Phase 6: Multi-Tool Agent

Weekly Exercises — Months 11–12

Phase Overview

By the end of this phase, you'll have a capable design system research assistant. You'll add discovery tools, analysis capabilities, and export features. You'll also build comprehensive tracing to understand agent behavior and enable debugging of complex interactions.

What You'll Build

- Discovery tools: list systems, list components, list content types
- Analysis tools: find components with specific features, gap analysis
- Export tools: generate comparison reports
- Comprehensive execution tracing

Key Python Concepts

State management with dataclasses, trace logging and serialization, complex multi-step workflows, report generation, advanced prompt engineering for multi-tool scenarios.

Week 1: Discovery Tools

Goal

Add tools that let the agent discover what's available: which design systems are loaded, what components each system has, and what content types exist.

Exercise

Add three discovery tools:

- **ListSystemsTool:** No parameters. Returns list of ingested design systems with doc counts.
- **ListComponentsTool:** Parameter: design_system (optional). Returns list of components, optionally filtered by system.
- **ListContentTypesTool:** No parameters. Returns available content types (props, usage, accessibility, etc.) with counts.

These tools query your registry's metadata. Add helper methods to your registry if needed: list_design_systems(), list_components(system: str | None), get_content_type_counts().

Test by asking the agent: "What design systems do you have?" and "What components are in Carbon?" Verify it uses the discovery tools before attempting to answer.

Why This Matters

Discovery tools prevent hallucination. Without them, the agent might claim to know about systems that aren't loaded or components that don't exist. These tools ground the agent's knowledge in reality.

Completion Checklist

- ☐ ListSystemsTool implemented
- ☐ ListComponentsTool with optional filtering
- ☐ ListContentTypesTool implemented
- ☐ Registry helper methods added
- ☐ Agent uses discovery tools appropriately

Week 2: Analysis Tools

Goal

Add tools for deeper analysis: finding components with specific features, identifying gaps between systems.

Exercise

Add two analysis tools:

- **FindComponentsByFeatureTool:** Parameters: feature (string, e.g., 'async loading', 'RTL support'), systems (optional list). Searches docs for components mentioning the feature.
- **ComponentGapAnalysisTool:** Parameters: system_a, system_b. Returns components present in one system but not the other.

FindComponentsByFeature can reuse your semantic search—search for the feature, group results by component, return the component list. ComponentGapAnalysis compares component lists between systems.

Test with questions like: "Which systems have combobox with async loading?" and "What components does Radix have that Spectrum doesn't?"

Why This Matters

These tools enable questions that would require multiple searches and manual comparison. Gap analysis is particularly useful for migration planning. This is where your tool becomes genuinely useful for real research tasks.

Completion Checklist

- ☐ FindComponentsByFeatureTool implemented
- ☐ ComponentGapAnalysisTool implemented
- ☐ Tools registered in registry
- ☐ Agent uses tools for analysis questions
- ☐ Results are accurate and useful

Week 3: Export Tools

Goal

Add tools that export comparison results as reports. This lets users save findings for documentation or sharing.

Exercise

Add an export tool:

- **ExportComparisonTool:** Parameters: component_name, systems (list), format (enum: 'markdown', 'json', 'html'). Generates and saves a comparison report file.

The tool should:

- Call your comparison extraction (reuse CompareComponentsTool logic)
- Format as the requested output type
- Save to an 'exports/' directory with timestamped filename
- Return the filepath and a summary

Create formatting functions for each output type. Markdown should be readable with tables. HTML should be self-contained with basic styling. JSON should be the raw ComponentComparison data.

Why This Matters

Export transforms ephemeral chat responses into persistent artifacts. Users can reference reports later, share with teammates, or include in documentation. This is a tangible output that demonstrates the tool's value.

Completion Checklist

- ☐ ExportComparisonTool implemented
- ☐ Markdown format is readable
- ☐ HTML format is self-contained
- ☐ JSON format is valid and complete
- ☐ Files saved with sensible names
- ☐ Agent confirms export success

Week 4: Execution Tracing

Goal

Build a tracing system that records everything the agent does: messages, tool calls, results, timing. This is essential for debugging and understanding agent behavior.

Exercise

Create agent/tracing.py with:

- **TraceEvent:** dataclass with timestamp, event_type (enum: 'user_message', 'assistant_message', 'tool_call', 'tool_result', 'error'), data (dict)
- **Trace:** dataclass with trace_id, started_at, events (list[TraceEvent]), metadata (dict)

- **Tracer:** class with start_trace(), add_event(), end_trace(), export_trace(format: str)

Integrate the tracer into AgentLoop:

- Start trace when run() is called
- Log user message, each LLM response, each tool call and result
- Include timing for each event
- End trace when run() completes

Add export formats: JSON (machine-readable) and a human-readable text format that shows the conversation flow clearly.

Why This Matters

Complex agents are hard to debug without traces. When the agent gives a wrong answer, traces let you see: what tools it called, what results it got, how it reasoned. This is how you improve prompt engineering and fix tool issues.

Completion Checklist

- ☐ TraceEvent and Trace dataclasses defined
- ☐ Tracer class with all methods
- ☐ AgentLoop creates traces automatically
- ☐ Timing included in events
- ☐ JSON export works
- ☐ Human-readable export is clear

Week 5: Trace Viewer

Goal

Build a way to review traces: a CLI command or simple web viewer that makes it easy to understand what the agent did.

