Design System Comparison Tool

A 12-Month Python & AI Learning Roadmap

Overview

This roadmap guides you through building a design system comparison tool—a searchable knowledge base that lets you explore component documentation across multiple public design systems and compare implementations side by side.

Each phase builds on previous work, reinforcing Python fundamentals while teaching AI/ML patterns. By working with design system content you already understand deeply, you'll develop better intuition for chunking strategies, metadata design, and query patterns.

Time commitment: 1–2 hours per week

By the end: You'll have a working agent that can search design system documentation, compare component implementations across systems, and return structured data about APIs, props, variants, and accessibility features.

Target Design Systems

Start with 3–4 systems to keep scope manageable. Expand as the project matures. Consider systems with different characteristics:

1. **Adobe Spectrum** — Your home turf; deep familiarity with component patterns and documentation structure
2. **IBM Carbon** — Enterprise-focused, strong accessibility documentation, React-based
3. **Radix UI** — Headless/unstyled primitives, different architectural philosophy
4. **Shoelace** — Web components, framework-agnostic, similar to Spectrum Web Components
5. **Material UI / Material Web** — Widely adopted, good baseline for comparison

You can swap these based on what you find most useful. The key is variety in architecture (React vs. web components vs. headless) and documentation style.

Phase 1: The Component Registry

Months 1–2

Project Goal

Build a system that ingests documentation from multiple design systems and creates a structured registry. Each chunk knows which system it comes from, which component it describes, and what type of content it contains (API reference, usage guidelines, code examples, accessibility notes).

Domain Decisions

This phase requires design system expertise to answer: What's the right chunk size for component docs? Should props be chunked separately from usage examples? How do you handle components with different names across systems (Dialog vs. Modal vs. AlertDialog)? Start simple—one markdown file per component—and refine based on what you learn.

Python Concepts

Concept	AI/RAG Connection
Dictionaries	Your registry maps chunk IDs to rich metadata: {system: 'spectrum', component: 'button', content_type: 'api', props: [...]}. Nested dictionaries model the hierarchical nature of design system documentation.
Nested data structures	Component documentation has natural hierarchy: system → component → section → content. Comfort navigating nested structures prepares you for LLM API responses, which follow similar patterns.
File I/O and pathlib	Ingesting docs from multiple systems means traversing directory structures, reading markdown/MDX files, and handling different file organizations. pathlib makes cross-platform file handling clean.
String manipulation & regex	Extracting component names from filenames, parsing frontmatter, splitting on headings, identifying code blocks—text processing is central to document ingestion pipelines.

Learning Objectives

1. Ingest markdown documentation from 3+ design systems
2. Implement chunking logic appropriate for component documentation
3. Build a registry with rich metadata (system, component, content type)
4. Filter chunks by metadata (e.g., all Button docs, all accessibility content)

AI Terminology

- **Chunk:** A segment of documentation that serves as the unit of retrieval. For design systems, this might be a component's prop table, a usage example, or accessibility guidelines.

- **Metadata:** Structured information about a chunk used for filtering and attribution. Critical for comparison queries: "show me Button from Spectrum vs. Carbon."
- **Ingestion pipeline:** The process of reading, processing, and storing documents. Your pipeline transforms raw markdown into searchable, attributed chunks.

Success Criteria

You can run a script that ingests Button documentation from three design systems and outputs a registry where you can query: "Give me all chunks from Spectrum" or "Give me all API reference chunks for Button."

Phase 2: Semantic Search Layer

Months 3–4

Project Goal

Add embedding support so you can search semantically. "How does this system handle focus management?" should find relevant content even if it doesn't use that exact phrase. Build batch processing to embed your full registry efficiently.

Domain Decisions

Design system terminology varies: "variant" vs. "appearance" vs. "style," "disabled" vs. "isDisabled." Semantic search handles this naturally—queries about "button states" will match content about "disabled," "loading," "pressed" even without keyword overlap. This is where RAG shines over keyword search.

Python Concepts

Concept	AI/RAG Connection
List comprehensions	Extracting text from chunks, filtering by system or component, reshaping API responses—these operations happen constantly. Comprehensions like [c['text'] for c in chunks if c['system'] == 'spectrum'] become second
Generators	As you add more design systems, memory becomes a constraint. Generators let you yield chunks in batches for embedding without loading everything at once—essential for scaling.
Batch processing	Embedding APIs have rate limits and token limits per request. You'll batch chunks intelligently—maybe by system or by component—to maximize throughput while staying within constraints.
Working with vectors	Embeddings are just lists of floats. Computing cosine similarity, finding top-k matches, and merging vectors back into your registry uses basic NumPy or pure Python math.

