Phase 5: Tool Use Layer

Weekly Exercises — Months 9–10

Phase Overview

By the end of this phase, your comparison tool becomes something an LLM can use autonomously. You'll define tools with schemas, build a conversation loop where the model decides which tools to call, and handle the back-and-forth of tool execution.

What You'll Build

- Tool definitions with JSON schemas
- A tool registry that maps tool names to implementations
- An agentic conversation loop
- Error handling for invalid tool calls

Key Python Concepts

Async/await, JSON schema generation from Pydantic, callable objects, dynamic dispatch, conversation state management, while loops with break conditions.

Week 1: Tool Definition Model

Goal

Define the data structures for representing tools. A tool has a name, description, parameters schema, and an implementation function.

Exercise

Create src/ds_compare/agent/tools.py with these Pydantic models:

- **ToolParameter:** name, type, description, required, enum (optional list)
- **ToolDefinition:** name, description, parameters list
- **ToolCall:** name, arguments dict
- **ToolResult:** tool_name, success bool, result, error (optional)

Add methods to ToolDefinition: to_openai_schema() and to_anthropic_schema() that convert to each provider's format. This maintains modularity across LLM providers.

Why This Matters

Tools bridge LLM reasoning and real-world actions. The schema tells the model what the tool does and what arguments it needs. Provider-agnostic definitions let the same tools work with Claude or GPT-4.

Completion Checklist

- ☐ All four Pydantic models defined
- ☐ to_openai_schema() produces valid format
- ☐ to_anthropic_schema() produces valid format
- ☐ Can create and convert sample ToolDefinitions

Week 2: First Tool — search_docs

Goal

Create your first tool wrapping semantic search. This lets the LLM search design system documentation.

Exercise

Create agent/implementations.py with Tool Protocol and SearchDocsTool:

- **Tool Protocol:** definition property, execute(args: dict) -> ToolResult
- **SearchDocsTool:** wraps your SemanticSearcher

Parameters: query (required), design_system (optional), content_type (optional enum), top_k (optional, default 5). The execute method validates arguments, calls searcher, formats results as readable text, returns ToolResult.

Why This Matters

This establishes your tool pattern. Formatting results as readable strings (not JSON) helps the LLM understand the information. Good descriptions help the model know when to use each tool.

Completion Checklist

- ☐ Tool Protocol defined
- ☐ SearchDocsTool implements Tool
- ☐ Clear description and parameter definitions
- ☐ execute() returns formatted results
- ☐ Handles invalid arguments gracefully

Week 3: Compare and Get Component Tools

Goal

Add tools for component comparison and retrieval, exposing your Phase 3 extraction capabilities to the agent.

Exercise

Add two tool classes:

- **GetComponentTool:** Parameters: component_name (required), design_system (required). Returns extracted ComponentSummary as formatted text.
- **CompareComponentsTool:** Parameters: component_name (required), systems (required, list of strings). Returns ComponentComparison as formatted text.

Create a format_component_summary() helper that converts ComponentSummary to readable text: component name, description, props list with types, variants, accessibility info. Create format_comparison() for side-by-side comparison text.

Test each tool by calling execute() directly. Verify the formatted output is readable and complete.

Why This Matters

These are your core tools—the comparison capability is what makes your tool unique. Good formatting is crucial: the LLM needs to understand the results to answer user questions. Think about what information a human would need to see.

Completion Checklist

- ☐ GetComponentTool implemented and tested
- ☐ CompareComponentsTool implemented and tested
- ☐ Formatting helpers produce readable output
- ☐ Error handling for unknown components/systems
- ☐ Both tools have clear descriptions

Week 4: Tool Registry

Goal

Build a registry that manages tools and dispatches calls. This centralizes tool management and enables dynamic tool discovery.

Exercise

Create agent/registry.py with ToolRegistry class:

- register(tool: Tool) -> None: Add a tool to the registry
- get(name: str) -> Tool | None: Get a tool by name
- list_tools() -> list[ToolDefinition]: Get all tool definitions
- execute(call: ToolCall) -> ToolResult: Dispatch a tool call
- get_schemas(format: str) -> list[dict]: Get all schemas in specified format ('openai' or 'anthropic')

The execute method should: look up the tool by name, return error ToolResult if not found, call tool.execute() with arguments, catch exceptions and return error ToolResult, log all calls.

Create a create_default_registry() factory that returns a registry pre-populated with your three tools, properly initialized with your searcher and extractor.

Why This Matters

The registry is the dispatch layer between LLM tool calls and your implementations. Centralizing this logic keeps the agent loop clean. The get_schemas method makes it easy to send tool definitions to any LLM provider.

Completion Checklist

- ☐ ToolRegistry with all methods
- ☐ execute() handles unknown tools
- ☐ execute() catches and reports exceptions
- ☐ get_schemas() works for both formats
- ☐ Factory function creates usable registry

Week 5: The Conversation Loop

Goal

Build the core agent loop: send message to LLM, check for tool calls, execute tools, send results back, repeat until done.

