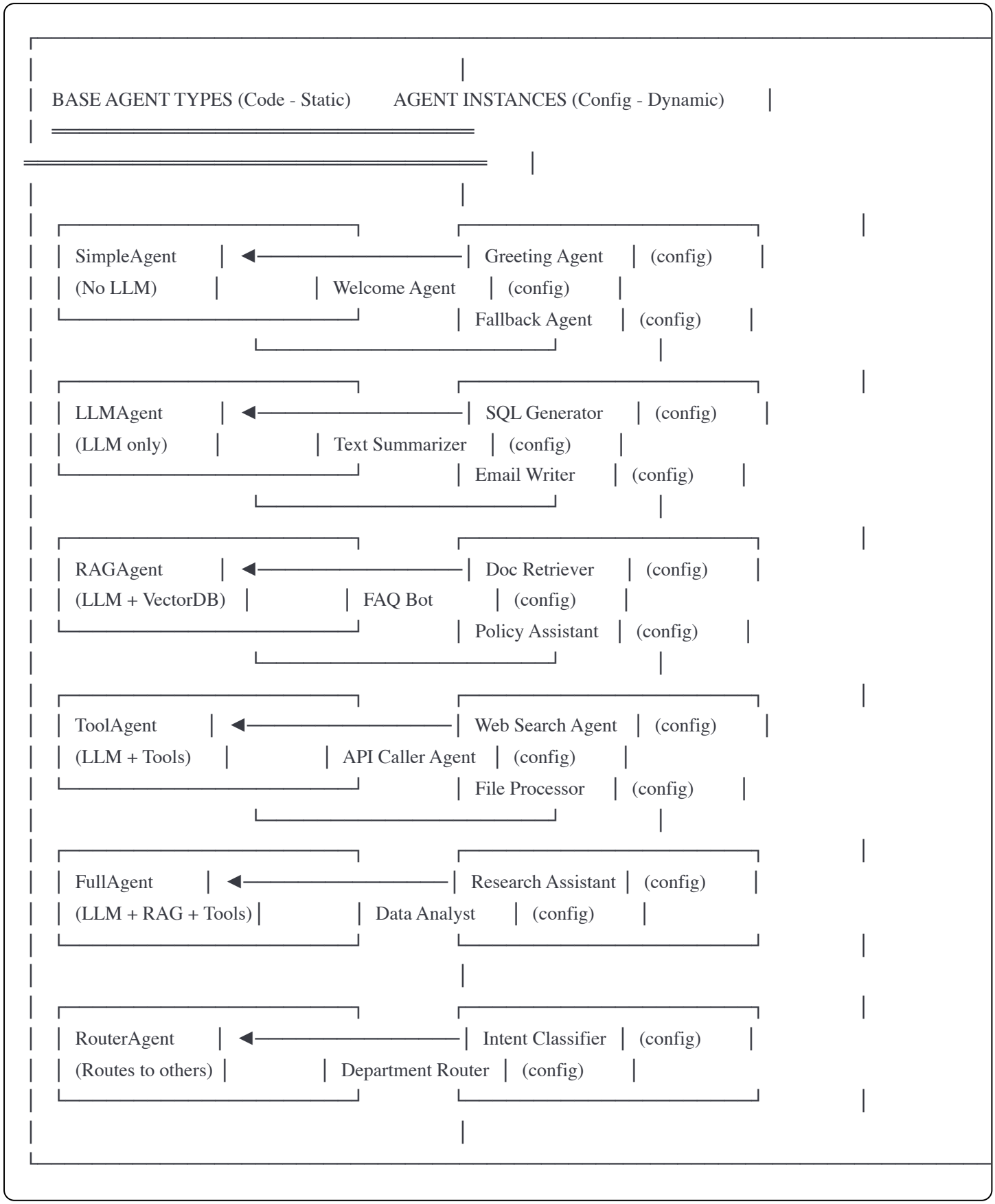
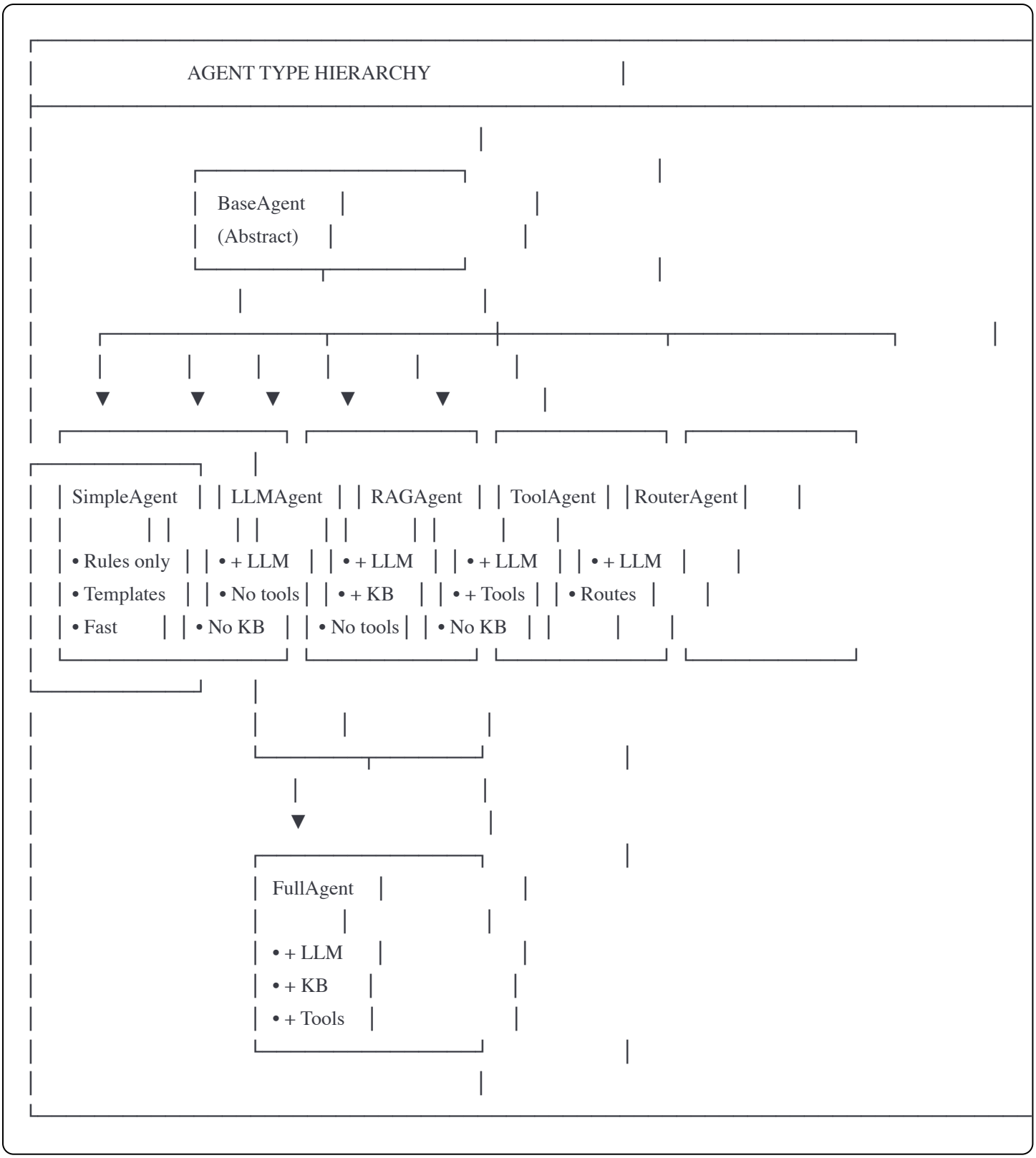
Dynamic Agent Architecture

