# Game publishing site

## **Project description**

Project will design a Game publishing site (like Steam) GPS (Game publishing site).It’s a publishing and social marketplace. The main inspiration was public marketplace Steam but also platforms like GOG, Epic Games Store. GPS will allow users to buy and review games. Create Game Studios and Publishers. Adding tags and genres to the created games.

## **Roles**

The primary role of users is the **PLAYER**. A Player can search for, purchase, add games to their wishlist, and write reviews. They can also join a game studio or act as a publisher.

## **Objects**

* **Game Studio** - An organisation consisting of users allowing them to manage developed games.
* **Publisher** - An organisation consisting of users allowing them to publish, tag and manage published games.
* **Review** - user's written opinion about the game. Each review has a timeStamp when it was published and other users can vote whether the review is helpful or not.
* **Score** - numerical score given by the user to the games based on his subjective judgement of how good the game is. Average of multiple scores gives the game score that determines how much the game will be recommended to the other users
* **Tag** - short label describing feature of a game (eg. Console compatible, german localization, accessible for disabled). Tag can be added just by the Publisher of the game.
* **Genre** - describes genre of a game for an easier classification of a game (eg. horror, multiplayer, platformer, puzzle). Genres can be added just by the Publisher of the game.
* **Wishlist** - a user’s collection of games he is planning on buying or is interested in.
* **Achievement** - achievement unlocked by a player for completing specific tasks or goals within a game.
* **Game View** - number of views of a game by users
* **Game Activity** - describes how much time has user spend on a game

## **Events and actions**

* User bought a game
* User wrote a review
* User gave a score to a game
* User wishlisted a game
* User created a Publishing Studio
* User added other user to the Publishing Studio
* User completed an Achievement
* User created a Game Studio
* User added other user to the Game Studio
* Publisher added a tag to the game
* Publisher added a genre to the game

## **ER Diagram**

## 

## 

## 

## **Analytical tasks**

Task 1 - Select top 100 most active users for last 7 days

* Business goal: Identify highly engaged players to target retention and reward programs.
* Input: Game Activity records (user\_id, number\_of\_hours).
* Output: user\_id, username, total playtime (hours).
* Granularity: By user.
* Sorting / top N: Top 100 by total playtime.
* Execution frequency: Run weekly.
* Acceptance criterion: Exactly 100 users with the highest total playtime in the last 7 days.

Task 2 - Calculate average number of purchased games per user

* Business goal: Understand overall player engagement by measuring how many games users typically purchase.
* Input: User (user\_id, games).
* Output: average\_number\_of\_games.
* Granularity: By user..
* Execution frequency: Run monthly.
* Acceptance criterion: A single numeric value representing the average number of purchased games per user, computed across all active users.

Task 3 - Select top 50 most played games globally in the last 30 days

* Business goal: Identify the most played games across all users for popularity insights.
* Input: Game Activity records (game\_id, number\_of\_hours).
* Output: game\_id, game title, total playtime, number of unique players.
* Granularity: By game.
* Sorting / top N: Top 50 by total playtime.
* Execution frequency: Run monthly.
* Acceptance criterion: Exactly 50 games returned, ordered by descending total playtime.

Task 4 - Find top-rated games by score per genre in the last month

* Business goal: Identify the highest-rated games per genre for monthly promotion and recommendation.
* Input: Score records (game\_id, value, created), Genre list for each game.
* Output: game\_id, game title, score of game, total number of ratings, genre.
* Granularity: By game.
* Sorting / top N: Games per genre sorted by their score.
* Execution frequency: Run monthly.
* Acceptance criterion: Top-rated game per genre.

Task 5 - Get top 20 games with the highest total number of wishlists.

* Business goal: Detect trending games based on wishlist.
* Input: Wishlist (games)
* Output: game\_id, game title, total\_wishlist\_task.
* Granularity: By game.
* Sorting / top N: Top 20 games by total wishlist count.
* Execution frequency: Run weekly.
* Acceptance criterion: Exactly 20 games returned, ranked by total wishlist count (descending).

Task 6 - Select most active game studios (studios that developed the most games) in the last 10 years

* Business goal: Surface studios with highest production activity or portfolio size for partnership/outreach.
* Input: Game (game\_studio\_id, release\_date).
* Output: studio\_id, studio\_name, number of games developed.
* Granularity: By game studio.
* Sorting / top N: Top 20 studios by number of games developed.
* Execution frequency: Run yearly.
* Acceptance criterion: Exactly 20 studios with the largest counts of released games in the last 10 years.

Task 7 - Get average score per publisher

* Business goal: Measure the average rating of games published by each publisher to assess quality.
* Input: Score records (game\_id, value), Game (publishing\_studio\_id).
* Output: publisher\_id, publisher\_name, average rating of all games.
* Granularity: By publisher.
* Sorting / top N: Sorted by average rating.
* Execution frequency: Run quarterly.
* Acceptance criterion: Each publisher has one aggregated rating value.

Task 8 - Find top-50 users with diverse gaming habits

* Business goal: Identify users who play across many genres.
* Input: Game Activity (game\_id, user\_id, number\_of\_hours), Game (genres)
* Output: user\_id, username, number of distinct genres played.
* Granularity: By user.
* Sorting / top N: Top 50 by number of genres.
* Execution frequency: Run monthly.
* Acceptance criterion: Exactly 50 users with the highest genre diversity.

Task 9 - Select top-20 games with most reviews in the last month

* Business goal: Identify games with the highest number of reviews.
* Input: Review (game\_id, created).
* Output: Game title, number of reviews.
* Granularity: By game.
* Sorting / top N: Top 20 by total reviews.
* Execution frequency: Run monthly.
* Acceptance criterion: Exactly 20 games with the highest review counts in the last month.

