A Large Dataset of Video Game Patch Notes

by

Saksham Tejpal

A thesis submitted in partial fulfillment

of the requirements for the degree of

Bachelor

in

Computer Science

Ontario Tech University

Oshawa, Canada

Supervisor: Dr. Cristiano Politowski

April 2026

Abstract

Video game patch notes are developer-authored summaries of game updates that document bug fixes, new features, and balance adjustments. Developers publish them directly to players, making them a chronological record of how games evolve after release. Studying patch notes at scale can reveal recurring problems, common feature patterns, and how studios of different sizes respond to player feedback. However, on the Steam platform, the largest digital distribution service for PC games, Steam does not systematically separate patch notes from general announcements such as sales events, community posts, or promotional content, making it impractical to study them without a dedicated extraction and classification pipeline.

In this thesis, we present a four-stage pipeline that collects, filters, classifies, and serves game patch notes from the Steam platform. We first retrieve metadata and news items for 145,622 games using the Steam Application Programming Interface (API), then apply keyword-based filtering to identify 1,085,168 candidate patch notes from 1,873,879 total news items. To improve precision, we apply a second filtering stage using Gemini 2.5 Flash, a Large Language Model (LLM), which confirms and classifies 817,765 genuine patch notes across 68,043 games. We process each confirmed patch note to extract individual change statements, yielding 8,312,048 classified text fragments tagged as bug fixes (36.2%), feature additions (45.8%), or balance changes (18.0%). Finally, we build a web application backed by a full-text search database that allows researchers and developers to search, filter, and

explore the classified dataset by game, tag type, or free-text query.

This work delivers the largest publicly accessible classified dataset of video game patch notes, along with the infrastructure researchers need to study game software evolution, developer communication patterns, and the cross-game recurrence of bugs and fixes.

Acknowledgements

I would like to thank my supervisor, Dr. Cristiano Politowski, for their guidance, feedback, and support throughout this project. Their insights into software engineering research and dataset construction were invaluable in shaping both the methodology and the written work.

I would also like to thank Dr. Jessie Galasso of McGill University, whose initial work in collecting the appdetails dataset provided the foundation upon which this project was built.

I would also like to thank Ontario Tech University and the Faculty of Science for providing the resources and academic environment that made this work possible.

Finally, I am grateful to my family and friends for their encouragement and patience throughout this thesis.

Contents

List of Figures

List of Tables

Abbreviations

API Application Programming Interface

BM25 Best Match 25 (relevance ranking algorithm)

CORS Cross-Origin Resource Sharing

CSV Comma-Separated Values

DLC Downloadable Content

FTS Full-Text Search

GSE Game Software Engineering

HTML HyperText Markup Language

JSON JavaScript Object Notation

JSONL JSON Lines (newline-delimited JSON)

LLM Large Language Model

RAG Retrieval-Augmented Generation

REST Representational State Transfer

SE Software Engineering

SHA1 Secure Hash Algorithm 1

SQL Structured Query Language

V&V Verification and Validation

WAL Write-Ahead Logging

Chapter 1

Introduction

1.1 Video Games as Complex Software Systems

The video game industry has grown into one of the most economically significant creative sectors in the world, generating over $189 billion in revenue in 2025 and engaging more than 3.6 billion players globally [8]. Unlike traditional software, games combine source code with a wide range of heterogeneous artifacts including graphics, animations, audio, narrative structures, and real-time physics simulations. Studios typically produce them through cross-disciplinary collaboration between programmers, artists, designers, and writers, each bringing distinct goals and creative constraints to the development process. The result is a class of software systems that differ fundamentally from conventional applications in their complexity, subjectivity, and continuous evolution.

Verification and Validation (V&V) for games illustrates this distinction clearly. In conventional software, correctness and performance serve as the primary quality criteria. In games, developers must also weigh additional dimensions such as fairness, balance, difficulty, immersion, and player satisfaction [9]. These qualities are inherently subjective and context-

dependent, making them difficult to evaluate through automated testing alone. As a result, game development follows an iterative, experimentally driven process: studios release builds, observe player behavior, collect feedback, and issue frequent updates to refine the experience.

1.2 Patch Notes and the Problem of Scale

These updates reach players through *patch notes*: developer-authored summaries of changes applied in a specific game version. Developers write patch notes in plain text with no defined structure, and their content typically enumerates bug fixes, newly added features, and balance adjustments, among others. From a software engineering perspective, patch notes function as a public changelog of a complex system's evolution, and studying them at scale can reveal how games change over time, what types of defects recur across titles, and how studios of different sizes communicate updates to their communities.

No large-scale classified dataset of game patch notes currently exists across the industry. Steam, the leading platform for PC game publishing, alone hosts over 145,622 games and 1,873,879 news items, yet its API does not separate patch notes from other types of announcements. The same endpoint returns patch notes alongside promotional posts, community events, developer blogs, and sales notices. Furthermore, even among genuine patch notes, each studio writes in a different style: some entries consist of a single sentence while others span thousands of words, with no standard format or classification. This makes it impossible to study game software evolution at scale without a dedicated extraction and classification pipeline.

In this thesis, we aim to understand how games evolve over time by building a large-scale dataset of classified patch notes. To build the dataset, we followed a four-step process:

1. **Data Collection**: We systematically retrieved metadata and news items for 145,622 games from the Steam API, building a local dataset of 1,873,879 news items.

2. **Keyword-Based Filtering**: Given that not all news items are patch notes, we filtered the dataset using a curated set of patch-note-specific keywords and regular expressions, identifying 1,085,168 candidates.

3. **LLM-Based Classification**: To further remove news items that were not genuine patch notes, we used a LLM (Gemini 2.5 Flash) to classify each candidate and extract individual change statements as bug fixes, feature additions, or balance changes. The final dataset contains 817,765 confirmed patch notes with 8,312,048 classified text fragments.

4. **Web Application**: We built a searchable public interface backed by a full-text search database, making the classified dataset accessible to researchers and developers who wish to query it by game, change type, or keyword. A publicly searchable dataset of this scale represents a valuable resource for the game software engineering research community.

The primary contributions of this thesis are:

1. A scalable data collection pipeline spanning 145,622 Steam games and 1,873,879 news items, designed to operate within Steam API rate limits over multiple days.