Core Concept



Agent Type Hierarchy

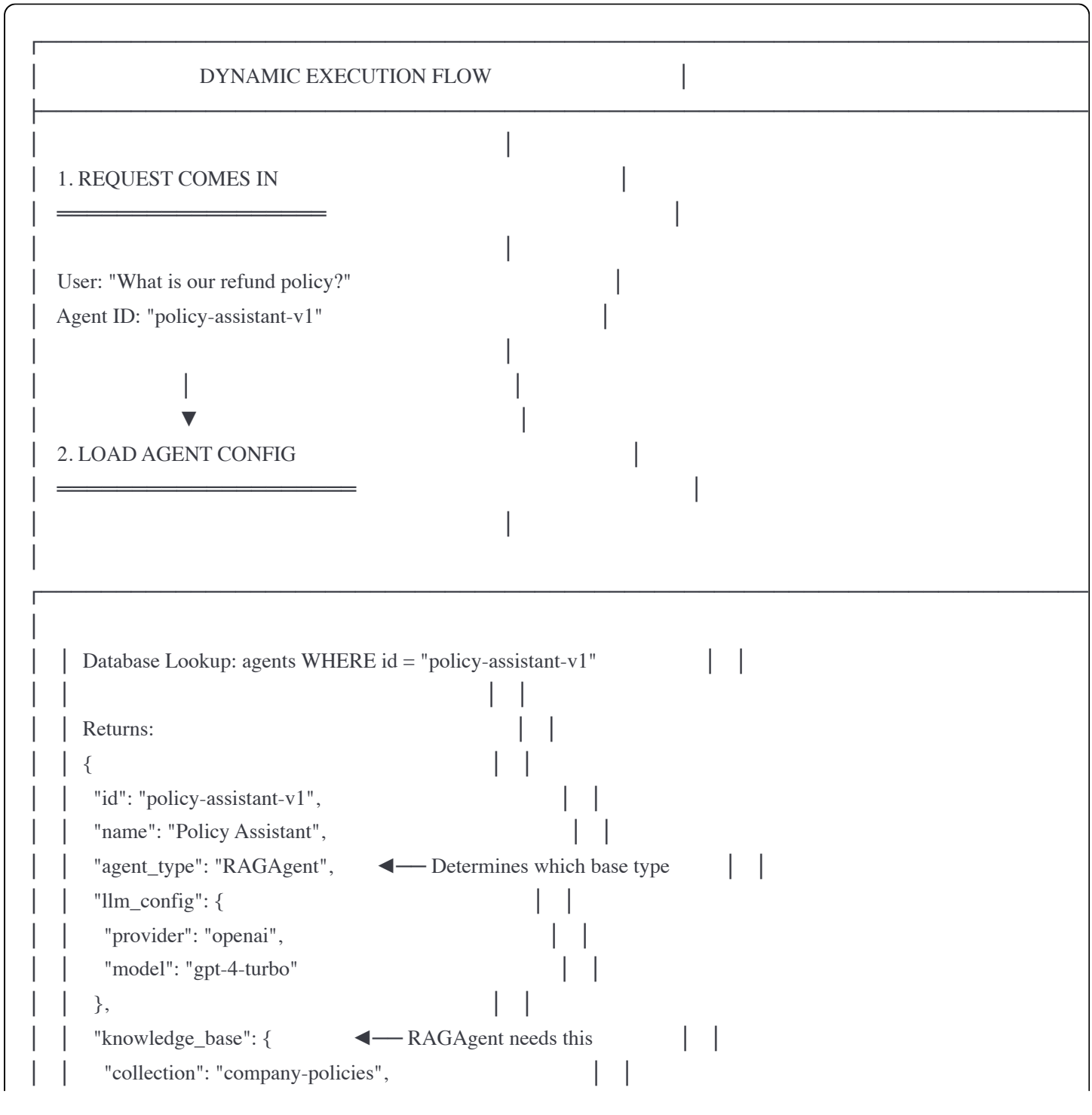


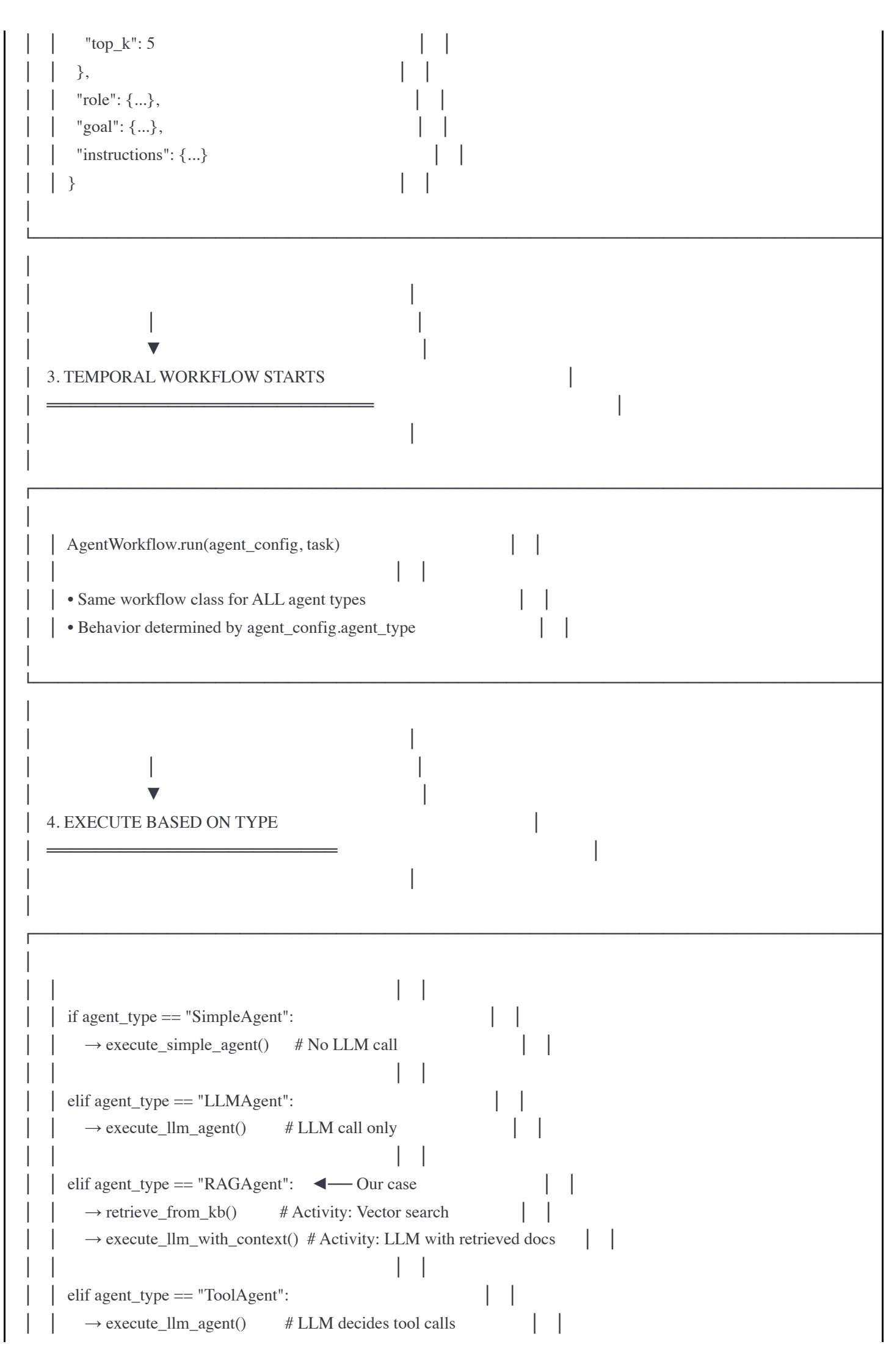
What Each Agent Type DOES vs NEEDS

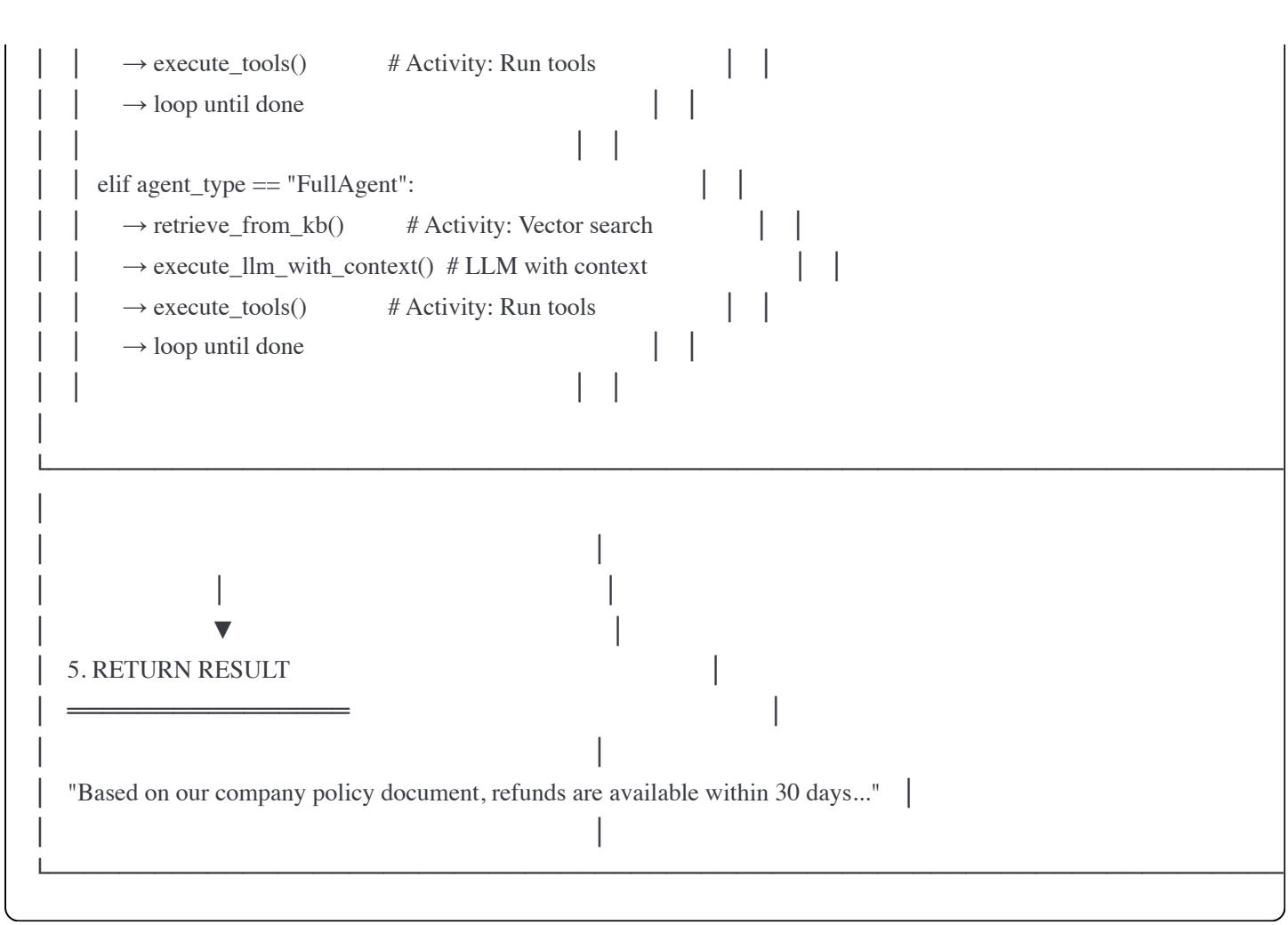
Agent Type	Needs LLM	Needs KB/Vector	Needs Tools	Use Case
SimpleAgent	❌	❌	❌	Greetings, routing by keywords, templates

Agent Type	Needs LLM	Needs KB/Vector	Needs Tools	Use Case
LLMAgent	✅	❌	❌	Text generation, summarization, SQL writing
RAGAgent	✅	✅	❌	Doc Q&A, knowledge lookup, FAQ bots
ToolAgent	✅	❌	✅	Web search, API calls, calculations
FullAgent	✅	✅	✅	Research, complex analysis, multi-step tasks
RouterAgent	✅	❌	❌	Intent classification, agent routing

Execution Flow




"top_k": 5
},
"role": {...},
"goal": {...},
"instructions": {...}
}
3. TEMPORAL WORKFLOW STARTS
AgentWorkflow.run(agent_config, task)
• Same workflow class for ALL agent types
• Behavior determined by agent_config.agent_type
4. EXECUTE BASED ON TYPE
if agent_type == "SimpleAgent":
→ execute_simple_agent() # No LLM call
elif agent_type == "LLMAgent":
→ execute_llm_agent() # LLM call only
elif agent_type == "RAGAgent": ◄── Our case
→ retrieve_from_kb() # Activity: Vector search
→ execute_llm_with_context() # Activity: LLM with retrieved docs
elif agent_type == "ToolAgent":
→ execute_llm_agent() # LLM decides tool calls



