Phase 2: Semantic Search Layer

Weekly Exercises — Months 3–4

Phase Overview

By the end of this phase, you'll be able to search your design system documentation semantically. "How does this system handle focus management?" will find relevant content even if it doesn't use that exact phrase. You'll also establish the modular embedding interface that lets users swap providers.

What You'll Build

- An embedding provider interface with implementations for OpenAI and local models
- Batch processing for efficient embedding of large document collections
- Vector storage integrated with your chunk registry
- Semantic search with metadata filtering

Key Python Concepts

List comprehensions, generators and yield, working with NumPy arrays, async basics (for API calls), environment variables for configuration, batch processing patterns.

Week 1: The Embedding Provider Interface

Goal

Define the interface for embedding providers. This abstraction lets users choose between OpenAI, local models, or any other embedding service without changing application code.

Exercise

Create a new module structure for embeddings:

```
src/ds_compare/
  embeddings/
    __init__.py
    base.py
```

`openai.py`

`local.py`

In base.py, define:

- **EmbeddingResult:** A dataclass with fields: text (str), vector (list[float]), model (str), dimensions (int)
- **EmbeddingProvider Protocol:** With methods embed(text: str) -> EmbeddingResult and embed_many(texts: list[str]) -> list[EmbeddingResult], plus properties model_name (str) and dimensions (int)

Also define an EmbeddingConfig dataclass with: provider (str), model (str), api_key (optional str), base_url (optional str for self-hosted), batch_size (int, default 100).

Why This Matters

This interface is what makes your tool truly modular. Someone using your tool can configure it to use OpenAI's text-embedding-3-small, or a local sentence-transformers model, or Anthropic's embeddings when available—all by changing config, not code. The dimensions property is important because different models produce different vector sizes, and your storage needs to know this.

Completion Checklist

- ☐ EmbeddingResult dataclass defined with all fields
- ☐ EmbeddingProvider Protocol defined with methods and properties
- ☐ EmbeddingConfig dataclass supports multiple provider types
- ☐ All type hints in place

Week 2: Local Embedding Provider

Goal

Implement an embedding provider using sentence-transformers, a local model that runs without API calls. This is free, private, and great for development and testing.

Exercise

In local.py, create a SentenceTransformerProvider class:

- Install sentence-transformers: pip install sentence-transformers
- Accept model name in __init__ (default: 'all-MiniLM-L6-v2'—small, fast, good quality)
- Load the model lazily (on first embed call, not in __init__)
- Implement embed() using model.encode()
- Implement embed_many() efficiently—sentence-transformers handles batching internally
- Return vectors as Python lists (convert from numpy arrays)

Test by embedding a few sample texts and verifying: vectors have the expected dimensions (384 for MiniLM), similar texts have higher cosine similarity than dissimilar texts.

Why This Matters

Local models are invaluable for development—no API costs, no rate limits, no network latency. Lazy loading is a pattern you'll see often; it avoids slow startup when the model isn't needed. Converting numpy arrays to lists keeps your data structures simple and JSON-serializable.

Completion Checklist

- ☐ SentenceTransformerProvider implements EmbeddingProvider
- ☐ Model loads lazily on first use
- ☐ embed() returns correct EmbeddingResult
- ☐ embed_many() works for lists of texts
- ☐ Vectors are Python lists, not numpy arrays
- ☐ Similar texts produce similar vectors (test with cosine similarity)

Week 3: OpenAI Embedding Provider

Goal

Implement an embedding provider using OpenAI's API. This gives access to higher-quality embeddings and demonstrates working with external APIs, including authentication and error handling.

Exercise

In openai.py, create an OpenAIEmbeddingProvider class:

- Install openai: pip install openai
- Accept api_key and model in __init__ (default model: 'text-embedding-3-small')
- Load API key from parameter, then fall back to OPENAI_API_KEY environment variable
- Implement embed() using the OpenAI client
- Implement embed_many() using the batch endpoint (accepts up to 2048 texts)
- Handle API errors gracefully—catch exceptions, provide meaningful error messages
- Parse the API response to extract vectors

Create a .env.example file showing required environment variables. Add .env to .gitignore if not already present.

