

AI Agent Framework Comparison

A high-level comparison of five AI agent frameworks for building multi-agent systems.

Visual Overview

Maturity vs Community Size

```
quadrantChart
  title Framework Maturity vs Community
  x-axis Low Maturity --> High Maturity
  y-axis Small Community --> Large Community
  quadrant-1 Mature & Popular
  quadrant-2 New but Popular
  quadrant-3 New & Growing
  quadrant-4 Mature & Niche
  Haystack: [0.95, 0.85]
  LangGraph: [0.55, 0.85]
  Pydantic AI: [0.35, 0.60]
  MS Agent Framework: [0.15, 0.35]
  Claude Agent SDK: [0.10, 0.25]
```

Framework Timeline

```
gantt
  title Framework Release Timeline
  dateFormat YYYY-MM
  axisFormat %Y
  section Established
    Haystack      :2019-11, 2026-01
  section Growing
    LangGraph     :2023-08, 2026-01
    Pydantic AI   :2024-06, 2026-01
  section New
    MS Agent Framework :2025-04, 2026-01
    Claude Agent SDK   :2025-06, 2026-01
```

GitHub Stars Comparison

```
xychart-beta
  title "GitHub Stars (thousands)"
  x-axis [Haystack, LangGraph, "Pydantic AI", "MS Agent", "Claude SDK"]
  y-axis "Stars (k)" 0 --> 30
  bar [24, 24, 14.5, 6.8, 4.4]
```

Quick Reference

| Framework | First Release | Stars | Primary Language | Maintainer |
|---|---------------|-------|------------------|---------------|
| Haystack | Nov 2019 | 24k | Python | deepset |
| LangGraph | Aug 2023 | 24k | Python | LangChain Inc |
| Pydantic AI | Jun 2024 | 14.5k | Python | Pydantic |
| Microsoft Agent Framework | Apr 2025 | 6.8k | Python/C# | Microsoft |
| Claude Agent SDK | Jun 2025 | 4.4k | Python | Anthropic |

Haystack

The most mature framework - Originally built for semantic search and QA, evolved to support LLM agents.

| | |
|----------------------|---|
| GitHub | https://github.com/deepset-ai/haystack |
| Documentation | https://docs.haystack.deepset.ai/docs/intro |
| Quick Start | https://haystack.deepset.ai/overview/quick-start |
| Tutorials | https://haystack.deepset.ai/tutorials |
| First Release | November 2019 (~6 years old) |
| Stars | 24,000 |
| Language | Python |
| License | Apache 2.0 |

Description: AI orchestration framework to build customizable, production-ready LLM applications.

Connect components (models, vector DBs, file converters) to pipelines or agents that can interact with your data.

Strengths:

- Most battle-tested in production
- Strong RAG and document processing capabilities
- Extensive integrations ecosystem

LangGraph

State machine approach - Low-level orchestration for building stateful agents as graphs.

| | |
|----------------------|---|
| GitHub | https://github.com/langchain-ai/langgraph |
| Documentation | https://docs.langchain.com/oss/python/langgraph/ |
| API Reference | https://reference.langchain.com/python/langgraph/ |
| First Release | August 2023 (~2.5 years old) |

| | |
|-----------------|----------------|
| Stars | 24,000 |
| Language | Python (99.3%) |
| License | MIT |

Description: Build resilient language agents as graphs. A low-level orchestration framework for building, managing, and deploying long-running, stateful agents.

Strengths:

- Explicit state management with conditional edges
 - Tight integration with LangChain ecosystem
 - Free structured course via LangChain Academy
 - Used by Klarna, Replit, Elastic
-

Pydantic AI

Type-safe agents - Production-grade AI applications the Pydantic way.

| | |
|----------------------|---|
| GitHub | https://github.com/pydantic/pydantic-ai |
| Documentation | https://ai.pydantic.dev/ |
| First Release | June 2024 (~1.5 years old) |
| Stars | 14,500 |
| Language | Python (99.8%) |
| License | MIT |

Description: GenAI Agent Framework built with Pydantic's philosophy of type safety and validation.

Strengths:

- Strong typing and Pydantic validation throughout
 - Structured outputs with validation
 - Dependency injection pattern
 - Observability via Pydantic Logfire
 - Multiple LLM provider support
-

Microsoft Agent Framework

Enterprise multi-language - Unified framework consolidating AutoGen and Semantic Kernel.

| | |
|----------------------|---|
| GitHub | https://github.com/microsoft/agent-framework |
| Documentation | https://learn.microsoft.com/agent-framework/overview/agent-framework-overview |
| Quick Start | https://learn.microsoft.com/agent-framework/tutorials/quick-start |

| | |
|----------------------|---|
| User Guide | https://learn.microsoft.com/en-us/agent-framework/user-guide/overview |
| Discord | https://discord.gg/b5zjErwbQM |
| First Release | April 2025 (~9 months old) |
| Stars | 6,800 |
| Languages | Python (50.8%), C# (44.3%), TypeScript (4.5%) |
| License | MIT |

Description: A framework for building, orchestrating and deploying AI agents and multi-agent workflows with support for Python and .NET.

Strengths:

- Multi-language support (Python, C#, TypeScript)
 - Migration paths from AutoGen and Semantic Kernel
 - Microsoft enterprise backing
 - Azure integration
-

Claude Agent SDK

Claude-native - Official SDK for building agents with Claude Code capabilities.

| | |
|------------------------|---|
| GitHub | https://github.com/anthropics/clause-agent-sdk-python |
| Documentation | https://platform.claude.com/docs/en/agent-sdk/python |
| Hooks Reference | https://docs.anthropic.com/en/docs/clause-code/hooks |
| First Release | June 2025 (~7 months old) |
| Stars | 4,400 |
| Language | Python |
| License | MIT |

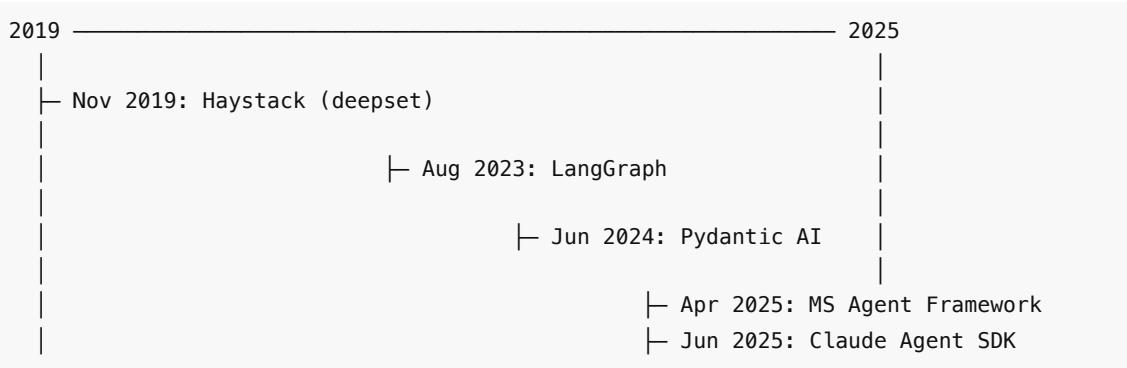
Description: Python SDK for Claude Agent - enables programmatic interaction with Claude Code, including file operations, code execution, and custom tool definitions.