Project Structure (Simplified)

```
magure-ai-platform/
│
├── src/
│   │
│   ├── agents/                    # AGENT TYPES (The Code)
│   │   ├── __init__.py
│   │   ├── base.py                # BaseAgent abstract class
│   │   ├── types/
│   │   │   ├── __init__.py
│   │   │   ├── simple_agent.py    # SimpleAgent - no LLM
│   │   │   ├── llm_agent.py       # LLMAgent - LLM only
│   │   │   ├── rag_agent.py       # RAGAgent - LLM + KB
│   │   │   ├── tool_agent.py      # ToolAgent - LLM + Tools
│   │   │   ├── full_agent.py      # FullAgent - LLM + KB + Tools
│   │   │   └── router_agent.py    # RouterAgent - routes to others
│   │   ├── factory.py             # Creates agent from config
│   │   └── registry.py            # Runtime agent registry
│   │
│   ├── workflows/                 # TEMPORAL WORKFLOWS
│   │   ├── __init__.py
```

```
│   │   ├── agent_workflow.py          # ONE workflow for ALL agents
│   │   └── orchestrator_workflow.py   # Multi-agent coordination
│   │
│   ├── activities/                    # TEMPORAL ACTIVITIES
│   │   ├── __init__.py
│   │   ├── llm_activities.py          # LLM API calls
│   │   ├── rag_activities.py          # KB retrieval
│   │   ├── tool_activities.py         # Tool execution
│   │   └── safety_activities.py       # Safety checks
│   │
│   ├── models/                        # DATA MODELS
│   │   ├── __init__.py
│   │   ├── agent_config.py            # Agent configuration schema
│   │   └── task.py                    # Task models
│   │
│   └── storage/                       # PERSISTENCE
│       ├── __init__.py
│       └── agent_repository.py        # CRUD for agent configs
│
└── workers/
    └── main_worker.py                 # Temporal worker
```

Code Implementation

1. Agent Configuration Model (What UI Creates)

python

```python
# src/models/agent_config.py
"""Agent configuration - stored in database, created from UI"""

from enum import Enum
from typing import Any, Dict, List, Optional
from pydantic import BaseModel, Field
from datetime import datetime


class AgentType(str, Enum):
    """Available agent types - maps to base agent classes"""
    SIMPLE = "SimpleAgent"
    LLM = "LLMAgent"
    RAG = "RAGAgent"
    TOOL = "ToolAgent"
    FULL = "FullAgent"
    ROUTER = "RouterAgent"


# ============================================
# LLM CONFIGURATION
# ============================================

class LLMConfig(BaseModel):
    """LLM provider configuration"""
    provider: str = "openai"  # openai, anthropic, azure, ollama
    model: str = "gpt-4-turbo"
    temperature: float = 0.7
    max_tokens: int = 4096

    class Config:
        extra = "allow"  # Allow provider-specific fields


# ============================================
# KNOWLEDGE BASE CONFIGURATION
# ============================================

class KnowledgeBaseConfig(BaseModel):
    """Knowledge base / RAG configuration"""
    enabled: bool = True
    collection_name: str
    embedding_model: str = "text-embedding-3-small"
    top_k: int = 5
    similarity_threshold: float = 0.7
    rerank: bool = False
```

```python
# ============================================
# TOOL CONFIGURATION
# ============================================

class ToolConfig(BaseModel):
    """Tool binding configuration"""
    tool_id: str
    enabled: bool = True
    requires_confirmation: bool = False
    config: Dict[str, Any] = Field(default_factory=dict)


# ============================================
# AGENT PERSONA
# ============================================

class AgentRole(BaseModel):
    """WHO the agent is"""
    title: str
    expertise: List[str] = Field(default_factory=list)
    personality: List[str] = Field(default_factory=list)
    communication_style: str = "professional"


class AgentGoal(BaseModel):
    """WHAT the agent achieves"""
    objective: str
    constraints: List[str] = Field(default_factory=list)
    success_indicators: List[str] = Field(default_factory=list)


class AgentInstructions(BaseModel):
    """HOW the agent operates"""
    steps: List[str] = Field(default_factory=list)
    rules: List[str] = Field(default_factory=list)
    prohibited: List[str] = Field(default_factory=list)
    output_format: Optional[str] = None


class AgentExample(BaseModel):
    """Few-shot example"""
    input: str
    output: str
    explanation: Optional[str] = None
```

```python
# ============================================
# SAFETY & GOVERNANCE
# ============================================

class SafetyConfig(BaseModel):
    """Safety configuration"""
    content_filtering: bool = True
    pii_detection: bool = True
    hallucination_check: bool = True
    max_iterations: int = 10
    timeout_seconds: int = 300


class GovernanceConfig(BaseModel):
    """Governance configuration"""
    audit_logging: bool = True
    require_confirmation_for: List[str] = Field(default_factory=list)
    allowed_data_classifications: List[str] = Field(
        default_factory=lambda: ["public", "internal"]
    )


# ============================================
# COMPLETE AGENT CONFIGURATION
# ============================================

class AgentConfig(BaseModel):
    """
    Complete agent configuration.
    This is what gets stored in the database and created from UI.
    """

    # Identity
    id: str
    name: str
    description: str
    version: str = "1.0.0"

    # Type - determines which base agent class to use
    agent_type: AgentType

    # Persona
    role: AgentRole
    goal: AgentGoal
    instructions: AgentInstructions
    examples: List[AgentExample] = Field(default_factory=list)
```

```python
    # Capabilities (optional based on agent_type)
    llm_config: Optional[LLMConfig] = None
    knowledge_base: Optional[KnowledgeBaseConfig] = None
    tools: List[ToolConfig] = Field(default_factory=list)

    # For RouterAgent
    routing_table: Optional[Dict[str, str]] = None  # intent -> agent_id

    # Safety & Governance
    safety: SafetyConfig = Field(default_factory=SafetyConfig)
    governance: GovernanceConfig = Field(default_factory=GovernanceConfig)

    # Metadata
    created_at: datetime = Field(default_factory=datetime.utcnow)
    updated_at: datetime = Field(default_factory=datetime.utcnow)
    created_by: Optional[str] = None
    is_active: bool = True
    tags: List[str] = Field(default_factory=list)

    def validate_for_type(self) -> List[str]:
        """Validate config has required fields for agent type"""
        errors = []

        if self.agent_type in [AgentType.LLM, AgentType.RAG, AgentType.TOOL, AgentType.FULL]:
            if not self.llm_config:
                errors.append(f"{self.agent_type} requires llm_config")

        if self.agent_type in [AgentType.RAG, AgentType.FULL]:
            if not self.knowledge_base:
                errors.append(f"{self.agent_type} requires knowledge_base")

        if self.agent_type in [AgentType.TOOL, AgentType.FULL]:
            if not self.tools:
                errors.append(f"{self.agent_type} requires at least one tool")

        if self.agent_type == AgentType.ROUTER:
            if not self.routing_table:
                errors.append("RouterAgent requires routing_table")

        return errors
```

2. Base Agent Classes

python

```python
# src/agents/base.py
"""Base agent class - all agent types inherit from this"""

from abc import ABC, abstractmethod
from dataclasses import dataclass
from typing import Any, Dict, List, Optional

from src.models.agent_config import AgentConfig


@dataclass
class AgentContext:
    """Runtime context passed to agent"""
    user_input: str
    session_id: str
    conversation_history: List[Dict[str, str]]
    metadata: Dict[str, Any]


@dataclass
class AgentResponse:
    """Response from agent execution"""
    content: str
    confidence: float = 1.0
    sources: List[str] = None
    tool_calls_made: List[Dict] = None
    needs_confirmation: bool = False
    route_to_agent: Optional[str] = None  # For RouterAgent
    metadata: Dict[str, Any] = None


class BaseAgent(ABC):
    """
    Abstract base agent class.

    All agent types inherit from this and implement execute().
    The config determines behavior, not the class.
    """

    def __init__(self, config: AgentConfig):
        self.config = config
        self.id = config.id
        self.name = config.name

    @abstractmethod
    async def execute(self, context: AgentContext) -> AgentResponse:
```

```
    """
    Execute the agent's logic.

    This is implemented differently by each agent type:
    - SimpleAgent: Template/rule matching
    - LLMAgent: Single LLM call
    - RAGAgent: Retrieve + LLM
    - ToolAgent: LLM + Tool loop
    - FullAgent: Retrieve + LLM + Tool loop
    - RouterAgent: Classify + Route
    """
    pass

def build_system_prompt(self) -> str:
    """Build system prompt from config"""
    parts = []

    # Role
    role = self.config.role
    parts.append(f"## ROLE\nYou are {role.title}.")
    if role.expertise:
        parts.append(f"Your expertise: {', '.join(role.expertise)}.")
    if role.personality:
        parts.append(f"Your personality: {', '.join(role.personality)}.")
    parts.append(f"Communication style: {role.communication_style}.")

    # Goal
    goal = self.config.goal
    parts.append(f"\n## GOAL\n{goal.objective}")
    if goal.constraints:
        parts.append("\nConstraints:")
        for c in goal.constraints:
            parts.append(f"- {c}")

    # Instructions
    instructions = self.config.instructions
    if instructions.steps:
        parts.append("\n## INSTRUCTIONS")
        for i, step in enumerate(instructions.steps, 1):
            parts.append(f"{i}. {step}")

    if instructions.rules:
        parts.append("\n## RULES")
        for rule in instructions.rules:
            parts.append(f"- {rule}")

    if instructions.prohibited:
```

```
        parts.append("\n## PROHIBITED")
        for p in instructions.prohibited:
            parts.append(f"- DO NOT: {p}")

    if instructions.output_format:
        parts.append(f"\n## OUTPUT FORMAT\n{instructions.output_format}")

    # Examples
    if self.config.examples:
        parts.append("\n## EXAMPLES")
        for ex in self.config.examples[:3]:  # Max 3 examples
            parts.append(f"\nInput: {ex.input}")
            parts.append(f"Output: {ex.output}")

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

```
python
```

```python
# src/agents/types/simple_agent.py
"""SimpleAgent - No LLM, rule-based responses"""

import re
from typing import Dict, List
from src.agents.base import BaseAgent, AgentContext, AgentResponse
from src.models.agent_config import AgentConfig


class SimpleAgent(BaseAgent):
    """
    Simple rule-based agent without LLM.

