Melvin Full-System Design

Version: draft 0.1
Author: ChatGPT (with Jake)
Purpose: End-to-end specification of how Melvin's binary, graph-based, always-on system should work.

---

1. High-Level Goals

Melvin is not an LLM with legs. He is a self-modifying, always-on, binary-level graph system whose job is to:

- Survive and perform in unpredictable environments with limited data.
- Learn from very few examples (dozens, not millions) by extracting patterns and reusing them aggressively.
- Integrate all modalities—text, sensors, and motor control—inside one unified substrate.
- Rewrite his own internal "code" (graph topology + patterns), not just tune a fixed set of weights.
- Run continuously, including background "sleep" processes (replay, compression, structural updates) while awake.

Core design principle:
One binary substrate. Everything else is patterns over that substrate.

---

2. Substrate: Single Binary Graph

The entire mind is one graph, persisted in memory (and on disk via mmap). No separate systems for text, sensors, or motor control.

Core elements:

- Node table: a single array nodes[]
- Edge table: a single array edges[]
- Pattern tables: patterns[] and pattern_slots[] for compact pattern templates

2.1 Node Types

We use three logical node types, but they all live in one shared nodes[] array.

- DATA node:
  - Holds a single byte (0–255).
  - Represents raw binary chunks from any channel (text, sensor, motor, etc.).
  - Has a small vector embedding h_data for semantics.

- BLANK node:
  - Represents a variable slot (e.g., A, B, OBJ, TOOL).
  - Byte field encodes the blank ID (A/B/C…).
  - Has a vector embedding h_blank that learns what typically fills this slot.

- PATTERN node:
  - Represents a reference to a reusable pattern template.
  - Does not directly hold structure; instead points to a Pattern entry in patterns[].

  - Has a vector embedding h_pattern encoding where and when this pattern is useful.

All nodes also carry dynamic state such as activation, decay, and timestamps.

2.2 Edge Structure

Edges connect any node to any node. Structurally, all edges are the same type; semantics are encoded by flags.

- Edge fields (conceptual):
  - src: NodeId (index into nodes[])
  - dst: NodeId (index into nodes[])
  - w: weight (importance / influence)
  - flags: bitfield describing semantics:
    - SEQ: time/sequence relation
    - CHAN: channel tagging (text/motor/sensor)
    - BIND: variable binding edge
    - SEM: higher-level semantic relation
    - REPLAY: links used for episodic replay

There is exactly one edges[] array; all connectivity is expressed here.

2.3 Patterns and Pattern Slots

Patterns represent reusable templates extracted from data. They are not separate graphs; instead, they are compact descriptors stored in:

- patterns[]: each Pattern has:
  - id
  - slot_offset, slot_count (range in pattern_slots[])
  - optional pattern-internal edges (for structural patterns)
  - strength (how often it helped / reliability)
- pattern_slots[]: each PatternSlot has:
  - kind (LITERAL_DATA or BLANK)
  - byte (literal value or blank ID)
  - role/position (e.g., sequence index, graph role)

Each PATTERN node in nodes[] has an aux field that stores a PatternId. Matching and binding are done by reading from these tables.

All of this lives in the same mmapped “brain” file; the C code only provides generic update rules.

---
3. Channels and Unified Encoding

All input/output streams are bytes. To distinguish modalities, we introduce channel nodes, not separate memories.

- Channel nodes (regular nodes with special meaning):
  - CH_TEXT
  - CH_MOTOR
  - CH_SENSOR
  - CH_VISION
  - CH_REWARD
  - (can be extended)

When a byte arrives on a particular channel:

- We ensure a DATA node exists for that byte value.
- We create sequence edges linking bytes over time (SEQ edges).
- We create CHAN edges from the channel node to each relevant DATA node.

Example:
A CAN motor command frame [ID, DLC, B0…B7] is represented as a sequence of DATA nodes, each connected to CH_MOTOR via edges with flags=CHAN.

