Phase 4: Polish and Instrumentation

Weekly Exercises — Months 7–8

Phase Overview

By the end of this phase, your code will be production-quality: comprehensive type hints, decorators for cross-cutting concerns, structured logging, caching for expensive operations, and clean module organization. You'll also add persistence so data survives between runs.

What You'll Build

- Decorators for timing, caching, and retry logic
- Structured logging throughout the pipeline
- Persistent storage for registry and vectors
- Comprehensive type hints verified with mypy

Key Python Concepts

Decorators (with and without arguments), functools.wraps, logging module, functools.lru_cache, context managers, type hints with generics, mypy for static type checking, JSON/SQLite persistence.

Week 1: The @timed Decorator

Goal

Write your first decorator that measures and logs function execution time. This is the foundation for understanding how decorators work.

Exercise

Create src/ds_compare/utils/decorators.py with a @timed decorator:

- Measure elapsed time using time.perf_counter()
- Log function name, arguments (summarized), and duration
- Use @functools.wraps to preserve the original function's metadata
- Return the original function's result unchanged
- Handle both sync and async functions (check with asyncio.iscoroutinefunction)

Apply @timed to your embed_chunks and search functions. Run a search and observe the timing logs.

Example output:

```
[TIMING] search(query='button states', top_k=5) completed in 0.342s
```

Why This Matters

Decorators add behavior without modifying function code—essential for cross-cutting concerns like logging and timing. In AI applications, knowing which operations are slow (embedding? LLM calls? search?) is critical for optimization. functools.wraps preserves introspection, which matters for debugging and documentation.

Completion Checklist

- ☐ @timed decorator implemented
- ☐ Uses @functools.wraps
- ☐ Handles async functions correctly
- ☐ Logs function name, args summary, and duration
- ☐ Applied to key functions and producing useful output

Week 2: The @retry Decorator

Goal

Write a decorator that retries failed operations with exponential backoff. This is essential for working with unreliable external APIs.

Exercise

Add a @retry decorator with configurable parameters:

- @retry(max_attempts=3, delay=1.0, backoff=2.0, exceptions=(Exception,))
- Catch only specified exception types
- Wait delay * (backoff ** attempt) between retries
- Log each retry attempt with the exception message
- After max_attempts, re-raise the last exception
- Handle async functions with asyncio.sleep

This requires a decorator with arguments—a decorator factory pattern. The structure is:

```
def retry(max_attempts=3, ...):

    def decorator(func):

        @functools.wraps(func)

        def wrapper(*args, **kwargs):

            # retry logic

        return wrapper

    return decorator
```

Apply @retry to your LLM API calls in the providers. Test by temporarily making an API call fail.

Why This Matters

External APIs fail transiently—rate limits, network blips, server errors. Retry with backoff handles these gracefully. The decorator factory pattern (decorator with arguments) is more complex but very powerful. You'll see this pattern throughout production Python code.

Completion Checklist

- ☐ @retry decorator with configurable parameters
- ☐ Exponential backoff implemented correctly
- ☐ Only catches specified exceptions
- ☐ Logs retry attempts
- ☐ Handles async functions
- ☐ Applied to API calls in providers

Week 3: Structured Logging

Goal

Replace print statements with structured logging. Proper logging makes debugging production issues possible.

Exercise

Create src/ds_compare/utils/logging.py:

- Configure the logging module with a consistent format
- Include timestamp, level, module name, and message
- Create a get_logger(name: str) helper that returns configured loggers
- Support log level configuration via environment variable (DS_COMPARE_LOG_LEVEL)
- Add file handler option for persistent logs

Go through your codebase and replace all print() calls with appropriate logging:

- logger.debug() for detailed internal state
- logger.info() for normal operations (ingesting file X, searching for Y)
- logger.warning() for recoverable issues (missing file, retry attempt)
- logger.error() for failures (API error, validation failure)

Why This Matters

The logging module is Python's built-in solution for production logging. Log levels let you control verbosity without changing code. Structured logs (with timestamps, levels, sources) are essential for debugging issues in production. This is table stakes for professional Python development.

Completion Checklist

- ☐ Logging configured with consistent format
- ☐ get_logger helper function works
- ☐ Log level configurable via environment
- ☐ All print() statements replaced with logging
- ☐ Appropriate log levels used throughout

Week 4: Caching with @cached

Goal

Implement caching for expensive operations like LLM extraction. Repeated comparisons of the same component should be instant.

Exercise

Create src/ds_compare/cache/ module with:

- **Cache Protocol:** get(key: str) -> Any | None, set(key: str, value: Any, ttl: int | None), delete(key: str), clear()
- **MemoryCache:** Simple dict-based implementation with optional TTL
- **DiskCache:** JSON file-based implementation for persistence

Create a @cached decorator:

- @cached(cache: Cache, key_fn: Callable | None = None, ttl: int | None = None)
- Generate cache key from function name + arguments (or use custom key_fn)
- Check cache before calling function
- Store result in cache after calling
- Log cache hits and misses

Apply caching to extract_component. Run the same extraction twice and verify the second call is instant (cache hit).

Why This Matters

LLM calls are slow and expensive. Caching avoids redundant work—comparing Button across systems twice should only call the LLM once per system. The TTL (time-to-live) option lets you invalidate stale data. DiskCache makes this persist across program restarts.

