Software Design Document for Project Safe Web

Cao Vo Nhat Minh - Student ID: 22125054

April 29, 2025

1 Introduction

1.1 Purpose

This Software Design Document (SDD) describes the architecture, components, and data flow of the Safe Web system — a Chrome Extension that uses AI to detect and hide images based on user-defined tags. The purpose of this document is to provide a comprehensive overview of the system's design for developers, maintainers, and reviewers. It serves as a guide for implementation, testing, and future enhancements.

1.2 Scope

Safe Web is an AI-powered Chrome extension designed to enhance the web browsing experience by automatically detecting and hiding images based on custom tags provided by the user. Using advanced machine learning models (such as OpenCLIP [8]) to understand image content, the extension filters out images that may be inappropriate, distracting, or unwanted according to user preferences.

The extension operates seamlessly while users browse the web, scanning all images on a page, analyzing their semantic meaning, and comparing them to user-defined tags. When an image matches a tag with a high similarity score, the extension automatically hides it, creating a cleaner, safer, and more personalized browsing environment.

Safe Web is lightweight, user-friendly, and customization, allowing users to dynamically update their filter tags, review hidden images, and fine-tune their browsing experience.

1.3 Overview

This document presents the software design for Safe Web. It is organized into sections that describe the system's purpose, architecture, user interface design, data structures, and component-level logic. The document begins with a high-level overview of the system, followed by detailed descriptions of each module, their interactions, the data flow, and the graphical user interface. It concludes with design diagrams and pseudo-code that define the system's functionality and guide implementation. This structure ensures clarity for both developers and reviewers. You can find the public source code at Github reporistory [4], this demo work uses cloud database storage from Aiven Console [1].

2 System Overview

The design of Safe Web emphasizes lightweight performance, ease of customization, and scalability. It consists of several core components:

- A background script to manage extension lifecycle events and communication between content scripts and the server.
- A content script to scan the DOM, process all images on the page, and dynamically hide inappropriate images.
- A popup user interface (popup.html) for users to add or remove custom tags and adjust filtering settings.
- A history page (history.html) that allows users to view a record of previously hidden (aborted) images, including metadata such as the host website, time hidden, and matched tag.
- A server-side FastAPI [10] application for handling authentication, tag storage, and user synchronization.
- A database to securely store user preferences, custom tags, and history logs.

3 System Architecture

3.1 Architectural Design

The app use the basic architecture of Chrome extension [6], Safe Web system is organized into a modular structure. The system responsibilities are partitioned into several high-level subsystems, each focusing on specific roles. These subsystems collaborate through well-defined interactions to achieve the complete functionality of the extension.

The architecture follows a modular MVC pattern (as show in **Figure 1**). Views include the browser, popup, and history UI. Controllers manage DOM scanning (content script), user input (popup), and logging (background). A FastAPI [10] server handles inference and database operations. MySQL [9] stores user and image log data. This separation enhances maintainability, performance, and scalability for real-time content filtering.

