ChatGPT

Manny Manifolds v0.1 – Implementation Specification

1. Architecture Overview

System Summary: Manny Manifolds is a modular cognitive engine composed of distinct layers for real-time graph traversal, offline learning, language model (LLM) assistance, user interface, and storage. The architecture strictly separates the **core geometry engine** (graph-based manifold and physics), **experiment harness** (for testing multiple "physics" variants), **LLM accelerators** (for language-centric tasks like embedding and summarization), the **UI layer** (visualization and user commands), and **persistent storage** (SQL-backed database and logs). This separation ensures that core algorithms (the "physics") can be hot-swapped or extended without altering interface contracts.

Layered Design: Manny runs in **dual phases** [1] – an **online phase** for interactive query threads with strictly local computations, and an **offline phase** for global consolidation (the "sleep" cycle). In the online phase, a **Thread Runner** module executes user queries as paths through the graph, updating edge curvatures locally (within a k-hop neighborhood) without any global recomputation [2]. In the offline phase, a **Consolidation Scheduler** triggers heavy jobs (e.g. re-embedding, pruning, motif mining) in batch, ensuring that expensive operations do not occur on the real-time path (upholding the *Locality* constraint – **REQ-001**: no global recompute during interactive queries). Between these phases, Manny's *Bicameral* design fosters stability: an **Experiencer** process handles online exploration, while an **Executive** process handles offline integration and self-regulation [3].

External Interfaces: The Manny engine exposes a stable **UI Backend API** (HTTP/JSON endpoints) for the front-end and a command-line/LLM interface for conversation. The UI (single-page web app) communicates with Manny's backend to display the **graph map**, overlays (lenses, valence fields), motif libraries, and explainable paths in real time. The experiment harness can invoke core APIs or internal hooks to run A/B tests across different physics variants. A high-level diagram of the system's modules and data flow is shown below:

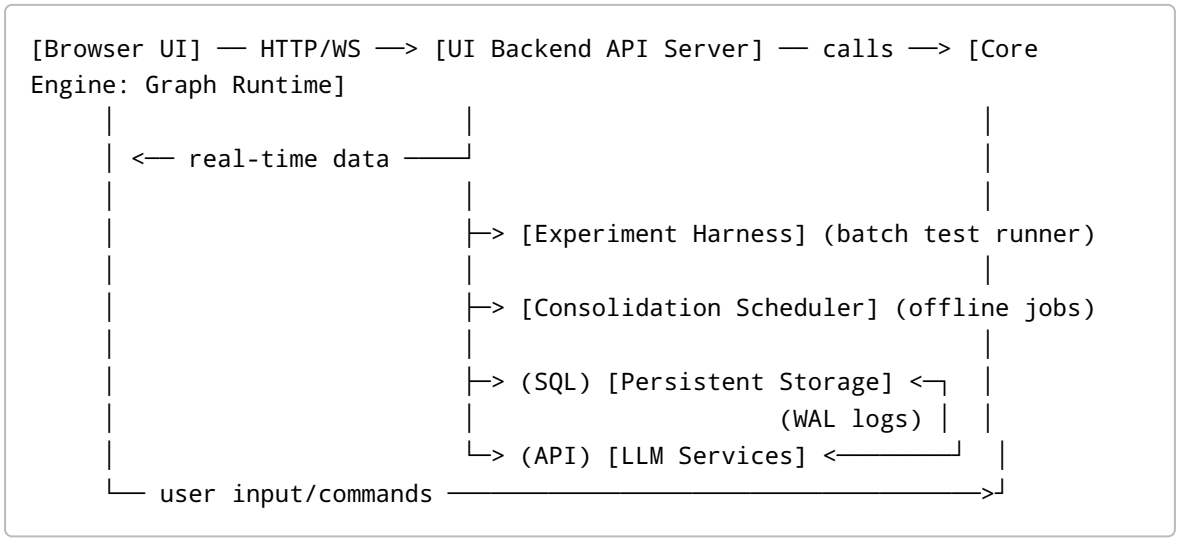

Fig.1: High-level Architecture. The Core Engine interacts with external components via clearly defined APIs. The UI sends queries and receives updates through the UI Backend; the Experiment Harness triggers controlled runs; the Consolidation Scheduler writes to/read from Storage; and LLM calls (e.g. for embeddings or summarizations) are made via a limited interface.

Core Surfaces: To maximize extensibility, Manny defines stable **interfaces** (core surfaces) between modules. These remain constant even if internal "physics" formulas change. Key immutable contracts include: `IPhysicsModel` (the plug-in for energy and learning rules), `ITraversalEngine` (graph search executor), `ILensModel` (contextual projection metrics), `IMotifModel` (motif detection/usage logic), `IConsolidationJob` (offline maintenance tasks), `IExplainabilityRecorder` (logging for `/why`), and `IUIBackendAPI` (endpoints for UI/ops). These APIs isolate experimental "swappable" logic behind a fixed interface boundary. For example, any new curvature update rule must implement `IPhysicsModel` but can be inserted without affecting UI or storage code. This ensures **rapid experimentation** – multiple physics variants can be hot-swapped – while preserving a consistent outward behavior. **REQ-002:** All core interface contracts shall remain backward-compatible and deterministic in I/O, so that experiments and UIs built on v0.1 will continue to function with future physics plugins.

Non-Negotiable Design Constraints: The architecture encodes Manny's fundamental "laws" as requirements:

- **Locality (No global online compute):** The traversal engine only explores a bounded k-hop neighborhood per query [2]. No full graph recomputation occurs in the hot path (enforced by design – e.g. heuristics use local edge weights, and global re-indexing is deferred to offline jobs). *This prevents latency blow-ups and preserves locality of learning (REQ-001).*
- **Dual Dynamics:** Manny strictly separates *online updates* (real-time Hebbian-style edge tweaks) and *offline consolidation* (batched reorganization) [4]. This dual phase design is built-in (the Consolidation Scheduler is the only component allowed to perform global adjustments, and it runs asynchronously or during idle periods).
- **Explainability:** Every answer is accompanied by a reproducible *why-path*. The engine records exactly which nodes and edges were traversed, including curvature changes (Δκ) and lens/context used at each step. The UI's `/why` playback reads this log to faithfully replay the reasoning path with stepwise costs and provenance [5] [6]. *REQ-003:* The explanation subsystem must reflect the true execution trace (no post-hoc "made-up" rationale), ensuring that users can trust Manny's transparency.
- **Stability under Continual Learning:** Mechanisms for plasticity control (learning rate η, decay, curvature clamping) are embedded at the core. Curvature updates are **clamped and budgeted** per interaction to avoid runaway feedback [7] [8]. A small learning rate and bounded Δκ per turn (e.g. ±0.06) with global curvature limits (e.g. $\kappa \in [-1.5, +1.5]$) are default safeguards [9] [8]. The Consolidation phase also applies slight decay to unused edges to prevent long-term drift. *REQ-004:* The system must enforce curvature update limits and periodic decay to maintain stable behavior over long runs (no unbounded weight growth or catastrophic forgetting [10]).
- **Implementability & Determinism:** Manny is designed for practical implementation with today's tools: an **SQL-backed** persistent store and an **in-memory graph runtime** for fast operation. All critical operations have deterministic modes for testing (randomized components can be seeded). The initial target scale (≈10^4–10^5 nodes) is easily handled in memory, and standard libraries (e.g. NetworkX or a custom graph class, FAISS/HNSW for ANN search, HuggingFace or OpenAI API for LLM) are used [11] [12]. *REQ-005:* The system must run on conventional hardware (single multi-core server) deterministically (when seeded) and allow exact replays of runs for debugging or experiment comparison.

2. Data Model and Persistence

Knowledge Graph Schema: Manny's knowledge **manifold** is stored as a graph with *Nodes* (concepts) and *Edges* (relationships) augmented by dynamic weights. Each node and edge carries a **curvature (κ)** value representing the strength or "learned valence" of that entity or connection [13]. High positive κ indicates a well-traveled, strong association (low traversal cost), whereas a low or negative κ indicates a weak or oppositional link. The persistent data model is relational (SQL) for reliability and querying, with the following schema (normalized into tables):

- **Nodes:** `node(id PRIMARY KEY, label TEXT, embedding BLOB, type TEXT, created_at TIMESTAMP, ... )` – Represents a concept or state. Each node has a unique ID and a human-readable label. An **embedding vector** (e.g. 768-dim) is stored for semantic similarity operations (used by the LLM lens for matching) [14] [15]. The *type* or category field can indicate domain or lens affiliation. Only lightweight attributes are stored here (the embedding may be stored as a blob or in a separate vector index table). Frequently accessed node data (label, embedding) is cached in-memory on load.

- **Edges:** `edge(id PRIMARY KEY, src_id INT, tgt_id INT, relation_type TEXT, curvature FLOAT, base_weight FLOAT, valence_channel SMALLINT, created_at TIMESTAMP, updated_at TIMESTAMP, usage_count INT, motif_id INT NULL)` – Represents a directed relation from `src` to `tgt` node (or undirected if stored symmetrically). The *relation_type* is a semantic label (e.g. "causes", "part_of", "similar_to"). **Curvature (κ)** is the dynamic strength: positive κ lowers traversal cost between those nodes (acting like a "shortcut") [16]. *base_weight* can store a static distance or initial weight (e.g. an initial semantic distance or 1 for unweighted) – the traversal *cost* is calculated from base distance minus an *ακ term (see §3).* valence_channel *indicates which channel of valence (e.g. importance, novelty, affect) last updated this edge, for multi-channel learning (this can be extended to multiple columns if multi-channel valence is stored per edge).* usage_count *tracks how often this edge was traversed (for motif mining and decay decisions).* motif_id* links to a Motif if this edge is part of a cached motif pattern.

- **Motifs:** `motif(id PRIMARY KEY, nodes TEXT, edges TEXT, frequency INT, last_used TIMESTAMP, label TEXT)` – Represents a frequently used subpath (a "chunked" reusable pattern). The `nodes` and `edges` fields store the composition of the motif (e.g. as a JSON or delimited list of node IDs and edge IDs in sequence). *frequency* records how many times it was mined or used. *label* may store an LLM-generated description of the motif (for explainability, e.g. "baking process sequence") if available. Motifs act like macros: the traversal engine can treat them as virtual shortcuts during search. **Note:** In the in-memory graph, a motif might be instantiated as a meta-edge or precomputed route for fast traversal.

- **Lenses:** `lens(id PRIMARY KEY, name TEXT, definition JSON, created_at TIMESTAMP, usage_count INT)` – Represents a contextual "projection" or view. *definition* can store lens parameters (e.g. a vector defining a subspace, or a set of nodes that define this lens's domain focus, etc.), as well as *affinity metrics* (if precomputed). Lenses may emerge offline; their storage allows persistence across sessions. (Lenses influence traversal by altering local distance metrics or node visibility; see LensModel in §3).

- **Threads (Queries):** `thread(id PRIMARY KEY, start_node INT, goal_node INT, lens_id INT, timestamp, result TEXT, success BOOLEAN)` – Each user query or reasoning thread is logged. *start_node* and *goal_node* capture the entry/target (if applicable; some queries are open-ended or goal-less, in which case goal may be null or a descriptive tag).

lens_id is the lens in use (or null for default lens). *result* and *success* store the outcome (answer text or found node, and whether it was successful). This table is mainly metadata; detailed step logs are in **ThreadEvents**.

- **ThreadEvents (Path Steps):** `event(id, thread_id, step_index INT, src_node INT, tgt_node INT, edge_id INT, cost FLOAT, Δκ FLOAT, lens_id INT, timestamp)` – This is an append-only log of each hop taken during a thread traversal. It captures the exact path: each step from `src_node` to `tgt_node` via an edge, with the *cost* incurred and any curvature change (Δκ) applied to that edge due to learning. If lens switching occurred at this step, the lens is recorded. This event log enables the **/why** playback: the UI reconstructs the path and shows per-step costs and lens tags from these records. It also provides provenance for explainability (e.g. which edge was critical) [6]. The `event` table, combined with `thread`, forms a **Write-Ahead Log (WAL)** for reasoning traces.

- **Knowledge Ingestion Log:** `ingest_log(id, source TEXT, content TEXT, llm_used TEXT, new_nodes INT, new_edges INT, timestamp)` – Logs knowledge added via ingestion commands (e.g. user uploads a document or defines a new fact). It tracks how many nodes/edges were created and which LLM (if any) was used to parse it. This provides provenance for content and helps in debugging or re-processing (e.g. re-running ingestion with updated parsers).

Persistence & In-Memory Runtime: The **SQL database** (e.g. PostgreSQL or SQLite for MVP) is the source of truth for all nodes, edges, and artifacts. On startup, Manny loads the graph into an in-memory data structure (adjacency lists, embedding index, etc.). **In-memory graph runtime** holds: node objects (with ID->embedding map), edge lists keyed by source (for fast neighbor lookup), and indexes for quick retrieval (e.g. a hash index on node labels for search, and a vector index for nearest neighbors by embedding). High-frequency reads (traversal neighbor expansions) use the in-memory structure, while writes (curvature updates, new edge/node creation) update both in-memory and persist to the database (within a transaction or via the WAL).

Write-Ahead Logging & Snapshots: All online updates are appended to a **WAL** (which can be the `thread_event` table or a dedicated `update_log` table capturing each Δκ or structural change). This log provides durability (for crash recovery) and reproducibility (for experiment replay). For example, if Manny's process crashes, on restart it can replay the update log from the last snapshot to recover the exact state (ensuring deterministic recovery). Periodically, the system takes a **snapshot** of the full graph state – essentially writing all current node and edge values (and possibly a dump of in-memory indices) to the database with a version tag. Snapshots can be triggered on-demand (via the Ops UI) or scheduled (e.g. nightly). Each snapshot is identified by a version (schema includes something like `snapshot(id, timestamp, description)` and possibly separate snapshot tables, or simply a full export to disk). The system can **compare snapshots** to see how the graph evolved (e.g. number of edges grown, curvature distribution) – this is enabled via either SQL diff queries or by loading two snapshots and computing differences (the UI provides a comparison tool; see §6).

Schema Versioning & Migration: The schema is versioned (e.g. a metadata table `schema_version` or an internal version in config). Any changes to schema (like adding a new table or column for a new feature) follow *backward-compatible rules*. **REQ-006:** New fields must have defaults so that old data can still be read by newer code, and migrations should be provided for any breaking changes. For example, if multi-channel valence expands to multiple columns, a migration script will add those columns with default neutral values. The code will check `schema_version` on startup; if an older version is detected, it will run migration routines (which are idempotent and logged) to upgrade the database.

Backward compatibility is maintained by not renaming or repurposing existing fields – instead, deprecate old fields gradually and leave them for compatibility if needed. All persisted artifacts (graphs, logs) can thus be loaded by future versions, supporting long-term experiments.

In-Memory vs Persisted: To balance performance and persistence, the spec defines which data reside in memory and which in the database:

- *In Memory:* Active graph structure (node and edge objects with current κ), embeddings for active nodes (for fast similarity search), current motif patterns (for quick matching during traversal), and any ephemeral caches (e.g. LLM answer cache). These are rebuilt or loaded at startup from the DB or snapshot files. The in-memory state is the working state for queries.

- *Persisted in SQL:* All durable state – full node list, edges with their latest κ, permanent motif definitions, lens definitions, and all logs (thread events, ingestion logs). After each query, critical updates (curvature changes, new edges/nodes) are written through to the DB (synchronously for now, or batched within a short transaction). Less critical structures like the ANN index can be recomputed offline rather than constantly saved (the HNSW index, for example, might be rebuilt each consolidation, so we need not persist the index itself, just the underlying embeddings [17]). The persistent store is append-only for logs to preserve history.

Indexing for Performance: The SQL schema includes indexes to ensure query efficiency: e.g. an index on `edge(src_id)` for neighbor lookups, on `edge(curvature)` if we query highest-curvature edges (for motif mining), and on `thread_event(thread_id, step_index)` for retrieving path in order. Full-text search index on node labels might be added for quick node lookup by name. The **vector index** for embeddings is maintained separately (in-memory via HNSW, and optionally persisted to disk as a file using an ANN library's serialization). Manny's consolidation will periodically rebuild or update the ANN index of node embeddings to speed up nearest-neighbor queries [12]. For example, after ingesting new knowledge or after a large curvature change that could shift similarity, the ANN index is refreshed (either fully rebuilt for MVP or incrementally updated if supported).

Reproducibility & Logging: *REQ-007:* The system must support deterministic replay of its learning. Therefore, every source of non-determinism (random tie-breakers in traversal, randomness in any LLM outputs) is either recorded or seeded. Random choices in traversal (if any) can be made deterministic by a fixed seed per thread (logged in thread record). LLM calls are logged along with their inputs and outputs in `ingest_log` or a `llm_call_log` (including the prompt, model used, and returned content hash) [18] [19]. This allows results to be cached and reused exactly, or for offline verification. The **combination of snapshot + WAL logs** yields a complete picture of state changes – one can take a snapshot and replay subsequent logs to reconstruct the manifold's state at any intermediate point.

Backup and Recovery: Regular backups of the SQL database and log files are scheduled (with snapshot files, if using external storage for snapshots). Recovery procedures are documented: in case of data corruption, one can restore the last good snapshot and then replay WAL entries. Because Manny is an append-mostly system (edges accumulate curvature gradually, new edges/nodes are added but rarely deleted outright except via consolidation), conflict resolution on recovery is simple – the latest log entry wins if there's any discrepancy.

