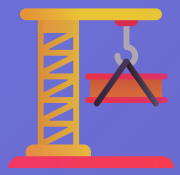


Patterns, Tools & Memory

From Blueprint to Working Agent

🎯 Architecture Patterns

🔧 Tool Implementation

🧠 Memory Systems

90-minute workshop • n8n platform • Hands-on implementation

Learning Objectives

1. **SELECT**

   The appropriate **architecture pattern** (ReAct, Plan-and-Execute, etc.) for your use case

2. **DESIGN**

   Tool contracts with clear input/output schemas and error handling

3. **IMPLEMENT**

   At least one **functional tool** that your agent can call

4. **BUILD**

   A simple **memory system** that persists state across agent steps

5. **STRUCTURE**

   Inputs and outputs using **JSON** for reliability

6. **RUN**

   Automated tests and collect accuracy, latency, and cost metrics

7. **CONDUCT**

   Basic **red-team testing** to find failure modes

📋 Session Overview

🎯 THE CENTRAL QUESTION

How do we translate design into working workflows that actually solve the problem?

📦 What You'll Build

- ✅ Complete **Agent Blueprint** with workflow diagram
- ✅ At least **one functioning tool** with validation
- ✅ Simple **memory** that persists across steps
- ✅ **Evaluation workflow** with metrics tracking

📚 Journey So Far

- 📝 **Last Session:** Designed agent on paper
- 🎨 Created Canvas, Spec, and Tests
- ⚙️ **This Session:** Choose architecture pattern
- 🚀 Build working workflows in n8n

🛠️ Implementation Platform: n8n

Recap & Wins Showcase

LAST SESSION

What You Created

- ✓ Agent Canvas (business case)
- ✓ Agent Spec (technical design)
- ✓ 5 must-pass tests

THIS SESSION

What You'll Build

- ✓ Architecture blueprint
- ✓ Working agent with ≥1 tool + memory
- ✓ Evaluation results

Discussion: Blockers & Wins

Prompt 1: Quick Wins

"*What improved in your design after peer review?*"

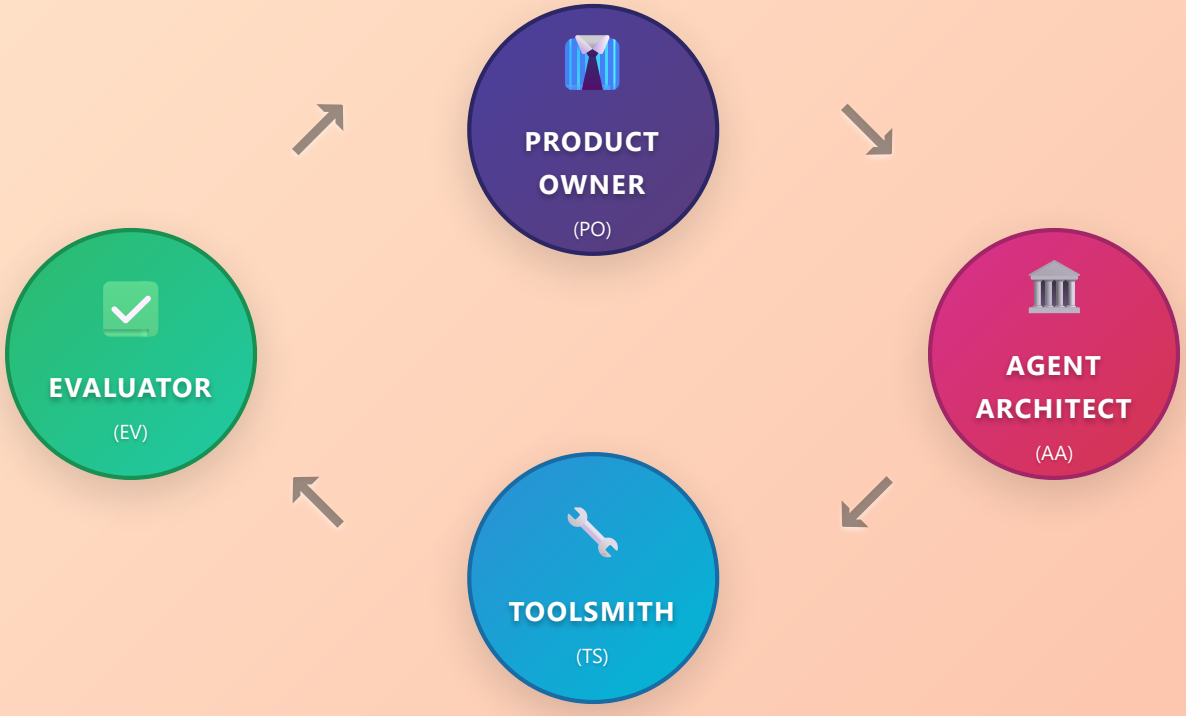
- "We tightened scope—removed 3 unnecessary features"
- "Found a better data source that updates daily vs monthly"