Task 10 - Get top-50 users with the most achievements in the selected game.

* Business goal: Recognize highly engaged players per game for social features.
* Input: Achievement (user\_id, game\_id, name)
* Output: user\_id, username, game\_id, number of achievements unlocked.
* Granularity: By user within a game.
* Sorting / top N: Top 50 users by achievements in the selected game.
* Execution frequency: Run on-demand.
* Acceptance criterion: Exactly 50 users per selected game, ordered by unlocked achievements.

Task 11 - Find top-30 largest game studios by number of developers (users)

* Business goal: Identify studios with the largest developer teams for B2B engagement and community building.
* Input: Game Studio (users list).
* Output: studio\_id, studio\_name, number of developer users.
* Granularity: By studio.
* Sorting / top N: Top 30 studios by number of developer users.
* Execution frequency: Run on demand.
* Acceptance criterion: Exactly 30 studios with the most developers.

Task 12 - Select top-10 favourite genre of game in the last two weeks

* Business goal: Spot short-term genre popularity to inform time-sensitive promotions.
* Input: Game Activity (game\_id, number\_of\_hours), Game (genres)
* Output: genre, total playtime, number of unique players
* Granularity: By genre.
* Sorting / top N: Top 10 ranked by total playtime.
* Execution frequency: Run every 14 days.
* Acceptance criterion: Exactly 10 genres returned, ordered by playtime.

Task 13 - Get top-20 most expensive games that were added during the last year.

* Business goal: Identify high-priced new releases for premium marketing and revenue forecasting.
* Input: Game (price, release\_date, publishing\_studio\_id, game\_studio\_id)
* Output: game\_id, game title, price, release date, publisher studio, game studio.
* Granularity: By game.
* Sorting / top N: Top 20 by highest price.
* Execution frequency: Run monthly.
* Acceptance criterion: Exactly 20 games with highest prices among last year’s releases.

Task 14 - Count daily views of a selected game for the last 30 days

* Business goal: Track player interest/visibility trends for a game.
* Input: Game view (game\_id, user\_id)
* Output: game\_id, game title, daily view count,l.
* Granularity: By day.
* Execution frequency: Run daily.
* Acceptance criterion: 30 count of daily views for the selected game.

Task 15 - Find top-100 users who added most games to their wishlist during the last 7 days.

* Business goal: Identify users actively discovering and saving titles for targeting recommendations and marketing.
* Input: Wishlist, User
* Output: user\_id, username, number of wishlist adds.
* Granularity: By user.
* Sorting / top N: Top 100 users by wishlist additions in the period.
* Execution frequency: Run weekly.
* Strength: Good horizontal scaling, fast counting and aggregation, persistent
* Weakness of other solutions:
  + MongoDB- slower for large data, lot of docs scans
  + Redis- not persistent
* Acceptance criterion: Exactly 100 users, sorted by wishlist additions.

Task 16 - Get top-100 most active reviewers in the last month

* Business goal: Identify users who write the most reviews to support community engagement programs.
* Input: Review records (user\_id, created).
* Output: user\_id, username, number of reviews written.
* Granularity: By user.
* Sorting / top N: Top 100 reviewers by count.
* Execution frequency: Run monthly.
* Strength: Quick aggregation in given time range, good write throughput
* Weakness of other solutions:
  + MongoDB- slower for large data, lot of docs scans
  + Redis- not persistent, not good for monthly analytics
* Acceptance criterion: Exactly 100 users with the most reviews created in the period.

Task 17 - Get top-50 tags that were used frequently in the last year.

* Business goal: Discover which tags are most often applied to games.
* Input: Game (tags list).
* Output: tag\_name, number of occurrences.
* Granularity: By tag.
* Sorting / top N: Top 50 tags by frequency.
* Execution frequency: Run yearly.
* Strength: Flexible in case of redesign, good at storing complex objects, strong tools for aggregation and filtering
* Weakness of other solutions:
  + Cassandra - not flexible, limited filtering and queries
  + Redis - could support pre-computed tags, limited search and filtering
* Acceptance criterion: Exactly 50 most frequent tags returned.

Task 18 - Count the number of newly registered users in the last year.

* Business goal: Measure user acquisition performance over the year.
* Input: User (registration\_date)
* Output: Total new users.
* Granularity: Aggregated (global).
* Execution frequency: Run yearly.
* Strength: Pretty good handling of time-based requests and counts, good throughput
* Weakness of other solutions:
  + MongoDB - slow with large data (cassandra is much more optimal)
  + Redis - not persistent
* Acceptance criterion: Single aggregated value representing count of new users in the year.

Task 19 - Top-10 most helpful reviews in the selected game in the last half-year

* Business goal: Surface influential reviews for highlighting and moderation.
* Input: Review (game\_id, helpful\_votes)
* Output: review\_id, review content, helpful votes counts.
* Granularity: By review (within a game).
* Sorting / top N: Top 10 by helpful votes (tie-breaker: recency or upvote ratio).
* Execution frequency: Run on-demand.
* Strength: Secondary indexes eg. for likes, powerful filtering and aggregation, complex object stores
* Weakness of other solutions:
  + Cassandra - Bad at filtering by non-key value
  + Redis - weak filter, search capability, not persistent (would need to be pre-computed)
* Acceptance criterion: 10 reviews for the selected game, sorted by helpfulness.

Task 20 - Recommend games by similar users (graph)

* Business goal: Recommend games to a user based on preferences of other users with similar game owning patterns, prioritizing games frequently owned by those similar users.
* Input: User identifier.
* Output: List of recommended games with: game\_id, game\_name, similar\_user\_count (number of similar users who own the game) .
* Granularity: By game.
* Sorting / top-N: Top 30 games sorted by similar\_user\_count (descending).
* Execution frequency: Daily
* Strength: Its a graph with significant information formed by relation between two objects… Perfect fit for Graph based DB
* Weakness of other solutions:
  + Cassandra - would need additional data stored to compute similarity of the games and players
  + MongoDB - Same as Cassandra… Potential high number of scans for a single recommendation
* Acceptance criterion: Each recommended game is not already owned or wishlisted by the target user; Recommendations come from users with at least one overlapping game with the target user.

