# 3.0 AGENTIC AI & ORCHESTRATION OVERVIEW

# LEARNING OBJECTIVES

By the end of Day 3 (morning theory block), you should be able to:

- Explain what **Agentic AI** is and how it differs from “single-shot” LLM calls.
- Describe common **agent patterns**: tool calling, ReAct, plan–act, multi-agent and graph-based flows.
- Map these patterns onto concrete frameworks:
  - **CrewAI** – multi-agent orchestration in Python.
  - **Langflow** – visual builder for LangChain-style flows.
  - **LangGraph** – graph-based orchestration for complex workflows.
  - **watsonx Orchestrate** – IBM’s production-grade agent platform.
- Understand how your **RAG accelerator** turns into a reusable tool for agents.

# 3.1 WHAT IS AGENTIC AI?

So far we mostly used LLMs like functions:

> *Prompt in → answer out*

Agentic AI adds **reasoning + action**:

- The LLM doesn’t just answer — it:
  - Decides **which tools to use** (APIs, RAG services, calculators…).
  - Plans a sequence of **steps**.
  - Executes tools, reads results, and continues reasoning.
- The result is an **agent**:
  - Has a goal (“help user with workshop questions”).
  - Has **capabilities** (tools, collaborators, knowledge bases).
  - Uses an LLM for planning and reflection.

Agentic AI is powerful when:

- You need **multi-step workflows** (e.g. search → filter → call RAG → summarize).
- You need to integrate with **existing systems** (ticketing, CRM, data lakes).
- You want **traceability**: which tools were used, which docs were read, etc.

# 3.2 CORE AGENT PATTERNS

We'll refer to these patterns throughout Day 3:

## 1. TOOL-CALLING AGENT

- LLM chooses **one tool** at a time based on user input.
- Example:
  - `rag_service_tool(question)` → calls your accelerator `/ask` endpoint.
  - `calculator_tool(expression)` → safe arithmetic evaluation.
- Agent picks a tool, calls it, and formats the result back to the user.

## 2. REACT (REASON + ACT)

- LLM alternates between **thinking** and **acting**:
  - **Thought:** "I should call the RAG service to get context."
  - **Action:** Use tool `rag_service_tool`.
  - **Observation:** Tool output.
  - Repeat…

Many frameworks (CrewAI, LangGraph, watsonx Orchestrate "react style") build on this idea.

## 3. PLAN–ACT

- LLM **plans a sequence** of steps, then executes them:

# 3.3 FRAMEWORK TOUR

In the morning we'll conceptually walk through four frameworks that implement these patterns.

## 3.3.1 CREWAI

- **What it is:** A Python library for multi-agent "crews".
- **Mental model:**
  - `Agent` – role, goal, backstory, tools.
  - `Task` – what needs doing, expected output.
  - `Crew` – group of agents + tasks + process (`Process.sequential`, `Process.hierarchical`, …).
- **Where it shines:**
  - Faster prototyping of **multi-agent** patterns.
  - Narrative workflows (research → writing → editing).
- **Workshop angle:**
  - A single CrewAI agent using your **accelerator RAG API + calculator**.

## 3.3.2 LANGFLOW

- **What it is:** A visual builder for LangChain flows.
- **Mental model:**
  - Drag-and-drop components (LLM, retriever, tools, routers, prompts).
  - Connect them as a **graph**.

# 3.4 MORNING FLOW (APPROX. 4H)

Suggested agenda (adapt to your group):

1. **Intro & recap (30–45 min)**
   - Why agents on top of RAG?
   - Quick recap of RAG architecture from Day 2.
2. **Agent patterns walkthrough (45 min)**
   - Tool-calling, ReAct, plan–act, multi-agent, graph-based.
3. **Framework lightning demos (90–120 min)**
   - **CrewAI mini-demo:**
     - One agent with a “support engineer” role.
     - Tools: calculator + stubbed RAG tool.
   - **Langflow mini-demo:**
     - Visual RAG chain that calls an LLM and shows citations.
   - **LangGraph mini-demo:**
     - Simple graph: retrieve → generate.
   - **Orchestrate concept demo:**
     - Show YAML / ADK structure for a “Hello World Agent”.
4. **Bridge to afternoon labs (15–30 min)**
   - Show how the accelerator `/ask` endpoint becomes **the** key tool.

