LLM MOVIES AND SHOPPING RECOMMENDATION SYSTEM

TECHNICAL DOCUMENTATION

Prepared for: IdeaNova Technologies
Document Version: 1.0
Date: December 16, 2025

Table of Contents

LLM MOVIES AND SHOPPING RECOMMENDATION SYSTEM 1

TECHNICAL DOCUMENTATION 1

EXECUTIVE SUMMARY 1

1. SYSTEM OVERVIEW 1

1.1 Core Capabilities 1

1.2 Business Value 1

2. SYSTEM ARCHITECTURE 1

2.1 Architectural Layers 1

2.2 Architecture Diagrams 2

2.3 Technology Stack 3

3. LLM ENGINE ARCHITECTURE 4

3.1 Common Infrastructure (`internal/engine/llm/common/`) 4

3.2 Movie Recommendation Engine (`internal/engine/llm/movies/`) 4

3.3 Shopping Recommendation Engine (`internal/engine/llm/shopping/`) 8

3.4 Combined LLM Server (`internal/engine/llm/llm.go`) 10

4. ARCHITECTURAL EVOLUTION AND TESTING 10

4.1 Development Phases 10

4.2 Concurrency Decision Analysis 11

4.3 Prompt Evolution Summary 12

4.4 Test Environment, Hardware Requirements, and Evaluation Methodology 13

4.5 Prompt Analysis 15

5. ENHANCED SCORING AND ADJUDICATION 15

5.1 Scoring System Evolution 15

5.2 Detailed Metric Definitions 16

5.3 GPT-5 Adjudication System 17

5.4 Rejected Metrics (Design Decisions) 19

6. SYSTEM DESIGN PRINCIPLES 19

6.1 Core Architecture Principles 19

6.2 Production Readiness 20

7. INTEGRATION GUIDE 20

7.1 API Endpoints 20

7.2 Request/Response Formats 21

7.3 Configuration 22

7.4 API Documentation 22

8. APPENDICES 22

8.1 Glossary 22

EXECUTIVE SUMMARY

This AI Recommendation Engine is a production-grade backend system that delivers personalized movie and shopping recommendations using multiple Large Language Models (LLMs). The system processes user profiles, watch history, and product catalogs to generate tailored recommendations in a consistent, schema-validated JSON format.

Built with Go and designed for React frontend integration, the system features parallel LLM execution, comprehensive performance tracking, advanced scoring mechanisms, and optional GPT-5 adjudication for model selection. The architecture ensures reliability, scalability, and consistent output quality for enterprise deployment.

1. SYSTEM OVERVIEW

1.1 Core Capabilities

- **Multi-Model Execution**: Parallel processing of multiple LLM models with timeout controls
- **Dual Domain Support**: Separate engines for movie and shopping recommendations
- **Advanced Scoring**: Multi-dimensional evaluation framework combining quality, performance, and efficiency metrics
- **Schema Enforcement**: Strict JSON validation ensuring consistent, parseable outputs
- **Intelligent Grounding**: Catalog validation and ID verification with fallback recovery
- **Performance Optimization**: Caching, batch processing, and efficient resource utilization
- **Optional AI Adjudication**: GPT-5 integration for semantic evaluation and final model selection

1.2 Business Value

- **Personalization Quality**: Leverages user watch history and behavioral signals for relevant recommendations
- **Model Comparison**: Comprehensive metrics enable data-driven model selection
- **Production Reliability**: Timeout management, error recovery, and consistent schema compliance
- **Scalability**: Stateless architecture supporting horizontal scaling
- **Auditability**: Complete logging of prompts, responses, scores, and decisions

2. SYSTEM ARCHITECTURE

2.1 Architectural Layers

CLIENT LAYER (React Frontend)

- User interfaces for submitting recommendation requests

- Real-time display of results, analytics, and model comparisons
- Batch request submission and result visualization
- Watch history and user profile management

API LAYER (Go Backend)

- RESTful endpoints for LLM execution and result retrieval
- Separate handler sets for movie and shopping domains
- Active run management with cancellation support
- CSV export functionality for analysis

LLM ENGINE LAYER

- Schema-driven prompt construction
- Parallel model execution with concurrency controls
- JSON validation and response parsing
- Grounding, scoring, and ranking algorithms
- Complete run lifecycle orchestration

DATA LAYER

- SQLite database for catalog storage (movies/products)
- Run logs, model results, and performance metrics
- User profiles, watch history, and preference data
- Efficient indexing for analytics queries

EXPORT & ANALYSIS LAYER

- CSV report generation with complete run data
- User information and recommendation exports
- Performance metrics for research and debugging

2.2 Architecture Diagrams

Detailed architectural diagrams are provided to supplement the layered overview above. Each diagram focuses on a specific aspect of system structure or execution flow.