The same byte (e.g., 0xA3) may appear in text and motor, but the channel edges and surrounding patterns differentiate its roles.

---
4. Node State and Activation Dynamics

Each node i has:

- byte: for DATA/BLANK nodes, the binary payload or blank ID.
- kind: DATA, BLANK, or PATTERN.
- a_i: scalar activation (current energy / firing level).
- h_i: small vector embedding (semantic state, e.g., 16–64 dimensions).
- other fields: in-degree, out-degree, first_out_edge, timestamps, etc.

Core update idea (per tick):

1. Message aggregation:
   For each node i, gather messages from incoming edges $j \to i$:
   $m_i = \Sigma_j (w_{ji} * a_j)$

   Optionally, also aggregate vector messages for h_i:
   $H_i = \Sigma_j (w_{ji} * h_j)$

2. Nonlinear update:
   $a_i(t+1) = (1 - \alpha) * a_i(t) + \alpha * f(m_i + bias_i - decay)$
   where f is a simple nonlinearity (e.g., clipped linear or tanh).

3. Embedding update (slow drift):
   $h_i(t+1) = (1 - \beta) * h_i(t) + \beta * g(H_i)$
   where g could be normalization and projection back into a bounded range.

4. Normalization / clipping:
   Keep activations and embeddings within safe ranges to avoid blowups.

This gives a continuous-time-like system where activations propagate through the graph at each tick.

---
5. Core Loop: Always-On Operation

The brain never stops; only the distribution of activity changes.

At a high level, each global tick does:

1. Input ingestion
2. Activation propagation

3. Pattern matching and binding
4. Prediction and output gating
5. Learning and credit assignment
6. Optional replay / structural updates (sleep work)

5.1 Input Ingestion

- Collect new bytes from active channels (text, sensors, motor feedback).
- For each byte:
  - Ensure a DATA node exists for its value.
  - Activate that node (set a_i to a burst value).
  - Connect it via SEQ edges to the previous byte in that channel's stream.
  - Add CHAN edges from the respective channel node.

5.2 Propagation

- For all nodes, update activation and embeddings based on inputs from neighbors.
- This spreads information across the graph and enables downstream patterns to see context.

5.3 Pattern Matching and Binding

- For each PATTERN node (or each Pattern in patterns[]):
  - Compute a match score against relevant sub-regions of the graph.
  - Use attention-like mechanisms:
    - patterns act as queries (q_P),
    - local context nodes provide keys (k_r),
    - compute similarity $s(P,r) \approx q_P \cdot k_r$,
    - turn scores into soft weights $\alpha_{P,r}$ via softmax or similar.
  - Patterns become active relative to how well they explain the current context.

- Binding:
  - For BLANK slots within patterns, find node sets that satisfy the pattern's constraints.
  - Create or update BIND edges between BLANK nodes and candidate DATA nodes.

5.4 Prediction and Output Gating

Patterns don't just match; they also propose:

- Next expected inputs (prediction patterns).
- Appropriate motor commands for current context (control patterns).
- When to produce external output (output gating patterns).

Output gating is itself a pattern, e.g.:

- When there is:
  - An active question context,
  - A filled answer slot with high confidence,
  - No conflicting patterns,

  then strongly activate OUT_TEXT(byte) nodes.

The host code sees high activation on OUT_TEXT nodes and pushes those bytes to stdout or to motor buses.

5.5 Learning and Credit Assignment

Each tick carries prediction and reward signals.

- Prediction:
  - For each stream, the graph predicts upcoming bytes or pattern activations.
  - When reality arrives, compute prediction error.
- Reward:
  - External reward signals (success/failure, task score) are injected as bytes to CH_REWARD and/or as a scalar into the learning routine.

Local learning rule (conceptual):

- Each edge and pattern keeps an eligibility trace (how much it contributed recently).
- Weight update:
  $\Delta w_{ij} \propto r * \text{eligibility}_{ij} + \lambda * (\text{prediction\_error\_component})$
- Pattern strength:
  - Increase when pattern improves prediction or control,
  - Decrease when pattern systematically mispredicts or harms reward.

