Architectural and Algorithmic Blueprint for a University-Scale Recommendation Engine

Executive Summary

The development of a recommendation engine for a localized, university-scale platform—specifically designed to match students with campus clubs and organizations based on interests and behavior—presents a highly specific set of socio-technical constraints and architectural opportunities. The platform, denoted as GatorCommunities, operates within an environment characterized by a small to medium user base numbering in the hundreds to low thousands, and an item catalog restricted to approximately 50 to 200 student organizations. Unlike internet-scale platforms that possess massive volumes of interaction data required to train deep neural architectures, a campus community discovery platform operates in an environment defined by severe initial data sparsity, strict latency requirements, highly specialized content, and seasonal spikes in user activity corresponding to academic semesters.

The technical stack architecture features a Next.js frontend, a Prisma-managed PostgreSQL database responsible for all state mutations, and a Python FastAPI backend strictly dedicated to read-only recommendation scoring. This segregation of responsibilities mandates a highly decoupled system where the recommendation engine operates without mutating the core transactional database. Furthermore, the availability of high-performance computing resources via the University of Florida's HiPerGator infrastructure offers robust batch-processing capabilities, allowing the system to offload heavy matrix computations from the live web server.

This report provides an exhaustive analysis of the algorithmic, architectural, and operational strategies required to build, deploy, and scale this recommendation system. It critically evaluates the proposed baseline of Jaccard and Cosine similarity, mathematically demonstrating their limitations in sparse environments. Consequently, it proposes superior, lightweight collaborative filtering alternatives tailored for sparse implicit feedback matrices that run efficiently on NumPy and SciPy. The analysis formulates advanced cold-start mitigation techniques utilizing graph-based dummy node injections, details the intricate integration of read-only database reflections utilizing SQLAlchemy, outlines the optimal deployment of SLURM-based batch pipelines, evaluates the necessity of job queues, and establishes a rigorous evaluation framework centered on rank-aware offline metrics and online behavioral telemetry.

Data Architecture and Implicit Signal Processing

Before algorithms can be evaluated, the nature of the data fueling the recommendation engine must be mathematically structured. The GatorCommunities platform relies on two primary data modalities: explicit metadata generated during user onboarding and implicit behavioral feedback generated through platform usage.

Explicit metadata consists of user-selected interest tags and community-defined tags and descriptions. This data is categorical, deterministic, and highly valuable during the initial stages of a user's lifecycle on the platform. Conversely, the behavioral data is implicit. Users do not provide explicit 1-to-5 star ratings for clubs; instead, their preferences are inferred through telemetry. The platform captures views, clicks, RSVPs, and joins. To formulate a recommendation matrix, these heterogeneous interactions must be aggregated into a single scalar value representing the system's confidence in the user's affinity for a specific club.

The proposed interaction weights are configured as views (0.5), clicks (1.0), RSVPs (2.0), and joins (3.0). In a mathematical formulation, the total interaction score r_{ui} for a user u and an item i can be represented as the linear combination of the frequency of each interaction type multiplied by its respective weight. However, treating these raw scores directly as explicit ratings introduces significant bias, as a user who clicks a club ten times might generate a higher score than a user who actually joins the club.

To mitigate this, the interaction matrix should be modeled as an implicit feedback confidence matrix, a standard practice for platforms relying on telemetry rather than ratings.[1] The raw aggregated score r_{ui} is transformed into a confidence metric c_{ui}, typically using a formulation such as $c_{ui} = 1 + \alpha r_{ui}$, where α is a scaling hyperparameter. The binary preference p_{ui} is set to 1 if $r_{ui} > 0$, and 0 otherwise. This mathematical distinction is crucial for the selection of collaborative filtering algorithms, as models designed for explicit feedback (like standard Singular Value Decomposition) perform poorly on implicit confidence data.[3]

Furthermore, the sparsity of this interaction matrix defines the algorithmic boundaries. With 1,000 users and 200 clubs, the theoretical interaction space contains 200,000 possible pairs. During the initial weeks of a semester, a typical user might interact with only 5 to 10 clubs. If the average user interacts with 5 clubs, the matrix contains only 5,000 non-zero entries, resulting in a sparsity level of 97.5%. The chosen algorithms must be exceptionally resilient to this lack of density.[4]

Algorithmic Foundations: Baseline Evaluation

The foundational proposal for the GatorCommunities recommendation engine involves a hybrid system combining content-based filtering via Jaccard similarity and collaborative filtering via user-based Cosine similarity. While these algorithms are accessible and easy to

implement, their application to a highly sparse, university-scale dataset requires critical mathematical examination.

The Viability of Jaccard Similarity

Content-based filtering relies exclusively on analyzing user and item metadata, which in this architecture takes the form of user-selected interest tags and community-defined tags. The Jaccard index is a highly effective, computationally inexpensive metric for comparing the similarity of finite sample sets.[7] It is mathematically defined as the size of the intersection divided by the size of the union of the two sets of tags:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

For a platform where users explicitly select tags during the onboarding phase, Jaccard similarity provides an immediate, highly interpretable baseline for content matching.[8] It is entirely immune to the interaction data sparsity problem because it relies solely on deterministic metadata rather than historical behavioral patterns.[10] If a user selects "Engineering" and "Robotics," and a club possesses the tags "Engineering," "Robotics," and "Competition," the Jaccard index calculates a straightforward proportional overlap.

The primary limitation of Jaccard similarity, and content-based filtering in general, is its inability to capture latent user preferences or surface serendipitous recommendations.[12] It strictly confines users to a "filter bubble" defined by their initial tag selections, ignoring the nuanced, often unpredictable ways in which student interests evolve through peer influence.[13] Therefore, while Jaccard similarity is a robust component for candidate generation and cold-start mitigation, it cannot serve as the sole engine for long-term user engagement.

