STANDARD OPERATING PROCEDURE: VIBECODING IMPLEMENTATION

USECASE 1: Prompt Similarity & Deduplication Service

Purpose

Identify duplicate and similar prompts in a voice AI system to eliminate redundancy and maintain prompt consistency across different layers (org, os, team, engine, directive).

Scope

Service operates on a fixed dataset of prompts stored in SQLite. It normalizes template variables, computes semantic embeddings, applies metadata-aware clustering, and exposes findings through a REST API.

Prerequisites

- Python 3.9+, FastAPI, SQLite3, sentence-transformers, scipy, numpy
- Initial dataset of prompts in JSON format with fields: prompt_id, category, layer, name, content
- Sample prompts available for validation

1: Data Foundation

Data Layer Implementation

The system begins with a PromptStore class that manages prompt persistence in SQLite. Each prompt record contains prompt_id (unique identifier following pattern [a-z0-9._]+), category, layer (one of: org, os, team, engine, directive), name, and content (minimum 10 characters). The database schema creates a single prompts table with these fields plus a created_at timestamp. Input validation rejects any prompt_id not matching the pattern or content under 10 characters, and ensures layer values conform to the enum. The PromptStore provides methods to load prompts from JSON files, save to database, retrieve individual or all prompts, and update existing records with validation at each step.

Note: The validation layer is critical here because bad data propagates through the entire system. If a prompt_id contains spaces or special characters, downstream processing fails silently. We validate early and reject aggressively.

2: Embeddings & Normalization

Variable Normalization

Before embedding, all template variables (like {{question_text}}) are replaced with semantic anchors following the pattern [VARIABLE_UPPERCASE_NAME]. This prevents embedding variance caused by different template syntax while preserving the semantic meaning of the prompt structure. A PromptNormalizer class extracts all {{variable_name}} patterns using regex, then replaces them. This ensures that "Ask the user {{question_text}}" and "Tell the user {{question_text}}" aren't considered different just because of template syntax.

Note: This is the critical insight for variable handling. Without normalization, similar prompts that use slightly different variable names get penalized in similarity scoring.

Embedding Generation

A lightweight sentence-transformers model (all-MiniLM-L6-v2) generates 384-dimensional embeddings from normalized content. The EmbeddingGenerator class handles single and batch embedding, caching the model on first use to avoid repeated downloads. For performance, embeddings use float32 (not float64) to reduce storage. The batch embedding target is 1000 prompts in under 30 seconds. A PromptEmbeddingStore persists normalized content and embeddings to SQLite as BLOB data, with methods to save, load, and batch-generate embeddings for an entire prompt set.

Note: The batch processing is essential for scaling. Individual embedding calls are fine for one prompt, but at 500+ prompts, you need efficient batch operations. The 30-second target keeps the system responsive.

3: Similarity & Clustering

Metadata-Aware Deduplication

The system doesn't use pure cosine similarity, which would incorrectly merge intentional design variants like os.style.warm and os.style.empathetic (both semantically similar but deliberately different). Instead, a three-tier system applies metadata filters:

Tier 1 (High Confidence, recommend merge): Same layer + same category + similarity >= 0.92 Tier 2 (Review needed): Same layer + different category + similarity >= 0.90
Tier 3 (Informational only): Different layer + similarity >= 0.88

The SimilarityComputer handles pairwise cosine similarity calculation (computing only the upper triangle to avoid redundancy). MetadataAwareMatcher filters each pair through the tier logic, returning a tier classification and confidence score. DuplicateClusterer uses hierarchical clustering with complete linkage to group similar prompts, cutting dendrograms at tier-specific thresholds. MergeRecommendationBuilder determines if a cluster should be merged, reviewed, or kept separate based on tier, and suggests a canonical prompt (typically the longest content) as the merge target.

Note: The tier system is non-negotiable. Without it, the service loses intentional design distinctions. Warm and empathetic are both high-quality responses; they're not duplicates.