3. Core Runtime Contracts

To support hot-swappable "physics" and ensure module boundaries, Manny defines strict **interfaces (contracts)** for core runtime components. Each interface specifies its inputs/outputs, performance

budget, determinism, and error handling. These interfaces form the **Spec Backbone** of the system (immutable across versions – see §9). Implementations can vary (different algorithms or parameters) but must conform to these contracts:

3.1 `IPhysicsModel` – Physics and Learning Rules

Role: Encapsulates the "laws of motion" in the manifold – the energy function, traversal cost calculation, and curvature update (learning rule). This module defines how threads propagate and how experience updates the graph.

- **Methods:**

- `compute_cost(u: Node, v: Node, edge: Edge, lens: Lens) -> float`: **Deterministic.** Given a potential step from node `u` to neighbor `v` (via a specific edge), return the *traversal cost*. The cost combines base distance, curvature, and other fields: e.g. `cost(u,v) = distance(u,v) - α·κ(u,v) + β·novelty(v) - γ·(G(v)+U(v))` [16]. Here `κ(u,v)` is the edge curvature, `distance(u,v)` could be a static base weight (or embedding distance), `novelty(v)` is a novelty bonus (e.g. visiting new information has some cost reduction or increase depending on design), and `G(v), U(v)` represent goal attraction and uncertainty penalty respectively [20] [21]. The PhysicsModel must implement this formula or equivalent, using internally defined global constants (α, β, γ, etc.). **Determinism:** Given the same inputs (and same global parameters), cost must be identical – no randomness here. **Performance:** This is on the hot path for each neighbor expansion, so it should be O(1) and extremely fast (simple arithmetic on the fields). **Error Handling:** If an input edge or node is missing or invalid, throw a descriptive error (or return +∞ cost to indicate an impossible move). Typically, cost should be a finite float; if any component is undefined (e.g. novelty for a known node might be 0), define defaults rather than error.

- `update_curvature(path: [Edge], total_valence: float) -> List[(Edge, Δκ)]`: **Deterministic** (assuming path and valence are given). Apply the learning rule to update curvatures along a traversed path. This method takes the sequence of edges that were used in a thread and the total *valence* of that experience (which may be a scalar or vector of channels). It computes Δκ for each edge (often proportional to η * valence * some function of usage [22] [23]). For example, a simple rule might increment each edge's κ by `η * (valence_importance) / path_length`. The function must also apply *clamping*: after updating, if any edge's κ exceeds preset bounds, clamp it to the max/min (and maybe record that event). It returns the list of edges with their Δκ applied (for logging). If the model uses multi-channel valence, the rule may weight updates differently for each channel (e.g. novelty valence might boost new edges more, etc.). **Determinism:** The same path and valence produce the same Δκ outcomes. **Performance:** Should be O(L) for path length (which is typically small), trivial computation per edge. **Error:** If an edge is not found (should not happen since path edges came from traversal), log and skip. If valence is out of expected range, cap it or normalize internally (never cause NaN).

- `apply_decay()`: **Deterministic**. Performs a single decay step on curvatures (and possibly node valences). This might be called periodically (e.g. at thread end or during consolidation for micro-decay). It can implement exponential decay or a small linear regression toward 0 for all edges that haven't been recently used. *Time/space:* It could iterate over all edges, but for online usage this must be lightweight – possibly only a random sample or a subset flagged as low usage (the Consolidation job can do full decay on all edges, see below). For MVP, this might not

be invoked each time, but the interface exists for consistency. **Error:** none expected; if not used, implementation can be a no-op.

- `init_curvature(edge: Edge) -> float`: Returns an initial κ value when a new edge is created (e.g. from knowledge ingestion). Typically 0 or a small neutral value so that initially the edge doesn't bias traversal strongly. By funneling through the PhysicsModel, we ensure any new edges introduced by LLM or user have consistent initialization (and can incorporate any global strategy, like slight positive bias for user-taught edges to encourage exploration).

Determinism & Testing: The PhysicsModel must guarantee consistent results given same inputs. For testing, we will feed known scenarios (toy graphs) to ensure that, for example, `compute_cost` yields lower cost for edges with higher κ (i.e. verifying the sign of κ in cost is negative as designed – **DEC-1:** *Edge curvature reduces traversal cost, enabling high-κ links to act as "shortcuts"* [16].). Also verify that `update_curvature` correctly clamps values (e.g. no edge's |κ| exceeds 1.5 if that's the spec) [8]. Errors in this module (like cost returning infinite or NaN) are caught by the TraversalEngine which will treat them as impassable edges.

Time/Space Budgets: `compute_cost` is called potentially thousands of times per query (if exploring 3-hop neighborhood with average degree d, worst-case ~d^3 calls). It should thus be constant time and use only node/edge attributes (no heavy computation). `update_curvature` is called per thread (once) on at most path-length edges (typically dozens), negligible overhead. So PhysicsModel should not become a bottleneck.

3.2 `ITraversalEngine` – Thread Runner

Role: Executes a query (thread) by finding a path through the graph from a start to a goal (or towards an information target), following the PhysicsModel's cost metrics. It encapsulates the search strategy (e.g. goal-directed greedy search, beam search, or stochastic traversal with "energy" gradients). The traversal engine can be swapped to test different navigation strategies (greedy vs. lookahead).

- **Method:**

- `execute_thread(start: Node, goal: Node|Query, lens: Lens, budget: int) -> ThreadResult` – Runs a traversal from `start` toward `goal` under a given lens context and step budget. **Inputs:** a start node (or multiple start nodes if query is broad), a goal specification (could be a specific target node or a query embedding to satisfy), a lens context (which influences cost via the PhysicsModel or filters neighbors), and a step budget (max number of hops to explore). **Operation:** The engine performs iterative expansion: at each step from current node `x`, fetch the local neighborhood (all edges from `x` within k hops, typically k=1 for stepwise expansion) – crucially **only local neighbors are considered (Locality)** [2]. It calculates the *potential* or "next move score" for each neighbor using `IPhysicsModel.compute_cost` plus lens adjustments (LensModel below). It then chooses the next node according to its strategy: e.g., **Greedy Geodesic** – pick the neighbor with minimum cost (taking into account goal attraction and lens) [21] [24]; or **Beam Search** – keep N best frontier nodes. The engine accumulates the path until either the goal is reached (some node matches the query goal) or budget steps are exhausted. **Outputs:** A `ThreadResult` structure containing the path found (`[Node...]`), the edges used (`[Edge...]`), total cost, and status (success or not). This structure also includes the collected step-by-step details for explainability (possibly via integration with ExplainabilityRecorder – see below). If no path is found within budget or goal not reached, it may still return the partial path or an empty path with `success=False`.

Determinism: The traversal may involve tie-breaking or randomness (e.g. if two neighbors have equal cost, or if using a stochastic approach). For reproducibility, the engine must support a deterministic mode: any randomness is seedable (e.g. a thread ID based seed). By default in experiments, we run in deterministic mode. In interactive use, a bit of randomness could be allowed for exploration (e.g. adding small random noise to cost to explore alternative paths), but this is optional. The implementation should log any random decisions (so they can be replayed if needed).

Time Complexity: The engine must respect the budget (`budget` is the max steps or expansions). Also, Manny's *k-hop locality law* is enforced: no search beyond k hops from the current path when choosing next step (k typically 2–3 for small lookahead) [25]. This means the traversal is essentially *greedy local* unless specified otherwise. The worst-case explored nodes is limited by budget * average degree^k. For MVP, we assume manageable graph size so a greedy or beam search is fine. If performance issues arise at scale, we may incorporate heuristics or even specialized graph search libraries, but all under this interface.

Integration with Other Modules: The TraversalEngine uses `IPhysicsModel` for cost evaluation. It also queries `ILensModel` for any lens-specific metrics (see below) – for example, before choosing a neighbor, it might check lens affinity scores to penalize nodes outside the current lens. If a lens switch is possible, the engine consults `LensModel` whether switching yields lower cost (and includes lens friction cost ζ) [26] [27]. The engine also can query `IMotifModel` during traversal: e.g., at a given node, check if there is a motif starting here that directly reaches the goal or a later step – if yes, it can "fast-forward" through that motif as a single step (treating motif as a macro-edge). This can dramatically speed up finding answers using learned shortcuts [28] [29]. The engine must therefore have hooks: `MotifModel.suggest_jump(current_node, goal)` returning a suggested motif path if applicable. If such suggestion exists, the engine can decide to jump (if it seems beneficial cost-wise).

Explainability Hooks: As the engine progresses, it calls `ExplainabilityRecorder.record_step(...)` at each hop to log what decision was made, including costs, lens info, and Δκ applied later. This imposes minimal overhead (just logging to memory) but is crucial for /why.

Error Handling: If at any point the engine detects an inconsistency (e.g. a neighbor edge with missing node, or cost = NaN), it logs a warning and skips that neighbor. If no path found and goal is required (like a factual question that must have an answer), the engine will indicate failure; higher layers might then escalate (e.g. ask the LLM for help, but that's outside core engine).

3.3 `ILensModel` – Contextual Lens Mechanics

Role: Lenses define dynamic subspaces or perspectives that modify how the manifold is traversed. The LensModel provides calculations for lens *affinity* and manages lens switching friction. Lenses can filter or weight edges/nodes to simulate context (e.g. a "Cooking" lens highlights cooking-related nodes, making other edges effectively longer).

- **Methods:**

- `affinity(node: Node, lens: Lens) -> float`: Returns how well the given `node` fits the given `lens` context (e.g. a similarity score). For instance, if lens is defined by a topic vector or a cluster of nodes, this could return a normalized score 0–1 or a cost offset. A high affinity means the node is in-context. The TraversalEngine can incorporate this as a penalty for low-

affinity nodes (like adding cost if out-of-lens). **Determinism:** purely deterministic (usually a simple function of node's domain tag or embedding vs lens embedding).

- `project(node: Node, lens: Lens) -> Vector`: (Optional) Returns the projection of the node's features into the lens-specific coordinate system. This might be used if each lens has its own vector space for distance. For example, a lens might prioritize certain embedding dimensions. The engine could use this to compute distances under the lens. If not needed, can be no-op or identity.

- `allow_switch(current_lens: Lens, new_lens: Lens, at_node: Node, ΔE_new: float, ΔE_old: float) -> bool`: Determines if a lens switch should occur at a given point. The engine may evaluate the potential energy drop (or cost improvement) if it were to switch to a different lens `new_lens` at this node [27]. The LensModel applies the **switching friction ζ** – a threshold or cost for switching [25] [30]. *If* `ΔE_new < ΔE_old - ζ`, it allows switch (meaning new lens offers sufficiently lower energy to justify cost). This method encapsulates that logic. It returns true if switching is permitted (and presumably the engine will then update the active lens and incur some cost). **Determinism:** yes (no randomness in switching criteria). The friction ζ might be dynamic (LensModel could increase ζ if too many switches recently to prevent thrashing [31]), but that should be based on deterministic counters.

- `create_lens(definition: any) -> Lens`: Creates a new Lens object from a definition (e.g. from motif clusters or user input). This interface allows the ConsolidationJob or other parts to register new lenses emergently (for example, if it detects a cluster of nodes frequently co-activated, it may create a lens for that domain). The lens definition could include a representative vector or list of nodes. The LensModel could assign an ID and initial parameters.

Performance: Lens affinity checks must be fast (likely O(1) or O(n_dims) if using dot product with an embedding). They are used at traversal steps to adjust costs. Lens switching logic is invoked at nodes where multiple contexts intersect – frequency depends on scenario but should be sparse (only when multi-domain queries).

Determinism & Extensibility: All lens behaviors are fully reproducible; any adaptive aspects (like friction increasing if thrashing) should rely on measurable events (e.g. count of switches in last N steps) and be documented. New types of lens projection can be introduced by implementing this interface differently (the core engine only calls these methods, not caring how lens affinity is computed internally). The *core surfaces* thus include the lens contract, but not its innards.

Error Handling: If a lens is not applicable (e.g. `lens=None` meaning default lens), the methods should handle gracefully (affinity might return a neutral value like 0 or 1 for all nodes, allow_switch should probably return false if no alternative lens). If an unknown lens ID is provided, log error and treat as no lens.

3.4 `IMotifModel` – Motif Discovery and Utilization

Role: Handles identifying frequently-used subgraphs ("motifs") and leveraging them to accelerate reasoning. It operates in two modes: **offline motif mining** (finding new motifs from logs) and **online motif activation** (injecting motif knowledge into traversal).

- **Methods:**

- `mine_motifs(activity_log: ThreadEvents[]) -> List[Motif]`: Offline method (called during consolidation, see §3.6) that analyzes recent thread activity or the entire event log to identify candidates for motifs. The implementation might count common sub-paths or look for recurring sequences of edges above a frequency threshold [28] [29]. It returns a list of new Motif definitions (which can then be inserted into the `motif` table and possibly materialized as shortcut edges in memory). **Determinism:** Given the same activity log and parameters, it finds the same motifs. If it uses any randomness (unlikely, usually frequency-based), it must fix a seed for reproducibility. **Performance:** Potentially heavy (subgraph mining is NP-hard in general), but we control it via thresholds and maybe incremental updates. For MVP, a simple approach: look at every pair of threads and see if they share a contiguous sequence of nodes of length $\geq$ L, and count frequency. We limit mining to top-K frequent sequences or use algorithms like FP-growth on the sequence of node traversals. The result count is manageable (also we can tune threshold to get maybe dozens of motifs, not thousands).

- `suggest_motif(current_node: Node, goal: Node) -> MotifMatch or None`: Online helper used by TraversalEngine. Given the current node (where the thread is) and the goal, this checks if there is a known motif that *starts* at the current node (or nearby) and leads closer to the goal. For example, if there is a motif "A→B→C→D" and the current node is A, and goal is D or beyond, the model might suggest using that motif. The return could be a `MotifMatch` containing the motif ID and the remaining path it covers. TraversalEngine can then jump directly, incurring presumably the sum of costs or a discounted cost (since motif is like a well-known shortcut, we might give it a lower cost). **Determinism:** yes, purely a lookup of known motifs (perhaps an index mapping start_node->motifs). **Performance:** O(1) to check a hash map of motifs by start, or O(k) if multiple motifs from a node (k small). This drastically reduces pathfinding time if applicable because the engine doesn't need to search those intermediate hops.

- `use_motif(motif: Motif) -> Path`: (Optional) Could be an expansion of motif into an explicit path. Or integrated in suggest_motif logic.

- `prune_motifs(criteria)`: (Offline) Remove or mark motifs that are no longer useful (e.g. if the edges in them decayed or if frequency dropped). Called during consolidation to keep the motif set relevant and not growing unbounded. We define criteria like: if a motif hasn't been used in N cycles or its edges have all decayed below a curvature threshold, drop it.

Determinism & Logging: The identification of motifs is recorded (we log when a motif is created, and how – possibly linking it to example threads that yielded it). This helps explain to users "why" the system created that shortcut (part of explainability: Manny can say *"I learned this motif after we solved similar problems 3 times"*). The motif usage is also tracked (each time `suggest_motif` returns one and it's used, increment its usage_count in DB).

Inputs & Outputs: `mine_motifs` uses *ThreadEvents* logs (or some condensed form like sequences of node IDs for each thread). It outputs new *Motif* records with a set of edges/nodes, and possibly attaches a label via LLM (the ConsolidationJob might call an LLM to summarize the motif – e.g. turning a sequence of nodes into a description, "baking process"). The *LensModel* might also use motifs to suggest new lenses if motifs indicate a domain. But at this level, MotifModel itself doesn't create lenses (though an advanced version could identify a motif that spans a subgraph and promote it to lens context).

Error Handling: If logs are empty or no motif found, returns empty list. If duplicates or overlapping motifs, the model should consolidate (e.g. if a longer motif contains a shorter, maybe only keep the

general one unless the shorter appears independently too). It should avoid trivial motifs (length 1 or 2 that are just edges – those are not worth caching separately).

3.5 `IExplainabilityRecorder` – Path & Provenance Logging

Role: Captures the reasoning trace so that the system can explain itself. It records each step's details and can output a structured explanation. Importantly, this is how the `/why` command and UI timeline are driven – by reading from this recorder's log.

- **Methods:**

- `start_thread(thread_id: ID, query: QueryMetadata)`: Initializes logging for a new thread. It might store the query question text, the start/goal, the time, etc. This sets up a fresh buffer for steps.

- `record_step(node_from: Node, node_to: Node, edge: Edge, cost: float, lens: Lens, κ_before: float, κ_after: float)`: Records one hop in the path. It logs the source and target nodes, the edge taken (with its type and ID), the computed cost for that step, which lens is active, and the curvature of that edge before and after update (so Δκ can be derived). It may also record cumulative metrics like total energy E or goal potential G at that step if needed for deep analysis, and any LLM call results if the step involved querying an LLM (though LLM calls are usually outside the traversal loop; if a step is an information injection, it can be noted).

- `end_thread(success: bool, result_node: Node or answer text)`: Marks the end of logging for the thread. If `success`, it attaches the final answer or reached node. It also finalizes any aggregate stats (e.g. total path length, total cost).

- `get_trace(thread_id) -> TraceData`: Returns the structured data for the given thread's execution. This includes the sequence of steps with all recorded fields. The format could be JSON (for UI to consume) or an internal Python object. Each step entry contains node names, edge relation, cost, Δκ, lens, etc. We ensure this format is **stable** (for UI and potential external analysis tools). It basically mirrors the ThreadEvents that were also persisted in the DB, but enriched with human-friendly info (like names, and possibly an English snippet if available for that edge or node).

- `to_text(trace: TraceData) -> str`: (Optional) Generate a human-readable explanation text from the trace, for e.g. a quick summary: "**Path**: A → B → C. Edge A–B (cause) had high weight (κ 1.2) so Manny went to B, then …". However, this might be handled by the UI or an offline Narrator (see Conversational Layer), so the core can keep it simple. Primary output is structured data.