The Vulnerability of Cosine Similarity in Sparse Matrices

The proposed collaborative filtering mechanism utilizes user-based Cosine similarity on the user-community interaction matrix. Cosine similarity measures the cosine of the angle between two non-zero vectors in an inner product space, determining their directional alignment regardless of their magnitude.[7]

$$C(u, v) = \frac{\sum_{i=1}^{n} r_{ui} r_{vi}}{\sqrt{\sum_{i=1}^{n} r_{ui}^2} \sqrt{\sum_{i=1}^{n} r_{vi}^2}}$$

In an environment characterized by thousands of active users and dense interaction logs, user-based collaborative filtering can identify behavioral cohorts effectively.[2] However, in a dataset with hundreds of users, 200 items, and a sparsity level exceeding 97%, raw user-based

Cosine similarity presents severe algorithmic vulnerabilities.[7]

The fundamental flaw lies in the numerator of the Cosine formula, which requires computing the dot product of two user interaction vectors. For the dot product to yield a non-zero value, two users must have interacted with at least one identical club. When user vectors are overwhelmingly populated with zeroes, the probability of two random users having overlapping interactions is exceptionally low.[7] Consequently, the dot product frequently evaluates to zero, leading the algorithm to conclude that the users are completely dissimilar (orthogonal), even if they share deep latent similarities.[7]

Furthermore, user-based collaborative filtering scales poorly as the user base grows and suffers from extreme variance when the number of interactions per user is low.[4] Small changes in a user's interaction history can drastically alter their similarity neighborhood, leading to unstable and erratic recommendations.[4] Therefore, while Cosine similarity is a mathematically sound metric for dense continuous vectors, applying it directly to sparse interaction matrices is an anti-pattern for this specific architectural context.

Lightweight Collaborative Filtering Alternatives

Given the mathematical limitations of raw user-based Cosine similarity, the architecture must transition toward algorithms capable of extracting latent, lower-dimensional patterns from highly sparse matrices. A strict constraint of the platform is the avoidance of heavy machine learning frameworks such as TensorFlow or PyTorch, favoring lightweight Python implementations utilizing NumPy and SciPy. This constraint optimally narrows the algorithmic selection to advanced matrix factorization techniques and regularized linear autoencoders.

Alternating Least Squares (ALS)

Alternating Least Squares (ALS) is a highly optimized matrix factorization technique specifically designed to process implicit feedback datasets, making it vastly superior to standard Singular Value Decomposition (SVD) for telemetry-based platforms.[3]

ALS operates by decomposing the sparse user-item interaction matrix R into two dense, lower-dimensional matrices: a user factor matrix U and an item factor matrix V.[14] The core assumption is that a user's affinity for a club can be approximated by the dot product of their respective latent vectors: $R \approx U \times V^T$.[15]

Because the interaction matrix contains a massive number of unobserved entries (clubs the user has not seen), standard gradient descent approaches struggle to optimize the loss function efficiently without overfitting to the observed data. ALS circumvents this by recognizing that the objective function is non-convex when both U and V are unknown, but

becomes a convex quadratic problem if one of the matrices is held constant.[15] The algorithm alternates between fixing the user matrix to solve for the item matrix optimally, and then fixing the item matrix to solve for the user matrix, repeating this process until convergence is achieved.[15]

For the GatorCommunities platform, the Python implicit library provides a highly optimized implementation of ALS that utilizes SciPy sparse matrices and multi-threaded Cython extensions.[14] ALS is exceptionally fast to train on small datasets, scales elegantly, and, most importantly, mathematically infers preferences for unobserved user-club pairs by routing similarities through the latent dimensions.[14]

Embarrassingly Shallow Autoencoders (EASE)

For a university-scale catalog restricted to 50 to 200 items, Embarrassingly Shallow Autoencoders (EASE) represents a state-of-the-art, highly aggressive alternative to traditional latent factor models.[17] Despite its simplicity, EASE consistently outperforms deep neural collaborative filtering architectures on sparse datasets.[17]

EASE is a linear model that learns an item-item weight matrix B directly from the user-item interaction matrix X. The algorithm seeks to predict a user's interaction vector by multiplying it by the learned weight matrix: $\hat{X} = X \times B$. To prevent the algorithm from learning the trivial identity matrix (where it simply predicts an item because the user has already interacted with it), EASE enforces a strict constraint that the diagonal of the weight matrix B must be zero.[19] The objective function minimizes the squared reconstruction error with L_2 regularization:

$$\min_{B} ||X - XB||_F^2 + \lambda ||B||_F^2 \quad \text{subject to} \quad \text{diag}(B) = 0$$

The profound advantage of EASE is that this objective function possesses a direct, closed-form mathematical solution.[17] It does not require epochs, learning rates, iterative gradient descent, or embedding dimension optimization.[17] The solution is calculated through a straightforward inversion of the item-item Gram matrix: $P = (X^T X + \lambda I)^{-1}$, making the training process incredibly fast using standard NumPy and SciPy linear algebra routines.[17]

Furthermore, because EASE is strictly an item-item model, it does not learn separate, isolated embeddings for users.[17] This architectural trait is highly advantageous for dynamic platforms. When a new user joins GatorCommunities and generates their very first interaction (e.g., clicking on a single club), EASE can instantly multiply that minimal interaction vector against the

pre-computed item-item matrix B to generate highly accurate predictions, without requiring the entire model to be retrained to learn a new user embedding.[17]

LightFM: The Hybrid Matrix Factorization Approach

While ALS and EASE excel at collaborative filtering, they remain inherently blind to the rich metadata available within the platform, such as the club descriptions and user onboarding tags. LightFM is a dedicated Python library engineered specifically to seamlessly integrate collaborative filtering with content-based filtering within a unified matrix factorization framework.[20]

LightFM represents users and items not merely as isolated latent vectors, but as the sum of the latent representations of their constituent features.[12] For a student club, its latent representation is computed by adding the embedding of the club's unique ID to the embeddings of its associated tags (e.g., "Engineering," "Social," "Academic"). If a user interacts with that club, the backpropagation process updates the embeddings for the specific user and club, but also simultaneously updates the embeddings for the associated tags.[22]

This mechanism is exceptionally potent for platforms characterized by sparse interactions but rich onboarding data.[21] It explicitly targets the top-K recommendation problem by utilizing the Weighted Approximate-Rank Pairwise (WARP) loss function.[22] WARP optimizes directly for ranking quality by sampling negative items until it finds one that the model currently ranks higher than a known positive item, penalizing the model based on how many samples were required.[22] For sorting a small catalog of 200 clubs into a personalized discovery feed, LightFM trained with WARP loss represents a highly sophisticated, yet computationally lightweight, hybrid solution.[18]

Algorithm	Core Mechanism	Mathematical Foundation	Optimal Use Case	Complexity / Framework
Jaccard Similarity	Content-based tag matching	Finite set intersection/union	Strict user cold starts, fallback baselines	Low / Pure Python or NumPy
Cosine Similarity	User-User vector matching	Inner product space angle	Dense interaction matrices (Not recommended here)	Low / SciPy distance metrics