4: API Layer

Endpoints

Four REST endpoints expose the functionality. POST /api/embeddings/generate regenerates embeddings for specified prompt_ids (or all if null), returning generated and updated counts with any errors. GET /api/prompts/{prompt_id}/similar returns similar prompts above a threshold, showing prompt_id, similarity_score, content_preview, layer, and category for each result. POST /api/search/semantic accepts a free-text query, generates an embedding, and returns semantically similar prompts by cosine distance. GET /api/analysis/duplicates returns clustered duplicates filtered by tier and threshold, showing cluster_id, tier, confidence, recommendation (MERGE/REVIEW/KEEP_SEPARATE), and variable summaries for each group.

Note: The /api/analysis/duplicates endpoint is the main deliverable. Users care about actionable deduplication recommendations, not raw similarity scores. The tier system makes this actionable.

5: Integration & Testing

Orchestration

The main.py initialize_service function idempotently sets up the entire service: loading prompts, normalizing and embedding them, computing similarities, and clustering by tiers. Running twice doesn't double-embed thanks to database checks. Integration tests validate end-to-end flows: loading the 12-sample dataset, verifying os.style prompts cluster as Tier 2 (not Tier 1), and confirming common.error prompts cluster as Tier 1. Performance benchmarks validate embedding speed (1000 prompts in under 30 seconds) and retrieval speed (similarity queries under 5ms).

Note: The idempotency guarantee is important for production stability. You want to be able to re-run initialization without side effects.

6: CLI & Visualization

Command-Line Interface

Four CLI commands support different workflows. analyze-duplicates lists duplicate clusters filtered by tier and threshold, outputting results as JSON. search-similar finds similar prompts to a given prompt_id, displaying results in a formatted table. semantic-search finds prompts similar to free-text queries. generate-embeddings regenerates embeddings for the dataset, showing progress. All commands accept standard options (db-path, output, threshold, tier).

Visualization

Three HTML visualizations are generated. cluster_visualization uses D3.js to show a force-directed graph where nodes are prompts (colored by tier) and edges represent similarity (thickness indicates score). similarity_matrix_heatmap displays a matrix heatmap (white = 0.0, red = 1.0 similarity) with dendrograms for hierarchical clustering. tier_breakdown_chart shows bar charts of cluster counts by tier with tooltips for details. An index.html links all visualizations together.

Note: The force-directed graph is most useful for understanding structure. You can immediately see which prompts cluster together and why.

USECASE 2: Call Outcome Prediction System (XGBoost + LSTM Ensemble)

Purpose

Predict the outcome of voice AI calls (completed, abandoned, transferred, or error) in real-time using an ensemble of tree-based and sequential models, with explanations for predictions.

Scope

Service trains on synthetic call data with realistic outcome correlation, engineers 28 features from event streams, trains XGBoost and LSTM models, and exposes predictions via REST API with explainability. The system monitors for prediction drift and triggers retraining when accuracy degrades.

Prerequisites

- Python 3.9+, XGBoost, TensorFlow/Keras, scikit-learn, scipy, numpy, pandas
- 500 synthetic call records generated with realistic event patterns
- Human validation data for calibration (50+ labeled examples)

1: Data Generation & Validation

Synthetic Dataset Creation

A dataset generator produces 500 call records with realistic outcome distributions: 60% completed, 25% abandoned, 10% transferred, 5% error. Each call contains 8-25 events (call_start, agent_speech, user_speech, silence, tool_call, call_end) with timestamps, durations, and word counts. The outcome labels are correlated with event patterns: abandoned calls have high silence ratios and low user words, completed calls have tools called and sustained engagement, transferred calls have high agent speech ratios, error calls end prematurely. This correlation is non-negotiable, if outcomes are random, the model learns nothing predictive.

Note: Realistic correlation is the foundation. Garbage in = garbage out. If the synthetic data has outcomes uncorrelated with events, the trained model is useless.

Data Validation

A validation pipeline checks schema compliance (all required fields present with correct types), event sequence validity (timestamps strictly increasing, event types sensible), and

outcome correlation strength (do abandoned calls actually have high silence?). The validator flags missing fields, timestamp violations, and outcome correlation failures. It computes statistics per outcome class: completed calls average 0.30 silence ratio, abandoned calls average 0.55. If these divergences don't exist, the correlation rules failed and data generation needs adjustment.