2. A keyword-based filtering methodology using curated, morphologically extended regular expressions to identify 1,085,168 candidate patch notes.

3. An LLM-based classification pipeline using two complementary workflows — a sequential/concurrent mode for real-time processing and a batch mode using JSON Lines (JSONL) files and the Gemini Batch API — that together confirmed and classified 817,765 patch notes with 8.3 million tagged text fragments.

4. A searchable public dataset and web interface exposing 817,765 classified patch notes through a Representational State Transfer (REST) API and browser interface, enabling

the research community to study game software evolution at unprecedented scale.

We found that feature additions represent the most common change type (45.8%), followed by bug fixes (36.2%) and balance changes (18.0%). The average patch note contains 10.16 classified text fragments, and 57.7% of notes mix more than one change type within a single release.

Chapter 2 provides background on the Steam platform, game software engineering, keyword filtering, LLMs, and the web technologies used. Chapter 3 describes the four-stage methodology in detail. Chapter 4 presents the resulting dataset and classification statistics. Chapter 5 surveys related work. Chapter 6 discusses challenges, summarizes contributions, and outlines future research directions.

Chapter 2

Background

2.1 The Steam Platform and Web API

Steam is the leading digital distribution platform for PC games, operated by Valve Corporation. As of 2025, Steam hosts a catalogue of over 247,000 applications, of which 145,622 are games [12]. It serves 132 million monthly active users and supports developers ranging from large AAA studios to solo independent creators. This diversity makes Steam a uniquely representative source for studying the breadth of the games industry.

Steam provides a publicly accessible API that exposes platform data including application lists, game metadata, and news feeds [11]. The *GetAppList* endpoint returns the full catalogue of application identifiers. The *appdetails* endpoint returns metadata for a specific application, including its type (game, Downloadable Content (DLC), video, software), name, developer, publisher, genres, categories, release date, and price. The *GetNewsForApp* endpoint retrieves up to a configurable number of recent news items for a game, with each item containing a title, content, URL, and publication date. Developers distribute patch notes through this final endpoint alongside all other types of game news.

The API enforces a rate limit of 100,000 calls per day (approximately 200 calls per 5 minutes). For a catalogue of 145,622 games, fully querying all metadata and news items requires careful pacing over multiple days. This rate limit is a central engineering constraint that shapes the data collection design described in Chapter 3.

2.2 Patch Notes in Game Software Engineering

Patch notes, also called release notes, changelogs, or update notes, are developer-authored documents that describe the changes applied in a specific software release. In the context of video games, patch notes are player-facing records that list bug fixes, new features, balance adjustments, and occasionally known issues or deprecations. They serve multiple roles simultaneously: they inform players of what has changed, document the evolution of the game's design, and provide accountability for developer decisions.

From a Software Engineering (SE) perspective, patch notes represent a form of controlled-vocabulary release documentation. Unlike internal commit messages or bug trackers, developers write them for a public audience and tend to use domain-specific language that reflects the types of changes common in game development. The three primary change categories are:

- **Bug fixes**: corrections to unintended behaviour, crashes, or inconsistencies. Developers signal these with language such as "fixed", "resolved", "corrected", or "addressed".
- **Feature additions**: new content, mechanics, or capabilities. Developers signal these with "added", "new", "introduced", or "implemented".
- **Balance changes**: adjustments to numerical values governing gameplay parameters such as damage, speed, or cost. Developers signal these with "increased", "reduced", "adjusted", "rebalanced", or "nerfed".

A single patch note entry may contain all three types simultaneously. The iterative nature of game development — driven by live player feedback, competitive balance considerations, and expanding content — means developers issue patch notes frequently, often weekly or even daily for live service games.

2.3 Keyword-Based Filtering

Keyword-based filtering is a recall-oriented approach to text classification in which a filter retains documents that contain any item from a predefined vocabulary. It is computationally inexpensive, fully transparent, and easily reproducible. For tasks where the goal is to avoid missing relevant documents, keyword filtering serves as a natural first pass.

The primary tradeoff is precision: a broad vocabulary captures many relevant documents but also admits false positives. In the context of Steam news, terms like "update" or "release" appear in patch notes but also in promotional announcements and community posts. Regular expressions handle morphological variants ("changed", "changes", "change") to ensure adequate coverage.

Keyword filtering alone cannot build a high-quality classified dataset. It can identify likely patch notes, but it cannot confirm them, distinguish different types of changes within them, or extract specific change statements from unstructured prose. These tasks require a more capable classifier.

2.4 Large Language Models for Text Classification

LLMs are neural language models trained on large text corpora to predict and generate natural language. When given a prompt describing a task, modern LLMs can perform classification, extraction, summarization, and reasoning without task-specific fine-tuning. Their ability to understand context, follow complex instructions, and produce structured

output makes them well-suited to nuanced classification tasks such as identifying and tagging patch note content.

For this project, we use **Gemini 2.5 Flash**, Google's lightweight, fast, and cost-efficient LLM. Key properties that motivated this choice include:

- **Structured output**: Gemini supports schema-constrained JavaScript Object Notation (JSON) output, allowing us to specify exact response formats and eliminate parsing failures.
- **Deterministic inference**: setting temperature to 0.0 produces reproducible, consistent classifications across runs.
- **Large context window**: necessary for processing long patch notes containing dozens of entries.
- **Batch API access**: Gemini provides an asynchronous batch processing API that accepts JSONL files and processes requests at significantly reduced cost, which is essential for processing over one million notes.

Using an LLM for classification introduces considerations not present in keyword filtering. We must validate model outputs, and the cost of calling the API for each item is non-trivial at scale. These constraints shape the dual-workflow architecture described in Section 3.3.

2.5 Batch API Processing

In synchronous API calls, the server processes each request immediately: the client sends a request, waits for the server to finish, and receives a response. This model suits interactive applications but grows expensive and slow at scale. For one million items, the accumulated wait time and per-call cost of synchronous requests become prohibitive.

Batch APIs decouple request submission from result retrieval. The client submits a file containing many requests, the server processes them asynchronously (typically overnight or within hours), and the client later downloads the results. Google's Gemini Batch API accepts requests formatted as JSONL files — one JSON object per line — and returns results in the same format. Batch processing typically costs 50% less than synchronous calls for the same model, making it the preferred approach for large-scale offline classification.

