Laozi by Pi.Tech

Lao Zi Question Generation PRD

Aligned to Adaptive Query Classification & Adaptive Context Architecture

Version	1.0
Date	May 2026
Classification	Confidential

Table of Contents

1. Introduction

Why Adaptive Query Classification

Traditional chatbot architectures fail with rich domain data for three reasons: Data Overload (sending all available data to the LLM wastes tokens and dilutes signal-to-noise), Stale Analytics (pre-computed results drift out of date without freshness guarantees), and Wrong Context (static context selection ignores user intent, producing irrelevant answers).

The Adaptive Query Classification architecture solves these problems with a two-pass system: first classify intent, then fetch only what matters. User intent drives dynamic data selection, pre-computed analytics ensure deterministic accuracy, and a self-reinforcing feedback loop makes the system more precise with every interaction.

> *→ Core principle: The LLM never sees raw data. It only narrates pre-computed, deterministic results. This ensures consistency and auditability.*

This document describes how the Lao Zi question generation and resolution system implements the Adaptive Query Architecture. Data enters from TimescaleDB via continuous aggregation. Questions are generated deterministically. User answers flow through the classification cascade, activate targeted data pipelines, pass the freshness gate, and resolve through a constrained LLM conversation bounded by a closed outcome universe.

2. The Classification Cascade

When a user provides a free-form answer to a generated question — or enters input outside the context of a specific question — the system must classify intent before it can route to the correct domain handler. The classification cascade provides three-layer determination with progressive fallback.

Layer 1: LLM Classifier

A lightweight, low-temperature LLM call (temp 0.1, max 50 tokens) classifies the user message into a domain category. The classifier sees the last 4 conversation messages for context and returns exactly one category word from a constrained enum. If the result is a specific domain (not "general"), classification is complete.

Layer 2: Pattern Matching (Regex Safety Net)

If the LLM returns "general", deterministic regex catches cases the LLM misses. Keyword patterns override with domain-specific categories:

Keyword Domain	Pattern	Override Category
Financial	expense, revenue, cash flow, profit, margin	financial_analysis
Transaction	payment, transfer, zelle, venmo, paypal, vendor	transaction_clarification
Ownership	partner, member, owner, LLC, S-Corp, ownership	entity_structure

Layer 3: Default Fallback

Only if both layers agree on "general" does the system route to the general handler with minimal context. This is the architecture's signal that the input is truly domain-agnostic and no targeted pipeline is needed.