Why This Matters

Working with external APIs is fundamental to AI development. You'll learn environment variable management (keeping secrets out of code), API client usage, and error handling. The fallback pattern (parameter → env var) is common and user-friendly. OpenAI's embeddings are high quality, making this a good production option.

Completion Checklist

- ☐ OpenAIEmbeddingProvider implements EmbeddingProvider
- ☐ API key loaded from parameter or environment
- ☐ embed() works with OpenAI API
- ☐ embed_many() uses batch endpoint
- ☐ API errors produce helpful error messages
- ☐ .env.example documents required variables

Week 4: Provider Factory and Configuration

Goal

Create a factory function that instantiates the correct embedding provider based on configuration. This completes the modularity story—users configure which provider they want, and your code handles the rest.

Exercise

In embeddings/__init__.py, implement:

- A get_embedding_provider(config: EmbeddingConfig) -> EmbeddingProvider factory function
- Support provider values: 'openai', 'sentence-transformers', 'local' (alias for sentence-transformers)
- Raise a clear error for unsupported providers
- Add a list_providers() -> list[str] function that returns available providers

Create a config.py in your project root (or src/ds_compare/) that loads configuration from a YAML or JSON file. Define a sample config:

```
embedding:

  provider: sentence-transformers

  model: all-MiniLM-L6-v2

  batch_size: 100
```

Test by loading config and verifying get_embedding_provider returns the correct implementation.

Why This Matters

The factory pattern centralizes object creation, making it easy to add new providers later. Configuration files separate deployment concerns from code—someone can switch from local to OpenAI embeddings just by changing config, no code changes needed. This is essential for a tool others will use.

Completion Checklist

- ☐ get_embedding_provider factory function works
- ☐ Supports both implemented providers
- ☐ Clear error messages for invalid providers
- ☐ Configuration loads from YAML/JSON file
- ☐ Can switch providers by changing config only

Week 5: Batch Embedding with Generators

Goal

Build a batch processing system that embeds chunks efficiently using generators. This handles large document collections without loading everything into memory.

Exercise

Create embeddings/batch.py with:

- A batch_chunks(chunks: Iterable[Chunk], batch_size: int) generator that yields lists of chunks
- An embed_chunks(chunks: list[Chunk], provider: EmbeddingProvider) function that returns chunks with vectors attached
- An embed_all(chunks: Iterable[Chunk], provider: EmbeddingProvider, batch_size: int) generator that yields embedded chunks
- Add progress reporting (print or logging) showing batches processed

Extend your Chunk dataclass (or create an EmbeddedChunk) to include an optional vector field: vector: list[float] | None = None

Test by embedding all chunks from your registry using the local provider. Time the operation and verify memory usage stays constant regardless of corpus size.

Why This Matters

Generators are essential for processing large datasets. Using yield instead of building a full list means you can embed millions of chunks without running out of memory. The batch_size parameter lets you tune the tradeoff between memory usage and API efficiency. This pattern appears throughout production ML systems.

Completion Checklist

- ☐ batch_chunks generator yields correct batch sizes
- ☐ embed_chunks attaches vectors to chunks
- ☐ embed_all processes entire corpus lazily
- ☐ Progress reporting shows batches processed
- ☐ Memory usage is constant (test with large corpus)

Week 6: Vector Storage in Registry

Goal

Extend your registry to store vectors alongside chunks and support similarity search. This bridges the gap between embedding and retrieval.

Exercise

Extend the ChunkRegistry Protocol with vector operations:

- **add_vectors(chunk_ids: list[str], vectors: list[list[float]]) -> None:** Associate vectors with existing chunks
- **get_vector(chunk_id: str) -> list[float] | None:** Retrieve a chunk's vector
- **has_vectors() -> bool:** Check if vectors have been added

Update InMemoryRegistry to implement these methods. Store vectors in a separate dictionary keyed by chunk_id.