The tradeoff is latency and complexity: batch jobs are not instantaneous, they require file upload management, and the pipeline must monitor them for completion. The pipeline must also handle job failures, network interruptions, and rate limiting gracefully to operate reliably at this scale.

2.6 Web Technologies

The web application component of this project uses several technologies chosen for their suitability to the dataset's characteristics.

FastAPI is a modern Python web framework for building REST APIs. It uses Python type annotations to automatically generate input validation and API documentation, and supports asynchronous request handling via Python's `asyncio`.

SQLite with FTS5 provides full-text search capabilities within a lightweight, file-based database. The Full-Text Search (FTS)5 extension implements the Best Match 25 (BM25) ranking algorithm, a probabilistic retrieval model that scores documents based on term frequency, inverse document frequency, and document length normalization. This allows relevance-ranked search over large text collections without a separate search engine.

Write-Ahead Logging (Write-Ahead Logging (WAL)) mode in SQLite allows concurrent readers to access the database while a write is in progress, which is important for a

web application serving multiple simultaneous queries.

Chapter 3

Approach

This chapter describes the four-stage pipeline that forms the methodological core of this thesis. Figure 3.1 provides an overview of all stages and the data artifacts produced at each step.

3.1 Stage 1: Data Collection

The first stage builds a local dataset of game metadata and news items by querying the Steam Web API.

3.1.1 Application List Retrieval

We begin by retrieving the full list of application identifiers on Steam using the *GetAppList* endpoint at `https://api.steampowered.com/ISteamApps/GetAppList/v2/`. This endpoint returns a JSON object containing the `appid` and `name` fields for every application on the platform. The result is stored in `output/applist.json`.

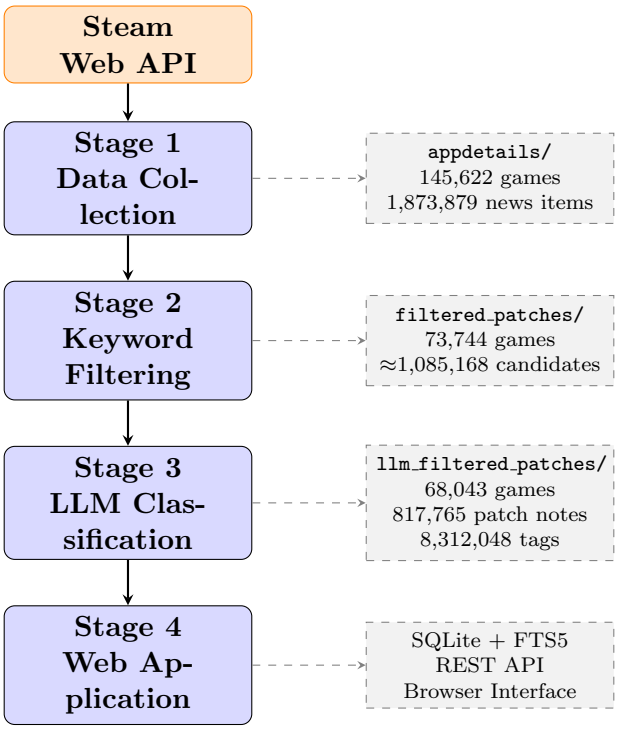

Figure 3.1: Overview of the four-stage patch note extraction, classification, and access pipeline.

3.1.2 Metadata Extraction and Game Identification

Steam hosts not only games but also DLCs, software tools, teasers, and videos — all assigned `appid` values. To isolate games, we query the *appdetails* endpoint at `https://store.steampowered.com/api/appdetails?appids=<APPID>` for each application. Applications where the returned `type` field equals `"game"` are retained. The full metadata JSON for each game is saved to `appdetails/{appid}.json`. All queries and their outcomes are logged to `output/queries.json` to prevent duplicates and enable incremental operation.

3.1.3 Rate Limit Management

The Steam API enforces a limit of 100,000 calls per day. Querying metadata for all 145,622 games therefore requires operating across multiple days. The extraction script is designed to run incrementally: it reads the existing `queries.json` log on startup and skips any `appid`

that has already been processed. This allows the pipeline to be paused and resumed without data loss or duplicate queries.

3.1.4 News Retrieval

Once game `appid` values are known, news items are retrieved using the *GetNewsForApp* endpoint at `https://api.steampowered.com/ISteamNews/GetNewsForApp/v2/?appid=<APPID>`. The full JSON response for each game is stored in `patches/raw_news/{appid}.json`. Each response contains a list of news items with fields including `title`, `url`, `contents` (full text), and `date` (Unix timestamp).

3.2 Stage 2: Keyword-Based Filtering

The second stage identifies likely patch notes from the raw news collection using keyword matching.

3.2.1 Keyword List Construction

We constructed the keyword list through a deliberate four-step process:

1. **Seeding with universal terms**: we began with terminology that is nearly universally used in software and game release notes: "patch notes", "update", "hotfix", "changelog", and "release notes".

2. **Manual scanning**: a sample of Steam news posts was manually reviewed to identify high-frequency action verbs that signal content changes: "added", "changed"/"changed", "improved", "bug(s) fix(es/ed)".

3. **Morphological variants via regex**: to maximize recall, regular expressions capture plural and conjugated forms. For example, `patch(es)?` captures both "patch" and

"patches", and `bug(s)?\s*fix(es|ed)?` captures "bug fix", "bug fixes", "bugs fixed", and so on.

4. **Retaining broad terms**: terms like "release" and "update" are broad enough to match some non-patch news, but their recall value — capturing many legitimate patch notes that would otherwise be missed — outweighs the cost of the false positives introduced.

The final keyword list includes: *patch(s) note(s)*, *update*, *hotfix*, *release*, *added*, *release note(s)*, *changelog*, *change(s/ed)*, *improved*, and *bug(s) fix(es/ed)*.

3.2.2 Filtering and Storage

A news item is retained as a candidate patch note if any keyword matches in either the item's title or its content. Matching items are saved to `patches/filtered_patches/{appid}.json`. Each file contains the game's `appId`, a total count, and an array of retained news items.

3.2.3 HTML Cleaning

The content fields of retrieved news items often include HyperText Markup Language (HTML) markup from Steam's rich text editor. Tags such as `<br>`, `<ul>`, and `<li>` are removed in a subsequent cleaning pass, leaving only plain text. The cleaned files are stored in `patches/cleaned_patches/`.