# 3.5 HOW THIS CONNECTS TO LABS

- **Day 2 → Day 3 bridge:**
  - Day 2 gave you a **production-like RAG service** (`/ask` endpoint).
  - Day 3 gives you **agents** that call that service as a tool.
- **Lab 3.1 – Local Agent in simple-watsonx-environment:**
  - Agent has two tools: RAG service, calculator.
  - Planner LLM chooses which tool to call, then composes final answer.
- **Agent frameworks (CrewAI / LangGraph / Orchestrate):**
  - You can re-express the same logic in different frameworks:
    - CrewAI for Python multi-agent setups.
    - LangGraph for stateful workflows and evaluation.
    - Orchestrate for production deployment and governance.

By the end of Day 3, your mental model should be:

> *Docs → RAG (Day 2) → Agent on top (Day 3) → Orchestrated & governed in watsonx (beyond the workshop).*

# 3.2 BRIDGE TO WATSONX ORCHESTRATE & GOVERNANCE

# LEARNING OBJECTIVES

- Understand how notebook patterns translate to watsonx Orchestrate & governance tools.
- Recognize where policies and monitoring fit.

# MAPPING ACCELERATOR + AGENT TO ORCHESTRATE

Think of watsonx Orchestrate as a **platform** that can host the patterns you built in notebooks.

- Conceptual mapping:
  - `accelerator/service/api.py` → a reusable “RAG answer” action (tool / service).
  - Agent notebook → blueprint for a multi-step orchestrated workflow.

## EXAMPLE ORCHESTRATE FLOW

1. Receive a user request (chat, API, UI).
2. Decide whether to:
   - Call the **RAG service** (accelerator `/ask` endpoint).
   - Call a calculator or other utility tool.
   - Escalate to a **human** (handoff / ticket).
3. Log all steps and outcomes for governance.

In Orchestrate terms:

- **Agent** = top-level orchestrator that plans and delegates.
- **Tools** = your APIs (RAG, calculator, external systems).
- **Connections** = how tools authenticate to external services.
- **Knowledge bases** = RAG-ready corpora defined in Orchestrate.

# GOVERNANCE & EVALUATION

watsonx.governance adds a layer of **control and insight** over your agents and models:

- **Model & asset catalog**
  - Track which models, prompts, tools, and agents exist.
  - Versioning and metadata.
- **Policies**
  - Allowed models / endpoints.
  - Risk profiles and approval workflows.
  - For example:
    - “Customer-facing agents may only use models with a given risk rating.”
- **Evaluation Studio**
  - Run experiments on your agents / RAG flows.
  - Compare LLMs, prompts, retrieval strategies.
  - Track metrics like faithfulness, answer relevance, content safety.
- **Monitoring**
  - Track runs and metrics over time.
  - Detect drift or degradation.
  - Investigate failures and edge-cases.

# USING EXISTING NOTEBOOKS AS A BRIDGE

You already have governance-related notebooks in `labs-src` and `accelerator`:

- `labs-src/ibm-watsonx-governance-governed-agentic-catalog.ipynb`:
  - Shows how to:
    - Register models & tools in a governed catalog.
    - Create and use governed tools (e.g. PII detectors, jailbreak detectors).
- `labs-src/ibm-watsonx-governance-evaluation-studio-getting-started.ipynb`:
  - Shows how to:
    - Define evaluation datasets.
    - Compute metrics like context relevance, faithfulness, answer relevance.
- `accelerator/assets/notebook/notebook:Analyze_Log_and_Feedback.ipynb`:
  - Concrete example of:
    - Pulling logs from a deployed service.
    - Analyzing quality and user feedback.
    - Bridging logs → evaluation datasets.

Your Day 3 agent logs can be fed into these notebooks as a **first step** towards full governance.

# FROM LOGS TO GOVERNANCE

Where do logs come from?

- `rag/pipeline.py`:
  - Can log retrievals and model calls (question, chunks, model, latency).
- `service/api.py`:
  - Can log user requests, status codes, errors.
- Agent notebook (Lab 3.1):
  - Can log tool choices, planner outputs, tool responses, final answers.

These logs can be:

1. Exported periodically as CSV/JSON.
2. Loaded into:
   - Evaluation Studio datasets.
   - Analysis notebooks (e.g. `Analyze_Log_and_Feedback.ipynb`).

Governance workflows can then:

- Detect drift in answer quality.
- Enforce thresholds:
  - “If faithfulness < 0.8, flag for review.”
- Support audit trails:

# EXAMPLE USE CASES

Some concrete patterns where orchestrate + governance shine:

- **Governed HR assistant**
  - Uses RAG over HR policies.
  - Tools for detecting PII and harmful content.
  - Governed model + tool catalog and metrics.
- **Customer support bot with monitored outputs**
  - Agent routes between FAQ RAG, ticket creation, and human handoff.
  - Governance monitors answer relevance and safety.
- **Internal knowledge bot with evaluation cycles**
  - Weekly evaluation runs on a curated question set.
  - Results feed into prompt/model/index improvements.