ALS	Implicit Matrix Factorization	Alternating convex optimization	Identifying latent preferences from implicit behavior	Medium / implicit library
EASE	Linear Item-Item Similarity	Closed-form L_2 regularized inversion	Highly sparse data, fast item-item predictions	Low / NumPy linear algebra
LightFM	Hybrid Feature Embeddings	WARP loss gradient descent	Unifying collaborative data with rich metadata tags	Medium / lightfm library

Hybrid Weighting and Optimization Strategies

The initial architectural proposal suggests a static hybrid weighting mechanism, utilizing a heuristic ratio of 60% content-based filtering to 40% collaborative filtering. While a fixed heuristic serves as a functional baseline during the platform's initial deployment, recommendation systems mature rapidly when these proportional weights are treated as tunable hyperparameters rather than static constants.[25]

Relying on a static 60/40 split assumes that user behavior and platform maturity remain constant. In reality, the optimal weighting shifts drastically based on the density of the interaction matrix.[25] During the first weeks of the fall semester, the system possesses minimal behavioral data, meaning the content-based component must dominate the scoring logic. As the semester progresses and the matrix fills with RSVPs and joins, the behavioral intelligence of the collaborative filtering model becomes vastly superior to static tags, necessitating a shift in weight toward the collaborative model.[28]

The combined score for a user-item pair should be mathematically defined as a dynamic weighted ensemble:

$$Score_{hybrid} = \alpha \cdot \text{Normalize}(Score_{content}) + (1 - \alpha) \cdot \text{Normali}$$

It is critical that both constituent scores are mathematically normalized (e.g., Min-Max scaling or Z-score normalization) before combination; otherwise, the differing magnitudes of the

Jaccard index (bounded between 0 and 1) and raw matrix factorization logits will render the α weight mathematically irrelevant.[25]

To optimize the α parameter, the system should avoid manual guesswork and employ algorithmic hyperparameter tuning.[30] Given the small scale of the dataset (hundreds of users), executing an exhaustive Grid Search alongside k-fold cross-validation requires negligible computational time.[28] The Grid Search evaluates discrete increments of α (e.g., 0.1, 0.2... 0.9) against a specific offline metric, such as NDCG@10, selecting the ratio that mathematically maximizes ranking accuracy.[29] Advanced evolutionary optimization techniques, such as Differential Evolution, can also be deployed to automate the fine-tuning of these weight parameters during the batch retraining pipeline, yielding measurable increases in both F1-scores and overall recommendation effectiveness.[25]

Formulating Cold-Start Mitigation Techniques

The cold-start problem is universally recognized as the most critical hurdle for nascent recommendation systems.[5] In the socio-technical context of a university application, the system faces severe cold-start scenarios on multiple fronts: strict user cold starts (incoming freshmen with zero historical interactions), strict item cold starts (newly established campus organizations lacking members), and a system-wide cold start (the initial launch day of the application).[5]

Addressing Strict User and System Cold Starts

When a user registers and possesses absolutely no interaction history, collaborative algorithms like ALS or EASE lack the mathematical input required to project the user into the latent space, rendering them incapable of generating a personalized score.[20] The immediate architectural fallback must consist of knowledge-based and popularity-based baseline systems.[11]

The knowledge-based recommendation system is driven entirely by the user's explicit onboarding tags.[11] By computing the Jaccard similarity between the user's selected interest portfolio and the comprehensive metadata tags of the campus clubs, the system bypasses the need for behavioral telemetry.[7] This ensures that within milliseconds of completing the onboarding flow, the user receives a highly relevant, localized feed customized to their stated academic and social interests.[7]

However, system design must account for user friction; a significant percentage of users may skip optional onboarding workflows, leaving both their interaction vector and metadata vector entirely empty. In this extreme scenario, the system must deploy a non-personalized popularity baseline.[11] Popularity is computed by aggregating the weighted implicit feedback scores (views, clicks, RSVPs, joins) across the entire user base.[34] While devoid of personalization,

recommending the most highly active, heavily engaged communities guarantees that the discovery feed is populated with high-quality content, preventing the catastrophic user experience of a blank screen.[32] This popularity feed acts as an engagement catalyst, encouraging the initial user interactions that subsequently feed the collaborative algorithms, thereby breaking the cold-start cycle.[32]

Advanced Metadata Injection via Dummy Nodes

Mitigating the item cold start—when a newly formed club is added to the database—presents a complex challenge for pure collaborative filtering models. Because the new club lacks interaction edges linking it to users, matrix factorization models naturally isolate the club in the latent space, preventing it from ever being recommended.[5] While LightFM handles this natively through tag embeddings [22], employing lightweight models like ALS or EASE requires a more creative architectural technique known as "dummy node injection".[16]

Dummy node injection mathematically coerces collaborative filtering algorithms into recognizing content-based similarities without altering the underlying matrix factorization engine.[16] The methodology involves modeling the recommender system as a massive bipartite graph consisting of user nodes and item nodes.[16]

When a new club is created with the tags "Robotics" and "Engineering":

1. **Synthetic User Creation**: The system instantiates synthetic "dummy users" for every distinct tag category present within the university ecosystem (e.g., a "Dummy_Robotics_User" and a "Dummy_Engineering_User").[16]
2. **Synthetic Edge Establishment**: The system registers synthetic interactions between the dummy users and every club on the platform that shares the respective tags.[16]
3. **Algorithmic Weighting**: Crucially, these synthetic interactions are assigned a severely discounted baseline weight within the implicit feedback matrix.[16] This ensures that the synthetic structural links influence the geometry of the latent space without mathematically overpowering genuine human interaction telemetry.[16]

When ALS or EASE factors the augmented interaction matrix, the algorithms perceive that every single "Robotics" club on campus has been interacted with by the identical synthetic user. Consequently, the latent item vectors for all "Robotics" clubs are gravitationally pulled closer together within the multi-dimensional vector space.[16] Even if the newly created club has zero real human members, its synthetic connection to the dummy node ensures its latent vector is positioned adjacent to established, highly popular robotics clubs.[16] If an active user subsequently interacts with an established robotics club, the model mathematically infers an affinity for the new club due to its geometric proximity, generating a serendipitous recommendation.[16] This technique effectively bridges explicit metadata and implicit collaborative filtering, yielding significant precision gains in sparse environments while

maintaining the exceptional computational speed of standard matrix algebra.[16]