Exercise

Add trace management to your CLI:

- ds-compare traces list: Show recent traces with IDs and summaries
- ds-compare traces show <id>: Display a trace in readable format
- ds-compare traces export <id> --format json|html: Export trace
- Auto-save traces to a traces/ directory

For the HTML export, create a self-contained viewer with:

- Timeline showing events in order
- Expandable sections for tool calls and results
- Timing information visualized
- Basic CSS for readability (inline, no external deps)

Why This Matters

Raw trace data is hard to read. A good viewer makes debugging pleasant. HTML export is shareable—you can send a trace to someone to show agent behavior. This is a professional touch that makes the tool genuinely usable.

Completion Checklist

- ☐ traces list command shows recent traces
- ☐ traces show displays readable output
- ☐ HTML export is self-contained and readable
- ☐ Traces auto-saved after each conversation
- ☐ Timeline shows clear event flow

Week 6: Advanced System Prompt

Goal

Refine the system prompt for multi-tool scenarios. With more tools, the agent needs clearer guidance on when to use each tool and how to chain them effectively.

Exercise

Update your system prompt to include:

- Tool selection guidance: when to use each tool
- Chain suggestions: "For migration questions, first use gap_analysis, then compare specific components"
- Examples of good tool usage for common question types
- Limitations: what the agent cannot do, when to say "I don't know"

Create a prompt testing framework:

- Define 10 test questions covering different capabilities
- Run each question against different prompt versions
- Manually evaluate: Did it use the right tools? Was the answer correct?
- Iterate on prompt based on results

Why This Matters

With many tools, the model can make poor choices. Explicit guidance in the system prompt improves tool selection. Testing frameworks make prompt engineering systematic rather than ad hoc. This is how production AI systems are developed.

Completion Checklist

- ☐ System prompt updated with tool guidance
- ☐ Chain suggestions included
- ☐ 10 test questions defined
- ☐ Prompt testing framework created
- ☐ Tool selection improved after iteration
- ☐ Limitations clearly stated

Week 7: Error Recovery and Fallbacks

Goal

Add sophisticated error recovery: fallback strategies when tools fail, graceful degradation, and helpful error messages for users.

Exercise

Implement fallback strategies:

- If component extraction fails, fall back to raw search results
- If a system is unavailable, compare remaining systems and note the gap
- If export fails, return the data in the response instead
- If LLM rate-limited, wait and retry with backoff

Add user-facing error handling:

- Transform technical errors into helpful messages
- Suggest alternatives when something fails
- Never show raw stack traces in chat

Test by intentionally breaking things: disconnect network during API call, corrupt a cache file, ask about non-existent systems. Verify graceful handling.

Why This Matters

Real systems fail. Graceful degradation provides partial value even when things break. Good error messages help users understand what happened and what to try next. This is the difference between a frustrating tool and a reliable one.

Completion Checklist

- ☐ Fallback strategies implemented
- ☐ User-facing errors are helpful
- ☐ No raw stack traces in chat
- ☐ Tested with intentional failures
- ☐ Partial results returned when possible

Week 8: Final Polish and Demo

Goal

Final testing, documentation updates, and preparation of a demo showcasing the complete system.

Exercise

Run comprehensive testing:

- Test all tools with valid and invalid inputs

- Test multi-turn conversations with complex questions
- Verify traces capture everything important
- Check that caching, persistence, and logging all work together

Update documentation:

- README with complete feature list
- All CLI commands documented
- Example conversations showing capabilities
- Guide for adding new design systems

Prepare a demo script:

- 5-minute walkthrough of key capabilities
- Show discovery, search, comparison, export
- Demonstrate a multi-step research question
- Show trace viewer for transparency

Why This Matters

A polished demo demonstrates competence. Good documentation makes the tool usable by others. Comprehensive testing ensures reliability. This final week transforms a project into something you can show off and others can use.

Completion Checklist

- ☐ All tools tested thoroughly
- ☐ Documentation complete and accurate
- ☐ Demo script prepared
- ☐ Demo runs smoothly
- ☐ Example conversations documented
- ☐ Project ready to share

Phase 6 Complete — Project Complete!

By the end of Week 8, you have:

- A complete design system comparison tool
- 8+ tools covering search, comparison, discovery, analysis, and export
- Comprehensive execution tracing
- Robust error handling and fallbacks
- Production-quality code with types, logging, and caching
- Complete documentation
- A demo showcasing the system

What You've Learned

Over 12 months, you've built real Python skills through a real project: dictionaries and data structures, file I/O, async programming, decorators, type hints, protocols and interfaces, Pydantic validation, working with APIs, prompt engineering, and agentic AI patterns.

More importantly, you understand how RAG systems work from the ground up. You can read library source code with confidence. You can debug LLM applications. You can design modular systems that others can extend.

What's Next

Ideas for continued development:

- Add more design systems
- Build a web interface
- Add code example comparison (not just docs)
- Integrate with Figma for design token comparison
- Publish as an open source tool
- Apply these patterns to a completely different domain

Congratulations on completing the roadmap!