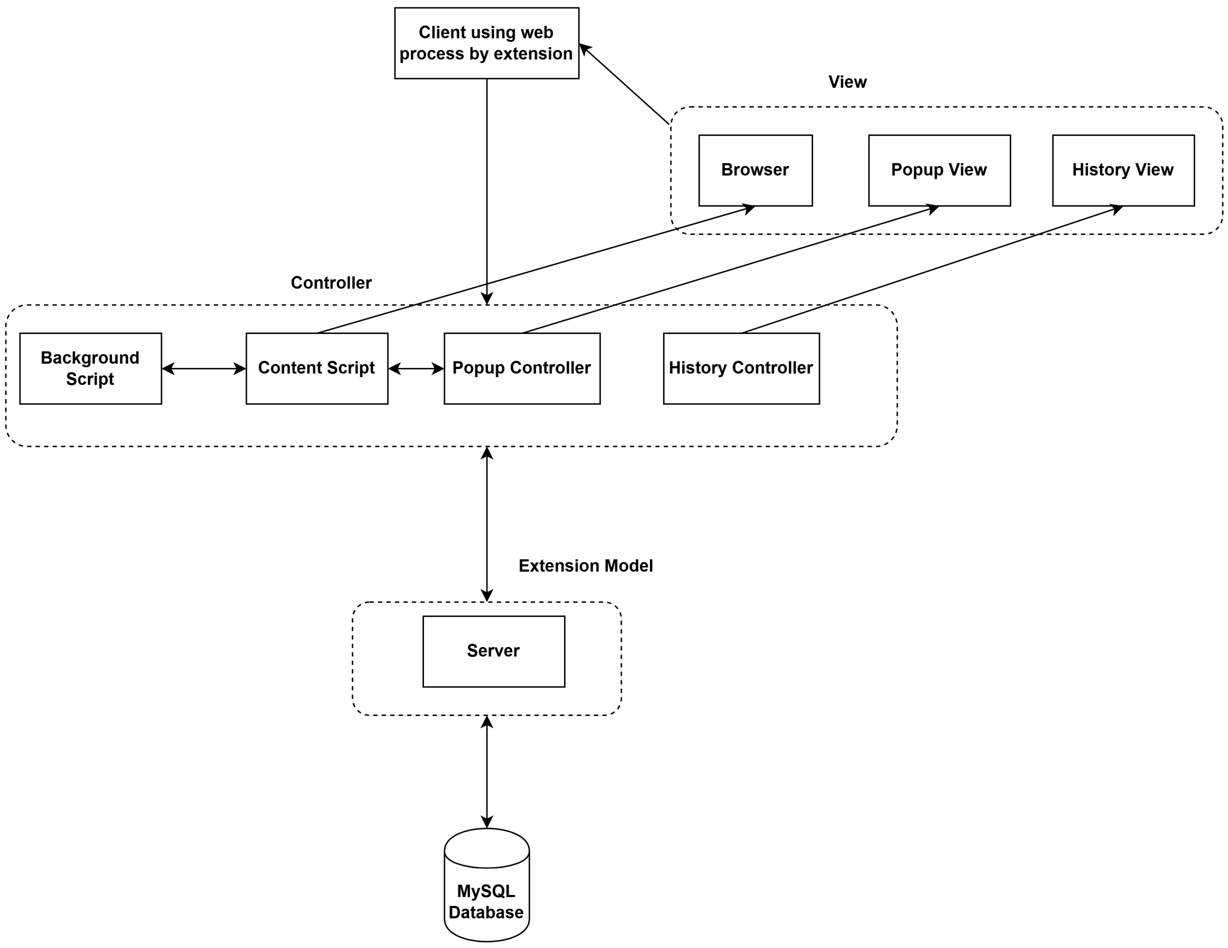


Figure 1: MVC based Architectural Design

3.1.1 content.js

Responsibility:
Handles all DOM manipulation tasks. It scans the web page for images, processes them, matches images against user-defined tags, and hides those deemed inappropriate.

Notes:
`content.js` operates solely within the context of the active webpage and does not communicate directly with the server. It relies on `background.js` for any external interactions if needed.

3.1.2 background.js

Responsibility:
Acts as the central communication hub for the extension.

- Manages OAuth authentication with Google Cloud credentials [7].
- Handles all API requests to the server (`server.py`), including sending user data, retrieving tags, and logging hidden images.

- Coordinates communication between content scripts (`content.js`) and other parts of the extension (popup and history pages).

3.1.3 popup.js and popup.html

Responsibility:
Provides the user interface for managing custom tags.

- Allows users to add, modify, or remove tags.
- Sends requests (through `background.js`) to update or fetch tag information from the server.

3.1.4 history.js + history.html

Responsibility:
Provides an independent interface to view the history of hidden (aborted) images.

- Opened via a user action from the popup interface, in a new browser tab.
- Directly communicates with the server (`server.py`), bypassing `background.js`.
- Retrieves and presents data such as the host webs, time hidden, and matched tags in a user-friendly table.

3.1.5 server.py

Responsibility:
Serves as the backend service, implemented using FastAPI [10].

- Processes API requests from `background.js` and `history.js`.
- Handles OAuth[7] token validation and secure communication.
- Coordinates internal operations by delegating database access to `database.py` and model inference tasks to `clip.py`.

3.1.6 database.py

Responsibility:
Provides an abstraction layer for all MySQL[9] database operations.

- Manages user authentication and profile data.
- Handles CRUD operations for user-defined tags.
- Stores and retrieves history of hidden images with metadata such as timestamps and image URLs.

3.1.7 clip.py

Responsibility:
Encapsulates all interactions with the CLIP[8] model used for image content analysis.

- Embeds images and text tags into a common vector space.
- Computes similarity scores between images and user-defined tags.
- Provides a filtering decision mechanism based on threshold settings.

3.1.8 MySQL[9] Database

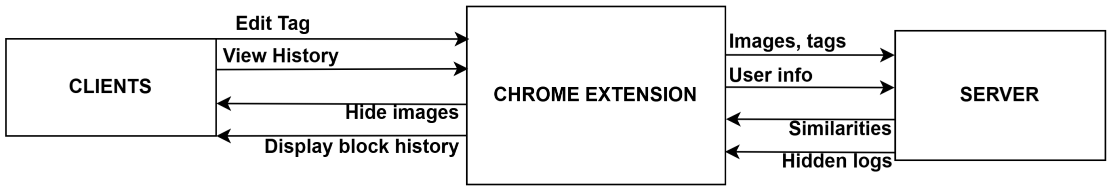
Responsibility:
Persistent storage for:

- User authentication and profile data.
- User-defined filtering tags.
- History of hidden images (including metadata such as timestamps and image URLs).

3.2 Decomposition Description

This section provides a functional decomposition of the Safe Web system. We describe the architecture using a top-level Data Flow Diagram (DFD) and a Structural Decomposition Diagram to illustrate how responsibilities are partitioned among the major modules.

3.2.1 Top-Level Data Flow Diagram (DFD)



Description:
The client interacts with the Chrome Extension [6] interfaces (popup.js, content.js, history.js). The extension communicates with the backend Server through API calls. The server coordinates database queries via `database.py` and performs image filtering inference using the CLIP[8] model via `clip.py`. Data is then returned back to the frontend.