System Architecture: Pre-compute versus Live Scoring

The architectural interaction paradigm between the Next.js frontend, the Prisma-managed PostgreSQL database, and the Python FastAPI recommendation backend requires rigorous analysis regarding latency budgets and computational overhead. The central architectural decision dictates whether to pre-compute recommendation scores asynchronously in batch processes or execute live calculations dynamically at request time.[35]

Latency Budgets and Mathematical Constraints

At the specific scale of a university platform—defined by an upper bound of a few thousand users and a catalog strictly limited to 200 items—the mathematical reality overwhelmingly favors a pre-computation architecture for the collaborative filtering engine.[36]

Live scoring introduces significant latency risks. Fetching high-dimensional interaction vectors, executing matrix multiplications, sorting arrays, and extracting the top K candidates during an active HTTP request consumes precious milliseconds, risking violation of standard 100ms API latency budgets.[36] Furthermore, live scoring scales poorly during traffic spikes, as every concurrent page load triggers complex linear algebra operations on the FastAPI server.

Conversely, an interaction matrix consisting of 5,000 users and 200 clubs contains exactly 1,000,000 possible distinct pairs. Calculating the entirety of the recommendation scores for the whole platform requires mere fractions of a second using optimized NumPy vectorization or the EASE closed-form matrix multiplication.[17] Because the total computational cost is trivial, pre-computing the top 50 recommendations for every single user in advance and caching the resulting integer arrays is the optimal strategy.[36] This ensures that when the Next.js frontend queries the FastAPI endpoint, the server simply performs an $O(1)$ lookup in a Redis cache or a dedicated PostgreSQL materialized view, responding in sub-millisecond times securely within stringent latency constraints.[36]

The Optimal Hybrid Serving Architecture

Live scoring, however, becomes an absolute necessity under the strict cold-start condition. When a new user completes the onboarding process and immediately navigates to their discovery feed, they possess a freshly generated User ID that has not yet been processed by the nightly batch pipeline.[32]

To reconcile these competing demands, the GatorCommunities platform must deploy a hybrid serving architecture:

- **The Pre-Compute Primary Path**: A scheduled batch process running periodically processes the entire interaction matrix, generating the ranked recommendations for all historically active users.[36] These pre-computed arrays are serialized and pushed to the database.
- **The Live Scoring Fallback**: The FastAPI backend receives a user ID. It attempts to fetch the pre-computed recommendations. If a cache miss occurs (indicating a brand-new user), the FastAPI backend instantly executes a live Jaccard similarity calculation utilizing the user's explicit onboarding tags.[11] Because Jaccard similarity is computationally lightweight and requires no matrix factorization, it executes safely within the latency budget, serving a dynamic content-based feed until the next scheduled batch run formally incorporates the user into the collaborative matrix.[32]

Evaluating Queue Processing versus Batch Retraining

The initial engineering inquiry raised the possibility of utilizing queue processing (e.g., Celery, RabbitMQ, or Redis Queues) to handle new interaction data in real-time. In the context of a small-scale, highly sparse recommendation engine, deploying a distributed task queue introduces unnecessary infrastructural complexity and operational overhead.[38]

Heavy message brokers like Celery are designed for asynchronous microservice architectures where prolonged, IO-bound tasks must be executed across multiple distributed worker nodes.[38] Maintaining a live Celery worker cluster simply to incrementally update a small 1000x200 matrix is an architectural anti-pattern.[38] Matrix factorization algorithms like ALS do not lend themselves to continuous, real-time incremental updates for single interactions; the entire matrix must generally be factored holistically to adjust the latent geometric space accurately.[15]

Instead of complex task queues, the Next.js frontend should simply write the raw telemetry events (views, clicks, RSVPs) into an append-only interaction log table within the PostgreSQL database.[36] If minor, lightweight asynchronous tasks are required within the Python layer, FastAPI's native BackgroundTasks module provides sufficient concurrency without requiring external message brokers or worker processes.[38] The actual algorithmic retraining is deferred entirely to a robust, scheduled batch pipeline running on high-performance computing infrastructure.

Database Integration: Prisma, PostgreSQL, and FastAPI

The architectural requirement that the Next.js frontend manages all state mutations via the Prisma Object-Relational Mapper (ORM), while the Python FastAPI backend maintains strictly read-only access to the shared PostgreSQL database, demands specific data access patterns

to prevent schema drift and application instability.[42]

Prisma utilizes its own declarative schema language (schema.prisma) to manage database migrations, table creation, and referential integrity.[42] The Python backend must never attempt to independently alter the database schema using tools like Alembic, nor should developers manually duplicate the ORM definitions in Python using standard SQLAlchemy declarative base classes.[45] Manual duplication introduces severe maintenance overhead; any change to the Prisma schema by the frontend team would require synchronized, manual updates to the Python codebase, inevitably leading to runtime crashes when the schemas diverge.[47]

SQLAlchemy Dynamic Schema Reflection

To interact with the Prisma-managed PostgreSQL database securely and dynamically, the Python FastAPI application must leverage SQLAlchemy's automap_base extension.[49] The Automap extension acts as an introspective bridge; upon application startup, it actively queries the PostgreSQL Information Schema, reflects the existing table structures, identifies primary and foreign key constraints established by Prisma, and generates mapped Python classes on the fly.[49]

Python

```
from sqlalchemy.ext.automap import automap_base
from sqlalchemy.orm import Session
from sqlalchemy import create_engine

# Establish connection to the shared PostgreSQL database
engine = create_engine("postgresql+psycopg2://user:password@host/dbname")

# Dynamically reflect the Prisma-managed schema
Base = automap_base()
Base.prepare(autoload_with=engine)

# Access dynamically generated classes mapping directly to Prisma tables
Users = Base.classes.User
Clubs = Base.classes.Club
Interactions = Base.classes.Interaction
```

This dynamic reflection ensures that the Python backend remains permanently synchronized with the frontend's database structure.[49] If the Next.js team adds a new column to the Club table via a Prisma migration, the SQLAlchemy Automap mechanism instantly recognizes and

exposes that column to the recommendation logic upon the next FastAPI service restart.[49]

Enforcing Read-Only Constraints and Transaction Safety

While database-level user permissions must explicitly restrict the Python service account credentials to SELECT privileges [50], application-level safeguards represent a critical defense-in-depth best practice. SQLAlchemy allows database connections and sessions to be explicitly configured for read-only transaction modes.[52]