Note: Validation is the gatekeeper. Bad synthetic data corrupts the model before training even starts. This step prevents that.

Phase 2: Feature Engineering

Temporal Features

From raw event streams, 28 features are computed. Timing features include total_duration_sec, time_to_first_user_speech_sec, time_to_first_tool_call_sec, avg_response_latency_sec, agent_response_latency_p75, and avg_silence_duration_sec. These capture how quickly the call progresses and whether the agent responds promptly. Speech dynamics features include agent_talk_ratio, user_talk_ratio, silence_ratio, silence_count, user_speech_trend (slope of user speech duration over time), speech_entropy (variance in turn lengths), and agent_flexibility (variance in agent speech durations). These capture the rhythm and balance of the conversation.

Note: The trends (user_speech_trend, user_engagement_slope) are critical for detecting call deterioration. If user speech is declining over time, the call may be heading toward abandonment.

Engagement & Progress Features

Engagement metrics include turn_count (speaker alternations), words_per_turn_user, words_per_turn_agent, user_engagement_slope (derivative of cumulative user words), interruption_count (agent jumping in <0.5s after user), and cumulative_user_words. Progress signals include tools_called_count, tools_per_minute, and survey_completion_rate. Contextual features preserve metadata: agent_id, org_id, call_purpose, time_of_day, day_of_week.

Note: The interruption_count is subtle but predictive. Frequent interruptions correlate with abandoned calls; smooth turn-taking correlates with completion.

Feature Validation

After computing all 28 features on 500 records, validation checks completeness (exactly 500 records with exactly 28 features each), checks for nulls (impute with 0 if any), and computes statistics per outcome class. For completed calls, average silence_ratio should be ~0.25; for abandoned, ~0.55. If distributions don't separate by outcome, feature engineering failed and needs redesign.

Note: The distribution separation is a hard signal of feature quality. If you compute 28 features and they don't separate outcomes, you're missing something important in your feature definitions.

Phase 3: XGBoost Model

Training

XGBoost trains on 400 records (80% train, 10% val, 10% test) with stratified splits to maintain outcome distribution. Five-fold cross-validation on the training set estimates generalization error. Fixed hyperparameters (max_depth=5, learning_rate=0.1, n_estimators=100, subsample=0.8, colsample_bytree=0.8) prevent overfitting without expensive tuning. Early stopping halts training if validation loss stops improving for 10 rounds. The model trains on normalized numeric features and label-encoded categorical features.

Note: Fixed hyperparameters are intentional. With only 500 samples, hyperparameter tuning risks overfitting. The chosen values are conservative and proven for this domain.

Evaluation

On the 50 test records, the model reports accuracy, precision, recall, and F1 (both macro-averaged across classes and per-class). A 4x4 confusion matrix shows which outcomes are confused (e.g., completed vs. transferred). The target is test_accuracy > 0.80. Feature importance (gain-based) ranks all 28 features by predictive power, with silence_ratio typically ranked first (high silence strongly predicts abandonment).

Note: The per-class metrics matter because classes are imbalanced. Error outcome (5% of data) is much rarer, so recall on error is naturally lower. Don't be alarmed if error F1 is 0.4 while completed F1 is 0.87.

Explainability (SHAP)

SHAP values decompose each prediction into feature contributions. Global SHAP importance (mean |SHAP value| per feature) provides a robust alternative to feature importance. Per-class SHAP values show which features push predictions toward each outcome: high silence_ratio has negative SHAP for completion (pushes away), positive SHAP for abandonment (pushes toward). Sample explanations for individual predictions show the top 5 supporting factors and top 3 opposing factors with concrete feature values and SHAP contributions.

Note: SHAP is critical for production deployment. When the model predicts an unexpected outcome, SHAP shows why (e.g., "high silence_ratio and low user_words push toward abandonment"). This explanation is what users trust.