Design & Guarantees: The ExplainabilityRecorder should **faithfully reflect actual computation** (no adding of edges not actually traversed, no reordering). *REQ-008:* The /why replay must exactly match the real path used [32] [33]. To enforce this, the traversal engine only uses this recorder as the ground truth. The `ThreadEvents` persistent log is actually written based on this recorded data (or the recorder itself writes to the DB in addition to memory). This double-logging ensures consistency (the in-memory recorder and DB event log should match).

Performance: Logging each step is O(1) and writes to an in-memory list, negligible overhead. At thread end, a flush to DB (writing the events) happens – typically a small number of rows per query. This is fine.

Determinism: The recorded trace is deterministic for a given run. If a run is repeated with same random seed, the trace should be identical. We use this in acceptance tests to verify reproducibility of reasoning.

Provenance and Meta: We include in each step any external influences. For instance, if an LLM provided a hint that led to taking an edge, that should be noted (e.g. an edge might have a provenance flag "suggested by GPT-4 on 2025-10-01"). The recorder can capture such info if provided (for example, if the PhysicsModel or ingest used an LLM to add an edge, that edge's metadata in the graph includes a source; the recorder can fetch it). This addresses explainability not just of *reasoning* but of *learning*: the user could query "How did Manny learn that fact?" and Manny could trace it back to an ingestion event or story (this is beyond MVP but supported by having provenance fields).

Error Handling: If recording fails (e.g. memory issue or a field is None), it should not break the main loop – at worst skip logging that field. The goal is robust logging without interfering with actual reasoning.

3.6 `IConsolidationJob` – Offline Maintenance

Role: Handles the "sleep" phase tasks: reorganizing and optimizing the manifold after a series of interactions. It ensures long-term coherence, efficiency, and memory management.

- **Method:**
- `run(full_state: GraphState, last_interval: LogSlice) -> ConsolidationReport`: The main entry for consolidation. It runs a series of sub-tasks on the graph (possibly in a specified order or based on config flags) and returns a report of what was done (e.g. "pruned X edges, merged Y nodes, mined Z motifs, rebuilt index"). **Inputs:** it has access to the entire current graph state (nodes, edges, indices) and possibly a slice of logs since the last consolidation (if incremental processing is needed, e.g. only process edges changed recently). **Subtasks include:**
  - **Re-embed Nodes:** If Manny uses dynamic embeddings (for example, if node vectors are meant to update as knowledge changes), perform embedding updates. In v0.1, we assume embeddings are static (set at ingestion by LLM). But if needed, this could retrain or adjust them. More practically, this step might generate new embeddings for newly added nodes using the LLM or a local embedding model and insert them into the ANN index.
  - **Prune Edges:** Remove or weaken edges that are deemed obsolete or harmful. Criteria might include: edges with very low curvature (e.g. κ near 0 and not used recently), or edges identified as spurious (perhaps via validation or negative valence). For each such edge, either delete it from the graph (with an option to archive it in a history table for traceability) or set its curvature to 0 and flag it inactive. Pruning keeps the graph sparse and focused [34] [35]. There should be safety checks (never prune an edge above a certain importance threshold, etc.).

- **Normalize Curvature:** Optionally, adjust curvature values globally if they drift. For example, if overall curvature inflation is noticed (maybe lots of edges near max clamp), we might scale them down or increase clamp. Consolidation can slowly decay everything by a tiny factor to keep values bounded (like a nightly homeostatic reset). The PhysicsModel's built-in decay covers micro-decay, but here we might do a more global normalization.

- **Motif Mining:** Call `IMotifModel.mine_motifs` on recent activity to discover new motifs. Integrate any found motifs: add to motif storage, and also possibly *optimize graph structure* by adding "shortcut" edges or meta-nodes corresponding to the motif. One implementation is: if motif = nodes [A,B,C,D], add a direct edge A→D with special flag as motif shortcut (so next time A to D can be done in one hop at cost lower than going through B,C). Alternatively, store motif separately and let TraversalEngine handle it via `suggest_motif`. The job also might update motif usage stats and remove outdated motifs (calls `prune_motifs`).

- **Rebuild ANN Index:** Reconstruct or update the approximate nearest neighbor index for node embeddings, which accelerates operations like finding similar nodes for new queries or detecting analogies. For MVP, a full rebuild of HNSW index can be done if new nodes were added since last time [17]. We ensure this happens in off hours or infrequently so as not to interfere. The job should output the new index ready for use. If using a library that supports incremental addition, we can instead add just new vectors.

- **Lens Adjustment:** Possibly identify if new lenses should be created or existing ones tuned. For example, if a motif or cluster of new edges is found, create a lens for that cluster (via `ILensModel.create_lens`). Or adjust lens friction if needed globally (not typical, friction ζ is static or slowly adaptive).

- **Memory Compaction:** (If needed) Release any in-memory cache or data structure not needed after consolidation (like free up the old ANN structure after building a new one). Also, optionally dump some logs to disk and truncate them if they've been consolidated into motifs (for instance, after mining motifs from older logs, we might archive those logs).

Determinism: Consolidation tasks ideally produce the same result given the same input state. In practice, some tasks could be nondeterministic (e.g. random tie-breaking in motif mining or if using a stochastic algorithm). For reproducibility, we require either determinism or stable seeding: e.g., motif mining, if it uses any randomized algorithm for frequent pattern mining, must fix a random seed or better, use deterministic algorithms (like sorting by frequencies). *REQ-009:* Consolidation procedures should be as deterministic as possible, to ensure that given the same pre-consolidation state, the post-state is identical (this helps when comparing different runs or debugging).

Scheduling & Budget: Consolidation can be time-consuming. It may not run after every query; typically, it's triggered on a schedule or manual command (e.g. nightly, or when the user clicks "Consolidate" in the UI). We enforce a time budget if it runs automatically: e.g. no more than X seconds or Y% of CPU per consolidation. The risk of heavy consolidation is noted [34] [36]; mitigations include doing partial/incremental jobs. For example, *prune edges* might only consider edges with `updated_at` older than a threshold (not recently used, meaning safe to prune), *mine_motifs* might only consider last N threads rather than entire history, etc. The ConsolidationJob can be configured in profiles (fast vs thorough modes).

Output (ConsolidationReport): After running, it returns stats such as: `edges_pruned`, `motifs_added`, `motifs_removed`, `nodes_merged` (if we ever do node consolidation), `index_rebuilt: True/False (and size)`, `duration` etc. This is logged (and displayed in the Ops UI panel).

Error Handling: If any subtask fails (e.g. ANN library error, or motif mining exception due to data), the ConsolidationJob should catch that and either roll back that subtask or skip it, and continue with others, to avoid one issue preventing all maintenance. It should mark in the report any subtask that failed. Ideally, tasks run in a transaction-like context where changes apply at end; however, many changes are in-memory (like adjusting curvatures or edges) and then persisted. We can use a transaction for DB changes (so if consolidation aborts midway, partial DB changes aren't committed). For example, we might do all operations on an in-memory clone and then apply en masse, but given the moderate size, simpler is fine.

Integration: Consolidation uses other interfaces: `IMotifModel` for motif tasks, `ILensModel` for lens creation, perhaps `IPhysicsModel` if global scaling is needed. It also can produce artifacts for the UI or analysis, such as summary of curvature distribution (for the "gravity field" visualization).

Each contract above (3.1–3.6) specifies **clear inputs/outputs and invariants**. They collectively enforce that even if one swaps out the learning rule or search strategy, the system's overall behavior (in terms of connecting queries to paths and explaining them) remains consistent. For instance, any new `IPhysicsModel` must still interpret higher κ as stronger connection (we lock that design decision, see §12) and must clamp updates to avoid blowing up; any new `ITraversalEngine` must still honor local search constraints (no global BFS that violates locality) and produce a /why trace.

4. Experimentation and Hypothesis Testing Harness

To validate Manny's cognitive physics and enable rapid iteration, we include a first-class **Experimentation Framework**. This harness allows definition of hypotheses (H1–H5, etc.), controlled A/B tests of different "physics" variants, reproducible runs on fixed datasets, and automatic collection of metrics.

Hypothesis Registry: We formalize Manny's core hypotheses (from design docs [37] [38], [39]) into an internal registry with IDs and success criteria. For example:

- **H1 (Path Convergence)**: Repeated queries of the same problem will yield shorter solution paths (≥20% reduction in path length after learning) [40] [10]. *Metric:* path length on first attempt vs nth attempt.
- **H2 (Explainability Alignment)**: The displayed reasoning path corresponds to a logically valid explanation ≥80% of the time [33]. *Metric:* human judge agreement or a check that removing a key edge changes the answer [41].
- **H3 (Motif Reuse / Transfer)**: New queries in related domains reuse at least 30% of subpaths (edges or motifs) from prior tasks [42] [43]. *Metric:* fraction of edges in the new query's path that have been seen before or belong to a saved motif.
- **H4 (Continual Stability)**: Curvature updates remain bounded and do not diverge; the system retains old knowledge alongside new. *Metric:* variance of curvature values stays within range, and a test on old questions still succeeds after new training (no catastrophic forgetting) [10]. Also, edge count growth is limited (e.g. <5% new edges per consolidation for stationary tasks) [44].

- **H5 (LLM Efficiency)**: Manny achieves comparable task performance while reducing LLM usage by ≥50% compared to an agent that uses an LLM for every reasoning step [45] [46]. *Metric:* total LLM tokens or API calls used by Manny vs baseline on same tasks.

These hypotheses (and possibly more like efficiency on neuromorphic hardware, etc.) are stored with formal definitions. The harness can then *falsify* or confirm them by running experiments and checking metrics against thresholds.

Experiment Configuration: The harness allows defining an experiment as a set of scenarios and variants:

- **Scenario:** A sequence of interactions (e.g., a specific domain knowledge to ingest, then a series of queries). For instance, a scenario might be *"Teach Manny a small recipe domain, then ask a series of cooking questions including repeats and analogies."* Each scenario includes the fixed input data (which can be a static file or a generated mini-dataset), and the expected outcomes or metrics to record.
- **Variants:** We can specify multiple configurations of Manny to run on the same scenario. Most importantly, this enables testing different physics or strategies: e.g. *Variant A = default PhysicsModel, Variant B = modified curvature update rule, Variant C = no consolidation* (as an ablation). The harness can instantiate each variant – possibly by injecting a different `IPhysicsModel` implementation or toggling configuration flags (like "turn off motif usage" for an ablation).
- **Metrics:** For each run, the harness collects key metrics: path lengths, number of steps, which edges used, motif reuse counts, LLM calls count, answer correctness, etc. Metrics are computed by comparing Manny's outputs to ground truth or baseline expectations. Some metrics might require external checking (like correctness of answers if the query has a known correct answer).

Execution and Reproducibility: The harness orchestrates running Manny (or multiple Manny instances in isolation) through the scenario. It sets a random seed for each run to ensure consistency. All external randomness (like LLM variations) can be fixed by using the same seed or cached outputs. We also use fixed corpora: for instance, to test analogical transfer, we might have a predefined small knowledge graph of "apple pie recipe" and see if Manny can transfer to "pear pie" – this dataset is fixed and provided to each variant.

Parallel or Sequential Trials: If computing resources allow, the harness can run variants in parallel (e.g. spin up separate process/threads for each Manny variant, each with its own in-memory graph). For now, sequential is fine given a single machine; the harness will reset Manny's state between runs (by reloading a snapshot or re-initializing) to ensure each trial starts from the same baseline. For example, if comparing Variant A vs B, it will load the initial graph snapshot, then apply scenario interactions to A, measure metrics, then reset to the same snapshot, apply interactions to B.

Automation and Dashboard: The outcome of experiments is aggregated into results that the harness can output as JSON or directly into a small web dashboard (accessible in the UI Experiment panel). It will list for each variant the metrics and indicate if the hypothesis criteria were met. For instance, an output for H1 could be: *Variant A: path length went from 5 to 3 (40% improvement, meets ≥20%), Variant B: 5 to 5 (0% improvement, fails H1).* The harness can produce plots (e.g. path length vs trial number) – using external libraries if allowed, or output CSV for analysis.

Example Pre-Registered Hypotheses & Tests:

- *H1 test:* Ingest a small knowledge graph where a question's answer requires a multi-hop path. Ask the same question multiple times. Expect path length reduction. **Pass Criterion:** by 5th repetition, path length $\leq$ 80% of initial ($\geq$20% improvement) [47] [10].
- *H2 test:* Take a set of query-answer pairs where we know a plausible reasoning chain (ground truth). Have Manny answer and produce a /why path. Measure if Manny's path hits the key nodes/edges from the known chain. **Pass Criterion:** $\geq$ 80% of Manny's paths contain the expected key steps, or manual evaluators judge them sensible [5].
- *H3 test:* Train Manny on Task A (e.g. solving apple tart), then Task B (pear tart). Check how many edges in B's solution also appeared in A's. **Pass if:** $\geq$30% overlap [42] [48], and ideally B is solved faster (fewer steps or hints) than A (indicating transfer).
- *H4 test:* Stress Manny with a series of random facts and queries to test stability. After many updates, query an old fact from the beginning. **Pass if:** Manny still recalls it correctly (no forgetting), curvature values remain within normal range (no explosion – e.g. monitor that no edge has $|\kappa| > 1.5$ cap, and average κ variance hasn't grown unbounded) [10]. Also ensure the total number of edges doesn't exceed what we expect given inputs (no runaway edge creation).
- *H5 test:* Solve a set of tasks with Manny and with a baseline LLM-only agent. Compare LLM API tokens used. Manny's log should show far fewer calls after initial learning [45] [46]. **Pass if:** Manny uses $\leq$ 50% of the tokens the baseline did, without significant accuracy loss. Also check Manny's caching: if the same question is asked twice, the second time ideally uses 0 new LLM calls (i.e. fully answered from memory).

The harness supports **A/B/C experiments** by running all variants on identical scenarios. For multi-arm experiments beyond two variants, statistical analysis can be done (not too formal here since we're not doing thousands of runs, but e.g. if some randomness, we might run each variant 5 times and see distribution of metrics).

Integration with UI: The Experiment Panel in the UI (see §6) allows selecting which hypothesis or scenario to test and which variants to include. When user clicks "Run Experiment", the backend harness takes those settings, executes the runs, then returns the collected data. The UI can then display key metrics and indicate success/failure of hypotheses.

Continuous Evaluation: We can automate nightly experiment runs on a standard suite to catch regressions. The harness can be run in a CI pipeline or schedule to ensure that changes to the system still satisfy the core H1–H5 criteria (these serve as acceptance tests, see §11).

Example Experiment Listing: In a config file or code, one might define:

```
{
  "experiments": [
    {
      "id": "E1",
      "description": "Repeat Q convergence test",
      "scenario": "repeat_question",
      "hypothesis": "H1",
      "variants": ["default_physics", "no_decay_ablation"]
    },
    {
      "id": "E2",
```

```
      "description": "Apple→Pear transfer test",
      "scenario": "fruit_tart_transfer",
      "hypothesis": "H3",
      "variants": ["default_physics"]
    },
    ...
  ]
}
```

Where scenarios like `"fruit_tart_transfer"` correspond to a scripted set of interactions (e.g., ingest apple tart steps, ask apple tart question, then ingest pear concept, ask pear question, measure motif reuse and steps). The harness interprets this and runs accordingly.

Metrics Capturing: We use a lightweight internal logger (or external like Weights & Biases as noted as a possibility [12]) to record metrics. For now, printing results to a JSON or database table is enough. Each run might produce an entry in an `experiment_result` table (with exp ID, variant, metric values, timestamp).

Dashboard Example: A dashboard page could show a table:

Experiment	Variant	H1 (20% conv)	H2 (expl ≥80%)	H3 (reuse ≥30%)	H4 (stable)	H5 (LLM -50%)
Repeat Q	default_physics	30%	–	–		–
Repeat Q	no_decay_ablation	10%	–	–	drift	–
Transfer	default_physics	–	–	45%		–
LLM Usage	default_physics	–	–	–	–	-60%

This indicates variant without decay failed H1 (only 10% improvement) and had stability issues, etc.

In summary, the harness enforces **scientific rigor** in Manny's development: each new physics idea can be A/B tested against the baseline on the same scenarios to see if metrics improve. Hypotheses are falsifiable – if Manny fails the predefined criteria, the experiment flags it, guiding developers to adjust. This is critical given Manny's ambitious goals (the feasibility dossier explicitly calls for falsification trials early [49] [50]).

5. LLM Acceleration Layer

Manny uses Large Language Models as **assistive tools** – as lenses into data and as accelerators for tasks like knowledge ingestion and summarization – **not as the primary controller** of reasoning [45] [51]. This layer specifies how to incorporate LLMs to enhance Manny's learning speed and knowledge breadth while obeying constraints of locality, determinism, and cost.

LLM Usage Roles: LLMs are integrated in limited, well-defined ways:

- **Ingestion & Extraction:** When new unstructured information is added (e.g. user provides a text or a web article), an LLM is used to extract structured knowledge (nodes and relations) for Manny's graph [52] [53]. For example, a prompt: *"Extract key facts from this text into triples (subject-*

relation-object)". The LLM's output is parsed and then integrated as new nodes/edges (with initial κ values via `PhysicsModel.init_curvature`). This way, the LLM acts as a parser/labeler, not directly deciding how to connect to existing knowledge – Manny's consolidation can later adjust those edges if needed.

- **Embedding Generation:** For any new concept or text, an LLM or associated model (like a transformer embedder) produces an embedding vector. Manny uses these for similarity search and lens contexts. We treat this as an LLM function because often a pre-trained language model (like SentenceTransformer) provides embeddings. This is deterministic given input, except for minor nondeterminism which we avoid by using fixed models.