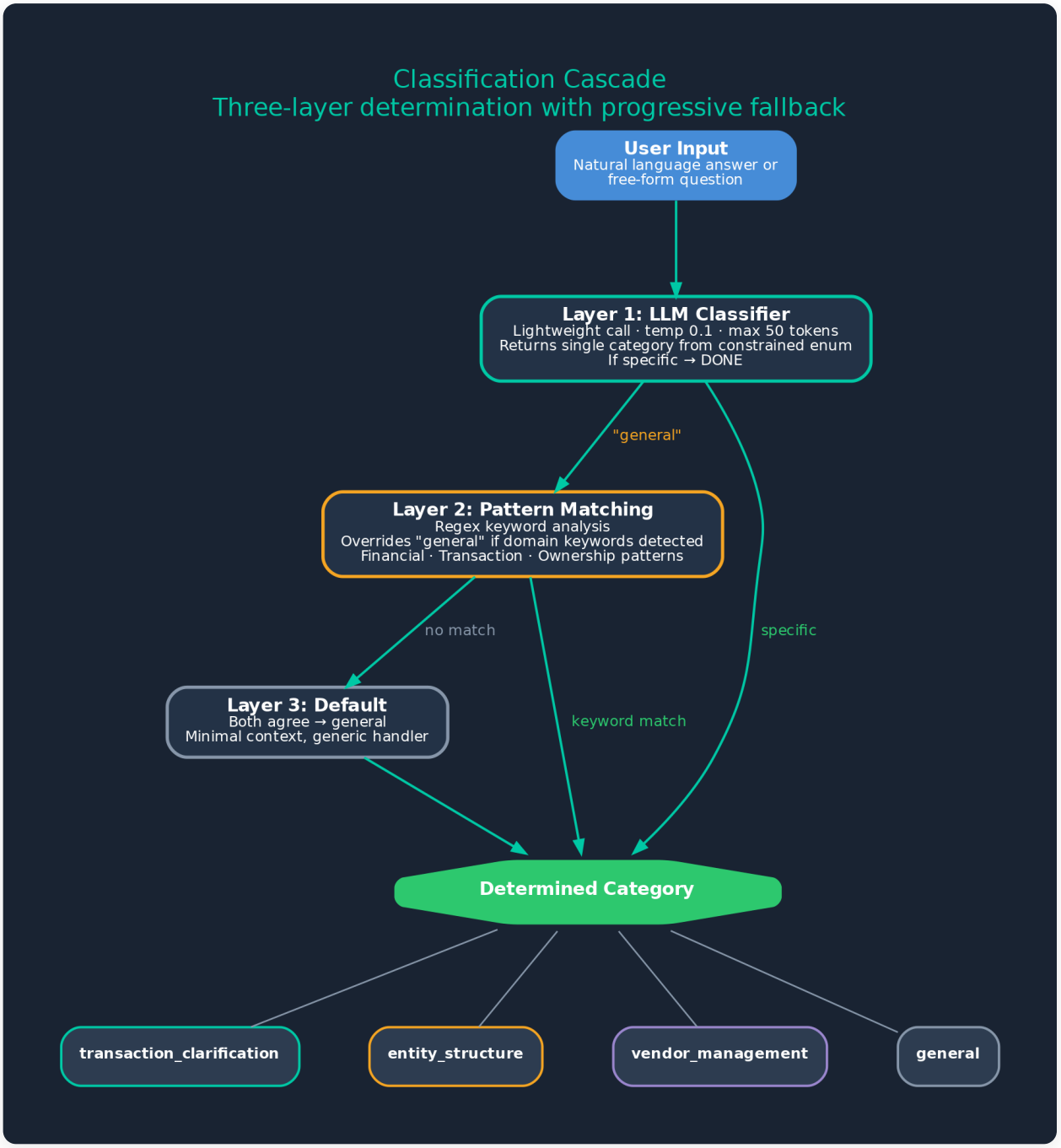

Figure 1 — Classification Cascade: Three-layer determination with progressive fallback

3. Category → Data Pipeline Mapping

Each determined category activates a targeted data pipeline — nothing more, nothing less. This is the key efficiency mechanism: instead of sending all available data to the LLM, only the pre-computed analytics relevant to the user's intent are fetched from the database.

> *→ Key insight: The LLM never sees raw data. It only narrates pre-computed, deterministic results. This ensures consistency and auditability.*

Category	Pre-computed Analytics	Contextual Data	Freshness Gate
transaction_clarification	Family-grouped questions, payee totals, QUESTION_PATTERNS matches	Outcome universe (OUTCOMES_BY_FAMILY), entity profile, payee history	Yes — re-bucket if stale (>1 hour)
entity_structure	Distribution equity metrics, partner records, ownership splits	Entity type, distribution patterns, SINGLE vs MULTI weights	Yes — re-query entity type on change
vendor_management	Vendor concentration, related-party links, payment frequency	Opaque entity classifications, vendor DB	Yes — re-classify vendors if new data
general	Summary aggregates only	Minimal context, last 4 messages	No (lightweight)

