GRAPH-BASED PRODUCT CONFIGURATION

A Technical Architecture for Composable Quoting, Pricing, and Production Tracking at Scale

Moving Beyond Relational Table-Driven MIS Systems

Technical Reference Document
February 2026

TABLE OF CONTENTS

1. The Problem: Why Relational MIS Systems Fail at Complexity

Traditional print Management Information Systems (MIS) were designed in an era when print products were relatively uniform: business cards, letterheads, brochures, and simple packaging. The data model reflected this simplicity. A job had a substrate, a press, a run length, and a handful of finishing operations. Pricing was a series of table lookups.

This architecture fails catastrophically when products become composable. Consider a modern specialty card product: a rigid card with a PET overlay, spot UV on select areas, foil stamping in two colors, a die-cut shape, edge painting, and a custom insert. Each layer can have its own substrate, its own set of processes, and its own quantity breaks. Processes can be conditional on other processes. The die cut affects the foil area which affects the foil cost.

1.1 The Table Lookup Bottleneck

In a relational MIS, each variable in the product specification triggers a database query. The system architecture typically follows this pattern:

1. User selects a base product. System queries the products table.
2. User selects a substrate. System queries the substrates table, joins with product-substrate compatibility table.
3. User selects a process (e.g., spot UV). System queries the process table, the process-rates table, the press-capability table.
4. System calculates pricing by joining rate tables with quantity break tables with markup tables.

For a simple product with 3 variables, this might be 8-12 queries. For your card with multiple layers and processes, this becomes 50-200+ sequential database round trips. Each round trip has network latency (typically 1-5ms even on a local database), query parsing overhead, and lock contention. At 200 queries averaging 3ms each, you are waiting 600ms just for database I/O before any business logic executes.

> **Key Insight**
> The fundamental problem is not that relational databases are slow. It is that the architecture requires one query per variable, and composable products have an unbounded number of variables. The solution is not faster queries. It is fewer queries.

1.2 The Schema Rigidity Problem

Relational MIS systems encode product structure in the schema itself. There is a layers table, a processes table, a layer_processes junction table, and so on. Each table has a fixed set of columns representing the known attributes of that entity type.

When you need to add a new process type that has attributes no existing process has (say, edge painting, which has a color attribute and a coverage percentage that no other finishing process uses), you face three bad options:

1. Add nullable columns to the processes table. This leads to increasingly sparse tables with dozens of unused columns, makes queries slower, and makes the schema incomprehensible.
2. Create a new table for the new process type with its own schema. This fractures your data model and forces the quoting engine to know about every table.
3. Use an Entity-Attribute-Value (EAV) pattern where attributes are stored as key-value rows. This solves the schema problem but destroys query performance and makes the data nearly impossible to reason about.

None of these options scale. The core issue is that relational schemas are designed for data that has a known, fixed structure. Composable products have a structure that is defined by the user at configuration time, not by the developer at design time.

1.3 The Coupling Problem

In most legacy MIS platforms, three concerns that should be independent are deeply entangled:

- Product Definition: what is this product, what are its components, what are the valid configurations?
- Pricing Logic: given a fully defined product, what does it cost?
- Production Tracking: as this product moves through the shop, what is the status of each operation?

When these are mixed together in the same tables and stored procedures, any change to one concern risks breaking the others. Adding a new process type means modifying the product definition schema, the pricing stored procedures, the UI forms, and the production tracking workflow. This is why the system is hard to improve: every change is a cross-cutting concern.

2. Core Concept: Products as Directed Acyclic Graphs (DAGs)

The central architectural insight of this document is that a composable product is not a row in a table. It is a graph. Specifically, it is a Directed Acyclic Graph (DAG) where:

- Each node represents a component, layer, process, or assembly step.
- Each directed edge represents a 'composed of' or 'depends on' relationship.
- The graph is acyclic because a component cannot contain itself (no circular dependencies).

2.1 What Is a DAG and Why Not a Tree?

A tree is a special case of a DAG where each node has exactly one parent. In a product configuration, a process can apply to multiple layers (e.g., a single lamination step that bonds layer 1 and layer 2 together). This means a process node can have multiple parent nodes, which makes it a DAG, not a tree.

In practice, most product configurations look tree-like, with occasional shared nodes. The important thing is that the data model supports both without requiring structural changes.

Here is a concrete example of a multi-layer specialty card modeled as a DAG:

```
Card (root node)
├── Layer 1: "Base Card"
│   type: substrate_layer
│   material: 350gsm Silk Art Board
│   dimensions: 88mm x 55mm
│   ├── Process: CMYK Offset Print (front)
│   │   type: print_process
│   │   colors: 4
│   │   sides: 1
│   │   coverage: 85%
│   ├── Process: CMYK Offset Print (back)
│   │   type: print_process
│   │   colors: 4
│   │   sides: 1
│   │   coverage: 60%
│   ├── Process: Spot UV
│   │   type: coating_process
│   │   area_percentage: 30%
│   │   mask_file: "logo-uv-mask.pdf"
│   └── Process: Die Cut
│       type: cutting_process
│       die_reference: "DC-4421"
│       complexity: "medium"
├── Layer 2: "Clear Overlay"
│   type: substrate_layer
│   material: 200µm Clear PET
│   dimensions: 88mm x 55mm
```

```
│    └── Process: Screen Print (white)
│        type: print_process
│        ink: "opaque white"
│        passes: 2
│        coverage: 40%
└── Assembly: Laminate
    type: assembly_process
    method: "pressure-sensitive adhesive"
    inputs: [Layer 1, Layer 2]   ← DAG: two parents
```

Every single node in this graph has a unique identifier, a type, a set of attributes specific to that type, and references to its children. This is the product definition. It is complete, self-describing, and arbitrarily deep.

2.2 Why This Model Is Fundamentally Different

In the relational model, the schema defines the structure. You must know in advance how many layers are possible, what process types exist, and what attributes each has. In the graph model, the schema defines only the rules for what types of nodes exist and how they can connect. The actual structure of any given product is data, not schema.

This means:

- Adding a new layer to a card does not require a schema migration. You create a new node and attach it to the root.
- Adding a new process type (e.g., embossing) does not require new tables. You define a new node type with its attributes and register it in the type system.
- A product with 2 layers and one with 20 layers use the same code paths. The only difference is the depth and breadth of the graph.

3. Data Modeling: The Product Configuration Graph

This chapter specifies the exact data structures that underpin the product configuration graph. These structures are designed to be stored in a document database or in a JSONB column within PostgreSQL.

3.1 The Universal Node Schema

Every node in the product configuration graph conforms to a universal schema. This is critical: the system never needs to ask 'what kind of thing is this?' to know how to traverse, store, or serialize it. The schema:

```
interface ConfigNode {
  id: string;              // UUID, globally unique
  type: string;            // e.g., 'substrate_layer', 'print_process'
  version: number;         // incremented on each mutation
  created_at: ISO8601;
  updated_at: ISO8601;

  // The type-specific attributes (open schema)
  attributes: Record<string, any>;

  // Ordered list of child node IDs
  children: string[];

  // Parent node IDs (supports DAG, not just tree)
  parents: string[];

  // Pricing function reference (resolved at quote time)
  pricing_function_id: string | null;

  // Production tracking metadata
  tracking: {
    status: 'pending' | 'in_progress' | 'complete' | 'on_hold';
    assigned_to: string | null;
    station: string | null;
    started_at: ISO8601 | null;
    completed_at: ISO8601 | null;
  };
}
```

The attributes field is the key to extensibility. It is a free-form JSON object whose expected shape is defined by the node's type. A substrate_layer node has material, dimensions, and gsm attributes. A print_process node has colors, coverage, and sides attributes. But the storage layer does not care about this distinction: it stores and retrieves the same shape regardless of content.