3.2.2 Structural Decomposition Diagram

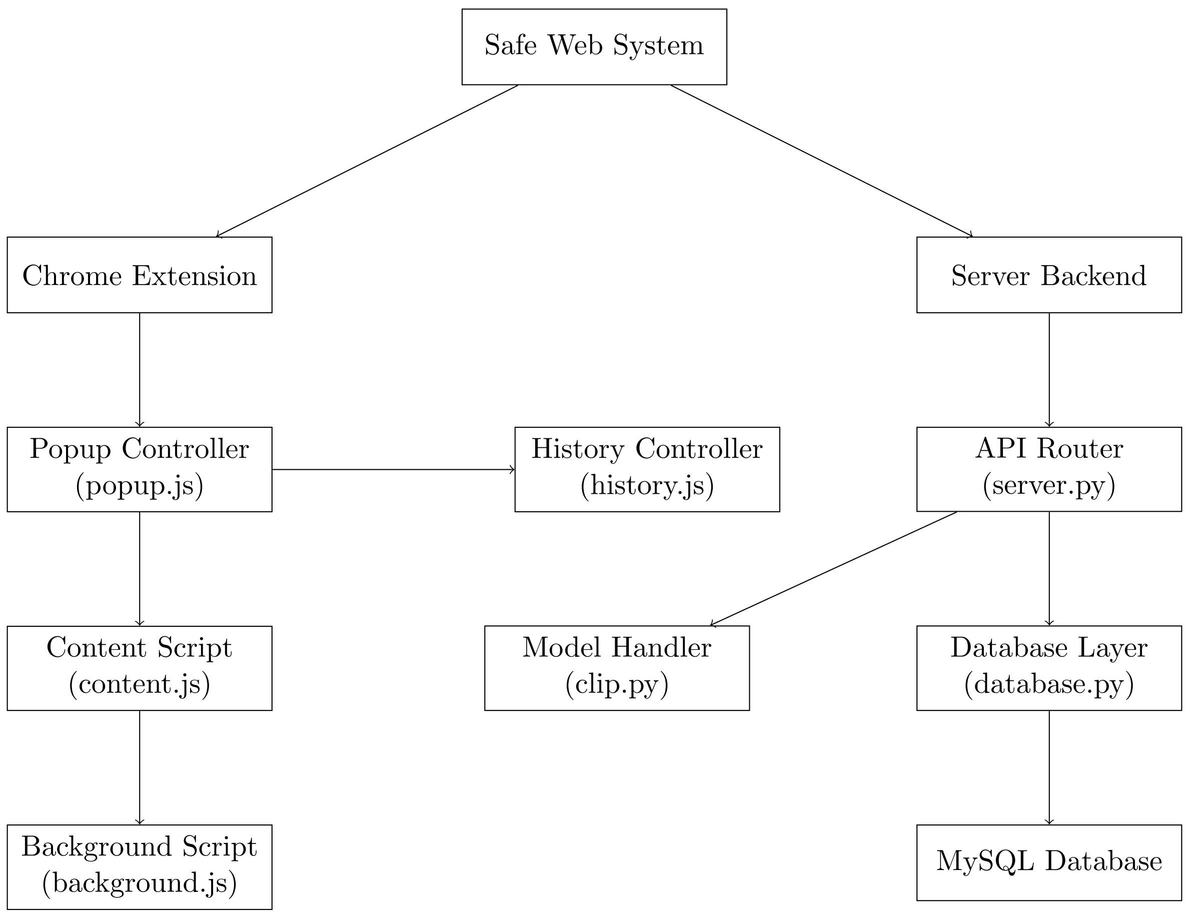

Safe Web System
Chrome Extension
Server Backend
Popup Controller (popup.js)
History Controller (history.js)
API Router (server.py)
Content Script (content.js)
Model Handler (clip.py)
Database Layer (database.py)
Background Script (background.js)
MySQL Database

Figure 2: Structural decomposition of Safe Web

3.3 Design Rationale

The architecture selected for SafeWebUS-2025 follows a classic, well-established Chrome Extension design pattern, adapted to accommodate AI-driven content filtering with server-side support.

3.3.1 Chrome Extension Frontend

We adopt a modular decomposition typical of Chrome Extensions [6]:

- **Content Scripts (content.js)**: Operate within the web page's context to manipulate the DOM without elevated permissions. This separation follows Chrome's security model, ensuring that direct page modifications are sandboxed.
- **Background Scripts (background.js)**: Serve as a centralized messaging hub and manage persistent tasks such as authentication (OAuth 2.0 [7] with Google Cloud), API communication, and shared data management. Using a background script is a Chrome-recommended best practice for maintaining extension state across pages and tabs.

- **Popup (popup.js / popup.html)**: Provides a lightweight user interface for immediate interaction, configuration, and quick settings adjustments. Popup pages allow users to interact with the extension without affecting the ongoing background processes.
- **History Page (history.js / history.html)**: Implemented as an independent tab for displaying user history data. It is designed as a standalone interface that communicates directly with the server, reflecting a scalable and modular design where not all user interfaces rely on background.js.

This frontend structure aligns with Chrome Extension [6] development guidelines, ensuring maintainability, scalability, and compliance with Chrome Web Store security policies.

3.3.2 Server Backend

The server backend is designed using FastAPI [10] to handle API endpoints efficiently. The backend architecture is partitioned into:

- **server.py**: Acts as the main routing and logic controller.
- **database.py**: Abstracts all database operations to ensure a clean separation of concerns.
- **clip.py**: Encapsulates all model inference tasks (using CLIP[8] for image tagging and matching).