Strengths:

- Native Claude Code integration (Read, Write, Bash tools)
- In-process MCP server support (no subprocess overhead)
- Permission hooks for fine-grained control
- Bundled CLI - no separate installation needed

Note: Requires Python 3.10+. Governed by Anthropic's Commercial Terms of Service.

Maturity Timeline



Installation

```
# Haystack
pip install haystack-ai

# LangGraph
pip install -U langgraph

# Pydantic AI
pip install pydantic-ai

# Microsoft Agent Framework
pip install -U agent-framework --pre

# Claude Agent SDK
pip install claude-agent-sdk
```

Tool Definition Syntax

Haystack

Three approaches - explicit `Tool` class, `@tool` decorator, or `create_tool_from_function`:

```
# Decorator approach (simplest)
from haystack.tools import tool
from typing import Annotated

@tool
def get_weather(
    city: Annotated[str, "the city for which to get the weather"] = "Munich"
):
    '''A simple function to get the current weather for a location.'''
    return f"Weather report for {city}: 20 Celsius, sunny"

# Explicit class (full control)
from haystack.tools import Tool
```

```

add_tool = Tool(
    name="addition_tool",
    description="This tool adds two numbers",
    parameters={
        "type": "object",
        "properties": {
            "a": {"type": "integer"},
            "b": {"type": "integer"}
        },
        "required": ["a", "b"]
    },
    function=add
)

```

LangGraph

Uses LangChain's tool system - pass functions directly or use decorator:

```

from langgraph.graph import StateGraph, MessagesState, START, END

def get_weather(city: str) -> str:
    """Get weather for a given city."""
    return f"It's always sunny in {city}!"

# Tools passed to create_agent() or bound to model

```

Pydantic AI

Decorator on agent instance with dependency injection via `RunContext`:

```

from pydantic_ai import Agent, RunContext

agent = Agent('anthropic:claude-sonnet-4-0')

@agent.tool
def get_weather(ctx: RunContext, city: str) -> str:
    """Get weather for a city.""" # Docstring = tool description
    return f"Weather in {city}: sunny"

```

Microsoft Agent Framework

Tools defined as Python functions passed to agent configuration:

```

from agent_framework.azure import AzureAIIClient

def get_weather(city: str) -> str:
    """Get weather for a given city."""
    return f"Weather in {city}: sunny"

# Tools bound during agent creation

```

Claude Agent SDK

In-process MCP server with `@tool` decorator:

```
from claude_agent_sdk import tool, create_sdk_mcp_server

@tool("get_weather", "Get weather for a city", {"city": str})
async def get_weather(args):
    return {
        "content": [{"type": "text", "text": f"Weather: sunny in {args['city']}"}]
    }

server = create_sdk_mcp_server(
    name="weather-tools",
    version="1.0.0",
    tools=[get_weather]
)
```

Multi-Agent Orchestration Patterns

Common Patterns Visualized

```
flowchart LR
    subgraph Sequential
        A1[Agent 1] --> A2[Agent 2] --> A3[Agent 3]
    end
```

```
flowchart TD
    subgraph Supervisor/Router
        S[Supervisor] --> |route| B1[Agent A]
        S --> |route| B2[Agent B]
        S --> |route| B3[Agent C]
        B1 --> S
        B2 --> S
        B3 --> S
    end
```

```
flowchart TD
    subgraph State Machine
        S1[Start] --> |condition| S2[State A]
        S1 --> |condition| S3[State B]
        S2 --> |success| S4[End]
        S2 --> |retry| S2
        S3 --> |success| S4
        S3 --> |error| S5[Fallback]
        S5 --> S4
    end
```

| Pattern | Haystack | LangGraph | Pydantic AI | MS Agent Framework | Claude Agent SDK |
|----------------------|---------------------------|-------------------------|----------------------------------|----------------------|----------------------|
| Sequential | Pipeline components | Graph edges | Programmatic hand-off | Agent chaining | Manual orchestration |
| Delegation | Agent as ComponentTool | Subgraphs | @agent.tool calling other agents | Nested agents | MCP tool calls |
| Supervisor | Coordinator agent + tools | Conditional routing | Application logic | Orchestrator pattern | Custom routing |
| State Machine | Pipeline branching | StateGraph + conditions | Graph-based control | State management | Manual state |

Pattern Details

Haystack: Wrap an Agent as a `ComponentTool` so a coordinator agent can invoke specialized sub-agents.

LangGraph: First-class support via `StateGraph` with `add_conditional_edges()` for routing decisions.

Pydantic AI: Five levels of complexity:

1. Single agent
2. Agent delegation (agent calls agent via tool)
3. Programmatic hand-off (app logic decides)
4. Graph-based control (state machines)
5. Deep agents (autonomous with planning)

MS Agent Framework: Inherits patterns from AutoGen (group chats) and Semantic Kernel (plugins).

Claude Agent SDK: Manual orchestration - you control the flow in your application code.

Production Readiness Assessment

| Factor | Haystack | LangGraph | Pydantic AI | MS Agent | Claude SDK |
|-----------------------|---------------------|----------------------|-------------|---------------|------------|
| Maturity | 6 years | 2.5 years | 1.5 years | 9 months | 7 months |
| Community | 24k stars | 24k stars | 14.5k stars | 6.8k stars | 4.4k stars |
| Enterprise Use | Yes (deepset cloud) | Yes (Klarna, Replit) | Growing | Yes (Azure) | Growing |
| Observability | Integrations | LangSmith | Logfire | Azure Monitor | Hooks |
| Type Safety | Partial | Partial | Strong | Partial | Partial |
| Documentation | Extensive | Extensive | Good | MS Learn | Growing |

| | | | | | |
|-------------------------|-------------|----------|----------|----------|----------|
| Breaking Changes | Stable (v2) | Evolving | Evolving | Very new | Very new |
|-------------------------|-------------|----------|----------|----------|----------|

Risk Assessment

```
xychart-beta
  title "Production Risk Level (lower = safer)"
  x-axis [Haystack, LangGraph, "Pydantic AI", "MS Agent", "Claude SDK"]
  y-axis "Risk Level" 0 --> 5
  bar [1, 2, 2, 3, 5]
```

| Framework | Risk Level | Notes |
|--------------------|-------------|--|
| Haystack | Low | Mature, stable API, Apache 2.0 license |
| LangGraph | Low-Medium | Active development, MIT license, large community |
| Pydantic AI | Low-Medium | Strong backing (Pydantic team), MIT, tested working |
| MS Agent Framework | Medium | Pre-release (--pre), but functional, Microsoft backing |
| Claude Agent SDK | High | MCP integration issues, requires Claude Code CLI |

January 2026 Testing Update:

- MS Agent Framework: Works with `pip install agent-framework --pre`. Uses familiar patterns (`ChatAgent`, `@tool` decorator). Pre-release but functional.
- Claude Agent SDK: Installs but MCP server has TaskGroup errors. Basic fallback works but full functionality requires additional Claude Code setup.

Philosophy & Mental Models

Understanding each framework's philosophy helps developers write idiomatic code and make better architectural decisions.

