Phase 1: The Component Registry

Weekly Exercises — Months 1–2

Phase Overview

By the end of this phase, you'll have a working ingestion system that can read design system documentation from multiple sources (local files, Git repos, web crawling) and produce a structured registry of chunks with rich metadata. You'll also establish the modular architecture that the rest of the project builds on.

What You'll Build

- A document loader interface with implementations for local files, Git repos, and web crawling
- A chunking system that breaks component documentation into retrievable pieces
- A registry that stores chunks with metadata (system, component, content type)
- Configuration system for adding new design systems

Key Python Concepts

Dictionaries and nested data structures, file I/O with pathlib, string manipulation, abstract base classes (ABC), Protocol classes, dataclasses, basic project structure.

Week 1: Project Structure and Core Types

Goal

Set up the project structure and define the core data types that will flow through your entire system. This establishes the vocabulary your code uses to talk about documents, chunks, and metadata.

Exercise

Create a new Python project with the following structure:

```
ds-compare/

  src/

    ds_compare/

      __init__.py
      models/
        __init__.py
        documents.py
      loaders/
        __init__.py
      chunkers/
        __init__.py
      registry/
        __init__.py
  tests/
  pyproject.toml
```

In documents.py, define dataclasses for your core types:

- **Document:** Represents a raw document before chunking. Fields: id, content, source_path, metadata (dict)
- **Chunk:** Represents a piece of a document. Fields: id, content, document_id, metadata (dict), start_index, end_index
- **DesignSystemInfo:** Metadata about a design system. Fields: name, version (optional), url (optional), framework (optional)
- **ComponentInfo:** Metadata about a component. Fields: name, category (optional), aliases (list of alternative names)

Why This Matters

Dataclasses give you typed, immutable data structures with minimal boilerplate. Every RAG system needs clear definitions of what a "document" and "chunk" are. Defining these upfront prevents confusion later when you're passing data between modules. The metadata dict keeps things flexible—you can add fields without changing the class definition.

Completion Checklist

- ☐ Project structure created with all directories
- ☐ pyproject.toml configured with project name and Python version
- ☐ All four dataclasses defined with type hints
- ☐ Can import from ds_compare.models.documents in a Python REPL
- ☐ Can create instances of each dataclass and access their fields

Week 2: The Loader Interface

Goal

Define an abstract interface for document loaders. This establishes the contract that all loaders must follow, enabling you to swap implementations without changing the rest of your code.

Exercise

In loaders/__init__.py, define a Protocol (or ABC) called DocumentLoader with the following methods:

- **load() -> list[Document]:** Returns all documents from the source
- **load_one(path: str) -> Document:** Loads a single document by path/identifier
- **supports(source: str) -> bool:** Returns True if this loader can handle the given source

Also define a LoaderConfig dataclass that holds configuration common to all loaders: base_path, file_extensions (list), recursive (bool), design_system (DesignSystemInfo).

Write a simple test that verifies you cannot instantiate the abstract DocumentLoader directly (if using ABC) or that a class missing required methods doesn't satisfy the Protocol.

Why This Matters

Interfaces (via Protocol or ABC) are how you achieve modularity in Python. By defining what a loader must do without specifying how, you enable multiple implementations (local files, Git, web) to be used interchangeably. This is the foundation of the "swap your embedding model" flexibility you want.

Completion Checklist

- ☐ DocumentLoader Protocol or ABC defined with all three methods
- ☐ LoaderConfig dataclass defined with all fields
- ☐ Type hints on all methods and fields
- ☐ Test verifies the interface cannot be used directly

Week 3: Local File Loader

Goal

Implement your first concrete loader that reads markdown and text files from a local directory. This is the simplest loader and will handle cloned Git repos and manually downloaded docs.

Exercise

Create loaders/local.py with a LocalFileLoader class that implements DocumentLoader:

- Accept a LoaderConfig in __init__
- Use pathlib.Path to traverse the directory structure
- Filter files by the configured extensions (.md, .mdx, .txt)
- Support recursive directory traversal based on config
- Generate unique document IDs (consider using the file path hash)

- Populate document metadata with: filename, relative_path, extension, design_system info from config

Test by creating a small test_docs/ folder with 3–4 markdown files and verifying that load() returns the expected Documents.