3.2 The Configuration Document

A complete product configuration is stored as a single document (not spread across tables). The document contains the root node and a flat map of all nodes in the graph:

```
interface ProductConfiguration {
  id: string;                          // Config UUID
  name: string;                        // Human-readable name
  version: number;
  root_node_id: string;                // Entry point for traversal
  nodes: Record<string, ConfigNode>;    // Flat map: node_id -> node
  metadata: {
    created_by: string;
    created_at: ISO8601;
    updated_at: ISO8601;
    template_id: string | null;         // If cloned from a template
    tags: string[];
  };
}
```

> **Why a Flat Map Instead of Nested Objects?**
> Storing nodes in a flat map with ID references (rather than nesting children directly) is essential for DAG support. If node A and node B both reference node C as a child, a nested structure would duplicate C. A flat map stores C once and lets A and B reference its ID. It also enables O(1) node lookup by ID, which is critical for the pricing engine.

3.3 Concrete Example: The Card as a Document

Here is the specialty card from Chapter 2 stored as an actual JSON configuration document. Study this carefully; it is the canonical reference for the data model:

```
{
  "id": "config-001",
  "name": "Premium Layered Business Card",
  "version": 1,
  "root_node_id": "node-root",
  "nodes": {
    "node-root": {
      "id": "node-root",
      "type": "product",
      "attributes": {
        "product_family": "business_card",
        "quantity": 5000
      },
      "children": ["node-layer1", "node-layer2", "node-asm"],
      "parents": [],
      "pricing_function_id": "pf-product-aggregator"
    },
    "node-layer1": {
      "id": "node-layer1",
      "type": "substrate_layer",
      "attributes": {
        "material": "350gsm Silk Art Board",
        "width_mm": 88,
```

```
        "height_mm": 55,
        "grain_direction": "long"
      },
      "children": ["node-cmyk-f", "node-cmyk-b", "node-uv",
                   "node-die"],
      "parents": ["node-root"],
      "pricing_function_id": "pf-substrate-cost"
    },
    "node-cmyk-f": {
      "id": "node-cmyk-f",
      "type": "print_process",
      "attributes": {
        "method": "offset",
        "colors": 4,
        "side": "front",
        "coverage_pct": 85
      },
      "children": [],
      "parents": ["node-layer1"],
      "pricing_function_id": "pf-offset-print"
    },
    "node-uv": {
      "id": "node-uv",
      "type": "coating_process",
      "attributes": {
        "coating_type": "spot_uv",
        "area_pct": 30,
        "thickness_microns": 25,
        "mask_file": "logo-uv-mask.pdf"
      },
      "children": [],
      "parents": ["node-layer1"],
      "pricing_function_id": "pf-spot-uv"
    },
    "node-asm": {
      "id": "node-asm",
      "type": "assembly_process",
      "attributes": {
        "method": "pressure_sensitive_adhesive",
        "registration_tolerance_mm": 0.5
      },
      "children": [],
      "parents": ["node-root"],
      "pricing_function_id": "pf-lamination"
    }
  }
}
```

Notice that the entire product, with all its layers, processes, and assembly steps, is one document. Loading it requires exactly one database read, not fifty.

4. The Node Type System: Extensibility Without Schema Changes

The node type system is what makes the architecture extensible without code deployments or database migrations. It is a registry of type definitions that describe what attributes a node of each type can have, what children it can have, and what pricing function it defaults to.

4.1 Type Definition Schema

```
interface NodeTypeDefinition {
  type_id: string;              // e.g., 'coating_process'
  display_name: string;         // e.g., 'Coating / Varnish'
  category: 'product' | 'layer' | 'process' | 'assembly';

  // JSON Schema for the attributes field
  attribute_schema: JSONSchema;

  // What node types can be children of this type
  allowed_child_types: string[];

  // Default pricing function (can be overridden per node)
  default_pricing_function_id: string;

  // Validation rules beyond schema
  constraints: Constraint[];

  // UI rendering hints
  ui_config: {
    icon: string;
    color: string;
    form_layout: FormField[];
  };
}
```

The attribute_schema field uses JSON Schema, a widely-supported standard for describing the shape of JSON data. This means validation happens at the application layer, not the database layer. When a user configures a product, the UI reads the type definition to render the correct form fields, and the backend validates the submitted attributes against the schema.

4.2 Example Type Definitions

Here are two example type definitions that illustrate how different process types can coexist without schema changes:

Spot UV Coating Type

```
{
  "type_id": "coating_spot_uv",
  "display_name": "Spot UV Coating",
```

```
  "category": "process",
  "attribute_schema": {
    "type": "object",
    "properties": {
      "area_pct": { "type": "number", "min": 1, "max": 100 },
      "thickness_microns": { "type": "number", "default": 25 },
      "mask_file": { "type": "string", "format": "uri" },
      "finish": { "type": "string", "enum": ["gloss", "matte"] }
    },
    "required": ["area_pct"]
  },
  "allowed_child_types": [],
  "default_pricing_function_id": "pf-spot-uv"
}
```

Edge Painting Type

```
{
  "type_id": "finishing_edge_paint",
  "display_name": "Edge Painting",
  "category": "process",
  "attribute_schema": {
    "type": "object",
    "properties": {
      "color": { "type": "string" },
      "color_code": { "type": "string", "pattern": "^PMS" },
      "edges": {
        "type": "array",
        "items": { "enum": ["top","bottom","left","right"] }
      },
      "total_thickness_mm": { "type": "number" }
    },
    "required": ["color", "edges", "total_thickness_mm"]
  },
  "allowed_child_types": [],
  "default_pricing_function_id": "pf-edge-paint"
}
```

The key point: these two process types have completely different attributes (area percentage vs. edge selection, mask files vs. color codes), yet they coexist in the same nodes collection without any schema changes. The type system tells the UI what form to render and tells the validator what shape to expect. The storage layer is oblivious to the difference.

4.3 Adding a New Type at Runtime

When your business needs to support a new process (say, holographic foil stamping), the workflow is:

1. Define the new type with its attribute schema, allowed child types, and default pricing function.
2. Register the type definition in the type registry (a simple database table or document).
3. Implement and register the pricing function for the new type.

4. The UI automatically renders the correct form based on the type definition. No frontend deployment needed if the UI is driven by the type schema.

There are zero database migrations. Zero schema changes. Zero changes to the quoting engine core code. The engine already knows how to traverse nodes and invoke pricing functions. It simply encounters a new type with a new function, and it works.

5. Pricing Architecture: Function Pipelines Over Table Lookups

This is the chapter that directly addresses the performance problem. The pricing architecture replaces sequential database lookups with in-memory function evaluation. The result is quoting that runs in milliseconds instead of seconds, regardless of product complexity.

5.1 The Pricing Function Contract

Every pricing function in the system conforms to a single interface:

```
interface PricingFunction {
  id: string;
  name: string;

  evaluate(
    node: ConfigNode,
    context: PricingContext,
    children_results: PricingResult[]
  ): PricingResult;
}

interface PricingContext {
  quantity: number;
  rate_tables: Record<string, RateTable>;  // Pre-loaded
  quantity_curves: Record<string, QuantityCurve>;
  markup_rules: MarkupRule[];
  global_params: Record<string, any>;  // waste factors etc.
}

interface PricingResult {
  node_id: string;
  setup_cost: number;
  unit_cost: number;
  total_cost: number;
  waste_cost: number;
  breakdown: LineItem[];  // Human-readable breakdown
  cached: boolean;
}
```

The critical design decision: the PricingContext contains all rate data, pre-loaded into memory. When the quoting engine starts, it loads all relevant rate tables in a single batch query (or from a cache). From that point on, no database queries are made during pricing computation. This is why the engine is fast.