Haystack: "Production-Ready Pipelines"

Core Philosophy: Modularity and production-readiness from day one.

Mental Model: Think of your application as a **pipeline of swappable components**. Each component (retriever, generator, ranker, agent) is a building block you can swap without rewriting core logic.

Key Principles:

- **Composability over monoliths** - Build pipelines from small, focused components
- **Swap without rewriting** - Change model providers, vector DBs, or tools without architectural changes
- **Control your data flow** - Add loops, branches, and custom routing to fit your use case
- **Reliability over complexity** - Clean architecture with careful dependency management

Developer Mindset:

"I'm building a workflow from composable pieces. Each piece does one thing well. If I need to change providers or add a step, I swap or insert a component."

Best suited for: Developers who value stability, have complex document processing needs, or want maximum flexibility in choosing infrastructure.

LangGraph: "Agents as State Machines"

Core Philosophy: Low-level orchestration with explicit state management.

Mental Model: Think of your agent as a **graph of states and transitions**. Each node is a processing step, each edge is a possible transition. State flows through the graph explicitly.

Key Principles:

- **Explicit over implicit** - You define every state and transition, nothing is hidden
- **Durable execution** - Agents persist through failures and resume from checkpoints
- **Human-in-the-loop** - Inspect and modify agent state at any point
- **Observability first** - Deep visibility into complex agent behavior

Developer Mindset:

"I'm designing a state machine. What are my states? What triggers transitions? What data needs to persist between steps? Where might I need human intervention?"

Best suited for: Developers who need fine-grained control over execution flow, are comfortable with graph-based thinking, or need robust recovery from failures.

Pydantic AI: "Type Safety as Foundation"

Core Philosophy: Catch errors at development time, not runtime.

Mental Model: Think of agents as **typed, reusable components** - like a FastAPI router or a well-typed class. The type system is your safety net; trust it.

Key Principles:

- **Types are contracts** - Define inputs, outputs, and dependencies with Pydantic models
- **Agents are reusable** - Instantiate once, use globally (like a FastAPI app)
- **Explicit dependencies** - Use dependency injection for testability and clarity
- **Observability is essential** - "You need to actually see what happened" (built-in Logfire support)

Developer Mindset:

"What types go in? What types come out? What dependencies does this agent need? If my types are right, my agent is probably right. If something fails, I can trace it."

Best suited for: Developers who love type hints, want IDE autocomplete and static analysis, or come from strongly-typed language backgrounds.

Microsoft Agent Framework: "Agents When Necessary"

Core Philosophy: Use agents for autonomous decision-making, functions for everything else.

Mental Model: Think in terms of "**Do I need autonomy?**" If a task is well-defined and sequential, write a function. If it requires exploration, planning, or conversation - use an agent.

Key Principles:

- **Pragmatism first** - "If you can write a function to handle the task, do that instead"
- **Agents + Workflows continuum** - Single agents for autonomy, workflows for orchestration
- **Enterprise-grade foundations** - Sessions, state management, filters, telemetry built-in
- **Multi-language parity** - Same patterns work in Python, C#, and TypeScript

Developer Mindset:

"Does this task need autonomous decision-making? If yes, agent. If no, function.
For complex multi-step processes, compose agents into workflows with explicit control."

Best suited for: Enterprise teams, .NET shops, developers migrating from AutoGen/Semantic Kernel, or those who want strong guardrails against over-using agents.

Claude Agent SDK: "Claude Code as Infrastructure"

Core Philosophy: Leverage Claude's native capabilities (file ops, code execution) directly.

Mental Model: Think of Claude Code as **infrastructure you're programming against**. Your code provides tools and hooks; Claude Code provides the execution environment.

Key Principles:

- **Native capabilities** - Read, Write, Bash tools are built-in, not simulated
- **Hooks for control** - Intercept any action at any lifecycle point
- **Permission-based security** - Fine-grained control over what Claude can do
- **Event-driven architecture** - React to session events, tool calls, and completions

Developer Mindset:

"Claude Code is my execution environment. I provide tools via MCP servers.
I use hooks to enforce policies, add context, or intercept dangerous operations."

Best suited for: Developers building Claude-native applications, those who want tight integration with Claude Code's capabilities, or teams needing fine-grained permission control.

Observability Deep Dive

Understanding what's happening inside your agents is critical for debugging, optimization, and trust.

Haystack: Integration Ecosystem

Approach: No built-in observability - instead, integrates with your choice of providers.

Available Integrations:

| Provider | Type | Maintained By |
|---------------|---------|---------------|
| Arize Phoenix | Tracing | Community |

| | | |
|-----------------------------------|-------------------------|--------------------|
| Arize AI | Tracing + Monitoring | Community |
| Langfuse | Tracing + Monitoring | deepset (official) |
| OpenLIT | Monitoring + Evaluation | Community |
| Opik | Tracing + Evaluation | Community |
| Traceloop | Quality Evaluation | Community |
| Weights & Biases Weave | Tracing + Visualization | deepset (official) |

What You Can See:

- Pipeline execution flow
- Component-level latency
- Token usage per step
- Error locations and stack traces

Setup Example:

```
# Langfuse integration
from haystack_integrations.components.connectors.langfuse import LangfuseConnector

tracer = LangfuseConnector()
pipeline.add_component("tracer", tracer)
```

LangGraph: LangSmith Integration

Approach: First-party observability via LangSmith (works with or without LangChain).

Key Features:

- **Full trace visibility** - See every step your agent takes
- **Quality tracking** - Measure and track quality over time
- **Framework agnostic** - Works with LangGraph, LangChain, or standalone
- **Development to production** - Same tooling from local dev to prod

What You Can See:

- Complete request lifecycle
- Token usage and latency per step
- Tool calls and their results
- State transitions in the graph
- Human feedback integration

Setup:

```
import os
os.environ["LANGCHAIN_TRACING_V2"] = "true"
os.environ["LANGCHAIN_API_KEY"] = "your-api-key"

# That's it - tracing is automatic
```

Pricing: Free tier available, paid tiers for higher volume.

Pydantic AI: Logfire Integration

Approach: Built on OpenTelemetry with first-party Logfire support. Two lines to enable.

Key Features:

- **Automatic instrumentation** - No code changes beyond setup
- **SQL queryable** - Query telemetry data with SQL
- **HTTP transparency** - See actual prompts/completions sent to providers
- **Multi-agent tracing** - Correlate traces across agent delegation

What You Can See:

- Agent runs (top-level traces)
- Model requests (API calls)
- Tool execution spans
- Token usage and latency
- Full request/response bodies (optional)

Setup:

```
import logfire

logfire.configure()
logfire.instrument_pydantic_ai()

# Now all agent runs are traced automatically
```

Pricing: Free tier available, commercial hosting, self-hosting on enterprise.

Microsoft Agent Framework: OpenTelemetry Native

Approach: Built on OpenTelemetry with Azure Monitor integration. Most comprehensive built-in observability.