By delegating database and AI model tasks to separate modules, the backend ensures:

- **Maintainability**: Each module can be updated independently.
- **Reusability**: Database access and model inference logic can be reused or extended for future features.
- **Scalability**: Backend services can be upgraded independently without affecting the frontend structure.

3.3.3 Shared Storage via Chrome Local Storage

Another key architectural element is the use of Chrome's built-in `chrome.storage.local` API. This storage mechanism is used to share data persistently across different extension components, such as `popup.js`, `content.js`, and `background.js`.

This choice offers several benefits:

- **Cross-context communication:** Since popup, content, and background scripts run in isolated contexts, local storage provides a shared medium for reading and writing data without direct messaging.
- **Persistence:** Data stored in `chrome.storage.local` persists across browser restarts, making it suitable for storing user preferences and settings.

- **Security and Performance:** Compared to using background messages for every small data sync, local storage reduces runtime overhead and latency.

For example, user-defined filtering tags are stored locally so that `content.js` can access them instantly without triggering a remote request. This reduces unnecessary API calls and ensures seamless real-time filtering even in offline mode.

3.3.4 Critical Issues Considered

Several critical factors influenced this architecture:

- **Security**: Following Chrome Extension's [6] principle of least privilege — content scripts have limited access; sensitive operations (authentication, API calls) are handled in background scripts.
- **Performance**: `content.js` processes DOM manipulation locally without unnecessary server calls. Server interactions are lightweight and asynchronous to avoid blocking the user interface.
- **Scalability**: By isolating the database and AI model modules, we can easily expand to more complex filtering or history tracking features without redesigning the system.
- **User Experience**: Decoupling `history.js` into a standalone page improves the browsing experience by offloading history review to a separate interface.
- **High-Frequency Data Logging**: The system performs frequent updates to log hidden images in near real time. To support this, a relational database (MySQL[9]) was chosen for its robust support of frequent inserts, query indexing, and structured data management. MySQL [9] ensures reliability, consistency, and high throughput in such high-write scenarios.

Conclusion: The architecture of Safe Web is based on best practices for Chrome Extension development combined with a modular server-side backend. This hybrid architecture ensures security, flexibility, scalability, and high responsiveness, aligning with the goals of the project.

4 Data Design

4.1 Data Description

The Safe Web system processes a variety of data types across both the frontend (Chrome Extension[6]) and backend (FastAPI[10] server). Data is transformed into well-defined structures suitable for persistent storage, real-time access, and model inference.

- **User Tags:** Tags defined by the user to filter inappropriate content. These are stored both in Chrome's `chrome.storage.local` for fast access on the client side.

- **Image Metadata:** When an image is hidden, relevant metadata (image URL, matched tags, timestamp, host web name) is logged to the backend and stored in the database. This data is used to display user history in `history.html`.
- **OAuth [7] Tokens:** Tokens obtained from Google OAuth [7] are stored temporarily in the background script for authenticated API requests. These are validated by the backend before processing.
- **CLIP Embeddings:** The server-side `clip.py` module generates image embeddings for semantic comparison with tag embeddings. These embeddings are processed on-the-fly and not stored persistently.
- **Database Tables:** The MySQL[9] database includes tables for users, tags, and hidden image logs. These tables are accessed and managed by the `database.py` module in the backend.

4.2 Data Dictionary

- **User**
  - `Id(VARCHAR(255))` : Primary key. A unique identifier for each user, typically provided by Google OAuth[7].
  - `Gmail(VARCHAR(255))` : The user's Gmail address associated with their Google account.
- **Abort**
  - `Id(VARCHAR(255))` : Foreign key referencing `User(Id)`. Identifies which user hid the image.
  - `WebUrl(VARCHAR(255))` : The URL of the webpage where the image was detected and hidden.
  - `ImageUrl (VARCHAR(255))`: The direct URL of the hidden image.
  - `Reason (VARCHAR(255))`: The user-defined tag or keyword that matched the image and triggered the hiding.
  - `TimeAbort(TIMESTAMP)` : The time when the image was hidden. Part of the composite primary key.

5 Component Design

This section presents a procedural description of the major functions and modules that form the Safe Web system. Each procedure is described using pseudo-code to highlight control flow, logic, and local data where appropriate.

5.1 content.js

Function: scanImages(tags, setUrlCheck)

```
1. Get current visible area (top and bottom scroll)
2. Select all <img> elements
3. Filter out small or invisible images
4. For each filtered image:
    a. Skip if image already processed or is an SVG/mask
    b. Mark image as processed (add to setUrlCheck)
    c. Send CHECK_IMAGE message to background.js with:
       - imageUrl, tags, webUrl
    d. On receiving response:
        i. If any frame has score >= threshold:
            - Replace image with defaultMaskUrl
```

5.2 background.js

Function: authenticateUser()

```
1. Use chrome.identity.getAuthToken()
2. If successful:
    a. Fetch user info from Google API
    b. Save user info to chrome.storage.local
    c. Send POST request to /add_user on server
```

Function: CHECK_IMAGE message listener

```
1. Get user ID from local storage
2. Send image_url, tags, id, web_url to /check_image API
3. Wait for response and return it to content.js
```