Phase 4: LSTM Model (Temporal Sequences)

Partial Sequence Generation

From each of the 500 calls, five partial sequences are generated (at 20%, 40%, 60%, 80%, 100% completion). Each partial sequence is truncated to that percentage of the call duration, with the call_end event removed (call is still ongoing). The outcome label is inherited from the original call (all five partials of an abandoned call are labeled abandoned). All 28 features

are recomputed on the truncated event stream. This creates 2500 training examples where the label is "what happens by the end of this call" but the features are measured partway through.

Note: This is the key innovation for LSTM. Predicting from partial sequences validates that the model learns temporal patterns, not just static features. If accuracy jumps from 60% at 20% completion to 85% at 100%, the model is genuinely learning temporal dynamics.

LSTM Training

Events are encoded as integers (call_start=0, agent_speech=1, etc.) and each event becomes a vector [event_type, duration_ms, words]. Sequences are padded to length 50 (typical call has 8-25 events). The LSTM architecture has embedding layer (input_dim=8, output_dim=16), LSTM layer (units=32, dropout=0.3), dense layer (units=64), and softmax output (units=4). Training on 2000 sequences (80%) with validation on 200 uses categorical_crossentropy loss and Adam optimizer. Early stopping halts if validation loss doesn't improve for 10 epochs.

Note: The embedding layer is critical. Raw event type integers (0-7) are meaningless to the LSTM. The embedding layer learns a dense representation of each event type, allowing the LSTM to reason about event similarity.

Evaluation

On 500 test sequences, the model reports test_accuracy, per-class metrics, and crucially, accuracy_by_completion_percent (accuracy at 20%, 40%, 60%, 80%, 100%). The target is monotonic improvement: accuracy at 20% < 40% < 60% < 80% < 100%. This validates that more data improves predictions. If accuracy at 80% is lower than at 60%, something is wrong (perhaps data distribution shifts at longer calls).

Note: Monotonic improvement is not guaranteed and signals model quality. If the trend is flat or declining, the LSTM isn't learning temporal patterns; it's memorizing.

Phase 5: Ensemble & API

Ensemble Strategy

The ensemble combines XGBoost (higher accuracy, better on full-data predictions) and LSTM (captures temporal dynamics, useful for partial calls). Weighted averaging: P_ensemble(class) = 0.65 * P_xgb(class) + 0.35 * P_lstm(class). At different call stages, the weighting adapts: 0-10s use XGBoost only (LSTM unreliable), 20-60% use both separately, 60-100% use ensemble. Probability calibration applies temperature scaling to match predicted confidence to actual accuracy.

Note: The weights (0.65 / 0.35) are based on accuracy difference. Since XGBoost is ~5% more accurate, it gets more weight. This is a principled choice, not arbitrary.

Prediction Endpoint

POST /api/predict accepts call_id, events_so_far (event array), and metadata (agent_id, org_id, etc.). The pipeline computes 28 features from events, runs through selected model(s), applies calibration, computes SHAP explanations, and returns predicted_outcome, confidence, risk_score, and top_factors (SHAP contributions). Latency target is <100ms (XGBoost <10ms, LSTM <50ms, SHAP <50ms).

Note: The risk_score (1 - confidence) is a useful companion to the prediction. High confidence is good, but you still want to know the failure risk explicitly.

Training Endpoint

POST /api/model/train accepts data_file and model_type (xgboost, lstm, or ensemble). The service regenerates features, retrains the selected model(s), evaluates on a held-out test set, and returns the trained model with metrics and a model_uri for future use.

Phase 6: Monitoring & Retraining

Drift Detection

Daily, accuracy is computed on calls with ground truth labels (outcomes confirmed after call ends). A 7-day and 30-day rolling accuracy tracks trends. Weekly, 10 recent calls are sampled, rated by humans (1 rater, simple accuracy check), and compared to model predictions. Spearman correlation measures agreement; target > 0.75. If correlation drops below 0.75, an alert recommends retuning. If it drops below 0.65, new evaluations halt and revert to human review.

Note: The correlation check is more informative than absolute accuracy. Absolute accuracy can drop due to data distribution shift (e.g., more abandoned calls), but correlation tells you if the model is still rank-ordered correctly.