5.6 Replay and Structural Updates (Sleep Work)

When there is slack (or continuously in a background budget), Melvin:

- Replays important episodes:
  - Sequences with high surprise, high reward, or high error.
  - Uses REPLAY edges or stored indices to re-activate those traces.
- Runs more expensive pattern induction and consolidation on replayed episodes:
  - Extracts new patterns,
  - Merges similar ones,
  - Prunes unused or harmful edges and patterns.

This is how “sleep while awake” works: front-end processing and background consolidation share the same loop but may be biased to different tasks at different times.

---
6. Ingestion, Episodes, and Channels

6.1 Chunking and Episodes

While the graph is continuous, learning benefits from episodic structure.

An episode can be defined as:

- A bounded time window where:
  - Context appears (input text/sensory),
  - Melvin produces some actions (motor commands / outputs),
  - The environment returns feedback (sensors, reward).

In practice:

- Episode boundaries can be marked by special channel bytes (e.g., newline, EOS markers, task-specific flags).
- We can track an episode ID per sequence of SEQ edges for credit assignment and replay.

6.2 Channels as Context

Channels tell Melvin where bytes came from without separate memory systems.

Examples:

- Text question:
  - Bytes from microphone → processed into text → CH_TEXT.
- Visual encoding:
  - Compressed or symbolic visual tokens → CH_VISION.
- Motor commands:
  - CAN/PWM frames → CH_MOTOR.
- Feedback:
  - Joint angles, force sensors → CH_SENSOR, CH_MOTOR_FEEDBACK.
- Reward:
  - Task success/failure → CH_REWARD.

Patterns are free to link any combination of channels. For example:

- A “pick up cup” pattern links CH_VISION (cup features), CH_MOTOR (motor frames), and CH_SENSOR (force profile).

---
7. Pattern Induction from Examples

7.1 Two-Example Rule for Sequences

Given two sequences (e.g., text, motor frames, or mixed):

1. Align them positionally (simple version: same length; more advanced: dynamic alignment).
2. For each position i:
   - If the nodes are the same (or equivalent):
     - Keep that node as a constant in the pattern.
   - If they differ:
     - Create a BLANK slot at that position.
3. Post-process blank slots:
   - If slot i and j always share fillers across examples, unify them as the same variable (A, B, etc.).
4. Store a Pattern describing:
   - Which positions are constants vs blanks.
   - How these positions are linked by SEQ edges.
   - Statistics about fillers per blank.

Example with arithmetic:

- 1+1=2
- 2+2=4

Common positions: + and = → constants.
Differing positions: digits → blanks.
Left/right digits always match → unify to A.
Result differs (2 vs 4), tracked as B.

Pattern: [A] + [A] = [B]

7.2 Pattern Induction over Graphs (Objects, Tools, etc.)

For richer sensorimotor episodes, each example is a full graph over time.

Algorithm (conceptual):

1. Identify corresponding subgraphs in example 1 and 2 (e.g., the “grip and lift”

portion).
2. For nodes and edges present in both examples in similar roles:
   - Mark as constants (e.g., structure of reach → grip → lift, hinge relations of tools).
3. For nodes and edges that differ but play similar roles (e.g., red vs blue cup, scissors vs pliers):
   - Mark as blanks with relational constraints (BLANK OBJ, BLANK TOOL).
4. Store the resulting pattern as a template graph with labeled blanks and edge relations.

This is how Melvin can learn “cup-like grasp” from two cups, or “pincer tool use” from scissors and pliers.

7.3 Pattern Library Growth

Over time:

- New patterns are added when:
  - They provide significant compression (explain many episodes),
  - They correlate strongly with success/reward.
- Old patterns are pruned or merged when:
  - Rarely used,
  - Outperformed by more general descendants.

The pattern library is the “code” written by data.

---
8. Pattern Matching and Application

Patterns don’t just recognize—they route and control.

