# PRD Ordering Guide — claude-mem-lite

**Date**: 2026-02-12 **Purpose**: Explain the reading order and implementation sequence for all PRDs, why each phase exists where it does, and flag issues found during independent review.

---

## Quick Reference: Reading Order

| Order | Document | Role |
|---|---|---|
| 1 | claude-mem-lite-architecture.md | Vision, design rationale, technology choices |
| 2 | claude-mem-lite-implementation-plan.md | Dependency graph, phase summaries, effort estimates |
| 3 | Key_Numbers_to_Keep_in.txt | Performance targets and constraints |
| 4 | prd-phase0-storage-layer-v2.md | SQLite schema, config, logging — the foundation |
| 5 | prd-amendments-phase0-phase2.md | Critical FK fix, Mermaid injection fix (read before implementing Phase 0 or 2) |
| 6 | prd-phase1-hooks-capture-v3.md | Hook scripts, Claude Code integration, data capture |
| 7 | prd-phase2-ast-tracker-v2.md | AST extraction, call graphs, change detection (can parallel Phase 1) |
| 8 | prd-phase3-worker-compression-v1.md | FastAPI worker, AI compression pipeline |
| 9 | prd-amendments-phase3.md | Critical: replaces os.fork() with subprocess.Popen, fixes structured output |
| 10 | prd-phase4-embeddings-search-v1.md | Qwen3 embeddings, LanceDB, hybrid search |
| 11 | prd-amendments-phase4.md | Async model loading fix, backfill wiring |
| 12 | prd-phase5-context-injection-v1.md | SessionStart context builder, token budgeting |
| 13 | prd-amendments-phase5.md | High: removes semantic search from Layer 4 at SessionStart |
| 14 | prd-phase6-learnings-engine-v2.md | Learning extraction, dedup, call graph self-healing |
| 15 | prd-phase7-eval-framework-v1.md | Offline compression evaluation, quality metrics |
| 16 | prd-amendments-phase7.md | Low-severity perf note, Sonnet pricing correction |
| 17 | prd-phase8-cli-reports-v1.md | Terminal UI, search CLI, mermaid export |

| Order | Document | Role |
|---|---|---|
| 18 | `prd-amendments-phase8.md` | Blocker: creates missing FTS5 virtual table |
| 19 | `prd-phase9-hardening-v1.md` | Production polish, deferred items backlog |
| 20 | `prd-amendments-phase9.md` | Dependency isolation for CLI-only installs |

**Rule**: Always read a phase's amendment document immediately after (or alongside) the base PRD. Amendments contain critical corrections — implementing a phase PRD without its amendments will produce bugs.

---

## Implementation Sequence and Rationale

### Phase 0: Storage Layer → Build first, no exceptions

**Why first**: Every other phase writes to or reads from SQLite. There is no useful work without the schema, config system, and logger. This is pure plumbing — no AI, no network, no external services.

**What it delivers**: SQLite tables (sessions, observations, function_map, call_graph, learnings, pending_queue, event_log), PRAGMA user_version migration system, Pydantic models, config management, JSONL + SQLite structured logging.

**Key amendment (phase0-phase2)**: Drop FK constraints on `function_map` and `call_graph` tables. Hooks fire in parallel — PostToolUse can precede SessionStart, so the referenced `sessions` row may not exist yet. Without this fix, you get silent `IntegrityError` crashes.

**Estimated effort**: ~1 session (4–6 hours)

---

### Phase 1: Hook Scripts + Direct Capture → First real integration with Claude Code

**Why second**: Once storage exists, you need data flowing into it. Phase 1 is the *only* way data enters the system — without hooks, there's nothing to compress, embed, search, or inject.

**What it delivers**: 4 hook scripts (PostToolUse capture, SessionStart context placeholder, Stop summary marker, SessionEnd cleanup), installer that registers hooks in `~/.claude/settings.json`, direct SQLite writes with no worker dependency.

**Key design decision**: Hooks write directly to SQLite, not through the worker. This eliminates the startup race condition that plagued the original claude-mem (issue #775 — context-hook fails when worker isn't ready). The worker processes the queue later; capture is never blocked.

**Critical v3 fix**: The `python3` shebang in hook scripts doesn't resolve correctly when installed via pipx. v3 writes `sys.executable` absolute path instead. Latency KPIs were also revised from impossible <10ms (Pydantic import alone costs more) to realistic <200ms.

