Slice – Phase 4 → Phase 5 Handoff

SLICE – PHASE 4 → PHASE 5 HANDOFF

1. Why this document exists

Phase 4 is now functionally complete. This handoff is written for whoever owns Phase 5, so they can:

- Understand what Phase 4 guarantees and exposes.
- Avoid breaking core contracts (schemas, risk interfaces, backtest outputs).
- Build Phase 5 features on top of a stable memory layer instead of re-inventing it.

Read this as a practical “integration surface” and checklist, not a marketing overview.

2. What you are inheriting from Phase 4

2.1 Stable schemas (Pydantic + DB)

You inherit canonical models for:

- Thesis (src/slice/models/thesis.py)
- Observation (src/slice/models/observation.py)
- Trade (src/slice/models/trade.py)
- Scenario (src/slice/models/scenario.py)

These models are:

- enforced by tests in tests/test_phase4_models.py
- in 1:1 correspondence with DB tables created in sql/phase4_schema.sql

You should treat these as **frozen interfaces** for Phase 5:

- You may add optional fields, but changing existing field names, types, or semantics will break:
- repositories
- validators
- ingestion pipeline
- tests
- and potentially docs written in earlier phases

2.2 Persistence layer

Repositories exist for all four models:

- ThesisRepository
- ObservationRepository
- TradeRepository
- ScenarioRepository

All DB writes for Phase 4 entities go through these repositories. This is critical:

- Do not write raw INSERT/UPDATEs in new Phase 5 code.
- If you need new behavior, add methods to repositories so that everything remains centralized.

2.3 Validation + normalization

The llm_validation package does two things well:

- Normalizes messy dicts (ids, timestamps, categories, enums).
- Converts validation failures into a structured ValidationResult (ok flag + list of ValidationIssue).

For Phase 5:

- You should feed user/LLM payloads through these validators instead of constructing Pydantic models by hand.
- If Phase 5 introduces new fields, extend the validators to keep errors coherent.

2.4 Observation semantic memory

The major new capability from Phase 4 is:

- You can take arbitrary text (“Fed outlook is hawkish due to sticky inflation”), turn it into a normalized Observation row, embed it using OpenAI, and store it with a pgvector column.
- You can recall nearest-neighbor observations for any new piece of text.

This pipeline is fully wired and tested:

- IngestionPipeline.ingest_observation_with_embedding
- search_similar_observations
- MemoryService.recall_similar_text
- MemoryContextBuilder.build_for_text
- ObservationMemoryWorkflow.ingest_and_build_context
- FastAPI endpoint: POST /api/v1/memory/observe_and_recall
- CLI script: scripts/memory_ingest_and_recall.py

Phase 5 is expected to lean heavily on this: every LLM-facing workflow that involves “what have I thought about this before?” should go through the memory layer instead of manually querying the DB.

3. Key invariants Phase 5 must respect

3.1 Do not change backtest/risk contracts

Even though Phase 4 mostly touched memory, it sits on top of Phase 3, which defined:

- BacktestResult, StrategyReturnSeries, TimeSeriesPoint (risk/schemas.py)
- RiskReport and its structure (risk/schemas.py, risk/report.py)

Phase 5 will likely want to show risk metrics or backtest summaries in LLM responses. You **must not** change:

- BacktestResult fields
- RiskReport fields

Instead, adapt these objects into whatever text/JSON the LLM sees. You inherit a stable risk engine, not a mutable one.

3.2 Writes must go through ingestion + repositories

New LLM or UI features in Phase 5 must not write directly to the database. The rule is:

- Raw dict → llm_validation.* → IngestionPipeline.* → Repository.* → DB

Reasons:

- Consistent validation and normalization.
- Central place for future auditing, logging, and invariants.
- Easier to introduce versioning or soft-deletes later.

3.3 Embedding model and dimensionality

Phase 4 uses a single embedding model (e.g., text-embedding-3-small). This implies:

- All vector rows in observation.embedding have the same dimensionality.
- The search function assumes this shape.