## **Data structure**

#### **User**

**Fields:**

* user\_id – stable identifier.
* username – display name of the user.
* email - email of the user.
* password - password of the user.
* registration\_date - the date when the user signed the application.
* country – user’s country of residence.
* games - list of game\_id which the user owns.

**Identifiers:**

* user\_id – unique across users; username is not unique.
* email - unique across users (case-insensitive)

**Cardinalities:**

* base ≥ 10k users; stretch ≥ 100k users.

**Distribution:**

* Names: May include duplicates across users.
* Countries: User distribution follows general platform demographics (majority from top 10 countries).

#### **Game**

**Fields:**

* game\_id – stable identifier.
* name – title of the game.
* price – official price in the marketplace.
* achievements – list of achievements unlockable by players.
* release\_date – official release date.
* supported\_languages – list of supported interface or subtitle languages.
* requirements – minimal and recommended system requirements.
* tags - list of tags.
* genres - list of genres.
* game\_studio\_id - game studio which developed the game.
* publishing\_studio\_id - publishing studio which released the game.
* reviews - list of review\_ids that belong to the game.
* score - list of score\_ids that belong to the game.

**Identifiers:**

* game\_id – unique across games; name is not unique.

**Cardinalities:**

* base ≥ 5k games; stretch ≥ 50k games.

**Distribution:**

* Genres: Most games have 1–5 genres, some up to 8.
* Tags: Most games have 0–10 tags, a few up to 20.
* Release dates: Most games released within the past decade.
* Price: Follows right-skewed distribution (many low-cost or free games).

#### **Achievement**

**Fields:**

* achievement\_id – stable identifier.
* game\_id – identifier of the game being played.
* users - list of user\_ids of players who completed the achievement.
* name – name of the achievement.

**Identifiers:**

* achievement\_id – unique across all achievements.

**Cardinalities:**

* base ≥ 50k achievements; stretch ≥ 100k achievements.

**Distribution:**

* Frequency: A small number of users generate most game achievement (top 20% users account for ~60% achievements).

#### **Game View**

**Fields:**

* game\_view\_id – stable identifier.
* game\_id – identifier of the game being played.
* user\_id – identifier of the user who played the game
* viewed - timestamp of the view
* number\_of\_views - how many times user viewed the game

**Identifiers:**

* game\_view\_id – unique across all game views.

**Cardinalities:**

* base ≥ 100k views; stretch ≥ 500k views.

**Distribution:**

* Frequency: A small number of games generate most game view (top 15% games for ~80% views).

#### **Game Activity**

**Fields:**

* activity\_id – stable identifier.
* game\_id – identifier of the game being played.
* user\_id – identifier of the user who played the game.
* number\_of\_hours – total number of hours the user spent playing the game.

**Identifiers:**

* activity\_id – unique across all game activity records.

**Cardinalities:**

* base ≥ 200k activity records; stretch ≥ 500k activity records.

**Distribution:**

* Duration: Most sessions last 50–100 hours; a few exceed 1000 hours.
* Frequency: A small number of users generate most activity (top 10% users account for ~70% of total hours).
* Coverage: Popular games appear in many activity records; niche titles in very few.

#### **Game Studio**

**Fields:**

* game\_studio\_id – stable identifier.
* name – studio name.
* description – short text describing the studio and its focus.
* users - list of user\_ids in game studio.

**Identifiers:**

* gameStudio\_id – unique across studios; name is not unique.

**Cardinalities:**

* base ≥ 1k studios; stretch ≥ 10k studios.

**Distribution:**

* Size: Most studios have 1–10 members.
* Output: Most studios develop 1–5 games; few large studios exceed 20.

#### **Publishing Studio**

**Fields:**

* publishing\_studio\_id – stable identifier.
* name – publisher name.
* description – short text describing the publisher and its activity.
* users - list of user\_ids in publishing studio.

**Identifiers:**

* publishing\_studio\_id – unique across publishers; name is not unique.

**Cardinalities:**

* base ≥ 500 publishers; stretch ≥ 5k publishers.

**Distribution:**

* Size: Most publishers have fewer than 20 games.
* Popularity: Top 20% of publishers generate about 60% of total game sales.

#### **Genre**

**Fields:**

* name – name of the genre (e.g., horror, strategy, platformer).
* description - description of the genre

**Identifiers:**

* name – unique across genres.

**Cardinalities:**

* base 20–40 genres; stretch ≤ 70 genres.

**Distribution:**

* Popularity: A small subset (around 15 genres) covers most games.

#### **Tag**

**Fields:**

* name – tag keyword describing a feature or property of a game (e.g., “multiplayer”, “VR-supported”).
* description - description of the tag

**Identifiers:**

* name – unique across tags.

**Cardinalities:**

* base ≥ 500 tags; stretch ≥ 5k tags.

**Distribution:**

* Most tags apply to 10–100 games; few generic tags appear across thousands of titles.

#### **Review**

**Fields:**

* review\_id – stable identifier.
* created – timestamp of review creation.
* user\_id – identifier of the review’s author.
* game\_id – identifier of the reviewed game.
* content – written text of the review.
* helpful\_votes – number of users who found the review helpful.

**Identifiers:**

* review\_id – unique across reviews; each user can post only one review per game

**Cardinalities:**

* base ≥ 100k reviews; stretch ≥ 1M reviews.

**Distribution:**

* Review length: Most reviews are short (under 300 words).