Learning Objectives

1. Rewrite loops as list comprehensions with filtering
2. Build a generator that yields chunks in configurable batch sizes
3. Embed all chunks and store vectors in your registry
4. Implement semantic search with cosine similarity

AI Terminology

- **Embedding:** A dense vector representing semantic meaning. Similar concepts have similar embeddings, enabling "fuzzy" matching beyond keywords.
- **Cosine similarity:** A measure of how similar two vectors are, ranging from -1 to 1. Higher values mean more similar content.
- **Top-k retrieval:** Finding the k most similar chunks to a query. You'll tune k based on how much context fits in your LLM's context window.
- **Token:** The unit of text that models process; roughly 4 characters or 0.75 words. Relevant for both embedding costs and context limits.

Success Criteria

You can search "keyboard navigation patterns" and get relevant results from multiple systems, even if they use different terminology. You can filter results by system or content type.

Phase 3: Structured Comparison Output

Months 5–6

Project Goal

Move beyond raw search results to structured comparison data. Define Pydantic models for component comparisons—normalized views that capture props, variants, accessibility features, and framework support across systems. Use an LLM to extract and structure this information from retrieved chunks.

Domain Decisions

Your design system expertise shapes the schema: What fields matter for comparison? Props and their types, certainly. But also: Does it support controlled/uncontrolled patterns? What ARIA roles does it use? Does it handle RTL? Is it composable or monolithic? You're defining what "comparing components" actually means.

Python Concepts

Concept	AI/RAG Connection
Pydantic models	Define ComponentComparison, PropDefinition, AccessibilityInfo models. Pydantic ensures the LLM returns exactly the structure your code expects—or raises a validation error you can handle.
Type hints	Complex nested models (list of systems, each with list of props, each with type info) require careful typing. This prepares you to read library source code and catch bugs early.
Structured output / Instructor	Libraries like Instructor wrap LLM APIs to guarantee schema-compliant responses. You pass your Pydantic model, and it handles prompting, parsing, and retries.
Optional fields and defaults	Not every system documents every field. Pydantic's Optional types and default values let you handle incomplete data gracefully—critical when comparing systems with different documentation depth.

Learning Objectives

1. Define Pydantic models for component comparison (props, variants, a11y)
2. Use Instructor or JSON mode to extract structured data from docs
3. Handle missing or inconsistent data across systems
4. Validate and normalize component names across systems

AI Terminology

- **Schema:** A formal definition of expected data structure. Your ComponentComparison schema defines what a "comparison" actually contains.
- **Structured output:** LLM responses constrained to match a predefined format. Essential for downstream code that expects specific fields.
- **Extraction:** Using an LLM to pull structured information from unstructured text. You're extracting prop definitions from prose documentation.
- **Normalization:** Converting varied representations to a common format. "isDisabled" and "disabled" both become a standard disabled property in your schema.

Success Criteria

Given "compare Button across systems," you get a validated ComponentComparison object with normalized prop lists, variant options, and accessibility notes for each system. Missing data is handled gracefully with Optional fields.

Phase 4: Polish and Instrumentation

Months 7–8

Project Goal

Make your code production-quality: add comprehensive type hints, write decorators for logging and timing, refactor into a clean module structure. Add caching so repeated queries don't re-embed or re-extract.

Domain Decisions

Component comparisons are expensive (multiple LLM calls for extraction). Caching results by component + systems makes sense. But cache invalidation matters too—when you update your Spectrum docs, cached comparisons involving Spectrum should refresh. Your module structure might separate: ingestion, embedding, retrieval, extraction, caching.

Python Concepts

Concept	AI/RAG Connection
Decorators	@timed for latency tracking, @cached for memoization, @retry for API failures. Decorators add these behaviors cleanly without cluttering function bodies—essential for LLM applications.
Context managers	Managing resources safely: database connections for your registry, file handles for cache, API sessions. Context managers handle cleanup even when errors occur.
Module organization	Separate concerns: ingestion/, embedding/, retrieval/, extraction/, cache/. Clean structure makes code maintainable as you add systems and features.
Caching strategies	functools.lru_cache for simple memoization, or disk-based caching for persistence. Understanding cache keys and invalidation prevents stale data bugs.

Learning Objectives

1. Write decorators for timing, caching, and retry logic
2. Add type hints to all functions with proper return types
3. Refactor into separate modules with clear responsibilities
4. Implement caching for expensive operations (embedding, extraction)

AI Terminology

- **Latency:** Time from request to response. Comparison queries involve multiple steps; knowing which step is slow helps optimization.
- **Memoization:** Caching function results by input. A comparison for Button across [Spectrum, Carbon] can be cached and reused.
- **Retry with backoff:** Automatically retrying failed API calls with increasing delays. Essential for unreliable external services.
- **Observability:** The ability to understand system behavior through logs, metrics, and traces. Prepares you for tools like LangSmith.

Success Criteria

Your project is organized into multiple modules. A @timed decorator logs execution duration. A @cached decorator avoids redundant LLM calls. Repeated queries are fast. All functions have type hints.

Phase 5: Tool Use Layer

Months 9–10

Project Goal

Turn your comparison tool into something an LLM can use autonomously. Define tools with schemas—search_docs, get_component, compare_components—and build a conversation loop where the model decides which to call based on user questions.