5.3 popup.js

Function: Save Tags

```
1. Collect input tag values (non-empty)
2. Save tags to chrome.storage.local
3. Notify content.js with updated tags (TAGS_UPDATED)
```

Function: View History Button

```
1. On click, open history.html in new tab using chrome.tabs.create
```

5.4 history.js

Function: fetchAndRenderChart()

```
1. Get user ID from chrome.storage.local
2. Call three async functions:
    a. renderAbortPerWeb(userId)
    b. renderAbortPerTag(userId)
    c. renderAbortPerHour(userId)
```

Each rendering function:

```
1. Send POST request to server with userId
2. Parse response
3. Populate Chart.js charts (bar/line) with counts
```

5.5 server.py

Route: /check_image

```
1. Validate tags
2. Load image from URL (or base64/gif)
3. Preprocess image
4. Pass image and tags to clip.get_best_tag_per_image()
5. Return similarity results
```

Route: /add_user

```
1. Forward user info to database.registerToDB()
2. Return result
```

5.6 clip.py

Function: get_best_tag_per_image()

```
1. Encode all tags using CLIP tokenizer
2. For each image frame:
    a. Encode image
    b. Compute cosine similarity with all tags
    c. Select best tag with highest score
    d. Save (user_id, image_url, web_url, best_tag, timestamp) to DB
    e. Append result with index, score, and best_tag
3. Return list of frame results
```

5.7 database.py

Function: registerToDB()

```
1. Connect to MySQL
2. INSERT INTO User (gmail, id)
3. Return success or error
```

Function: getCountByWeb / getCountByTag

```
1. SELECT COUNT(*) GROUP BY WebUrl or Reason WHERE Id = ?
2. Return list of (label, count) results
```

Function: getCountByHour

```
1. Query last 24 hours: HOUR(TimeAbort), COUNT(*)
2. Return complete dict of 24-hour abort counts
```

6 Humnan Interface Design

6.1 Overview of User Interface

From the user's perspective, Safe Web is operated primarily through two interfaces: `popup.html` and `history.html`.

The `popup.html` interface allows users to manage their filtering tags. Users can add, edit, or remove tags using an intuitive tag editor. When tags are saved, they are stored locally and applied immediately across active pages. A confirmation message ("Tags saved!") provides feedback.

Additionally, the popup includes a "View History" button that opens `history.html` in a new tab. This page presents visual statistics on hidden images, categorized by website, tag reason, and time of day. Feedback is shown using bar and line charts built with Chart.js [5], allowing users to monitor and understand their browsing filter activity.

This user interface design emphasizes simplicity, clarity, and real-time responsiveness, enabling efficient content control with clear feedback.

6.2 Screen Images

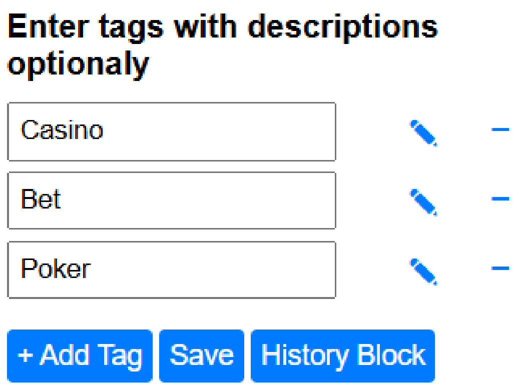
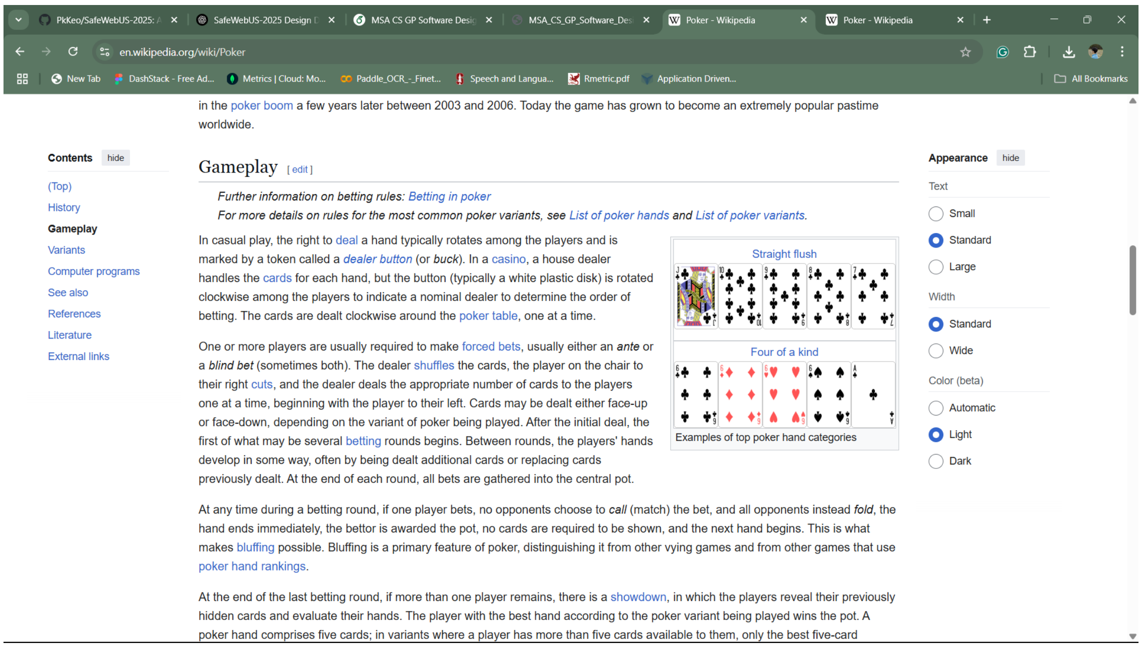