#### **Score**

**Fields:**

* score\_id – stable identifier.
* created – timestamp when score was assigned.
* user\_id – identifier of the scoring user.
* game\_id – identifier of the scored game.
* value – numerical rating (0–10 scale).

**Identifiers:**

* score\_id – unique across scores; each user can assign at most one score per game.

**Cardinalities:**

* base ≥ 500k scores; stretch ≥ 5M scores.

**Distribution:**

* Values: Ratings follow a normal-like distribution centered around 7–8.
* Activity: Heavy users contribute 60% of all scores.

#### **Wishlist**

**Fields:**

* wishlist\_id – stable identifier.
* user\_id – wishlist owner.
* visibility – “private” or “public”.
* games – list of games added to the wishlist.

**Identifiers:**

* wishlist\_id – unique across wishlists; each user has one wishlist (can be empty).

**Cardinalities:**

* base ≥ 10k wishlists; stretch ≥ 100k wishlists.

**Distribution:**

* Size: Most wishlists contain 1–10 games; few exceed 50.
* Visibility: Roughly 30% of wishlists are private.

## **Dataset**

* Generates a synthetic Game Publishing Site (GPS) dataset according to the data structure.
* Code for data generation was generated by chatgpt.

#### **Output files:**

* **users.jsonl** - list of user records containing IDs, usernames, emails, passwords, registration dates, and country of residence. Each user record also includes an initially empty list of owned games.
* **games.jsonl** - list of games with identifiers, names, prices, release dates, supported languages, requirements, genres, tags, associated studios and publishers, and lists of related reviews, scores, and achievements.
* **achievements.jsonl** - list of game achievements, each with an ID, game reference, list of users who completed it, and achievement name.
* **game\_views.jsonl** - user–game view interactions including the game and user identifiers, timestamp of the view, and number of views per record.
* **activities.jsonl** - gameplay activity records specifying how many hours a user has spent playing a given game.
* **studios.json** - list of game studios with identifiers, names, descriptions, and associated developer user IDs.
* **publishers.json** - list of publishing studios with identifiers, names, descriptions, and member user IDs.
* **genres.json** - genre definitions with names and descriptions (e.g., Action, RPG, Simulation).
* **tags.json** - keyword tags describing features of games (e.g., Multiplayer, Story-rich, VR-supported).
* **reviews.jsonl** - review entries including IDs, creation timestamps, user and game identifiers, review content, and helpful-vote counts.
* **scores.jsonl** - user rating entries linking users to games with numeric scores on a 0–10 scale and timestamps.
* **wishlists.jsonl** - wishlist data where each record represents a user’s wishlist with ID, visibility flag (public/private), and list of added games.

#### **Generator parameters:**

* **‘--seed’** - random seed for reproducibility (example: 20250927).
* **’--n-users’** - number of users to generate (base: 10 000, stretch: 100 000).
* **’--n-games’** - number of games to generate (base: 5 000, stretch: 50 000).
* **’--n-achievements’** - number of achievements to generate (base: 50 000, stretch: 100 000).
* **’--n-game-views’** - total number of game view records (base: 100 000, stretch: 500 000).
* **’--n-activities’** - total number of game activity records (base: 200 000, stretch: 500 000).
* **’--n-studios’** - number of game studios to generate (base: 1 000, stretch: 10 000).
* **’--n-publishers’** - number of publishing studios to generate (base: 500, stretch: 5 000).
* **’--n-reviews’** - number of review records to generate (base: 100 000, stretch: 1 000 000).
* **’--n-scores’** - number of score (rating) records to generate (base: 500 000, stretch: 5 000 000).
* **’--outdir’** - output directory path (default: ./out\_gps). Specifies where the generated dataset files will be stored.

#### **Reproducibility:**

python generate\_gps\_dataset.py \

--seed 20250927 \

--n-users 10000 \

--n-games 5000 \

--n-achievements 50000 \

--n-game-views 100000 \

--n-activities 200000 \

--n-studios 1000 \

--n-publishers 500 \

--n-reviews 100000 \

--n-scores 500000 \

--outdir ./out\_gps

# **HW1**

## **NOSQL databases for each task**

### **Task 1 - Top 100 most active users (last 7 days)**

#### **Primary DBMS: Redis**

**Fast updates and lookups:**

* Keys (user\_id) are known in advance, so Redis can perform direct updates (INCRBY) and lookups without scanning or filtering.

**Sliding 7-day windows using sorted sets (ZSET):**

* Redis sorted sets allow storing activity scores per day and maintain order automatically. Retrieving the top-N users for the last 7 days is instant.

**Low overhead:**

* No complex joins or aggregations - just counters per user.

#### **Alternative: Cassandra**

**Per-user, time-ordered storage:**

* The query always targets one user and a short, recent time window. A wide-column layout clusters all events for a user by timestamp, so a 7-day slice is a single contiguous read rather than many scattered lookups.

**How it would work:**

* Store playtime events partitioned by user\_id and date, with running totals updated daily for each user.

**When to choose it:**

* When handling massive data volumes and long-term historical tracking is also required.

**Trade-off:**

* Computing top-N across all users is slower than Redis because it requires aggregation across partitions.

#### **Alternative 2: MongoDB**

**How it would work:**

* Filter by user and the last 7 days, then sort. Use an aggregation pipeline to sum total playtime per user.

**When to choose it:**

* When analytical flexibility is important - e.g., filtering by region, platform, or user type.

**Trade-off:**

* Aggregation queries are slower for frequent leaderboard updates compared to Redis.

#### **Why others are weaker**

* Neo4j: Graph traversal unnecessary.

#### **Expected Output Format**

* List of users: user\_id, username, total\_playtime (hours)

#### **Freshness / Latency Goal**

* Daily; response time of milliseconds.