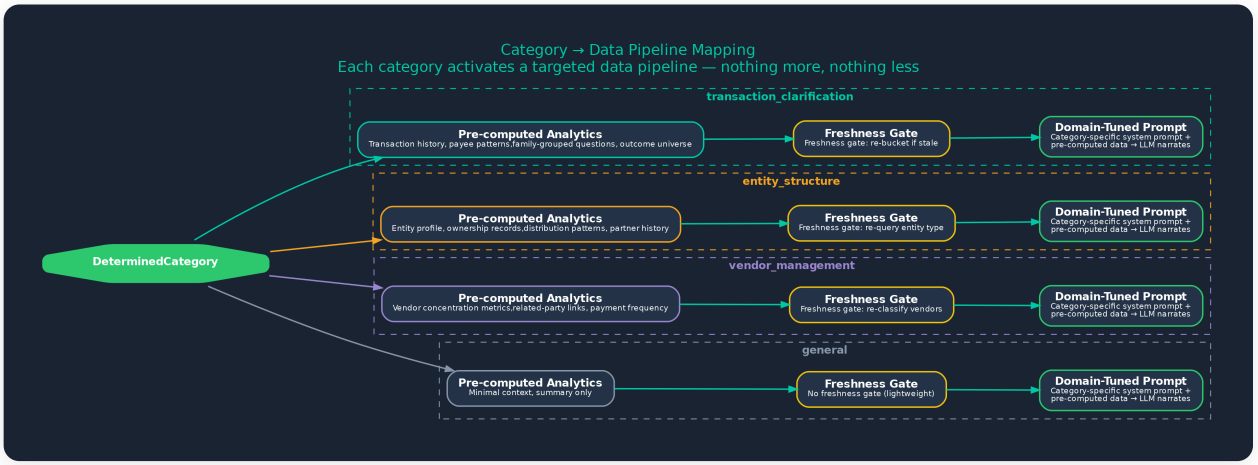

Figure 2 — Category → Data Pipeline Mapping: Each category activates targeted analytics

Domain-Tuned Prompts

After the data pipeline delivers pre-computed analytics, the system assembles a domain-tuned prompt. Each category maps to a specific system prompt, context loader, set of valid actions, and token budget. The LLM receives structured, auditable summaries — never raw transaction records.

Category	System Prompt	Valid Actions	Max Tokens
transaction_clarification	TRANSACTION_CLARIFICATION_PROMPT	whitelist, reclassify, confirm, ignore	800
entity_structure	ENTITY_STRUCTURE_PROMPT	define_partners, update_entity_type, confirm	600
vendor_management	VENDOR_MANAGEMENT_PROMPT	whitelist, blacklist, reclassify	600
general	GENERAL_PROMPT	acknowledge	1000

4. Freshness Guarantee Protocol

Pre-computed analytics can drift out of date. Without freshness guarantees, the AI narrates obsolete information. The Freshness Guarantee Protocol ensures the system always works with current data, not stale snapshots.

Five-Step Protocol

Step	Action	Detail
1. Check Age	Query latest computedAt timestamp from DB	Compare against staleness threshold
2. Stale?	Compare against threshold (e.g. 1 hour)	If fresh, proceed directly to serve
3. Trigger	Re-run analytics engine with current data	stage1Bucketing() + groupByFamily() + generateQuestions()
4. Poll	Check DB every 2s for new computedAt	Wait for fresh results to land
5. Serve	Return fresh data to LLM context	Domain-tuned prompt assembled from current analytics

Timeout Safety

If recomputation exceeds the maximum wait window, the system proceeds with the best available data and includes an analytics age note so the AI can transparently communicate data freshness to the user. The system never blocks indefinitely.

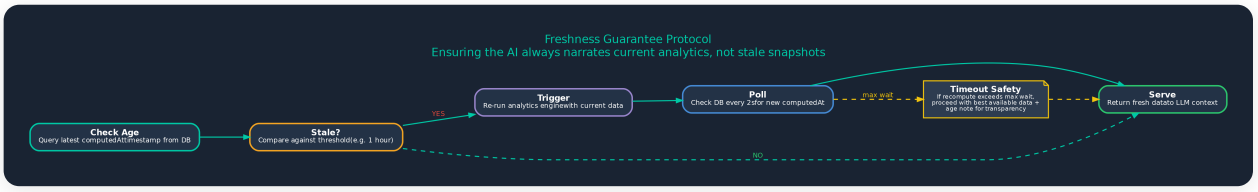

Figure 3 — Freshness Guarantee Protocol: Check Age → Stale? → Trigger → Poll → Serve

Staleness in the Question Pipeline

In the Lao Zi question system, staleness manifests when a user resolves a question (e.g., whitelists a payee) but the analytics haven't been re-run with the new user_bucket overrides. The freshness gate catches this: before generating new questions or assembling context for the next chat turn, the system checks whether the latest analytics include the most recent resolution. If not, it triggers a re-process.