    Use cases:
    - Greeting/welcome messages
    - Simple FAQ with exact matches
    - Keyword-based routing
    - Template responses
    """

    def __init__(self, config: AgentConfig):
        super().__init__(config)
        self._compile_patterns()

    def _compile_patterns(self):
        """Pre-compile regex patterns from examples"""
        self.patterns: List[tuple] = []

        for example in self.config.examples:
            # Convert input to regex pattern
            pattern = re.compile(
                re.escape(example.input).replace(r'\*', '.*'),
                re.IGNORECASE
            )
            self.patterns.append((pattern, example.output))

    async def execute(self, context: AgentContext) -> AgentResponse:
        """Execute simple pattern matching"""
        user_input = context.user_input.lower().strip()

        # Try exact patterns first
        for pattern, response in self.patterns:
            if pattern.search(user_input):
                return AgentResponse(
                    content=response,
                    confidence=1.0,
```

```
                metadata={"match_type": "pattern"}
            )

    # Keyword matching from instructions
    for rule in self.config.instructions.rules:
        if ":" in rule:
            keyword, response = rule.split(":", 1)
            if keyword.strip().lower() in user_input:
                return AgentResponse(
                    content=response.strip(),
                    confidence=0.8,
                    metadata={"match_type": "keyword"}
                )

    # Default response
    default_response = self.config.goal.objective
    return AgentResponse(
        content=f"I can help you with: {default_response}",
        confidence=0.5,
        metadata={"match_type": "default"}
    )
```

python

```python
# src/agents/types/llm_agent.py
"""LLMAgent - Basic LLM-powered agent"""

from src.agents.base import BaseAgent, AgentContext, AgentResponse
from src.models.agent_config import AgentConfig
from src.llm.client import LLMClient, LLMMessage


class LLMAgent(BaseAgent):
    """
    LLM-powered agent without tools or knowledge base.

    Use cases:
    - Text generation
    - Summarization
    - SQL generation
    - Translation
    - Simple Q&A
    """

    def __init__(self, config: AgentConfig):
        super().__init__(config)

        # Initialize LLM client
        llm_config = config.llm_config
        self.llm_client = LLMClient(
            provider=llm_config.provider,
            model=llm_config.model,
        )
        self.temperature = llm_config.temperature
        self.max_tokens = llm_config.max_tokens

    async def execute(self, context: AgentContext) -> AgentResponse:
        """Execute LLM completion"""

        # Build messages
        messages = [
            LLMMessage(role="system", content=self.build_system_prompt())
        ]

        # Add conversation history
        for msg in context.conversation_history[-10:]:  # Last 10 messages
            messages.append(LLMMessage(
                role=msg["role"],
                content=msg["content"]
            ))
```

```python
        # Add current user input
        messages.append(LLMMessage(role="user", content=context.user_input))

        # Call LLM
        response = await self.llm_client.chat_completion(
            messages=messages,
            temperature=self.temperature,
            max_tokens=self.max_tokens,
        )

        return AgentResponse(
            content=response.content,
            confidence=0.9,
            metadata={
                "model": response.model,
                "usage": response.usage,
            }
        )
```

```python
```

```python
# src/agents/types/rag_agent.py
"""RAGAgent - LLM with Knowledge Base retrieval"""

from typing import List
from src.agents.base import BaseAgent, AgentContext, AgentResponse
from src.models.agent_config import AgentConfig
from src.llm.client import LLMClient, LLMMessage
from src.knowledge.retriever import KnowledgeRetriever


class RAGAgent(BaseAgent):
    """
    RAG-powered agent with knowledge base retrieval.

    Use cases:
    - Document Q&A
    - Policy lookup
    - FAQ bots
    - Knowledge search
    """

    def __init__(self, config: AgentConfig):
        super().__init__(config)

        # Initialize LLM client
        llm_config = config.llm_config
        self.llm_client = LLMClient(
            provider=llm_config.provider,
            model=llm_config.model,
        )

        # Initialize knowledge retriever
        kb_config = config.knowledge_base
        self.retriever = KnowledgeRetriever(
            collection_name=kb_config.collection_name,
            embedding_model=kb_config.embedding_model,
            top_k=kb_config.top_k,
            similarity_threshold=kb_config.similarity_threshold,
        )

    async def execute(self, context: AgentContext) -> AgentResponse:
        """Execute RAG: Retrieve then Generate"""

        # Step 1: Retrieve relevant documents
        retrieved_docs = await self.retriever.retrieve(context.user_input)
```

```python
    # Step 2: Build context from retrieved docs
    context_text = self._format_retrieved_docs(retrieved_docs)

    # Step 3: Build prompt with context
    system_prompt = self.build_system_prompt()
    system_prompt += f"\n\n## KNOWLEDGE CONTEXT\n{context_text}"
    system_prompt += "\n\nUse the above context to answer. If the answer is not in the context, say so."

    messages = [
        LLMMessage(role="system", content=system_prompt),
        LLMMessage(role="user", content=context.user_input),
    ]

    # Step 4: Generate response
    response = await self.llm_client.chat_completion(
        messages=messages,
        temperature=self.config.llm_config.temperature,
        max_tokens=self.config.llm_config.max_tokens,
    )

    return AgentResponse(
        content=response.content,
        confidence=0.85,
        sources=[doc.source for doc in retrieved_docs],
        metadata={
            "model": response.model,
            "retrieved_count": len(retrieved_docs),
        }
    )

def _format_retrieved_docs(self, docs: List) -> str:
    """Format retrieved documents for context"""
    parts = []
    for i, doc in enumerate(docs, 1):
        parts.append(f"[Source {i}: {doc.source}]\n{doc.content}\n")
    return "\n".join(parts)
```

```python
```

```python
# src/agents/types/tool_agent.py
"""ToolAgent - LLM with tool/function calling"""

import json
from typing import List, Dict, Any
from src.agents.base import BaseAgent, AgentContext, AgentResponse
from src.models.agent_config import AgentConfig
from src.llm.client import LLMClient, LLMMessage
from src.tools.registry import ToolRegistry
from src.tools.executor import ToolExecutor


class ToolAgent(BaseAgent):
    """
    Tool-using agent with LLM function calling.

    Use cases:
    - Web search
    - API calls
    - Calculations
    - File operations
    - Database queries
    """

    def __init__(self, config: AgentConfig):
        super().__init__(config)

        # Initialize LLM client
        llm_config = config.llm_config
        self.llm_client = LLMClient(
            provider=llm_config.provider,
            model=llm_config.model,
        )

        # Initialize tool executor
        self.tool_executor = ToolExecutor()

        # Load tool definitions
        self.tools = self._load_tools()

    def _load_tools(self) -> List[Dict]:
        """Load tool definitions in OpenAI format"""
        tool_definitions = []

        for tool_config in self.config.tools:
            if tool_config.enabled:
```

```
            tool_def = ToolRegistry.get_definition(tool_config.tool_id)
            if tool_def:
                tool_definitions.append(tool_def.to_openai_format())

        return tool_definitions

    async def execute(self, context: AgentContext) -> AgentResponse:
        """Execute tool-using agent loop"""

        messages = [
            LLMMessage(role="system", content=self.build_system_prompt()),
            LLMMessage(role="user", content=context.user_input),
        ]

        tool_calls_made = []
        max_iterations = self.config.safety.max_iterations

        for iteration in range(max_iterations):
            # Call LLM with tools
            response = await self.llm_client.chat_completion(
                messages=messages,
                temperature=self.config.llm_config.temperature,
                max_tokens=self.config.llm_config.max_tokens,
                tools=self.tools,
            )

            # Check if done (no tool calls)
            if not response.tool_calls:
                return AgentResponse(
                    content=response.content,
                    confidence=0.9,
                    tool_calls_made=tool_calls_made,
                    metadata={
                        "iterations": iteration + 1,
                        "model": response.model,
                    }
                )

            # Execute tool calls
            messages.append(LLMMessage(
                role="assistant",
                content=response.content or "",
                tool_calls=response.tool_calls,
            ))

            for tool_call in response.tool_calls:
                tool_name = tool_call["function"]["name"]
```

```python
            tool_args = json.loads(tool_call["function"]["arguments"])

            # Check if confirmation required
            tool_config = next(
                (t for t in self.config.tools if t.tool_id == tool_name),
                None
            )
            if tool_config and tool_config.requires_confirmation:
                return AgentResponse(
                    content=f"I need to use {tool_name}. Please confirm.",
                    needs_confirmation=True,
                    metadata={"pending_tool_call": tool_call}
                )

            # Execute tool
            result = await self.tool_executor.execute(tool_name, tool_args)
            tool_calls_made.append({
                "tool": tool_name,
                "args": tool_args,
                "result": result,
            })

            # Add result to messages
            messages.append(LLMMessage(
                role="tool",
                content=json.dumps(result),
                tool_call_id=tool_call["id"],
            ))

    # Max iterations reached
    return AgentResponse(
        content="I've reached the maximum number of steps. Here's what I found so far.",
        confidence=0.6,
        tool_calls_made=tool_calls_made,
        metadata={"max_iterations_reached": True}
    )
```

python

```python
# src/agents/types/full_agent.py
"""FullAgent - LLM with both Knowledge Base and Tools"""

import json
from typing import List, Dict
from src.agents.base import BaseAgent, AgentContext, AgentResponse
from src.models.agent_config import AgentConfig
from src.llm.client import LLMClient, LLMMessage
from src.knowledge.retriever import KnowledgeRetriever
from src.tools.registry import ToolRegistry
from src.tools.executor import ToolExecutor


class FullAgent(BaseAgent):
    """
    Full-featured agent with LLM, Knowledge Base, and Tools.