3.2.4 Metadata Compilation

To support sampling, clustering, and statistical analysis, we compile a Comma-Separated Values (CSV) file (`output/games_metadata_totals.csv`) where each row corresponds to a game and includes the following fields: `name`, `steam_appid`, `required_age`, `is_free`, `number_dlc`, `developers`, `publishers`, `price_currency`, `price_initial`, `price_final`, `windows`,

`mac`, `linux`, `metacritic_score`, `categories`, `genres`, `recommendations_total`, `achievements_total`, and `release_date`. Two additional computed fields are appended: `total_news` (the total number of retrieved news items for that game) and `total_patches` (the number of candidate patch notes identified after keyword filtering).

3.3 Stage 3: LLM-Based Classification

The keyword filtering stage maximizes recall but admits false positives: approximately 25% of the retained items are not genuine patch notes. Furthermore, keyword matching cannot distinguish bug fixes from feature announcements, nor can it extract specific change statements from multi-paragraph patch notes. The third stage addresses both limitations using LLM-based classification.

3.3.1 Design Goals

The classification stage is designed to accomplish three goals simultaneously:

1. **Verification**: determine whether a keyword-filtered news item is a genuine patch note, or a false positive such as a promotional post, event announcement, or developer blog.

2. **Extraction**: for confirmed patch notes, identify the individual change statements embedded in the free text.

3. **Classification**: label each extracted statement as a bug fix, feature addition, or balance change.

Extraction uses exact substrings from the original text rather than paraphrases. This preserves the developer's own language, making the resulting dataset useful for linguistic and terminology studies.

3.3.2 Prompt Engineering

Prompt design is critical for producing consistent, parseable output from the LLM. The system prompt used in this project is structured into four sections:

1. Task Definition. The model is instructed to read the provided title and content, determine whether the item is a patch note, and if so, extract classified change statements.

2. Output Requirements. The model must return a JSON array only. No explanatory text, no markdown formatting, and no preamble. An empty array (`[]`) is returned when the item is not a patch note.

3. Quotation Rules. Extracted text must be exact substrings of the input. The model must not paraphrase, infer intent, or correct spelling. Case and punctuation must be preserved.

4. Classification Rules with Examples. Each tag type is defined with trigger vocabulary and illustrative examples:

- `bug`: statements containing "fixed", "resolved", "addressed", or "corrected".
- `feature`: statements containing "added", "new", "introduced", or "implemented".
- `balance_change`: statements containing "increased", "reduced", "adjusted", or "rebalanced".

Prompt 3.1 shows the full system prompt used in both workflows. Prompt 3.2 shows the user prompt format, which provides the patch note title and content to the model.

Prompt 3.1: System prompt

```
You are a strict PATCH NOTE CLASSIFICATION and EXTRACTION engine
for video game patch notes.

Task:
```

```
From the given text, classify if the text is a patch note or just
news. If it is a patch note, classify and extract only
lines/snippets that are explicitly one of:
  bug
  feature
  balance_change

Output requirements:
Output JSON only.
Do not include any text other than the JSON output.
Do not include markdown tags of json.
Output must be a JSON array of objects, each object has exactly
one key from: "bug", "feature", "balance_change".
The value must be a verbatim quotation copied exactly from the
input text.
If you cannot find any explicit patch-note items, output an
empty array [].

Quotation rules:
Quotes must be contiguous substrings from the input, not
paraphrases.
Prefer quoting a whole bullet line / sentence that contains
the evidence.
Do not modify spelling/casing/punctuation.
Do not merge separate items into one quote; emit multiple
objects instead.
Output can have multiple Quotation objects of the same type if
multiple distinct items are found.

Classification rules (do not infer):

bug: A change that explicitly fixes an error/defect in the game
(e.g., crash, broken behavior, incorrect calculation, exploit),
typically signaled by "fixed", "resolved", "addressed",
"corrected".
Examples of bug fixes:
  {"bug":"- Fixed an empty item panel sometimes showing on the
           scoreboard."},
  {"bug":"- Fixed bug where some clients would crash on bomb
           explosion."},
  {"bug":"- Fixed a bug where removing/adding a silencer would
```

```
          also drop a magazine on the ground."},
  {"bug":"* Fixed Necronomicon Level 2 & 3 Manaburn hotkey not
          working."},
  {"bug":"* Fixed Dipping Nets consuming more than one net ammo
          per fish caught"}

feature: A change that adds or introduces new functionality or
content (e.g., new mode, system, item, map, UI capability),
typically signaled by "added", "new", "introduced",
"implemented".
Examples of new features:
  {"feature":"- Added the Highlights feature."},
  {"feature":"- A new game mode has been added: Deathmatch."},
  {"feature":"- Added three new playable heroes: Queen of Pain,
              Slark, and Templar Assassin"},
  {"feature":"* Added a new Summer Terrain"},
  {"feature":"* Added The Center Official ARK Mod/Map as Free DLC"}

balance_change: A change that adjusts tuning to alter gameplay
outcomes without being framed as a defect fix -- usually stat,
cost, cooldown, rate, scaling, or rule adjustments (buffs/nerfs),
typically signaled by "increased", "reduced", "adjusted",
"rebalanced".
Examples of balance changes:
  {"balance_change":"- Reduced Molotov price from 500 to 400."},
  {"balance_change":"- XM1014 - reduced damage to 20."},
  {"balance_change":"* Eul's Scepter: Bonus movement speed reduced
                     from +40 to +30"},
  {"balance_change":"* Mekansm cooldown increased from 45 to 65"},
  {"balance_change":"- Reduced Shell Resistance by 30%
                     (from 80% to 50%)"}
```

Prompt 3.2: User prompt format

```
Title:

{title}


Content:

{content}
```

Additional model settings include temperature 0.0 for deterministic output, disabled harm filters (game content may contain references to violence), and schema-constrained JSON output via the `responseMimeType` parameter. Disabling extended reasoning (`thinking_budget: 0`) reduces cost without affecting classification quality for this straightforward extraction task.

3.3.3 Sequential and Concurrent Workflow

The simpler of the two processing workflows is implemented in `scripts/llm_filtering/filter_patch_not`
This script processes game files one at a time: for each game in the input directory, it loads the filtered patch note file, submits one API call per note, and writes results to the output directory. Progress and errors are logged to `temp/logs/seq_llm_filter.log`.