Exercise

Create agent/loop.py with AgentLoop class:

- __init__(llm: LLMProvider, registry: ToolRegistry, system_prompt: str)
- run(user_message: str, max_iterations: int = 10) -> str
- Maintains conversation history as list of messages

The run() method implements this loop:

- 1. Add user message to history
- 2. Call LLM with history and tool schemas
- 3. If response contains tool calls: execute each, add results to history, go to step 2
- 4. If response is text only: return the text
- 5. If max_iterations reached: return partial result with warning

You'll need to extend your LLM provider interface with a method that supports tool use: complete_with_tools(messages, tools, ...) that can return either text or tool calls.

Why This Matters

This is the heart of an agent. The loop lets the model reason, act, observe, and continue until it has enough information. Managing conversation history is crucial—the model needs to see previous tool results to make good decisions.

Completion Checklist

- ☐ AgentLoop class implemented
- ☐ LLM provider extended with tool support
- ☐ Loop executes tools and continues
- ☐ Conversation history maintained correctly
- ☐ Max iterations prevents infinite loops

Week 6: Error Handling and Validation

Goal

Add robust error handling for invalid tool calls. LLMs sometimes hallucinate tool names or provide wrong argument types.

Exercise

Enhance the registry and loop with validation:

- In ToolRegistry.execute(): validate argument types against tool schema before execution
- Check required arguments are present
- Check enum values are valid
- Return helpful error messages that guide the model to correct usage

Update AgentLoop to handle tool errors gracefully:

- When tool returns error ToolResult, include error message in history
- Let the model see the error and try again
- Track consecutive errors; give up after 3 failed attempts on same tool

Test by manually crafting invalid ToolCalls and verifying the error messages help correct the issue.

Why This Matters

LLMs make mistakes. Good error messages let the model self-correct. Without validation, you'd get cryptic Python exceptions that don't help the model understand what went wrong. This is the difference between a fragile demo and a robust system.

Completion Checklist

- ☐ Argument validation before execution
- ☐ Helpful error messages for invalid calls
- ☐ Agent loop handles errors gracefully
- ☐ Consecutive error tracking prevents loops
- ☐ Model can self-correct after errors

Week 7: System Prompt and Personality

Goal

Craft the system prompt that shapes agent behavior. A good system prompt makes the agent helpful, focused, and knowledgeable about its capabilities.

Exercise

Create agent/prompts.py with a detailed system prompt that includes:

- Role: "You are a design system expert assistant..."

- Capabilities: what the agent can do (search docs, compare components)
- Available design systems: list the systems that have been ingested
- Guidelines: use search before comparison, cite sources, admit uncertainty
- Response format: be concise, use examples when helpful

Make the system prompt configurable—it should accept the list of available systems dynamically. Create a build_system_prompt(available_systems: list[str]) -> str function.

Test different prompt variations. Ask the same question with different prompts and compare response quality.

Why This Matters

The system prompt shapes everything. It determines whether the agent rambles or is concise, whether it uses tools effectively or ignores them, whether it admits limitations or hallucinates. Prompt engineering is a real skill; iteration and testing matter.

Completion Checklist

- ☐ Comprehensive system prompt written
- ☐ Dynamic system list inclusion
- ☐ Agent behaves according to guidelines
- ☐ Agent uses tools appropriately
- ☐ Response quality improved by prompt

Week 8: Interactive CLI and Testing

Goal

Add an interactive chat mode to your CLI and test the complete agent with real questions.

Exercise

Add a chat command to your CLI:

- ds-compare chat: Start interactive session
- Print welcome message with available commands (/quit, /clear, /debug)
- Read user input in a loop
- Pass to AgentLoop.run() and print response
- /debug toggle: show tool calls as they happen
- /clear: reset conversation history

Test with these questions:

- "What props does Button have in Spectrum?"
- "How does Button compare between Spectrum and Carbon?"
- "Which design systems have the best accessibility docs for form components?"
- "What's the difference between Dialog and Modal?"

Document any issues: wrong tool choices, poor responses, missing capabilities. These inform Phase 6 improvements.

Why This Matters

Interactive testing reveals real-world behavior. You'll discover edge cases, unclear tool descriptions, and response quality issues that unit tests miss. The debug mode is invaluable for understanding agent behavior.

Completion Checklist

- ☐ Interactive chat mode works
- ☐ Commands (/quit, /clear, /debug) implemented
- ☐ Agent answers test questions correctly
- ☐ Agent uses tools appropriately
- ☐ Issues documented for Phase 6
- ☐ Debug mode shows tool execution

Phase 5 Complete

By the end of Week 8, you have:

- Tool definitions with provider-agnostic schemas
- Three core tools: search, get component, compare components
- A tool registry with validation and dispatch
- An agent loop that executes tools and manages conversation
- Robust error handling for invalid tool calls
- A crafted system prompt
- Interactive chat mode for testing

You now have a working agent! Before Phase 6, spend time using it. Note which questions it handles well, which it struggles with, and what capabilities are missing. Phase 6 will add more tools and sophistication based on these observations.