    Use cases:
    - Research assistants
    - Complex analysis
    - Multi-step reasoning
    - Data processing with context
    """

    def __init__(self, config: AgentConfig):
        super().__init__(config)

        # Initialize LLM
        llm_config = config.llm_config
        self.llm_client = LLMClient(
            provider=llm_config.provider,
            model=llm_config.model,
        )

        # Initialize knowledge retriever
        kb_config = config.knowledge_base
        self.retriever = KnowledgeRetriever(
            collection_name=kb_config.collection_name,
            embedding_model=kb_config.embedding_model,
            top_k=kb_config.top_k,
        )

        # Initialize tool executor
        self.tool_executor = ToolExecutor()
        self.tools = self._load_tools()
```

```
def _load_tools(self) -> List[Dict]:
    """Load tool definitions"""
    tool_definitions = []
    for tool_config in self.config.tools:
        if tool_config.enabled:
            tool_def = ToolRegistry.get_definition(tool_config.tool_id)
            if tool_def:
                tool_definitions.append(tool_def.to_openai_format())
    return tool_definitions

async def execute(self, context: AgentContext) -> AgentResponse:
    """Execute full agent: Retrieve + LLM + Tools loop"""

    # Step 1: Retrieve relevant knowledge
    retrieved_docs = await self.retriever.retrieve(context.user_input)
    knowledge_context = self._format_docs(retrieved_docs)

    # Step 2: Build system prompt with knowledge
    system_prompt = self.build_system_prompt()
    system_prompt += f"\n\n## KNOWLEDGE CONTEXT\n{knowledge_context}"

    messages = [
        LLMMessage(role="system", content=system_prompt),
        LLMMessage(role="user", content=context.user_input),
    ]

    tool_calls_made = []
    sources = [doc.source for doc in retrieved_docs]

    # Step 3: Agent loop
    for iteration in range(self.config.safety.max_iterations):
        response = await self.llm_client.chat_completion(
            messages=messages,
            temperature=self.config.llm_config.temperature,
            max_tokens=self.config.llm_config.max_tokens,
            tools=self.tools,
        )

        # Done if no tool calls
        if not response.tool_calls:
            return AgentResponse(
                content=response.content,
                confidence=0.9,
                sources=sources,
                tool_calls_made=tool_calls_made,
                metadata={"iterations": iteration + 1}
            )
```

```python
        # Process tool calls
        messages.append(LLMMessage(
            role="assistant",
            content=response.content or "",
            tool_calls=response.tool_calls,
        ))

        for tool_call in response.tool_calls:
            tool_name = tool_call["function"]["name"]
            tool_args = json.loads(tool_call["function"]["arguments"])

            result = await self.tool_executor.execute(tool_name, tool_args)
            tool_calls_made.append({
                "tool": tool_name,
                "args": tool_args,
                "result": result,
            })

            messages.append(LLMMessage(
                role="tool",
                content=json.dumps(result),
                tool_call_id=tool_call["id"],
            ))

    return AgentResponse(
        content="Reached maximum iterations.",
        confidence=0.6,
        sources=sources,
        tool_calls_made=tool_calls_made,
    )

def _format_docs(self, docs: List) -> str:
    """Format retrieved documents"""
    return "\n\n".join([
        f"[{doc.source}]\n{doc.content}"
        for doc in docs
    ])
```

python

```python
# src/agents/types/router_agent.py
"""RouterAgent - Routes requests to appropriate agents"""

from src.agents.base import BaseAgent, AgentContext, AgentResponse
from src.models.agent_config import AgentConfig
from src.llm.client import LLMClient, LLMMessage


class RouterAgent(BaseAgent):
    """
    Router agent that classifies intent and routes to other agents.

    Use cases:
    - Intent classification
    - Department routing
    - Multi-agent orchestration
    """

    def __init__(self, config: AgentConfig):
        super().__init__(config)

        # Initialize LLM for classification
        llm_config = config.llm_config
        self.llm_client = LLMClient(
            provider=llm_config.provider,
            model=llm_config.model,
        )

        # Routing table: intent -> agent_id
        self.routing_table = config.routing_table or {}

    async def execute(self, context: AgentContext) -> AgentResponse:
        """Classify intent and route to appropriate agent"""

        # Build classification prompt
        intents = list(self.routing_table.keys())
        intent_list = "\n".join([f"- {intent}" for intent in intents])

        classification_prompt = f"""Classify the user's intent into ONE of these categories:
{intent_list}

Respond with ONLY the intent name, nothing else.

User message: {context.user_input}

Intent:"""
```

```
        messages = [
            LLMMessage(role="system", content="You are an intent classifier. Respond with only the intent name."),
            LLMMessage(role="user", content=classification_prompt),
        ]

        response = await self.llm_client.chat_completion(
            messages=messages,
            temperature=0.1,  # Low temp for classification
            max_tokens=50,
        )

        # Extract intent
        detected_intent = response.content.strip().lower()

        # Find matching agent
        target_agent = None
        for intent, agent_id in self.routing_table.items():
            if intent.lower() in detected_intent or detected_intent in intent.lower():
                target_agent = agent_id
                break

        if target_agent:
            return AgentResponse(
                content=f"Routing to {target_agent}",
                confidence=0.9,
                route_to_agent=target_agent,
                metadata={
                    "detected_intent": detected_intent,
                    "routed_to": target_agent,
                }
            )

        # No match - return default or error
        return AgentResponse(
            content="I'm not sure how to help with that. Could you rephrase?",
            confidence=0.5,
            metadata={
                "detected_intent": detected_intent,
                "no_route_found": True,
            }
        )
```

3. Agent Factory

```
python
```

```python
# src/agents/factory.py
"""Factory for creating agents from configuration"""

from typing import Dict, Type
from src.agents.base import BaseAgent
from src.models.agent_config import AgentConfig, AgentType

# Import all agent types
from src.agents.types.simple_agent import SimpleAgent
from src.agents.types.llm_agent import LLMAgent
from src.agents.types.rag_agent import RAGAgent
from src.agents.types.tool_agent import ToolAgent
from src.agents.types.full_agent import FullAgent
from src.agents.types.router_agent import RouterAgent


# Agent type to class mapping
AGENT_TYPE_MAP: Dict[AgentType, Type[BaseAgent]] = {
    AgentType.SIMPLE: SimpleAgent,
    AgentType.LLM: LLMAgent,
    AgentType.RAG: RAGAgent,
    AgentType.TOOL: ToolAgent,
    AgentType.FULL: FullAgent,
    AgentType.ROUTER: RouterAgent,
}


class AgentFactory:
    """
    Factory for creating agent instances from configuration.

    This is the key component that enables dynamic agent creation.
    The UI creates AgentConfig, this factory creates the actual agent.
    """

    @staticmethod
    def create(config: AgentConfig) -> BaseAgent:
        """
        Create an agent instance from configuration.

        Args:
            config: AgentConfig loaded from database

        Returns:
            Instantiated agent of the appropriate type
```

```
        Raises:
            ValueError: If agent type is invalid or config validation fails
        """
        # Validate config for the agent type
        errors = config.validate_for_type()
        if errors:
            raise ValueError(f"Invalid config for {config.agent_type}: {errors}")

        # Get the agent class
        agent_class = AGENT_TYPE_MAP.get(config.agent_type)
        if not agent_class:
            raise ValueError(f"Unknown agent type: {config.agent_type}")

        # Create and return the agent
        return agent_class(config)

    @staticmethod
    def get_available_types() -> Dict[str, str]:
        """Get available agent types and descriptions"""
        return {
            AgentType.SIMPLE.value: "Rule-based agent without LLM",
            AgentType.LLM.value: "LLM-powered agent for text generation",
            AgentType.RAG.value: "LLM with knowledge base retrieval",
            AgentType.TOOL.value: "LLM with tool/function calling",
            AgentType.FULL.value: "LLM with both knowledge base and tools",
            AgentType.ROUTER.value: "Routes requests to other agents",
        }
```

4. Temporal Workflow (Single Workflow for All Agents)

python

```python
# src/workflows/agent_workflow.py
"""
Single Temporal workflow that handles ALL agent types.

The workflow is generic - agent behavior is determined by configuration.
"""

from dataclasses import dataclass
from datetime import timedelta
from typing import Any, Dict, List, Optional

from temporalio import workflow
from temporalio.common import RetryPolicy

# Import activities (will be implemented as activities)
with workflow.unsafe.imports_passed_through():
    from src.models.agent_config import AgentConfig, AgentType
    from src.models.task import Task, TaskResult


@dataclass
class AgentWorkflowInput:
    """Input to the agent workflow"""
    agent_config_dict: Dict[str, Any]  # Serialized AgentConfig
    task_description: str
    session_id: str
    conversation_history: List[Dict[str, str]]
    context: Dict[str, Any]


@dataclass
class AgentWorkflowOutput:
    """Output from the agent workflow"""
    success: bool
    content: str
    confidence: float
    sources: Optional[List[str]] = None
    tool_calls: Optional[List[Dict]] = None
    route_to_agent: Optional[str] = None
    iterations: int = 1
    error: Optional[str] = None
    metadata: Dict[str, Any] = None


# Activity signatures (implemented in activities/)
@workflow.defn
```

```python
class AgentWorkflow:
    """
    Universal agent workflow.