Key Features:

- **OpenTelemetry standard** - Follows GenAI Semantic Conventions
- **Traces, Logs, Metrics** - All three pillars supported
- **Azure Monitor integration** - First-class support for Azure environments
- **Sensitive data controls** - Enable/disable prompt logging

What You Can See:

- `invoke_agent <name>` - Top-level agent invocation
- `chat <model>` - Chat model calls
- `execute_tool <function>` - Tool execution
- Operation duration histograms
- Token usage histograms
- Function invocation timing

Setup (Python):

```

from agent_framework.observability import configure_otel_providers

# Option 1: Environment variables (recommended)
# Set OTEL_EXPORTER_OTLP_ENDPOINT, ENABLE_INSTRUMENTATION=true
configure_otel_providers()

# Option 2: Azure Monitor
from azure.monitor.opentelemetry import configure_azure_monitor
configure_azure_monitor(connection_string="...")

```

Environment Variables:

| Variable | Purpose |
|-----------------------------|-----------------------|
| ENABLE_INSTRUMENTATION | Enable OpenTelemetry |
| ENABLE_SENSITIVE_DATA | Log prompts/responses |
| OTEL_EXPORTER_OTLP_ENDPOINT | OTLP endpoint |
| OTEL_SERVICE_NAME | Service identifier |

Local Development: Use Aspire Dashboard (Docker) for local trace visualization.

Claude Agent SDK: Hook-Based Observability

Approach: Event-driven hooks at every lifecycle point. You build observability into your hooks.

Hook Lifecycle Events:

| Event | When It Fires | Use For |
|--------------------|-----------------------|----------------------------|
| SessionStart | Session begins | Load context, set env vars |
| UserPromptSubmit | User submits prompt | Validate, add context |
| PreToolUse | Before tool execution | Approve/deny/modify |
| PostToolUse | After tool succeeds | Log, validate results |
| PostToolUseFailure | After tool fails | Error handling |
| SubagentStart | Spawning subagent | Track nested agents |
| SubagentStop | Subagent finishes | Aggregate results |
| Stop | Claude finishes | Verify completion |
| SessionEnd | Session terminates | Cleanup, final logging |

What You Can Capture:

- Every tool call (name, input, output)
- Session and transcript paths
- Permission decisions

- Subagent execution chains
- Custom metrics via your hook scripts

Setup:

```
// ~/.claude/settings.json
{
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "*",
        "hooks": [
          {
            "type": "command",
            "command": "/path/to/log-tool-use.py"
          }
        ]
      }
    ]
  }
}
```

Hook Input (JSON via stdin):

```
{
  "session_id": "abc123",
  "transcript_path": "/path/to/transcript.jsonl",
  "tool_name": "Bash",
  "tool_input": {"command": "ls -la"},
  "tool_response": {"output": "..."}
}
```

Key Difference: Unlike other frameworks, observability is DIY – you decide what to log, where to send it, and how to visualize it. Maximum flexibility, more setup required.

Observability Comparison

```
xychart-beta
  title "Observability Setup Effort (lower = easier)"
  x-axis [Haystack, LangGraph, "Pydantic AI", "MS Agent", "Claude SDK"]
  y-axis "Effort Level" 0 --> 5
  bar [3, 1, 1, 1, 5]
```

| Aspect | Haystack | LangGraph | Pydantic AI | MS Agent | Claude SDK |
|-------------------------|----------|-------------|---------------|---------------|------------|
| Built-in | No | No | No | Yes | Hooks only |
| First-party Tool | - | LangSmith | Logfire | Azure Monitor | - |
| Standard | Varies | Proprietary | OpenTelemetry | OpenTelemetry | Custom |
| Setup Effort | Medium | Low | Low | Low | High |

| Self-host Option | Yes | No | Enterprise | Yes | Yes (DIY) |
|------------------|-----------------|-----|------------|-----------------|-----------|
| Token Tracking | Via integration | Yes | Yes | Yes | DIY |
| Multi-agent | Yes | Yes | Yes | Yes | Via hooks |
| Free Tier | Varies | Yes | Yes | Azure free tier | N/A |

Framework Overhead

Note: LLM API costs are model-dependent, not framework-dependent. Framework overhead is minimal compared to LLM latency/cost.

| Factor | Haystack | LangGraph | Pydantic AI | MS Agent | Claude SDK |
|------------------|-----------------------|--------------------|-------------|-----------------|--------------------|
| Dependencies | Medium | Medium (LangChain) | Light | Light | Light |
| Python Version | 3.9+ | 3.9+ | 3.9+ | 3.10+ | 3.10+ |
| Startup Time | Medium | Medium | Fast | Fast | Fast (CLI bundled) |
| Memory Footprint | Higher (RAG features) | Medium | Light | Light | Light |
| Model Lock-in | None | None | None | Azure-optimized | Claude-only |

Failure Modes & Error Handling

How each framework handles things going wrong - critical for production systems.

Error Handling Comparison

| Aspect | Haystack | LangGraph | Pydantic AI | MS Agent | Claude SDK |
|-------------------|-----------------------|---------------------|-------------------|-----------------|-------------------------|
| Validation Errors | Component-level | State validation | Pydantic models | Type hints | Hook-based |
| LLM Failures | Retry via integration | Checkpoint + resume | Auto-retry to LLM | Middleware | ProcessError |
| Tool Failures | Pipeline continues | Node-level handling | Error fed to LLM | Filters | PostToolUseFailure hook |
| Recovery | Pipeline breakpoints | Checkpoint resume | Reflection loop | State snapshots | Session resume |

| | | | | | |
|--------------------|--------|---------------------|--------|--------|-----------|
| Idempotency | Manual | First-class support | Manual | Manual | Via hooks |
|--------------------|--------|---------------------|--------|--------|-----------|

Haystack: Pipeline-Based Error Handling

Approach: Errors propagate through the pipeline; use breakpoints for debugging.

```
# Pipeline with error handling via try/except
from haystack import Pipeline
from haystack.components.generators import OpenAIGenerator

pipeline = Pipeline()
pipeline.add_component("generator", OpenAIGenerator())

try:
    result = pipeline.run({"generator": {"prompt": "Hello"}})
except Exception as e:
    # Access pipeline state at failure point
    print(f"Pipeline failed: {e}")
```

Key Features:

- **Pipeline breakpoints** - Pause and inspect state at any component
- **Component isolation** - Failures in one component don't necessarily stop others
- **Fallback patterns** - Tutorials show "Agentic RAG with Fallback to Websearch"

Failure Modes to Watch:

- Component timeout (no built-in timeout management)
- Invalid data types between components
- External API failures (handled per-integration)

LangGraph: Checkpoint-Based Recovery

Approach: Durable execution with automatic checkpointing. Resume from last good state.

```
from langgraph.graph import StateGraph
from langgraph.checkpoint.memory import InMemorySaver

# Enable checkpointing
checkpointer = InMemorySaver()
graph = workflow.compile(checkpointer=checkpointer)

# Run with thread ID for tracking
config = {"configurable": {"thread_id": "my-thread-123"}}
result = graph.invoke(input_data, config)

# If failure occurs, resume from checkpoint
# Same thread_id + None input = resume from last checkpoint
resumed = graph.invoke(None, config)
```

Durability Modes:

| Mode | Behavior | Use Case |
|---------|---|--------------------|
| "exit" | Persist only at completion | Best performance |
| "async" | Persist asynchronously during execution | Balanced |
| "sync" | Persist synchronously before each step | Highest durability |

Key Principle - Idempotency:

"For reliable retry mechanisms, ensure side effects are idempotent."

