# Agent Framework Architecture Deep Dive

> Expanded narrative for Slide #5 (Architecture Overview) with runnable code examples and file references. This document is intended for engineers and solution architects who need to understand the core building blocks, how they compose, and how to extend them safely.

## 1. Conceptual Map

User / System Input ─▶ Workflow (graph) ─▶ Executors / AgentExecutors ─▶ Chat Client (LLM) ─▶ Model  
 │ ▲  
 │ │  
 ├── Tools / Plugins ◀──────────┘  
 │  
 ├── Event Stream (run\_stream) ─▶ Observability / UI (DevUI, Tracing, Viz)  
 │  
 └── Output Emission (WorkflowOutputEvent)

Core components:

- \*\*Chat Agents\*\*: Stateful, tool-aware conversational reasoning units.

- \*\*Agent Executors\*\*: Adapters that let agents participate as workflow nodes.

- \*\*Workflow Builder / Graph\*\*: Fluent API for composing execution DAGs (linear, branching, parallel, looping, conditional, fan-in/out, sub‑workflows).

- \*\*Chat Clients\*\*: Provider abstractions (Azure OpenAI, Azure AI Foundry, OpenAI) for model invocation + agent creation.

- \*\*Tools & Plugins\*\*: Structured, schema-emitting Python callables (sync/async) surfaced to the model for tool calling.

- \*\*Runtime\*\*: In‑process orchestrator providing streaming, context propagation, event instrumentation, type‑aware message passing.

## 2. Chat Agents

Represents the core reasoning loop. A ChatAgent owns:

- Instructions / system persona.

- Message history management (user ↔ assistant turns, tool call traces).

- Tool registry (extracted from annotated Python functions or plugin descriptors).

- Streaming & middleware hooks.

### Key Responsibilities

- Normalize inputs into ChatMessage objects.

- Decide when to call tools vs respond directly.

- Aggregate multi‑chunk streaming responses.

### Minimal Example

from agent\_framework import ChatAgent  
from agent\_framework.openai import OpenAIChatClient # or agent\_framework.azure import AzureOpenAIChatClient  
  
client = OpenAIChatClient(api\_key="...")  
agent = ChatAgent(  
 chat\_client=client,  
 instructions=(  
 "You are a concise assistant. When given a topic, respond with a tight 2-sentence summary."  
 ),  
 tools=[], # add later  
)  
  
result = await agent.run("Explain vector embeddings")  
print(result.text)

### With a Tool

from typing import Annotated  
from pydantic import Field  
  
async def get\_weather(  
 location: Annotated[str, Field(description="City name like 'Seattle' or 'London,UK'")]  
) -> str:  
 return f"Weather for {location}: (placeholder)" # Replace with real implementation  
  
agent\_with\_tool = ChatAgent(  
 chat\_client=client,  
 instructions="Use tools when asked about weather.",  
 tools=[get\_weather],  
)  
  
response = await agent\_with\_tool.run("What's the weather in Seattle?")  
print(response.text)

## 3. Tools & Plugins

Tools are introspected for signature + metadata to build a schema the model can invoke.

- Uses typing.Annotated + pydantic.Field (or string literal) for rich parameter descriptions.

- Supports async & sync functions.

- Automatic argument validation before execution.

### Real Example (Weather Tool)

\_File: python/samples/getting\_started/agents/azure\_ai\_real\_weather/weather\_utils.py\_

async def get\_real\_weather(  
 location: Annotated[  
 str, Field(description="The city name to get weather for (e.g., 'Seattle', 'London', 'Tokyo')")  
 ],  
) -> str:  
 # Makes authenticated HTTP request to OpenWeatherMap, returns formatted multi-line string.  
 ... # See full implementation in repository

### Design Notes

- Returning a structured object (Pydantic model) is supported and recommended for downstream automation.

- Tools are pure functions from (validated inputs) → (text / structured response).