    This single workflow handles all agent types:
    - SimpleAgent: Direct response, no activities
    - LLMAgent: llm_completion activity
    - RAGAgent: retrieve_knowledge + llm_completion activities
    - ToolAgent: llm_completion + execute_tool activities (loop)
    - FullAgent: retrieve_knowledge + llm_completion + execute_tool (loop)
    - RouterAgent: llm_completion (classification) activity

    The workflow determines which activities to call based on agent_type.
    """

    def __init__(self):
        self.status = "initialized"
        self.current_iteration = 0
        self.pending_confirmation = None

    @workflow.run
    async def run(self, input: AgentWorkflowInput) -> AgentWorkflowOutput:
        """Main workflow execution"""

        workflow.logger.info(f"Starting agent workflow for session {input.session_id}")

        # Parse agent config
        agent_type = AgentType(input.agent_config_dict["agent_type"])
        agent_config = input.agent_config_dict

        self.status = "running"

        try:
            # Route to appropriate execution method based on type
            if agent_type == AgentType.SIMPLE:
                return await self._execute_simple_agent(agent_config, input)

            elif agent_type == AgentType.LLM:
                return await self._execute_llm_agent(agent_config, input)

            elif agent_type == AgentType.RAG:
                return await self._execute_rag_agent(agent_config, input)

            elif agent_type == AgentType.TOOL:
                return await self._execute_tool_agent(agent_config, input)

            elif agent_type == AgentType.FULL:
```

```
            return await self._execute_full_agent(agent_config, input)

        elif agent_type == AgentType.ROUTER:
            return await self._execute_router_agent(agent_config, input)

        else:
            raise ValueError(f"Unknown agent type: {agent_type}")

    except Exception as e:
        workflow.logger.error(f"Agent workflow failed: {e}")
        self.status = "failed"
        return AgentWorkflowOutput(
            success=False,
            content="",
            confidence=0.0,
            error=str(e),
        )

# ============================================
# SIMPLE AGENT (No Activities)
# ============================================

async def _execute_simple_agent(
    self,
    config: Dict,
    input: AgentWorkflowInput
) -> AgentWorkflowOutput:
    """Execute simple rule-based agent"""

    # Simple agents use pattern matching - run as local activity
    result = await workflow.execute_activity(
        "execute_simple_agent",
        args=[config, input.task_description],
        start_to_close_timeout=timedelta(seconds=10),
    )

    return AgentWorkflowOutput(
        success=True,
        content=result["content"],
        confidence=result["confidence"],
        iterations=1,
        metadata=result.get("metadata", {}),
    )

# ============================================
# LLM AGENT
# ============================================
```

```python
async def _execute_llm_agent(
    self,
    config: Dict,
    input: AgentWorkflowInput
) -> AgentWorkflowOutput:
    """Execute LLM-only agent"""

    # Pre-execution safety check
    safety_result = await workflow.execute_activity(
        "safety_pre_check",
        args=[input.task_description, config.get("safety", {})],
        start_to_close_timeout=timedelta(seconds=30),
        retry_policy=RetryPolicy(maximum_attempts=2),
    )

    if not safety_result["passed"]:
        return AgentWorkflowOutput(
            success=False,
            content=safety_result["message"],
            confidence=0.0,
            error="Safety check failed",
        )

    # Execute LLM completion
    llm_result = await workflow.execute_activity(
        "llm_completion",
        args=[{
            "system_prompt": self._build_system_prompt(config),
            "user_message": input.task_description,
            "conversation_history": input.conversation_history,
            "llm_config": config["llm_config"],
        }],
        start_to_close_timeout=timedelta(seconds=120),
        retry_policy=RetryPolicy(
            maximum_attempts=3,
            initial_interval=timedelta(seconds=5),
        ),
    )

    # Post-execution safety check (hallucination, etc.)
    post_safety = await workflow.execute_activity(
        "safety_post_check",
        args=[llm_result["content"], config.get("safety", {})],
        start_to_close_timeout=timedelta(seconds=30),
    )
```

```
    return AgentWorkflowOutput(
        success=True,
        content=llm_result["content"],
        confidence=post_safety.get("confidence", 0.9),
        iterations=1,
        metadata={
            "model": llm_result.get("model"),
            "usage": llm_result.get("usage"),
            "safety_score": post_safety.get("score"),
        },
    )

# ============================================
# RAG AGENT
# ============================================

async def _execute_rag_agent(
    self,
    config: Dict,
    input: AgentWorkflowInput
) -> AgentWorkflowOutput:
    """Execute RAG agent: Retrieve + Generate"""

    # Step 1: Retrieve from knowledge base
    kb_config = config["knowledge_base"]
    retrieve_result = await workflow.execute_activity(
        "retrieve_knowledge",
        args=[{
            "query": input.task_description,
            "collection": kb_config["collection_name"],
            "top_k": kb_config.get("top_k", 5),
            "threshold": kb_config.get("similarity_threshold", 0.7),
        }],
        start_to_close_timeout=timedelta(seconds=30),
        retry_policy=RetryPolicy(maximum_attempts=2),
    )

    # Step 2: Build augmented prompt
    context_text = self._format_retrieved_docs(retrieve_result["documents"])
    system_prompt = self._build_system_prompt(config)
    system_prompt += f"\n\n## KNOWLEDGE CONTEXT\n{context_text}"
    system_prompt += "\n\nAnswer based on the context. If not found, say so."

    # Step 3: Generate response with LLM
    llm_result = await workflow.execute_activity(
        "llm_completion",
        args=[{
```

```
            "system_prompt": system_prompt,
            "user_message": input.task_description,
            "conversation_history": input.conversation_history,
            "llm_config": config["llm_config"],
        }],
        start_to_close_timeout=timedelta(seconds=120),
        retry_policy=RetryPolicy(maximum_attempts=3),
    )

    # Step 4: Hallucination check (verify grounding)
    grounding_result = await workflow.execute_activity(
        "verify_grounding",
        args=[{
            "response": llm_result["content"],
            "sources": retrieve_result["documents"],
        }],
        start_to_close_timeout=timedelta(seconds=30),
    )

    return AgentWorkflowOutput(
        success=True,
        content=llm_result["content"],
        confidence=grounding_result.get("confidence", 0.85),
        sources=[doc["source"] for doc in retrieve_result["documents"]],
        iterations=1,
        metadata={
            "retrieved_count": len(retrieve_result["documents"]),
            "grounding_score": grounding_result.get("score"),
        },
    )

# ============================================
# TOOL AGENT
# ============================================

async def _execute_tool_agent(
    self,
    config: Dict,
    input: AgentWorkflowInput
) -> AgentWorkflowOutput:
    """Execute tool-using agent with loop"""

    messages = [
        {"role": "system", "content": self._build_system_prompt(config)},
        {"role": "user", "content": input.task_description},
    ]
```

```
    # Load tool definitions
    tools = await workflow.execute_activity(
        "load_tool_definitions",
        args=[[t["tool_id"] for t in config.get("tools", [])]],
        start_to_close_timeout=timedelta(seconds=10),
    )

    tool_calls_made = []
    max_iterations = config.get("safety", {}).get("max_iterations", 10)

    for iteration in range(max_iterations):
        self.current_iteration = iteration + 1

        # Call LLM with tools
        llm_result = await workflow.execute_activity(
            "llm_completion_with_tools",
            args=[{
                "messages": messages,
                "tools": tools,
                "llm_config": config["llm_config"],
            }],
            start_to_close_timeout=timedelta(seconds=120),
        )

        # Check if done
        if not llm_result.get("tool_calls"):
            return AgentWorkflowOutput(
                success=True,
                content=llm_result["content"],
                confidence=0.9,
                tool_calls=tool_calls_made,
                iterations=iteration + 1,
            )

        # Execute tool calls
        messages.append({
            "role": "assistant",
            "content": llm_result.get("content", ""),
            "tool_calls": llm_result["tool_calls"],
        })

        for tool_call in llm_result["tool_calls"]:
            # Check if confirmation required
            tool_config = next(
                (t for t in config.get("tools", [])
                 if t["tool_id"] == tool_call["function"]["name"]),
                None
```

```
        )

        if tool_config and tool_config.get("requires_confirmation"):
            # Wait for confirmation signal
            self.pending_confirmation = tool_call
            self.status = "waiting_confirmation"

            confirmed = await workflow.wait_condition(
                lambda: self.pending_confirmation is None,
                timeout=timedelta(hours=1),
            )

            if not confirmed:
                return AgentWorkflowOutput(
                    success=False,
                    content="Tool execution not confirmed",
                    confidence=0.0,
                    error="Confirmation timeout",
                )

        # Execute tool
        tool_result = await workflow.execute_activity(
            "execute_tool",
            args=[{
                "tool_name": tool_call["function"]["name"],
                "arguments": tool_call["function"]["arguments"],
            }],
            start_to_close_timeout=timedelta(seconds=60),
            retry_policy=RetryPolicy(maximum_attempts=2),
        )

        tool_calls_made.append({
            "tool": tool_call["function"]["name"],
            "result": tool_result,
        })

        messages.append({
            "role": "tool",
            "tool_call_id": tool_call["id"],
            "content": str(tool_result),
        })

    # Max iterations reached
    return AgentWorkflowOutput(
        success=True,
        content="Maximum iterations reached.",
        confidence=0.6,
```

```
            tool_calls=tool_calls_made,
            iterations=max_iterations,
            metadata={"max_iterations_reached": True},
        )

    # ============================================
    # FULL AGENT (RAG + Tools)
    # ============================================

    async def _execute_full_agent(
        self,
        config: Dict,
        input: AgentWorkflowInput
    ) -> AgentWorkflowOutput:
        """Execute full agent: RAG + LLM + Tools"""

        # Step 1: Retrieve knowledge
        kb_config = config["knowledge_base"]
        retrieve_result = await workflow.execute_activity(
            "retrieve_knowledge",
            args=[{
                "query": input.task_description,
                "collection": kb_config["collection_name"],
                "top_k": kb_config.get("top_k", 5),
            }],
            start_to_close_timeout=timedelta(seconds=30),
        )

        # Build system prompt with knowledge
        context_text = self._format_retrieved_docs(retrieve_result["documents"])
        system_prompt = self._build_system_prompt(config)
        system_prompt += f"\n\n## KNOWLEDGE CONTEXT\n{context_text}"

        # Load tools
        tools = await workflow.execute_activity(
            "load_tool_definitions",
            args=[[t["tool_id"] for t in config.get("tools", [])]],
            start_to_close_timeout=timedelta(seconds=10),
        )

        messages = [
            {"role": "system", "content": system_prompt},
            {"role": "user", "content": input.task_description},
        ]

        tool_calls_made = []
        sources = [doc["source"] for doc in retrieve_result["documents"]]
```

```
    max_iterations = config.get("safety", {}).get("max_iterations", 10)

    # Agent loop
    for iteration in range(max_iterations):
        self.current_iteration = iteration + 1

        llm_result = await workflow.execute_activity(
            "llm_completion_with_tools",
            args=[{
                "messages": messages,
                "tools": tools,
                "llm_config": config["llm_config"],
            }],
            start_to_close_timeout=timedelta(seconds=120),
        )

        if not llm_result.get("tool_calls"):
            return AgentWorkflowOutput(
                success=True,
                content=llm_result["content"],
                confidence=0.9,
                sources=sources,
                tool_calls=tool_calls_made,
                iterations=iteration + 1,
            )

        # Process tool calls (same as tool agent)
        messages.append({
            "role": "assistant",
            "content": llm_result.get("content", ""),
            "tool_calls": llm_result["tool_calls"],
        })

        for tool_call in llm_result["tool_calls"]:
            tool_result = await workflow.execute_activity(
                "execute_tool",
                args=[{
                    "tool_name": tool_call["function"]["name"],
                    "arguments": tool_call["function"]["arguments"],
                }],
                start_to_close_timeout=timedelta(seconds=60),
            )

            tool_calls_made.append({
                "tool": tool_call["function"]["name"],
                "result": tool_result,
            })
```

```
                messages.append({
                    "role": "tool",
                    "tool_call_id": tool_call["id"],
                    "content": str(tool_result),
                })

        return AgentWorkflowOutput(
            success=True,
            content="Maximum iterations reached.",
            confidence=0.6,
            sources=sources,
            tool_calls=tool_calls_made,
            iterations=max_iterations,
        )

    # ============================================
    # ROUTER AGENT
    # ============================================

    async def _execute_router_agent(
        self,
        config: Dict,
        input: AgentWorkflowInput
    ) -> AgentWorkflowOutput:
        """Execute router agent for intent classification"""

        routing_table = config.get("routing_table", {})
        intents = list(routing_table.keys())

        classification_prompt = f"""Classify the user's intent into ONE of these categories:
{chr(10).join(f'- {intent}' for intent in intents)}

Respond with ONLY the intent name.