8.1 Matching

Given a live context (a subgraph/sequence currently active):

- For each pattern P:
  - Compute a match score based on:
    - How many constant slots align,
    - How well blank constraints can be satisfied,
    - Similarity in embeddings h_i between pattern slots and candidate nodes.
- Use attention-like competition:
  - Normalize scores among patterns that cover the same region.
  - Allow multiple patterns to be partially active if helpful.

8.2 Binding

For blanks:

- Find candidates that satisfy relational constraints (edges and roles).
- Create BIND edges between BLANK nodes and candidate DATA nodes.
- The BLANK embeddings h_blank update to reflect the distribution of fillers.

8.3 Using Patterns to Predict and Control

Once P is active with certain bindings:

- P can:
  - Predict missing nodes (fill blanks) → “what comes next” or “what object fits here?”.

  - Propose motor commands that historically follow this configuration.
  - Gate output (when to speak, when to act).

Examples:

- Arithmetic:
  - Pattern [A]+[A]=[B] with A bound to '3' → predict B as '6'.
- Cup grasp:
  - Pattern P_cup_grasp with OBJ bound to a new cup-like visual cluster → propose a motor trajectory previously associated with successful grasps.
- Pincer tool pattern:
  - Pattern P_tool_pincer generalizes from scissors to pliers as long as structural BLANKs can be filled consistently.

---
9. Learning: Weights, Patterns, and Structure

Learning happens at three levels:

1) Weight learning (fast, local).
2) Pattern strength / statistics (medium-term).
3) Structural learning (slow: new nodes, edges, patterns; pruning).

9.1 Weight Learning

Edges and patterns maintain eligibility traces (e.g., decaying values that track recent involvement).

Upon receiving prediction error or reward r:

- For each edge (i,j):
  $\Delta w_{ij} = \eta * r * \text{eligibility}_{ij} + \beta * (\text{prediction\_gradient\_component})$
- Eligibility_ij decays over time and spikes when the edge is active.

This gives:
- Hebbian-like plasticity modulated by reward/prediction error.
- Local, online updates; no global backprop.

9.2 Pattern Statistics

Each pattern P tracks:

- Usage count.
- Success rate (how often its predictions/control actions were correct or rewarded).
- Compression gain (how many episodes it helps explain vs memory cost).

Pattern strength increases when:

- P matches and leads to good outcome or accurate prediction.

Pattern strength decreases when:

- P repeatedly mispredicts or leads to poor reward.

9.3 Structural Learning

Structural changes include:

- Creating new patterns from episodes (induction).
- Merging similar patterns (clustering based on slot structure and statistics).
- Pruning patterns and edges that are:
  - Rarely used,
  - Weak,
  - Redundant relative to stronger patterns.

This keeps the graph compact and focused on useful regularities.

---
10. Motor, Text, and Sensor Integration

Because everything is bytes in one graph, integrating modalities is primarily about patterns over channels.

10.1 Motor Control at Binary Level

Motor output path:

- When PATTERNs plus context strongly activate a sequence of DATA nodes linked to CH_MOTOR:
  - Host code packages the active sequence into a CAN/PWM frame.
  - Frame is sent to actuators.

Motor feedback path:

- Incoming motor telemetry (e.g., joint positions, status) is converted to bytes and fed to CH_MOTOR_FEEDBACK / CH_SENSOR exactly like other streams.

Teaching motor control:

- Episodes contain:
  - Context (vision, text, internal state),
  - Motor commands (CH_MOTOR),
  - Resulting sensory feedback,
  - Reward.
- Pattern induction + weight learning discover mappings:
  - Context patterns → motor command patterns → good outcomes.

10.2 Text and Symbolic Reasoning

Text is just another channel:

- UTF-8 bytes or tokenized representations feed into CH_TEXT.
- Patterns capture:
  - Arithmetic relations,
  - Language patterns,
  - Question/answer structures.