By passing the postgresql_readonly flag into the connection execution options, the application informs the PostgreSQL engine that the ensuing transaction is strictly analytical.[52] This not only prevents inadvertent data mutation but also allows the PostgreSQL query planner to optimize lock management and resource allocation for read-heavy operations.[52]

Python

```
with engine.connect() as conn:
    # Enforce strictly read-only execution at the application driver level
    conn = conn.execution_options(
        isolation_level="SERIALIZABLE",
        postgresql_readonly=True
    )
    with conn.begin():
        # Execute complex matrix extractions securely
        # Any accidental INSERT or UPDATE command will raise an immediate exception
```

Furthermore, the choice of database driver impacts asynchronous performance within the FastAPI framework. While asyncpg provides fully asynchronous, non-blocking I/O, rigorous performance benchmarking often reveals that synchronous database operations utilizing psycopg2—combined with standard synchronous def route definitions in FastAPI—can paradoxically yield superior requests-per-second throughput for specific, highly repetitive query profiles.[54] Thorough load testing under simulated campus traffic patterns must dictate the final driver configuration.[55]

High-Performance Computing: The HiPerGator Training Pipeline

The University of Florida's HiPerGator infrastructure offers a profound computational advantage, permitting the architecture to offload all computationally intensive machine

learning workloads from the lightweight web servers.[56] The recommendation training pipeline requires extracting historical telemetry from the PostgreSQL database, constructing the implicit feedback matrices, executing the matrix factorization algorithms, and subsequently injecting the optimized recommendation scores back into the system cache.[37]

Job Orchestration on the SLURM Scheduler

HiPerGator utilizes the Simple Linux Utility for Resource Management (SLURM) to allocate computational resources and schedule batch jobs across the cluster.[60] As previously established, employing complex external message queues for this task is architecturally redundant. The model retraining pipeline must rely entirely on SLURM's native, highly fault-tolerant batch processing capabilities.[61]

For periodic, scheduled retraining (e.g., executing nightly at 3:00 AM to process the previous day's telemetry), the scrontab (Slurm Crontab) utility is the designated, optimal tool.[62] scrontab functions with syntax identical to standard Unix cron, but executes the tasks directly within the distributed SLURM batch environment.[62] This ensures high availability; unlike a local cron job running on a single server, a scrontab job is scheduled across the entire cluster, guaranteeing execution even if individual compute nodes experience hardware failure.[62]

A standard training job script submitted to SLURM dictates the precise computational boundaries of the task.[63] Given the highly constrained scale of the dataset (the entire 1000x200 matrix occupies megabytes of memory), requesting massive multi-node MPI allocations or expensive GPU acceleration is an egregious misuse of HPC resources.[60] Algorithms like EASE and ALS are highly optimized for CPU execution. A standard single-node, multi-threaded CPU job allocation is the most efficient configuration for this workload.[60]

Bash

```
#!/bin/bash
#SBATCH --job-name=gator_recsys_batch
#SBATCH --mail-type=FAIL
#SBATCH --mail-user=admin@university.edu
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=4
#SBATCH --mem=8gb
#SBATCH --time=00:30:00
#SBATCH --output=recsys_train_%j.log

# Load environmental dependencies
```

```
module load python/3.10
module load anaconda3
conda activate gator_recsys_env

# Execute the extraction, training, and caching pipeline
python /blue/group_name/recsys/pipeline/train_and_cache.py
```

Architectural Note: Memory requests on HPC clusters must be rigorously calibrated. Users frequently over-request memory by orders of magnitude (e.g., requesting 100GB for a 50MB matrix), which severely degrades job scheduling queue times and overall cluster throughput.[63] For a sparse matrix of this specific dimensionality, an 8GB RAM allocation provides a highly conservative, mathematically secure safety margin.[60]

Storage Tier Optimization and Data Locality

Effective integration with the HiPerGator ecosystem requires a precise understanding of the distinct storage tiers and their respective I/O profiles.[67] Mismanaging data locality is a primary cause of pipeline bottlenecks on supercomputers.

- **/blue Storage**: The high-performance parallel file system (/blue) must serve as the persistent, authoritative repository for the Python virtual environments, the core application codebase, and the serialized final model artifacts (e.g., storing the EASE weight matrices or LightFM latent embeddings as .pkl files).[67] The /blue tier is optimized for large-scale parallel access.[67]
- **/scratch Storage**: Conversely, the active processing of raw data must occur on local scratch storage. All HiPerGator compute nodes are equipped with high-speed, flash-based local storage accessible via the $SLURM_TMPDIR environment variable.[68] When the Python script executes, it should query the PostgreSQL database, write the raw extracted interaction logs to the highly performant flash-based /scratch drives, construct and factor the matrices in memory, and then output the final recommendations back to the database or cache.[68] Attempting to perform heavy read/write matrix operations directly within a user's $HOME directory is a severe anti-pattern that violates HPC policies and cripples parallel processing speeds.[71]

Comprehensive Evaluation Framework and Quality Metrics

The efficacy of a recommendation system cannot be distilled into a singular metric. A mathematically sound evaluation framework must establish a dichotomy between offline algorithmic metrics—calculated rigorously during the batch training phase—and online behavioral metrics—monitored continuously via live application telemetry.[73]

Offline Algorithmic Evaluation Protocols

Evaluating a collaborative recommendation model requires strict temporal validation. The interaction dataset must be split chronologically; the model trains on historical interaction data (e.g., the first 8 weeks of the semester) and attempts to predict future interactions (the subsequent 2 weeks).[37] Utilizing standard randomized cross-validation splits severely violates the chronological nature of human behavior, resulting in data leakage where the model uses future information to predict past events.[37]

The offline evaluation focuses entirely on the system's ability to rank relevant items within a highly constrained subset presented to the user, universally denoted as K. In the context of the GatorCommunities UI, if the screen displays 10 clubs at a time, $K = 10$. The pytrec_eval Python library, serving as a standardized interface to the TREC evaluation tool, is perfectly suited for computing these rank-aware retrieval metrics.[75]