User message: {input.task_description}

Intent:"""

        llm_result = await workflow.execute_activity(
            "llm_completion",
            args=[{
                "system_prompt": "You are an intent classifier. Respond with only the intent name.",
                "user_message": classification_prompt,
                "llm_config": {
                    **config["llm_config"],
                    "temperature": 0.1,  # Low temp for classification
```

```
            "max_tokens": 50,
        },
    }],
    start_to_close_timeout=timedelta(seconds=30),
)

detected_intent = llm_result["content"].strip().lower()

# Find matching agent
target_agent = None
for intent, agent_id in routing_table.items():
    if intent.lower() in detected_intent:
        target_agent = agent_id
        break

return AgentWorkflowOutput(
    success=True,
    content=f"Routing to: {target_agent}" if target_agent else "No route found",
    confidence=0.9 if target_agent else 0.5,
    route_to_agent=target_agent,
    iterations=1,
    metadata={"detected_intent": detected_intent},
)

# ============================================
# SIGNALS & QUERIES
# ============================================

@workflow.signal
async def confirm_tool_execution(self, confirmed: bool):
    """Signal to confirm/deny pending tool execution"""
    if confirmed:
        self.pending_confirmation = None
        self.status = "running"
    else:
        self.pending_confirmation = "denied"

@workflow.query
def get_status(self) -> Dict[str, Any]:
    """Query current workflow status"""
    return {
        "status": self.status,
        "iteration": self.current_iteration,
        "pending_confirmation": self.pending_confirmation is not None,
    }

# ============================================
```

```
    # HELPERS
    # ============================================

    def _build_system_prompt(self, config: Dict) -> str:
        """Build system prompt from config"""
        parts = []

        # Role
        role = config.get("role", {})
        if role.get("title"):
            parts.append(f"## ROLE\nYou are {role['title']}.")
        if role.get("expertise"):
            parts.append(f"Expertise: {', '.join(role['expertise'])}.")

        # Goal
        goal = config.get("goal", {})
        if goal.get("objective"):
            parts.append(f"\n## GOAL\n{goal['objective']}")
        if goal.get("constraints"):
            parts.append("Constraints:\n" + "\n".join(f"- {c}" for c in goal["constraints"]))

        # Instructions
        instructions = config.get("instructions", {})
        if instructions.get("steps"):
            parts.append("\n## INSTRUCTIONS")
            for i, step in enumerate(instructions["steps"], 1):
                parts.append(f"{i}. {step}")

        if instructions.get("prohibited"):
            parts.append("\n## PROHIBITED")
            for p in instructions["prohibited"]:
                parts.append(f"- DO NOT: {p}")

        # Examples
        examples = config.get("examples", [])
        if examples:
            parts.append("\n## EXAMPLES")
            for ex in examples[:3]:
                parts.append(f"\nInput: {ex['input']}\nOutput: {ex['output']}")

        return "\n".join(parts)

    def _format_retrieved_docs(self, docs: List[Dict]) -> str:
        """Format retrieved documents"""
        return "\n\n".join([
            f"[Source: {doc.get('source', 'Unknown')}]\n{doc.get('content', '')}"
```

```
        for doc in docs
    ])
```

5. Activities

python

```python
# src/activities/__init__.py
"""Temporal activities for agent execution"""

from temporalio import activity
from typing import Any, Dict, List
import json

from src.llm.client import LLMClient, LLMMessage
from src.knowledge.retriever import KnowledgeRetriever
from src.tools.executor import ToolExecutor
from src.tools.registry import ToolRegistry
from src.safety.content_filter import ContentFilter
from src.safety.hallucination_manager import HallucinationChecker


# ============================================
# SIMPLE AGENT ACTIVITY
# ============================================

@activity.defn
async def execute_simple_agent(config: Dict, user_input: str) -> Dict[str, Any]:
    """Execute simple pattern-matching agent"""
    import re

    user_input_lower = user_input.lower()

    # Try examples as patterns
    for example in config.get("examples", []):
        pattern = re.compile(re.escape(example["input"]).replace(r'\*', '.*'), re.IGNORECASE)
        if pattern.search(user_input_lower):
            return {
                "content": example["output"],
                "confidence": 1.0,
                "metadata": {"match_type": "pattern"}
            }

    # Default response
    goal = config.get("goal", {}).get("objective", "assist you")
    return {
        "content": f"I can help you with: {goal}",
        "confidence": 0.5,
        "metadata": {"match_type": "default"}
    }


# ============================================
```

```python
# LLM ACTIVITIES
# ==========================================

@activity.defn
async def llm_completion(params: Dict[str, Any]) -> Dict[str, Any]:
    """Execute LLM completion"""

    llm_config = params["llm_config"]
    client = LLMClient(
        provider=llm_config.get("provider", "openai"),
        model=llm_config.get("model", "gpt-4-turbo"),
    )

    messages = [
        LLMMessage(role="system", content=params["system_prompt"]),
    ]

    # Add conversation history
    for msg in params.get("conversation_history", [])[-10:]:
        messages.append(LLMMessage(role=msg["role"], content=msg["content"]))

    # Add user message
    messages.append(LLMMessage(role="user", content=params["user_message"]))

    response = await client.chat_completion(
        messages=messages,
        temperature=llm_config.get("temperature", 0.7),
        max_tokens=llm_config.get("max_tokens", 4096),
    )

    return {
        "content": response.content,
        "model": response.model,
        "usage": response.usage,
    }


@activity.defn
async def llm_completion_with_tools(params: Dict[str, Any]) -> Dict[str, Any]:
    """Execute LLM completion with tool calling"""

    llm_config = params["llm_config"]
    client = LLMClient(
        provider=llm_config.get("provider", "openai"),
        model=llm_config.get("model", "gpt-4-turbo"),
    )
```

```
    # Convert messages
    messages = []
    for msg in params["messages"]:
        llm_msg = LLMMessage(
            role=msg["role"],
            content=msg.get("content", ""),
        )
        if "tool_calls" in msg:
            llm_msg.tool_calls = msg["tool_calls"]
        if "tool_call_id" in msg:
            llm_msg.tool_call_id = msg["tool_call_id"]
        messages.append(llm_msg)

    response = await client.chat_completion(
        messages=messages,
        temperature=llm_config.get("temperature", 0.7),
        max_tokens=llm_config.get("max_tokens", 4096),
        tools=params.get("tools", []),
    )

    return {
        "content": response.content,
        "tool_calls": response.tool_calls,
        "model": response.model,
        "usage": response.usage,
    }


# ============================================
# KNOWLEDGE BASE ACTIVITIES
# ============================================

@activity.defn
async def retrieve_knowledge(params: Dict[str, Any]) -> Dict[str, Any]:
    """Retrieve documents from knowledge base"""

    retriever = KnowledgeRetriever(
        collection_name=params["collection"],
        top_k=params.get("top_k", 5),
        similarity_threshold=params.get("threshold", 0.7),
    )

    documents = await retriever.retrieve(params["query"])

    return {
        "documents": [
            {
```

```
            "content": doc.content,
            "source": doc.source,
            "score": doc.score,
        }
        for doc in documents
    ]
}


# ================================================
# TOOL ACTIVITIES
# ================================================

@activity.defn
async def load_tool_definitions(tool_ids: List[str]) -> List[Dict]:
    """Load tool definitions in OpenAI format"""

    definitions = []
    for tool_id in tool_ids:
        tool_def = ToolRegistry.get_definition(tool_id)
        if tool_def:
            definitions.append(tool_def.to_openai_format())

    return definitions


@activity.defn
async def execute_tool(params: Dict[str, Any]) -> Any:
    """Execute a tool"""

    executor = ToolExecutor()

    tool_name = params["tool_name"]
    arguments = params["arguments"]

    if isinstance(arguments, str):
        arguments = json.loads(arguments)

    result = await executor.execute(tool_name, arguments)

    return result


# ================================================
# SAFETY ACTIVITIES
# ================================================
```

```python
@activity.defn
async def safety_pre_check(content: str, safety_config: Dict) -> Dict[str, Any]:
    """Pre-execution safety check"""

    filter = ContentFilter()

    result = await filter.check_input(
        content=content,
        check_toxicity=safety_config.get("content_filtering", True),
        check_pii=safety_config.get("pii_detection", True),
        check_injection=True,
    )

    return {
        "passed": result.passed,
        "message": result.message,
        "flags": result.flags,
    }


@activity.defn
async def safety_post_check(content: str, safety_config: Dict) -> Dict[str, Any]:
    """Post-execution safety check"""

    filter = ContentFilter()

    result = await filter.check_output(
        content=content,
        check_toxicity=safety_config.get("content_filtering", True),
        check_pii=safety_config.get("pii_detection", True),
    )

    return {
        "passed": result.passed,
        "confidence": result.confidence,
        "score": result.safety_score,
    }


@activity.defn
async def verify_grounding(params: Dict[str, Any]) -> Dict[str, Any]:
    """Verify response is grounded in sources"""

    checker = HallucinationChecker()

    result = await checker.verify_grounding(
        response=params["response"],
```