Create a helper function cosine_similarity(a: list[float], b: list[float]) -> float in a new utils/math.py module. You can use numpy or implement it in pure Python.

Test by adding vectors from Week 5's embedding run and verifying you can retrieve them by chunk_id.

Why This Matters

Keeping vectors separate from chunks is a common pattern—vectors are large (hundreds to thousands of floats) and you don't always need them. The cosine_similarity function is the core of semantic search: higher values mean more similar content. You're building toward the search functionality next week.

Completion Checklist

- ☐ ChunkRegistry Protocol includes vector methods
- ☐ InMemoryRegistry stores and retrieves vectors
- ☐ cosine_similarity function works correctly
- ☐ Vectors from embedding pass are stored in registry
- ☐ has_vectors() returns correct state

Week 7: Semantic Search Implementation

Goal

Implement semantic search that finds the most relevant chunks for a query. This is the core retrieval functionality that makes your tool useful.

Exercise

Create a new module src/ds_compare/search/ with:

- **SearchResult:** A dataclass with: chunk (Chunk), score (float), rank (int)
- **Searcher Protocol:** With method search(query: str, top_k: int, filters: dict | None) -> list[SearchResult]
- **SemanticSearcher:** Implementation that takes registry and embedding_provider in __init__

The search method should:

- Embed the query using the provider
- If filters provided, first get matching chunks from registry
- Compute cosine similarity between query vector and all candidate chunk vectors
- Sort by similarity score descending
- Return top_k results as SearchResult objects

Test with queries like "keyboard navigation", "button variants", "accessibility". Verify results are relevant and filtered correctly when filters are applied.

Why This Matters

This is the payoff of semantic search: "focus management" finds content about focus even if those exact words aren't used. The filter-

then-rank pattern is efficient—metadata filtering is fast (dictionary lookup), while similarity computation is slow (math on every vector). You want to minimize the candidates before doing expensive operations.

Completion Checklist

- ☐ SearchResult and Searcher Protocol defined
- ☐ SemanticSearcher implements search correctly
- ☐ Results are sorted by similarity score
- ☐ Metadata filters work (e.g., design_system='spectrum')
- ☐ Queries return relevant results
- ☐ top_k parameter limits results correctly

Week 8: Search CLI and Integration Test

Goal

Build a command-line interface for searching and run a full integration test. This makes your tool usable and validates that all components work together.

Exercise

Create a simple CLI in src/ds_compare/cli.py using argparse (or click if you prefer):

- **ds-compare ingest --source <path> --type <local|git|web> --system <name>:** Load and chunk documents
- **ds-compare embed:** Embed all chunks in registry
- **ds-compare search <query> --top-k <n> --system <filter>:** Search and display results
- **ds-compare stats:** Show registry statistics (chunk count, systems, etc.)

Add a console entry point in pyproject.toml so you can run ds-compare from the command line.

Run a full integration test: ingest docs from two design systems, embed everything, search for "button states", verify results include content from both systems ranked by relevance.

Why This Matters

CLIs make tools usable. Even if you later add a web interface, a CLI is invaluable for debugging, scripting, and quick lookups. The integration test validates your entire pipeline: load → chunk → embed → search. If this works, your Phase 1 and Phase 2 code is solid.

Completion Checklist

- ☐ CLI commands work from terminal
- ☐ ingest command loads and chunks documents
- ☐ embed command processes all chunks
- ☐ search command returns formatted results
- ☐ Integration test passes with two design systems
- ☐ Results show relevance across different terminology

Phase 2 Complete

By the end of Week 8, you have:

- A modular embedding system with swappable providers (local and OpenAI)
- Efficient batch processing using generators
- Vector storage integrated with your chunk registry
- Semantic search with metadata filtering
- A working CLI for ingestion, embedding, and search
- An end-to-end pipeline tested with real design system documentation

Before moving to Phase 3, verify you can search for concepts across design systems and get relevant results despite terminology differences. Try queries like "how to disable a button", "loading indicators", "form validation". The semantic search should surface relevant content regardless of exact wording.