5.2 Example: The Spot UV Pricing Function

Here is a concrete pricing function implementation for spot UV coating. Notice that it reads from the pre-loaded context, not from the database:

```
function evaluateSpotUV(node, context, children_results) {
  const { area_pct, thickness_microns } = node.attributes;
  const qty = context.quantity;

  // Read from pre-loaded rate table (already in memory)
  const rates = context.rate_tables['spot_uv'];

  // Setup cost: fixed per run
  const setup = rates.setup_base;

  // Material cost: based on area and thickness
  const area_factor = area_pct / 100;
  const thickness_factor = thickness_microns / rates.base_thickness;
  const material_per_unit = rates.material_rate
    * area_factor
    * thickness_factor;

  // Apply quantity curve (pre-loaded)
  const qty_curve = context.quantity_curves['coating'];
  const qty_multiplier = qty_curve.evaluate(qty);

  // Waste factor from global params
  const waste_factor = context.global_params.coating_waste_pct;
  const effective_qty = qty * (1 + waste_factor / 100);

  const unit_cost = material_per_unit * qty_multiplier;
  const total = setup + (unit_cost * effective_qty);

  return {
    node_id: node.id,
    setup_cost: setup,
    unit_cost: unit_cost,
    total_cost: total,
    waste_cost: unit_cost * (effective_qty - qty),
    breakdown: [
      { label: 'UV Setup', amount: setup },
      { label: `UV Material (${area_pct}% area)`,
        amount: unit_cost * qty },
      { label: `UV Waste (${waste_factor}%)`,
        amount: unit_cost * (effective_qty - qty) },
    ],
    cached: false
  };
}
```

This function executes in microseconds. There are no database calls, no network round trips, no query parsing. It is pure arithmetic on pre-loaded data. Even if you have 50 such functions in a complex product, the total pricing computation takes less than a millisecond.

5.3 The Aggregator Pattern

Parent nodes in the graph use aggregator pricing functions that roll up their children's results. The root product node, for example, sums all children and applies markup:

```
function evaluateProductAggregator(node, context, children_results) {
  const subtotal = children_results.reduce(
    (sum, r) => sum + r.total_cost, 0
  );

  // Apply markup rules (also pre-loaded)
  const markup = context.markup_rules
    .filter(r => r.applies_to(node, context))
    .reduce((price, rule) => rule.apply(price), subtotal);

  return {
    node_id: node.id,
    setup_cost: children_results.reduce((s,r) => s + r.setup_cost, 0),
    unit_cost: markup / context.quantity,
    total_cost: markup,
    waste_cost: children_results.reduce((s,r) => s + r.waste_cost, 0),
    breakdown: [
      ...children_results.flatMap(r => r.breakdown),
      { label: 'Markup', amount: markup - subtotal },
    ],
    cached: false
  };
}
```

5.4 Dependency-Aware Pricing

Some processes depend on the results of other processes. For example, die cutting cost depends on the substrate thickness, which is an attribute of the parent layer, not the die cut node itself. The pricing context solves this:

```
function evaluateDieCut(node, context, children_results) {
  // Walk up to the parent layer to get material thickness
  const parent_layer = context.resolve_node(node.parents[0]);
  const thickness = parent_layer.attributes.gsm;

  // Thicker stock = more die pressure = higher cost
  const thickness_multiplier = thickness > 300 ? 1.25 : 1.0;

  // ... rest of pricing logic
}
```

Because the entire configuration graph is loaded into memory as a single document, resolving parent or sibling nodes is an O(1) hash map lookup, not a database join.

6. The Quote Engine: Tree Traversal and Memoization

The quote engine is the orchestrator that ties together the product graph and the pricing functions. It is surprisingly simple because the complexity has been pushed into the data model and the pricing functions.

6.1 The Core Algorithm

The engine uses a bottom-up (post-order) traversal of the product graph. It evaluates leaf nodes first (processes with no children), then their parents, then their parents' parents, up to the root:

```
function generateQuote(config: ProductConfiguration): Quote {
  // STEP 1: Load all rate data in a single batch
  const context = loadPricingContext(config);

  // STEP 2: Build memoization cache
  const cache = new Map<string, PricingResult>();

  // STEP 3: Recursive post-order traversal
  function evaluateNode(nodeId: string): PricingResult {
    // Check cache first (critical for DAG shared nodes)
    if (cache.has(nodeId)) return cache.get(nodeId);

    const node = config.nodes[nodeId];

    // Evaluate all children first (post-order)
    const childResults = node.children.map(
      childId => evaluateNode(childId)
    );

    // Resolve and execute this node's pricing function
    const pricingFn = resolvePricingFunction(
      node.pricing_function_id
    );
    const result = pricingFn.evaluate(
      node, context, childResults
    );

    // Cache the result
    cache.set(nodeId, result);
    return result;
  }

  // STEP 4: Start from root
  const rootResult = evaluateNode(config.root_node_id);

  // STEP 5: Assemble the quote
  return {
    config_id: config.id,
    total: rootResult.total_cost,
    unit_price: rootResult.unit_cost,
    breakdown: rootResult.breakdown,
    all_node_results: Object.fromEntries(cache),
    generated_at: new Date().toISOString()
```

```
  };
}
```

6.2 Memoization and DAG Handling

The memoization cache is essential for correctness and performance in a DAG. If the lamination assembly node is a child of both the root product and is referenced by another assembly step, without memoization it would be priced twice. With memoization, the second reference hits the cache and returns instantly.

This also enables efficient 'what-if' quoting. If the user changes one attribute of one node (say, increasing the spot UV area from 30% to 50%), the engine can invalidate only the cache entries for that node and its ancestors. All other nodes retain their cached results. In practice, re-quoting after a single change evaluates only 2-5 nodes instead of the entire graph.

6.3 Performance Analysis

Metric	Legacy Table-Driven MIS	Graph-Based Engine
DB queries per quote	50–200+ (sequential)	1–3 (batch load)
Compute time (simple product)	200–500ms	< 5ms
Compute time (complex product)	2–8 seconds	10–50ms
Re-quote after single change	Full recalculation	2–5 node re-evaluation
Memory usage	Low (all on DB)	Moderate (rate tables in RAM)
Adding new process type	Schema migration + code	Type registration only

The speed difference is not incremental. It is architectural. The legacy system is I/O-bound (waiting for database). The graph engine is CPU-bound (doing arithmetic), and modern CPUs do arithmetic in nanoseconds.

7. Storage Strategy: Document vs. Relational Hybrid

A common misconception is that adopting a graph-based product model means abandoning relational databases entirely. It does not. The optimal architecture uses a hybrid approach that leverages the strengths of each paradigm.

7.1 What Goes Where

Data Type	Storage	Rationale
Product configurations	Document store (JSONB or MongoDB)	Hierarchical, schema-variable, loaded as unit
Node type definitions	Relational table	Fixed schema, queried by type_id, rarely changes
Rate tables	Relational tables	Tabular data, queried by ranges, updated independently
Pricing functions	Code registry (in-app)	Executable logic, version-controlled in source
Quotes (generated)	Document store	Snapshot of config + pricing results, immutable
Events / audit trail	Append-only event store	Time-series, never updated, high write throughput
Users, customers, orders	Relational tables	Standard CRUD data with referential integrity

7.2 PostgreSQL JSONB: The Practical Sweet Spot

For most teams, PostgreSQL with JSONB columns offers the best of both worlds. You get document storage semantics for configurations and quotes, relational storage for everything else, ACID transactions across both, and the operational simplicity of a single database engine.