Domain Decisions

What tools make sense? Perhaps: search_docs(query, systems?, content_type?) for open-ended search; get_component(name, system) for targeted retrieval; compare_components(name, systems) for structured comparison; list_components(system) for discovery. Your domain knowledge shapes which capabilities are useful.

Python Concepts

Concept	AI/RAG Connection
Async/await	Tool calling involves multiple LLM roundtrips and external calls. Async code lets you handle concurrent operations—like fetching from multiple systems in parallel.
Function signatures as schemas	Tools are defined by their interface: name, description, parameters with types. The LLM uses this schema to decide when and how to call your functions.
Control flow with LLM decisions	The model decides whether to call a tool, which tool, and what arguments. Your code handles this dynamic flow, including unexpected decisions or invalid calls.
Error handling in loops	Models can request invalid calls (wrong component name, unsupported system). Robust error handling keeps the conversation going—maybe suggesting valid options.

Learning Objectives

1. Define tool schemas for search, retrieval, and comparison
2. Build an async conversation loop that handles tool calls
3. Execute tool calls and return results to the model
4. Handle invalid requests gracefully (unknown component, unsupported system)

AI Terminology

- **Tool / Function calling:** LLM capability to request execution of external functions. The model outputs structured tool requests; your code executes them.
- **Tool schema:** JSON definition of a tool's interface. Good descriptions help the model choose the right tool for a given question.
- **Agentic loop:** Iterative process: reason → call tool → observe result → continue. The loop runs until the model has enough information to answer.
- **Grounding:** Connecting LLM responses to retrieved information. Your tool results "ground" the model's answer in actual documentation.

Success Criteria

You can ask "How does Spectrum's Button compare to Carbon's?" The model calls compare_components, your code executes, and the model formulates an answer grounded in actual documentation. Invalid requests ("compare FooBar component") are handled gracefully.

Phase 6: Multi-Tool Agent

Months 11–12

Project Goal

Expand into a capable design system research assistant. Add tools for discovery (list all components, list all systems), analysis (find components with specific features), and export (generate comparison reports). Build comprehensive tracing to understand agent behavior.

Domain Decisions

Advanced queries become possible: "Which systems have a Combobox that supports async loading?" "Find all components in Radix that don't exist in Spectrum." "Generate a migration checklist from Material to Carbon." These require combining tools creatively—the model orchestrates multi-step research workflows.

Python Concepts

Concept	AI/RAG Connection
Tool routing	With 5+ tools, you need clean dispatch logic. A tool registry mapping names to callables keeps the code organized as capabilities grow.
Pre-execution validation	Before executing, validate arguments: Is this a known system? Does this component exist? Catching errors early provides better feedback to the model.
State management	Multi-turn research accumulates context. Tracking conversation history, tool results, and intermediate findings—often with dataclasses—prevents state bugs.
Tracing and debugging	Complex agents are hard to debug. Logging each decision, tool call, and result creates traces you can analyze when the agent gives unexpected answers.

Learning Objectives

1. Implement 5+ tools covering search, retrieval, comparison, discovery, and export
2. Build a tool registry that routes calls to the correct function
3. Validate tool calls before execution with helpful error messages

4. Create comprehensive traces of agent research sessions

AI Terminology

- **Agent:** A system that uses an LLM to reason about tasks, decide which tools to use, and iterate until complete. Your design system assistant is an agent.
- **Tool chaining:** Using output from one tool as input to another. "List Radix components" → "Compare each to Spectrum" is a chain.
- **ReAct pattern:** Reasoning + Acting. The model explains its plan, acts, observes results, updates its reasoning. Visible in traces.
- **Trace:** A complete record of agent execution: prompts, responses, tool calls, results. Essential for debugging and improvement.

Success Criteria

You have a working design system research assistant that can answer complex questions like "Which systems have the most complete accessibility documentation for form components?" You can review a trace of any session to understand exactly what the agent did and why.

Session Guide

Each 1–2 hour session should feel productive. Here's a template:

Typical Session Structure

- **First 10 minutes:** Review where you left off. Read your last code. Recall what worked and what didn't.
- **Next 60–90 minutes:** Write code. When stuck, ask Claude to explain concepts or review your approach—but write the code yourself.
- **Last 10 minutes:** Note what you learned, what's still confusing, and what to tackle next session.

Productive Questions to Ask Claude

- "Can you review this function and suggest more idiomatic Python?"
- "I'm getting this error—what's happening and how would you debug it?"
- "What's the difference between X and Y in this context?"
- "How would a production RAG system handle this differently?"
- "What chunking strategy would work best for component API docs?"
- "Show me an example of [concept] in the context of design system documentation."

Design System–Specific Questions

Your domain expertise will surface questions that pure AI tutorials miss:

- "How should I chunk a component page that has props, slots, events, and CSS custom properties?"
- "What metadata is most useful for filtering—component category? Framework? Maturity level?"
- "How do I normalize prop names across systems that use different conventions?"
- "What's a good schema for representing component variants across systems?"