System Architecture Diagram

This diagram presents the end-to-end structure of the AI Recommendation Engine, illustrating interactions between the client, API services, LLM engine, data layer, and export components. It highlights domain separation, shared services, and data persistence boundaries.

- **Repository path:** `sd06_personalization_1/Documentation/Architecture/system_architecture.pdf`
- **Direct link:**

System_Architecture.pdf

LLM Execution Workflow Diagram

This diagram details the lifecycle of an LLM recommendation run, including request intake, prompt construction, parallel model execution, validation, scoring, ranking, and result persistence. It emphasizes concurrency handling and execution orchestration.

- **Repository path:** `sd06_personalization_1/Documentation/Architecture/llm_execution_workflow.pdf`
- **Direct link:** LLM_Execution_Workflow.pdf

Database Schema Diagram

This diagram illustrates the logical data model supporting the recommendation engine, including core entities, relationships, and constraints used for catalog storage, run tracking, user profiles, and result persistence.

- **Repository path:** `sd06_personalization_1/Documentation/Architecture/schema_diagram.pdf`
- **Direct link:** Schema_Diagram.pdf

2.3 Technology Stack

Backend

- Go (Core implementation)
- chi Router (HTTP routing and middleware)
- SQLite (Embedded database)
- Context management (Timeout and cancellation handling)
- JSON Schema validation

Frontend

- React.js (User interface)
- Fetch API (Backend communication)
- Modern UI libraries

Development & Operations

- Swagger/OpenAPI (API documentation)
- Git (Version control)

3. LLM ENGINE ARCHITECTURE

3.1 Common Infrastructure (`internal/engine/llm/common/`)

CONFIGURATION MANAGEMENT (`config.go`)

Defines critical system parameters:

- **Allowed LLM Models**: Validated model allowlist preventing unsupported model execution
- **Candidate Pool Size Cap**: Maximum catalog items per prompt (ensures token limit compliance)
- **Timeout Configuration**:
  - Per-model timeout (prevents stalled executions)
  - Overall pipeline timeout (guarantees system responsiveness)
- **Prompt Version Identifier**: Enables reproducibility and historical comparison
- **OpenAI Adjudication Settings**: Optional GPT-5 evaluation configuration
- **Token Limits**: Input/output constraints for safe prompt sizing

DATABASE LAYER (`database.go`)

Core persistence functionality:

- **Schema Creation**: Initializes tables for users, runs, metrics, and indexes
- **Candidate Selection**: Optimized queries retrieving catalog entries with metadata (genres, audio, price, images)
- **Result Storage**: Persists recommendations, metrics, scoring breakdowns, and prompt information
- **Constraint Enforcement**: Validates ranking, schema correctness, and caching consistency

UTILITY FUNCTIONS (`utils.go`)

Shared helper components:

- Watch history representation
- Recommendation output formats
- Model scoring structures
- JSON encoding utilities
- Active run registry for cancellation and monitoring

3.2 Movie Recommendation Engine (`internal/engine/llm/movies/`)

SERVER INTEGRATION (`movies_server.go`)

Entry point connecting the movie engine to the HTTP API surface. Registers all movie-related routes and handlers for external client consumption.