Root Cause Analysis

When drift is detected, the system identifies which calls caused it. If divergences are systematic (model always underscores empathy-related outcomes), the root cause is clear (model bias). If divergences are concentrated in a subset (e.g., night-time calls), the root cause is context-specific data shift. This analysis guides retuning.

Note: Without root cause analysis, retuning is blind. With it, you can make targeted improvements.

Retraining Trigger

Retrain if accuracy drops below 0.79 (5% drop from baseline 0.84), feature drift detected in >3 dimensions, outcome distribution changes >15%, or latency exceeds 150ms for >5% of requests. When triggered, new training data is collected (last month's calls with confirmed outcomes), features are recomputed, and both XGBoost and LSTM are retrained. New models are validated on a held-out test set; only deploy if accuracy > 0.80 and Spearman correlation > 0.75.

Note: The multiple triggers ensure robustness. A single trigger (accuracy drop) might be noise; multiple triggers (accuracy drop + feature drift + latency increase) indicates a systemic problem needing retraining.

USECASE 3: LLM Response Quality Evaluator

Purpose

Evaluate voice AI responses on dimensions like task completion, empathy, conciseness, naturalness, safety, and clarity. Provide actionable suggestions for improvement and identify when responses need human review.

Scope

Service uses an LLM judge to score responses on six dimensions, applies metadata-aware weighting based on context, calibrates scores against human raters, detects drift, and exposes evaluation via REST API.

Prerequisites

- Python 3.9+, FastAPI, Claude Opus (or GPT-4-turbo) API access, Pydantic v2
- Initial sample evaluation cases (3-5 labeled examples from different contexts)
- Human raters available for calibration (~20-30 cases)

Phase 1: Schema & Dimension Design

Data Contracts

The system defines strict Pydantic schemas for all inputs and outputs. An EvaluationRequest includes response (the text to evaluate), conversation_history (prior turns), directive (what the response was supposed to accomplish), and metadata (agent_type, prompt_version, language, context_type). An EvaluationResult includes dimension_scores (task_completion, empathy, conciseness, naturalness, safety, clarity, each with score 1-10, reasoning, and confidence 0-1), overall_score, flags array (auto-flagged issues), and suggestions array (up to 3 actionable improvements). Validation rejects invalid scores (<1 or >10), reasoning under 15 characters, confidence outside [0, 1].

Note: The schema is the contract between system components. If the judge produces output that doesn't match the schema, the entire system fails. Strict validation catches this early.

Dimension Definitions

Six dimensions define response quality. Task Completion (did it accomplish the directive?) uses a rubric: 9-10 is fully accomplished, 5-6 is partially accomplished, 1-2 is completely

failed. Empathy (appropriate emotional attunement?) uses a rubric: 9-10 shows the user feels heard, 5-6 is neutral professional, 1-2 is dismissive or shaming. Conciseness (efficient without being curt?) uses a rubric: 9-10 is every word serves purpose, 5-6 has acceptable filler, 1-2 rambles. Naturalness (sounds like real conversation?) uses a rubric: 9-10 flows naturally, 5-6 is slightly formal, 1-2 sounds scripted. Safety (no harmful, biased, or inappropriate content?) uses a rubric: 10 is perfect safety, 6-7 is acceptable with low-risk issues, 1-3 has significant risk. Clarity (easy to understand?) uses a rubric: 9-10 is crystal clear, 5-6 mostly clear with one confusing element, 1-2 incomprehensible.

The dimensions are not equally weighted. For verification contexts (collect data), weights are: task_completion 40%, safety 30%, conciseness 20%, clarity 10%. For screening contexts (sensitive topics), weights are: empathy 35%, task_completion 30%, naturalness 20%, safety 15%. These weights are context-aware and empirically validated.

Note: The weights are the system's "values." If you weight task_completion high, you prioritize getting information; if you weight empathy high, you prioritize user comfort. The context determines the right balance.

