Slice – Phase 4 Completion

SLICE – PHASE 4 COMPLETION DOCUMENT

1. Scope of Phase 4

Phase 4 adds the “thesis/observation/trade/scenario” layer plus semantic memory on top of the existing Slice stack.

Concretely, Phase 4 is responsible for:

- Canonical Pydantic 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)
- Database schema for these entities, including pgvector-based embeddings for observations.
- Repositories that read/write these models to Postgres.
- LLM-facing validation and normalization layer to enforce schema and guardrails.
- Ingestion pipeline that validates and persists data.
- Semantic memory based on OpenAI embeddings + pgvector (observation-level).
- A small FastAPI app and CLI to exercise the memory pipeline.
- Tests covering models, ingestion, and memory recall.

This document describes what has been implemented and what invariants Phase 5 must respect.

2. Environment and Infrastructure

2.1 Database

- Engine: PostgreSQL 17 (Homebrew on macOS)
- Extension: pgvector installed and enabled in the “slice” database.
- Connection string (via .env):
- SLICE_DB_URL="postgresql+psycopg2://slice_user:slice_password@localhost:5432/slice"

DB initialization flow:

- Base schema: sql/slice_schema.sql
- Phase 4 additions: sql/phase4_schema.sql
- Application helper:
- scripts/apply_phase4_schema.py calls slice.db.apply_schema / apply_phase4_schema.
- Connectivity helper:
- slice.db.get_engine(): singleton SQLAlchemy engine on SLICE_DB_URL.
- slice.db.ping(): “SELECT 1” sanity check used by slice-init and tests.

2.2 OpenAI embeddings

- Environment variables (from .env):
- OPENAI_API_KEY=...
- OPENAI_EMBEDDING_MODEL=text-embedding-3-small (or similar)
- Client code:
- src/slice/embeddings.py
- embed_observation_text(text: str) -> list[float]

This function is the single dependency on OpenAI for Phase 4.

2.3 Shell workflow

Phase 4 assumes the following terminal helper is present in ~/.zshrc:

- Function: slice-init()

Behavior:

- cd ~/dev/slice
- Activate .venv
- Start PostgreSQL 17 via brew services start postgresql@17 (if brew is available)
- Source .env with “set -a; source .env; set +a” semantics
- Export PYTHONPATH=src
- Call slice.db.ping() to verify DB connectivity
- Print “Slice environment initialized.”

This is the standard entry point for all scripts, tests, and ad hoc experiments.

3. Data Models (Pydantic)

3.1 Common enums

File: src/slice/models/common.py

- ThesisStatus: ACTIVE, PARKED, CLOSED, etc.
- Direction: LONG, SHORT
- TradeType: LIVE, SIMULATED, PAPER
- Sentiment: VERY_BULLISH, BULLISH, NEUTRAL, BEARISH, VERY_BEARISH, ANXIOUS, CONFIDENT

3.2 Thesis

File: src/slice/models/thesis.py

Key fields:

- id: str (primary key, “thesis_...” or user-provided)
- title: str
- hypothesis: str (natural-language hypothesis)
- drivers: list[str] (supporting reasons)
- disconfirmers: list[str] (what would invalidate the thesis)
- expression: list[ThesisExpressionLeg]
- ThesisExpressionLeg:
- asset: str (e.g., GLD, XLE)
- direction: Direction (LONG or SHORT)
- size_pct: float (target portfolio percentage)
- start_date: str (YYYY-MM-DD)
- review_date: Optional[str]
- status: ThesisStatus
- tags: list[str]
- monitor_indices: list[str]
- notes: Optional[str]

Validators:

- Non-empty title, hypothesis, drivers, disconfirmers.
- At least one expression leg.