RUN ORCHESTRATION (`movies_run.go`)

Phase 1: Input Validation and Canonicalization

- Produces deterministic input representation for caching
- Reduces redundant LLM calls by approximately 40%

Phase 2: Model Selection Strategy

- Query parameter: `?model=llama3.2`
- Query CSV: `?models=a,b,c`
- JSON body: `{"models": ["deepseek", "mistral"]}`
- Default: All configured models
- Validation against allowlist ensures authorized models only

Phase 3: Candidate Pre-filtering

- Reduces 1000+ initial candidates to 150-300 optimized selections
- Filtering criteria:
    - Age appropriateness
    - Watch history exclusion
    - Genre similarity thresholds
    - Language and content-type preferences
    - Content quality heuristics

Phase 4: Prompt Engineering

- System role definition with strict JSON output rules
- User profile and watch history interpretation
- Allowed ID list inclusion
- Personalization constraints
- Typical length: 1200-1800 tokens

Phase 5: Concurrent Model Execution

- All models run in parallel
- Per-model timeout (default: 30 seconds)
- Overall pipeline timeout (default: 60 seconds)
- Individual model failures do not abort pipeline

Phase 6: Enhanced Scoring System

Component	Weight	Purpose
Quality Score	35%	Relevance, personalization, diversity, novelty
Catalog Score	10%	Catalog validity assurance
Latency Score	35%	Performance measurement
Token Efficiency	10%	Cost-efficient output
Redundancy Score	10%	Anti-duplication

Phase 7: GPT-5 Adjudication (Optional)

- Evaluates personalization, content quality, technical excellence, business value
- Produces ranked model list, curated recommendations, evaluation metadata
- Activated only when OpenAI API key is provided

Phase 8: Result Persistence and Response

- Writes to `movie_recommendations` and `movie_metrics` tables
- Updates user profile and watch history
- Returns structured output with metadata, per-model results, scores, rankings
- Caches successful runs for future identical requests

PROMPT CONSTRUCTION (`movies_processing.go`)

Prompt Strategy

The prompt establishes system identity and consequences:

```
"You are a STRICT backend movie recommendation component.
Your output will be parsed by another program.
ANY deviation from the required JSON format will BREAK the system."
```

Key elements:

- Explicit formatting constraints with clear failure consequences
- JSON schema enforcement (reduces formatting errors by ~70%)
- Watch history interpretation with quantitative behavioral signals:
  - ≥80% watched → strongly liked/engaged
  - 30-79% watched → somewhat interested
  - <30% watched → abandoned/disliked
- Mandatory personalization rules enforcing prioritization, exclusion, diversity

Output Processing Pipeline

1. **Loose JSON Parsing**: Multiple recovery strategies (direct unmarshal, substring extraction, array fallback, text salvage)

2. **Grounding and Validation**: ID validation, duplicate elimination, reasoning sanitization, fallback slot filling
3. **Text Salvage**: Emergency process scanning raw text for valid IDs (recovers ~85% of malformed replies)

Ollama API Integration

Default optimization parameters:

- Temperature: 0.2
- Top-p: 0.95
- Max tokens: 240
- Repeat penalty: 1.07
- Context window: 4096

CANDIDATE FILTERING (`movies_filtering.go`)

Multi-Layer Filtering Architecture

1. **Mandatory Filters**: Age appropriateness, MPAA/TV rating mappings, conservative defaults for minors
2. **Content Quality Filters**: Removes concerts, live performances, music albums, trailers, incomplete metadata
3. **Behavioral Filters**: Watch history matching (exact, partial, fuzzy >80%), exclusion for ≥80% watched content
4. **Genre-Based Filtering**: Weighted genre distribution, dynamic similarity thresholds, adaptive rigor
5. **Fallback Strategy**: Ensures minimum 3×topK candidates through progressive threshold relaxation

SCORING IMPLEMENTATION (`movies_scoring.go`)

Quality Score Components (35% of Total)

1. **Reasoning Score**: Evaluates fallback usage, generic patterns, specificity
2. **Genre Match Score**: Cosine similarity between user and recommendation genre distributions
3. **Diversity Score**: Measures genre variety across recommendations
4. **Novelty Score**: Identifies overlap with watch history

Additional Scoring Metrics

- **Catalog Score (10%)**: Penalizes invalid/non-existent IDs
- **Latency Score (35%)**: Rewards faster models relative to fastest
- **Token Efficiency (10%)**: Evaluates output length optimality
- **Redundancy Score (10%)**: Penalizes repeated recommendations