```
    sources=params["sources"],
)

return {
    "grounded": result.is_grounded,
    "confidence": result.confidence,
    "score": result.grounding_score,
    "ungrounded_claims": result.ungrounded_claims,
}
```

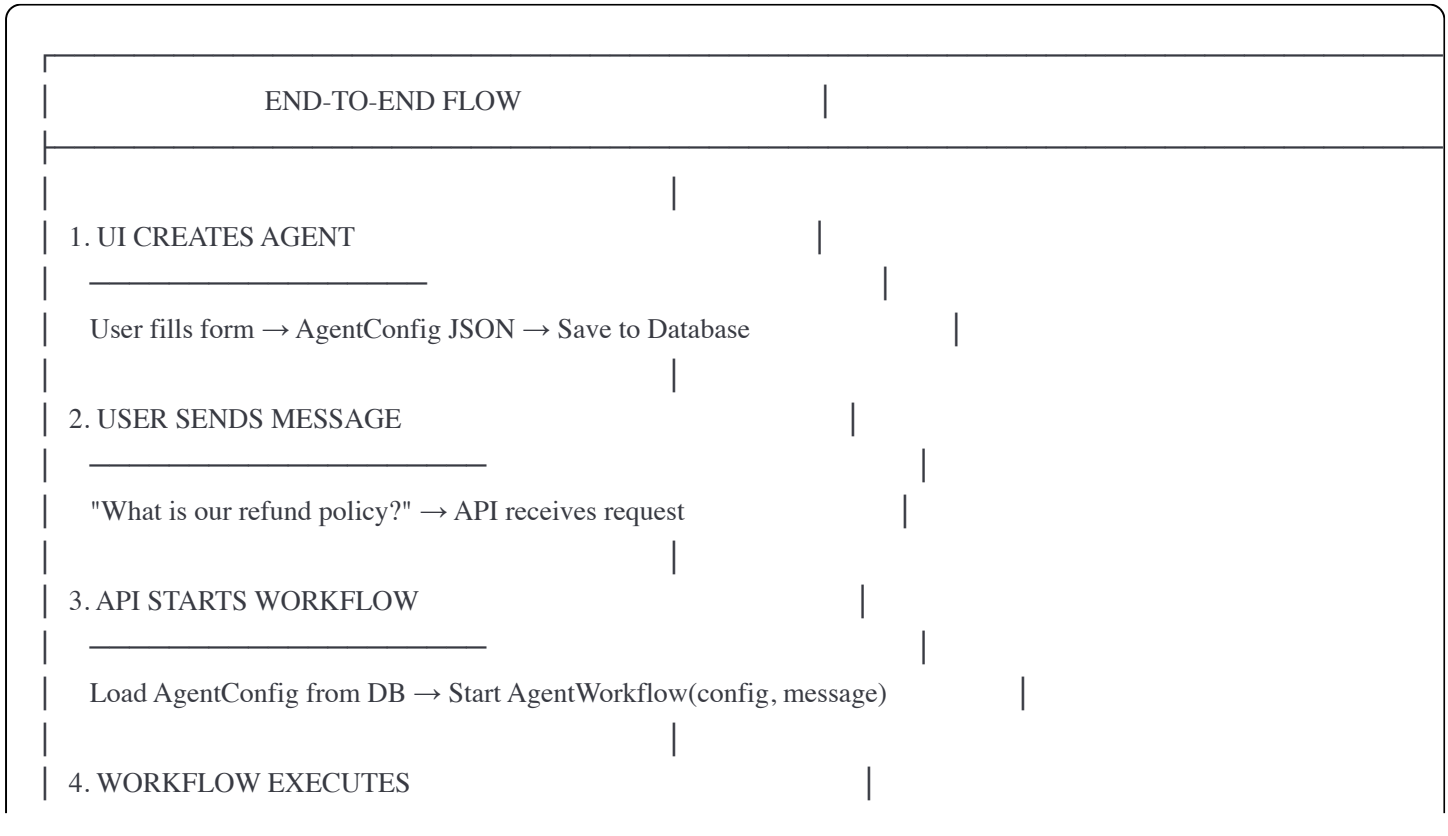
6. Worker Setup

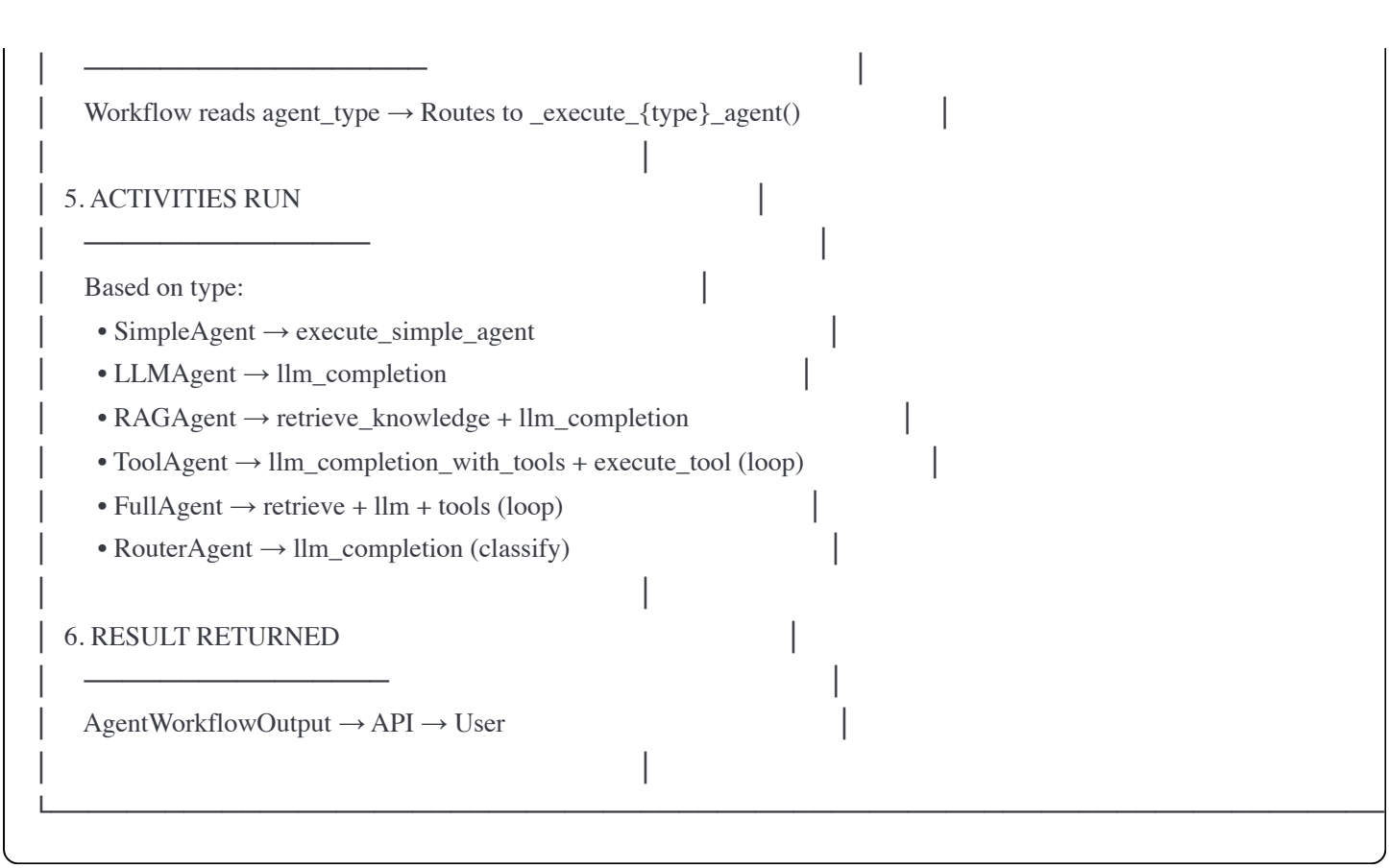
```python
```

```python
# workers/main_worker.py
"""Main Temporal worker"""

import asyncio
import structlog
from temporalio.client import Client
from temporalio.worker import Worker

from src.config.settings import get_settings
from src.workflows.agent_workflow import AgentWorkflow
from src.activities import (
    execute_simple_agent,
    llm_completion,
    llm_completion_with_tools,
    retrieve_knowledge,
    load_tool_definitions,
    execute_tool,
    safety_pre_check,
    safety_post_check,
    verify_grounding,
)

logger = structlog.get_logger()
settings = get_settings()


async def main():
    """Run the Temporal worker"""

    logger.info("Starting Temporal worker",
                host=settings.temporal_host,
                queue=settings.temporal_task_queue)

    # Connect to Temporal
    client = await Client.connect(
        settings.temporal_host,
        namespace=settings.temporal_namespace,
    )

    # Create worker with workflows and activities
    worker = Worker(
        client,
        task_queue=settings.temporal_task_queue,
        workflows=[AgentWorkflow],
        activities=[
            # Simple agent
```

```
            execute_simple_agent,
            # LLM
            llm_completion,
            llm_completion_with_tools,
            # Knowledge
            retrieve_knowledge,
            # Tools
            load_tool_definitions,
            execute_tool,
            # Safety
            safety_pre_check,
            safety_post_check,
            verify_grounding,
        ],
    )

    logger.info("Worker started, waiting for tasks...")

    # Run the worker
    await worker.run()


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

Summary: How It All Works





Key Points

1. **One Workflow, Many Agents**: Single `AgentWorkflow` handles all agent types
2. **Behavior from Config**: Agent type determines which activities run
3. **Factory Pattern**: `AgentFactory.create(config)` instantiates correct class
4. **Activities are Reusable**: Same LLM/Tool activities used across types
5. **Dynamic Creation**: New agents created from UI without code changes

---

Next Steps

1. **Phase 1**: Set up infrastructure (Docker, Temporal, DBs)
2. **Phase 2**: Implement base activities (LLM, Tools)
3. **Phase 3**: Implement workflow with all agent types
4. **Phase 4**: Add safety & governance activities
5. **Phase 5**: Build API layer
6. **Phase 6**: Testing infrastructure
7. **Phase 7**: UI integration