**Estimated effort**: 1–2 sessions (6–10 hours)

---

**Phase 2: AST Tracker → Can parallel Phase 1**

**Why here**: Phase 2 depends only on Phase 0 (it needs `function_map` and `call_graph` tables). It has zero dependency on Phase 1's hooks. This means you can develop and test Phase 2 in parallel with Phase 1 if desired.

**What it delivers**: Python AST extraction (functions, methods, classes, signatures, decorators, body hashes), call graph construction with confidence-typed edges, change detection (new/modified/deleted/unchanged), Mermaid diagram generation.

**Why it exists at all**: The function map is Layer 2 of context injection (~500 tokens). It gives Claude a lightweight file overview without reading full source. This is novel — the original claude-mem has no AST tracking.

**Integration point**: After both Phase 1 and Phase 2 are complete, you wire AST scanning into the PostToolUse hook. This is a follow-up integration task, not part of either phase's core deliverable.

**Key amendment (phase0-phase2)**: Mermaid node IDs must be sanitized. Raw qualified names like `auth.service.AuthService.authenticate` contain dots that break Mermaid syntax. The amendment adds a `_safe_id()` helper.

**Estimated effort**: 2 sessions (8–12 hours)

---

**Phase 3: Worker Service + Compression → The AI pipeline begins**

**Why here**: Phase 3 needs Phase 0 (storage) and Phase 1 (data in the queue). This is the first phase that calls an external API (Claude Haiku 4.5 for compression). It's also the highest-risk phase — bad compression makes everything downstream useless.

**What it delivers**: FastAPI worker over Unix Domain Socket, queue processor (poll pending_queue → compress → store), AI compressor via Anthropic SDK, daemon lifecycle (PID file, idle timeout, health endpoint), session summarization.

**Critical amendment**: The original PRD used `os.fork()` for daemonization. This is unsafe with threaded libraries (aiosqlite, httpx, uvloop) and actively deprecated in Python 3.14. The amendment replaces it with `subprocess.Popen`, which is the correct modern approach.

**Second amendment**: The compression prompt should use Anthropic's structured outputs API (`response_format` with JSON schema) instead of hoping for valid JSON from free-form text. This eliminates an entire class of parsing errors.

**Why it's high-risk**: Compression quality determines whether the entire system is useful. A 10KB tool output compressed to 500 tokens must preserve *actionable information*. The PRD defines the prompt structure and quality criteria, but real prompt iteration happens during implementation. Budget extra time here.

**Estimated effort**: 2–3 sessions (10–16 hours)

---

**Phase 4: Embeddings + Search → Makes observations retrievable**

**Why here**: Requires Phase 3 (compressed observations to embed). This is where LanceDB enters the project and where the Qwen3-Embedding-0.6B model gets loaded.

**What it delivers**: Qwen3-Embedding-0.6B integration via sentence-transformers, LanceDB tables (observations, summaries, learnings) with vector + FTS indexes, hybrid search (vector + BM25 via RRF reranking), FTS-only fallback, search API endpoints, Claude Code SKILL.md for on-demand search.

**Key amendment**: Model loading (`SentenceTransformer(...)`) is a CPU-heavy 3–5s operation that was called synchronously inside the async lifespan. The amendment wraps it in `asyncio.to_thread()`. Also, `backfill_embeddings()` was defined but never called — the amendment wires it as a background task after startup.

**Technical note on LanceDB**: The PRDs describe LanceDB as "still Alpha (0.29.1)." This is partially outdated. As of Feb 2026, the Lance SDK core (Rust) graduated to 1.0.0, and the Lance file format 2.1 is stable. The Python wrapper (`lancedb` on PyPI) is at 0.29.2 — pre-1.0 versioning but actively used in production by multiple organizations. The risk is lower than the PRDs suggest, though API churn is still possible.

**Estimated effort**: 2–3 sessions (10–16 hours)

---

**Phase 5: Context Injection → The payoff**

**Why here**: Requires Phase 4 (search for relevant observations) and Phase 3 (worker + compressed data). This is where the system delivers its core value — Claude gets useful context from past sessions at SessionStart.

**What it delivers**: Progressive disclosure context builder with token budgeting (default 2000 tokens), 4 layers: session index (~300 tokens), function map (~500 tokens), active learnings (~300 tokens), relevant observations (~600 tokens). Updated context-hook that calls the worker. Dual-path: rich context when worker is available, basic context from SQLite when it's not.