Figure 3: Extension popup interface

The popup interface (**Figure 3**) allows users to input and manage filtering tags. Each tag represents a keyword the system will use to identify and hide inappropriate images. Users can add new tags using the "+ Add Tag" button, edit existing ones via the pencil icon, or remove them using the minus sign. After editing, clicking "Save" stores the tags locally and updates the content filtering in real time. The "History Block" button opens a new tab showing a statistical overview of hidden images.



in the poker boom a few years later between 2003 and 2006. Today the game has grown to become an extremely popular pastime worldwide.

Contents hide

(Top)
History
Gameplay
Variants
Computer programs
See also
References
Literature
External links

Gameplay [edit]

Further information on betting rules: Betting in poker
For more details on rules for the most common poker variants, see List of poker hands and List of poker variants.

In casual play, the right to deal a hand typically rotates among the players and is marked by a token called a *dealer button* (or *buck*). In a casino, a house dealer handles the cards for each hand, but the button (typically a white plastic disk) is rotated clockwise among the players to indicate a nominal dealer to determine the order of betting. The cards are dealt clockwise around the poker table, one at a time.

One or more players are usually required to make forced bets, usually either an *ante* or a *blind bet* (sometimes both). The dealer shuffles the cards, the player on the chair to their right cuts, and the dealer deals the appropriate number of cards to the players one at a time, beginning with the player to their left. Cards may be dealt either face-up or face-down, depending on the variant of poker being played. After the initial deal, the first of what may be several betting rounds begins. Between rounds, the players' hands develop in some way, often by being dealt additional cards or replacing cards previously dealt. At the end of each round, all bets are gathered into the central pot.

Examples of top poker hand categories

At any time during a betting round, if one player bets, no opponents choose to *call* (match) the bet, and all opponents instead *fold*, the hand ends immediately, the bettor is awarded the pot, no cards are required to be shown, and the next hand begins. This is what makes bluffing possible. Bluffing is a primary feature of poker, distinguishing it from other vying games and from other games that use poker hand rankings.

At the end of the last betting round, if more than one player remains, there is a showdown, in which the players reveal their previously hidden cards and evaluate their hands. The player with the best hand according to the poker variant being played wins the pot. A poker hand comprises five cards; in variants where a player has more than five cards available to them, only the best five-card

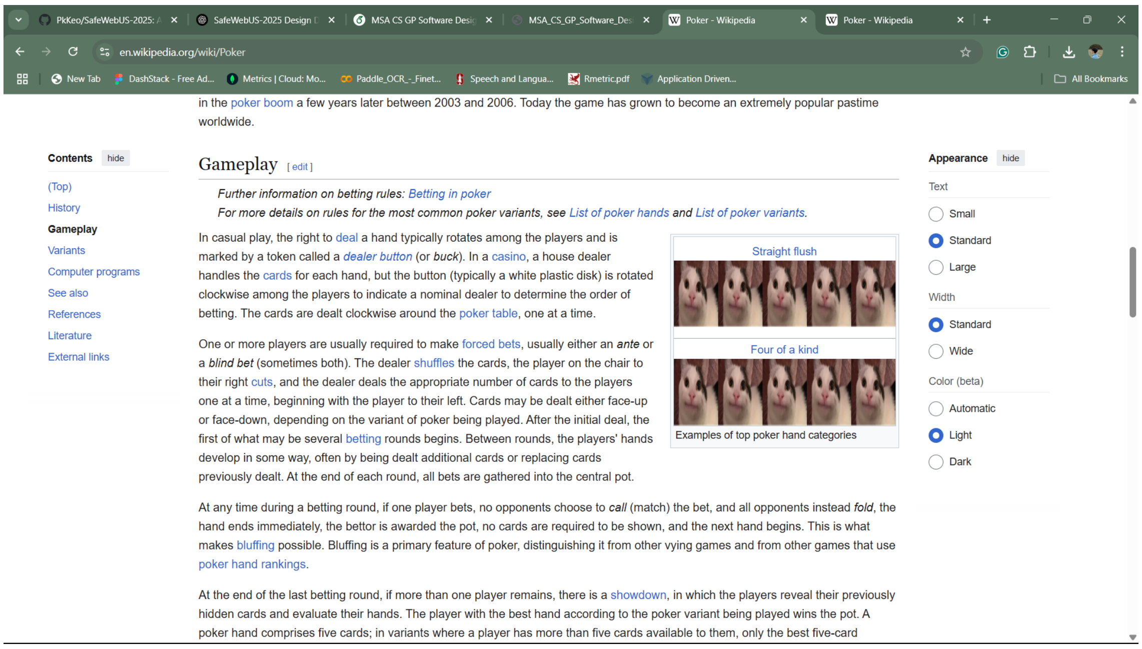
Appearance hide

Text
Small
[✓ Standard]
Large

Width
[✓ Standard]
Wide

Color (beta)
Automatic
[✓ Light]
Dark

Figure 4: Original website



in the poker boom a few years later between 2003 and 2006. Today the game has grown to become an extremely popular pastime worldwide.