The product configurations table would look like this:

```
CREATE TABLE product_configurations (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  name TEXT NOT NULL,
  version INTEGER NOT NULL DEFAULT 1,
  root_node_id TEXT NOT NULL,
  config JSONB NOT NULL,              -- The entire node graph
  metadata JSONB DEFAULT '{}',
  created_at TIMESTAMPTZ DEFAULT NOW(),
  updated_at TIMESTAMPTZ DEFAULT NOW(),

  -- GIN index for querying into the JSON
  CONSTRAINT valid_config CHECK (
    config ? 'nodes' AND config->'nodes' ? root_node_id
  )
);
```

```
-- Index for finding configs by node type
CREATE INDEX idx_config_node_types ON product_configurations
  USING GIN ((config->'nodes'));

-- Index for finding configs by product family
CREATE INDEX idx_config_family ON product_configurations
  ((config->'nodes'->root_node_id->'attributes'->>'product_family'));
```

The key insight is that the JSON document is the product definition, but you can still create indexes into it for search and filtering. You get the flexibility of a document model with the query power of a relational database.

7.3 Loading Strategy: One Read, Full Graph

When the quoting engine needs to price a configuration, it performs exactly one database read:

```
SELECT config FROM product_configurations WHERE id = $1;
```

This returns the entire product graph as a single JSON document. The engine deserializes it into the in-memory ConfigNode structures and proceeds with evaluation. Compare this to the legacy approach of 50-200 sequential queries, and the performance advantage becomes obvious.

Rate tables are loaded similarly in batch:

```
SELECT table_name, rates FROM rate_tables
  WHERE table_name = ANY($1);  -- Array of needed table names
```

This is typically 1-3 queries total regardless of product complexity. The rate table names needed can be determined by scanning the node types in the configuration graph.

8. Event Sourcing: Granular Production Tracking

Event sourcing is the architectural pattern that enables the granular tracking you need. Instead of updating a status field on a database row (which overwrites history), every state change is recorded as an immutable event. The current state of any object is computed by replaying its events.

8.1 The Event Schema

```
interface ProductionEvent {
  event_id: string;            // UUID
  event_type: string;          // e.g., 'process_started'
  timestamp: ISO8601;

  // What this event is about
  config_id: string;           // Which product configuration
  node_id: string;             // Which specific node in the graph

  // Who / what triggered this event
  actor: {
    type: 'operator' | 'machine' | 'system';
    id: string;
    name: string;
  };

  // Event-specific data
  payload: Record<string, any>;

  // Sequence number for ordering
  sequence: number;
}
```

8.2 Example Event Stream for a Single Node

Here is the event stream for the spot UV process on layer 1 of the card, from job creation through completion:

```
{ event_type: 'node_created',
  node_id: 'node-uv',
  payload: { initial_status: 'pending' } }

{ event_type: 'process_scheduled',
  node_id: 'node-uv',
  payload: { station: 'UV-Coater-3', scheduled: '2026-03-15T09:00' } }

{ event_type: 'process_started',
  node_id: 'node-uv',
  actor: { type: 'operator', id: 'emp-42', name: 'John' },
  payload: { station: 'UV-Coater-3', sheets_planned: 5200 } }

{ event_type: 'waste_recorded',
```

```
  node_id: 'node-uv',
  payload: { sheets_wasted: 85, reason: 'coating_defect' } }

{ event_type: 'process_completed',
  node_id: 'node-uv',
  actor: { type: 'operator', id: 'emp-42', name: 'John' },
  payload: { sheets_good: 5115, duration_minutes: 45 } }
```

Because every event is immutable and timestamped, you can answer questions that traditional status-field tracking cannot:

- How long did the UV coating actually take? (difference between started and completed timestamps)
- What was the waste rate? (waste_recorded events / sheets_planned)
- Who ran the job and on which machine? (actor and station fields)
- Was the job rescheduled? (multiple process_scheduled events)
- What was the status of this node at 2:30 PM yesterday? (replay events up to that timestamp)

8.3 Current State Projection

While event sourcing stores the full history, you still need fast access to current state for dashboards and queries. This is achieved through projections, which are materialized views that are updated as events arrive:

```
-- Materialized current state, updated by event processor
CREATE TABLE node_current_state (
  config_id UUID NOT NULL,
  node_id TEXT NOT NULL,
  status TEXT NOT NULL,
  station TEXT,
  operator_id TEXT,
  started_at TIMESTAMPTZ,
  completed_at TIMESTAMPTZ,
  sheets_good INTEGER,
  sheets_wasted INTEGER,
  last_event_sequence BIGINT,
  updated_at TIMESTAMPTZ,
  PRIMARY KEY (config_id, node_id)
);
```

This projection gives you O(1) lookup for current status while preserving full history in the event store. The projection is derived data that can be rebuilt from events at any time. If you need a new projection (say, a per-station workload view), you can create it by replaying the existing events without changing any existing code.

9. CQRS: Separating Reads from Writes

Command Query Responsibility Segregation (CQRS) is a pattern that naturally complements event sourcing and the graph-based product model. The core idea is that the data model optimized for writes (creating and modifying configurations, recording events) is different from the data model optimized for reads (listing jobs, showing dashboards, generating reports).

9.1 The Write Side: Commands

The write side handles commands: create a configuration, update a node attribute, record a production event, generate a quote. Each command validates, mutates state, and emits events:

```
// Command: Update a node attribute
async function updateNodeAttribute(cmd) {
  const { config_id, node_id, attribute_key, new_value } = cmd;

  // Load the configuration document
  const config = await db.getConfig(config_id);
  const node = config.nodes[node_id];

  // Validate against the type schema
  const typeDef = typeRegistry.get(node.type);
  typeDef.validateAttribute(attribute_key, new_value);

  // Apply the mutation
  node.attributes[attribute_key] = new_value;
  node.version++;
  node.updated_at = new Date().toISOString();
  config.version++;

  // Persist (single document write)
  await db.saveConfig(config);

  // Emit event for downstream consumers
  await eventBus.emit({
    event_type: 'node_attribute_updated',
    config_id, node_id,
    payload: { attribute_key, old_value, new_value }
  });
}
```

9.2 The Read Side: Query-Optimized Projections

The read side maintains denormalized views optimized for specific queries. These are updated asynchronously by consuming events from the write side:

```
// Read model: Job Board View (for shop floor dashboard)
CREATE TABLE read_job_board (
  config_id UUID,
  product_name TEXT,
```

```
  customer_name TEXT,
  total_nodes INTEGER,
  nodes_complete INTEGER,
  progress_pct NUMERIC,
  current_bottleneck TEXT,     -- which node is blocking?
  estimated_completion TIMESTAMPTZ,
  priority INTEGER,
  updated_at TIMESTAMPTZ
);

// Read model: Station Queue (for machine operators)
CREATE TABLE read_station_queue (
  station_id TEXT,
  config_id UUID,
  node_id TEXT,
  process_type TEXT,
  process_name TEXT,
  scheduled_time TIMESTAMPTZ,
  estimated_duration INTERVAL,
  dependencies_met BOOLEAN,
  priority INTEGER
);
```

Each read model is a projection that serves a specific use case. The job board projection aggregates node status for a high-level overview. The station queue projection denormalizes for operator-facing scheduling. Neither requires joining across tables at query time because the data is already shaped for the query.

9.3 The Benefits for Your Use Case

CQRS solves several problems you are likely facing:

- The quoting UI can read from a fast, denormalized product catalog while the write side handles complex configuration mutations atomically.
- The shop floor dashboard can read from a pre-computed job progress view without querying the event store in real time.
- Reports (cost vs. estimate, waste analysis, throughput) can be built as dedicated projections without affecting the performance of the transactional system.
- Each projection can be on its own update schedule: the job board might update every 5 seconds, while the monthly cost report projection updates nightly.

10. Caching, Performance, and Scale

10.1 Three-Layer Caching Strategy

The system uses three levels of caching to achieve consistent sub-100ms quoting performance:

Layer 1: Rate Table Cache

Rate tables change infrequently (typically when you renegotiate supplier prices). They are loaded into an in-memory cache (Redis or application-level) with a TTL of 1 hour, and invalidated explicitly when rates change. This eliminates the most common database reads.