- Declarative metadata drives automatic OpenAI / Azure tool schema formatting.

## 4. Chat Clients

Abstractions for model backends that provide:

- Chat completion API surface.

- Agent factory helpers (create\_agent(...)).

- Model / endpoint configuration & credential handling.

### Azure Example

from agent\_framework.azure import AzureOpenAIChatClient  
from azure.identity import AzureCliCredential  
  
chat\_client = AzureOpenAIChatClient(  
 credential=AzureCliCredential(), # reuses az login  
 # optionally: endpoint="https://my-resource.openai.azure.com", deployment\_name="gpt-4o-mini"  
)  
  
agent = chat\_client.create\_agent(  
 instructions="You are a fact-focused summarizer.",  
 tools=[],  
)

## 5. Agent Executors

Bridges a ChatAgent into the workflow engine as a node. Adds:

- Standardized request/response dataclasses (AgentExecutorRequest, AgentExecutorResponse).

- Streaming integration with workflow event bus.

- Uniform error + cancellation handling.

### Example (Fan-Out Domains)

\_File: python/samples/getting\_started/workflow/parallelism/fan\_out\_fan\_in\_edges.py\_

researcher = AgentExecutor(  
 chat\_client.create\_agent(instructions="Research perspective."),  
 id="researcher",  
)  
marketer = AgentExecutor(  
 chat\_client.create\_agent(instructions="Marketing perspective."),  
 id="marketer",  
)  
legal = AgentExecutor(  
 chat\_client.create\_agent(instructions="Legal/compliance perspective."),  
 id="legal",  
)

## 6. Workflow Builder & Graph

Provides a fluent API to declare topology then build an immutable executable workflow.

### Common Patterns Supported

- Sequential chains: .add\_edge(a, b)

- Parallel fan-out: .add\_fan\_out\_edges(dispatcher, [a, b, c])

- Fan-in aggregation: .add\_fan\_in\_edges([a, b, c], aggregator)

- Conditional routing: edge conditions / switch constructs

- Looping constructs & checkpointing

- Sub-workflows as composite executors

### Example (Fan-Out / Fan-In)

workflow = (  
 WorkflowBuilder()  
 .set\_start\_executor(dispatcher)  
 .add\_fan\_out\_edges(dispatcher, [researcher, marketer, legal])  
 .add\_fan\_in\_edges([researcher, marketer, legal], aggregator)  
 .build()  
)

### Custom Executor (Dispatcher Extract)

class DispatchToExperts(Executor):  
 def \_\_init\_\_(self, expert\_ids: list[str]):  
 super().\_\_init\_\_(id="dispatch\_to\_experts")  
 self.\_expert\_ids = expert\_ids  
  
 @handler  
 async def dispatch(self, prompt: str, ctx: WorkflowContext[AgentExecutorRequest]):  
 initial = ChatMessage(Role.USER, text=prompt)  
 for expert\_id in self.\_expert\_ids:  
 await ctx.send\_message(  
 AgentExecutorRequest(messages=[initial], should\_respond=True),  
 target\_id=expert\_id,  
 )

## 7. Runtime & Streaming

The runtime executes the compiled graph:

- Maintains per-run WorkflowContext (message bus, cancellation tokens, state containers).

- Emits structured events (AgentRunEvent, WorkflowOutputEvent, tool invocation events, etc.).

- Supports async streaming via run\_stream() for incremental progress consumption.