Contents hide

(Top)
History
Gameplay
Variants
Computer programs
See also
References
Literature
External links

Gameplay [edit]

Further information on betting rules: Betting in poker
For more details on rules for the most common poker variants, see List of poker hands and List of poker variants.

In casual play, the right to deal a hand typically rotates among the players and is marked by a token called a *dealer button* (or *buck*). In a casino, a house dealer handles the cards for each hand, but the button (typically a white plastic disk) is rotated clockwise among the players to indicate a nominal dealer to determine the order of betting. The cards are dealt clockwise around the poker table, one at a time.

One or more players are usually required to make forced bets, usually either an *ante* or a *blind bet* (sometimes both). The dealer shuffles the cards, the player on the chair to their right cuts, and the dealer deals the appropriate number of cards to the players one at a time, beginning with the player to their left. Cards may be dealt either face-up or face-down, depending on the variant of poker being played. After the initial deal, the first of what may be several betting rounds begins. Between rounds, the players' hands develop in some way, often by being dealt additional cards or replacing cards previously dealt. At the end of each round, all bets are gathered into the central pot.

Examples of top poker hand categories

At any time during a betting round, if one player bets, no opponents choose to *call* (match) the bet, and all opponents instead *fold*, the hand ends immediately, the bettor is awarded the pot, no cards are required to be shown, and the next hand begins. This is what makes bluffing possible. Bluffing is a primary feature of poker, distinguishing it from other vying games and from other games that use poker hand rankings.

At the end of the last betting round, if more than one player remains, there is a showdown, in which the players reveal their previously hidden cards and evaluate their hands. The player with the best hand according to the poker variant being played wins the pot. A poker hand comprises five cards; in variants where a player has more than five cards available to them, only the best five-card

Appearance hide

Text
Small
[✓ Standard]
Large

Width
[✓ Standard]
Wide

Color (beta)
Automatic
[✓ Light]
Dark

Figure 5: Changes on website images

Figure 4 and **Figure 5** shows the work of the extension to hide the images that related to previous tags that a user saves in **Figure 3**.

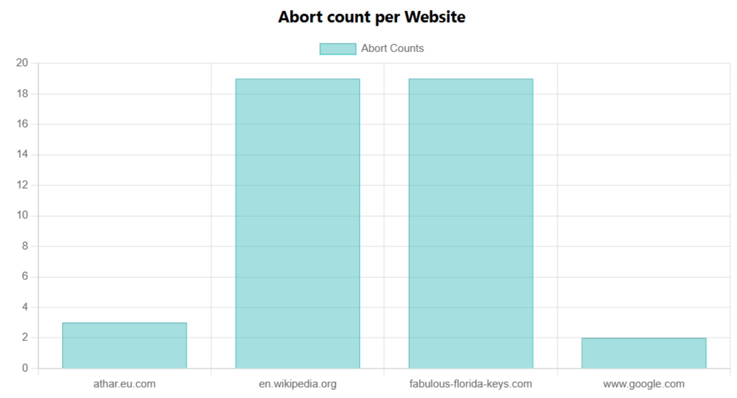


Figure 6: Chart of top host webs containing hidden images

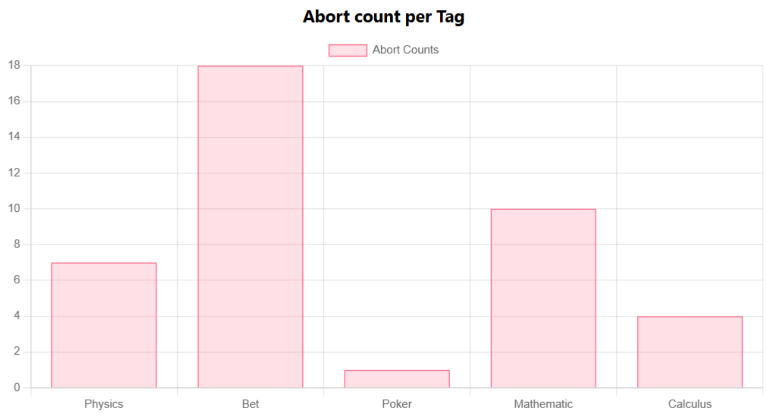


Figure 7: Chart of top tags matching hidden images

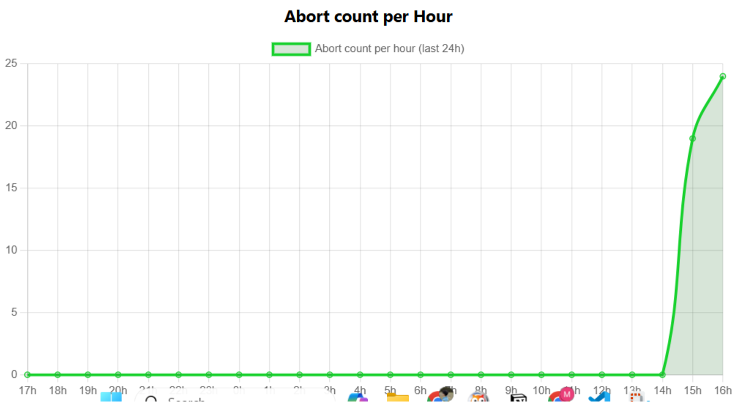


Figure 8: Chart of user last 24 hours hidden images

For the initial version of this extension. The developers propose 3 kinds of insights for user logs. The distribution of hidden image due to host web (**Figure 6**), tag (**Figure 7**) and time (**Figure 8**).