# HOW TO GO FROM LAB TO PRODUCTION

From Day 3 notebooks to real systems:

- Extract the best patterns from:
  - `agent_watsonx.ipynb` (agent loop + tools).
  - `accelerator` service (RAG microservice).
  - Governance notebooks (evaluation + catalog).
- Embed them into your delivery pipeline:
  - Use `Makefile` / `pyproject.toml` / Dockerfiles to package the accelerator.
  - Deploy accelerator API behind a stable endpoint.
  - Register it as a tool / connection in Orchestrate.
- Connect Orchestrate deployments to governance:
  - Ensure models and agents are registered in the catalog.
  - Route logs to Evaluation Studio or custom analysis.

# DISCUSSION PROMPTS

To close the session, consider:

- Where would you use **orchestration** and **governance** in your organization?
- Which components from this workshop are closest to your production needs?
- What would you need to:
  - Onboard your own data?
  - Apply your internal policies?
  - Connect to your existing IT systems?

Capture these ideas — they are great seeds for the **capstone day** and follow-up projects.

# 3.3 RECAP & NEXT STEPS (2H DISCUSSION)

# SESSION OVERVIEW

This is a 2-hour interactive recap and discussion to consolidate learning and plan next steps.

# PART 1 – TIMELINE RECAP

Walk through the workshop story:

- **Day 0:** Environment setup
  - Two env repos (Ollama + watsonx).
  - `accelerator` repo.
  - `labs-src` reference notebooks.
- **Day 1:** LLM basics & prompting
  - Prompt patterns and reliability.
  - Compare local vs hosted models (Ollama vs watsonx).
- **Day 2:** RAG
  - Local RAG notebooks (Ollama + Chroma).
  - RAG notebooks in `labs-src` (Elasticsearch, Chroma, Granite).
  - Accelerator ingestion + retriever + pipeline skeleton.
- **Day 3:** Orchestration & agents
  - Agent notebook calling the accelerator API.
  - Accelerator API & (optional) Streamlit UI.
  - Governance & Orchestrate notebooks.

# PART 2 – GROUP REFLECTION

Suggested questions for the group:

- What was most surprising across the three days?
- Biggest “aha” moments?
- Hardest parts of the labs?
- Where did you hit friction with:
  - Environments?
  - Libraries / SDKs?
  - Conceptual pieces (RAG, agents, governance)?

Capture answers on a shared board or document.

# PART 3 – OPEN Q&A & TROUBLESHOOTING

Use this time to address remaining technical issues:

- Ollama / watsonx environments:
  - Model loading, API keys, network.
- Accelerator service:
  - `/ask` endpoint behaviour.
  - Vector DB connectivity.
  - Logging & configuration.
- Agent notebooks:
  - Tool selection.
  - JSON formatting issues.
  - Error handling.

Encourage participants to share **their code** and walk through solutions live.

# PART 4 – FROM WORKSHOP TO REAL PROJECTS

Discuss how to turn the accelerator into something “real”:

- **Production microservice**
    - Hardened FastAPI app.
    - Config via `service/deps.py` and environment variables.
    - Health checks, observability, logging.
- **Scalable ingestion pipeline**
    - Batch extraction / chunking / embedding.
    - Scheduling (nightly or event-driven).
    - Monitoring index size and freshness.
- **Integration**
    - CI/CD pipelines using `Makefile`, `pyproject.toml`, Dockerfiles.
    - Deployment targets: Kubernetes / OpenShift / Cloud Foundry / VM.
    - Governance and evaluation workflows with watsonx.governance.

# PART 5 – SUGGESTED NEXT STEPS

Potential directions for deeper dives:

- **Retrieval & ranking**
  - Hybrid search (lexical + semantic).
  - Rerankers and scoring strategies in `retriever.py`.
- **Prompting & safety**
  - Advanced prompt strategies in `prompt.py`.
  - Safety / refusal patterns and guardrails.
- **Multi-tenancy**
  - Tenant-aware configuration in `service/deps.py`.
  - Per-tenant indices and model settings.
- **UI & UX**
  - Richer Streamlit UI (`ui/app.py`).
  - Conversation history, feedback buttons, source highlighting.

Provide pointers to:

- Internal docs and repos.
- IBM public samples (`labs-src`, GitHub repos).
- Governance and Orchestrate documentation.

# OPTIONAL: FEEDBACK FORM

Share a link or QR code to a short survey covering:

- Overall satisfaction.
- Clarity of explanations.
- Lab difficulty.
- Which topics participants want more of.

Use this to plan the **capstone day** and any follow-up sessions.