Ranking Algorithm

1. Filter invalid models (schema errors, catalog mismatches)
2. Sort by total score (descending)
3. Assign rank values

HTTP HANDLERS (`movies_handlers.go`)

Handler Patterns

1. **Health Check**: Verifies database connectivity, Ollama availability, response latency
2. **Run Export**: Generates CSV with user profile, consensus recommendations, per-model results
3. **Leaderboard**: Provides model performance summaries, historical trends, pagination
4. **User Profile Management**: Retrieves demographics, preferences, watch history

Cache Management

Cache key components:

- Model name
- TopK value
- Candidate cap
- Catalog checksum
- Prompt version
- Canonical input JSON

Cache invalidation triggers:

- Catalog changes
- Prompt template updates
- Model configuration changes
- Parameter modifications

Active Run Management

- Cancellation propagation via contexts
- Database state updates
- Controlled goroutine shutdown
- Immediate user feedback

3.3 Shopping Recommendation Engine (`internal/engine/llm/shopping/`)

The shopping engine mirrors the movie engine architecture with domain-specific adaptations for product recommendations.

CORE COMPONENTS

Server Integration (`shopping_server.go`)

- HTTP API integration for shopping recommendations
- Parallel structure to movie server with product-focused flows

Run Orchestration (`shopping_run.go`)

Phases 1-8 identical to movie engine with shopping-specific adaptations:

- Input normalization for wishlist, cart, demographics
- Product-based candidate filtering (availability, category relevance, quality checks)
- Shopping-specific prompt engineering (wishlist/cart influence, category distribution, budget considerations)
- Concurrent execution with same timeout controls
- Shopping scoring using product semantics
- GPT-5 adjudication with shopping criteria (category relevance, intent alignment, commercial appeal)
- Persistence to `shopping_recommendations` and `shopping_metrics` tables

Prompt Construction (`shopping_processing.go`)

Structured identically to movie prompts but rewritten for product context:

- Strict JSON compliance enforcement
- Wishlist and cart as primary personalization signals
- Product category interpretation instead of genres
- Price, description, and metadata awareness
- Same three-phase output processing (parsing, grounding, salvage)

Filtering Logic

- Category-based similarity (replaces genre similarity)
- Cart-based exclusion with upsell pattern support
- Wishlist-derived preference signals
- Product metadata completeness validation
- Same 3×topK fallback strategy

Scoring (`shopping_scoring.go`)

Identical framework to movie scoring with product-specific semantics:

- Reasoning quality (product-focused)
- Category distribution matching
- Token and latency evaluation

- Redundancy detection using product IDs

Handlers (`shopping_handlers.go`)

Shopping-specific HTTP operations:

- Run initiation and result retrieval
- Per-model output fetching
- Leaderboard and latest run endpoints
- Wishlist/cart updates
- CSV export with product data
- Active run lifecycle management

Type Definitions (`shopping_types.go`)

Core structures:

- `ShopCandidate`: Catalog product representation
- `ShopRunInput`: Full input schema
- `ShopRec/ShopRecOut`: Validated recommendation structures
- `PerShopModelResp`: Per-model response with metrics
- `EnhancedShoppingGPTEvaluation`: GPT adjudication schema

Product-specific fields: Price, Currency, Description, image references

3.4 Combined LLM Server (`internal/engine/llm/llm.go`)

Integrates movie and shopping engines into unified server:

- Creates and stores engine instances
- Mounts respective routes
- Provides combined batch endpoint (runs both engines per user)
- Handles shared configuration
- Type conversion utilities (watch history format translation)

4. ARCHITECTURAL EVOLUTION AND TESTING

4.1 Development Phases

PHASE 1: Baseline Model Execution

- Individual model execution with simple prompts
- Result: Inconsistent outputs, invalid JSON, catalog constraint violations

PHASE 2: Prompt Refinement Cycles

- Iterative prompt evolution increasing strictness and specificity
- Improvements: Schema enforcement, reasoning guidelines, content prohibitions, ordering/uniqueness rules

PHASE 3: Concurrency Implementation

- Challenge: Sequential execution took 10+ minutes per run
- Solution: Goroutine-based parallel execution
- Initial issues: System lag, unbounded concurrency, timeout failures
- Resolution: Per-model timeouts, overall run timeouts, controlled concurrency, context cancellation, proper waitgroup usage