#### **Access Pattern**

* Increment counters by user\_id during activity.
* Periodically compute top 100 from ZSET or pre-aggregated counters.

### **Task 2 - Calculate average number of purchased games per user**

#### **Primary DBMS: MongoDB**

**Simple aggregation:**

* MongoDB can easily get the size of each user's games array and group to compute the average across all users.

**Flexible schema:**

* Easy to extend with filters (e.g., active users only, by region or platform) without altering storage.

**Monthly batch-suited:**

* Monthly calculation fits MongoDB’s on-demand aggregation without needing precomputed counters.

#### **Alternative: Cassandra**

**How it would work:**

* For each user, maintain a counter column (purchase\_count) that increments whenever a purchase occurs. To compute the global average, scan all rows to sum counts and divide by total users.

**When to choose it:**

* When the dataset will be massive and write operations dominate read operations. Suitable if only numeric aggregates rather than lists of purchased items is needed.

**Trade-off:**

* Calculating the average requires scanning all partitions, which is slower and less convenient than MongoDB’s direct aggregation.

#### **Why others are weaker**

* Redis: Key-value model isn’t efficient for scanning all users or computing global aggregates.
* Neo4j: Graph traversal unnecessary

#### **Expected Output Format**

* Non-negative integer: average\_number\_of\_games

#### **Freshness / Latency Goal**

* Weekly; response time of seconds.

#### **Access Pattern**

* Read all user documents → compute number of games per user → aggregate to get the average.

### **Task 3 - Top 50 most played games (last 30 days)**

#### **Primary DBMS: Cassandra**

**Time-series efficiency:**

* Well-suited for storing and querying large volumes of activity records partitioned by game\_id and date.

**High write throughput:**

* Handles continuous game activity inserts without performance degradation.

**Pre-aggregated counters:**

* Maintain per-game totals of playtime and unique players during ingestion for fast monthly reads.

#### **Alternative: MongoDB**

**How it would work:**

* Store each game activity as a document, then use an aggregation pipeline to group by game\_id, sum total playtime for the last 30 days, and sort descending to get the top 50.

**When to choose it:**

* When analytical flexibility is important, such as filtering by genre, platform, or user segments.

**Trade-off**:

* Scanning and aggregating large collections is slower than Cassandra’s precomputed counters but allows more adaptable analytics.

#### **Why others are weaker**

* Redis: Lacks filtering over large historical windows.
* Neo4j: Graph traversal unnecessary.

#### **Expected Output Format**

* List of 50 games: game\_id, game\_title, total\_playtime, unique\_players

#### **Freshness / Latency Goal**

* Monthly batch; latency of seconds acceptable .

#### **Access Pattern**

* Aggregate all playtime per game\_id for the last 30 days.
* Sort and return top 50.

### **Task 4 - Top-rated games per genre (last month)**

#### **Primary DBMS: MongoDB**

**Aggregation with joins:**

* MongoDB can join Scores with Games to access genres and compute averages per game.

**Multi-level grouping:**

* group by game\_id (to compute average score) and again by genre (to find top-rated games per genre) handled efficiently in one pipeline.

**Monthly batch-friendly:**

* Runs monthly, so MongoDB’s on-demand aggregation is well-suited and avoids maintaining live counters.

#### **Alternative: Cassandra**

**How it would work:**

* Maintain per-game counters for total score and rating count during ingestion, partitioned by genre and month.

**When to choose it:**

* When data volume is very high and per-genre, per-month reports are generated regularly with a fixed structure.

**Trade-off:**

* Adding new filters or ranking criteria would require new tables or schema changes; less flexible than MongoDB’s ad-hoc aggregations.

#### **Why others are weaker**

* Redis: Cannot filter by attributes like genre.
* Neo4j: Graph traversal unnecessary.

#### **Expected Output Format**

* List of genres with games: genre, game\_id, game\_title, avg\_score, number\_of\_ratings

#### **Freshness / Latency Goal**

* Monthly; response time of seconds.

#### **Access Pattern**

* Filter scores by last month.
* Join with game → genre.
* Group by genre.

### **Task 5 - Top 20 games by total wishlists**

#### **Primary DBMS: Redis**

**Fast leaderboard updates:**

* Each wishlist event increments a sorted set score, keeping counts accurate in real time.

**Low latency:**

* In-memory operations ensure sub-second reads and writes, ideal for frequently updated popularity metrics.

**Simplicity:**

* No complex joins or aggregations - just counters per game\_id.

#### **Alternative: MongoDB**

**How it would work:**

* Aggregate wishlist documents weekly using group by game\_id and count to compute totals.

**When to choose it:**

* When updates are less frequent (e.g., weekly/monthly dashboards) and exact real-time ranking isn’t required.

**Trade-off:**

* Aggregations are slower than Redis for continuous updates, but MongoDB provides more flexibility to include extra dimensions (e.g., genre or region).

#### **Why others are weaker**

* Cassandra: Can store counters per game, but sorting the top 20 globally is inefficient without an additional aggregation service.
* Neo4j: Graph traversal unnecessary.

#### **Expected Output Format**

* List of games: game\_id, game\_title, total\_wishlist\_count

#### **Freshness / Latency Goal**

* Daily; Latency of milliseconds acceptable.

#### **Access Pattern**

* Increment counter when wishlist event occurs.
* Retrieve top 20 via ZSET.

### **Task 6 - Most active studios (last 10 years)**

#### **Primary DBMS: Cassandra**

**Optimized for time-based queries:**

* Easily stores and queries large historical datasets filtered by release year.

**High ingestion performance:**

* Suited for continuous addition of game records over many years.

**Yearly Batch-friendly:**

* Yearly execution favors simple read queries from precomputed counters.