Layer 2: Subtree Price Cache

When a product configuration has not changed, its pricing result is cached as a whole. The cache key is a hash of the configuration document content. Any mutation to any node changes the hash and invalidates the cache. This gives instant re-quoting for unchanged configurations.

Layer 3: Node-Level Memoization

During a single quote computation, the memoization map described in Chapter 6 prevents redundant computation of shared DAG nodes. This is per-request and lives only in the quote engine's working memory.

10.2 Scaling Patterns

Challenge	Solution	Implementation
High quote volume	Horizontal scaling of quote engine	Stateless quote service behind load balancer; all state in DB/cache
Large configurations (100+ nodes)	Parallel branch evaluation	Promise.all() on independent subtrees; merge at aggregator
Many concurrent users editing configs	Optimistic concurrency	Version field on configs; reject writes with stale version
High event throughput	Partitioned event store	Partition by config_id; each partition is append-only sequential
Read model latency	Async projection updates	Event consumers update read models; tolerate 1-5s staleness

10.3 Parallel Subtree Evaluation

One of the most powerful performance optimizations in the graph engine is parallel evaluation of independent subtrees. In the card example, Layer 1 and Layer 2 have no dependencies on each other. Their pricing subtrees can be evaluated in parallel:

```
async function evaluateNodeParallel(nodeId, config, context, cache) {
```

```
  if (cache.has(nodeId)) return cache.get(nodeId);

  const node = config.nodes[nodeId];

  // Evaluate independent children in parallel
  const childResults = await Promise.all(
    node.children.map(childId =>
      evaluateNodeParallel(childId, config, context, cache)
    )
  );

  const pricingFn = resolvePricingFunction(node.pricing_function_id);
  const result = pricingFn.evaluate(node, context, childResults);
  cache.set(nodeId, result);
  return result;
}
```

For a product with 10 independent layers, each with 5 processes, this evaluates all 50 process nodes in parallel rather than sequentially. On a modern multi-core machine, this can be 5-10x faster than serial evaluation for deeply branched products.

11. Migration Strategy: From Legacy to Graph

Migration from a table-driven MIS to a graph-based architecture is a multi-phase effort. The recommended approach uses the Strangler Fig pattern: build the new system alongside the old one, progressively routing traffic to the new system until the old system can be retired.

11.1 Phase 1: Shadow Mode (Weeks 1-4)

Build the graph engine as a standalone service. Write adapters that convert existing product definitions from the legacy tables into graph configurations. Run both engines in parallel: the legacy system generates the actual quote, and the graph engine generates a shadow quote. Compare results to validate accuracy.

11.2 Phase 2: Read Path Migration (Weeks 5-8)

Switch the quoting UI to read from the graph engine while the legacy system remains the source of truth for writes. All product configuration changes still happen in the legacy system and are synced to the graph engine via an adapter. This is low-risk: if the graph engine returns an incorrect quote, the legacy system is still available.

11.3 Phase 3: Write Path Migration (Weeks 9-16)

Build the new product configuration UI that writes directly to the graph engine. Maintain a reverse sync that writes back to the legacy system for any downstream processes that still depend on it (invoicing, shipping, etc.). This phase requires the most careful coordination.

11.4 Phase 4: Legacy Retirement (Weeks 17+)

Once all read and write paths go through the graph engine, the legacy system becomes a historical archive. It can be maintained in read-only mode for auditing purposes and eventually decommissioned.

> **Migration Rule of Thumb**
>
> Never attempt a big-bang migration. The Strangler Fig approach lets you validate the new system against the old one at every stage, reducing risk dramatically. If the graph engine produces a quote that differs from the legacy system by more than 1%, that is a bug in the graph engine that needs to be fixed before proceeding.

12. Implementation Reference: Data Structures and Pseudocode

This chapter provides the complete pseudocode for all core operations. These are designed to be directly translatable to TypeScript, Python, or any language an LLM or developer will use for implementation.

12.1 Configuration CRUD Operations

```
// CREATE a new configuration from a template
function createConfiguration(templateId, customizations) {
  const template = db.getTemplate(templateId);
  const config = deepClone(template);
  config.id = generateUUID();
  config.version = 1;
  config.metadata.template_id = templateId;

  // Apply customizations (e.g., quantity, material choices)
  for (const [nodeId, attrs] of Object.entries(customizations)) {
    const node = config.nodes[nodeId];
    const typeDef = typeRegistry.get(node.type);
    for (const [key, value] of Object.entries(attrs)) {
      typeDef.validateAttribute(key, value);
      node.attributes[key] = value;
    }
  }

  db.saveConfig(config);
  return config;
}
```

12.2 Adding a Node to an Existing Configuration

```
function addNode(configId, parentNodeId, nodeType, attributes) {
  const config = db.getConfig(configId);
  const parentNode = config.nodes[parentNodeId];
  const typeDef = typeRegistry.get(nodeType);

  // Validate: can this type be a child of the parent?
  const parentTypeDef = typeRegistry.get(parentNode.type);
  if (!parentTypeDef.allowed_child_types.includes(nodeType)) {
    throw new Error(
      `${nodeType} cannot be a child of ${parentNode.type}`
    );
  }

  // Validate attributes against type schema
  typeDef.validateAttributes(attributes);

  // Create the new node
  const newNode = {
    id: generateUUID(),
    type: nodeType,
    version: 1,
    created_at: now(),
    updated_at: now(),
```

```
    attributes: attributes,
    children: [],
    parents: [parentNodeId],
    pricing_function_id: typeDef.default_pricing_function_id,
    tracking: { status: 'pending' }
  };

  // Wire it into the graph
  config.nodes[newNode.id] = newNode;
  parentNode.children.push(newNode.id);
  config.version++;

  db.saveConfig(config);
  eventBus.emit({
    event_type: 'node_added',
    config_id: configId,
    node_id: newNode.id,
    payload: { parent_id: parentNodeId, type: nodeType }
  });

  return newNode;
}
```

12.3 Complete Quote Generation with Caching

```
async function generateQuoteWithCache(configId, quantity) {
  // Check for cached quote
  const config = await db.getConfig(configId);
  const cacheKey = hashConfig(config, quantity);
  const cached = await cache.get(cacheKey);
  if (cached) return cached;

  // Determine which rate tables are needed
  const nodeTypes = new Set(
    Object.values(config.nodes).map(n => n.type)
  );
  const neededTables = typeRegistry
    .getRequiredRateTables(nodeTypes);

  // Batch-load all rate data (1-2 queries)
  const rateTables = await db.getRateTables(neededTables);
  const qtyCurves = await db.getQuantityCurves(neededTables);

  // Build the pricing context
  const context = {
    quantity,
    rate_tables: rateTables,
    quantity_curves: qtyCurves,
    markup_rules: await db.getMarkupRules(),
    global_params: await db.getGlobalParams(),
    resolve_node: (id) => config.nodes[id]
  };

  // Execute the pricing traversal
  const memo = new Map();
  const rootResult = await evaluateNodeParallel(
```

```
    config.root_node_id, config, context, memo
  );

  const quote = {
    config_id: config.id,
    config_version: config.version,
    quantity,
    total: rootResult.total_cost,
    unit_price: rootResult.unit_cost,
    setup_total: rootResult.setup_cost,
    waste_total: rootResult.waste_cost,
    breakdown: rootResult.breakdown,
    node_results: Object.fromEntries(memo),
    generated_at: new Date().toISOString()
  };

  // Cache for future requests
  await cache.set(cacheKey, quote, { ttl: 3600 });

  return quote;
}
```

13. Architecture Decision Records

This chapter documents the key architectural decisions and their rationales. These are invaluable for future maintainers and for providing context to any LLM assisting with implementation.