5. The Adaptive Context Path

The architecture is not static. Each user interaction drives the system to become more precise through a self-reinforcing feedback loop. When a user answers a question, the resolution creates an event that triggers analytics recomputation. The next query benefits from fresher, richer analytics.

Step	What Happens	What Improves
1. User Answers	Natural language response to generated question	Raw signal enters the system
2. Classify Intent	LLM + Regex cascade determines domain	Classification improves with conversation context window
3. Select Pipeline	Category → targeted analytics fetch	Richer context from previous resolutions
4. Freshness Gate	If stale, recompute before proceeding	Ensures latest user overrides are reflected
5. Resolve → Side-Effect	LLM maps intent, code writes user_bucket	Transaction gets permanent owner classification
6. Logged → Recompute	Resolution event triggers analytics daemon	Next query gets richer: whitelisted payees, partner defs, vendor classifications

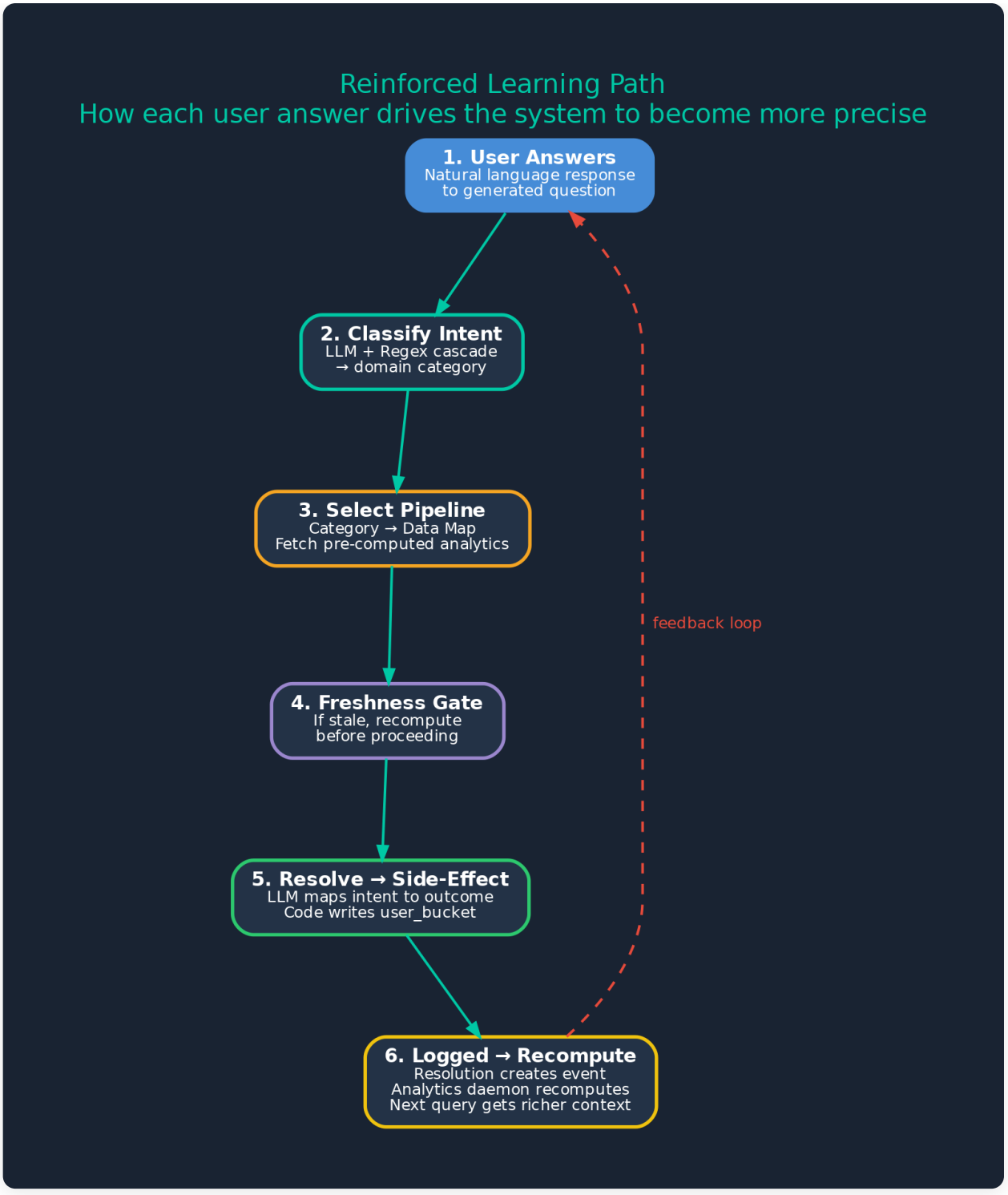

Figure 4 — The Adaptive Context Path: Each answer makes the next question smarter

What Accumulates

The system builds per-user context profiles that compound over time across four layers:

Context Layer	Content	Refresh Rule
Structure Awareness	Entity type, ownership, partner definitions	On user update (define_partners)
Historical Baselines	Computed metrics, running averages	On new data ingest
Relationship Mapping	Vendor → related-party links, payee classifications	On resolution (whitelist, reclassify)
Noise Filtering	Whitelisted payees, confirmed-clean patterns	Accumulates monotonically (never shrinks)

The system's contextual effectiveness compounds through richer structured context — the LLM becomes more relevant because its input data becomes more specific, not because the model itself learns.

6. End-to-End: From User Input to Resolution

The following diagram traces a concrete example through the full architecture. A user says "That's my plumber" in response to a question about payments to an opaque entity.