If a task fails partway through, resumption re-runs it. Use idempotency keys or result verification to prevent duplicate actions.

Failure Modes to Watch:

- Non-idempotent side effects on retry
- Checkpoint storage failures
- State serialization errors

Pydantic AI: Validation-Driven Recovery

Approach: Let validation errors guide the LLM to self-correct. Automatic reflection loops.

```
from pydantic_ai import Agent
from pydantic import BaseModel

class WeatherResponse(BaseModel):
    temperature: float
    conditions: str

agent = Agent('openai:gpt-4', result_type=WeatherResponse)

# If LLM returns invalid data, Pydantic AI:
# 1. Catches validation error
# 2. Sends error back to LLM
# 3. LLM retries with corrected output
result = await agent.run("What's the weather in Sydney?")
```

Error Handling Features:

- **Validation recovery** - "Errors are passed back to the LLM so it can retry"
- **Structured output retry** - If response doesn't match schema, agent is "prompted to try again"
- **HTTP request retries** - Built-in retry logic for transient API failures
- **Durable execution** - "Preserve progress across transient API failures"

Failure Modes to Watch:

- Infinite retry loops on fundamentally invalid requests
- LLM unable to produce valid schema after multiple attempts
- Dependency injection failures

Microsoft Agent Framework: Middleware-Based Interception

Approach: Use middleware/filters to intercept and handle errors at any layer.

```
from agent_framework import AIAGent
from agent_framework.middleware import RetryMiddleware, LoggingMiddleware

# Stack middleware for comprehensive error handling
agent = (
    AIAGent(client, instructions="...")
    .with_middleware(RetryMiddleware(max_retries=3))
    .with_middleware(LoggingMiddleware())
)

# Middleware intercepts:
# - Pre-execution (validation, rate limiting)
# - Post-execution (logging, metrics)
# - Errors (retry logic, fallbacks)
```

Key Features:

- **Filters** - Intercept agent actions before/after execution
- **Middleware stack** - Compose error handling behaviors
- **Session state** - Restore from session snapshots
- **Azure integration** - Application Insights for error tracking

Failure Modes to Watch:

- Middleware ordering issues
- State corruption on partial failures
- Azure service dependencies

Claude Agent SDK: Hook-Based Error Handling

Approach: React to failures via lifecycle hooks. You control recovery logic.

```
from claude_agent_sdk import ClaudeSDKClient, ClaudeAgentOptions, HookMatcher

async def handle_tool_failure(input_data, tool_use_id, context):
    """Called when any tool fails."""
    tool_name = input_data.get('tool_name')
    error = input_data.get('tool_response', {}).get('error')

    # Log the failure
    print(f"Tool {tool_name} failed: {error}")

    # Optionally provide guidance to Claude
    return {
        'hookSpecificOutput': {
            'additionalContext': f"The {tool_name} tool failed. Try an alternative approach."
        }
    }
```

```

    }

options = ClaudeAgentOptions(
    hooks={
        'PostToolUseFailure': [
            HookMatcher(hooks=[handle_tool_failure])
        ]
    }
)

```

Exception Types:

| Exception | When |
|--------------------|--|
| CLINotFoundError | Claude Code CLI not installed |
| CLIConnectionError | Failed to connect to Claude Code |
| ProcessError | Claude Code process failed (has exit_code, stderr) |
| CLIJSONDecodeError | Response parsing failed |

Recovery Options:

- resume option to continue from previous session
- fallback_model for automatic model fallback
- Hook-based retry logic (DIY)

Failure Modes to Watch:

- CLI process crashes
- Hook script failures
- Permission denials interrupting workflow

Complex Workflow Patterns

How each framework handles branching, loops, parallel execution, and conditional logic.

Workflow Capabilities Comparison

| Pattern | Haystack | LangGraph | Pydantic AI | MS Agent | Claude SDI |
|---------------|-------------------|-------------------------|-------------------|-------------------|-------------------|
| Branching | ConditionalRouter | add_conditional_edges() | Application logic | Workflow graphs | Hook decision |
| Loops | Component cycles | Graph cycles | While loops | Workflow loops | Recursive prompts |
| Parallel | AsyncPipeline | Parallel nodes | asyncio.gather | Parallel branches | Multiple agents |
| Human-in-loop | External | First-class | Manual | Workflow pause | Permissions hooks |

| State Machines | Pipeline state | StateGraph (native) | Pydantic Graph | Workflow state | Manual |
|----------------|----------------|---------------------|----------------|----------------|--------|
|----------------|----------------|---------------------|----------------|----------------|--------|

Haystack: Pipeline Branching & Loops

Branching with ConditionalRouter:

```
from haystack.components.routers import ConditionalRouter

routes = [
    {"condition": "{{query|length > 100}}", "output": "long_query", "output_type": "str"},
    {"condition": "{{query|length <= 100}}", "output": "short_query", "output_type": "str"},
]

router = ConditionalRouter(routes=routes)
pipeline.add_component("router", router)

# Connect different paths
pipeline.connect("router.long_query", "detailed_processor")
pipeline.connect("router.short_query", "quick_processor")
```

Self-Correcting Loops:

```
# Validator loops back to generator for correction
pipeline.connect("generator", "validator")
pipeline.connect("validator.invalid", "generator") # Loop back
pipeline.connect("validator.valid", "output") # Continue
```

Parallel Execution:

```
from haystack import AsyncPipeline

# Components run in parallel when dependencies allow
async_pipeline = AsyncPipeline()
# Independent branches execute concurrently
```

LangGraph: Native Graph-Based Workflows

Conditional Branching:

```
from langgraph.graph import StateGraph, END

def should_continue(state):
    if state["error_count"] > 3:
        return "fallback"
    elif state["needs_review"]:
        return "human_review"
```

```

        return "continue"

graph = StateGraph(MyState)
graph.add_node("process", process_node)
graph.add_node("fallback", fallback_node)
graph.add_node("human_review", review_node)

# Conditional routing
graph.add_conditional_edges(
    "process",
    should_continue,
    {
        "continue": "process",      # Loop
        "fallback": "fallback",    # Branch
        "human_review": "human_review"
    }
)

```

Human-in-the-Loop (First-Class):

```

from langgraph.checkpoint.memory import InMemorySaver

# Interrupt before sensitive operations
graph.add_node("sensitive_action", sensitive_node)
graph.set_entry_point("sensitive_action")

# Compile with interrupt_before
app = graph.compile(
    checkpointer=InMemorySaver(),
    interrupt_before=["sensitive_action"]
)

# Execution pauses, human reviews, then resume

```

Parallel Execution:

```

# Multiple nodes can execute in parallel if no dependencies
graph.add_node("fetch_weather", weather_node)
graph.add_node("fetch_news", news_node)
graph.add_edge(START, "fetch_weather")
graph.add_edge(START, "fetch_news") # Both run in parallel
graph.add_edge("fetch_weather", "combine")
graph.add_edge("fetch_news", "combine")