#### **Alternative: MongoDB**

**How it would work:**

* Run an aggregation pipeline filtering games from the last 10 years ($match on release\_date) and grouping by studio\_id to count total games.

**When to choose it:**

* When the schema or filters may evolve - e.g., adding genre, region without schema redesign.

**Trade-off:**

* Full collection scans are slower on large datasets, and aggregation performance decreases as data grows compared to Cassandra’s pre-aggregated counters.

#### **Why others are weaker**

* Redis: Poor for long-term aggregates.
* Neo4j: No relationships between studios and games needed.

#### **Expected Output Format**

* List of game studios: studio\_id, studio\_name, number\_of\_games

#### **Freshness / Latency Goal**

* Yearly; latency of minutes acceptable.

#### **Access Pattern**

* Filter by release\_date ≥ (current\_year−10).
* Group by studio\_id.
* Sort and return top 20.

### **Task 7 - Average score per publisher**

#### **Primary DBMS: MongoDB**

**Join-friendly aggregation:**

* MongoDB can join Scores with Games to access publisher info directly inside the aggregation pipeline.

**Flexible analytics:**

* Easy to extend the same pipeline with new dimensions without schema changes.

**Quarterly Batch-friendly:**

* Runs quarterly, so MongoDB’s on-demand aggregation is fast enough without maintaining live counters.

#### **Alternative: Cassandra**

**Ingestion-time aggregation:**

* Maintain per-publisher counters (rating\_sum, rating\_count) during writes for instant lookup later.

**Best for high throughput:**

* Chosen when rating volume is massive and report structure (per-publisher averages) is fixed.

**Trade-off:**

* Adding new breakdowns would require new tables or ingestion logic due to rigid schema.

#### **Why others are weaker**

* Redis: No joins, poor for long-term aggregates.
* Neo4j: Graph traversal unnecessary.

#### **Expected Output Format**

* List of publishers: publisher\_id, publisher\_name, average\_rating

#### **Freshness / Latency Goal**

* Quarterly; seconds acceptable.

#### **Access Pattern**

* Join scores with games to find publisher.
* Group by publisher\_id.
* Compute average.

### **Task 8 - Find top-50 users with diverse gaming habits**

#### **Primary DBMS: MongoDB**

**Flexible aggregation:**

* Combines user activity (Game Activity) with game genre data in a single aggregation pipeline.

**Distinct counting:**

* MongoDB is able to compute the number of unique genres each user has played without manual deduplication.

**Top-N ranking:**

* Supports sorting and limiting in query to produce the top-50 users efficiently.

#### **Alternative: Cassandra**

**How it would work:**

* There would be per user genre set or counter. Query precomputed counts and sort top-50 offline.

**When to choose it:**

* When the system receives high volume of continuous game activity events and requires rapid writes.

**Trade-off:**

* Less flexible - schema changes are required for new metrics or filters.

#### **Why others are weaker**

* Redis: Good for caching precomputed top-50 results, but cannot compute distinct genre counts directly.
* Neo4j: Graph traversal unnecessary.

#### **Expected Output Format**

* List of 50 users: user\_id, username, number of distinct genres played

#### **Freshness / Latency Goal**

* Monthly, seconds - minutes.

#### **Access Pattern**

* Game Activity -> join with game genres -> count distinct genres per user -> sort top-50

### **Task 9 - Select top-20 games with most reviews in the last month**

#### **Primary DBMS: MongoDB**

**Flexible aggregation:**

* Filters reviews by date and groups by game\_id in a single aggregation pipeline.

**Count and top-N ranking:**

* Counts the number of reviews per game and sorts descending to produce the top-20 games.

#### **Alternative: Cassandra**

**How it would work:**

* Maintain per-game counters for each review at ingestion. Query the counters for the last month and sort to determine top-20 games.

**When to choose it:**

* Chosen when the system handles a very high volume of review writes and requires fast ingestion with predictable queries.

**Trade-off:**

* Less flexible when changing reporting periods compared to MongoDB.

#### **Why others are weaker**

* Redis: Only for serving precomputed top-20 results. Cannot compute aggregates directly from raw review data.
* Neo4j: Graph traversal unnecessary.

#### **Expected Output Format**

* List of 20 games: game title, number of reviews

#### **Freshness / Latency Goal**

* Monthly, seconds - minutes.

#### **Access Pattern**

* Filter review records by created date -> group by game\_id -> count reviews -> sort descending -> select top-20

### **Task 10 - Top-50 users with the most achievements in the selected game.**

#### **Primary DBMS: Redis**

**Instant leaderboard updates:**

* Each achievement event increments a sorted set score per user per game, keeping counts accurate in real time.

**Low latency:**

* In-memory operations ensure sub-millisecond reads and writes, ideal for on-demand queries.

**No complex joins or aggregations:**

* No complex joins or aggregations. Maintain counters per (user\_id, game\_id) pair and query the top-50 directly

#### **Alternative: MongoDB**

**How it would work:**

* Aggregate achievement documents for the selected game and group by user\_id to count achievements.

**When to choose it:**

* Good for on-demand analytics where slight latency is acceptable and where the dataset is of moderate size.

**Trade-off:**

* Aggregations are slower than Redis for real-time queries, but MongoDB allows flexible filtering or adding extra parameters.

#### **Why others are weaker**

* Cassandra: Can store counters per user per game but requires precomputing and additional steps to sort globally for top-50, less flexible for on-demand queries.
* Neo4j: Graph traversal unnecessary.

#### **Expected Output Format**

* List of 50 users: user\_id, username, game\_id, number of achievements unlocked

#### **Freshness / Latency Goal**

* On-demand, milliseconds.