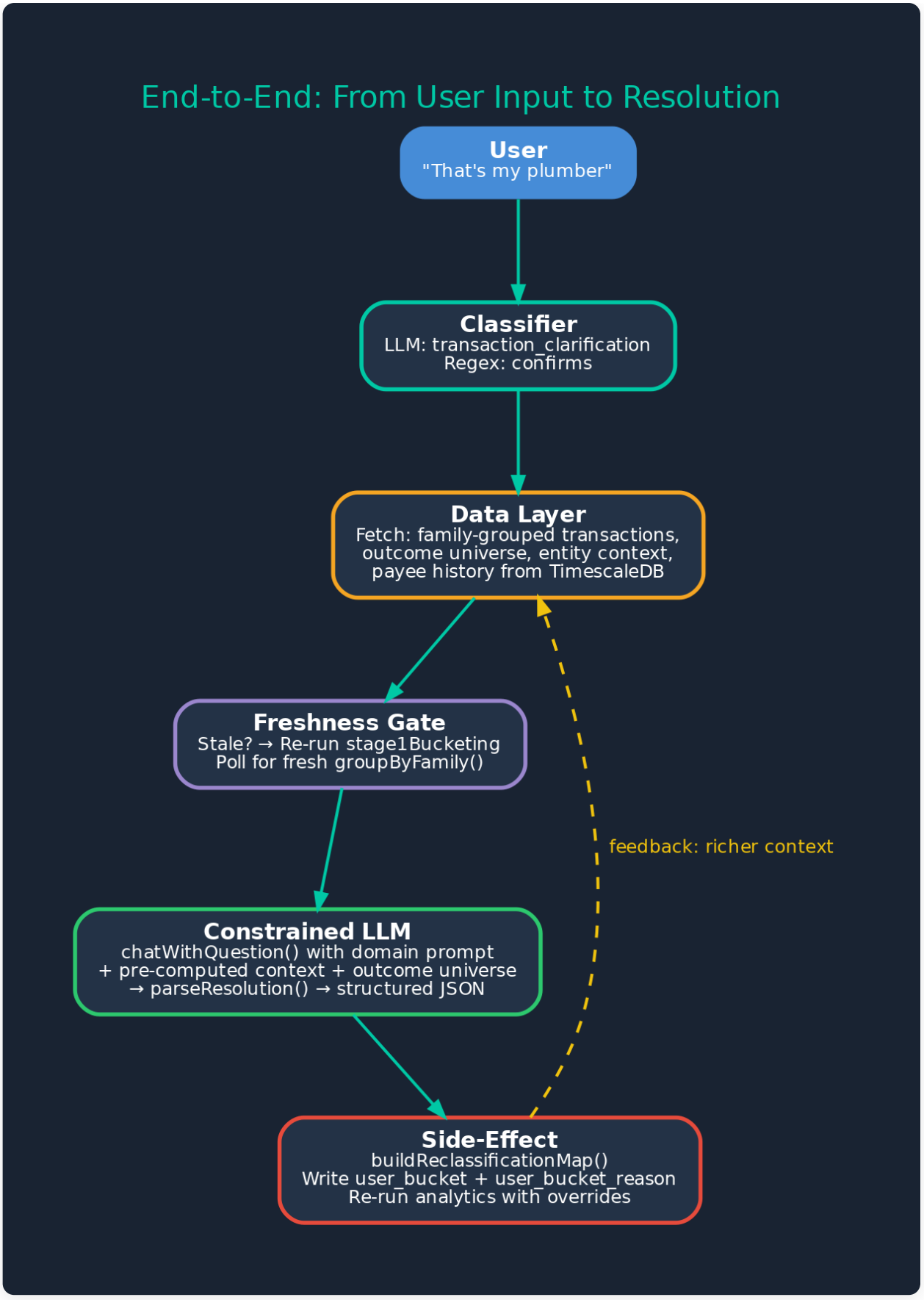

Figure 5 — End-to-End: From User Input to Resolution

Walkthrough

1. User says: "That's my plumber"

2. Classifier: LLM returns "transaction_clarification". Regex confirms ("vendor" keyword match). Done at Layer 1.

3. Data Layer: Fetch family-grouped transactions for this question, outcome universe (OUTCOMES_BY_FAMILY["opaque_entities"]), entity profile, payee history from TimescaleDB continuous aggregation views.

4. Freshness Gate: Check if analytics include the latest user_bucket overrides. If stale, re-run stage1Bucketing + groupByFamily. Poll until fresh.

5. Constrained LLM: chatWithQuestion() with domain-tuned system prompt + pre-computed context + outcome universe. The LLM recognizes "plumber" maps to "vendor providing services" via recognitionHints[]. parseResolution() extracts: {action: "whitelist", whitelistedPayees: ["ABC Plumbing LLC"], reasoning: "Regular plumber for office maintenance"}.

6. Side-Effect: buildReclassificationMap() writes user_bucket = IGNORE, user_bucket_reason = "Whitelisted by owner: Regular plumber". Analytics re-run. ABC Plumbing LLC is now permanently clean. Next question about this entity will not be generated.

7. Feedback: The resolution enriches the noise filtering context layer. The next analytics run skips ABC Plumbing LLC. Future questions about similar entities benefit from this richer context.

7. The Narration Principle

Why the LLM Never Touches Raw Data

In a traditional architecture, the LLM interprets raw data: it receives thousands of records, calculates averages, finds patterns, and produces answers. The results vary between calls (non-deterministic), the token budget is consumed by data rather than insight, and every number is unverifiable — the LLM is a black box with hallucination risk.

In this architecture, a deterministic engine pre-computes all analytics. The LLM receives structured, auditable summaries and narrates them. Same data → same answer (reproducible). Tokens are used for narration, not computation. Every number is traceable to a source. Zero hallucination risk for reported metrics.

> *✗ Traditional: LLM interprets raw data → inconsistent, unverifiable answers*

> ✓ *This architecture: LLM narrates results → consistent, auditable, trustworthy*

Enforcement in the Question Pipeline

The question pipeline enforces the narration principle at every stage: Stage 1 bucketing is pure regex + rules (no LLM). Stage 2 family classification and question generation use deterministic templates. Stage 3 chat resolution constrains the LLM to a closed outcome universe — it can only select from valid actions, never invent new ones. Side-effects are executed by deterministic code, never by the LLM.