Why This Matters

pathlib is the modern, cross-platform way to handle file paths in Python. You'll use it constantly. This exercise also introduces the pattern of reading configuration, iterating over files, and transforming raw content into your domain objects (Documents). The metadata population is crucial—without good metadata, you can't filter or attribute chunks later.

Completion Checklist

- ☐ LocalFileLoader class implements all DocumentLoader methods
- ☐ Recursive traversal works correctly
- ☐ File extension filtering works
- ☐ Each Document has populated metadata including design_system
- ☐ Test with sample files passes

Week 4: Git Repository Loader

Goal

Implement a loader that clones a Git repository and extracts documentation files. This enables ingesting design systems directly from their source repos without manual downloading.

Exercise

Create loaders/git.py with a GitRepoLoader class:

- Extend LoaderConfig with GitLoaderConfig adding: repo_url, branch (default: main), sparse_paths (optional list of paths to checkout)
- Clone the repo to a temporary directory (use tempfile module)
- Reuse LocalFileLoader internally to actually read the files
- Add repo URL and commit hash to document metadata
- Clean up the temp directory when done (consider using a context manager)

For Git operations, you can use subprocess to call git directly or use the gitpython library. Subprocess is simpler and has no dependencies.

Test by loading docs from a public design system repo like Shoelace (https://github.com/shoelace-style/shoelace) targeting their docs/ directory.

Why This Matters

Composition over inheritance: GitRepoLoader doesn't duplicate LocalFileLoader's logic—it delegates to it. This is a key pattern. The tempfile module and context managers ensure cleanup even if errors

occur. Adding commit hash to metadata enables reproducibility—you can trace exactly which version of docs produced a given chunk.

Completion Checklist

- ☐ GitRepoLoader clones repos successfully
- ☐ Delegates to LocalFileLoader for file reading
- ☐ Metadata includes repo_url and commit_hash
- ☐ Temp directory is cleaned up after use
- ☐ Successfully loads docs from a real design system repo

Week 5: Web Crawler Loader

Goal

Implement a loader that crawls a documentation website and extracts content. This handles design systems that don't have easily accessible source files or where the rendered docs differ from the source markdown.

Exercise

Create loaders/web.py with a WebCrawlerLoader class:

- Extend config with WebLoaderConfig: start_url, allowed_domains (list), max_pages (int), url_patterns (list of regex patterns to include)
- Use requests to fetch pages and BeautifulSoup to parse HTML
- Extract main content (look for <main>, <article>, or role="main")
- Convert HTML to plain text or markdown (consider html2text library)
- Follow links within allowed_domains up to max_pages
- Add rate limiting (time.sleep between requests)
- Metadata: url, title (from <title>), crawl_timestamp

Test by crawling a few pages from a design system's documentation site. Start with max_pages=5 to keep it manageable.

Why This Matters

Web scraping is messy but sometimes necessary. You'll learn to handle HTML parsing, link extraction, and the ethics of crawling (rate limiting, respecting robots.txt). The url_patterns config lets you focus on component pages rather than crawling entire sites. This loader completes your "ingest from anywhere" capability.

Completion Checklist

- ☐ WebCrawlerLoader fetches and parses HTML pages
- ☐ Extracts main content, not navigation/footers
- ☐ Follows links within constraints (domain, max_pages, patterns)
- ☐ Rate limiting prevents hammering servers
- ☐ Metadata includes URL and crawl timestamp

Week 6: The Chunker Interface and Basic Implementation

Goal

Define the chunking interface and implement a basic text splitter. Chunking is where your design system knowledge starts to matter—how you split docs affects retrieval quality.

Exercise

In chunkers/__init__.py, define a Chunker Protocol:

- **chunk(document: Document) -> list[Chunk]:** Splits a document into chunks
- **chunk_many(documents: list[Document]) -> list[Chunk]:** Convenience method for multiple documents

Create chunkers/text.py with a BasicTextChunker that:

- Accepts config: chunk_size (characters), chunk_overlap (characters)
- Splits on paragraph boundaries (double newlines) when possible
- Falls back to sentence boundaries, then character boundaries
- Maintains overlap between chunks for context continuity
- Generates unique chunk IDs (document_id + position)
- Copies document metadata to each chunk, adds chunk-specific metadata (position, total_chunks)

Why This Matters

Chunking strategy dramatically affects retrieval quality. Too small and you lose context; too large and you waste embedding space on irrelevant content. Overlap helps when important information spans chunk boundaries. This basic chunker is your fallback—next week you'll build a smarter one for component docs specifically.