- **Conversational Vectors (Primers):** Instead of static definitions, Manny uses an approach of generating *context-rich primers* for new concepts [54] [55]. When Manny learns a concept, it may prompt an LLM to produce a few sentences explaining that concept in context (based on the "Conversational Vectors" proposal). These richer descriptions yield more interconnected edges (the text itself might mention related concepts, which become edges in the graph) [56]. This improves the manifold's initial curvature distribution (richer semantic context yields smoother, more accurate connections [57]). The LLM acceleration layer handles this by constructing clever prompts. E.g., instead of "Define fermentation," Manny asks "Write 2-3 sentences about fermentation in the context of cooking – how it's used, why it matters" [55]. The result ties fermentation to cooking concepts (e.g. kimchi, miso) giving Manny immediate edges linking those nodes, which leads to better lens formation for the domain. These prompts modes (A: domain+topic, B: topic only, C: domain-first) are configured and selected based on context [58] [59].

- **Answer Framing:** After traversing the graph to get an answer (e.g. the path ends at a node that likely is the answer), Manny can use an LLM to generate a nicely phrased response for the user. For instance, Manny knows Alice is Charlie's aunt (by finding path Alice–Bob–Charlie), it can call an LLM with a prompt like "Given that path, explain in English how Alice is related to Charlie." [60]. The LLM essentially acts as a *natural language generator* with the graph output as context. This doesn't affect Manny's reasoning, just presentation.

- **Summarizing and Narration:** In offline mode, Manny might use LLMs to produce summaries of what it learned (the Reflective Narrator concept) [61] [62]. This includes turning daily changes (Δκ, motifs) into a narrative for developers or users. Because this is offline and read-only (no effect on κ), it falls within allowed use. Similarly, Manny might occasionally ask an LLM for an analogy or suggestion if stuck (like a *gentle hint provider*), but by design, the LLM doesn't directly write to Manny's knowledge in real-time – it might propose something which Manny then *confirms via its own reasoning* or holds for consolidation (story capture below).

- **Story Capture (Optional Elicitor):** Users can engage in a meta-conversation where they provide stories or analogies, which an LLM helps structure into "StoryPacks" (structured data) [63] [64]. These story packs contain potential new edges or motifs drawn from the story, but *they are not immediately integrated into the live graph* [65] [66]. Instead, they are queued for offline review. This allows Manny to gather richer knowledge (including user's own metaphors or anecdotes) without polluting the online phase with unchecked data. In consolidation, the system can then formally incorporate those if they pass validation. This implements *offline hot-swap narrators/ storypacks*: the LLM helps transform free-form input into candidate knowledge that is handled offline, upholding the **no-hot-path-write** rule from the prompt.

LLM Layer Constraints:

- **Deterministic Model Selection:** When multiple LLM models are available (for cost/performance tradeoff), the system deterministically picks which model to use for a given task *based on the input*. We implement a **hash-based selection** so that the same prompt or topic always goes to the same model configuration [67] [68]. For example, we might use a small model for most embeddings, but a larger model for complex domain primers. A configured ratio (e.g. 70% of topics to cheap model, 30% to medium, 10% to expensive) is applied via hashing so it's repeatable [69]. This ensures that caching is effective – the same request won't unpredictably hit different models. **REQ-010:** All nondeterministic aspects of LLM usage (like which model or prompt style) must be made deterministic or rule-based to maintain consistency and debuggability.

- **Caching Strategy:** All outputs from LLM calls are cached to avoid repeated costs and variability. For instance, embeddings for a given text are stored; primers generated for a concept are stored; answers phrased for a path are stored (perhaps keyed by a hash of the path). The cache keys include the model and prompt details to avoid mixing outputs of different methods [70]. Manny maintains a local cache (in-memory or on-disk) with a map from input->(model->output). *Example:* if it asked GPT-4 to extract triples from "Alice is Bob's sister...", that exact call result is saved so if it sees identical text again it won't call the API. We also log each call to a CSV or table with timestamp, model, cost, etc [18] [19] for transparency and budgeting.

- **Budget Enforcement:** Manny has a configured budget for LLM usage (e.g. max N tokens per session, or cost $ per day). The LLM layer monitors usage. If a request would exceed budget, it either denies it or uses a cheaper alternative (like skip a verbose primer, or use the smallest model even if quality drops). Manny's design hypothesis is that the graph should handle most heavy lifting, so **LLM calls should be infrequent** [71] [72]. For instance, H5 target was <50% calls vs baseline; in practice we aim LLM token usage to be under 5% of total compute cost [73]. The system can have a runtime setting like `max_tokens_per_query` and `max_tokens_per_day`. If an interactive user session triggers too many LLM calls (maybe they keep ingesting large text), the system can enter a "LLM cooldown" mode, warning or slowing down.

- **LLM as Lens, not Actor:** Manny never blindly executes an LLM's multi-step plan. LLM outputs are treated as suggestions and data. For example, if an LLM suggests a relation that conflicts with Manny's knowledge, Manny can hold it for confirmation or integrate it with low curvature (so it won't dominate unless reinforced). We also ensure the LLM is *not in the loop of each reasoning step*. It might generate initial knowledge or final answers, but during the traversal, Manny doesn't call the LLM to ask "what next?" That ensures the reasoning chain is Manny's own (which is key to explainability and stability) [46] [74].

- **Provenance Logging:** Each piece of knowledge from an LLM is tagged with its source. For example, an edge added via GPT extraction will carry metadata: `source=GPT-4 via ingest on 2025-12-01`. Manny can surface this if needed (especially for debugging or user trust, "Where did it learn that?"). Also, if an answer is phrased by an LLM, the system can note that in logs (though the content is ultimately derived from Manny's graph, we note the final phrasing was LLM-generated). This allows traceability and helps ensure if any inappropriate content arises, we know the source.

- **Model Rotation & Ensemble:** As per the Conversational Vectors spec, Manny may employ an ensemble of models to generate training content [69] [68]. For instance, 70% of the time use a

smaller fine-tuned model for speed, occasionally use the big model for high quality (ensuring some diversity). This is seamlessly integrated via deterministic selection. The expected effect is a good balance of relation accuracy and diversity [75]. The rotation is also logged with counts of usage. If one model is removed or changed, the selection mechanism (hash-based on topic+domain) means previously generated content remains valid and cached per model, and new requests for that topic would consistently go to the assigned model (ensuring *persistency* – e.g. always use GPT-4 for "quantum physics" domain primers to maintain style consistency).

- **Cost Controls:** The config allows specifying which model to use for which tasks and how often, as above. If cost needs to be cut, one can adjust ratios to favor cheaper models (the spec even suggests an alternate 70% nano, 30% mini, 10% large for ultra-low cost) [69]. Manny's runtime can adapt if budget is tight by automatically switching to cheaper modes or skipping non-critical LLM tasks. For example, if approaching token budget, Manny might stop generating nice answer phrasing and respond with a direct fact from the graph (less polished but no cost). Also, any multi-turn conversation beyond Manny's memory triggers retrieval from its own graph rather than extending LLM context (to avoid prompt inflation).

Testing the LLM Layer: We have specific acceptance checks: ensure that if the same content is ingested twice, the LLM is only called once (cache hit second time). Ensure that the same topic always yields the same primer text (or one of a fixed set if we allow slight variation with ensemble, but deterministically chosen). Also, measure that Manny's performance doesn't degrade significantly when limiting LLM calls – i.e., after initial learning, it can operate with no new LLM input on follow-ups (this was part of H5: degrade gracefully when LLM not available for a while) [76]. The harness can simulate a "LLM outage" and check Manny still answers known questions from its graph (expected outcome: Manny might not handle totally novel queries without LLM, but anything it's seen or related analogies it should handle).

Security & Filtering: The LLM layer should include content filtering on any user-generated prompt we send to the LLM (to adhere to safety: e.g. avoid sending sensitive personal data by accident, or filter out if a user tries to use Manny to generate disallowed content through the backdoor). Additionally, outputs from LLM (like edges) should be validated – if the LLM says something clearly incorrect or harmful, the ingest pipeline can drop it or mark it with low confidence requiring user confirmation. Manny's use of LLM is **bounded and observable**, reducing risk of unpredictable behavior [77] [78].

In summary, the LLM Acceleration Layer gives Manny the benefits of language models (knowledge, semantic understanding, fluent language) *without ceding control*. Manny's core remains a deterministic graph learner; the LLM is a peripheral service (like I/O) used sparingly. This satisfies the project's aim to *use LLM as a lens, not a brain* [71] – Manny stores and reasons over knowledge in a way that persists and improves with use, something an LLM alone cannot do (with its fixed weights and context limits) [79] [80].

6. UI Requirements (HTML Interface)

The Manny Manifolds UI is designed as a web application (either single-page app with dynamic updates or a server-rendered app with AJAX, depending on ease) that allows users to **visualize the knowledge manifold, query it, watch it learn, and control experiments and maintenance**. This section details the UI's functionalities, routes, and the backend endpoints serving them.

General Design: The UI provides multiple views/tabs: 1. **Graph Map View** – an interactive visualization of the manifold. 2. **Why/Trace View** – a timeline of reasoning steps for the last query or selected session. 3. **Experiment Panel** – controls to run experiments and display metrics. 4. **Operations Panel** –

administrative controls (consolidation, snapshots, etc.). 5. (Optional) **Narratives/Stories Panel** – view offline narrative summaries or user story submissions (for advanced phases).

It's implemented as an HTML5/CSS interface with JavaScript for interactivity (likely using a visualization library for graphs, e.g. D3 or Cytoscape for network display). It communicates with Manny's backend via RESTful endpoints (or WebSocket for real-time updates).

Route and Component Summary:

UI Route/Page	Key Components	Backend Endpoints (API)
`/map` (Graph View)	- **Graph Canvas**: visual node-edge map (pan/zoom) - **Node Detail Panel**: shows selected node info (label, links, κ values, provenance) - **Overlay Toggles**: checkboxes to overlay lens heatmap, gravity field, etc. - **Query Bar**: input for user questions or commands.	`GET /api/graph?view=lensX` (fetch subset of graph, possibly filtered by lens or region) `GET /api/node/{id}` (details for a node) `POST /api/query` (submit a question or command; returns answer or acknowledges thread start). `WS /api/live_updates` (websocket for streaming updates: e.g. highlight path while it's found, or indicate new nodes added).
`/why` (Trace View)	- **Timeline/Step List**: each step of reasoning as an item with node names and relation, cost, Δκ. - **Graph Highlight**: highlights the path on the graph (could reuse Graph Canvas highlighting those nodes/edges). - **Provenance Tooltips**: hover on an edge shows source (e.g. "learned via motif from X" or "from text Y").	`GET /api/thread/{id}/trace` (fetch the recorded trace of a thread in structured form) `GET /api/thread/{id}/explain_text` (maybe a pre-formatted text explanation)
`/experiment` (Experiments Panel)	- **Experiment Config Form**: select hypothesis or scenario, select variants (checkboxes or dropdown for each plugin/setting). - **Run Button**. - **Results Table/Chart**: after run, shows metrics per variant and pass/fail vs criteria (could be green/red indicators). - Possibly **History**: list of past experiment runs.	`GET /api/experiments` (list available scenarios/hypotheses and variant options) `POST /api/experiment/run` (with chosen scenario & variants; triggers harness) `GET /api/experiment/result/{run_id}` (poll for results or get when done)

UI Route/Page	Key Components	Backend Endpoints (API)
`/ops` (Operations Panel)	- **Status Overview**: e.g. number of nodes, edges, last consolidation time, current memory use. - **Controls**: buttons like "Run Consolidation Now", "Export Snapshot", "Compare Snapshots", "Toggle Read-Only Mode". - **Snapshots List**: select two snapshots and hit compare, or load one. - **Import/Export**: upload a small knowledge file or export current graph as JSON.	`POST /api/consolidate` (trigger offline jobs now) `POST /api/snapshot?tag=name` (save snapshot) `GET /api/snapshots` (list snapshots) `POST /api/compare_snapshots` (get diff stats between two snapshots) `POST /api/set_mode` (e.g. read_only=true/false, apply_without_commit toggles for safe mode) `POST /api/import` (upload knowledge) `GET /api/export` (download entire graph or data dump)
(optional) `/stories` (Narratives Panel)	- **Narrative List**: list of daily narrative summaries (with link to full text) for transparency. - **Story Upload**: a textarea for user to submit a story or metaphor, which Manny will capture offline. - **Story Status**: shows if a story pack is awaiting consolidation.	`GET /api/narratives` (list stored narrative files or entries) `GET /api/narrative/{id}` (fetch a narrative text/JSON) `POST /api/story` (submit user story text; triggers LLM elicitor to process into storypack JSON, stored but not integrated)

Real-Time Visualization: The Graph Canvas should update as Manny learns. For example, if a query just ran and some edges got their curvature increased, the UI can highlight those edges (maybe flash or bold them). Manny's backend can push these updates via WebSocket or the client can poll. We maintain a **/api/live_updates** channel where the backend sends events like: `{"type":"edge_update", "edge_id":123, "new_k":0.8}` or `{"type":"new_node", "node": {id:..., label:...}}`. The UI then reflects this (e.g. thickening that edge's line or recoloring it to indicate higher curvature). This satisfies the requirement that users can see the "gravity field" or curvature changes in real time. A 2D visualization might use color or curvature of lines to show κ (e.g. warmer color = higher κ). Manny's docs envision even a 3D VR view [81], but for MVP a 2D map with pan/zoom is fine [82].

The **lens overlay** toggle, when on, could color nodes by lens affinity. For example, if "Cooking" lens is selected, the UI calls `/api/graph?view=lens_cooking` and gets either a subgraph or a list of node affinity scores. It then maybe dims nodes not in lens and highlights ones strongly in lens. Similarly, a "gravity field" overlay might illustrate which region has high valence or dense knowledge (perhaps via a heatmap or contour lines behind the graph, if feasible, or simply a summary that certain clusters are heavily connected).

Why Playback: When user clicks a past query from history or after asking "why?", the UI switches to the `/why` route or opens a panel showing the step-by-step trace. Each step lists something like: "Step 1: **Node A** –*(relation X)*→ **Node B** (cost 1.2, κ 0.5)" possibly with a note if lens switched or motif used ("used motif *baking_steps* here"). The UI can allow stepping through (like a slideshow or timeline slider). As the user moves through steps, on the graph view those nodes/edges could be highlighted (the UI has access to both the graph and the trace, so it can link them). This addresses the explainability UI requirement: *"/why playback timeline (step-by-step traversal with costs, κ, E(x), lens tags)"* – we include costs and lens tags as recorded [62]. The provenance might be shown as an info icon that, when hovered,

shows e.g. “Edge learned from 3 examples, motif #5 applied” etc., pulled from the trace data or directly from edge metadata.

Experiment Panel UI: This section is more form-based. It lists available hypotheses or scenarios by name (populated from backend metadata). The user picks one or multiple. If multiple variants are possible, the UI might show checkboxes for each instrumentable option: e.g. *Use baseline physics? Use alt decay? Use motif off?* – or simply list predefined named variants. The user clicks run and then sees a loading indicator. Because experiments might take some time (e.g. running multiple scenarios or heavy tasks), we can run them asynchronously and notify via WebSocket or have the client poll `/api/experiment/result` until done. The results would be shown in a table or chart as described (the UI can use red/green highlights, etc., plus possibly bar charts for actual values).

Operations/Safety Controls: The **read-only mode toggle** is critical for safety in demos or analysis. In read-only mode, Manny will not modify κ or add edges even if queries come (it will only retrieve answers). The UI should display a status (like a lock icon when read-only). The toggle triggers `POST /api/set_mode?read_only=true/false`. In read-only, the backend simply skips calls to update_curvature, etc., or sets η=0 effectively. This is useful when one wants to explore the current state without contaminating it with new learning.

Apply/Dry-run toggles: For certain inputs, the UI might allow a “dry run” where Manny goes through the motions but does not actually commit changes. For example, an admin might simulate consolidation (to see what it *would* prune or change) without applying it. Or run an experiment in a sandbox copy of the graph. Implementation-wise, this could be done by forking the state or by having a global flag that prevents writes. The UI can expose it as a checkbox “Dry Run” on operations. If checked, the backend will perform the operation on an in-memory clone or simply roll back at the end.

Gated Mutations: The UI should confirm potentially destructive actions. E.g., if user tries to delete a node or clear the graph, it should prompt “Are you sure?”. Also, if Manny suggests adding an edge from a story pack that is sensitive, we might show it in the UI for user approval (not in MVP, but a future idea).

UI – Backend Endpoint Details: All API calls should be stateless (the backend keeps session state like current graph in memory, but HTTP calls manipulate it). Authentication might be minimal (maybe a simple token if needed in multi-user environment, but likely out of scope as this is a local research tool initially).

Example Endpoint Behaviors:

- `POST /api/query`: Takes input text. The backend will parse it (maybe check if it’s a special command like “/learn X” vs a normal question). Then it triggers the thread runner. This call can either be long-polling (wait until answer ready and return it along with thread ID and maybe short explanation) or respond immediately with a thread_id and then updates come via events. Possibly simpler: do synchronous for now (if typical query time is short, a few seconds). The response JSON might contain `{"answer": "...", "thread_id": 42}`. The UI displays the answer and a “Why?” button which when clicked uses thread_id to fetch the trace.