PHASE 4: Scoring and Validation Enhancement

- Implemented loose parsing → grounding → salvage pipeline
- Added quality scoring beyond structural validity

PHASE 5: Prompt and Catalog Alignment

- Migrated from internal IDs to English titles for reasoning
- Maintained ID mapping for final output
- Dramatic improvement in catalog grounding

PHASE 6: Final Validation

- Prompt version 05 achieved consistent, specific reasoning
- Stable production behavior validated

4.2 Concurrency Decision Analysis

WHY PARALLEL EXECUTION WAS CHOSEN

Sequential execution rejected because:

- 4-6× longer total execution time
- Prevented real-time model comparison
- Poor scalability for batch processing
- Inadequate frontend responsiveness

Parallel execution benefits:

- Reduced run time from 10+ minutes to 3-4 minutes
- Fair model comparison with identical inputs
- Improved user experience
- Scalable architecture

CONCURRENCY STABILIZATION MEASURES

- Per-model timeout enforcement
- Overall pipeline timeout
- Controlled concurrency slots
- Context-based cancellation
- Correct waitgroup synchronization
- Safe shared-resource locking

4.3 Prompt Evolution Summary

PROMPT 01: Simple Schema (Complete Failure)

- Basic JSON request with minimal constraints
- Issues: Invented titles, invalid IDs, no personalization, irrelevant content
- Root cause: LLMs treated task as creative writing

PROMPT 02: Hard Constraints (Partial Success)

- Added rating rules, content exclusions, format requirements
- Improvements: Better JSON structure, less invented content
- Remaining issues: Music content, test items, generic reasoning, format violations
- Root cause: Constraints were suggestions, not requirements

PROMPT 03: Taste-Driven Focus (Better but Inconsistent)

- Focused on taste over demographics, required specific reasoning patterns
- Improvements: Better personalization, more thoughtful recommendations
- Issues: Cross-model inconsistency, banned phrase usage, quality variation
- Root cause: Missing enforcement mechanism

PROMPT 04: Maximum Constraints (Still Problematic)

- Absolute bans on generic reasoning, strict safety rules, hard content exclusions
- Result: Better than previous versions but still required manual validation
- Key insight: Problem was presentation to LLM, not constraint content

PROMPT 05: System Component Identity (Production-Ready)

Critical Innovation: Identity and consequence framing

Old approach: "You are a movie recommender..."
New approach: "You are a STRICT backend movie recommendation component. Your output will be parsed by another program. ANY deviation from the required JSON format will BREAK the system."

Psychological Shifts:

1. Identity: From "creative assistant" to "system component"
2. Consequences: Clear failure implications
3. Parsing reality: Emphasizes machine readability
4. Mandatory vs. optional: Removes all ambiguity

Structural Advantages:

- Hierarchical flow: Identity → Consequences → Format → Content → Output
- Negative space: Explicit prohibitions
- Mathematical rules: Unambiguous percentage-based logic
- Complete examples: Clear success criteria

Why This Works:

- System identity creates restrictive role understanding
- Parse failure creates urgency to avoid "breaking the system"
- Programmatic parsing emphasizes machine consumption
- Binary requirements eliminate interpretation ambiguity

4.4 Test Environment, Hardware Requirements, and Evaluation Methodology

Test Environment & Hardware Specifications

All local system testing, including concurrent execution of multiple Large Language Models (LLMs), was performed on a single, consistent hardware environment to ensure fair comparison and reproducibility of results.

System Information

- Device Model: HP Pavilion Laptop 15-eg0xxx
- Operating System: Windows 11 Home (64-bit)
- System Type: x64-based PC
- BIOS Mode: UEFI

Processor (CPU)

- CPU: 11th Gen Intel® Core™ i5-1135G7
- Base Frequency: 2.40 GHz
- Architecture: Quad-core (8 logical threads)

Memory (RAM)

- Installed RAM: 16 GB
- Usable RAM: 15.8 GB

Storage

- Primary Drive: NVMe SSD
- Disk Type: SSD (NVMe)
- Usage:
    - Local LLM model storage
    - SQLite database
    - Application runtime and logs

This configuration supported parallel LLM execution using CPU-based inference without GPU acceleration.