```

Pydantic AI: Programmatic Control + Optional Graph

Simple Branching (Application Logic):

```

from pydantic_ai import Agent

```

```

weather_agent = Agent('openai:gpt-4', system_prompt="Weather expert")
news_agent = Agent('openai:gpt-4', system_prompt="News analyst")

async def route_query(query: str):
    if "weather" in query.lower():
        return await weather_agent.run(query)
    elif "news" in query.lower():
        return await news_agent.run(query)
    else:
        # Default agent or error
        return await general_agent.run(query)

```

Parallel Execution:

```

import asyncio

async def parallel_agents(query: str):
    # Run multiple agents concurrently
    results = await asyncio.gather(
        weather_agent.run(query),
        news_agent.run(query),
        sentiment_agent.run(query)
    )
    return combine_results(results)

```

Pydantic Graph (For Complex Workflows):

```

from pydantic_ai.graph import Graph, Node
from dataclasses import dataclass

@dataclass
class FetchData(Node):
    url: str

    async def run(self) -> "ProcessData | ErrorState":
        try:
            data = await fetch(self.url)
            return ProcessData(data=data)
        except Exception:
            return ErrorState(message="Fetch failed")

@dataclass
class ProcessData(Node):
    data: dict

    async def run(self) -> "Complete | NeedsReview":
        if self.data.get("confidence") > 0.9:
            return Complete(result=self.data)
        return NeedsReview(data=self.data)

```

```
# Type annotations define the graph edges automatically
```

Caution from docs: "If you're not confident a graph-based approach is a good idea, it might be unnecessary." Use simple programmatic control first.

Microsoft Agent Framework: Workflow Orchestration

Explicit Workflow Graphs:

```
from agent_framework.workflows import Workflow, WorkflowNode

workflow = Workflow()

# Define nodes
workflow.add_node("classify", classify_agent)
workflow.add_node("process_urgent", urgent_agent)
workflow.add_node("process_normal", normal_agent)

# Conditional routing
workflow.add_conditional_edge(
    "classify",
    lambda state: "urgent" if state["priority"] == "high" else "normal",
    {"urgent": "process_urgent", "normal": "process_normal"}
)
```

Parallel Branches:

```
# Multiple agents run in parallel
workflow.add_parallel_nodes(
    "gather_info",
    [weather_agent, traffic_agent, calendar_agent]
)
```

Human-in-the-Loop:

```
# Workflows can pause for human input
workflow.add_node("human_review", human_review_node)
workflow.set_interrupt_before("human_review")

# Checkpointing for long-running processes
workflow.enable_checkpointing()
```

Claude Agent SDK: Hook-Driven Control Flow

Branching via Hooks:

```
async def route_based_on_tool(input_data, tool_use_id, context):
    """Intercept and redirect tool calls."""

```

```

tool_name = input_data.get('tool_name')

    if tool_name == "Write" and "/sensitive/" in input_data.get('tool_input',
{}).get('file_path', ''):

        return {
            'hookSpecificOutput': {
                'permissionDecision': 'deny',
                'permissionDecisionReason': 'Sensitive directory - use secure_write
tool instead'
            }
        }
    return {}

```

Parallel Agents:

```

import asyncio
from claude_agent_sdk import query, ClaudeAgentOptions

async def parallel_queries():
    tasks = [
        query(prompt="Analyze code quality", options=code_options),
        query(prompt="Check security issues", options=security_options),
        query(prompt="Review documentation", options=docs_options)
    ]

    # Process all agents in parallel
    results = []
    for task in asyncio.as_completed(tasks):
        async for message in await task:
            results.append(message)

```

Stop Hook for Conditional Continuation:

```

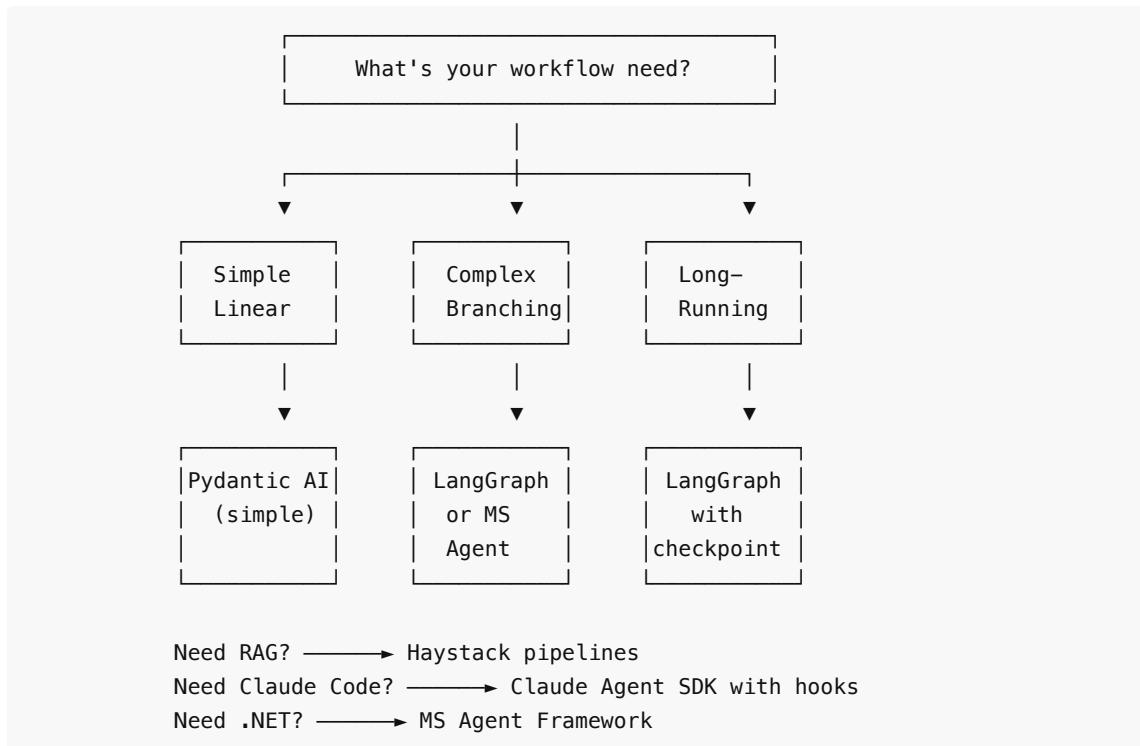
async def should_continue(input_data, tool_use_id, context):
    """Decide if Claude should continue working."""
    # Check if all tasks are complete
    transcript = read_transcript(input_data['transcript_path'])

    if not all_tasks_done(transcript):
        return {
            'decision': 'block',
            'reason': 'Not all tasks completed. Continue with remaining items.'
        }
    return {}

options = ClaudeAgentOptions(
    hooks={
        'Stop': [HookMatcher(hooks=[should_continue])]
    }
)

```

Workflow Pattern Decision Guide



Real-World Implementation Testing

We implemented the **same use case** across multiple frameworks to compare developer experience and behavior.

Note on framework versions:

- **Tested:** LangGraph, Haystack, Pydantic AI, AutoGen 0.4+
- **Not tested (too new):** Microsoft Agent Framework (Apr 2025), Claude Agent SDK (Jun 2025)

MS Agent Framework inherits patterns from AutoGen/Semantic Kernel. Claude Agent SDK is Claude-specific and wasn't part of the original evaluation scope.