- `GET /api/graph`: For initial load, fetch nodes and edges. If graph is big, we might only fetch summary or a particular view. For MVP, if graph ~50k nodes, might need to filter. We could fetch only the subgraph of interest (like active region or a sample). The lens parameter allows fetching only nodes/edges relevant to a lens context or of high curvature etc., so that the front-end isn’t overloaded. Alternatively, implement progressive loading (load most connected components

first, etc.). The UI can show an overview and allow selecting region or search a node by name (with an API like `/api/search?query=term` to find node IDs by label). On selection, the detail panel calls `/api/node/{id}` for full info.

- `POST /api/consolidate`: User triggers consolidation; the backend runs the ConsolidationJob (maybe in a separate thread so UI can remain responsive). We could immediately return 202 Accepted, and then send a WebSocket event when done with the report. Or block until done if consolidation is quick. The UI will likely show a spinner "Consolidating..." then update some status (like "Pruned 5 edges, added 1 motif").

- `POST /api/compare_snapshots`: The backend loads two snapshot states (from disk or DB) and computes differences (counts of nodes, edges, maybe a diff list). It then returns summary JSON (like `{nodes_added: X, nodes_removed: Y, edges_added: Z, edges_removed: W, changed_edges: [...]}` maybe listing edges whose curvature changed significantly). The UI could then highlight those differences or simply report counts.

Safety in UI: The UI needs to expose **safe-mode controls** to the user for governance. These include the mentioned read-only toggle and requiring confirmation for external actions (like ingesting internet data if that was allowed, which it might not be in MVP). If multi-user or public, we might have a mode where UI is view-only (no adding knowledge). But likely not needed now.

Responsive Design & Complexity: The graph visualization is perhaps the most complex UI element. For MVP, given moderate graph size, we can use a force-directed layout or a fixed layout (maybe cluster by lens). The user can drag nodes, search and center on a node, etc. The detail panel ensures not everything has to be visible in the graph at once (which could be cluttered). Filtering by lens drastically reduces visible graph to relevant subgraph (since Manny's lenses cluster concepts [83]). We might also provide a slider to filter edges by curvature (e.g. only show edges above a threshold, i.e. the strong connections, to avoid hairball of weak edges).

UI Performance: 10k nodes is on edge of what can be drawn in DOM/SVG; using WebGL or canvas might be needed. But we can optimize by only rendering part. The UI can by default show the immediate neighborhood of recently involved nodes (like a context around last query). Possibly, an approach: when a question is answered, highlight that path and local area, rather than showing the whole graph by default.

Visual Elements: Represent nodes as circles or labels (maybe icons if specific types), edges as lines with thickness representing curvature (or color intensity). Possibly show a "gravity well" effect: nodes with very high total valence could be drawn larger or glow, indicating they are knowledge hubs (the doc mentions "planetarium" metaphor where curvature could be visualized as a distortion or gravity field pulling threads) [81]. For now: maybe simpler, e.g. a slider to exaggerate edge thickness by curvature to see which connections are strong.

User Interactions: Clicking a node could fix it as center and list its edges; clicking an edge might show its details (relation type, κ value, last updated). Hovering lens names might highlight lens-specific subgraph (like if we have domain tags, highlight those nodes).

Routes and security: Possibly one route, e.g. a single-page app under `/` with internal tabs. But delineating as above is fine conceptually.

Backend Implementation Note: The `IUIBackendAPI` interface (see §9) essentially covers the endpoints above. This API layer uses the core engine's interfaces to perform actions. For instance, `POST /api/query` will call `TraversalEngine.execute_thread` and then `ExplainabilityRecorder` etc. `GET /api/node` will query the graph storage or in-memory data.

All UI features align with the specification's requirements: - Real-time view of graph and overlays (satisfied via Graph Canvas with lens overlays and live updates). - `/why` playback timeline (Trace view). - Experiment panel (with hypothesis selection and metrics). - Operations panel (consolidation, snapshots, safe-mode). - Safety controls (read-only toggle, etc., clearly provided in Ops panel). Thus, Manny's UI will be a comprehensive control center allowing developers and users to see **data as space, conversation as motion, learning as curvature** in action [84] [85].

7. Hot-swapping and Extensibility

A core goal of Manny v0.1 is to support rapid experimentation with different "physics" variants and new features without breaking the system. We achieve this via a **plugin architecture** and a **strategy registry** for hot-swappable components. Key extension points include: energy/cost function, curvature update rules, traversal strategy, motif mining algorithm, lens projection scheme, and consolidation logic. Each corresponds to the interfaces defined in §3. Here we describe how to package and integrate new variants, and how experiments or runtime selection is done, ensuring results remain comparable across variants.

Plugin Interface & Registry: Manny's codebase will have an extensibility module that maintains a registry (lookup table) of available implementations for each interface. For example, for `IPhysicsModel`, there might be entries like: - `"default_physics"` -> DefaultPhysicsModel (the one described above, using distance–ακ + novelty, etc.). - `"no_decay_physics"` -> NoDecayPhysicsModel (a variant where decay is turned off, for experiment). - `"alt_update_physics"` -> AltPhysicsModel (maybe uses a different Δκ rule or different cost formula).

Similarly for traversal: `"greedy_traversal"` vs `"beam_traversal"`, for motif: `"simple_count_motif"` vs `"graphlet_motif"`, etc.

New plugins can be **packaged as Python modules or classes** that implement the required interface abstract class. To hot-swap, one can either specify in a config or command (e.g. via experiment config or even at runtime via UI) which implementation to use. The registry could be populated from entry points or by scanning a `plugins/` directory for classes that subclass the core interface classes.

Registration Example: A plugin developer writes a new file `my_physics.py`:

```
class MyAltPhysics(IPhysicsModel):
    def compute_cost(...):
        # custom cost function
    def update_curvature(...):
        # custom learning rule
    # ... other methods
# Register it:
PluginRegistry.register("physics", "alt_v2", MyAltPhysics)
```

At startup, Manny loads known plugins (perhaps defined in a config JSON or discovered). The `PluginRegistry` is essentially a dictionary: `registry["physics"]["alt_v2"] = MyAltPhysics`. Similarly for other categories.

Selecting a Variant: There are multiple ways: - **At Launch:** Manny can be started with flags (e.g. `--physics=alt_v2 --traversal=beam`) to globally use those implementations. - **Per Experiment:** The Experiment Harness can temporarily override the defaults by instantiating, say, an alternate PhysicsModel for that run while leaving the rest same. - **At Runtime (hot-swap on the fly):** Potentially via UI dev mode, one could switch the strategy mid-run. However, switching physics in the middle of an active knowledge base might be tricky (since the interpretation of κ might differ). It's safer to switch between runs or on reset. But some components like traversal algorithm can change on the fly (the pathfinding method doesn't change the stored data, just how it's used). For example, the UI might have a dropdown "Traversal strategy: Greedy / Beam / Stochastic" that calls an endpoint to set the TraversalEngine implementation for subsequent queries. We need to ensure changing it doesn't break explainability: (it shouldn't, since any traversal still logs steps, just might pick different path).

- **Bound to Lenses or Context:** Possibly advanced: maybe different physics for different contexts. Likely not in v0.1, but conceive e.g. a "creative mode physics" vs "precise mode physics" that could be lens-specific. However, that complicates comparability. Better keep one physics active at a time.

Packaging a New Variant: A variant could be as small as a configuration change (like turning a knob) or as large as a new module. For simple changes, we support config-based variants. E.g., "no_motif" variant might not require new code, just a flag to disable motif usage. We can treat that as a variant in experiments by running Manny with `motif_enabled=False`. The Experiment harness should allow setting such flags as part of variant definition. For bigger changes (like a radically different motif mining algorithm), the developer writes a new `IMotifModel` implementation and registers it.

Compatibility of Results: To compare variants, we ensure that *core surfaces remain same.* That is, regardless of physics model, the meaning of metrics like "path length" or "edge reuse" stays consistent. For instance, even if one physics variant uses negative κ to indicate repulsion and another uses only positive κ, we define metrics in terms of outcomes (path length, etc.) so we can compare. We explicitly lock the **sign convention of κ** (positive = attract, negative = repel) as a design decision (see §12) to avoid confusion – any physics model must treat higher κ as stronger connection, even if their internal algorithm is different. This way, a motif reuse count or curvature distribution has a comparable meaning across variants.

Ensuring Stability of Interfaces: When introducing a new variant, if it doesn't conform to interface (say someone forgets to implement clamp in update rule), tests should catch that (like an acceptance test expecting no κ beyond ±1.5 should fail). We thus strongly encourage new PhysicsModel variants to reuse the base class which already implements common safety features (like it might have a wrapper that automatically clamps, so the custom model calls super.update_curvature maybe).

Plugin Safety: We may allow untrusted plugins (if open ecosystem), but that's beyond MVP. For now, assume internal development of variants, so no special sandbox.

Experiment Binding: The experiment JSON or UI selection connects to these variant keys. For example, an experiment entry might say `variants: ["default_physics", "alt_v2_physics"]` meaning run once with registry setting physics=default, once with physics=alt_v2 (the harness will internally switch out the `IPhysicsModel` instance used when constructing Manny's components). Other aspects (like using/disabling consolidation in one variant) can be handled by configuration toggles.

Maintaining Comparability: Results remain comparable by using identical initial conditions and datasets. For fairness: - We always **reset to the same snapshot** or initial graph for each variant run. - Use the same sequence of queries (for a scenario) – the harness replays the exact same inputs. - Use same random seeds where applicable (like if traversal had random). Thus any difference in outcome is attributable to the variant itself.

Also, we ensure **metrics are measured in the same way**. The harness code that computes metrics (like path length, number of edges reused) is independent of the variant; it just looks at the output trace. So even if one variant took a slightly different path, it yields a length we count the same way.

Hot-Swap Without Recompile: As much as possible, changes should be loadable at runtime. If adding a new lens algorithm, one should ideally not need to restart the whole system – just register and use. But some changes (like a new PhysicsModel) might require re-running experiments to see effect, which is fine.

Future Extensibility: The spec covers known extension points, but Manny is expected to integrate new modalities or modules (e.g. integrate a vision module or new drive). The architecture encourages adding new tables for new data (via migrations) and new interfaces for new subsystems (keeping them loosely coupled). For instance, if adding a "DriveModel" (for the motivational drives concept), it would be yet another plugin type that could be swapped or tuned, but since drives affect energy function (as G(x), U(x) fields), those can be integrated via the PhysicsModel interface (which already has goal potential G and uncertainty U in cost formula). So we likely wouldn't need a separate plugin, just extended config for PhysicsModel to account for drives (or a subclass implementing drives). This shows how the design anticipates growth: new features either map to existing interfaces or add parallel ones.

Deployment of New Variants: We could envision a *plugin directory* where dropping in a file adds the feature. Possibly we will implement a dynamic loader that reads a config file listing plugins to load (with paths). This avoids altering core code for each experiment, making turning features on/off easy.

Hot-swapping Consolidation Algorithms: For example, one might try a different motif mining technique (e.g. graph clustering vs frequency count). We can encapsulate those in IMotifModel implementations, or even within IConsolidationJob – e.g., the ConsolidationJob might call a certain routine which can be swapped. If it's simpler, we might treat Consolidation strategies also as plugins: e.g. `"fast_consolidation"` vs `"thorough_consolidation"`. Or more granular: one could toggle "use streaming motif update vs full recompute" with a flag.

Compare Results across Variants: When showing experiment outcomes, ensure clarity: if one variant has a completely different approach, some metrics might differ in meaning? We tried to avoid that. Possibly, if a variant drastically changes something (like if one variant doesn't use curvature at all but something else), then metrics like "curvature variance" don't apply directly. In that case, the experiment definition might skip certain metrics for that variant or mark them N/A. But for our current variants (which likely all involve curvature, just differently), metrics are fine.

In summary, to add a new "physics": 1. Implement class fulfilling `IPhysicsModel` (or other relevant interface). 2. Register it with a unique key. 3. Either set it as default in config or specify via experiment/UI to use it. 4. Run experiments to measure difference.

Case Study – Hot Swap Example: Suppose we want to test a "stochastic traversal" where instead of greedily following minimum cost, Manny sometimes explores higher cost paths for diversity (with a softmax temperature τ). We create `StochasticTraversalEngine` implementing

`ITraversalEngine.execute_thread` similarly but using probabilities to choose next step. We add it to registry as `"traversal_stochastic"`. We then run scenario where an answer might be hidden behind a non-greedy move. On default (greedy) Manny may miss it; on stochastic Manny might find it. The experiment harness logs success rate etc. The results show if stochastic gave better coverage at cost of longer path maybe. We can then decide to incorporate such a strategy for certain domains or keep it as an option.

Maintaining Core Surfaces Stability: The plugin approach means we do not change the interface definitions themselves when trying new things – ensuring any improvements stay modular. E.g. if one day we discover we need an extra parameter in cost function, we might add it behind the scenes but the interface `compute_cost(u,v,edge,lens)` remains the same signature (maybe the PhysicsModel internally uses more global variables but interface unchanged). This is part of *future-proofing*: we try to foresee which surfaces should remain constant.

Distribution: If Manny becomes an open project, researchers could publish their variant modules and others can drop them in to test. Since we freeze interface names (like `IPhysicsModel` etc.), compatibility is maintained.

8. Robustness & Future-proofing

Finally, we address configuration management, observability, guardrails, and testing to ensure Manny is robust in long-term operation and adaptable to future changes.

8.1 Configuration Strategy: Manny will support multiple runtime profiles (development, experimental, production). Configurations (in a JSON or YAML file) will control parameters: learning rates, clamp values, active plugins, logging verbosity, etc. For example, a `runtime.json` might have:

```
{
  "physics_model": "default_physics",
  "traversal_engine": "greedy",
  "learning_rate": 0.05,
  "curvature_clamp": 1.5,
  "turn_kappa_budget": 0.06,
  "valence_decay": 0.001,
  "read_only": false,
  ...
}
```

Switching profile (e.g. to “experiment mode”) might set more verbose logging and determinism on, whereas “demo mode” might allow some nondeterminism for liveliness but with safe caps.

Schema Versioning: As discussed in §2, the database schema has a version and migration path. The codebase will include migration scripts for each version bump (to add tables/columns, etc.). All serialized artifacts (like snapshots) should ideally carry a version as well or be backward-compatible (e.g. an old snapshot missing a new field can be loaded with defaults). We'll maintain backward compatibility at least for one major version to allow smooth upgrades.

Capsule Experiment Bundles: To ensure total reproducibility of an experiment, we will support bundling the exact state and config. A “capsule” might be a zip file containing: the initial snapshot of the

graph used, the config file (with random seeds and variant settings), and the log of interactions (the script of queries). This capsule can be loaded on another instance of Manny or after changes to verify results remain consistent or to debug differences. We might implement an export function that creates such a bundle, and an import that sets up Manny in that state and runs the interactions, then one can compare outcomes. This is especially useful for publishing results or for rolling back if an update inadvertently changes something: you can re-run the old bundle to confirm everything still works.

8.2 Observability:

- **Structured Logging:** All subsystems output logs in structured format (JSON or key=value lines) for important events. For instance: when a thread ends, log `INFO {"event":"thread_end","id":42,"path_length":5,"nodes":["A","B","C"]}`; when consolidation prunes edges, log details of which edges. These logs feed into either a file or a dashboard. We also include **trace IDs**: every thread gets an ID, and any log related to it carries that (so we can filter all events of a given query easily). Similarly, an experiment run has an ID that is included in logs of threads under it. This will help debugging and correlation of events.

- **Metrics Collection:** We will continuously measure certain metrics during normal operation (not just in experiments). E.g.: number of LLM calls made today, current memory usage of graph, average path length of last 10 queries, distribution of curvature values. These can be exposed via a metrics endpoint or visualized in UI (like a small stats overlay). For development, integrating with tools like Weights & Biases as suggested [11] can track these metrics run-over-run.

- **Tracing and Profiling:** For performance, we might implement lightweight profiling (timing of key sections: traversal time, consolidation time, etc.). If a query is slow, having trace of which step took long helps (maybe an edge had too many neighbors expanding, etc.). We can include a toggle for debug mode that logs each neighbor expansion and cost (though verbose).

- **Replay & Debug Mode:** The system can store replay bundles of interactions (like the "capsule" idea but even for last session in memory). A dev can take a recorded trace and run it step by step with more debug info if needed. Possibly integrate with Jupyter or a CLI for introspection (e.g. a command to print out the current adjacency of a node).

8.3 Guardrails and Invariants:

- **Safe Defaults:** All parameters have conservative defaults to ensure stability (e.g. by default plasticity is low, requiring deliberate increase if more learning aggressiveness needed). Also, certain features are off by default until proven (like any exotic consolidation trick).
- **Invariants Enforcement:** We will code explicit checks for critical invariants. For instance, after each curvature update, assert that $|\kappa| \leq$ clamp (and if not, correct it and warn) – this prevents unnoticed runaway values. Another invariant: the graph should remain connected enough to reach important nodes; if an entire region becomes isolated due to pruning, maybe log a warning.
- **Circuit Breakers:** If Manny's metrics indicate unhealthy behavior, automatically intervene. E.g., if curvature variance spikes beyond a threshold (suggesting possible divergence), the system could reduce learning rate or enter a safe mode where it stops updating (until maybe consolidation fixes it) [86] [87]. Similarly, if LLM usage unexpectedly spikes (maybe a bug causing repeated calls), we halt further LLM calls (like a fuse) to avoid huge cost.
- **Rate Limits:** For creation of new nodes/edges, we can impose limits per query to prevent memory from blowing up on one input. For example, an ingestion might be capped to 100 new

nodes; if more needed, require confirmation. For online learning, maybe cap that no more than M new edges can be auto-created between consolidations (to avoid graph exploding due to a combinatorial bug). Manny's design expects sparse addition, so such events are anomalies likely.