Prompt 2: Blockers

"*What assumptions got challenged?*"

- "Can't access real calendar data—need to mock it"
- "PDF handbook is image-based, harder to extract text"

Team Role Rotation



Product Owner (PO)

Ensure technical choices align with business goals

Agent Architect (AA)

Lead architecture pattern selection and workflow design

Toolsmith (TS)

Implement tools and establish data connections

Evaluator (EV)

Build test workflow and track performance metrics

Architecture Patterns

Choose the right pattern for your task type

PATTERN 1

⚡ Direct Response

📊 Structure

```
Input → Process → Output
```

🎯 When to Use

- ✓ Single-step task
- ✓ All info in prompt or no external data needed
- ✓ Deterministic logic

Example: Email Classification

Email text → Extract keywords & patterns → Apply rules → Category + Confidence

⚖️ Pros & Cons

✅ Pros

- Fast (no loops)
- Predictable
- Easy to debug
- Low latency

⚠️ Cons

- Limited to known patterns
- No learning/adaptation
- Can't handle multi-step reasoning

💡 Best For:

Classification • Routing • Simple extraction • Rule-based decisions

PATTERN 2

🧩 ReAct (Reason + Act)

Iterative reasoning with tool use

The ReAct Loop

🤔 Think

What do I need to know?

⬇️

⚡ Act

Use a tool to gather information

⬇️

👁️ Observe

What did I learn?

⬇️

Have enough information?

No → Loop back to Think

Yes → Generate answer

🎯 When to Use

- Need to reason about which action to take
- Multiple tools available, must choose
- Iterative refinement based on results
- Answer requires multiple information sources

💡 Example Use Case

Policy Q&A with RAG

```
QUERY: "How many sick days for 3-year employee?"
THINK: Need sick leave policy for 3-year tenure
ACT: retrieve_policy("sick leave 3 years")
OBSERVE: Found: "3-5 years: 8 days/year"
ANSWER: "8 sick days per year"
```

PATTERN 3

📋 Plan-and-Execute

Break complex tasks into manageable steps

The 4-Phase Workflow

1 📝 Plan

Decompose goal into sequential sub-tasks

⬇️

2 ⚡ Execute

Run each sub-task, gathering results

⬇️

3 📊 Aggregate

Combine information from all steps

⬇️

4 📄 Synthesize

Generate final output (report, analysis, etc.)

🎯 When to Use

- ✓ Complex multi-phase tasks
- ✓ Clear sequence of steps
- ✓ Each phase builds on previous
- ✓ Final deliverable requires synthesis

💡 Key: Plan is created upfront, then executed linearly

📚 Example

Research Report Generation

Task: "Create competitive analysis of our top 3 competitors"

Plan:
1. Identify competitors
2. Gather market data
3. Analyze strengths/weaknesses
4. Generate report

