Phase 3: Structured Comparison Output

Weekly Exercises — Months 5–6

Phase Overview

By the end of this phase, you'll move beyond raw search results to structured comparison data. You'll define Pydantic models that capture what matters when comparing components—props, variants, accessibility features—and use an LLM to extract this information from documentation.

What You'll Build

- Pydantic models for component comparisons (props, variants, accessibility)
- An LLM provider interface with modular implementations
- Extraction pipelines that convert documentation into structured data
- Comparison output that normalizes data across design systems

Key Python Concepts

Pydantic models and validation, Optional types and defaults, nested models, JSON schema generation, working with LLM APIs, prompt engineering basics, error handling for unreliable outputs.

Week 1: Core Comparison Models

Goal

Define the Pydantic models that represent component information. Your design system expertise shapes these models—you decide what fields matter for meaningful comparison.

Exercise

Create src/ds_compare/comparison/models.py with these Pydantic models:

- **PropDefinition:** name (str), type (str), required (bool), default (str | None), description (str | None)
- **VariantInfo:** name (str), description (str | None), is_default (bool)
- **AccessibilityInfo:** aria_role (str | None), keyboard_interactions (list[str]), screen_reader_notes (str | None)
- **ComponentSummary:** name (str), design_system (str), description (str | None), props (list[PropDefinition]), variants (list[VariantInfo]), accessibility (AccessibilityInfo | None)

- **ComponentComparison:** component_name (str), systems (list[ComponentSummary]), generated_at (datetime)

Use Field() with descriptions. Test by creating instances and verifying validation catches bad data.

Why This Matters

These models define what "comparing components" means. Pydantic validates automatically—malformed LLM output gives clear errors, not silent failures. Your domain expertise shapes the schema; a generic developer wouldn't know to include aria_role or keyboard_interactions.

Completion Checklist

- ☐ All five Pydantic models defined with type hints
- ☐ Optional used correctly for nullable fields
- ☐ Can create valid instances of each model
- ☐ Validation rejects invalid data

Week 2: The LLM Provider Interface

Goal

Define an interface for LLM providers, mirroring the embedding pattern. This keeps extraction code decoupled from specific services.

Exercise

Create src/ds_compare/llm/ with base.py defining:

- **LLMResponse:** dataclass with content (str), model (str), usage (dict with prompt_tokens, completion_tokens)
- **LLMProvider Protocol:** complete(prompt: str, system: str | None, temperature: float) -> LLMResponse
- **complete_json(prompt: str, schema: type[BaseModel], system: str | None) -> BaseModel:** Returns validated Pydantic model
- **LLMConfig:** dataclass with provider (str), model (str), api_key (str | None), temperature (float = 0.0)

The complete_json method is key—it should prompt the LLM for JSON output, parse the response, and validate against the provided Pydantic model.

Why This Matters

The complete_json method encapsulates the entire flow of getting structured data from an LLM: prompting for JSON, parsing the response, validating with Pydantic. This abstraction lets you swap between Claude, GPT-4, or local models without changing extraction code.

Completion Checklist

- ☐ LLMResponse dataclass defined
- ☐ LLMProvider Protocol with both complete methods
- ☐ LLMConfig supports multiple providers

- ☐ Type hints complete

Week 3: Anthropic LLM Provider

Goal

Implement an LLM provider using the Anthropic API. Claude excels at structured extraction tasks and following detailed instructions.

Exercise

Create llm/anthropic.py with AnthropicProvider:

- Install: pip install anthropic
- Load API key from parameter or ANTHROPIC_API_KEY env var
- Implement complete() using client.messages.create()
- For complete_json(): generate JSON schema from Pydantic model using model.model_json_schema()
- Include schema in prompt and instruct model to return only valid JSON
- Parse response and validate with the Pydantic model
- Handle JSON parsing errors with clear messages

Test by extracting a simple model (like PropDefinition) from a snippet of documentation text.

Why This Matters

This is where LLMs become useful for data extraction. The model_json_schema() method generates a schema the LLM can follow. Prompt engineering matters here—clear instructions about returning only JSON prevent the model from adding explanatory text that breaks parsing.

Completion Checklist

- ☐ AnthropicProvider implements LLMProvider
- ☐ API key loaded from env or parameter
- ☐ complete() returns LLMResponse
- ☐ complete_json() returns validated Pydantic model
- ☐ JSON parse errors handled gracefully

Week 4: OpenAI Provider and Factory

Goal

Add an OpenAI provider and create a factory function. This completes the LLM abstraction layer with two provider options.

Exercise

Create llm/openai.py with OpenAIProvider:

- Similar structure to AnthropicProvider

- Use response_format={'type': 'json_object'} for JSON mode
- Include schema in system prompt for complete_json()

In llm/__init__.py, implement:

- get_llm_provider(config: LLMConfig) -> LLMProvider factory
- Support 'anthropic' and 'openai' provider values
- Add llm section to your config.yaml

Test both providers with the same extraction task. Compare results quality.

Why This Matters

Different LLMs have different strengths. OpenAI's JSON mode guarantees valid JSON (though not schema compliance). Having both options lets users choose based on cost, quality, or preference. The factory pattern makes this transparent to the rest of your code.

Completion Checklist

- ☐ OpenAIProvider implements LLMProvider
- ☐ Uses JSON mode for reliable parsing
- ☐ Factory function returns correct provider
- ☐ Config file controls provider selection
- ☐ Both providers produce valid Pydantic models

Week 5: Component Extraction Pipeline

Goal

Build the extraction pipeline that converts documentation chunks into ComponentSummary objects. This is where retrieval meets structured output.