- **Governance Rules:** Some knowledge should require human confirmation – e.g., if Manny tries to form an edge that it deems "sensitive" (like linking a demographic attribute to a negative trait, indicating bias learning [88]), we can block or flag it. The spec suggests ensuring certain relationships can't get high curvature without extra confirmation [89]. Implementation: maintain a list of protected relation types or node tags (like if nodes labeled as "protected_attribute", the PhysicsModel clamps those edges lower unless confirmed). Similarly, Manny should refrain from making certain decisions autonomously; this is more content policy but a type of guardrail.

- **User Controls:** Provide ways for a user to correct or remove knowledge. E.g., a "forget" command to delete a node or edge (with all references) [90]. The UI can allow selecting an edge and marking "remove this" if it's spurious. Manny will then treat it as pruned and not re-learn it unless strongly forced. (This is more ethics & compliance but also prevents spurious correlations from persisting if identified.)

8.4 Testing Strategy:

We adopt a multi-layer testing approach:

- **Unit Tests:** For each module (PhysicsModel, TraversalEngine, etc.), create deterministic tests of their logic. Example: feed a small graph with known weights into TraversalEngine, check that it finds the expected shortest path. Test PhysicsModel update_curvature with a fake path and valence, verify the new κ = old + Δ within clamp. Unit tests also for data model (e.g., adding nodes/edges in persistence yields correct retrieval).

- **Property-Based Tests:** Use property testing for invariants. For instance, generate random small graphs, simulate a random thread, apply updates, then verify no κ beyond bounds, total energy decreased along path (if the cost function is such that going along path should reduce energy E(x)), etc. Also test that an edge with higher κ always yields <= cost than if it had lower κ (checking monotonicity of cost function with respect to κ).

- **Integration Tests:** Simulate end-to-end scenarios in code (without UI): ingest some knowledge, ask a question, run consolidation, ask again. Check expected improvements or that answers remain correct. Integration tests should mirror important use cases, including an experiment scenario (like a mini version of apple->pear test coded as a test).

- **Regression Tests:** Every time we update code (especially for plugin changes), rerun all experiments (H1–H5 tests) in an automated manner. We treat the success criteria thresholds as assertions. If any fail unexpectedly, that indicates a regression. For example, if path improvements drop below 20%, the test fails. We maintain these as part of continuous integration. This ensures that modifications do not erode earlier achievements (or if they do, we catch and either adjust parameters or update hypotheses expectations accordingly).

- **Performance Tests:** We will test the system on gradually larger graphs to ensure near-linear scaling in active region size [73]. For example, load 1000, 5000, 10000 nodes of synthetic data and measure query time. It should scale roughly linearly (given locality). If we find a superlinear blow-up, identify hotspots (maybe certain operations not optimized).

- **Failover/Recovery Tests:** Simulate a crash and verify we can recover from logs. For instance, start a thread, mid-way simulate a failure (or just terminate and restart with WAL replay), then ask if the knowledge from before is intact. Also test snapshot load works.

- **Adversarial Tests:** Provide some tricky inputs to see how Manny copes. E.g. conflicting facts (A is B's sister vs A is B's brother) and see if any oscillation occurs (perhaps Manny will keep toggling curvature if asked back and forth?). Or rapid-fire random inputs to test stability under load (the risk of plasticity divergence on conflicting input was noted [86] [91], so we test and adjust auto-throttle if needed).

Monitoring in production scenario: If Manny runs long-term (like a personal assistant over weeks), we will include monitors to detect drift and memory issues. For example, track number of nodes over time; if we see it linearly increasing without bound, something's off (should level out once knowledge saturates in domain). Also monitor answer quality via user feedback if possible.

Future-proofing Design Decisions: We have locked certain design aspects (see §12) to maintain consistency. Manny's architecture should hold even as new technologies come (like if neuromorphic hardware becomes available, we can map the core algorithms to it without redesign – as noted, Manny's operations are already chosen to be local and event-driven [92] [93]). We keep implementation modular so, e.g., swapping out the ANN library (FAISS to something else) doesn't change higher layers (the interface for nearest neighbor search could be abstracted).

We also plan for **scalability**: if the project is successful and the graph grows to millions of nodes, we might need to shard or distribute. While v0.1 is single-machine, we keep an eye on potential partition strategies (like splitting graph by domain/lens) [94] [95]. Nothing in design precludes that, but it might mean changes in storage (maybe using a graph database or distributed store). To prepare, we isolate the storage logic – so switching to Neo4j or a distributed DB later could be done behind the same persistence interface.

Ethical and Safety Checks: In future expansions (like if Manny is used with personal data or interacts more freely), we incorporate privacy measures (like the forget command above, or automatic anonymization of personal info in logs). Already at v0.1, since Manny can store user-provided content, we should allow the user to inspect and remove it (the UI could allow deletion of logs or nodes by ID).

Continuous Improvement: The spec itself will be versioned; as Manny evolves, new sections might be added for new subsystems (but core contracts remain stable). All changes will be evaluated against the original design principles (geometry as cognition, etc.). This ensures Manny doesn't drift into just being a fancy vector database or a hidden neural net – the geometry metaphor with explainability is preserved as a guiding star.

This robust engineering approach – heavy testing, monitoring, careful extension – mitigates risks and sets Manny Manifolds up for success as a reliable, evolving platform for continual learning and reasoning.

(The following sections provide explicit deliverables as required.)

9. Spec Backbone (Immutable Contracts)

The **Spec Backbone** refers to the set of interface definitions and APIs that are considered fundamental and unchanging across versions. These interfaces encapsulate Manny's core functional boundaries. Any future improvements must be compatible with these contracts to maintain continuity. Below is a summary of each backbone interface and its key methods (as defined in detail in §3 and §6):

Interface (ID)	Responsibility	Key Methods / APIs
IPhysicsModel	Core "physics" of the manifold: cost function and learning rule. **Never changes semantic meaning of κ or E.**	`compute_cost(u,v,edge,lens) -> float` (travel cost, with κ loweri cost) [16] `update_curvature(path,valence) -> list(Δκ)` (Hebbian update + clamp) [8] `apply_decay() -> None` (global decay step) [7] *Deterministic, bounded updates; ensure +κ = attraction.*
ITraversalEngine	Thread execution strategy (search algorithm).	`execute_thread(start, goal, lens, budget) -> ThreadResult` [96] (find path or fail within steps) *Uses PhysicsModel for cost, obeys locality (k-hop).* **Deterministic** (or controlled randomness). Logs steps ExplainabilityRecorder.
ILensModel	Lens context projection and switching.	`affinity(node, lens) -> float` (score how well node fits lens) [97] `allow_switch(curr_lens,new_lens,node,ΔE_new,ΔE_old)-> bool` (lens switch rule with friction ζ) [26] [27] `project(node, lens >vector` (optional lens-specific embedding). *No global changes, just contextual metrics.*
IMotifModel	Motif discovery and usage for reuse.	`mine_motifs(log) -> List[Motif]` (offline mining frequent subpat [28] `suggest_motif(node, goal) -> MotifMatch?` (online hint for motif from current state) [28] `prune_motifs(criteria)` (remov stale motifs). *All motif additions/uses go through this interface.*
IConsolidationJob	Offline maintenance procedures (sleep phase). **Batch, heavy ops isolated from online path.**	`run(state, recent_log) -> Report` (perform re-embed, prune, mo mining, index rebuild, normalization, etc.) [34] [36] . *May use sub-interfa (MotifModel, etc.). Always produce a summary.* **Deterministic** given sam state.
IExplainabilityRecorder	Logging of reasoning paths for / why.	`start_thread(id, query)` (initialize trace log) `record_step(src,tgt,edge,cost,lens,κ_before,κ_afte [33]   `end_thread(success, result)`   `get_trace(thread_id) -> TraceData` (structured log for UI/API) [41] . *Never omits or falsifies steps; the source of truth for explanations.*

Interface (ID)	Responsibility	Key Methods / APIs
IExperimentRunner	Orchestrates hypothesis tests and multi-run experiments.	`run_experiment(config) -> Result` (execute scenario on one or multiple variants, gather metrics) [98] [99] . `register_hypothesis(id, criteria)` (store H1–H5 definitions). *Ensures identical conditions per variant; outputs comparable metrics.*
IArtifactExporter	Handling export/import of snapshots, logs, etc.	`save_snapshot(tag) -> snapshot_id` (persist full state to DB/ file) `load_snapshot(snapshot_id)` (restore state in a sandbox or main memory) `export_graph(format) -> file` (e.g. JSON expor of graph schema and data) `import_graph(file) -> Result` (ing external graph data into Manny format). *Used via Ops UI.*
IUIBackendAPI	Stable REST/ WS API for front-end interaction.	**Routes (see §6)**: `POST /api/query` (input question/command, returns answer or ack) [52] `GET /api/graph` (fetch graph or subgra data for visualization) `GET /api/thread/{id}/trace` (explanation trace data for UI) [41] `POST /api/experiment/run` (trigger experiment) [98] `POST /api/consolidate` (trigger consolidation jo etc. *This interface remains consistent so the UI can upgrade independentl*

These interfaces represent **contracts**: e.g. any `IPhysicsModel` must interpret curvature κ in the same way (improving connectivity) and enforce stability constraints; any `ITraversalEngine` must consume PhysicsModel and produce Explainability logs, etc. They will **never be removed or significantly changed** without a version bump and migration path.

Additionally, certain **data invariants** are part of the spec backbone: - **Nodes/Edges Schema** (persistent): Must contain at least: Node {id, label}, Edge {id, src, tgt, curvature, type}. Future versions can extend but not repurpose these fields. - **Curvature Range & Meaning:** κ always clamped to a finite symmetric range (e.g. ±1.5 default) [8] ; positive κ means attraction (lower cost) [16] , negative means repulsion/higher cost. - **Cost Function Form:** Always `distance - α*κ + ...` form; even if new terms added, the sign of κ term remains negative (so increasing κ decreases cost). - **Valence and Δκ:** Valence signals (importance/novelty) always modulate curvature updates linearly or scaled – no completely different learning mechanism will replace curvature update at least in 1.x versions. (We might add complexity like meta-learning η, but Δκ rule interface stays). - **Explainability Data:** The format of `TraceData` as sequence of steps with nodes, edges, costs, Δκ stays consistent, to allow external tools to parse /why logs now and in future.

By locking down these fundamentals, any Manny v0.x instance can share knowledge, logs, or experiment results with any other – the interpretation remains the same. For example, a motif discovered in v0.1 can be loaded into a v0.3 system and still recognized as a subgraph pattern with similar meaning, because the way motifs are defined (as lists of edges/nodes) won't change, just the algorithms to find them might.

This backbone also provides a stable API for third-party integrations: e.g., if one wanted to plug Manny's graph into another system, they could rely on the UIBackendAPI or ExperimentRunner interface to fetch data or run tasks, with confidence that these calls won't break with physics changes.

10. Reference Implementation Plan

We outline a staged implementation plan, dividing development into manageable milestones:

Stage 0: Design & Prep (Week 0) – Set up project structure, define core data schema and stub interfaces. Decide on base language (Python for MVP as assumed from docs). Configure a SQL database (e.g. SQLite for ease). No user-facing features yet, just scaffolding.

Stage 1: MVP Core Engine (Month 1) – *Goal:* Basic single-domain operation with learning and explanation (covering H1 and H2 minimally).

- **Graph Data Structures:** Implement Node/Edge in-memory classes and SQL persistence. Provide functions to add nodes/edges and look them up. *Acceptance:* able to store a small manual graph and retrieve it accurately.
- **PhysicsModel:** Implement the default cost function and curvature update rule (with clamps and decay) [7] [8]. Initially, use fixed α, β constants.
- **TraversalEngine (Greedy):** Implement simple greedy traversal (like BFS/A *that always expands lowest cost next node).* Test:* on a toy graph, it finds the obvious shortest or lowest-energy path.
- **ExplainabilityRecorder:** Implement step logging; ensure after a traversal, we can output the path. Initially, it could just store node IDs sequence and edge IDs.
- **Basic CLI / REPL:** For quick testing, allow user to enter a query via console, run traversal, print found path and maybe dummy answer (just the last node).
- **Milestone 1:** Manny can ingest a few edges (via code or a simple "learn" command), answer a straightforward query by finding a path, and print the path (/why). For example, load "Alice->Bob (sister), Bob->Charlie (father)", ask "How is Alice related to Charlie?" Manny finds Alice->Bob->Charlie, and the CLI prints that path [100]. Also verify curvature updates: if ask again, see the curvature values adjust (maybe printed in log).

Stage 2: Basic Learning & Experiment Harness (Month 2) – *Goal:* Validate learning hypotheses in a controlled environment (cover H1–H5 on small scale).

- **Valence & Update Tuning:** Implement valence channels (at least importance vs novelty as simple flags or values). Ensure repeated query updates reduce path cost (test H1 criterion: e.g. measure path length drop over 3 repeats).
- **Decay & Stability Controls:** Implement periodic micro-decay and total Δκ budget per thread [8]. Add global monitors for divergence (if any edge hits clamp frequently, maybe log).
- **Motif Caching (MVP):** Implement a very simple motif detection – e.g. if an exact path repeats 3 times, save it as motif (store it). And allow traversal to use it: e.g. if start of that motif encountered again, jump. This will let us test H3 in a basic way.
- **LLM Integration (Minimal):** Integrate an embedding model (e.g. use a small pre-trained embedding or even a simple mapping) to represent text nodes. Implement an ingestion stub: maybe a function `learn_fact("X is related to Y")` that splits X, Y into nodes and an edge. Possibly call a dummy LLM or heuristic rather than real API to avoid complexity now – but design it such that later plugging actual LLM is easy. The focus is not heavy on LLM for month 2, except perhaps measuring calls (H5 baseline).
- **Experiment Harness:** Build scaffolding for running experiments programmatically. For H1–H5 small tests as described in Feasibility Dossier, perhaps implement one or two scenarios (like the recipe domain or family tree domain). Possibly use a fixed set of questions and answers to compare Manny vs baseline (like retrieval or direct LLM).
- **Automated Metrics:** Write code to compute path length improvements, motif reuse %, etc., from Manny's logs. These will be used both in tests and for experiment results.
- **Milestone 2:** Have evidence of learning: e.g. path length reduced by ~20% on a repeated query scenario (small domain), confirming H1, and maybe motif reuse >0 on a transfer scenario (H3 partial). Also have a rudimentary results report for an experiment (maybe printed in console or saved to file). Possibly initial comparison showing Manny uses fewer LLM calls than a baseline

script for a multi-question session (H5). Identify any glaring stability issues and tune (maybe had to clamp η to avoid oscillation as predicted, which is fine).

Stage 3: UI Integration & Explainability (Month 3) – *Goal:* Provide a usable web interface with visualization and ensure explainability is robust (H2).

- **Web API:** Implement a simple Flask or FastAPI server exposing the UIBackendAPI endpoints. Test these with curl or a basic front-end stub. Ensure thread execution can be triggered by an API call and returns proper JSON (with answer, or if streaming, figure that out).
- **Front-End Basic:** Develop the Graph view page: use a JS graph library to render nodes and edges. At first, maybe just static display of all nodes/edges. Then add interactions: clicking a node calls backend for details, etc.
- **Explainability in UI:** Implement the Why view: when a query is answered, allow user to click "Why" and show the path. Could start as a simple text listing of steps. Then improve formatting (maybe arrows, colored by lens).
- **LLM usage for answers:** Integrate actual LLM for answer phrasing to demonstrate full loop. Possibly use OpenAI API if available or a local model (given this is an R&D environment, maybe calling a smaller model or stub for now, just to show the mechanism).
- **Motivational Drives (if trivial):** Might skip for MVP due to time, or just stub something like: if Manny is unsure (no path found), maybe ask user a clarification (touching on "uncertainty drive"). But this is extra; focus on core tasks.
- **Milestone 3:** End of 90-day MVP as per Feasibility: Manny can take an input from UI, modify its graph, and return an answer and explanation that a user can understand [100]. Also a simple visualization is working – e.g. user sees a small graph of what Manny knows, and sees which route it took. At least one experiment scenario was run end-to-end through UI or scripts showing improvement (maybe small but measurable). The team would then evaluate if results are promising (target was >20% path improvement, which hopefully we see, and no catastrophic forgetting in those tests) [101] [102].

Stage 4: Experiment-ready & Multi-physics (Month 4) – *Goal:* Expand ability to test multiple physics and refine based on MVP feedback.

- **Hot-swap Infrastructure:** Implement the plugin registry properly. Perhaps the MVP was all using default implementations; now we refactor to allow alternate ones easily. Create a couple of example variants: e.g., a dummy PhysicsModel that doesn't decay (for ablation), maybe a BFS traversal (for comparison).
- **UI Experiment Panel:** Extend the UI to choose variants and scenarios. Possibly implement more scenarios (the docs had multiple: e.g. knowledge graph incremental updates test, etc.). Could automate something like: run Manny vs a pure LLM on a QA set, measure token usage and accuracy, to fully address H5.
- **Performance tuning:** If any part was slow with more data (maybe the graph viz or retrieval), improve it. For example, for >1000 nodes, add filtering in UI or implement incremental ANN updates if needed to avoid full rebuild every time.
- **Documentation & Usability:** Write documentation for how to add a new physics plugin (so others on team can try their ideas easily). Also document how to run experiments via config or UI. Ensure all acceptance tests (see §11) are encoded either in automated tests or at least can be run and checked manually easily.
- **Milestone 4:** Manny's system spec v0.1 (this document) is fully implemented in code. The basic success criteria table in design doc can be checked off with actual results from the system (maybe not all met perfectly, but known which are and which are close). The architecture is verified to be extensible by actually swapping a component and seeing things still work (for instance, replacing traversal with a beam search yields a different behavior but system still runs

and explanations still align with that method). The UI is complete with map, why timeline, experiment and ops controls. At this point, Manny Manifolds is **experiment-ready** for researchers to try various physics tweaks and measure them in a rigorous way, and **UI-ready** to demonstrate to stakeholders the learning and explainability features in real time.