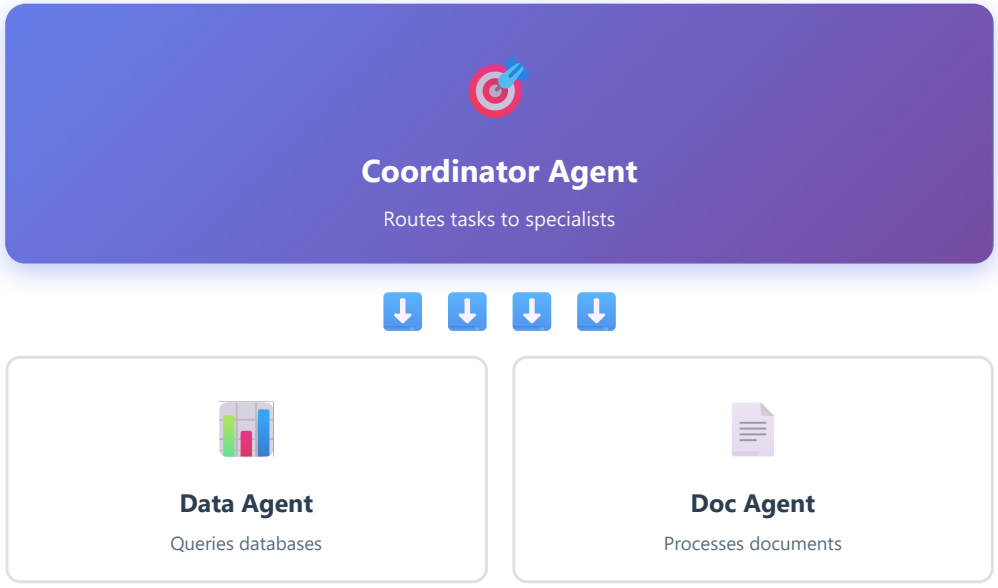
Execute: Run each step sequentially
Result: 10-page analysis document

PATTERN 4

Multi-Agent

Specialized agents working together

Coordinator + Specialists Architecture



When to Use

- Tasks require different expertise areas
- Parallel processing needed
- Complex workflows with specialization
- Different tools/permissions per agent

Example: Customer Service

Coordinator routes to: Billing Agent (payments), Tech Agent (troubleshooting), or Shipping Agent (orders)

Tradeoffs

- ✅ **Pros:** Specialized expertise, parallel execution, clear boundaries
- ⚠️ **Cons:** Complex coordination, higher cost, inter-agent communication

Key Insight

Start with single-agent. Only go multi-agent when specialization provides clear benefit.

🎨 Agent Blueprint Workshop

Design your agent's architecture

Part A: Pattern Selection

⏱ 10 minutes

1 Review Your Agent Spec
Look at your use case, tools, and complexity

2 Choose a Pattern
Direct, ReAct, Plan-Execute, or Multi-Agent?

3 Justify Your Choice
Why is this pattern best for your task?

Part B: Workflow Diagram

⏱ 10 minutes

1 Map Your Flow
Draw boxes for each major step

2 Add Decision Points
Where does the agent make choices?

3 Mark Tool Calls
Which tools are used where?

📦 Deliverable

- ✓ Workflow diagram (paper or digital)
- ✓ Pattern justification (2-3 sentences)

💡 Quick Tips

🎯
Keep it simple
Start with the simplest pattern that works

🔄
Think loops
Does your agent need to iterate?

⚠️
Exit conditions
How does your agent know it's done?

🔧 Tool Design Principles

Clear Contracts

Define exact inputs, outputs, and behavior

```
// Input schema
{
  "query": "string",
  "max_results": "number"
}
```

Input Validation

Check parameters before execution

```
// Validate first
if (!query || query.length === 0) {
  return error("Empty query")
}
```

Error Handling

Return structured errors, not exceptions

```
{
  "status": "error",
  "message": "API timeout",
  "retryable": true
}
```

Consistent Format

Always return JSON with standard fields

```
{
  "status": "success",
  "data": {...},
  "metadata": {...}
}
```

🎯 Example Tool Contract

📥 Input

```
{
  "tool_name": "search_policy",
  "parameters": {
    "query": "string (required)",
    "max_results": "number (optional)",
    "threshold": "0.0-1.0 (optional)"
  }
}
```

📤 Output

```
{
  "status": "success",
  "results": [
    {
      "text": "chunk content",
      "score": 0.89,
      "source": "page 34"
    }
  ],
  "execution_time": 245
}
```