Exercise

Create comparison/extraction.py with ComponentExtractor class:

- __init__ takes LLMProvider and Searcher
- extract_component(name: str, design_system: str) -> ComponentSummary
- Internally: search for relevant chunks, concatenate content, call LLM with extraction prompt

Write a detailed extraction prompt that:

- Explains the task: extract component information from documentation
- Provides the documentation content
- Includes the JSON schema
- Specifies to return ONLY valid JSON, no explanations
- Handles missing information gracefully (use null for unknown fields)

Test by extracting Button from one design system. Verify the output matches the actual documentation.

Why This Matters

This is the core of turning unstructured docs into structured data. Prompt engineering is critical—clear instructions reduce extraction errors. Combining search results with LLM extraction is a classic RAG pattern: retrieve context, then generate structured output based on that context.

Completion Checklist

- ☐ ComponentExtractor class implemented
- ☐ Searches for relevant chunks before extraction
- ☐ Extraction prompt is clear and detailed
- ☐ Returns valid ComponentSummary
- ☐ Extracted data matches source documentation

Week 6: Multi-System Comparison

Goal

Build the comparison function that extracts component information from multiple systems and returns a unified ComponentComparison.

Exercise

Add to ComponentExtractor:

- compare_component(name: str, systems: list[str]) -> ComponentComparison
- Call extract_component for each system
- Handle cases where a component doesn't exist in a system (skip with warning)
- Assemble results into ComponentComparison with timestamp
- Add progress reporting (extracting from system X...)

Handle component name normalization—the same component might be called 'Dialog' in one system and 'Modal' in another. For now, accept aliases as a parameter: compare_component(name: str, systems: list[str], aliases: dict[str, str] | None = None).

Test by comparing Button across Spectrum and Carbon. Verify both summaries are populated.

Why This Matters

This is the core feature of your tool—comparing components across systems. The aliases parameter acknowledges that naming varies; you'll expand this into smarter normalization later. Graceful handling of missing components prevents the whole comparison from failing when one system lacks a component.

Completion Checklist

- ☐ compare_component method implemented
- ☐ Extracts from multiple systems
- ☐ Handles missing components gracefully
- ☐ Alias parameter works for name variations
- ☐ Returns valid ComponentComparison with multiple summaries

Week 7: Validation and Error Handling

Goal

Add robust error handling for LLM extraction failures. LLMs are unreliable—sometimes they return malformed JSON or miss required fields. Good error handling makes the difference between a demo and a usable tool.

Exercise

Create comparison/errors.py with custom exceptions:

- **ExtractionError:** Base exception for extraction failures
- **JSONParseError:** LLM returned invalid JSON
- **ValidationError:** JSON valid but doesn't match schema
- **ComponentNotFoundError:** No relevant chunks found for component

Update ComponentExtractor with retry logic:

- On JSON parse failure, retry up to 3 times with more explicit prompt
- On validation failure, log which fields failed and retry
- After max retries, raise appropriate exception with context
- Log all extraction attempts for debugging

Why This Matters

LLMs fail in predictable ways. Sometimes they add explanatory text before JSON. Sometimes they hallucinate fields. Retry logic with clearer prompts often succeeds on the second attempt. Custom exceptions let calling code handle different failure modes appropriately. This is essential for production AI systems.

Completion Checklist

- ☐ Custom exception classes defined
- ☐ Retry logic implemented with max attempts
- ☐ Retries use clearer prompts
- ☐ Failures logged with context
- ☐ Exceptions include useful debugging information

Week 8: CLI and Integration Test

Goal

Add comparison commands to your CLI and run a full integration test. This validates the entire Phase 3 pipeline.

Exercise

Add commands to your CLI:

- **ds-compare extract <component> --system <name>:** Extract single component, output JSON
- **ds-compare compare <component> --systems <list>:** Compare across systems, output formatted comparison
- **--output <file>:** Optional flag to save JSON to file

For the compare output, create a simple text formatter that shows a side-by-side comparison of props, variants, and accessibility features.

Run integration test: compare Button across Spectrum, Carbon, and Shoelace. Verify:

- All three systems have extracted data
- Props include expected entries (variant, disabled, size, etc.)
- Accessibility info populated where available
- JSON output is valid and parseable

Why This Matters

This is the payoff: you can now run a single command to compare how different design systems implement the same component. The JSON output enables programmatic use—feeding into reports, dashboards, or further analysis. The integration test validates that every component from Phase 1–3 works together.

Completion Checklist

- ☐ extract command works
- ☐ compare command works with multiple systems
- ☐ JSON output option saves to file
- ☐ Text formatter shows readable comparison
- ☐ Integration test passes with 3 design systems
- ☐ Extracted data is accurate against source docs

Phase 3 Complete

By the end of Week 8, you have:

- Pydantic models defining component comparison structure
- Modular LLM provider interface with Anthropic and OpenAI implementations
- Extraction pipeline that converts docs to structured data
- Multi-system comparison capability
- Robust error handling for LLM failures
- CLI commands for extraction and comparison

Before moving to Phase 4, run several comparisons and evaluate extraction quality. Note where the LLM misses information or makes errors—this feedback will inform prompt improvements. Try comparing at least 5 different components across your ingested systems.