ADR-001: Document Storage for Product Configurations

Decision: Store product configurations as JSON documents (PostgreSQL JSONB) rather than normalized relational tables.

Context: Product configurations are hierarchical, schema-variable, and always loaded as a complete unit. Relational normalization fragments the data across many tables, requiring expensive joins to reconstitute.

Consequences: Single-read loading, flexible schema, no migrations for new product types. Trade-off: cannot use SQL joins to query across node attributes without extracting to GIN-indexed JSONB paths.

ADR-002: Pricing Functions as Code, Not Database Rules

Decision: Implement pricing logic as registered code functions rather than database-stored rules or stored procedures.

Context: Pricing logic requires conditional branching, mathematical operations, and access to multiple data points simultaneously. Database-stored rules (EAV patterns or stored procedures) are hard to test, hard to version control, and hard to debug.

Consequences: Pricing logic is testable with unit tests, version-controlled in Git, and executes at CPU speed in memory. Trade-off: adding a new pricing function requires a code deployment (mitigated by the type registration system which handles attribute changes without deployment).

ADR-003: Event Sourcing for Production Tracking

Decision: Use event sourcing (immutable event log) instead of mutable status fields for production tracking.

Context: Production tracking requires not just current status but full history: when did each step start, who did it, what was the waste, was it rescheduled? Mutable status fields destroy this history.

Consequences: Complete audit trail, ability to reconstruct state at any point in time, ability to build new projections from existing events. Trade-off: slightly more complex implementation, requires event consumers to maintain read-side projections.

ADR-004: CQRS for Read/Write Separation

Decision: Separate read and write data models. Writes go to the configuration store and event store. Reads come from purpose-built projections.

Context: The shop floor dashboard, quoting UI, and reporting system all need different views of the same data. Trying to serve all these from a single normalized database leads to either slow queries or complex denormalization that is hard to maintain.

Consequences: Each read use case gets an optimized data model. Trade-off: eventual consistency between write and read sides (typically 1-5 seconds), more infrastructure to operate.

14. Graphs vs. Relational: A Deep Side-by-Side Comparison

This chapter exists to bridge the conceptual gap for anyone coming from a relational database background. The graph model does not replace relational databases. It replaces the practice of using relational databases for data that is fundamentally hierarchical and schema-variable. Understanding exactly where each paradigm wins and loses is essential for making good implementation decisions.

14.1 The Core Difference: Fixed Schema vs. Data-Defined Structure

In a relational database, the schema (tables, columns, data types, foreign keys) is defined by a developer and enforced by the database engine. The structure of the data is determined before any data exists. This works brilliantly for data that is uniform: every customer has a name, an email, and an address. Every invoice has a date, a total, and a customer reference. The shape is known and stable.

In the graph/document model used in this architecture, the schema defines only the rules for what types of nodes can exist and how they connect. The actual structure of any specific product is determined by the user at configuration time. A two-layer card and a twelve-layer card use the same code, the same storage, and the same pricing engine. The difference between them is the shape of the data, not the shape of the schema.

To make this concrete, consider what happens in each paradigm when you need to model a new card that has 6 layers instead of the usual 2.

The Relational Approach

If the original schema was designed for 2 layers (common in MIS systems that have a front and back concept), you face a structural problem. The layers table might have a job_id, a side (front/back), and attributes for that side. To support 6 layers, you need to either redesign the table to support arbitrary layers (breaking all existing queries and reports that assumed 2), add layer_3 through layer_6 columns (the wide-table antipattern that leads to sparse, incomprehensible schemas), or create a separate generic_layers table that exists alongside the original layers table (fragmenting the model and forcing the application to check both).

Each approach requires a database migration, changes to every query that touches layers, changes to the UI, changes to the pricing logic, and regression testing of all existing products. This is typically weeks of development work and carries significant risk of breaking existing quotes.

The Graph Approach

In the graph model, a 6-layer card is simply a root node with 6 child nodes of type substrate_layer, each with their own children (processes). No schema change. No migration. No code change. The traversal engine already handles N children. The pricing engine already evaluates children recursively. The UI already renders children dynamically from the type definitions. You create the configuration, and it works.

This is not a theoretical advantage. It is the difference between weeks of development (relational) and minutes of configuration (graph).

14.2 Query Pattern Comparison

A fair comparison must acknowledge that relational databases are superior for certain query patterns. Here is an honest side-by-side:

Query Pattern	Relational Advantage	Graph/Document Advantage
Load a complete product config	Requires 5-15 JOINs across tables. Slow, complex query.	Single document read. O(1). Massively faster.
Price a complex product	50-200 sequential queries for rate lookups.	1-3 batch loads, then in-memory computation. 100x faster.
Find all jobs using material X	Simple WHERE clause on indexed column. Fast and natural.	Requires JSONB path query or GIN index. Works but less ergonomic.
Aggregate monthly revenue by process type	GROUP BY on relational columns. SQL shines here.	Requires extracting data from documents into a reporting projection.
Add a new process type to the system	ALTER TABLE or new table. Migration required.	Register a new type definition. Zero migration.
Change one attribute on one process	UPDATE one row. Simple.	Load document, modify in memory, save document. Slightly more work.
Generate a detailed cost breakdown	Complex multi-table JOIN with subqueries.	Tree traversal of pre-computed node results. Cleaner.
Historical audit: who changed what when	Requires trigger-based audit tables or CDC.	Native with event sourcing. Every change is already an event.

The pattern is clear: the graph model wins decisively for operations that involve loading, pricing, and manipulating product configurations (the hot path of your quoting system). The relational model wins for ad-hoc analytical queries across large datasets. This is precisely why the architecture recommends a hybrid: configurations in JSONB documents, analytics in relational projections.

14.3 The Real Comparison: It Is About Data Loading, Not Storage

The most common misunderstanding is that this is about replacing PostgreSQL with MongoDB or some other document database. It is not. You can implement this entire architecture in PostgreSQL using JSONB columns. The database engine is the same. What changes is the data access pattern.

In the relational MIS, fetching a product for quoting looks like this:

```
-- Legacy: 12+ queries to load one product for quoting
SELECT * FROM jobs WHERE id = $1;
SELECT * FROM job_layers WHERE job_id = $1;
SELECT * FROM layer_substrates WHERE layer_id IN (...);
SELECT * FROM layer_processes WHERE layer_id IN (...);
SELECT * FROM process_rates WHERE process_id IN (...);
SELECT * FROM process_params WHERE process_id IN (...);
SELECT * FROM quantity_breaks WHERE rate_id IN (...);
SELECT * FROM press_configs WHERE process_id IN (...);
SELECT * FROM finishing_options WHERE job_id = $1;
SELECT * FROM assembly_steps WHERE job_id = $1;
SELECT * FROM markup_rules WHERE customer_id = $2;
SELECT * FROM waste_factors WHERE process_type IN (...);
-- ... application code stitches all of this together
```

In the graph architecture, loading the same product looks like this:

```
-- Graph: 1 query to load the complete product
SELECT config FROM product_configurations WHERE id = $1;
-- Done. The entire product graph is in memory.
```

Both use PostgreSQL. Both use SQL. The difference is architectural: one fragments data across tables and reconstructs it with queries; the other stores it as a complete unit and loads it in one read. The graph approach treats the database as a storage engine for documents, not as a computation engine for joins.

14.4 The Myth of Normalization for Everything

Database normalization (1NF, 2NF, 3NF, BCNF) is one of the most important concepts in computer science, and it is correct for the data it was designed for: data where the structure is fixed, where individual fields are frequently queried in isolation, and where update anomalies must be prevented.