Evaluation Metric	Mathematical Focus	GatorCommunities Application Context
Precision@K	Fraction of recommended items that are relevant	Out of the 10 clubs shown on screen, how many did the user eventually interact with? High precision minimizes visual noise.[18]
Recall@K	Fraction of total relevant items successfully retrieved	Out of the total 4 clubs the user joined this semester, how many were discovered through the top 10 recommendations? High recall ensures maximum discovery.[18]
NDCG@K	Positional ranking quality based on graded relevance	Did the algorithm place the club the user *Joined* (weight 3.0) higher on the screen than the club they merely *Viewed* (weight

		0.5)?.[18]

Normalized Discounted Cumulative Gain (NDCG@K) represents the most critical, mathematically comprehensive metric for evaluating recommendation systems.[73] While Precision and Recall treat all items equally regardless of where they appear in the list, NDCG heavily penalizes the algorithm if highly relevant items are buried at the bottom of the feed.[74]

NDCG relies on the calculation of Discounted Cumulative Gain (DCG), which introduces a logarithmic discount factor based on the item's positional rank i:

$$DCG@K = \sum_{i=1}^{K} \frac{rel_i}{\log_2(i+1)}$$

By explicitly utilizing the platform's predefined interaction weights (views=0.5, clicks=1.0, RSVPs=2.0, joins=3.0) as the graded relevance scores (rel_i), the algorithm is forced to optimize for high-value conversions.[73] An algorithm that ranks a "Joined" club at position 1 and a "Viewed" club at position 10 will generate a vastly superior DCG score than the inverse arrangement.[75] NDCG ultimately normalizes this score against the Ideal DCG (IDCG)—the theoretical best possible sorting of the list—ensuring the final metric scales smoothly and comparably between 0 and 1 across all users.[75]

Behavioral Telemetry and Online Business Metrics

Relying exclusively on mathematically optimized offline metrics like NDCG frequently leads to highly accurate but functionally flawed systems. An algorithm prone to popularity bias might achieve exceptional offline accuracy by simply recommending the largest, most universally popular clubs to every single student on campus.[73] While mathematically sound, this degrades the user experience by destroying localized discovery and marginalizing smaller, niche organizations.[11]

To counteract algorithmic tunnel vision, offline metrics must be paired with continuous online behavioral telemetry [73]:

1. **Diversity**: Evaluates the statistical variance of recommended items across the entire user base.[73] High intra-list diversity ensures the UI is not monopolized by monolithic categories; a user shouldn't be subjected to an entire screen of identical "Computer Science" clubs if their interests are multifaceted.[3]
2. **Novelty and Serendipity**: Novelty computationally assesses the system's ability to recommend obscure items the user is highly unlikely to know about.[73] Serendipity measures the unexpectedness or pleasant surprise of a recommendation.[73] These metrics

are vital for helping newly established or specialized clubs gain visibility, directly combatting the inherent popularity bias that inherently plagues standard collaborative filtering matrices.[12]

3. **Click-Through Rate (CTR) and Conversion Velocity**: Once the algorithm is deployed into the Next.js production environment, the offline simulations must be correlated with live user telemetry.[73] CTR tracks immediate, superficial engagement with the recommendations. However, the ultimate arbiter of success is the Conversion Rate, determined by tracking the ratio of recommended club impressions that culminate in a high-value "RSVP" or "Join" event.[73] Continuous, structured A/B testing of the hybrid weighting parameter (α) against these live conversion rates is essential for the continuous optimization of the GatorCommunities platform.[11]

Conclusion

Constructing a highly effective, low-latency recommendation engine for a small-scale, high-sparsity university platform requires a deliberate architectural pivot away from resource-intensive neural networks. The engineering focus must center on mathematically elegant, computationally efficient linear algebra solutions.

The integration of Jaccard similarity for rapid, deterministic content matching guarantees robust cold-start performance, providing instantaneous relevance for new users. Concurrently, advanced lightweight matrix factorization techniques—specifically Embarrassingly Shallow Autoencoders (EASE) and LightFM—provide unparalleled capabilities in extracting latent behavioral preferences from highly sparse implicit feedback matrices. By architecturally decoupling the heavy recommendation logic into a nightly, pre-computed batch pipeline orchestrated on the UF HiPerGator SLURM scheduler, the system absolutely guarantees sub-millisecond API response times via the Python FastAPI backend. Furthermore, employing SQLAlchemy schema reflection inherently protects the operational integrity of the primary Prisma-managed PostgreSQL database, enforcing strict read-only compliance without introducing maintenance overhead.

This meticulously designed, multi-faceted algorithmic blueprint ensures that the GatorCommunities platform will deliver serendipitous, highly personalized community discovery from the moment of launch, scaling gracefully and adapting dynamically as student engagement patterns evolve across the academic year. By systematically evaluating the models using rank-aware offline metrics like NDCG alongside rich behavioral telemetry, the architecture establishes a sustainable foundation for continuous, data-driven optimization.

Works cited

1. Building a Recommender System using Machine Learning - Leonie Monigatti, accessed February 20, 2026, https://www.leoniemonigatti.com/blog/recommender-system-ml.html

2. Build a Recommendation Engine With Collaborative Filtering - Real Python, accessed February 20, 2026, https://realpython.com/build-recommendation-engine-collaborative-filtering/
3. Recommender Systems in Python - Kaggle, accessed February 20, 2026, https://www.kaggle.com/code/prashant111/recommender-systems-in-python
4. 7 machine learning algorithms for recommendation engines - Lumenalta, accessed February 20, 2026, https://lumenalta.com/insights/7-machine-learning-algorithms-for-recommendation-engines
5. Resolving Cold Start Problem Using User Demographics and Machine Learning Techniques for Movie Recommender Systems - SJSU ScholarWorks, accessed February 20, 2026, https://scholarworks.sjsu.edu/etd_projects/649/
6. Handling Massive Sparse Data in Recommendation Systems - World Scientific Publishing, accessed February 20, 2026, https://www.worldscientific.com/doi/pdf/10.1142/S0219649224500217?download=true
7. Recommender Systems: Similarity Metrics to know | by Julian Werner - Medium, accessed February 20, 2026, https://medium.com/@juliwern/recommender-systems-similarity-metrics-to-know-2844c151639d
8. Applications and differences for Jaccard similarity and Cosine Similarity, accessed February 20, 2026, https://datascience.stackexchange.com/questions/5121/applications-and-differences-for-jaccard-similarity-and-cosine-similarity
9. What is Jaccard Similarity? | IBM, accessed February 20, 2026, https://www.ibm.com/think/topics/jaccard-similarity
10. KLove Does Part of a Recommender System - RPubs, accessed February 20, 2026, https://rpubs.com/KLove/CosineAndJaccard
11. How to solve the cold start problem in recommender systems - Things Solver, accessed February 20, 2026, https://thingsolver.com/blog/the-cold-start-problem/
12. LightFM Hybrid Recommendation system - Kaggle, accessed February 20, 2026, https://www.kaggle.com/code/tariquepce/lightfm-hybrid-recommendation-system
13. Various Implementations of Collaborative Filtering | by Sumeet Agrawal - Medium, accessed February 20, 2026, https://medium.com/@Sumeet_Agrawal/various-implementations-of-collaborative-filtering-7429eec37ab9
14. Alternating Least Squares (ALS) and Bayesian Personalized Ranking (BPR) models for recommendations - data science, accessed February 20, 2026, https://www.stepbystepdatascience.com/alternating-least-squares
15. Deep Dive into Matrix Factorization for Recommender Systems: From Basics to Implementation | by Elie A. | Medium, accessed February 20, 2026, https://medium.com/@eliasah/deep-dive-into-matrix-factorization-for-recommender-systems-from-basics-to-implementation-79e4f1ea1660