6.3 Screen Objects and Actions

The system includes two primary interactive screens: the Popup interface and the History view. Each contains UI elements (screen objects) that enable user interaction and provide feedback.

Popup Interface:

- **Tag Input Fields:** Users can enter or edit custom filtering tags. Each field is accompanied by an edit icon for inline updates and a remove icon to delete the tag.
- **Add Tag Button:** Creates a new empty tag input row.
- **Save Button:** Stores the current list of tags in local storage and applies them to content filtering in real time.
- **History Block Button:** Opens the History view in a new browser tab.

History View:

- **Web Chart (Figure 6):** Displays the number of hidden images per website host, helping users identify which domains are most frequently filtered.
- **Tag Chart (Figure 7):** Visualizes how often each tag was triggered in image blocking, offering insight into tag effectiveness.
- **Hour Chart (Figure 8):** Shows a 24-hour timeline of filtering activity, helping users understand when most blocking occurs.

Each screen object is designed for clarity, interactivity, and responsiveness, allowing users to easily manage settings and understand their filtering behavior.

7 Effectiveness, Limitations, and Future Work

7.1 Effectiveness

SafeWebUS-2025 achieves practical performance and usability by applying intelligent filtering heuristics. To reduce client-side overhead and unnecessary server load, the extension only checks images that appear within the current viewport (neighborhood range) and does so at regular intervals. Additionally, it filters out small or irrelevant images by enforcing a minimum size threshold, ensuring that only visually significant content is sent for analysis. This strategy enhances responsiveness while maintaining accuracy.

7.2 Limitations

While the system performs well in most scenarios, several limitations exist:

- **SVG images are ignored**: The extension does not process SVG images, as they are typically used for interface elements like icons and buttons. However, this could potentially be exploited as an evasion strategy for inappropriate content.

- **Video is unsupported**: Although video content poses a greater risk in terms of realism and impact, it is not currently supported. Video contains many frames and requires substantial server resources, as well as a model specialized in temporal or batch processing — unlike CLIP[8], which is designed for single-frame analysis.
- **Limited file type support**: The system currently supports standard raster image formats and common GIFs. Other less common or complex image types are not handled, primarily due to limitations of OpenCLIP [8] input compatibility.

7.3 Future Work

Future enhancements of the Safe Web system may include:

- Incorporating frame-based video analysis using models that support temporal information (e.g., TimeSformer [3] or ViViT [2]).
- Expanding file type compatibility by preprocessing additional image formats into OpenCLIP-compatible tensors [8].
- Introducing optional SVG analysis with stricter content heuristics to prevent evasion via vector-based content.
- Implementing adaptive image scanning frequency, dynamically adjusted based on scroll behavior or tab activity, to optimize performance and reduce unnecessary computation.
- Extending the system for supervised content monitoring, enabling parents, educators, or administrators to review filtered image logs and enforce safe browsing policies.
- Exploring semantic image search, where users can find images similar to a text query (e.g., "Ctrl+F for images"), using text-to-image matching for in-browser visual navigation.

References

[1] Aiven, "Aiven console documentation," 2024, accessed: 2025-04-29. [Online]. Available: https://docs.aiven.io/docs

[2] A. Arnab, M. Dehghani, G. Heigold, C. Sun, M. Lučić, and C. Schmid, "Vivit: A video vision transformer," *arXiv preprint arXiv:2103.15691*, 2021. [Online]. Available: https://arxiv.org/abs/2103.15691

[3] G. Bertasius, H. Wang, and L. Torresani, "Is space-time attention all you need for video understanding?" in *Proceedings of the International Conference on Machine Learning (ICML)*, July 2021.

[4] M. Cao, "SafeWebUS-2025: AI-powered Chrome extension for image filtering," 2025, accessed: 2025-04-29. [Online]. Available: https://github.com/PkKeo/SafeWebUS-2025

[5] C. Contributors, “Chart.js: Simple html5 charts,” https://www.chartjs.org, 2024, accessed: 2025-04-29.

[6] G. Developers, “Chrome extensions documentation: Getting started,” 2024, accessed: 2025-04-29. [Online]. Available: https://developer.chrome.com/docs/extensions

[7] Google, “Google identity platform: Oauth 2.0 authorization,” https://developers.google.com/identity/protocols/oauth2, 2024, accessed: 2025-04-29.

[8] mlfoundations, “OpenCLIP: Open-source implementation of openai’s clip models,” https://github.com/mlfoundations/open_clip, 2023, accessed: 2025-04-29.

[9] Oracle Corporation, “MySQL: The world’s most popular open source database,” 2024, accessed: 2025-04-29. [Online]. Available: https://www.mysql.com

[10] S. Ramírez, “Fastapi: Modern, fast (high-performance) web framework for building apis with python,” 2024, accessed: 2025-04-29. [Online]. Available: https://fastapi.tiangolo.com