Product configurations violate all three assumptions. Their structure is variable (different products have different shapes). Individual fields are almost never queried in isolation (you never need just the UV coating area percentage without the rest of the product context). And update anomalies are managed through versioning, not normalization (you do not want a rate table change to retroactively alter a historical quote).

The graph model is not anti-normalization. It is normalization applied correctly: the data that is uniform and stable (rate tables, customers, users) stays normalized in relational tables. The data that is hierarchical and variable (product configurations) is stored in the format that matches its nature.

The Litmus Test

Ask yourself: when I load this data, do I always need the complete unit, or do I frequently need individual pieces? If you always load the complete product configuration as a whole to price it, display it, or track it, that is a document. If you frequently need to query one field across thousands of records (e.g., find all customers in California), that is a relational table. Use each tool for what it is good at.

15. Why This Architecture Isn't Universal: Honest Bottlenecks

If graph-based product configuration is so clearly superior for composable products, why is the entire print industry not using it? The answer involves technical trade-offs, organizational realities, and market dynamics. This chapter is deliberately candid because understanding the risks is as important as understanding the benefits.

15.1 The Talent Gap

This is the single biggest obstacle. The relational model is taught in every computer science program. Every junior developer knows SQL. The concepts in this paper, including graph data modeling, event sourcing, CQRS, function-pipeline pricing, and document storage, come from the domain-driven design (DDD) community, and they are well-established in fintech, logistics, and large-scale e-commerce. But they have barely penetrated the print MIS market.

The practical consequence: hiring developers who can build and maintain this system is harder and more expensive than hiring developers who can maintain a traditional relational MIS. If you lose a key developer, finding a replacement who understands event sourcing and graph traversal is not a job-board-and-wait-a-week situation. This is a real operational risk that must be mitigated through documentation, pair programming, and the use of LLM-assisted development (which is specifically why this paper exists in this level of detail).

15.2 Event Sourcing Operational Complexity

Event sourcing is powerful, but it introduces failure modes that traditional CRUD systems do not have:

- Projection desynchronization. If the event consumer that updates the shop floor dashboard crashes or falls behind, the dashboard shows stale data. In a CRUD system, the data is always current because reads and writes hit the same table. With event sourcing, you need monitoring, alerting, and replay mechanisms to handle consumer failures.
- Event schema evolution. Over time, the shape of your events will change. A process_started event in version 1 might have different fields than in version 2. You need an event versioning and upcasting strategy so that old events can still be processed by new code. This is solvable (there are well-established patterns), but it is additional complexity that a simple UPDATE statement does not have.
- Storage growth. Events are append-only. They accumulate forever. A busy shop generating thousands of events per day will have millions of events within a year. You need archiving, compaction, or snapshotting strategies to keep the event store performant. In a CRUD system, the database stays roughly the same size because you are updating in place.

- Debugging difficulty. When a projection shows wrong data, the debugging process is: examine the event stream, find which event was incorrect or missing, determine whether the bug is in the event producer or the event consumer, and replay events to verify. This is more complex than looking at a row in a table and checking the UPDATE query.

15.3 CQRS and Eventual Consistency

CQRS separates reads from writes, which means the read side can be stale. When an operator marks a process complete on the shop floor, the event is written to the event store immediately. But the dashboard projection might not be updated for 1-5 seconds (or longer under load). During that window, anyone looking at the dashboard sees the old status.

For most use cases, this delay is irrelevant. Nobody cares if the dashboard updates in 1 second or 3 seconds. But there are edge cases:

- An operator marks a process complete, then immediately checks the dashboard to verify. It still shows in progress. They mark it complete again. Now you have duplicate events. You need idempotency handling.
- A manager pulls a report while a batch of events is being processed. The report shows partial data. You need to communicate clearly that projections are eventually consistent, or implement read-after-write consistency for critical paths.
- Two operators update the same node simultaneously. With CQRS, the write side handles this through optimistic concurrency (version checking). But the read side might show one operator's change before the other's, causing confusion.

These are all solvable problems with known patterns. But they add complexity that a simple relational system reading from a single source of truth does not have. You are trading simplicity for power, and you need to be honest about that trade.

15.4 Document Storage Query Limitations

When your product configuration lives in a JSON document, certain queries become harder. In a relational system, 'find all jobs that used foil stamping on paper heavier than 300gsm in the last quarter' is a straightforward SQL query with a few JOINs and WHERE clauses on indexed columns.

In the document model, this query requires reaching into the JSON structure of every configuration document to find nodes where type equals foil_process and the parent layer's gsm attribute exceeds 300. PostgreSQL can do this with JSONB path queries and GIN indexes, but the syntax is less intuitive, the query plans can be unpredictable, and performance depends heavily on how the GIN index is structured.

The mitigation (and the recommendation in this architecture) is to maintain relational projections for analytical queries. The event stream feeds a projection that extracts the relevant attributes into flat relational tables optimized for reporting. But this means maintaining two representations of the same data, which is the fundamental trade-off of CQRS.

15.5 The Debugging Surface Area

In a relational MIS, a wrong quote can be debugged by opening the database, examining the rate table, and tracing the SQL query. The path from input to output is linear and visible.

In the graph pricing engine, a wrong quote means:

1. Load the configuration document and inspect the node graph.
2. Identify which node's pricing result is incorrect.
3. Examine that node's pricing function, its input attributes, and the pricing context.
4. Check whether the rate tables were loaded correctly into the context.
5. Check whether memoization returned a stale cached result.
6. If the node depends on parent or sibling attributes, check the graph traversal order.

This is more steps and more abstraction layers than the relational approach. The pricing function pipeline is more powerful, but the debugging surface area is larger. This is mitigated through comprehensive logging at each evaluation step, unit tests for each pricing function, and integration tests that compare graph engine quotes against known-correct values.

15.6 Organizational Inertia: The Real Blocker

The biggest reason print MIS vendors do not adopt this architecture is not technical. It is organizational. Consider the position of an established MIS vendor:

- They have a codebase of 500,000+ lines built over 15-20 years on a relational model. Rewriting is a multi-year, multi-million-dollar effort with high failure risk.
- Their development team has 10-20 years of expertise in the relational model. Retraining them on event sourcing and graph modeling is a 6-12 month investment with uncertain outcomes.
- Their customers have workflows, reports, and integrations built around the current schema. Migrating them is a customer-facing disruption.
- Their sales pipeline does not pause during a rewrite. Competitors are shipping features while they are rebuilding infrastructure.
- The ROI is hard to prove in advance. The performance and flexibility improvements are real, but they do not map neatly to a sales pitch. Customers do not buy 'graph-based product modeling.' They buy 'faster quoting' and 'easier product setup,' and the vendor has to trust that the architectural investment will eventually deliver those outcomes.

This is why most MIS vendors incrementally patch their existing systems rather than rearchitect. Patching is lower risk, ships faster, and keeps the team in their comfort zone. The result is the system you are currently experiencing: functional but slow, rigid, and increasingly painful as products become more complex.

Your position is different. You are not carrying 20 years of legacy. You are building new (or willing to rebuild), and you are experiencing the pain that motivates the investment. That changes the calculus entirely.

15.7 When NOT to Use This Architecture

For intellectual honesty, here are cases where the graph model is overkill and a simple relational schema is the better choice:

- Products are flat and uniform. If every product is a single substrate with CMYK print and maybe one finishing process, the relational model handles this perfectly. The graph model adds complexity without meaningful benefit.
- Quote volume is low. If you generate 10 quotes per day, the performance difference between 200ms (relational) and 5ms (graph) is irrelevant. The graph model pays off at scale, when users are configuring products interactively and expect instant feedback.
- The team cannot maintain it. If you do not have (or cannot develop) the engineering capability to operate event sourcing and CQRS, the operational complexity will outweigh the architectural benefits. A well-maintained relational system is better than a poorly maintained graph system.
- You need ad-hoc reporting as the primary use case. If your system is 80% reporting and 20% quoting, the relational model's query flexibility is more valuable than the graph model's loading performance.