16. A Recommender System: Collaborative Filtering with Sparse ..., accessed February 20, 2026, https://superlinked.com/vectorhub/articles/recommender-system-collaborative-filtering-sparse-metadata
17. EASE the Embarrassingly Shallow Autoencoder recommender model - data science, accessed February 20, 2026, https://www.stepbystepdatascience.com/ease
18. A Comparative Study of Recommender Systems under Big Data Constraints - arXiv, accessed February 20, 2026, https://arxiv.org/html/2504.08457v1
19. Embarrassingly Shallow Autoencoders (EASE) for Recommendations | by Jay Franck | TDS Archive | Medium, accessed February 20, 2026, https://medium.com/data-science/embarrassingly-shallow-autoencoders-ease-for-recommendations-f91117f02851
20. Recommender Systems - A Complete Guide to Machine Learning Models, accessed February 20, 2026, https://towardsdatascience.com/recommender-systems-a-complete-guide-to-machine-learning-models-96d3f94ea748/
21. Building a Recommendation System with LightFM | by Murat Yıldırım | Medium, accessed February 20, 2026, https://medium.com/@murattyldrm7/building-a-recommendation-system-with-lightfm-35394c8d90fb
22. A Comparative Study of Collaborative Filtering in Product Recommendation - ResearchGate, accessed February 20, 2026, https://www.researchgate.net/publication/364503381_A_Comparative_Study_of_Collaborative_Filtering_in_Product_Recommendation
23. Personalized Restaurant Recommender System Using A Hybrid Approach, accessed February 20, 2026, https://sites.northwestern.edu/msia/2019/04/24/personalized-restaurant-recommender-system-using-hybrid-approach/
24. Recommendation System in Python: LightFM - Towards Data Science, accessed February 20, 2026, https://towardsdatascience.com/recommendation-system-in-python-lightfm-61c85010ce17/
25. Optimization-driven enhancements in recommendation systems: A computational approach, accessed February 20, 2026, https://www.aimsciences.org/article/doi/10.3934/jimo.2025136
26. Estimating Optimal Weights in Hybrid Recommender Systems, accessed February 20, 2026, https://d-nb.info/1231361441/34
27. Offline Optimization for User-specific Hybrid Recommender Systems - Biblio, accessed February 20, 2026, https://backoffice.biblio.ugent.be/download/6972238/6972244
28. Hybrid Recommender Systems with Surprise - Kaggle, accessed February 20, 2026, https://www.kaggle.com/code/robottums/hybrid-recommender-systems-with-surprise

29. Hybrid Recommendation Systems: Combining Methods for Optimal Performance | Kaggle, accessed February 20, 2026, https://www.kaggle.com/discussions/general/566925
30. A deep learning based hybrid recommendation model for internet users - PMC, accessed February 20, 2026, https://pmc.ncbi.nlm.nih.gov/articles/PMC11599862/
31. How to Optimize Machine Learning Models with Grid Search in Python | Aionlinecourse, accessed February 20, 2026, https://www.aionlinecourse.com/blog/how-to-optimize-machine-learning-models-with-grid-search-in-python
32. How we solve the “cold start problem” in an ML recommendation system - Reddit, accessed February 20, 2026, https://www.reddit.com/r/ProductManagement/comments/1j5rss9/how_we_solve_the_cold_start_problem_in_an_ml/
33. The Cold Start Problem for Recommender Systems | by Mark Milankovich - Medium, accessed February 20, 2026, https://medium.com/@markmilankovich/the-cold-start-problem-for-recommender-systems-89a76505a7
34. Addressing the Cold-Start Problem in Recommender Systems Based on Frequent Patterns, accessed February 20, 2026, https://www.mdpi.com/1999-4893/16/4/182
35. Recommendation systems overview | Machine Learning - Google for Developers, accessed February 20, 2026, https://developers.google.com/machine-learning/recommendation/overview/types
36. MLOps Challenges: 7 Production Problems and How to Fix Them - DEV Community, accessed February 20, 2026, https://dev.to/apprecode/mlops-challenges-7-production-problems-and-how-to-fix-them-5goc
37. AI Recommendation Systems: Fast Real-Time Infrastructure Guide 2026 - Redis, accessed February 20, 2026, https://redis.io/blog/real-time-ai-recommendation-systems/
38. What's the difference between FastAPI background tasks and Celery tasks? - Stack Overflow, accessed February 20, 2026, https://stackoverflow.com/questions/74508774/whats-the-difference-between-fastapi-background-tasks-and-celery-tasks
39. Background Tasks with FastAPI Background Tasks and Celery + Redis - FastAPI Beyond CRUD (Part 20) - YouTube, accessed February 20, 2026, https://www.youtube.com/watch?v=eAHAKowv6hk
40. Celery and Background Tasks. Using FastAPI with long running tasks | by Hitoruna | Medium, accessed February 20, 2026, https://medium.com/@hitorunajp/celery-and-background-tasks-aebb234cae5d
41. Background Tasks with FastAPI Background Tasks and Celery + Redis - FastAPI Beyond CRUD (Part 20) - DEV Community, accessed February 20, 2026, https://dev.to/jod35/background-tasks-with-fastapi-background-tasks-and-celery-redis-fastapi-beyond-crud-part-20-1ilh
42. Building a CRUD FastAPI app with SQLAlchemy - Mattermost, accessed February