Because text, sensors, and motor are in one graph, patterns naturally form bridging structures, e.g.:

- “If user says ‘raise arm’, and environment is safe → apply motor pattern P_raise_arm.”

---
11. Sleep, Replay, and Compression

Melvin does not turn off. When external input is sparse or while idle, he:

- Replays selected episodes using REPLAY edges.
- Runs pattern induction over replayed episodes to discover new patterns.
- Consolidates knowledge:
  - Strengthens useful patterns/edges,
  - Prunes or merges weak/redundant ones.

Replay selection criteria:

- High surprise (large prediction errors).
- High reward (successful behaviors).
- High uncertainty (unstable predictions).

This turns a few real-world experiences into many internal training passes without requiring more environment data.

---
12. Data Efficiency Principles

To work with 1/10th the data of an LLM-like system, Melvin relies on:

1. Structure-first learning:
   - Extract reusable patterns from very few examples by identifying invariants (constants) and blanks (variables).
2. Aggressive sharing:
   - Same DATA nodes reused across all contexts.
   - Patterns and blanks reused across tasks and modalities.
3. Surprise-driven updates:
   - Focus learning on episodes where predictions fail or outcomes are surprising, not on redundant data.
4. Episodic replay:
   - Reuse each real-world episode many times internally.
5. Online adaptation:
   - Always learning; no strict train/freeze phases.

---
13. Minimal Non-Learned Seed

The only parts that must be hand-designed in C:

- Data structures:
  - Node, Edge, Pattern, PatternSlot, channel nodes.
- Dynamics:
  - Activation update equations.
  - Basic normalization, decay, and noise rules.
- Learning framework:
  - Eligibility traces and local weight updates.
  - Mechanisms for pattern induction, merging, pruning.
- I/O adapters:
  - Byte ingestion from various channels.
  - Mapping OUT_TEXT/OUT_MOTOR activations to actual stdout and motor buses.
- Scheduler:
  - Basic loop and resource splitting between live input handling and background replay/compression.

Everything else—arithmetic, language structure, motor skills, “when to speak”, “how to pick up a cup”, “how to use a pincer tool”—is represented by patterns and graph structure that emerge from data.

---
14. Example Scenarios

14.1 Arithmetic: 3+3=?

- Input bytes: "3+3=?
" on CH_TEXT.
- SEQ and CHAN edges link them.
- Pattern [A]+[A]=[B] matches and binds A to '3'.
- Lower-level circuits (built from data) map '3' to value(3), compute value(6), map back to '6' DATA node.
- Output gating pattern sees:
  - Question context,
  - Answer slot filled with '6',
  - High confidence,
  then activates OUT_TEXT('6').

14.2 Picking Up Any Cup

- Two episodes: picking up a red cup and a blue cup.
- Pattern induction finds common reach-grip-lift structure and marks cup-specific details as blanks (OBJ, GRASP_REGION).
- P_cup_grasp pattern encodes “graspable object with certain affordances”.
- Faced with a new cup-like object, Melvin binds OBJ to that object and reuses P_cup_grasp motor sequence, with local adjustment guided by feedback.

14.3 Scissors to Pliers

- Two episodes: using scissors and using pliers.
- Pattern induction over their sensorimotor graphs finds a shared pincer structure:
  - Two handles, a pivot, jaws applying force.
- Pattern P_tool_pincer encodes this relational structure with blanks for TOOL and TARGET.
- Faced with new pliers or other pincer tool, Melvin can bind TOOL to the new object and reuse the action pattern.

---
15. Future Extensions

Future improvements can include:

- More sophisticated graph alignment for pattern induction (subgraph isomorphism approximations).
- Multiple timescales of memory (fast plastic edges vs slow structural changes).
- Richer internal modulators (e.g., synthetic neuromodulators influencing plasticity).
- Integration with external systems (LLMs) for distillation: use LLMs as teachers, but keep Melvin grounded and structurally adaptive.
- Advanced visualization tools for the running brain (3D node/edge display, activation overlays, pattern firing timelines).

End of Document