Completion Checklist

- ☐ Chunker Protocol defined
- ☐ BasicTextChunker respects chunk_size limits
- ☐ Prefers splitting on paragraph/sentence boundaries
- ☐ Overlap works correctly
- ☐ Chunk metadata includes position and parent document info
- ☐ Test with sample documents produces expected chunks

Week 7: Component-Aware Chunker

Goal

Implement a chunker that understands the structure of component documentation. This leverages your design system expertise—you know that props tables, usage examples, and accessibility notes are distinct sections that should probably stay together.

Exercise

Create chunkers/component.py with a ComponentDocChunker that:

- Parses markdown headings to identify sections (## Props, ## Usage, ## Accessibility, etc.)
- Keeps each major section as a separate chunk when under size limit
- Falls back to BasicTextChunker for oversized sections
- Extracts component name from the document (first H1, or filename)
- Adds content_type to chunk metadata: 'props', 'usage', 'accessibility', 'examples', 'overview', 'other'
- Identifies code blocks and optionally keeps them with their surrounding context

Test with actual Button documentation from Spectrum or another system. Verify that the chunks make sense—props stay together, examples stay together.

Why This Matters

This is where domain knowledge creates better AI systems. A generic chunker doesn't know that splitting a props table in half is bad. You do. The content_type metadata enables queries like "show me accessibility docs for Button across all systems"—critical for your comparison tool.

Completion Checklist

- ☐ ComponentDocChunker parses markdown heading structure
- ☐ Major sections become individual chunks
- ☐ content_type metadata is populated correctly
- ☐ Component name is extracted
- ☐ Falls back to BasicTextChunker for large sections
- ☐ Real component docs produce sensible chunks

Week 8: The Registry

Goal

Build the registry that stores chunks and enables querying by metadata. This completes Phase 1—you'll have a working ingestion pipeline from source to searchable registry.

Exercise

Create registry/base.py with a ChunkRegistry Protocol:

- **add(chunks: list[Chunk]) -> None:** Add chunks to the registry
- **get(chunk_id: str) -> Chunk | None:** Retrieve a specific chunk
- **query(filters: dict) -> list[Chunk]:** Find chunks matching metadata filters
- **list_values(field: str) -> list[str]:** Get unique values for a metadata field
- **count() -> int:** Total chunks in registry

Implement registry/memory.py with an InMemoryRegistry using dictionaries. The query method should support filtering by any metadata field: query({"design_system": "spectrum", "content_type": "props"}).

Write an integration test that: loads docs from a local folder using LocalFileLoader, chunks them with ComponentDocChunker, adds them to InMemoryRegistry, and queries for all "props" chunks from a specific system.

Why This Matters

The registry is your system's memory. Good metadata + flexible querying enables the comparison features you want. Starting with an in-memory implementation keeps things simple; you could later swap in a SQLite or vector database implementation without changing the rest of your code—that's the power of the Protocol interface.

Completion Checklist

- ☐ ChunkRegistry Protocol defined with all methods
- ☐ InMemoryRegistry implements the Protocol
- ☐ Metadata filtering works correctly
- ☐ list_values returns unique values
- ☐ Integration test passes: load → chunk → store → query
- ☐ Can query chunks by design_system, component, and content_type

Phase 1 Complete

By the end of Week 8, you have:

- A modular architecture with clear interfaces (DocumentLoader, Chunker, ChunkRegistry)
- Three loader implementations covering local files, Git repos, and web crawling
- Two chunker implementations: basic text splitting and component-aware chunking
- A registry that stores chunks with rich metadata and supports flexible queries
- An end-to-end pipeline: source → documents → chunks → registry

Before moving to Phase 2, ingest documentation from at least two design systems (e.g., Spectrum and Shoelace) and verify you can query chunks by system, component, and content type. This corpus will be the foundation for semantic search in the next phase.