Phase 2: Judge Prompt & Calibration

Judge Prompt Design

A single comprehensive prompt acts as the evaluator. The system role explains: "You are an expert evaluator of healthcare voice AI responses." The prompt receives conversation context (prior turns), the directive (what the response was supposed to do), the user input (what prompted the response), the response to evaluate, and the metadata (context_type, agent_type). For each dimension, the prompt states the definition, shows a rubric (1-10 scale with clear boundaries), and requires explicit reasoning: observed behavior (what did the response actually do?), rubric fit (which band?), red flags (if any), and evidence (quote relevant part).

The judge outputs JSON: {"dimension_name": {"score": 7, "reasoning": "...", "confidence": 0.85}, ...}. Confidence is 0.1-1.0 (never 0), with high confidence (0.8-1.0) for clear-cut cases, medium (0.5-0.8) for some ambiguity, low (0.1-0.5) for borderline cases. The prompt is instructed to be deterministic: "use fixed reference points; for borderline cases, default to the lower score."

Note: The determinism instruction is critical. Without it, the LLM introduces randomness (temperature effects, different phrasings of ambiguity). With it, you get reproducible scores.

Calibration Process

From 3 initial sample cases, the calibration set is expanded to 20-30 cases by generating variants. For each sample, create a slight improvement variant (score should increase by 0.5-1.5), a degradation variant (score should decrease by 0.5-1.5), and a minimal change variant (score ±0.3). This creates realistic edge cases. Each case is rated by 3 independent human raters (majority vote on dimension scores). Compute inter-rater agreement using Krippendorff's alpha; target > 0.70 per dimension.

Run the judge prompt on all 20-30 cases. Compare LLM scores to human consensus using Spearman rank correlation coefficient (SCC). Target SCC > 0.80 per dimension. If any dimension < 0.80, analyze divergences: Is the LLM systematically underscoring? Is it confusing dimensions? Does it depend on context (e.g., empathy scores diverge only in verification contexts)? Modify the judge prompt to address divergences (e.g., add specificity: "Look for explicit acknowledgment like 'I understand'"). Retest on calibration set; only deploy if SCC > 0.80 across all dimensions.

Note: The SCC is the gatekeeper. If your calibration SCC is 0.75, you know in production that dimensions will correlate 0.75 with human judgment. This is acceptable but not great. Below 0.70, the judge is too unreliable.

Phase 3: API Endpoints

/api/evaluate

POST /api/evaluate accepts an EvaluationRequest and returns an EvaluationResult. The pipeline validates input, injects context-aware dimension weights, calls the judge prompt (using Claude Opus for accuracy), parses JSON response, validates against schema, computes overall_score = Σ(dimension_score * weight), detects auto-flags (any dimension < 4, safety < 8, confidence < 0.6, reasoning < 20 words), and generates suggestions if overall_score < 8. Suggestions are actionable (e.g., "Add empathetic acknowledgment: Replace 'noted' with 'I hear you, that sounds difficult'") and prioritized by impact. Latency target is <5 seconds (including LLM call). Caching stores identical requests for 24 hours; if LLM times out, return cached result (if available).

Note: The <5 second target is aggressive but necessary for real-time dashboards. This drives the caching strategy; without caching, you'd often miss the target.

/api/compare

POST /api/compare accepts the same context and two different responses (response_a, response_b). The pipeline evaluates both, computes per-dimension winners (difference > 1 point = winner, else tie), and computes overall winner using weighted difference (favoring context-appropriate dimensions). The output includes winner (a/b/tie), per-dimension comparison, and recommendation (actionable suggestion on which variant to use). Confidence_in_winner reflects clarity of the win; < 0.6 recommends human review. Bias detection alerts if either response wins >65% of time (indicates systematic bias toward that position).

Note: The bias detection is subtle but important. If response_a always wins because it's evaluated first, the system is biased. Detection prevents this.

/api/evaluate/batch

POST /api/evaluate/batch accepts up to 500 evaluation requests and returns individual scores plus aggregate statistics (mean, std, min, p25, median, p75, max per dimension; per agent_id, per context_type, per prompt_version). Cost optimization: use GPT-4o-mini (not Opus) for batch, expecting 50% cache hit rate and total cost < $0.03 per response. Anomaly

detection identifies outliers (score > 3 std from mean) and low-confidence evals (< 0.6). Output includes individual_scores, aggregate_stats, anomalies[], metadata (total_evaluated, cache_hit_rate, cost, latency). Filtering allows slicing by agent_id, context_type, date_range.