Test Scenario: Insurance Weather Verification

Use Case: Australian insurance CAT (catastrophic) event verification

- **Agent 1 - Weather Verification:** Geocodes location → fetches BOM weather observations
- **Agent 2 - Claims Eligibility:** Pure LLM reasoning to apply business rules

Business Rules:

- APPROVED: Both thunderstorms AND strong wind observed
- REVIEW: Only one weather type observed
- DENIED: Neither observed

Test Data: Brisbane, QLD, 4000 on 2025-03-07

Implementation Findings

LangGraph

What worked well:

- Clean StateGraph + TypedDict state management
- @tool decorator from langchain_core is familiar
- Explicit node → edge → END structure easy to visualize

Code sample from testing:

```
class ClaimState(TypedDict):
    city: str
    coordinates: dict | None
    weather_data: dict | None
    eligibility_decision: str | None
    messages: Annotated[list, add]

workflow = StateGraph(ClaimState)
workflow.add_node("weather_agent", weather_agent_node)
workflow.add_node("eligibility_agent", eligibility_agent_node)
workflow.set_entry_point("weather_agent")
workflow.add_edge("weather_agent", "eligibility_agent")
workflow.add_edge("eligibility_agent", END)
```

Gotcha: Manual tool call loop handling required - had to iterate through response.tool_calls and invoke each tool.

Haystack

What worked well:

- OpenAIChatGenerator with tools parameter straightforward
- ChatMessage.from_tool() for tool results

Code sample from testing:

```
tools = [{

    "type": "function",
    "function": {
        "name": "geocode_location",
        "description": "Convert address to coordinates",
        "parameters": {...}
    }
}]

weather_generator = OpenAIChatGenerator(
    model=DEFAULT_MODEL,
    generation_kwargs={"tools": tools}
)
```

Gotcha: Required manual iteration loop with max_iterations guard:

```

while iteration < max_iterations:
    response = weather_generator.run(messages=messages)
    if reply.tool_calls:
        # Process tool calls manually
    else:
        break # Final response

```

Pydantic AI

What worked well:

- **Cleanest code** of all frameworks tested
- Pydantic models for structured output = automatic validation
- `@agent.tool` with `RunContext` for dependency injection
- Async-native throughout

Code sample from testing:

```

class WeatherVerificationResult(BaseModel):
    location: str
    latitude: float
    longitude: float
    thunderstorms: str
    strong_wind: str
    severe_weather_confirmed: bool

weather_agent = Agent(
    'openai:gpt-4o-mini',
    deps_type=AppDependencies,
    output_type=WeatherVerificationResult,
    instructions="..."
)

@weather_agent.tool
async def geocode(ctx: RunContext[AppDependencies], city: str, state: str, postcode: str) -> dict:
    # Tool implementation with access to shared http_client via ctx.deps

```

Result: Typed `.output` attribute with full IDE autocomplete. If LLM returns invalid data, Pydantic AI automatically re-prompts.

AutoGen 0.4+ (now superseded by MS Agent Framework)

Important: This testing was done with **AutoGen 0.4+**, which has since been **replaced by Microsoft Agent Framework** (April 2025). The new framework consolidates AutoGen + Semantic Kernel into a unified API. Patterns may differ.

What worked well:

- `AssistantAgent` with tools as plain async functions
- Simple API for single-agent use

Challenge encountered:

"Some models struggle with multi-step tool calling in RoundRobinGroupChat"

Workaround required: Had to call tools directly instead of letting agent orchestrate:

```
# Instead of letting agent use tools via RoundRobin,  
# we call tools directly and have the eligibility agent do reasoning  
geo_result = await geocode_location(TEST_CITY, TEST_STATE, TEST_POSTCODE)  
weather_result = await get_bom_weather(geo_data["latitude"], ...)  
  
# Then pass to eligibility agent  
eligibility_result = await eligibility_agent.run(task=f"Based on:  
{weather_summary}")
```

Migration note: Microsoft provides migration guides from both AutoGen and Semantic Kernel to the new Agent Framework. The multi-step tool calling issues may be resolved in the new framework.

API Testing Results

| API | Status | Sample Response |
|---------------------|---|--|
| Nominatim Geocoding | <input checked="" type="checkbox"/> Working | Brisbane → (-27.47, 153.02) |
| BOM Weather | <input checked="" type="checkbox"/> Working | Returns thunderstorm/wind observations |

Lines of Code Comparison

```
xychart-beta  
    title "Lines of Code for Same Use Case (lower = simpler)"  
    x-axis ["Pydantic AI", Haystack, LangGraph, "MS Agent", "Claude SDK"]  
    y-axis "Lines of Code" 0 --> 200  
    bar [90, 110, 120, 120, 180]
```

| Framework | LoC (same use case) | Notes |
|--------------------|---------------------|-----------------------------------|
| Pydantic AI | ~90 | Cleanest, most Pythonic |
| Haystack | ~110 | Tool schema definition verbose |
| LangGraph | ~120 | Explicit but verbose state setup |
| MS Agent Framework | ~120 | Clean API with ChatAgent + @tool |
| Claude Agent SDK | ~180 | MCP server setup + fallback logic |

Note: Claude Agent SDK demo includes fallback mode due to MCP issues, inflating LoC.

Key Takeaways from Testing

- 1. Pydantic AI had the best developer experience** - Type safety, structured outputs, and clean async code. Recommended for regulated industries where data validation matters.

2. **LangGraph** requires more boilerplate but gives explicit control over state flow - good for complex workflows where you need to visualize the graph.
3. **Haystack's tool definition is verbose** (JSON schema) compared to decorator approaches, but pipeline model is intuitive for RAG scenarios.
4. **MS Agent Framework works as pre-release** - Install with `pip install agent-framework --pre`. Uses ChatAgent with `OpenAIChatClient` and `@tool` decorator. Familiar patterns, Microsoft backing.
5. **Claude Agent SDK needs more setup** - Installs fine but MCP server integration requires Claude Code CLI. Fallback mode works for basic use.
6. **All frameworks successfully completed the use case** - Either directly or via fallback. Differences were in code elegance and setup complexity, not capability.
7. **DeepSeek works as a drop-in replacement** - All frameworks worked with DeepSeek via OpenAI-compatible API. Good for cost-conscious development.

New Framework Testing (January 2026)

We ran all five frameworks against **DeepSeek** (`deepseek-chat`) using the same Brisbane weather verification test case.

```
pie showData
  title "Test Results: Framework Status"
  "✅ Works Directly" : 4
  "⚠ Fallback Only" : 1
```

Test Results:

| Framework | Status | Decision | Notes |
|---------------------------|------------|----------|---|
| Pydantic AI | ✅ Direct | APPROVED | Both thunderstorms + wind observed |
| LangGraph | ✅ Direct | REVIEW | Only wind observed |
| Haystack | ✅ Direct | REVIEW | Only wind observed, retried BOM once |
| MS Agent Framework | ✅ Direct | REVIEW | Works with <code>pip install agent-framework --pre</code> |
| Claude Agent SDK | ✅ Fallback | REVIEW | MCP server error, fallback worked |

Key Findings:

1. **MS Agent Framework works!** - Package `agent-framework` is available as pre-release (`pip install agent-framework --pre`). Uses ChatAgent with `OpenAIChatClient`. The `@tool` decorator works similarly to other frameworks. Successfully called both geocoding and BOM weather tools.
2. **Claude Agent SDK requires Claude Code CLI** - The SDK installs (`pip install claude-agent-sdk`) but the MCP server integration had TaskGroup errors. The fallback (direct tool execution) works. Full functionality likely requires Claude Code CLI setup.