The sequential workflow is synchronous within a single process but can be run as multiple concurrent instances on non-overlapping subsets of game files. This makes it suitable for smaller batches and for situations where immediate feedback is preferred over maximum throughput. It also served as the development and validation environment for prompt design, since responses are visible in real time.

3.3.4 Batch Processing Workflow

For large-scale processing of the full 1.08 million note dataset, the sequential workflow would be too slow and too costly. The batch processing workflow, implemented in `scripts/llm_filtering/batch`
uses the Gemini Batch API to submit large volumes of requests asynchronously, achieving approximately 50% cost reduction compared to synchronous calls.

The workflow consists of four components.

JSONL Conversion (`jsonl_convertor.py`)

The first step converts the per-game JSON files into JSONL format, where each line is a self-contained request object. Conversion uses a two-pass algorithm:

- **Pass 1 — Key Assignment**: Each patch note is assigned a deterministic unique key using Secure Hash Algorithm 1 (SHA1) hashing of the note's URL: `key = "{appid}::{sha1(url)}"`. These keys are written back to the game files and cached on disk. Using a content-derived hash rather than a sequential counter means the same note always receives the same key regardless of processing order, enabling idempotent re-runs and safe resume-from-failure.

- **Pass 2 — JSONL Generation**: For each note, a JSONL line is generated containing the full request structure: the model name, the system and user prompts, the response schema definition, and the note's key as a request identifier. The response schema instructs the model to return an array of objects, each with exactly one field whose name is the tag type (`bug`, `feature`, or `balance_change`) and whose value is the extracted text substring.

To allow parallel submission and resumable processing, the final JSONL file is split into parts of 2,000 lines each, stored as `batch_parts/part_001.jsonl`, `part_002.jsonl`, and so on.

Job Management (`job_processor.py`)

Each JSONL part is submitted to the Gemini Batch API as a separate job. The job manager handles three concerns that are critical for reliable long-running operation:

- **Upload reuse**: Submitting the same JSONL file multiple times (e.g., after a failure) would normally require re-uploading the file each time. The job manager caches file upload identifiers in `job_id/upload_ids.json` and reuses existing uploads if the same

file is submitted again, saving bandwidth and time.

- **Job persistence**: Submitted job identifiers are stored in `job_id/job_ids.json`. If the monitoring process is interrupted — due to a crash, power outage, or manual termination — it can be restarted and will resume monitoring the already-submitted jobs from their last known state.

- **Async polling**: Job status is checked asynchronously every 10 seconds. The manager awaits final states (`SUCCEEDED`, `FAILED`, or `CANCELLED`). Rate-limiting errors (HTTP 429, `RESOURCE_EXHAUSTED`) are captured and re-raised as typed exceptions for the orchestration layer to handle.

Batch Orchestration (`process_batch.py`)

The orchestrator manages the full submission-monitoring-parsing lifecycle for all JSONL parts. Its concurrency model is designed to maximize throughput while respecting the Gemini API's rate limits:

- Jobs are submitted sequentially, one at a time. Concurrent submissions would trigger rate limits.

- Once submitted, each job is monitored asynchronously in a background task. Up to 50 jobs can be in-flight (submitted but not yet complete) simultaneously.

- When a 429 rate-limit error is received during submission, the current part is re-queued and the orchestrator waits for one in-flight job to complete before retrying, naturally throttling submission speed.

- If no jobs are currently running, a conservative 500-second wait is applied before retrying.

This decoupling of submission and monitoring — submit sequentially to avoid rate limits, poll

all in-flight jobs in parallel — is the key design decision that enables safe, high-throughput batch processing.

The orchestrator exposes an interactive menu with five options: (1) generate JSONL from input files, (2) submit batch jobs and automatically continue to parsing upon completion, (3) check the status of submitted jobs, (4) parse completed results and apply tags, and (5) cancel all active jobs.

Results Parsing (`results_parser.py`)

Once all batch jobs complete, their result files (JSONL format, one response per line) are merged into a single `results.jsonl` file. The parser then:

1. Reads each response and extracts the tag array from the model's output.
2. Builds a results map indexed by `appid` and note key: `results_map[appid][key] = [tags]`.
3. Skips notes where the model returned an empty array, confirming they are not genuine patch notes.
4. Loads each game's original filtered patch file, matches notes by key, and embeds the extracted tags.
5. Exports the enriched files to `patches/llm_filtered_patches/{appid}.json` in batches of five games at a time to manage memory on large datasets.

The resulting files retain the original note structure (title, date, url, content, key) and add a `tags` field containing the list of classified text fragments.

3.3.5 Hybrid Concurrent Execution

Running only the batch workflow would leave uncovered any notes in files that could not be processed in a given batch job window. Running only the sequential workflow would be significantly more expensive and slower. The solution adopted in this project is to run both workflows simultaneously on non-overlapping subsets of the input files.

During processing, the sequential/concurrent instances handled files that were either too small to merit batching or needed to be processed immediately, while the batch pipeline processed the bulk of the dataset asynchronously. The batch workflow's cost savings offset the sequential workflow's speed advantage, and together they minimized both total cost and total wall-clock time. This hybrid concurrent execution is one of the primary engineering contributions of the project.

3.4 Stage 4: Web Application

To make the classified dataset accessible without requiring researchers to process raw JSON files, we built a web application that ingests the LLM-filtered data into a searchable database and exposes it through a REST API and browser interface. The web application is a secondary artifact intended to demonstrate the dataset's usability; the pipeline stages described above are the primary contributions.

3.4.1 Data Ingestion

The ingestion script (`web/ingest.py`) reads all LLM-filtered game files and loads their content into a normalized SQLite database. During ingestion, several data quality steps are applied:

- **Unicode sanitization**: invalid surrogate characters and NUL bytes — which can appear in Steam content due to inconsistent encoding practices — are removed to

prevent database insertion errors.

- **BBCode cleanup**: Steam uses a BBCode markup language (`[*]`, `[h1]`, `[/h1]`, etc.) for rich text. These tags are stripped during ingestion, leaving clean plain text.
- **Batch commits**: the database is committed every 1,000 game files to manage memory usage during the multi-hour ingestion process.