**Critical amendment**: The original PRD used the *previous session's summary* as the semantic search query for Layer 4 observations. This causes context pollution when consecutive sessions work on different topics (e.g., "DB schema refactor" followed by "frontend CSS work" would inject SQL-related observations into the CSS session). The amendment removes semantic search from Layer 4 at SessionStart entirely, replacing it with recency-based retrieval. Semantic search is available on-demand via the Phase 4 SKILL.md.

**Estimated effort**: 1–2 sessions (5–8 hours)

---

**Phase 6: Learnings Engine + Call Graph Self-Healing → System gets smarter over time**

**Why here**: Requires Phase 3 (summarizer) and Phase 4 (embeddings for dedup). This is the "intelligence" layer — the system accumulates project-specific knowledge across sessions.

**What it delivers**: Learning extraction from session summaries via Claude API, dedup via semantic similarity (cosine > 0.90), confidence evolution (diminishing returns formula: `min(0.95, current + 0.2 × (1 - current))`),

contradiction detection embedded in extraction prompt, call graph self-healing (confirm/discover edges from observations).

**Why it's v2**: The v1→v2 review caught a serious bug — the contradiction logic penalized the old learning but never inserted the new one. The system forgot the topic entirely. v2 fixes this: penalize old, insert new, embed both.

**Estimated effort**: 2 sessions (10–14 hours)

---

**Phase 7: Eval Framework → Measure what you built**

**Why here**: Depends on Phase 3 (compressor to evaluate), Phase 4 (embeddings), Phase 5 (context injection to measure), Phase 6 (learnings). This is intentionally late — you need real data from actual usage before evaluation is meaningful.

**What it delivers**: Offline compression quality scoring (deterministic + LLM-judge), A/B model comparison via replay (not online routing — sensible for single-developer scale), SQL analysis queries for system monitoring, CLI entry point (`claude-mem eval`).

**Key design correction**: The implementation plan proposed online A/B routing (50/50 Haiku/Sonnet per observation). The PRD correctly rejects this — at ~50 observations/day, you'd need weeks for statistical significance while paying 3× more for the Sonnet half. Offline replay against stored raw outputs is cheaper, faster, and deterministic.

**Estimated effort**: ~1 session

---

**Phase 8: CLI Reports → Human-facing presentation layer**

**Why here**: Depends on nearly everything (Phase 0 storage, Phase 2 AST, Phase 4 search, Phase 6 learnings, Phase 7 eval). It's a read-only layer — it doesn't generate data, it surfaces it.

**What it delivers**: `claude-mem report` (session summary, function changes, learnings), `claude-mem search <query>` (CLI wrapper around hybrid search), `claude-mem mermaid` (call graph export), integration with Phase 7's eval commands.

**Blocker amendment**: The FTS5 fallback in the search command queries `observations_fts` — a table that doesn't exist anywhere in the schema. Phase 4 explicitly rejected SQLite FTS5 in favor of LanceDB's Tantivy. The amendment adds a migration to create the FTS5 virtual table with sync triggers, enabling search to work when the worker is down.

**Estimated effort**: ~1 session

---

**Phase 9: Hardening → Production readiness**

**Why last**: This is the deferred-items backlog from all previous phases. It's not a single deliverable — it's a prioritized list of edge cases, optimizations, and polish.

**Key items (Tier 1 — required for daily use)**: `pyproject.toml` finalization with extras groups (`[worker]`, `[dev]`), `prune` command for disk cleanup, error recovery for mid-session worker crashes and corrupt databases, ONNX INT8 quantization for Qwen3 (conditional on Phase 4 profiling showing >150ms latency).

**Key amendment**: When the CLI-only package is installed (without `[worker]` extras), the `compress --pending` command must not import `lancedb` or `sentence-transformers`. The amendment specifies dependency-isolated inline compression using only the `Compressor` class (which depends only on `anthropic` + `pydantic`).

**Estimated effort**: 2–3 sessions initially, then ongoing

---

## Dependency Graph (Corrected)


**Critical path**: 0 → 1 → 3 → 4 → 5

This is the minimum viable path to a working memory system. After Phase 5, Claude gets context at SessionStart. Everything else enhances quality, observability, or robustness.

---

## Independent Review: Issues and Observations

The PRDs are thorough and the amendment process caught real bugs before implementation. That said, reviewing with fresh eyes and against current standards, several things stand out.

### 1. The "lite" framing vs. actual complexity

The project is called "claude-mem-**lite**" but specifies 10 phases, 50,000+ words of PRD documentation, and 15–

19 implementation sessions. The dependency stack includes FastAPI, uvicorn, LanceDB, sentence-transformers, PyTorch (transitive from sentence-transformers), the Anthropic SDK, aiosqlite, and a 1.2GB embedding model.