Hardware Requirements for Concurrent Model Execution

To run multiple LLM models concurrently in a local environment, the following **minimum hardware requirements** are recommended:

- CPU: Modern multi-core processor (minimum 4 cores / 8 threads)
- Memory:
    - Minimum: 16 GB RAM
    - Recommended for larger models: 24–32 GB RAM
- Storage:
    - SSD required (NVMe recommended)
    - Minimum free space depends on number and size of models loaded
- Operating System:
    - 64-bit Windows, macOS, or Linux

These requirements ensure stable parallel execution, prevent memory thrashing, and allow consistent latency measurements across models.

Larger Model Constraints

While the system architecture supports flexible model registration, practical limitations exist for larger models when running on consumer-grade hardware.

- Larger models (7B+ parameters) significantly increase memory usage
- Concurrent execution of multiple large models may exceed available RAM
- CPU-only inference increases latency for large models

For this reason:

- Most evaluation runs focused on small-to-mid-sized instruction-tuned models
- Larger models are recommended to be evaluated on GPU-enabled or higher-memory systems in future deployments

Data Collection Methodology

To ensure consistency and fairness across model evaluations, the following methodology was applied:

- All test runs were executed on the same physical device
- No cloud GPUs or external inference services were used for local models
- Identical user profiles, watch history, catalogs, and prompt versions were used across runs
- Each model was executed multiple times to normalize variance
- Latency was measured relatively (normalized against the fastest model per run)
- Caching was disabled during evaluation runs unless explicitly tested

GPT-5 adjudication, when enabled, was performed via API and did not affect local hardware performance metrics.

4.5 Prompt Analysis

This document details the evolution of prompt design, including iterative refinements, constraint enforcement strategies, schema alignment decisions, and qualitative evaluation of prompt effectiveness across models.

- **Repository path:** `sd06_personalization_1/Documentation/Analyses/prompt_analysis.pdf`
- **Direct link:** prompt_analysis.pdf

5. ENHANCED SCORING AND ADJUDICATION

5.1 Scoring System Evolution

ORIGINAL BASELINE SYSTEM

Metric	Weight	Purpose	Limitations
Quality Score	30%	Penalizes fallback/generic reasons	No semantic content measurement
Latency Score	30%	Relative to fastest model	Overweighted
Catalog Score	20%	In-catalog item count	Too simplistic
Token Efficiency	10%	Deviation from optimal count	Arbitrary optimization
Redundancy Score	10%	Penalizes duplicates	Good but limited

Key Limitations: No semantic understanding, no genre matching intelligence, structural-only quality measurement, performance-dominated ranking

ENHANCED PRODUCTION SYSTEM

Enhanced Quality Metrics (45% of Total Score)

Metric	Weight	Purpose	Implementation
Genre Match Score	40%	Content alignment from watch history	Cosine similarity with watch history
Reasoning Score	25%	Explanation quality	Penalizes generic/fallback reasoning
Diversity Score	20%	Variety in recommendations	UniqueGenres / TotalRecommendations
Novelty Score	15%	Prevents repeated content	% not in watch history

Quality Score Formula:

```
QualityScore =
  GenreMatchScore × 0.40 +
  ReasoningScore × 0.25 +
  DiversityScore × 0.20 +
  NoveltyScore × 0.15
```

Total Score Formula:

```
TotalScore =
  QualityScore × 0.35 +
  CatalogScore × 0.10 +
  LatencyScore × 0.35 +
  TokenEfficiency × 0.10 +
  RedundancyScore × 0.10
```

This ensures quality dominates selection while maintaining system stability and performance requirements.