3.3 Observation

File: src/slice/models/observation.py

Key fields:

- id: str (e.g., “obs_YYYYMMDDHHMMSS_xxx”; generator lives in llm_validation.normalization)
- timestamp: datetime
- text: str (the actual observation)
- thesis_ref: list[str] (one or more thesis IDs)
- sentiment: Sentiment (enum)
- categories: list[str] (free-form tags, normalized from comma-separated input)
- actionable: str (e.g., “monitoring”, “review”, “urgent”)

This is the primary “memory item” for Phase 4.

3.4 Trade

File: src/slice/models/trade.py

Key fields:

- id: str
- asset: str
- action: str (“buy” / “sell”)
- quantity: float

- price: float
- timestamp: datetime
- type: TradeType (LIVE / SIMULATED / PAPER)
- thesis_ref: str (foreign key-ish to Thesis.id)

Phase 4 does not embed trades, but schema + persistence are in place.

3.5 Scenario

File: src/slice/models/scenario.py

Key fields:

- scenario_id: str
- name: str
- assumptions: dict[str, str]
- expected_impact: dict[str, float] (asset symbol → expected P&L; or return)
- description: Optional[str]

This is separate from risk.scenarios.ScenarioConfig, which is used for factor-based scenario analysis. Phase 4 scenario is “story-level” and attaches to thesis/macro reasoning, not directly to factor regression.

4. Database Schema

File: sql/phase4_schema.sql

4.1 thesis table

Columns (simplified):

- id TEXT PRIMARY KEY
- title TEXT NOT NULL
- hypothesis TEXT NOT NULL
- drivers JSONB NOT NULL
- disconfirmers JSONB NOT NULL
- expression JSONB NOT NULL
- start_date DATE NOT NULL
- review_date DATE NULL
- status TEXT NOT NULL
- tags JSONB NOT NULL
- monitor_indices JSONB NOT NULL
- notes TEXT NULL

4.2 observation table

Columns:

- id TEXT PRIMARY KEY
- timestamp TIMESTAMPTZ NOT NULL
- text TEXT NOT NULL
- thesis_ref TEXT[] NOT NULL
- categories JSONB NOT NULL
- sentiment TEXT NOT NULL
- actionable TEXT NOT NULL
- embedding VECTOR NULL (pgvector type)

Note: embedding can be NULL for older rows or non-embedded writes, but Phase 4's main flow always fills it.

4.3 trade table

Columns:

- id TEXT PRIMARY KEY
- asset TEXT NOT NULL
- action TEXT NOT NULL
- quantity DOUBLE PRECISION NOT NULL
- price DOUBLE PRECISION NOT NULL
- timestamp TIMESTAMPTZ NOT NULL
- type TEXT NOT NULL
- thesis_ref TEXT NOT NULL

4.4 scenario table

Columns:

- scenario_id TEXT PRIMARY KEY
- name TEXT NOT NULL
- assumptions JSONB NOT NULL
- expected_impact JSONB NOT NULL
- description TEXT NULL

5. Repositories

All repositories use slice.db.get_engine() and parameterized SQL (sqlalchemy.text).

5.1 ThesisRepository

File: src/slice/repositories/thesis_repo.py

Responsibilities:

- insert(thesis: Thesis) -> None
- Upsert via ON CONFLICT (id) DO UPDATE.
- JSONB fields (drivers, disconfirmers, expression, tags, monitor_indices) are json.dumps(...) at the repo layer.
- get(thesis_id: str) -> Optional[Thesis]
- list_all() -> List[Thesis]

5.2 ObservationRepository

File: src/slice/repositories/observation_repo.py

Responsibilities:

- insert(observation: Observation, embedding_vector: Optional[list[float]]) -> None
- Uses CAST(:embedding AS vector) for pgvector.
- categories is json.dumps(list_of_str).
- thesis_ref stored as TEXT[].
- get(observation_id: str) -> Optional[Observation]
- Converts JSONB categories → Python list.
- list_for_thesis(thesis_id: str) -> List[Observation]
- WHERE :thesis_id = ANY(thesis_ref)