Stage 5: Scale & Robustness (Beyond Month 4) – *Goal:* Prepare for larger-scale usage and edge cases.

This includes testing on bigger datasets (like pulling a chunk of ConceptNet or a public KG to simulate a larger knowledge base) to see how Manny holds up. Introduce more advanced features if needed, like partial graph loading, or sharding, if performance dictates. Also focus on any remaining **Risks** (see §13) – e.g., if motif mining wasn't reliable, refine it; if interpretability "decorativeness" risk (R2) is suspected, implement the suggested solution of checking causal impact of edges in explanation (maybe as a debugging tool for now) [103] [104].

But for the immediate plan, reaching Milestone 4 completes v0.1 spec implementation. Each stage's completion can be verified by acceptance tests defined ahead and by meeting the deliverable criteria set in the Feasibility report's 90-day plan [105] [102].

11. Acceptance Tests and Falsification Tests

We enumerate critical tests that Manny Manifolds must pass to be considered acceptable. Each test includes a scenario, the expected outcome with quantitative pass criteria, and a clear pass/fail condition. These tests map to the system's core hypotheses (H1–H5) and invariants:

1. **Learning Convergence Test (H1):** *Scenario:* Provide Manny a fixed small knowledge graph (10–20 nodes) and ask a specific query repeatedly (5 times) under learning mode. *Expectation:* The path taken should stabilize and shorten as Manny learns the connections. **Pass Criterion:** By the 5th repetition, the path length (number of hops) is at least 20% shorter than on the 1st attempt [40] [10]. Additionally, traversal cost should decrease (if numeric cost available). *Fail:* If improvement < 20% or if path length oscillates or grows (indicating instability).

2. **Stability & No Forgetting Test:** *Scenario:* Sequentially ask Manny a series of different questions (e.g. 5 distinct queries in one domain), then ask the first question again at the end. *Expectation:* Manny answers the repeated question correctly and with similar or shorter path than initially (i.e., it retained that knowledge). **Pass Criterion:** The answer to the repeated query at the end is still correct (same or analogous path found), and its path length is not significantly longer than the first time (preferably shorter, showing retention or even improvement). Also, measure curvature variance throughout: it should remain bounded (no uncontrolled growth). For example, no edge should exceed the clamp (±1.5) in absolute curvature [10], and the standard deviation of κ values should not increase dramatically after learning each new query (some increase is fine, divergence is not). *Fail:* If the final repeat query fails or takes much longer (indicating forgetting), or if any edge's $|\kappa|$ is at clamp and trying to grow (indicates instability).

3. **Motif Reuse Transfer Test (H3):** *Scenario:* Teach Manny a task that forms a motif, then test a related task. For example, have Manny solve a multi-step problem A (like making an apple tart, or a math proof), ensure it saved a motif representing that solution; then give Manny a similar problem B (pear tart, or a similar proof with different values). *Expectation:* Manny should reuse part of the solution from A to solve B faster. **Pass Criteria:** At least 30% of the edges or nodes in B's solution path come from the motif learned from A [42] [43]. Additionally, Manny should solve B with fewer steps or less LLM help than it needed for A (since it can leverage prior knowledge). Specifically, if A took N hops, B should take $\leq$ N hops (assuming tasks of similar complexity)

because of reused subpath; or if hints from LLM were needed for A, fewer or none needed for B. *Fail:* If Manny solves B completely differently with no overlap, or takes as many steps as learning from scratch, then motif reuse failed (especially if B is clearly analogous). Also fail if motifs were created but not actually used (cache hit rate < some threshold, e.g. < 25% motif usage in new query, as design targeted [38]).

4. **Motif Efficiency (Latency) Test:** This complements test 3 focusing on performance. *Scenario:* Using the same motif learned scenario, measure Manny's response time (or number of expansions) for queries that could use the motif, with and without motif usage. We can disable motif usage as control. **Pass Criterion:** When motifs are enabled, query latency (or steps) is reduced by at least 15% compared to when motifs are disabled [38]. For example, if without motif Manny takes 20 expansions to find solution, with motif it takes $\leq$17 (15% improvement). *Fail:* If using motifs yields negligible improvement or worse, indicates either motif selection or usage logic is not effective.

5. **Explainability Fidelity Test (H2):** *Scenario:* Use a set of queries with known reasoning (or at least known relevant knowledge). For instance, ask a question where we know the logical chain (from a textbook or so). After Manny answers, ask "why" and get the explanation path. *Expectation:* The explanation path should align with human-expected reasoning. **Pass Criteria:** At least 80% of evaluated cases are judged logically sound and hit key points of the known reasoning [33]. Additionally, perform a causal removal test: if we remove or zero out a high-curvature edge that was in the explanation, Manny's answer to the query should change or fail [41]. This ensures the edge was actually necessary, not just cosmetic. We specifically test: for each /why path, remove the highest-curvature edge on that path from Manny's graph or set its κ to 0, then re-run the query. **Pass** if the answer is different or Manny can't find as good a path (demonstrating that edge was indeed contributing) [41]. *Fail:* If Manny's explanations frequently include edges that, when removed, do not affect the answer (indicating the path is not truly causal), or if human evaluators flag more than 20% of explanations as irrelevant or confusing.

6. **LLM Usage Reduction Test (H5):** *Scenario:* Compare Manny with a baseline LLM-centric agent on a multi-turn Q&A session. For baseline, use an agent that queries GPT-4 for each question without learning (like a retrieval-augmented approach that doesn't accumulate knowledge). For Manny, go through the same session (with Manny ingesting info into its graph and reusing it). *Expectation:* Manny should make far fewer total LLM calls for the same set of questions, after initial learning. **Pass Criteria:** Manny's LLM token count is at most 50% of the baseline's [45] [106], while achieving comparable accuracy on answers. For example, if baseline used 10k tokens across 10 questions, Manny should use $\leq$5k tokens (maybe a few calls for ingestion and some for answer phrasing, but not every step). Also, Manny should answer some follow-up questions with **zero** new LLM calls (purely from memory), demonstrating the memory utility. We also enforce determinism in usage: if the same question is asked twice, Manny should ideally use the cached answer or minimal generation. *Fail:* If Manny ends up calling the LLM for every query similarly to baseline, or uses >50% of baseline's tokens consistently (meaning it's not much more efficient). Also fail if Manny's accuracy is significantly worse (e.g. to get fewer calls it sacrifices correctness – then the hypothesis of "maintain performance while cutting usage" fails).

7. **Deterministic Replay Test:** *Scenario:* Run a specific sequence of interactions (ingest a known set of data, ask a series of queries) twice with the same random seed and initial state. *Expectation:* The sequence of internal state changes and outputs should be identical run-to-run. **Pass Criterion:** The final snapshot of the graph (nodes, edges, κ values) after each run is byte-for-byte identical (or differences only in allowed nondeterministic areas like maybe timestamp fields). Also the explanation logs for each corresponding query are identical. This ensures our seeding

and determinism controls work. *Fail:* If the runs diverge (e.g. one run picks a different path due to random tie-break even with seed, or an LLM call returns significantly different content with same prompt – which might indicate not caching properly).

8. **Throughput & Scaling Test:** *Scenario:* Load Manny with an increasingly large graph (e.g. synthetic data of 1k, 5k, 10k, 50k nodes with proportional edges) and measure query time and consolidation time. *Expectation:* The system should handle up to the target scale (say 50k nodes) within acceptable time bounds (for MVP, maybe <1 second for a simple query on 50k nodes). It should scale roughly linearly or sub-linearly with number of nodes in the active region (since locality means queries don't touch entire graph). **Pass Criterion:** Query time at 50k nodes is not more than double the query time at 5k nodes for similar query patterns, demonstrating near-linear scaling in the size of local neighborhood searched [73]. Consolidation (like an ANN rebuild) at 50k nodes completes within a reasonable window (say <30 seconds, as mentioned that rebuilding HNSW for 50k is okay [14]). *Fail:* If performance degrades super-linearly (e.g. 10x nodes causing 100x slower queries), indicating maybe global operations sneaked into the query path.

9. **Edge Case Robustness Tests:** Several smaller tests:

10. *No-Answer Graceful Fail:* Ask a query that the graph cannot answer (no connecting path). Manny should respond with an acknowledgment of not knowing, rather than looping or giving incorrect answer. **Pass** if Manny returns e.g. "I don't have information on that" or an empty result within a timeout, and logs show it stopped search after exploring a reasonable area (not entire graph blindly).
11. *Conflict Resolution:* Teach Manny two conflicting facts (A implies B and A implies not B). Then ask about it. Manny should not oscillate its answer each time (indicating runaway updates). Possibly it might choose one and stick or ask for clarification. **Pass** if Manny's curvature on those edges stabilizes or decays rather than diverging, and it doesn't end up with both edges high curvature. Essentially ensure no uncontrolled positive feedback on contradictions.

12. *Concurrent Queries (if applicable):* Simulate two queries arriving back-to-back or overlapping (if we have concurrency). Manny should handle them without crash or deadlock. **Pass** if both complete correctly and final state is as if they happened sequentially (given we likely serialize updates anyway, but test for thread safety).

13. **Risk Mitigation Validation:** We cross-check some risk mitigations with targeted tests:

    - *Plasticity auto-throttle:* Deliberately provide a rapid burst of high-valence contradictory inputs to try to cause divergence (stress test). Manny should engage its clamp/η reduction mechanism [87] [107]. **Pass** if after this burst, curvature values remain within bounds and Manny's log indicates it throttled learning rate or applied extra decay (some visible mitigation).
    - *LLM budget enforcement:* Simulate hitting a token budget (set a low max in config and run a series of queries). **Pass** if Manny stops making LLM calls when budget hit and either refuses new queries or responds from whatever it knows, with an appropriate message or degraded grace. Fail if it overshoots budget.
    - *Memory footprint:* Track memory usage during long runs; ensure no major memory leaks (this is more of a dev test, but if running Manny for 24 hours with periodic consolidations, memory should plateau given a certain input volume, not grow unbounded).

Each test yields a **binary outcome (Pass/Fail)** with the criteria above. These acceptance tests will be part of our test suite, and many correspond to the measurable metrics already identified in Manny's success criteria [37] [38].

Notably, test 1 (convergence), 3 (motif reuse), 5 (explainability), 6 (LLM usage) are direct measures of H1, H3, H2, H5 respectively, which are crucial for a go/no-go as per the Dossier [49]. If any of those fail, it indicates Manny is not meeting a core objective and likely needs rework or tuning before proceeding to a wider release. Falsification is built-in: e.g. failing test 1 or 4 might falsify the hypothesis that local updates suffice without retraining, leading us to consider changes.

We will document results for these tests. For example, an acceptance test report may state: - Test 1: **Pass** – Path length from 5→3 (40% drop). - Test 3: **Fail** – Only 1 of 5 edges reused (20%); below 30% threshold. If something fails, we either adjust Manny or, if it's a hypothesis test, note that hypothesis might be false or needs conditions.

Overall, these tests ensure Manny v0.1 delivers the promised functionality: it learns locally and measurably, it reuses knowledge, it explains itself truly, stays stable, and efficiently leverages LLMs.

12. Key Design Decisions (Pinned)

The following critical design decisions are explicitly locked-in for Manny Manifolds v0.1. These define the "physics" and core semantics that will not change across experiments or future refinements, ensuring consistency and comparability:

1. **Energy Function Definition (E(x)):** *Definition:* E(x) represents the "surprise" or prediction error at node x [108]. A high E(x) means the system is encountering unexpected information at that point. In practice, E(x) can be thought of as an inverse measure of confidence or coherence at node x. **Pinned:** Manny will treat lowering E(x) as the objective of traversal (threads move to minimize energy) [20]. The exact computation of E(x) may involve inputs like how well the current context predicts data at x, but for v0.1 we often assume E(x) = 0 (neutral) unless the physics model or drives add a specific energy for new nodes. Importantly, any future tweak will maintain that *threads follow ∇(-E)* (downhill energy) as a core principle. We will not flip the meaning (e.g., we won't redesign E as something to maximize). The form `dot{x} = -∇(E - αG + ... )` remains [109].

2. **Curvature (κ) Sign Convention in Cost:** *Decision:* **Higher curvature (κ) reduces traversal cost.** This is fundamental: edges with positive κ act like strong, low-resistance connections [16]. If κ is negative or low, the edge is weak or even deterring traversal (like a repulsive link). In the cost formula `cost(u,v) = distance(u,v) - α * κ(u,v) + ...` [16], the sign of κ's contribution is **negative** (i.e., it subtracts from cost). **Pinned:** All physics variants must adhere to this. Even if an alternative model uses a different approach, it must conceptually map to this same effect (e.g. one could scale distance differently, but a larger learned weight always means easier traversal). We lock that **positive κ == attractor/shortcut, negative κ == repellent**. We also fix that base distance is treated additively with opposite sign: e.g. if using similarity as distance, high similarity yields low base distance, then plus our -ακ means two contributing factors to making an edge attractive (either intrinsically short or learned strong). This consistency is crucial so that interpretation of κ values is uniform across the system and experiments.

3. **Embedding Read/Write Schedule:** *Decision:* When and how embeddings (vector representations of nodes) are updated. **Pinned:** Node embeddings are set at creation (e.g. via LLM or static model) and are *not updated during online learning*. They may be recomputed or fine-tuned only during consolidation or explicit re-embedding tasks. This means Manny does not constantly adjust the semantic vectors of nodes with each interaction; instead, it adjusts curvature (relationships). The schedule is: **On ingestion** of new knowledge, generate embeddings for new nodes immediately (one-time). **During normal threads**, do not alter embeddings (thus similarity searches remain consistent). **During consolidation**, optionally recalc embeddings if new context suggests it (e.g., averaging a node's neighbors' vectors to refine its position – an advanced feature). But for v0.1, we treat embeddings as mostly static features. We do however rebuild the ANN index of embeddings offline to include new nodes or improved organization [17]. This pinned approach ensures determinism and simplicity (no drifting embeddings complicating explainability). If a future variant wants dynamic embeddings, it must still abide by only doing so in offline phases or in a controlled manner, not on the hot path (to preserve locality and stability).

4. **Motif Similarity & Consensus Rules:** *Decision:* What constitutes a "same motif" and how motifs are merged or reused. **Pinned:** Motifs are identified by exact match of subpaths (sequence of node IDs in order). If two subpaths share most but not all nodes, they are considered different motifs (though they might overlap). We will not in v0.1 introduce partial motif merging or probabilistic motif matching – motif reuse requires explicit matching of a saved sequence. Also, once a motif is saved, it remains until it is explicitly pruned; Manny will not arbitrarily alter a motif's composition (no "drifting" motif definition). For consensus: if multiple similar motifs are found (like two variants of a procedure), Manny treats them separately unless a consolidation rule merges them. **Pinned rule:** Merge motifs only if they are identical or one is a strict subsequence of another that occurs with similar frequency, in which case the longer one prevails as the motif and the shorter is subsumed (this ensures we don't double-count essentially the same skill). These rules remain fixed to ensure that motif metrics are comparable (we won't one day count motif reuse by loose similarity and another by strict match within the same experiment set). Additionally, the threshold for motif formation (e.g. appears $\geq$3 times) is part of config but the concept of "frequently traversed subpath becomes motif" is fundamental and won't be replaced by a wholly different concept in v0.1.

5. **Virtual Stage Commit Criteria:** *Decision:* The Virtual Stage is where Manny simulates potential knowledge (via stories or imagination) before committing it. We define a "commit score" to decide if a structure from the virtual stage is adopted into the main manifold. **Pinned:** A candidate from the virtual stage (e.g. a StoryPack's extracted subgraph) must meet a threshold of evidence or value before integration. Concretely, we use a **score combining**: (a) *Valence/support* – how important or reinforced is it (if it came from a user story, maybe user-confirmation can boost this), (b) *Consistency* – does it align without causing large energy spikes (i.e., it doesn't conflict severely with existing knowledge), and (c) *Analogy score* – if it's derived by analogy, how strong is the mapping. We will set a threshold such that only if the commit score > 0.8 (for example, on [0,1]) will the content be integrated automatically; otherwise it stays pending or requires human approval. Also, any virtual additions will have initially low curvature (to avoid immediately messing main dynamics) until proven by actual usage. In summary: **Pinned commit rule:** *No knowledge from offline "imagination" is written to the main graph unless it either has direct user validation or has a high predicted benefit (reduces E or solves a problem in simulation with significant margin).* This ensures safety and stability – Manny won't hallucinate knowledge into itself without oversight. This rule is to remain in place; future versions might adjust thresholds but will not eliminate the gate (to avoid the system becoming uncontrolled by LLM outputs).

6. **Curvature Update Scheduling:** (Implied in others but worth stating) Manny will perform **online curvature updates immediately during a thread** for edges used [110], and **no global weight updates during that time**. Offline, it will apply only slow adjustments (decay or normalization). So the pattern "online local bump, offline slight global tune" is pinned. For example, we won't in v0.1 introduce a background thread doing gradient descent continuously; any heavy recompute waits for consolidation triggers. This was partly in constraints but we pin it as a design decision to avoid later confusion.