3.4.2 Database Design

The database schema (`web/database.py`) uses four tables:

- `games`: stores per-game metadata (app_id, name, developer, publisher) sourced from the SteamSpy community dataset.
- `notes`: stores each patch note's title, date, url, content, and SHA1-based key.
- `tags`: stores individual classified text fragments with a `tag_type` field (`bug`, `feature`, or `balance_change`) and a foreign key referencing the parent note.
- `notes_fts`: a virtual FTS5 table mirroring note keys, titles, and content for full-text search.

Seven database indexes — including composite indexes on (`note_key`, `tag_type`) — ensure efficient query execution across the dataset's 817,765 notes and 8.3 million tags. WAL mode is enabled to allow simultaneous reads from multiple web clients.

3.4.3 Backend API

The backend (`web/main.py`) is built with FastAPI and exposes eight endpoints covering search, tag-type browsing, game lookup, and individual note retrieval. A reusable `query_notes()` function in `web/routers/query_utils.py` implements FTS5-integrated query building with

configurable tag filtering, game scoping, BM25 ranking, and offset-based pagination.

Game name lookup supports fuzzy matching using a composite similarity score combining character-level sequence matching, token coverage, and Jaccard token overlap, with an 86% threshold for accepting a match.

3.4.4 Frontend

The browser interface (`web/client/`) is built with vanilla JavaScript and HTML/CSS. It implements a four-layer caching system: HTTP response caching (LRU, 200 entries, 60-second TTL), assembled dataset caching per query scope, summary result caching, and in-flight request deduplication. Pages are fetched in chunks of 250 results, with client-side filtering applied before displaying 20 results per page. Expanded note cards highlight matched tag text using `<mark>` elements.

Chapter 4

Results

4.1 Dataset Overview

Table 4.1 summarizes the number of games and items at each stage of the pipeline.

Table 4.1: Number of games and items at each pipeline stage.

Pipeline Stage	**Games**	**Notes / Items**
Steam catalogue	145,622	—
Raw news collected	142,983	1,873,879
After keyword filtering	73,744	≈1,085,168
After LLM filtering	68,043	817,765

The full dataset of 145,622 games spans a wide range of types: approximately 13% are free-to-play, while the remaining 87% are paid games. 95% of games carry at least one genre and one category, enabling fine-grained filtering and genre-stratified analysis. The dataset includes studios ranging from single-developer hobbyist projects to large commercial operations, making it broadly representative of the Steam ecosystem.

An example patch note from Counter-Strike (appID: 10), dated December 12, 2023, illustrates the type of content the dataset contains:

> *"Fixes for joining games through invites and Steam friends list. Miscellaneous UI fixes for Dedicated Servers. Fixed an occasional crash when opening the scoreboard. Fixed camera movement after death in multiplayer. Fixed crash when creating a server if the server creator's Steam name was entirely non-ASCII characters."*

The pipeline would extract five classified bug-fix tags from this note, each corresponding to one of the five sentences above.

4.2 Keyword Filtering Results

The keyword filter retained 1,085,168 of the 1,873,879 raw news items (57.8%). We discarded the remaining 42.2% as clearly non-patch news items that matched none of the filter keywords.

Keyword filtering achieves high recall: the filter captures the vast majority of genuine patch notes in the dataset. However, as the subsequent LLM classification stage reveals, a substantial fraction of retained items are not genuine patch notes. They are false positives such as general development updates, event announcements, or community posts that happen to contain patch-note vocabulary. This precision gap motivates the second filtering stage.

4.3 LLM Classification Results

Of the 1,085,168 keyword-filtered candidates, the LLM confirmed 817,765 as genuine patch notes (75.3% retention rate). The LLM rejected the remaining 24.7% — returning an empty array for each, indicating the items were not patch notes. These rejected items correspond to the keyword filter's false positives.

Table 4.2 presents the distribution of classified text fragments across the three tag types.

Table 4.2: Tag type distribution across 817,765 confirmed patch notes.

Tag Type	**Count**	**Percentage**
Feature additions	3,807,024	45.8%
Bug fixes	3,008,808	36.2%
Balance changes	1,496,201	18.0%
Total tags	**8,312,048**	**100%**

Feature additions represent the most commonly documented change type, accounting for nearly half of all classified fragments. Bug fixes rank second, and balance changes — most characteristic of competitive and live service games — rank lowest overall. This reflects the fact that many games in the dataset are single-player titles with fewer ongoing balance requirements.

The average patch note contains 10.16 classified text fragments, indicating that most patch notes document multiple individual changes rather than a single update. Of the 817,765 confirmed notes, 472,218 (57.7%) contain fragments of more than one tag type, meaning that a typical patch note mixes bug fixes, new features, and balance adjustments within a single release.

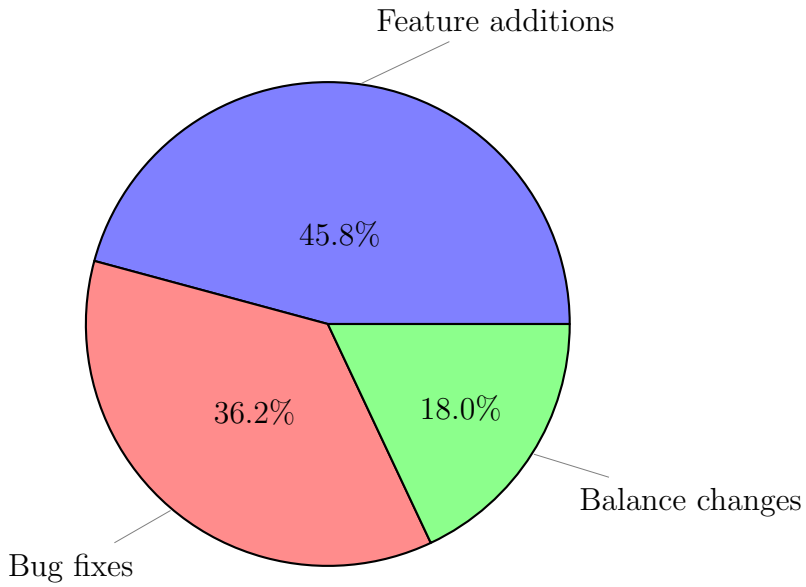

Figure 4.1: Distribution of the 8,312,048 classified text fragments by tag type.