5.3 TradeRepository

File: src/slice/repositories/trade_repo.py

Responsibilities:

- insert(trade: Trade) -> None (upsert on id)
- get(trade_id: str) -> Optional[Trade]
- list_for_thesis(thesis_id: str) -> List[Trade]

5.4 ScenarioRepository

File: src/slice/repositories/scenario_repo.py

Responsibilities:

- insert(scenario: Scenario) -> None (upsert)
- assumptions/expected_impact stored as JSONB.
- get(scenario_id: str) -> Optional[Scenario]
- list_all() -> List[Scenario]

6. LLM Validation Layer

Package: src/slice/llm_validation

6.1 Shared machinery

- ValidationResult
- ok: bool
- errors: list[ValidationIssue]
- model: Optional[PydanticModel]
- ValidationIssue
- field: str
- code: str
- message: str
- context: dict[str, Any]

6.2 Normalization utilities

File: llm_validation/normalization.py

- generate_observation_id() → “obs_YYYYMMDDHHMMSS_xxxxxxxx”
- normalize_categories(raw) → list[str]
- Accepts comma-separated strings or list-like inputs.
- normalize_timestamp(...) (uses datetime.utcnow for now)

6.3 Per-entity validators

Files:

- llm_validation/thesis_validation.py → validate_thesis(raw: dict) -> ValidationResult
- llm_validation/observation_validation.py → validate_observation(raw: dict) -> ValidationResult
- llm_validation/trade_validation.py → validate_trade(raw: dict) -> ValidationResult
- llm_validation/scenario_validation.py → validate_scenario(raw: dict) -> ValidationResult

Pattern:

- Fill missing ID for observations if not provided.
- Normalize categories and timestamp.
- Coerce enum fields from strings (uppercasing; mapping to Sentiment, TradeType, ThesisStatus).
- Catch and report all Pydantic errors as ValidationIssue list.

Prompt templates (PromptBuilder) are stubbed but Phase 4 does not yet call OpenAI for validation – only for embeddings.

7. Ingestion Pipeline

File: src/slice/ingest/pipeline.py

Class: IngestionPipeline

Methods:

- ingest_thesis(raw: dict) -> ValidationResult
- validate_thesis → ThesisRepository.insert → return ValidationResult.

- ingest_observation(raw: dict, embedding_vector=None) -> ValidationResult
- validate_observation → ObservationRepository.insert(result.model, embedding_vector).

- ingest_observation_with_embedding(raw: dict) -> ValidationResult
- validate_observation → text = model.text → embed_observation_text(text) → ingest_observation(raw, embedding_vector).

- ingest_trade(raw: dict) -> ValidationResult
- validate_trade → TradeRepository.insert.

- ingest_scenario(raw: dict) -> ValidationResult
- validate_scenario → ScenarioRepository.insert.

8. Semantic Memory Layer

8.1 Embedding client

File: src/slice/embeddings.py

- embed_observation_text(text: str) -> list[float]
- Uses OpenAI’s embeddings API with OPENAI_EMBEDDING_MODEL.
- Returns raw embedding vector as list[float].

8.2 Low-level retrieval

File: src/slice/memory/retrieval.py

Function: search_similar_observations(...)

Signature:

- search_similar_observations(
query_text: str,
k: int = 5,
since: Optional[str] = None,
until: Optional[str] = None,
categories: Optional[list[str]] = None,
sentiment: Optional[str] = None,
) -> list[tuple[Observation, float]]

Implementation:

- Embeds query_text via embed_observation_text.
- Builds pgvector literal “[0.1234,0.5678,...]”.
- Runs SQL:

SELECT
id,
timestamp,
text,
thesis_ref,
categories,
sentiment,
actionable,
embedding <-> CAST(:embedding AS vector) AS distance
FROM observation
WHERE embedding IS NOT NULL
AND (optionally) timestamp filters, category filters, sentiment filters
ORDER BY embedding <-> CAST(:embedding AS vector)
LIMIT :k;

- Returns list of (Observation, distance).