5.2 Detailed Metric Definitions

1. GENRE MATCH SCORE (40% of Quality)

- Uses watch history with percentage-based weighting (≥80% = strong, 30-79% = moderate, <30% = weak)
- Converts genres to weighted vectors
- Applies cosine similarity between user preference and recommendation vectors
- Scaled 0-100

2. DIVERSITY SCORE (20% of Quality)

- Calculation: (Unique Genres / Total Recommendations) × 100
- High score = broad coverage, Low score = repetitive

3. REASONING SCORE (25% of Quality)

- Penalties per recommendation:
  - "fallback pick" or empty: -0.8
  - "matches profile" or generic: -0.3
  - Specific, personalized: 0
- Rewards custom, user-aware reasoning (0-100 scale)

4. NOVELTY SCORE (15% of Quality)

- Calculation: (Non-Watch-History Recommendations / Total) × 100
- Encourages discovery and prevents stagnation

5. CATALOG SCORE (10% of Total)

- Calculation: (Valid Catalog Items / Total) × 100
- Invalid/hallucinated items receive hard penalties

6. LATENCY SCORE (35% of Total)

- Fastest model receives highest score
- Others scaled proportionally (0-100 normalized)

7. TOKEN EFFICIENCY SCORE (10% of Total)

- Based on prompt and output token counts
- Models closest to optimal range score highest

8. REDUNDANCY SCORE (10% of Total)

- Penalizes duplicate IDs, excessive genre repetition, overlapping semantics

5.3 GPT-5 Adjudication System

WHEN GPT-5 IS USED

- Conditionally enabled (requires OpenAI API key)
- Fully skipped when key absent
- Makes final model selection when enabled
- Receives all internal scores for semantic evaluation

GPT-5 EVALUATION METRICS

Metric	Weight	Purpose	Evaluation Method
Personalization	40%	User-recommendation fit	Age, gender, location, watch-history alignment
Content Quality	30%	Intrinsic recommendation quality	Genre consistency, diversity, novelty
Technical Excellence	20%	Output correctness	Reasoning clarity, format correctness, catalog compliance
Business Value	10%	Practical value	User satisfaction and engagement estimates

GPT-5 RESPONSE STRUCTURE

Per model:

- `overall_score` (0-100)
- `personalization`, `content_quality`, `technical_excellence`, `business_value` score s (0-100)
- `strengths` (2-3 bullets)
- `weaknesses` (2-3 bullets)
- `confidence` (0.0-1.0)
- `rank` (1 = best)

Overall:

- `consensus_recommendations`: Final curated recommendation IDs
- `evaluation_metadata`: Evaluation time, models evaluated, total recommendations

DECISION PROCESS

1. System sends all model recommendations, internal scores, user profile to GPT-5
2. GPT-5 evaluates each model using four criteria
3. GPT-5 assigns overall scores and final ranked list
4. Model with rank=1 becomes final production model
5. GPT-5's consensus recommendations become final output
6. All scores and rankings stored for auditing

EVOLUTION FROM BASELINE TO ENHANCED ADJUDICATION

Baseline Approach:

- Simple ranking only
- Limited data: latency, in-catalog rate, basic scores
- Output: `{"ranking_json": [...], "gpt_result_ids": [...]}`
- No numeric semantic scoring, no explanations, no confidence scores

Enhanced Approach:

- Full semantic evaluation with numeric scoring
- Comprehensive data: structured recommendations with reasons, all internal metrics
- Detailed output: model evaluations, breakdown scores, strengths/weaknesses, confidence, rank
- Fully explainable and auditable decisions

5.4 Rejected Metrics (Design Decisions)

Metric	Purpose	Rejection Reason
Sentiment Score	Measure explanation positivity	Positivity ≠ correctness; overlaps with Reasoning Score
Popularity/Trending Score	Boost trending content	Conflicted with personalization-first design
Cross-Session Stability Score	Track multi-run consistency	Suppressed novelty, increased compute cost
Explanation Length Score	Control verbosity	Length ≠ clarity; redundant with Reasoning Score
Model Agreement Score	Reward cross-model consensus	Reduced diversity, overlapped with consensus logic

6. SYSTEM DESIGN PRINCIPLES

6.1 Core Architecture Principles

FAULT TOLERANCE

- Individual model failures do not abort pipeline
- Graceful degradation with fallback recommendations
- Error recovery through text salvage mechanisms

OBSERVABILITY

- Structured logging for all operations
- Complete audit trail of prompts, responses, scores
- Performance metrics for monitoring and optimization

SECURITY AND PRIVACY

- No sensitive data in logs
- User data isolation
- Secure API key management for external services