Note: The cost target ($0.03) is tight but achievable with caching. Without optimization, XGBoost-based evaluation (no caching) costs ~$0.05 per response.

/api/improve

POST /api/improve accepts a response and context, identifies lowest-scoring dimensions, generates improved versions (2-3 variants with different optimization strategies), predicts score improvement, and returns best variant with change tracking. Changes are categorized (lexical, structural, additive, subtractive) and the top 3 are reported. Confidence_in_improvement = (predicted_score - original_score) / (improvement_target); < 50% recommends human review. Supports iterative improvement (use improved response as input to another /api/improve call).

Note: The improvement endpoint is a learning tool. It shows how to incrementally enhance responses, which is valuable for agent training.

Phase 4: Monitoring & Drift Detection

Weekly Drift Checks

Every week, sample 10 recent responses (stratified by agent_type, context_type, score range). Have 1-2 humans rate these samples on the same dimensions. Compute Spearman rank correlation (SCC) between LLM scores and human scores. If SCC < 0.75 for any dimension, alert and recommend retuning. If SCC < 0.65, halt new evaluations and revert to manual review.

Root Cause Analysis

When drift detected, identify divergent cases (LLM score differs >1 from human). Categorize: systematic underscoring (LLM always -1 on dimension X), oversensitivity (LLM penalizes feature Y too much), context-specific confusion (SCC drops only for context_type=verification). Propose prompt modification (add specificity, adjust rubric, correct weights). Test on calibration set; only deploy if SCC improves and stays > 0.78.

Note: Root cause analysis prevents shooting in the dark. Without it, you modify the prompt randomly hoping for improvement. With it, you make targeted fixes.

Confidence-Based Escalation

If retuning fails to restore SCC > 0.75, enable confidence-based fallback: auto-escalate responses with LLM confidence < 0.6 to human review. This forces higher-confidence evaluations only, trading volume for quality. Expected outcome: 2-3 weeks in, <5% of evaluations are low-confidence (escalated).

Note: This is a graceful degradation strategy. If the judge starts failing, the system self-heals by asking humans for help on ambiguous cases.

Phase 5: Implementation & Validation

Phase-by-Phase Testing

Phase 1 (schemas): Validate input validation works on bad data (empty string, null values, out-of-range scores). Phase 2 (judge prompt): Test on 3 sample cases, verify output structure and SCC > 0.80 on calibration set. Phase 3 (APIs): Mock test each endpoint (evaluate, compare, batch, improve) with sample data, verify response formats. Phase 4 (monitoring): Simulate drift by modifying judge prompt, verify drift detection triggers. Phase 5 (integration): End-to-end test: upload 50 real responses, evaluate, compare variants, monitor drift. Full integration test: upload 500 responses, run batch evaluation, verify cost target (<$25 total cost for 500 evaluations) and latency (average <5s per individual request, batch under 10 min for 500).

Note: Testing at each phase prevents late-stage surprises. If you skip Phase 2 (judge calibration), you deploy an uncalibrated judge that produces meaningless scores.

Phase 6: Dashboards & Observability

Monitoring Dashboards

System Health Dashboard shows request volume, success rate, error rate, p50/p99 latency, alerts on error rate > 5%, latency > 4s, cache hit rate < 25%. Evaluation Quality Dashboard shows mean score and distribution per dimension, per-agent statistics, per-prompt-version comparisons, weekly SCC (drift metric). Cost & Efficiency Dashboard shows cost per evaluation, cache hit rate, LLM calls per second, cost trends. Flagged Responses Dashboard shows flag volume, flag types, flag rate by context_type, recent flagged responses. A/B Testing Dashboard tracks ongoing prompt variant experiments, showing overall winner, per-dimension comparison, and experiment status.

Note: The SCC dashboard is the most important. When SCC drops, you know there's a problem. Without visibility, you don't notice drift until responses become obviously bad.