8.3 MemoryService

File: src/slice/memory/service.py

- MemoryService.recall_similar_text(text: str, k: int = 5) -> list[tuple[Observation, float]]
- Thin wrapper around search_similar_observations.

8.4 ContextBuilder

File: src/slice/memory/context_builder.py

- MemoryContextBuilder.build_for_text(text: str, k: int = 5, max_chars: int = 2000) -> dict

Returns:

- {
"context_block": "...formatted text...",
"matches": List[Observation],
}

The context_block is a human/LLM-readable multi-line string of the form:

Relevant prior observations:
- id: ...
distance: 0.85
categories: fed, inflation
sentiment: BEARISH

thesis_ref: fed_rates
text: ...

Truncates to max_chars.

8.5 ObservationMemoryWorkflow

File: src/slice/memory/workflow.py

- ObservationMemoryWorkflow.ingest_and_build_context(raw: dict, k: int, max_chars: int) -> ObservationMemoryResult

Where ObservationMemoryResult is a small dataclass / Pydantic model with:

- ok: bool
- errors: list[ValidationIssue]
- observation_id: Optional[str]
- context_block: Optional[str]

Flow:

- Validate + normalize raw via validate_observation.
- Generate embedding and insert via IngestionPipeline.ingest_observation_with_embedding.
- Call MemoryContextBuilder.build_for_text(...) to get surrounding context.
- Return both the ValidationResult status and the assembled context.

9. API and CLI

9.1 FastAPI app

File: src/slice/api/memory_app.py

- Endpoint: POST /api/v1/memory/observe_and_recall

Request body:

- text: str
- thesis_ref: Optional[str or list[str]]
- sentiment: Optional[str]
- categories: Optional[list[str or comma-separated str]]
- k: Optional[int] (default 5)

Behavior:

- Build raw observation dict.
- Call ObservationMemoryWorkflow.ingest_and_build_context.
- Return JSON:

{
"ok": bool,
"errors": [...],
"observation_id": "...",
"context_block": "...",
"matches": [] # reserved for future richer payload
}

9.2 CLI script

File: scripts/memory_ingest_and_recall.py

Usage example:

- slice-init
- python scripts/memory_ingest_and_recall.py "Fed worried about sticky inflation and higher-for-longer policy" --thesis-ref fed_rates --sentiment BEARISH --categories "fed,inflation,rates" -k 3

Output:

- ok flag
- observation ID
- context block printed to stdout.

10. Tests and Status

10.1 Tests

- tests/test_phase4_models.py
- Asserts Pydantic models for thesis/observation/trade/scenario accept valid payloads and reject invalid ones.

- tests/test_phase4_memory_pipeline.py
- End-to-end: ingest a few observations with embeddings and recall via MemoryService.recall_similar_text.

- tests/test_phase4_observation_workflow.py
- Smoke test for ObservationMemoryWorkflow.ingest_and_build_context.

All Phase 4 tests are passing under pytest with pytest.ini configured as:

[pytest]
testpaths = tests

pythonpath = src

10.2 Sanity scripts

Ad hoc scripts and REPL snippets were used to confirm:

- DB connectivity and schema presence.
- pgvector extension installed and usable.
- Embeddings insert successfully and can be queried.
- MemoryContextBuilder produces coherent context blocks.

11. Known Limitations / Out of Scope for Phase 4

- No Alembic migrations: schema managed via raw SQL files.
- Only observations are embedded; Theses/Trades/Scenarios do not yet participate in semantic search.
- No UI integration; only API + CLI.
- LLM is not yet used to generate or modify theses, trades, or scenarios; it only provides embeddings.
- No retention policies or archival processes for old observations.

Within these boundaries, Phase 4 is complete: core schemas, persistence, validation, embedding pipeline, and semantic recall for observations are fully implemented and tested.