Phase 5 must:

- Not silently switch the embedding model without a migration plan.
- If you ever change the model, either:
- Recompute embeddings for all existing observations, or
- Store model_id along with embedding and restrict searches to a consistent subset.

For now, treat OPENAI_EMBEDDING_MODEL as a constant.

3.4 “Memory context” is structured text, not JSON

MemoryContextBuilder.build_for_text returns a **text block**. It's constructed specifically to be concatenated into an LLM prompt.

Phase 5 should treat it as an opaque chunk of text that can be prefixed before the main user query. If you want richer structured context for tools, you can extend the builder to also return structured metadata, but do not change the existing string format without adjusting tests and API expectations.

4. How Phase 5 should use Phase 4

4.1 The core pattern: "Observe, recall, reason"

Any new Phase 5 feature that involves the LLM should follow this pattern:

1) Observe

- Take the user's new thought, summary, or thesis snippet.
- Normalize it into an Observation:
- Build a dict with text, thesis_ref (if any), categories, sentiment.
- Pass it through ObservationMemoryWorkflow.ingest_and_build_context (or at least IngestionPipeline.ingest_observation_with_embedding if you don't need context yet).

2) Recall

- Use MemoryService.recall_similar_text(text, k=...) or MemoryContextBuilder.build_for_text(text, ...).
- This gives you K nearest prior observations and a context_block summarizing them.

3) Reason

- Construct an LLM prompt like:

- System: "You are Slice Macro Copilot…"
- Context: context_block from MemoryContextBuilder
- User: the new question or thesis

- Ask the LLM to reason *in light of* prior observations instead of hallucinating fresh context every time.

This ensures Phase 5 logic is **memory-aware** by design.

4.2 Integrating with backtests and risk reports

Phase 3 already provides:

- Quant backtests via quant_engine/interface/run_backtest.py.
- Risk reports via engine/analytics_pipeline.py (BacktestResult → RiskReport).

Phase 5 should not recompute raw analytics logic. Instead:

- Run or fetch backtests/risk reports as separate steps (e.g., precomputed or triggered by user action).
- When crafting LLM prompts, attach the relevant numbers or a textual summary derived from:
- RiskReport.risk_metrics
- RiskReport.risk_rails
- RiskReport.factor_model
- RiskReport.scenarios

Guideline:

- Memory context gives “what you’ve thought/seen before.”
- Risk/Backtest context gives “how the portfolio or strategy actually behaved.”
- The LLM’s job is to synthesize both.

5. Suggested Phase 5 workstreams

Below is a concrete breakdown of what Phase 5 should do, using Phase 4 as the foundation.

5.1 Expand memory to Theses and Trades (optional but powerful)

Right now, only observations are embedded. Phase 5 could:

- Add embedding support for Theses and Trades:
- New functions: embed_thesis_text, embed_trade_note (if trades carry notes).
- New vector columns: thesis.embedding, trade.embedding (with pgvector).
- New retrieval utilities: search_similar_theses, search_similar_trades.
- A unified MemoryService layer that can recall across entity types, not just observations.

Constraints:

- Keep existing tables and models backward compatible.
- Only add optional columns and functions.

5.2 LLM-facing “session orchestrator”

Design and implement a higher-level orchestrator that manages “Slice sessions” for the user.

Responsibilities:

- Given an incoming user message:
- Log it as an Observation via ObservationMemoryWorkflow.ingest_and_build_context.
- Fetch relevant prior observations (and possibly theses/trades in future).
- Optionally fetch relevant risk/backtest summaries.
- Build a consolidated prompt for the LLM that includes:
- Memory context block.
- Numerical or textual summary of portfolio/risk context.
- The user’s new question or thesis.