For a single-developer personal tool, this is substantial. The original claude-mem that this is meant to simplify uses Express.js + ChromaDB + SQLite — arguably a lighter runtime footprint than what's specified here (PyTorch alone is hundreds of MB).

**Not necessarily wrong** — the architectural choices are individually defensible. But calling it "lite" sets expectations that the implementation doesn't match. This matters because scope creep in personal tools often means they never ship.

**2. Python 3.14 minimum is aggressive but defensible**

Requiring Python ≥3.14 (released Oct 7, 2025, now at 3.14.3) is bleeding-edge. Most CI systems, Docker base images, and developer machines still default to 3.12 or 3.13. However, since this is a personal tool (not a library others install), and since 3.14 is now in stable bugfix releases, the choice is defensible. Just be aware that any collaboration or machine migration requires 3.14+.

**3. Over-specification before prototyping**

The project has comprehensive PRDs with specific line numbers, exact API calls, and code snippets — yet no working code exists. The amendments demonstrate this problem: the review process found FK constraint race conditions, a non-existent FTS5 table, a daemonization approach incompatible with Python 3.14, and prompt assumptions that don't match the actual SDK API.

These are problems that a 2-hour prototype of Phases 0+1 would have surfaced naturally. The amendment process works as a safety net, but the cost of writing and reviewing 50,000+ words of spec before writing any code is significant for a solo developer.

**Suggestion**: Consider implementing Phases 0–1 as a spike *before* finalizing the remaining PRDs. Real data from actual hook latency measurements and queue behavior would ground Phases 3+ in reality rather than assumptions.

**4. LanceDB risk is lower than stated**

The PRDs repeatedly flag LanceDB as "still Alpha (0.29.1)." As of Feb 2026, the Lance SDK core (Rust) reached 1.0.0, the file format 2.1 is stable, and multiple organizations deploy LanceDB in production. The Python wrapper version number is pre-1.0 but stable releases ship every 2 weeks with regression testing. The risk is real (API surface may shift) but overstated in the docs.

**5. Qwen3-Embedding model size vs. alternatives**

The 0.6B model is ~1.2GB FP16 and takes 3–5 seconds to load. For a tool that starts a worker per coding session, this is noticeable. The architecture doc dismissed `gte-modernbert-base` (ModernBERT) as "4 points lower on code (71.1 vs 75.0)" — but ModernBERT loads in <1 second and runs at 15–25ms per embedding vs. 80–150ms for Qwen3.

For a personal memory tool where the embedding corpus is small (hundreds to low thousands of observations), the quality gap between 71.1 and 75.0 on MTEB-Code benchmarks is unlikely to produce noticeably different search results. The latency gap, however, is very noticeable every session start.

**This is a judgment call, not a clear mistake.** But if Phase 4 latency testing shows the Qwen3 load time is painful, switching to ModernBERT would be a low-risk fallback. The Phase 9 ONNX INT8 quantization is another mitigation, but it adds build complexity.

**6. Amendment documents are essential — not optional reading**

The amendments contain blocker-severity fixes. Implementing Phase 3 without the os.fork() → subprocess.Popen fix will produce deadlocks. Implementing Phase 8 without the FTS5 migration will produce dead code. The amendments must be treated as part of their respective PRDs, not as separate optional addenda.

---

## Recommended Approach for Implementation

Given the project's scope, the realistic path to a usable tool is:

1. **Implement Phases 0–1 first** as a proof-of-concept. Get hooks firing and data landing in SQLite. Measure real latency. This validates or invalidates assumptions across all subsequent PRDs.
2. **Implement Phase 3 next** (worker + compression). This is the highest-risk phase. If compression quality is poor, the entire system is wasted effort. Invest time in prompt iteration here before moving forward.
3. **Implement Phase 4** (embeddings + search). At this point you have a useful system — compressed observations that are searchable.
4. **Implement Phase 5** (context injection). Now Claude gets context at SessionStart. This is the MVP — the minimum viable product that delivers the core value proposition.
5. **Phase 2** (AST tracker) can slot in whenever you have bandwidth. It's independent and doesn't block the critical path. Wire it into the hooks after both Phase 1 and Phase 2 are done.
6. **Phases 6–9** are enhancements. Implement them as the need arises from actual daily use, not speculatively.

**Total effort for MVP (Phases 0, 1, 3, 4, 5)**: ~8–13 sessions. This gets you a working memory system. Everything else makes it better.