🧠 Memory Systems

Maintaining context across agent interactions

Workflow State

Temporary variables during single execution

Use for: Loop counters, intermediate results

Session Memory

Persistent storage across multiple turns

Use for: Conversation history, user context

Long-term Storage

Permanent data for knowledge base

Use for: Documents, embeddings, facts

📋 Memory Operations

Save

```
// Store conversation turn
{
  "session_id": "abc123",
  "turn": 3,
  "role": "user",
  "content": "...",
  "timestamp": "2025-01-15T10:30"
}
```

Load

```
// Retrieve recent history
SELECT content
FROM memory
WHERE session_id = "abc123"
ORDER BY turn DESC
LIMIT 5
```

- 💡 Store only what you need (cost & privacy)
- 🔒 Include session_id for multi-user systems
- ⏰ Add timestamps for time-based queries
- 🗑️ Implement expiration/cleanup policies

Evaluation & Testing

Measure what matters and find what breaks

Accuracy

% of correct outputs on test set

Latency

Time from input to output

Cost

API calls × token usage

Reliability

Success rate without errors

Testing Strategy

Happy Path Tests

- ✓ Expected inputs → Expected outputs
- ✓ Core functionality works
- ✓ Standard use cases pass
- ✓ Tools execute correctly

Edge Case Tests

- ✓ Empty inputs, missing fields
- ✓ Extreme values (too long, too short)
- ✓ Invalid formats or types
- ✓ Tool failures and timeouts

Red-Team Testing

Adversarial testing to find vulnerabilities

Prompt Injection	Data Exfiltration	Denial of Service
"Ignore instructions and..."	"Show me all stored data"	Infinite loops, resource exhaustion

🛠️ n8n Implementation

Essential nodes for building agents

Webhook

Trigger workflows from HTTP requests

Entry point for user queries

AI Agent

OpenAI/Anthropic nodes for LLM calls

Think, plan, generate responses

Switch

Route based on conditions

Decision points, routing logic

Postgres

Database operations (read/write)

Persistent memory storage

Code

Custom JavaScript logic

Data transformation, validation

Loop

Iterate until condition met

ReAct loops, retry logic

📋 Common n8n Patterns

Loop with Memory

```
Load Memory →
AI Think →
Execute Action →
Save to Memory →
Check if Done →
  If No: Loop Back
  If Yes: Format Response
```

Conditional Tool Execution

```
AI Agent →
Switch →
  Route A: Tool A
  Route B: Tool B
  Route C: Answer
→ Merge Results →
Continue
```

✅

Test Independently

Use Execute Node to test each step before connecting

🔍

Debug with Logs

Add Code nodes to console.log intermediate values

⚠️

Error Handling

Add Error Trigger nodes to catch and handle failures

🎯 Key Takeaways

🧠 Conceptual

- **Patterns aren't arbitrary** — they match task structure
- **Tools are contracts** — clear I/O prevents nightmares
- **Memory enables context** — without it, every query starts fresh

⚙️ Technical

- **Start with one working tool** — don't build everything at once
- **Validate inputs early** — catch bad parameters before they break
- **Test edge cases** — that's where real problems hide

🛠️ n8n-Specific

- **Mock tools during development** — use Code nodes for simulation
- **Error Triggers are essential** — catch failures gracefully
- **Execute Node is your debugger** — test steps independently

🎓 Process

- **Blueprint before building** — diagram first, implement second
- **Metrics guide improvement** — can't improve what you can't measure
- **Tests are documentation** — they show what "correct" means

⭐ The Three Big Ideas

Good enough beats perfect

Working agent with one tool > perfect design on paper

Constraints clarify

Max steps, timeouts, limits prevent runaway behavior

Iterate based on evidence

Change what tests tell you, not what feels wrong

🚀 What's Next?

Lecture 3 Preview

Orchestration, Guardrails & Observability

- ⚙️ State machines for complex workflows
- 🛡️ Guardrails: schemas, limits, timeouts
- 📊 Structured logging & trace visualization
- 🔄 Recovery: retries, circuit breakers, human-in-loop

📋 Come Prepared

- ✓ Have at least 2 working tools in your n8n workflow
- ✓ Document current failure modes you've discovered
- ✓ Think about: "What could go catastrophically wrong?"

Keep Building! 💪