#### **Access Pattern**

* Increment counter on each achievement event -> retrieve top-50 users for the selected game with sorted set

### **Task 11 - Find top-30 largest game studios by number of developers**

### **Primary DBMS: Redis**

**Instant ranking updates:**

* Each time a user is added to a studio, the studio’s counter in a sorted set is incremented, maintaining an up to date ranking.

**Low latency:**

* In-memory operations ensure sub-millisecond reads and writes, ideal for on-demand queries.

**No complex joins or aggregations:**

* No complex joins or aggregations. The system only needs to maintain a counter per studio.

#### **Alternative: MongoDB**

**How it would work:**

* Aggregate studio documents by counting the number of users in each studio.

**When to choose it:**

* Suitable when the number of updates is moderate and real-time ranking is not required. Good for on-demand analytics.

**Trade-off:**

* Aggregations are slower than Redis for real-time queries, but allows flexible filtering.

#### **Why others are weaker**

* Cassandra: Can store counters per studio, but global sorting for top-30 would require additional steps and is less flexible for on-demand queries.
* Neo4j: Graph traversal unnecessary.

#### **Expected Output Format**

* List of 30 studios: studio\_id, studio\_name, number of developer users

#### **Freshness / Latency Goal**

* On-demand, milliseconds.

#### **Access Pattern**

* Increment counter when new user joins the studio -> retrieve top-30 studios with sorted set

### **Task 12 - Select top-10 favourite genre of game in the last two weeks**

### **Primary DBMS: MongoDB**

**Flexible aggregation:**

* Combines user game activity with genre information to compute total playtime and unique player counts per genre in a single query.

**Top-N ranking:**

* Supports sorting and limiting in query to produce the top-50 users efficiently.

#### **Alternative: Cassandra**

**How it would work:**

* Maintain per genre cumulative playtime and unique player counters during ingestion. Query pre-aggregated values and sort to get the top-10 genres.

**When to choose it:**

* Suitable for high write volumes where frequent updates occur and predictable queries are required.

**Trade-off:**

* Less flexible with unplanned queries or queries with additional metrics compared to MongoDB.

#### **Why others are weaker**

* Redis: Can serve precomputed top-10 results, but cannot compute aggregations like total playtime and unique players directly from raw data.
* Neo4j: Graph traversal unnecessary.

#### **Expected Output Format**

* List of 10 games: genre, total playtime, number of unique players

#### **Freshness / Latency Goal**

* Monthly, seconds - minutes.

#### **Access Pattern**

* Join game activity with genre information -> aggregate total playtime and count unique players -> sort by total playtime descending -> select top-10 genres.

### **Task 13 - Top-20 most expensive games added during the last year**

### **Primary DBMS: MongoDB**

**Flexible aggregation:**

* Filters games by release\_date within the last year and sorts by price to select the top-20.

**Top-N ranking:**

* Supports sorting and limiting in single query to produce the top-20 highest priced games.

**Batch-friendly:**

* Monthly execution works well with MongoDB aggregation and moderate query frequency.

#### **Alternative: Cassandra**

**How it would work:**

* Maintain per-game price and release date at ingestion. Query the last year’s releases and sort externally to determine the top-20.

**When to choose it:**

* Suitable for high write volumes and when the schema and queries are predictable.

**Trade-off:**

* Less flexible with unplanned queries or queries with additional metrics compared to MongoDB.

#### **Why others are weaker**

* Redis: Can serve precomputed top-20 results but cannot compute filtering and sorting on raw data.
* Neo4j: Graph traversal unnecessary.

#### **Expected Output Format**

* List of 20 games: game\_id, game title, price, release date, publisher studio, game studio.

#### **Freshness / Latency Goal**

* Monthly, seconds - minutes.

#### **Access Pattern**

* Filter games by release\_date -> sort by price descending -> select top-20 -> join publisher and game studio.

### **Task 14 - Count daily views of a selected game for the last 30 days**

### **Primary DBMS: Cassandra**

**Time-series:**

* Stores daily view counts per game, supporting fast writes for each view event.

**Fixed queries:**

* Queries are fixed and predictable: “get daily counts for the last 30 days” per game, which is handled efficiently in Cassandra.

**Scalable for high write volume:**

* Can ingest many view events per day without slowing performance.

#### **Alternative: MongoDB**

**How it would work:**

* Aggregate Game View documents by day to count daily views for the last 30 days.

**When to choose it:**

* Suitable when the data volume is moderate and queries may require flexible filtering or additional fields.

**Trade-off:**

* Less efficient than Cassandra for continuous high volume writes, but more flexible.

#### **Why others are weaker**

* Redis: Can store precomputed daily counts, but not suitable for storing raw event streams or long-term time-series aggregation.
* Neo4j: Graph traversal unnecessary.

#### **Expected Output Format**

* Non negative integer - view count.

#### **Freshness / Latency Goal**

* Daily, milliseconds.

#### **Access Pattern**

* Record each view event -> increment daily counter per game -> retrieve last 30 days counts with range query.

### **Task 15 - Top 100 users who added most games to wishlist (last 7 days)**

#### **Primary DBMS: Cassandra**

**Efficient time-series aggregation:**

* Data can be divided and aggregated by week or date, enabling quick counting of wishlist adds per user in the last 7 days.

**High write throughput:**

* Handles large volumes of wishlist events efficiently, good for frequent user activity (like wishlist adds).

**Excellent top-N Queries:**

* Pre-aggregated counters allow fast query of top-N users.

#### **Alternative: MongoDB**

**How it would work:**

* Aggregate wishlist documents by user\_id, filter by date, and count additions.

**When to choose it:**

* when flexible filtering (e.g., region or platform) is needed.

**Trade-off:**

* Slower aggregation in DocBased DBMS than in wide-collumn DBMS

#### **Why others are weaker**

* Neo4j: Graph traversal unnecessary. No relationship to traverse. Not a graph.
* Redis: Not persistent and optimal for long-term historical data storage.

