Models

```
def create_report_writer_agent() -> Agent:
    return Agent(
        role="Data Quality Report Writer",
        # Uses default model (gpt-4o-mini)
    )

def create_senior_editor_agent() -> Agent:
    return Agent(
        role="Senior Report Editor",
        llm="gpt-4o",  # Better model for polish
    )
```

> "I use GPT-4o-mini for most agents – it's fast and cheap. The editor uses GPT-4o because quality matters most there. This is cost optimization – matching model capability to task complexity."

Policy Documents Walkthrough

Opening

> "This system analyzes policy documents for compliance gaps. Three agents – ingestion, analysis, reporting – work sequentially to read documents, identify issues, and produce actionable reports."

Key Feature: Document Tools

```
class DocumentReaderTool(BaseTool):
    name = "document_reader"
    description = """Read policy documents.

    Args:
        filename: File to read, or 'list' for available files

    Returns: Document content or file list"""

    def _run(self, filename: str = "list") -> str:
        if filename == "list":
            files = os.listdir("policy_documents")
            return f"Available: {files}"

        with open(f"policy_documents/{filename}") as f:
            return f.read()
```

> "The tools are simple – just file I/O. But they're critical because they give agents access to the documents. The 'list' option helps agents discover what's available."

Key Feature: Delegation

```
analysis_agent = Agent(
    role="Policy Compliance Analyst",
    allow_delegation=True,  # Can ask other agents for help
)
```

> "The analysis agent can delegate. If it needs more document searches during analysis, it can request help. This makes the system more flexible than rigid sequential flow."

Common Follow-up Questions

"Why did you choose CrewAI?" > "CrewAI is intuitive for business workflows. The role/goal/backstory pattern maps directly to how I think about specialists. Sequential process matches how these workflows actually run. For more complex orchestration, I might consider LangGraph."

"How do you test this?" > "Tools get unit tests since they're deterministic. For agents, I test output structure – does it have required fields? Smoke tests verify crews complete. I save known-good outputs for regression testing."

"What's the latency?" > "About 30-60 seconds for the full loan workflow. Most time is LLM generation. For faster response, I could reduce agents, use streaming, or pre-compute common patterns."

"How would you scale this?" > "Run crews in parallel for independent applications. Use a queue (Celery, SQS) for background processing. Cache repeated tool calls. The architecture is stateless so horizontal scaling is straightforward."

Practice Exercise

Pick one of your projects and practice:

1. **30-second pitch**: What does it do, why multi-agent?
2. **Architecture walk**: Draw the agents and flow
3. **Code deep-dive**: Explain one agent, one tool, one task
4. **Design decisions**: Why these choices?
5. **Improvements**: What would you add?

Congratulations!

You've completed **The AI Developer Bible: Multi-Agent Systems Edition**.

You now have: - Deep understanding of multi-agent architectures - Hands-on experience with 3 production projects - Production patterns for deployment - Interview preparation with Q&A and system design

Go build autonomous systems!