STATELESS SCALING

- No session state on server
- Horizontal scalability support
- Cache-friendly architecture

PERFORMANCE OPTIMIZATION

- Intelligent caching reducing redundant LLM calls by ~40%
- Connection pooling for database operations
- Batch processing support for efficiency

6.2 Production Readiness

RELIABILITY MEASURES

- Timeout enforcement at multiple levels
- Graceful handling of malformed LLM outputs
- Comprehensive error handling and logging

QUALITY ASSURANCE

- Schema validation for all outputs
- Catalog grounding verification
- Multi-dimensional scoring for quality control

MONITORING AND DEBUGGING

- Health check endpoints
- Active run tracking and management
- CSV export for detailed analysis
- Leaderboard for model performance trends

MAINTAINABILITY

- Modular architecture with clear separation of concerns
- Shared infrastructure between movie and shopping engines
- Versioned prompts for reproducibility
- Comprehensive documentation

7. INTEGRATION GUIDE

7.1 API Endpoints

MOVIE RECOMMENDATIONS

- `POST /api/movies/run` - Execute recommendation run
- `GET /api/movies/run/{id}` - Retrieve run results
- `GET /api/movies/leaderboard` - Model performance summary
- `GET /api/movies/export/{id}` - Export run to CSV
- `GET /api/movies/health` - System health check

SHOPPING RECOMMENDATIONS

- `POST /api/shopping/run` - Execute recommendation run
- `GET /api/shopping/run/{id}` - Retrieve run results
- `GET /api/shopping/leaderboard` - Model performance summary
- `GET /api/shopping/export/{id}` - Export run to CSV

USER MANAGEMENT

- `GET /api/user/{id}` - Retrieve user profile
- `PUT /api/user/{id}/wishlist` - Update wishlist
- `PUT /api/user/{id}/cart` - Update cart

RUN MANAGEMENT

- `GET /api/active-runs` - List active runs
- `POST /api/stop-run/{id}` - Cancel running operation

7.2 Request/Response Formats

Movie Recommendation Request:

```
{
  "user_id": "string",
  "top_k": 5,
  "models": ["phi3:mini", "qwen2.5:3b-instruct"],
  "preferred_movies": ["movie_id_1", "movie_id_2"],
  "watch_history": [
    {"id": "movie_id", "watched_percent": 85}
  ]
}
```

Response Structure:

```
{
  "run_id": "string",
  "timestamp": "ISO8601",
  "consensus_recommendations": [...],
  "model_results": {
    "model_name": {
      "recommendations": [...],
      "scores": {...},
      "rank": 1,
```

```
      "latency_ms": 130379
    }
  },
  "gpt_evaluation": {...}
}
```

7.3 Configuration

Environment Variables:

- `DATABASE_PATH`: SQLite database location
- `OLLAMA_URL`: Ollama API endpoint
- `OPENAI_API_KEY`: Optional GPT-5 adjudication key
- `CANDIDATE_CAP`: Maximum candidates per prompt
- `PER_MODEL_TIMEOUT_MS`: Individual model timeout
- `OVERALL_TIMEOUT_MS`: Total pipeline timeout

7.4 API Documentation

This documentation provides detailed specifications for all exposed REST APIs, including endpoint definitions, request and response formats, supported parameters, error handling, and integration guidance for client applications.

- **Repository path:** `sd06_personalization_1/ai-recommendation-engine/docs/swagger.yaml`
- **Direct link:** https://git.las.iastate.edu/SeniorDesignComS/2025fall/402c/sd06_personalization_1/-/blob/main/ai-recommendation-engine/docs/swagger.yaml

8. APPENDICES

8.1 Glossary

- **Grounding**: Validation that recommended IDs exist in catalog
- **Candidate Pool**: Subset of catalog items passed to LLM
- **Consensus Recommendations**: Final curated recommendations (from GPT-5 or best model)
- **Fallback Pick**: Generic recommendation used when LLM output is invalid
- **Watch Percentage**: Proportion of content consumed by user
- **Adjudication**: Final evaluation and model selection process

Document Control

Version	Date	Author	Status
1.0	December 16, 2025	ISU Development Team	Final

Prepared for IdeaNova Technologies