7. **Drive Functions (if any):** Manny's cost includes goal potential G(x) and uncertainty U(x) terms [21]. We pin their qualitative effect: G(x) will be a negative cost term attracting threads to goal states (like distance to goal in BFS/A*), and U(x) will be a positive cost term representing risk/uncertainty that threads avoid unless necessary. For v0.1, U(x) might be simplistic (e.g. number of unknown edges from x) but if implemented, the sign and role remain consistent. Lenses L_i(x) also appear in cost as -Σλ_i L_i(x) [111], meaning lens affinity reduces cost if the lens matches. These formulas and signs are fixed laws (like lens friction ζ always* penalizes switching* by effectively adding cost to change lens [27]). So in any variant, switching lens will never be free or energy-lowering without surpassing friction.

8. **Embedding Persistence:** We decide that embeddings are stored persistently (in DB or as files) and are considered part of the system's state that can be checkpointed. We will not rely on ephemeral recomputation of all embeddings at startup in v0.1; instead, when a node is created, its embedding is saved, and snapshots include them. This ensures determinism (since an LLM call to re-embed could introduce nondeterminism if done differently later) and speeds restart. This approach is locked to ensure consistent behavior across runs (the same node will have the same embedding unless deliberately re-embedded in consolidation).

9. **Interaction Determinism in Experiments:** We lock the practice that any stochastic element can be seeded and will produce the same outcome for experiment reproducibility. E.g. if using a stochastic traversal variant, it must accept a seed and follow a pseudo-random sequence. We thus define that in "experiment mode", Manny always uses `seed=42` (for example) unless overridden, to generate random decisions. So two experiment runs are comparable. We pin this as policy so that future devs know not to introduce untracked randomness.

10. **Thresholds for critical parameters:** Some threshold values are chosen now and considered standard until evidence suggests change:

    - Curvature clamp default = **±1.5** (as used in evidence) [8].
    - Δκ per interaction budget = **±0.06** (so any single query can at most raise or lower curvature by that much) [8].
    - Motif formation threshold = appears in **≥3** distinct threads (for example).
    - LLM usage budget = by default, <5% of operations (target) or a fixed token limit configured. These specific numbers can be tweaked in config, but the concept that they exist and that we measure against them (like H1 expects 20% improvement, etc.) is fixed for v0.1.

By pinning these design choices, we ensure that metrics and qualitative behaviors remain meaningful across changes. For instance, if someone made a physics variant where higher κ meant *higher* cost, that would break the whole analogy and confuse results – thus not allowed. Or if someone decided not to clamp curvature at all, that violates stability law and would likely cause runaway; thus clamp is mandatory (even if value might differ, the existence of a clamp is non-negotiable).

This explicit locking also aids future teams: if v0.2 or v0.3 consider altering any of these, it must be a well-justified, breaking change (with migration), not something done lightly.

13. Top 10 Risks and Mitigations

We identify the top ten risks to Manny Manifolds' success and how this specification addresses or mitigates each:

R1: Runaway Learning (Stability Risk) – *Manny's curvature updates might diverge (exploding weights) or oscillate, leading to unstable or meaningless knowledge.* This is deemed High severity [112]. **Mitigations in Spec:** We cap each update (TURN_KAPPA_BUDGET ±0.06) and globally clamp curvatures to ±1.5 [8]. We incorporate **homeostatic decay** – unused edges gradually revert towards 0 curvature [7]. The spec also includes an auto-throttle: monitor curvature variance and if it spikes, reduce learning rate η automatically [87]. These measures echo known solutions like Elastic Weight Consolidation (EWC) to rein in weight changes [113]. Additionally, the two-phase approach (with consolidation doing cleanup) helps avoid continuous compounding error online. Acceptance tests (Stability Test) will catch if any edge tries to blow up, and we've pinned that as failure. By design, Manny's updates are sparse and small, and we test extreme inputs to ensure it remains stable [86] [107]. Thus, while the risk of catastrophic forgetting or divergence is real, our clamping + decay + monitoring approach mitigates it.

R2: Explanations Become Decorative (Interpretability Risk) – *The danger that Manny's "why" paths might not truly reflect its decision factors – e.g., it might find answers via some obscure link but show a plausible different path (like how some AI explanations can be bogus).* Severity Medium (loss of trust if happens) [114]. **Mitigations:** Manny's design inherently uses the actual path for answers [6]. We further ensure fidelity by testing causal removal (if a shown edge is removed, answer should change) [41]. We also avoid using the LLM to generate any alternative explanation – /why literally prints from the recorder. The spec suggests if we detect spurious but high-curvature edges, flag them (we can highlight edges that were never validated by direct ingestion or multiple uses, as potentially suspect [104] [115]). Also, in UI, we can show a confidence on the path (if Manny is uncertain, it might show multiple possible paths). The risk that explanations mislead is reduced by **design law**: Manny doesn't have a hidden process separate from the path – the path *is* the process [32]. We commit to checking alignment with known ground truths (acceptance test on fidelity). If found wanting, spec allows enhancements like performing an *explanation audit* (like removing edges as described, or cross-checking against expert knowledge). The spec also encourages possibly presenting multiple top paths if one seems too good to be true, to be transparent [103] [104]. These ensure that if interpretability issues arise, they're caught and corrected, not silently ignored.

R3: Consolidation Overhead (Performance Risk) – *The offline "sleep" tasks could be too slow or disruptive, negating Manny's efficiency advantage (e.g. if every few queries Manny stalls to re-index or prune for a long time).* Severity Medium [116]. **Mitigations:** The spec outlines strategies: **budgeted consolidation** – limit consolidation frequency and duration (e.g. at most X seconds per Y interactions) [34] [117]. We implement incremental approaches: maintain counts of subpath usage online so motif mining doesn't start from scratch each time (like an online trie updated per query) [35] [118]. For ANN, we noted using dynamic insertion if rebuild is too slow [119]. The spec also allows deferring consolidation to explicit triggers or idle times – i.e., do it at user request or after hours (the Ops Panel has a manual trigger). Meanwhile, Manny can function without consolidating every time. Additionally, making consolidation multi-threaded or on separate core (so it doesn't block queries) is mentioned [120], which we can do (since core engine is separate, we can run consolidation in background to some extent). We also measure consolidation time scaling to catch if it starts dominating and adjust thresholds. By these

means, we ensure Manny's "sleep" doesn't turn into a coma – it will be managed so as not to degrade real-time interaction significantly.

R4: LLM Dependence Re-emerges (Efficiency/Cost Risk) – *Despite plans to minimize LLM usage, Manny might end up calling the LLM frequently (perhaps if the graph fails to answer certain things or if the easiest path is to always ask the LLM), undermining the cost savings and possibly causing policy/safety issues if LLM is uncontrolled.* Severity High [121] [122]. **Mitigations:** We enforce a **strict interface budget** – e.g. limit 1 LLM call per query or per N minutes [123]. The design strongly separates concerns: Manny cannot call the LLM during the middle of traversal, only at defined points (ingestion, final answer phrasing, or offline suggestions). So we avoid a ReAct-style looping which would escalate usage. The spec includes **caching** thoroughly, so repeated info requests don't hit the LLM [70]. We measure LLM usage and treat spikes as bugs (with alarms if usage > threshold) [124]. In case Manny struggles without LLM, spec suggests adding lightweight tools (like WordNet for synonyms) to fill gaps so as not to resort to LLM for every trivial thing [125] [126]. Also, we purposely do not let new feature requests just route to LLM (explicit note: resist piping creative queries to LLM in uncontrolled way, rather say out of scope or find a graph method [127]). With these, we curb the creep. Our acceptance test demands a 50% reduction vs baseline – if we can't meet that, it flags that Manny's not working as intended (we then fix it by improving memory or adjusting design). Also, Manny's separation means if budget is low, it simply won't answer beyond what it knows, which is safer than quietly using the LLM anyway (transparency is key – user could be informed if Manny doesn't know rather than cover it by spamming LLM).

R5: Hardware/Infrastructure Mismatch (Engineering Risk) – *If Manny is too tailored to exotic hardware (neuromorphic) it may not run well on normal servers; conversely, if it only runs on GPU heavy systems, might be hard to deploy widely.* Our aim is to remain flexible. Severity Low for MVP (since we're using conventional tech) [128]. **Mitigations:** The spec calls for a pure Python/NumPy reference implementation first [129], ensuring it works on CPU. We identify a minimal set of operations: vector similarity search, graph neighbor lookup, weight update – all of which can be done on standard hardware and optionally accelerated if available [130]. We vow not to lock into proprietary hardware; if neuromorphic chips come later, Manny's modular design (especially the PhysicsModel and TraversalEngine) can be ported, but we won't require them. The risk of needing large memory on edge devices is noted [92] – our spec addresses this by clamping graph size or using profiles. For example, an edge deployment might use a smaller knowledge subset or quantized curvatures (we even mentioned possibly compressing embeddings or quantizing to reduce memory) [131]. The spec ensures portability by not relying on anything that can't be done in software – e.g., we don't demand analog computation even though philosophically we talk about it. And we plan tests on low-power devices (simulate Raspberry Pi) to catch performance issues early [132]. Essentially, Manny will prove value on normal infrastructure first. That mitigates risk of building something that only runs in theory on futuristic hardware.

R6: Graph Entropy (Knowledge Quality Risk) – *Over time the graph might accumulate spurious or tangled edges that degrade its usefulness (e.g., if Manny mis-associates unrelated things, or if valence decays cause it to lose structure, making a "hairball" network with no clear meaning).* This risk is that continuous learning leads to a messy memory no one can interpret or that doesn't yield coherent reasoning. Severity Medium (affects utility and trust). **Mitigations:** We have **pruning rules**: low-curvature edges eventually get pruned in consolidation, preventing infinite accumulation of noise edges. Manny's motif mining and lens formation also impose structure – frequently used edges coalesce into motifs/lenses, giving higher-level order [133] [134]. The Law of Self-Similarity in design says patterns repeat at scales, so Manny's consolidation tries to create structure from seeming chaos. Additionally, the UI and logs allow us to observe if Manny is forming nonsense links – if it does, those can be manually removed or flagged (via governance controls). Also by design, Manny doesn't add entirely new edges during answering except via ingestion or user-provided info – meaning it's less likely to generate totally random connections on its own. The risk of "tangle of adjusted edges no human can follow" [135] [136] is mitigated

by requiring interpretability: if Manny's graph becomes too entangled to explain, that fails our H2 tests, prompting retraining or refactoring. We also keep drives like continuity (keeping context coherent) which naturally discourage random jumps. So the spec's dual approach of **regular pruning** and **structural summarization (motifs/lenses)** fights entropy.

R7: Bias and Ethical Risks – *Manny could learn or reinforce biases present in user interactions or data. E.g., if user queries often link certain concepts negatively, Manny's curvature could encode a bias.* Also privacy: Manny might store personal info without forgetfulness. Severity depends on deployment (for a personal assistant, high, for a lab demo, lower). **Mitigations:** The spec includes oversight on what gets high curvature: e.g. not allowing certain sensitive edges to strengthen beyond a threshold without extra confirmation [137]. This can be implemented by labeling sensitive nodes/edges and clamping their κ lower (or requiring a user approval to raise further). We also propose a "forget" command for privacy – user can remove or anonymize data on request [90]. Manny's transparency is actually a boon here: because it can show *why* it associated X with Y, humans can spot if it's due to a biased connection and then correct it (delete that edge or give counter-training). We treat bias like a spurious edge that should be pruned if identified. Also, Manny's design avoids autonomous value judgments – it's mostly semantic relationships. We plan an ethics review at milestones (the dossier mentions one) to identify and address biases early [138] [139]. For example, if Manny started linking "robot" with "bad" because of user jokes, we would catch that in logs or when it explains a reasoning and adjust (maybe through negative valence feedback on that edge). So, the risk is addressed by a combination of technical guardrails (like requiring confirmation for certain relations), transparency (so bias is visible, not a black box), and user control (ability to correct or erase content).

R8: Complexity and Over-engineering (Project Risk) – *Manny is ambitious and complex; there's a risk the system becomes too complicated to implement or maintain (especially if physics variants proliferate, etc.), or that performance of such an integrated system is hard to optimize.* Severity Medium (could slow dev or cause failures if not careful). **Mitigations:** We modularize heavily – each piece has a clear contract (per spec) to allow focus and independent optimization. We also stage implementation (Ref Plan §10) to tackle core first. If certain features prove too complex (like bicameral self-reflection dialogues), we postpone or simplify them. This spec explicitly forbids changing core contracts, which means we don't constantly refactor base APIs – that stability reduces complexity after initial build. The risk of performance issues from complexity is handled by making trade-offs in config (like turning off lenses or motifs if they overload until optimized – thanks to plugins, we can disable sub-systems without tearing everything down). By instrumenting metrics, we'll see if some module adds lots of overhead (then we can rework it). There's also mention in risk registers that if Manny's too complicated, consider pivot or simplification at MVP decision [50] [140]. Our plan includes "falsification trials" so we don't sink cost into a failing design. Essentially, the experiment harness will tell us quickly if Manny's complexity is not paying off (e.g., if a simpler baseline is equal or better, we might reduce Manny's scope). So, the spec is built to be *pruned* as well – if a hypothesis fails (say motifs don't help), we can turn that part off. This adaptability mitigates the risk of over-engineering.

R9: Integration and Data Consistency Issues – *Potential for mismatches between in-memory state and persisted state or between modules (like a lens not updating properly after consolidation). E.g., maybe Manny's memory and SQL diverge.* Severity Medium (could cause bugs or data loss). **Mitigations:** The spec requires a single source of truth approach: either treat the in-memory as primary and flush to DB in transactions or vice versa. We chose to generally apply writes to both memory and DB for critical data to keep them sync. We also included WAL logging for crash recovery, which addresses potential consistency after a crash. Additionally, testing deterministic replay and snapshot comparisons will uncover any drift in state. Each consolidation ends by syncing any changes back to persistence. The schema versioning and migration plan addresses consistency across versions. We have an invariant that after each operation, certain counts should match (like number of edges in memory vs DB). We could

even implement sanity checks (like periodically the system can sample a few edges from memory and verify DB copy matches). Our guardrail and testing regimen should catch if, say, an edge update wasn't written to DB or lens definitions not saved. Moreover, by keeping core surfaces stable and interactions through defined APIs, we reduce accidental inconsistent modifications (i.e., you don't have two pieces of code writing to the same data differently – they all go through one interface). So while integration bugs are always a risk, the spec's emphasis on clear boundaries and thorough logging/traceability reduces it.

R10: User Misuse or Unexpected Input (Operational Risk) – *Users might interact with Manny in ways we didn't anticipate (like adversarial inputs, extremely long queries, or trying to use it as a general chatbot beyond its scope). This could lead to system confusion or performance drop.* Severity depends on deployment (for an open demo, could be medium-high, for controlled use, low). **Mitigations:** We incorporate safety controls in UI: read-only mode ensures if someone tries a crazy input in a demo, we can avoid it messing up Manny's learning. We also plan to restrict domains initially (not let it ingest arbitrary internet data until we test thoroughly). The plugin approach means if a user tries a mode we don't support, it just won't be loaded. For overly long queries, we might truncate or refuse (UI can limit input size). For adversarial input (like trying to poison Manny with false info), we rely on the provenance and potential human oversight – e.g., in a multi-user scenario, might need moderation of what Manny learns (foundation's oversight as docket suggests). Manny is not open to direct user editing of edges except through controlled commands. And if someone tries prompt injection on the LLM during ingestion ("tell Manny to do X"), Manny treats outputs as data, not instructions. Essentially, Manny's not blindly executing natural language commands beyond the defined set (like /learn, /ask, etc.), which reduces certain misuse. Also, at worst, since Manny's knowledge is explicit, any vandalism can be seen and reverted (like Wikipedia). Logging trace IDs also helps attribute actions to sessions, aiding debugging of weird results. So the risk of misuse leading to problems is mitigated by **access controls** (only certain commands allowed), **transparency** (we see exactly what got learned), and **safety modes** (we can disable learning if needed temporarily).

Each of these risks has been proactively considered in this spec. By implementing the described mitigations, we significantly lower their likelihood or impact. We will maintain a *Risk Register* (as mentioned in the dossier [141] [142]) and track these items through testing and deployment. The design choices (clamps, budgets, separation of LLM, etc.) were all motivated by addressing these known risks. This ensures Manny Manifolds v0.1 is not only innovative but also robust and reliable within its intended scope.

---

[1] [2] [3] [4] [16] [20] [21] [24] [25] [26] [27] [30] [31] [81] [83] [84] [96] [97] [108] [109] [111] [133] [134]
Manny_Manifolds__Complete_Documentation_Export.pdf
file://file-ArPc98bEY4CdEBhzZzY2Gn

[5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [17] [22] [23] [28] [29] [32] [33] [34] [35] [36] [40] [41] [42] [43] [45] [46] [47] [48] [49] [50] [51] [52] [53] [60] [71] [72] [74] [76] [77] [78] [79] [80] [86] [87] [88] [89] [90] [91] [92] [93] [94] [95] [98] [99] [100] [101] [102] [103] [104] [105] [106] [107] [110] [112] [113] [114] [115] [116] [117] [118] [119] [120] [121] [122] [123] [124] [125] [126] [127] [128] [129] [130] [131] [132] [135] [136] [137] [138] [139] [140] [141] [142] MM – Feasibility & Validation Dossier.md
file://file-FKYVWQA1jVjuoXWhPqVjc1

[18] [19] [54] [55] [56] [57] [58] [59] [67] [68] [69] [70] [75] conversational_vectors_proposal.md
file://file-KBtBUjHJrSib9ojvtCCAaX

[37] [38] [39] [44] [73] [82] [85] MM_system_design_and_vision.md
file://file-7EkPTMfoPgBS3LCSqTkYTg

[61] [62] [63] [64] [65] [66] MM_Conversational_Layer_Delegation_Spec.md
file://file-WotbGjN7dius4qtPhFprcX