Completion Checklist

- ☐ Cache Protocol defined
- ☐ MemoryCache and DiskCache implemented
- ☐ @cached decorator works
- ☐ Cache hits logged
- ☐ Extraction caching verified (second call instant)
- ☐ TTL expiration works

Week 5: Persistent Registry

Goal

Add a persistent registry implementation so chunks and vectors survive between runs. No more re-ingesting and re-embedding every time you start.

Exercise

Create registry/sqlite.py with SQLiteRegistry:

- Use sqlite3 (built-in, no dependencies)
- Create tables: chunks (id, content, document_id, metadata JSON), vectors (chunk_id, vector BLOB)
- Implement all ChunkRegistry Protocol methods
- Store vectors as numpy arrays serialized with pickle or as JSON
- Use context manager for database connections
- Add index on metadata fields used for filtering (design_system, component)

Update your configuration to choose between memory and SQLite registry. Test by: ingesting docs, closing the program, restarting, and verifying chunks are still there.

Why This Matters

Persistence transforms a script into a usable tool. Ingesting and embedding docs is slow; doing it once and reusing is essential. SQLite is perfect for this—single file, no server, built into Python. The Protocol interface means the rest of your code doesn't care which registry you use.

Completion Checklist

- ☐ SQLiteRegistry implements ChunkRegistry Protocol
- ☐ Tables created with appropriate schema
- ☐ Vectors stored and retrieved correctly
- ☐ Metadata filtering works via SQL queries
- ☐ Data persists across program restarts
- ☐ Configurable via config file

Week 6: Comprehensive Type Hints

Goal

Add comprehensive type hints throughout the codebase and verify them with mypy. This catches bugs early and makes the code self-documenting.

Exercise

Install and configure mypy:

- pip install mypy
- Create mypy.ini or add [tool.mypy] to pyproject.toml
- Enable strict mode (strict = true) for maximum checking

Go through each module and ensure all functions have:

- Parameter type annotations
- Return type annotations
- Generic types where appropriate (list[str], dict[str, Any])
- Union types for optional parameters (str | None)

Run mypy src/ds_compare and fix all errors. Common issues: missing return types, incompatible types, missing imports from typing module.

Why This Matters

Type hints catch bugs before runtime—mypy can find type mismatches, missing return statements, and incorrect function calls. They also serve as documentation, making code easier to understand. Modern Python libraries expect type hints, and writing typed code prepares you to read their source.

Completion Checklist

- ☐ mypy configured in project
- ☐ All functions have parameter and return type hints
- ☐ Generic types used appropriately
- ☐ mypy passes with no errors
- ☐ Strict mode enabled

Week 7: Module Reorganization

Goal

Review and reorganize your module structure for clarity. Clean architecture makes the codebase maintainable as it grows.

Exercise

Review your current structure and reorganize if needed. A clean structure might look like:

```
src/ds_compare/

  models/          # Data models (Document, Chunk, etc.)

  loaders/          # Document loaders

  chunkers/         # Chunking strategies

  embeddings/       # Embedding providers

  search/           # Search functionality

  llm/              # LLM providers

  comparison/       # Comparison models and extraction

  registry/         # Storage backends

  cache/            # Caching implementations

  utils/            # Decorators, logging, helpers

  cli.py            # Command-line interface
  config.py          # Configuration loading
```

For each module's __init__.py, export the public interface—the classes and functions other modules should use. Keep implementation details private.

Create a src/ds_compare/__init__.py that exports the main entry points: the CLI, key classes for programmatic use.

Why This Matters

Good module structure makes code navigable. Someone looking at your project should immediately understand where to find loader code, search code, etc. Explicit exports in __init__.py define the public API and hide implementation details. This is how professional Python packages are organized.

Completion Checklist

- ☐ Module structure is logical and navigable
- ☐ Each module has clear responsibility
- ☐ __init__.py files export public interfaces
- ☐ Top-level package exports main entry points
- ☐ All imports still work after reorganization

Week 8: Documentation and README

Goal

Write documentation that lets someone else (or future you) use and extend the tool. Good docs complete the "production-quality" transformation.

Exercise

Create/update README.md with:

- Project overview: what it does, who it's for
- Installation instructions
- Quick start guide with example commands
- Configuration reference (all config options explained)
- CLI command reference
- How to add a new design system

Add docstrings to key classes and functions. Use Google or NumPy docstring format for consistency. At minimum, document:

- All Protocol classes (the interfaces)
- Public methods on main classes
- Complex functions with non-obvious behavior

Why This Matters

Documentation is what makes code usable by others—including future you. A good README is often the difference between a project being used and being ignored. Docstrings enable IDE autocompletion and help text. This is the final polish that makes your project professional.

Completion Checklist

- ☐ README covers all sections
- ☐ Installation instructions tested on clean environment
- ☐ Quick start actually works as written
- ☐ Key classes and methods have docstrings
- ☐ Configuration options documented
- ☐ Someone new could add a design system following the docs

Phase 4 Complete

By the end of Week 8, you have:

- Decorators for timing, retry, and caching
- Structured logging throughout the codebase
- Persistent storage with SQLite
- Comprehensive type hints verified by mypy
- Clean module organization
- Documentation for users and developers

Your tool is now production-quality. Before moving to Phase 5, verify everything works end-to-end: ingest multiple design systems, embed, search, compare components—all with logging, caching, and persistence working. The foundation is solid for building the agentic features in the next phase.