16. Incremental Adoption: What to Build First and Why

The architecture described in this paper has many moving parts: a graph data model, a type system, a function-pipeline pricing engine, event sourcing, CQRS projections, and a multi-layer caching strategy. Attempting to build all of these simultaneously is a recipe for failure. This chapter provides a concrete, phased adoption plan that delivers value at each stage.

16.1 Phase 1: The Product Configuration Graph (Weeks 1-6)

Build this first because it is the foundation everything else depends on, and it delivers immediate value independent of the other components.

What you build: The ConfigNode schema, the flat-map ProductConfiguration document structure, the NodeTypeDefinition registry, and a basic CRUD API for creating, reading, and modifying configurations. Store configurations in PostgreSQL JSONB.

What you skip for now: Event sourcing, CQRS, caching. Use simple CRUD operations (load document, modify, save) with no event log.

What you validate: Can you model your most complex existing product as a configuration graph? Does it capture all the information currently spread across your legacy tables? Can your team add a new process type by registering a type definition without code changes?

Value delivered: You now have a flexible product model that can represent any product complexity. Even without the pricing engine, this is useful as a configuration tool and product catalog.

16.2 Phase 2: The Pricing Engine (Weeks 7-12)

Build this second because it delivers the most visible performance improvement and directly addresses the pain of slow quoting.

What you build: The PricingFunction interface, the PricingContext with pre-loaded rate tables, the post-order traversal engine with memoization, and concrete pricing functions for your most common process types (start with 5-8 functions covering 80% of your products).

What you skip for now: Parallel subtree evaluation (use sequential traversal first, optimize later). Advanced caching (the per-request memoization map is sufficient for now).

What you validate: Run shadow quotes. For every quote generated by the legacy system, also generate a quote from the graph engine. Compare results. Any difference greater than 1% is a bug. Track accuracy across 100+ quotes before trusting the engine for production use.

Value delivered: Quoting drops from seconds to milliseconds. Interactive product configuration with live pricing becomes possible. Users can tweak options and see price changes instantly.

16.3 Phase 3: Event Sourcing for Production Tracking (Weeks 13-20)

Build this third because it requires the product graph to exist (events reference nodes), and it adds the granular tracking capability.

What you build: The ProductionEvent schema, an append-only event store (can be a simple PostgreSQL table initially), event producers that emit events when node status changes, and a single projection: node_current_state.

What you skip for now: Multiple projections, real-time event streaming (use polling initially), complex event replay tooling.

What you validate: Can you track the status of every node in a product configuration independently? Can you answer 'when did this process start, who ran it, and how much waste was there?' from the event log?

Value delivered: Complete audit trail for production. Per-process, per-layer tracking. The ability to analyze actual vs. estimated costs at any granularity.

16.4 Phase 4: CQRS Projections and Advanced Features (Weeks 21+)

Build this last because it optimizes read patterns that only matter once you have meaningful data flowing through the system.

What you build: Purpose-built read projections (shop floor dashboard, station queue, cost analysis report). Async event consumers that maintain projections. Rate table caching in Redis. Subtree price caching.

What you validate: Dashboard latency under load. Projection consistency. Cache hit rates. Event consumer resilience (what happens when a consumer goes down and comes back up?).

Value delivered: Fast, purpose-built views for every user persona (estimators, operators, managers). Scalable read performance independent of write load.

16.5 The Golden Rule of Incremental Adoption

Build Only What Hurts Today

At each phase, you should be solving a pain you are actively feeling. If quoting is slow, Phase 2 addresses that. If you cannot track processes granularly, Phase 3 addresses that. If your dashboards are slow under load, Phase 4 addresses that. Never build infrastructure in anticipation of pain you have not yet experienced. YAGNI (You Aren’t Gonna Need It) applies to architecture patterns just as much as to code features. Event sourcing is powerful, but if your current tracking needs are simple and low-volume, a well-designed CRUD system with good logging may serve you well for a year while you focus on the configuration graph and pricing engine.

16.6 Using an LLM for Implementation

This document is deliberately written to serve as context for LLM-assisted development. When working with an LLM to implement this architecture, the most effective approach is:

1. Share the relevant chapter with the LLM before asking it to write code. For example, if you are implementing the pricing engine, provide Chapters 5 and 6 as context. The data structures and function signatures give the LLM a concrete target.
2. Start with the interfaces and type definitions. Have the LLM generate the TypeScript or Python interfaces from the pseudocode in this paper. These become the contract that all implementation code conforms to.
3. Implement one pricing function at a time. Give the LLM the PricingFunction interface and the specific rate table structure for one process type. Validate the output against a known-correct quote from your legacy system before moving to the next function.
4. Use the Architecture Decision Records (Chapter 13) as guardrails. When the LLM proposes an approach that contradicts an ADR, push back with the rationale from the ADR. This keeps the implementation aligned with the architecture.
5. Test exhaustively at each phase boundary. Before starting Phase 2, the product configuration graph should be fully validated. Before starting Phase 3, shadow quoting should show consistent accuracy. Do not build forward on an unvalidated foundation.

17. Glossary

ConfigNode — The universal data structure representing any element in a product configuration: a product, layer, process, or assembly step.

Configuration Graph — The complete DAG of ConfigNodes that defines a product. Stored as a single JSON document.

CQRS — Command Query Responsibility Segregation. A pattern where the write-side data model is different from the read-side data model. Writes go to the event store; reads come from purpose-built projections.

DAG — Directed Acyclic Graph. A graph where edges have direction and there are no cycles. Products are modeled as DAGs because a process can serve multiple layers (shared node).

Eventual Consistency — The property of a distributed system where read-side data may lag behind write-side data by a short interval (typically 1-5 seconds). A fundamental trade-off of CQRS.

Event Sourcing — A pattern where state changes are stored as immutable events rather than overwriting current state. Current state is derived by replaying the event log.

Event Upcasting — The process of converting old event formats to new ones during replay, enabling event schema evolution without data migration.

GIN Index — Generalized Inverted Index. A PostgreSQL index type that enables fast queries into JSONB documents.

Idempotency — The property where performing an operation multiple times produces the same result as performing it once. Critical for event consumers that may process the same event more than once.

JSONB — Binary JSON storage in PostgreSQL. Supports indexing, querying, and partial updates on JSON documents.

Memoization — Caching the result of a function call so that identical calls return the cached result instead of re-computing. Used in the pricing engine to avoid redundant evaluation of shared DAG nodes.

Node Type Definition — A schema that describes the attributes, allowed children, and default pricing function for a category of ConfigNode. Stored in a registry; drives UI rendering and validation.

Optimistic Concurrency — A strategy where updates check a version number before writing. If the version has changed since the data was read, the update is rejected and must be retried.

Post-Order Traversal — A tree/graph traversal that visits children before parents. Used by the pricing engine to compute costs bottom-up.

Pricing Context — The pre-loaded collection of rate tables, quantity curves, and global parameters used during quote computation. Loaded once per quote, not per node.

Pricing Function — A registered function that computes the cost of a single ConfigNode given its attributes, context, and children's results.

Projection — A read-side view of data derived from events. Optimized for a specific query pattern (e.g., job board, station queue). Can be rebuilt from events at any time.

Shadow Quoting — Running the new pricing engine in parallel with the legacy system to compare outputs. Used during migration to validate accuracy before switching over.

Strangler Fig — A migration pattern where a new system is built alongside the old one, progressively taking over traffic until the old system is retired. Named after the strangler fig tree that grows around its host.

YAGNI — You Aren't Gonna Need It. A principle that advises against building infrastructure or features in anticipation of needs that have not yet materialized.

End of Document