4.4 Web Application

The web application provides a browser-accessible interface for exploring the classified patch note dataset. Users can search by free text, which the application matches against note titles and content using BM25 ranking, filter by tag type (bugs, features, or balance changes), and scope queries to a specific game by name or `appid`. Users can expand each matching note to view its full content, with the application highlighting classified tag text inline.

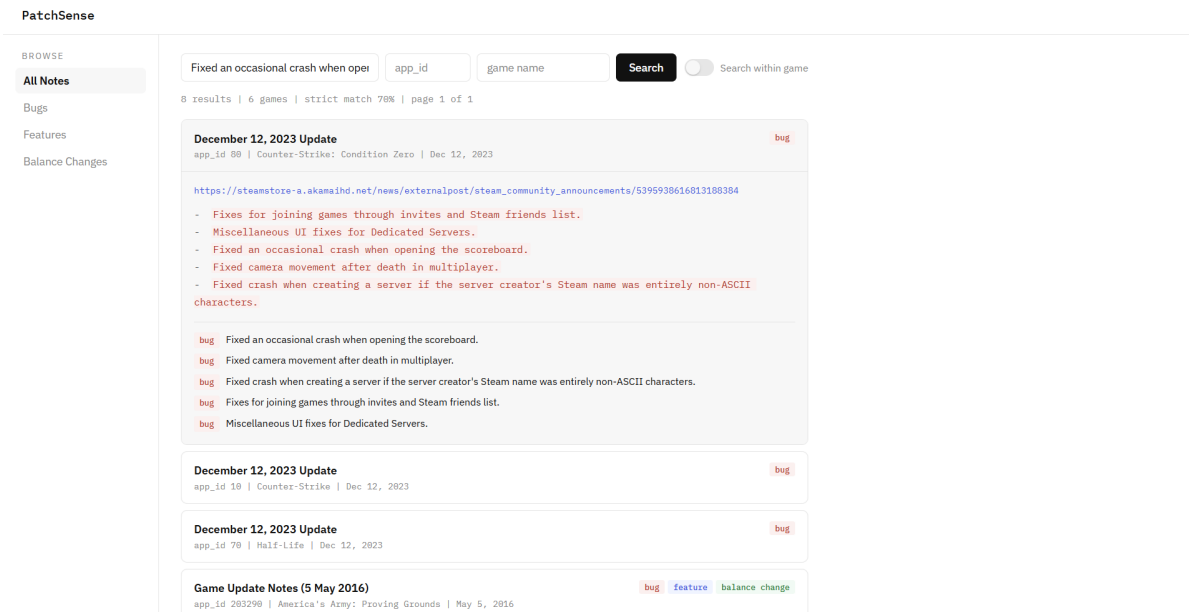

Figure 4.2: The web application's search interface showing filtered patch notes with highlighted tags.

Fuzzy game name matching lets users find games by approximate name without knowing the exact `appid`, lowering the barrier for researchers unfamiliar with Steam's internal identifiers.

Chapter 5

Related Work

5.1 Game Datasets

Several datasets support video game research. PlayMyData provides curated metadata for nearly 100,000 multi-platform games from 1993 to 2023, including descriptions, genres, user ratings, and platform availability [1]. While rich in metadata, it does not include patch notes or update histories. A dataset of 200 game postmortems identifies over 1,000 software engineering problems encountered during game development, drawn from developer-authored retrospectives [10]. Jiang and Zheng propose a deep learning approach to classify 50,000 PC games by genre using multi-modal features [3]. The AGAIN dataset links 37 hours of gameplay footage to physiological player emotion measurements, supporting affective computing research [7]. ViGGO provides a corpus of 7,000 meaning representations from 100 games for natural language generation evaluation [4].

None of these datasets provide patch note content at scale. Our dataset is the first to classify patch note text fragments across a large number of games spanning all major genres.

5.2 Patch Note Studies

Several studies examine how game updates affect player behavior, though they focus on individual games. Research on DOTA 2 shows that major updates introducing new content or significant gameplay changes substantially increase player engagement, while minor updates and bug fixes alone have less effect [13, 14]. Studies on League of Legends similarly find that balance changes (buffs and nerfs) affect competitive outcomes and player retention in measurable ways [5]. Lin et al. study urgent updates on Steam and find that frequent urgent updates can disrupt player experience, though hot patching techniques can mitigate these effects when carefully managed [6]. Hanna and Petke examine the challenges and limitations of hot patching at the language and runtime level [2].

These studies demonstrate the research value of patch note data but are limited by working with single games or small curated sets. Our dataset enables the same types of analyses at a scale of 68,043 games simultaneously, enabling cross-genre and cross-studio comparisons that were previously impossible.

5.3 Game Software Engineering

Beyond patch notes, the broader field of Game Software Engineering (GSE) addresses the unique challenges of software quality in games. Politowski et al. survey game testing practices and find that V&V in games differs fundamentally from traditional software due to the difficulty of specifying correct game behavior [9]. Their work highlights the need for empirical data sources that can inform both research and practice in game software quality. Our classified patch note dataset provides such a source at unprecedented scale.

Chapter 6

Conclusions

6.1 Summary

This thesis presents a four-stage pipeline for collecting, filtering, classifying, and serving video game patch notes at scale from the Steam platform. Starting from 145,622 games and 1,873,879 news items, keyword-based filtering reduces the search space to 1,085,168 candidate patch notes. LLM-based classification using Gemini 2.5 Flash then confirms and classifies 817,765 genuine patch notes across 68,043 games, producing 8,312,048 tagged text fragments spanning bug fixes (36.2%), feature additions (45.8%), and balance changes (18.0%). A web application backed by an FTS5 database makes this dataset interactively searchable.

The result is the largest publicly accessible classified dataset of video game patch notes, with supporting infrastructure for researcher access and future analysis.

6.2 Challenges

Building this pipeline at scale surfaced several non-trivial engineering and research challenges.

Keyword Precision vs. Recall. Designing the keyword filter required balancing competing objectives. Broad terms such as "update" and "release" are necessary to ensure that genuine patch notes are not missed (high recall), but they also admit false positives. The 24.7% rejection rate observed in the LLM stage quantifies the cost of this tradeoff and confirms that keyword filtering alone is insufficient for a high-quality dataset. The two-stage approach — keyword filtering for recall, LLM classification for precision — addresses both goals at the cost of added pipeline complexity.