3. **BOM API variability** - Pydantic AI got "Observed" for both weather types while others got only wind. This is timing/caching variance from the BOM API, not a framework difference.

4. **DeepSeek works well** - All frameworks successfully used DeepSeek as the LLM backend via OpenAI-compatible API.

Installation Commands:

```
# Activate environment
conda activate dory

# Frameworks that work directly
pip install pydantic-ai langchain langchain-openai langgraph haystack-ai

# MS Agent Framework (pre-release)
pip install agent-framework --pre

# Claude Agent SDK
pip install claude-agent-sdk
# Note: May need Claude Code CLI for full MCP functionality
```

Run demos:

```
python pydantic_ai_demo.py
python langgraph_demo.py
python haystack_demo.py
python ms_agent_framework_demo.py
python claude_agent_sdk_demo.py
```

LLM Learnability Benchmark

A separate study measured **how easily an LLM can produce working code** with each framework, given access to documentation.

Source: github.com/srepho/fwork-learnability

Methodology

- **Model:** DeepSeek V3 (training cutoff Dec 2024) as a "temporal firewall"
- **Approach:** "Turns to Working Code" - LLM attempts implementation, receives error feedback, iterates up to 10 times
- **Documentation levels:** None, Minimal, Moderate, Full
- **Task:** Tier 1 classification tasks (48 total trials)

Results

```
xychart-beta
  title "LLM Learnability: Success Rate (%)"
  x-axis ["Pydantic AI", "Direct API", LangGraph, Haystack]
```

```
y-axis "Success Rate %" 0 --> 100
bar [92, 83, 83, 75]
```

```
xychart-beta
  title "Average Turns to Working Code (lower = easier)"
  x-axis ["Pydantic AI", "Direct API", LangGraph, Haystack]
  y-axis "Avg Turns" 0 --> 6
  bar [3.4, 2.4, 4.7, 3.0]
```

| Framework | Success Rate | Avg Turns to Success | First-Attempt Success | Optimal Doc Level |
|--------------------|--------------------|----------------------|-----------------------|-------------------|
| Pydantic AI | 92% (11/12) | 3.4 | 3/12 (25%) | Moderate |
| Direct API | 83% (10/12) | 2.4 | 1/12 (8%) | Moderate |
| LangGraph | 83% (10/12) | 4.7 | 0/12 (0%) | Minimal |
| Haystack | 75% (9/12) | 3.0 | 3/12 (25%) | Minimal |

Key Findings

1. Pydantic AI is the most learnable framework

- Highest success rate (92%)
- Good first-attempt success rate (25%)
- Benefits from moderate documentation

2. LangGraph requires more iterations

- Zero first-attempt successes
- Highest average turns (4.7) despite 83% eventual success
- Suggests steeper learning curve even for LLMs

3. Counterintuitive documentation results

- Haystack and LangGraph performed **worse** with full documentation
- Minimal docs outperformed full docs in some cases
- Likely cause: full docs captured marketing content rather than code examples

4. High model contamination detected

- All frameworks achieved 67-100% success with **zero documentation**
- Suggests DeepSeek V3's training data includes these frameworks
- "Temporal firewall" less effective than hoped

Error Patterns (17% failure rate across all trials)

| Error Type | Occurrences | Example |
|---------------------------|-------------|---|
| Syntax errors | 40 | Unterminated strings |
| API hallucinations | 26 | PydanticAI's fabricated result_type parameter |

| | | |
|--------------|----|-------------------------------------|
| Logic errors | 15 | Incorrect output despite valid code |
|--------------|----|-------------------------------------|

Implications for Framework Selection

| If you prioritize... | Choose... | Why |
|--|-------------------------|---|
| LLM-assisted development | Pydantic AI | Highest success rate, LLMs learn it fastest |
| Minimal documentation needs | Haystack or LangGraph | Work well with minimal docs |
| First-attempt correctness | Pydantic AI or Haystack | 25% first-attempt success |
| Eventual success with iteration | Any (all >75%) | All frameworks reachable with feedback |

Caveats

- Only Tier 1 (basic) tasks tested; Tier 2-3 (tool use, agent-native) pending
- Documentation fetcher captured landing pages, not tutorials
- Results reflect LLM learnability, not necessarily human learnability

Decision Flowchart

Use this flowchart to select the right framework for your use case:

```

flowchart TD
    A[Start: What's your primary need?] --> B{Heavy RAG/  
Document Processing?}
    B -->|Yes| C["Haystack"]
    B -->|No| D{Complex State  
Machine/Workflows?}
    D -->|Yes| E{Need Checkpointing  
& Recovery?}
    E -->|Yes| F["LangGraph"]
    E -->|No| G{Azure/.NET  
Environment?}
    G -->|Yes| H["MS Agent Framework"]
    G -->|No| F
    D -->|No| I{Type Safety  
Critical?}
    I -->|Yes| J["Pydantic AI"]
    I -->|No| K{Claude-Native  
Required?}
    K -->|Yes| L["Claude Agent SDK"]
    K -->|No| M{Simplest  
Setup?}
    M -->|Yes| J
    M -->|No| N{Largest  
Community?}
    N -->|Yes| O{Choose based on ecosystem}
    O --> C
    O --> F

    style C fill:#90EE90
    style F fill:#90EE90
    style J fill:#90EE90

```

```
style H fill:#FFE4B5  
style L fill:#FFB6C1
```

Quick Decision Matrix

```
pie showData  
    title "Framework Recommendation by Use Case"  
    "RAG/Documents" : 25  
    "Type Safety" : 30  
    "Complex Workflows" : 25  
    "Azure/Enterprise" : 15  
    "Claude-Native" : 5
```

Summary: When to Use Each

| Framework | Best For | Status (Jan 2026) |
|---------------------------|--|---------------------------|
| Haystack | RAG-heavy applications, document processing, production stability | ✓ Production ready |
| LangGraph | Complex state machines, explicit control flow, LangChain ecosystem | ✓ Production ready |
| Pydantic AI | Type-safe applications, structured outputs, clean Python code | ✓ Production ready |
| MS Agent Framework | Azure environments, .NET shops, migrating from AutoGen/SK | ✓ Pre-release, functional |
| Claude Agent SDK | Claude-native apps, leveraging Claude Code capabilities | ⚠ Needs setup |

Recommendation

For production use today: Choose **Pydantic AI**, **LangGraph**, **Haystack**, or **MS Agent Framework**.

Consider carefully:

- **MS Agent Framework** - Works but is pre-release (`--pre` flag required). Good if already in Azure ecosystem.
- **Claude Agent SDK** - MCP server integration needs work. Best if you're willing to set up Claude Code CLI.