- Ensure every LLM answer is grounded in:
- Past observations (semantic memory)
- Real portfolio/risk data (Phase 3 engine)

Implementation sketch:

- New module: src/slice/session/orchestrator.py
- Core function: run_session_step(user_text: str, options: SessionOptions) -> SessionResponse
- SessionResponse includes:
- observation_id
- raw LLM response
- context used (memory + risk) for debugging/logging

5.3 API endpoints for session orchestration

Extend FastAPI app(s) to expose Phase 5 functionality:

- Example endpoints:
- POST /api/v1/session/step
- body: { "text": "...", "thesis_ref": "...", "mode": "diagnostic" | "trade_idea" }
- returns: { "response": "...", "observation_id": "...", "context_block": "...", "risk_snapshot": {...} }

- Reuse existing /api/v1/memory/observe_and_recall endpoint internally rather than duplicating logic.

5.4 Guardrails, logging, and observability

Phase 4 already surfaces structured ValidationResult on the ingestion side. Phase 5 should:

- Log every LLM interaction alongside:
- observation_id
- context_block hash or truncated version
- risk snapshot hash or truncated version
- model name and parameters used
- Avoid initiating trades or broker actions; Phase 5 should remain advisory, not autonomous.

5.5 (Optional) Introduce Alembic migrations

If you decide to harden the schema story in Phase 5, now is the time to introduce Alembic:

- alembic init migrations
- Create a base migration representing slice_schema.sql + phase4_schema.sql.
- Future schema tweaks (new columns, indexes) should be written as revisions.

This will make later phases - especially when collaborating with more devs or deploying to a server - more robust.

6. Concrete “do not break” list

To keep the system coherent, Phase 5 must not:

1) Change the meaning or types of:
- Thesis, Observation, Trade, Scenario fields in src/slice/models.
- BacktestResult, StrategyReturnSeries, TimeSeriesPoint, RiskReport in src/slice/risk/schemas.py.

2) Bypass repositories for DB writes.

3) Write embeddings with a different dimension into the existing observation.embedding column without a clear migration.

4) Change the signature of:
- IngestionPipeline.ingest_observation_with_embedding
- MemoryService.recall_similar_text
- ObservationMemoryWorkflow.ingest_and_build_context
- FastAPI endpoint /api/v1/memory/observe_and_recall

5) Assume that embeddings or memory will be instantaneous if you increase usage significantly; you may need batching, caching, or rate limiting in future phases.

7. How to start Phase 5 practically

A realistic starting sequence:

1) Run all tests to ensure you’re on a clean slate:

- slice-init
- pytest -q

2) Manually exercise the memory API once to get a feel for it:

- Start the FastAPI app (if not already running):
- slice-init
- uvicorn slice.api.memory_app:app --reload

- From another shell:
- curl -X POST http://127.0.0.1:8000/api/v1/memory/observe_and_recall -H "Content-Type: application/json" -d '{
"text": "Fed is worried about sticky inflation and keeps rates high.",
"thesis_ref": "fed_rates",
"sentiment": "BEARISH",
"categories": ["fed", "inflation", "rates"],
"k": 3
}'

- Confirm the response returns ok: true and a non-empty context_block.

3) Prototype a minimal session orchestrator that:

- Accepts user text.
- Calls ObservationMemoryWorkflow.ingest_and_build_context.
- Generates a simple LLM response that just summarizes the memory context back to you.
- Once that works, layer in risk/backtest context next.

This approach keeps Phase 5 grounded, incremental, and aligned with the contracts Phase 4 has already established.

8. Summary

By the end of Phase 4, you have:

- Stable Pydantic models and DB tables for thesis/observation/trade/scenario.
- A validated ingestion pipeline that normalizes and persists these entities.
- A working semantic memory system for observations, backed by OpenAI embeddings and pgvector.
- An API and CLI entrypoint to exercise “observe and recall” end-to-end.
- Tests proving that the models, ingestion, and memory workflow operate correctly.

Phase 5’s job is not to rebuild any of that, but to:

- Treat it as a stable substrate.
- Route user and LLM interaction through it.
- Add orchestration, context assembly, and reasoning layers that leverage both memory and risk/backtest data to produce grounded, repeatable macro decision support.