API Rate Limits and Rate-Limiting Errors. Both the Steam API (100,000 calls/day) and the Gemini Batch API enforce rate limits. For the Steam API, incremental processing across multiple days with query logging solved the problem. For the Gemini Batch API, 429 (`RESOURCE_EXHAUSTED`) errors during job submission required a more sophisticated solution: the orchestrator submits jobs sequentially (preventing concurrent 429 triggers), monitors all submitted jobs asynchronously in parallel, and re-queues any part that encounters a rate limit error rather than failing it outright.

Cost and Speed at Scale. Processing 1.08 million notes synchronously with the Gemini API would be both slow and expensive. The batch API reduces cost by approximately 50% but introduces latency, as batch jobs complete asynchronously over hours rather than immediately. Running the sequential and batch workflows concurrently on non-overlapping subsets of the data was the practical solution: the batch pipeline handled the bulk of the dataset cheaply, while the sequential pipeline covered the remainder in real time. This hybrid approach balanced cost against latency without sacrificing completeness.

Asynchronous Concurrent Processing. Processing over one million notes efficiently required designing an asynchronous job management system using Python's `asyncio`. The batch orchestrator had to maintain up to 50 in-flight jobs simultaneously — each an asynchronous polling task — while sequentially submitting new jobs to avoid triggering rate limits. Coordinating these two concurrent workflows (sequential and batch) running simul-

taneously on non-overlapping file subsets, without double-processing any game, required careful state tracking. Managing this concurrency without race conditions or missed files was a non-trivial engineering challenge at this scale.

Resume-from-Failure. Long-running batch API jobs spanning multiple hours are vulnerable to process interruptions. Without persistence, an interrupted run would lose track of which jobs had been submitted and which results were still pending. SHA1-based deterministic key generation ensures that re-running the JSONL conversion always produces the same keys, avoiding duplication. Persistent job ID and file upload ID caches allow the monitoring process to resume from its last state without re-submitting already-completed jobs.

Data Quality at Scale. Raw Steam content includes invalid Unicode surrogates, NUL bytes, and Steam BBCode markup that must be handled before the data can be stored or searched. These issues are common in web-scraped content but are easy to overlook. A dedicated sanitization layer in the ingestion script prevents downstream failures in the database and search engine.

6.3 Future Work

The classified patch note dataset opens several directions for future research and system development.

Semantic Clustering and Analysis. The 8.3 million classified text fragments can be embedded and clustered semantically to identify recurring bug categories, common feature patterns, and balance adjustment strategies across genres and studios. For example, clustering bug fix texts might reveal that physics collision bugs, networking desync issues, and UI rendering errors appear consistently across games developed with the same engine, regardless of genre. This would provide empirical grounding for generalizations about game software quality.

Game Evolution Studies. The dataset spans multiple years of patch history for many games. Longitudinal analysis could examine how the ratio of bug fixes to feature additions changes across a game's lifecycle — early releases may focus more on bug fixes; mature games may shift toward balance — or how update frequency and content type differ between AAA and independent studios.

RAG-Based Developer Tool. The most practically impactful future direction is building a Retrieval-Augmented Generation (RAG)-based AI assistant that uses the classified patch note corpus as its grounding knowledge base. A developer encountering an unfamiliar bug could describe it in natural language, and the system would retrieve semantically similar bug fix statements from the dataset, along with the game context in which each fix appeared. This would transform the static dataset into an active resource for solving real development problems, grounded in the collective experience of thousands of game studios.

Cross-Platform Expansion. The current dataset is limited to Steam. Expanding the pipeline to other PC and console distribution platforms — Epic Games Store, GOG, itch.io, PlayStation Network, Xbox Live — would broaden coverage to console-exclusive games and independent titles not distributed through Steam, enabling platform-comparative analysis.

Bibliography

[1] Andrea D'Angelo, Claudio Di Sipio, Cristiano Politowski, and Riccardo Rubei. PlayMyData: A curated dataset of multi-platform video games. In *Proceedings of the 21st International Conference on Mining Software Repositories*, pages 525–529. ACM, 2024.

[2] Carol Hanna and Justyna Petke. Hot patching hot fixes: Reflection and perspectives. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1781–1786. IEEE, 2023.

[3] Yuhang Jiang and Lukun Zheng. Deep learning for video game genre classification. *Multimedia Tools and Applications*, 82(14):21085–21099, 2023.

[4] Juraj Juraska, Kevin K. Bowden, and Marilyn Walker. ViGGO: A video game corpus for data-to-text generation in open-domain conversation. *arXiv preprint arXiv:1910.12129*, 2019.

[5] Artian Kica, Andrew La Manna, Lindsay O'Donnell, Tom Paolillo, and Mark Claypool. Nerfs, buffs and bugs – analysis of the impact of patching on League of Legends. In *2016 International Conference on Collaboration Technologies and Systems*, pages 128–135. IEEE, 2016.

[6] Dayi Lin, Cor-Paul Bezemer, and Ahmed E. Hassan. Studying the urgent updates of popular games on the Steam platform. *Empirical Software Engineering*, 22(4):2095–

2126, 2017.

[7] David Melhart, Antonios Liapis, and Georgios N. Yannakakis. The arousal video game annotation (AGAIN) dataset. *IEEE Transactions on Affective Computing*, 13(4):2171–2184, 2022.

[8] Newzoo. Global games market to hit $189 billion in 2025, 2025.

[9] Cristiano Politowski, Fabio Petrillo, and Yann-Gaël Guéhéneuc. A survey of video game testing. In *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*, pages 90–99. IEEE, 2021.

[10] Cristiano Politowski, Fabio Petrillo, Gabriel Cavalheiro Ullmann, Josias De Andrade Werly, and Yann-Gaël Guéhéneuc. Dataset of video game development problems. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 553–557. ACM, 2020.

[11] Valve Corporation. Steam web API documentation.

[12] Valve Corporation. Steam platform statistics, 2025.

[13] Xiaofang Zhong and Jinjie Xu. Game updates enhance players' engagement: A case of DOTA2. In *2021 4th International Conference on Information Management and Management Science*, pages 117–123. ACM, 2021.

[14] Xiaofang Zhong and Jinjie Xu. Measuring the effect of game updates on player engagement: A cue from DOTA2. *Entertainment Computing*, 43:100506, 2022.