### Streaming Consumption

async for event in workflow.run\_stream("Launch plan for budget e-bike"):  
 if isinstance(event, AgentRunEvent):  
 print("Agent step:", event.executor\_id, event.status)  
 elif isinstance(event, WorkflowOutputEvent):  
 print("Final Output:\n", event.data)

### Visualization Hook

from agent\_framework import WorkflowViz  
viz = WorkflowViz(workflow)  
print(viz.to\_mermaid())  
# Optionally export SVG (requires viz extras & GraphViz installed): viz.export\_svg("workflow.svg")

## 8. Cross-Cutting Concerns

|  |  |
| --- | --- |
| Concern | Mechanism |
| Environment / Secrets | `.env` + `dotenv` loading (scripts) / Azure Identity (managed creds) |
| Observability | Event stream, tracing (OTel instrumentation), structured events |
| Extensibility | Custom Executors, additional tool functions, middleware wrappers |
| Error Handling | Structured error events, status propagation, guarded tool execution |
| Security | Principle of least privilege for credentials; tool input validation via Pydantic |
| Determinism Controls | Model parameters in Chat Client (temperature, top\_p, etc.) |
| State Management | Shared state objects / checkpointing samples |

## 9. End-to-End Mini Example

# 1. Define a tool  
async def get\_weather(city: Annotated[str, Field(description="City name")]):  
 return f"Weather for {city}: Sunny (demo)"  
  
# 2. Create client & agent  
chat\_client = AzureOpenAIChatClient(credential=AzureCliCredential())  
agent = chat\_client.create\_agent(  
 instructions="Answer user questions; call tools when needed.",  
 tools=[get\_weather],  
)  
  
# 3. Wrap in executor and simple workflow  
executor = AgentExecutor(agent, id="weather")  
workflow = WorkflowBuilder().set\_start\_executor(executor).build()  
  
# 4. Stream run  
async for ev in workflow.run\_stream("What's the weather in Berlin?"):  
 if isinstance(ev, WorkflowOutputEvent):  
 print(ev.data)

## 10. Extension Points & Roadmap Placeholders

|  |  |  |
| --- | --- | --- |
| Area | Today | Future Direction |
| Deployment | In-process Python | Multi-host / distributed runtime, queue-backed execution |
| Scheduling | Manual invocation | Cron / event-driven triggers |
| State Persistence | Sample checkpoint patterns | Pluggable durable stores (Redis, Cosmos DB) |
| Policy & Guardrails | Tool input validation | Rich policy engine (rate limits, content filters) |
| Multi-Language | Python + early .NET | Full parity & language-agnostic protocol |

## 11. Best Practices Summary

- Keep tool responses structured when they will feed automation (prefer models over free text).

- Use fan-out only when parallelism meaningfully reduces latency versus serial prompts.

- Introduce aggregation nodes that perform semantic reduction, not just concatenation.

- Guard external calls (network / APIs) with timeouts and explicit error messages.

- Stream results in user-facing UX; buffer only when atomic output is required.

## 12. File Reference Index

|  |  |  |
| --- | --- | --- |
| Component | Representative File | Purpose |
| Weather Tool | `python/samples/getting\_started/agents/azure\_ai\_real\_weather/weather\_utils.py` | Real external API tool |
| Fan-Out/Fan-In Workflow | `python/samples/getting\_started/workflow/parallelism/fan\_out\_fan\_in\_edges.py` | Parallel pattern |
| Visualization | `python/samples/getting\_started/workflow/visualization/concurrent\_with\_visualization.py` | Graph rendering |
| Tutorial Notebook | `python/samples/getting\_started/workflow/\_start-here/workflow\_tutorial.ipynb` | Stepwise learning |
| DevUI Weather Agent | `python/packages/devui/samples/weather\_agent\_azure/weather\_agent\_demo.ipynb` | Interactive agent demo |

## 13. Glossary

|  |  |
| --- | --- |
| Term | Definition |
| Agent | Reasoning entity with instructions + optional tools. |
| Executor | Executable unit in workflow graph (can be custom or an AgentExecutor). |
| AgentExecutor | Adapter that lets an agent run as an executor node. |
| Tool | Callable exposed to the model for grounded actions. |
| Workflow | Compiled directed graph of executors and edges. |
| Event Stream | Async sequence of structured runtime events. |

---

Contributions & improvements welcome. See CONTRIBUTING.md.