#### **Expected Output Format**

* List of 100 users: user\_id, username, wishlist\_adds

#### **Freshness / Latency Goal**

* Weekly; 95 % of runs finish < 5 seconds.

#### **Access Pattern**

* Increment counters by user\_id during adding to wishlist.
* Periodically compute top 100 users or pre-aggregated counters.

### **Task 16 - Top-100 most active reviewers (last month)**

#### **Primary DBMS: Cassandra**

**Efficient time-series aggregation:**

* Data can be divided and aggregated by week or date, enabling quick counting of review writes per user in the last month.

**High write throughput:**

* Handles large volumes of wishlist events efficiently, good for frequent user activity (like review writes).
* Scaleable even with millions of data

**Excellent top-N Queries:**

* Pre-aggregated counters allow fast query of top-N users.

#### **Alternative: MongoDB**

**How it would work:**

* Aggregate review documents by user\_id, filter by month, and count reviews.

**When to choose it:**

* when flexible filtering is needed.

**Trade-off:**

* Aggregation over large time ranges is slower than Cassandra’s counter-based approach

#### **Why others are weaker**

* Neo4j: Graph traversal unnecessary. No relationship to traverse. Not a graph.
* Redis: Not persistent and optimal for long-term historical data storage.

#### **Expected Output Format**

* List of 100 users: user\_id, username, review\_count

#### **Freshness / Latency Goal**

* Monthly; 95 % of runs finish < 10 seconds.

#### **Access Pattern**

* Increment counters by user\_id during posting review.
* Periodically compute top 100 users or pre-aggregated counters.

### **Task 17 - Top-50 tags used most frequently (last year)**

#### **Primary DBMS: MongoDB**

**Flexible schema:**

* Tags are stored in arrays; MongoDB unwinds and counts them.

**Powerful filtering and aggregation:**

* Allows for fast and efficient computing of tag frequency and possible filtering of unwanted tags (eg. hend-held compatible, accessibility for VI)

**Good for analytics:**

* Supports filtering and sorting by tag counts in a single query.

#### **Alternative: Cassandra**

**How it would work:**

* Precomputed tags counters per tag per year and stored in a wide table.

**When to choose it:**

* When datasets are large and DocBased BDS are slow.

**Trade-off:**

* Not so much flexibility. Designed just for counting frequency of tags

#### **Why others are weaker**

* Redis: Requires precomputed counters and manual updates.
* Neo4j: Not suited for tag frequency; no relationships to traverse.

#### **Expected Output Format**

* List of 50 tags: tag\_name, occurrences

#### **Freshness / Latency Goal**

* Yearly; order of minutes.

#### **Access Pattern**

* Unwind and aggregate tags in all games, group by tags, query top 50.

### **Task 18 - Count number of newly registered users (last year)**

#### **Primary DBMS: Cassandra**

**Efficient time-series aggregation:**

* Data can be divided and aggregated by year or date-range, enabling quick counting of users in the last year.

**High write throughput:**

* Handles large volumes, good scalability.

#### **Alternative: MongoDB**

**How it would work:**

* Query all users where registration\_date ≥ last year, count results.

**When to choose it:**

* When datasets are not large enough to overwhelm mongoDB or other DocBased DBMS.

**Trade-off:**

* Slower on very large Datasets

#### **Why others are weaker**

* Neo4j: Graph traversal unnecessary. No relationship to traverse. Not a graph.
* Redis: Not persistent and optimal for long-term historical data storage.

#### **Expected Output Format**

* Single value: total\_new\_users

#### **Freshness / Latency Goal**

* Yearly; order of seconds.

#### **Access Pattern**

* Query by registration date, count, store yearly summary

### **Task 19 - Top-10 most helpful reviews in the selected game (last 6 months) (or on new game patch)**

#### **Primary DBMS: MongoDB**

**Powerful filtering and aggregation:**

* Secondary indexes allow fast filtering by game\_id and sorting by helpful\_votes.

**Optimal for storing complex objects:**

* Supports filtering and sorting by tag counts in a single query.

#### **Alternative: Cassandra**

**How it would work:**

* Store helpful-vote counters separately and query by game partition.

**When to choose it:**

* When datasets are large and DocBased BDS are slow. When performance on predefined queries needs to be predictable

**Trade-off:**

* Hard filters and not so flexible

#### **Why others are weaker**

* Redis: Not persistent. Not optimal for long-term analytics
* Neo4j: no relationships to traverse.

#### **Expected Output Format**

* List of 10 reviews: review\_id, content, helpful\_votes

#### **Freshness / Latency Goal**

* On-request; order of seconds.

#### **Access Pattern**

* Query by game\_id, sort by helpful\_votes, limit 10.

### **Task 20 - Recommend games by similar users**

#### **Primary DBMS: Neo4j**

**Naturally relationship based model:**

* Naturally represents the relationship between user-user and user-game in the graph.

**Perfect for multi-step traversal:**

* Would allow further better recommendation from multiple node steps.

**Excellent for recommendations and community analysis**

* Results update automatically as users add or remove games.

#### **Alternative: Cassandra**

**How it would work:**

* Pre-computed similarity scores and similar users (for example friends)

**When to choose it:**

* When recommendations don't have to be optimal and data size or latency is the problem

**Trade-off:**

* Not real time recommendations and less flexible and intuitive

#### **Why others are weaker**

* MongoDB: Can only store pre-computed lists. Would need additional data to compute similarity of games and players.
* Redis: Not suited for complex user–game graph traversal.

#### **Expected Output Format**

* List of N recommended games: game\_id, game\_name, similar\_user\_count

#### **Freshness / Latency Goal**

* Daily, order of seconds.

#### **Access Pattern**

* graph traversal (user-game-user), sort, filter