20, 2026, https://mattermost.com/blog/building-a-crud-fastapi-app-with-sqlalchemy/
43. Master Prisma ORM: Next.js & FastAPI Setup | Kite Metric, accessed February 20, 2026, https://kitemetric.com/blogs/streamline-your-database-interactions-with-prisma-orm
44. How to use Prisma ORM with multiple database schemas, accessed February 20, 2026, https://www.prisma.io/docs/orm/prisma-schema/data-model/multi-schema
45. FastAPI Best Practices : r/Python - Reddit, accessed February 20, 2026, https://www.reddit.com/r/Python/comments/wrt7om/fastapi_best_practices/
46. Easy Python Web Backend Using FastAPI, Prisma, Postgres, and Docker - AbandonTech, accessed February 20, 2026, https://blog.abandontech.cloud/easy-python-web-backend-using-fastapi-postgres-and-prisma/
47. Deploy fastAPI with Prisma ORM on Vercel - Reddit, accessed February 20, 2026, https://www.reddit.com/r/FastAPI/comments/1fh6prp/deploy_fastapi_with_prisma_orm_on_vercel/
48. Read only schema · prisma prisma · Discussion #19512 - GitHub, accessed February 20, 2026, https://github.com/prisma/prisma/discussions/19512
49. Automap — SQLAlchemy 1.4 Documentation, accessed February 20, 2026, https://docs.sqlalchemy.org/14/orm/extensions/automap.html
50. sqlalchemy existing database query - python - Stack Overflow, accessed February 20, 2026, https://stackoverflow.com/questions/39955521/sqlalchemy-existing-database-query
51. Help accessing views from a previously existing database using SQLAlchemy - Reddit, accessed February 20, 2026, https://www.reddit.com/r/learnpython/comments/11os2qy/help_accessing_views_from_a_previously_existing/
52. PostgreSQL — SQLAlchemy 2.1 Documentation, accessed February 20, 2026, http://docs.sqlalchemy.org/en/latest/dialects/postgresql.html
53. How to use READ ONLY transaction mode in SQLAlchemy? - Stack Overflow, accessed February 20, 2026, https://stackoverflow.com/questions/25904020/how-to-use-read-only-transaction-mode-in-sqlalchemy
54. Concurrency and async / await - FastAPI, accessed February 20, 2026, https://fastapi.tiangolo.com/async/
55. Understanding benchmarking and setting up Async (Postgres+asyncpg vs Postgres+psycopg2) in FastAPI #13732 - GitHub, accessed February 20, 2026, https://github.com/fastapi/fastapi/discussions/13732
56. FinTech Research Center | UF Warrington, accessed February 20, 2026, https://warrington.ufl.edu/faculty-research/research-centers/fintech-center/
57. A Comparative Study of Recommender Systems under Big Data Constraints, accessed February 20, 2026, https://www.engineegroup.com/articles/TCSIT-10-199.php

58. UFIT-RC Documentation - University of Florida, accessed February 20, 2026, https://docs.rc.ufl.edu/
59. What are the best practices when it comes to applying complex algorithms in data pipelines?, accessed February 20, 2026, https://www.reddit.com/r/dataengineering/comments/1o0rhhv/what_are_the_best_practices_when_it_comes_to/
60. SLURM Scheduler | Gator-AIM's Hub, accessed February 20, 2026, https://gatoraim.com/docs/research/hipergator/hipergator_scheduler/
61. Quick Start User Guide - Slurm Workload Manager - SchedMD, accessed February 20, 2026, https://slurm.schedmd.com/quickstart.html
62. Reoccurring Jobs - UFIT-RC Documentation - University of Florida, accessed February 20, 2026, https://docs.rc.ufl.edu/scheduler/scrontab/
63. Sample Slurm Scripts - UFIT-RC Documentation - University of Florida, accessed February 20, 2026, https://docs.rc.ufl.edu/scheduler/sample_job_scripts/
64. 'Running jobs on HiPerGator' - HOME - Natya Hans, accessed February 20, 2026, https://natyahans.github.io/posts/2019/05/RunningJobsHiPerGator/
65. Slurm - Princeton Research Computing, accessed February 20, 2026, https://researchcomputing.princeton.edu/support/knowledge-base/slurm
66. HiPerGator: SLURM Scripts for MPI Jobs - Information Technology - University of Florida, accessed February 20, 2026, https://it.ufl.edu/training/offerings/items/hipergator-slurm-scripts-for-mpi-jobs.html
67. Storage Resources - University of Florida Information Technology, accessed February 20, 2026, https://it.ufl.edu/rc/about/our-services/storage-resources/
68. Practical Storage Use - UFIT-RC Documentation - University of Florida, accessed February 20, 2026, https://docs.rc.ufl.edu/quickstart/practical_storage/
69. Choosing storage for deep learning: a comprehensive guide, accessed February 20, 2026, https://nebius.com/blog/posts/choosing-storage-for-deep-learning
70. General FAQ - UFIT-RC Documentation - University of Florida, accessed February 20, 2026, https://docs.rc.ufl.edu/support/faq/
71. Best Practices for Jobs - NERSC Documentation, accessed February 20, 2026, https://docs.nersc.gov/jobs/best-practices/
72. HiPerGator Usage Policies - University of Florida Information Technology, accessed February 20, 2026, https://it.ufl.edu/rc/documentation/policies/hipergator-usage-policies/
73. 10 metrics to evaluate recommender and ranking systems - Evidently AI, accessed February 20, 2026, https://www.evidentlyai.com/ranking-metrics/evaluating-recommender-systems
74. Evaluation Metrics for Search and Recommendation Systems | Shaped Blog, accessed February 20, 2026, https://www.shaped.ai/blog/evaluation-metrics-for-search-and-recommendation-systems
75. Evaluation Metrics for Search and Recommendation Systems ..., accessed February 20, 2026, https://weaviate.io/blog/retrieval-evaluation-metrics
76. Evaluation Metrics for Recommendation Systems | by Nick McCarthy - Medium,

accessed February 20, 2026, https://medium.com/@mccartni/evaluation-metrics-for-recommendation-systems-dfd7f93441c1
77. A Comprehensive Survey of Evaluation Techniques for Recommendation Systems - arXiv, accessed February 20, 2026, https://arxiv.org/html/2312.16015v2