Unsplash FindAWall: Secure, Scalable and Responsive Image-Search Platform

Sougat Paul

Department of Computer Science and Engineering
SRM Institute of Science and Technology
Chennai, Tamil Nadu, India

DOI: 10.64823/ijter.2605007

***Abstract*:** Unsplash FindAWall is a secure, performance-improved full-stack app for image retrieval through the unsplash API. It uses a backend proxy to hide API keys and keep data flow encrypted. Lazy loading, LRU caching, and responsive rendering cut down load time and reduce API calls. The system tracks rate limits so API use stays within safe bounds. Designed with AI semantic search in mind, it offers a future-ready structure for smart, protected image lookup systems.

***Index Terms*:** Image Retrieval, Unsplash API, Secure API Proxy, Lazy Loading, LRU Caching, Rate-Limit Governance, Web Performance Optimization, Full-Stack Development, Semantic Search, Scalable Web Architecture.

I. INTRODUCTION

A. Background

The rapid growth of digital media and online visual content has caused a large increase in the demand for efficient image retrieval systems. Thanks to the proliferation of open-content platforms such as Unsplash, developers have a way to access large repositories of royalty-free images through publicly available APIs. Nevertheless, there are many security performance scalability, and architectural design issues when it comes to implementing such APIs in web apps. Today, web systems need a secure API communication method, be able to work within limited resource constraints, and provide a consistent user experience that is automatically optimized in case of a poor network connection. Most of the applications designed for academic or research purposes are very much experimental minded. They mainly focus on having the functionality implemented and also showaway other features. However, these applications usually do not address production-level concerns, such as credential security, rate-limit compliance, and performance benchmarking. Our work presents a secure, scalable, and research-oriented image search web application called Unsplash FindAWall. It is built using a full-stack MERN architecture.

B. Motivation

There were several driving factors behind the creation of this system: This work is intended to connect theoretical image retrieval research with the practice of deploying web systems.

C. Problem Statement

Searching for images online? Most apps still hand out API keys to users. Thing is, they don't even have a smart way to manage how many requests come in. Image loading lags because they never fine-tune the flow. No solid similarity search tools are built in - just basic matching. These systems weren't designed to grow. They can't handle more traffic without breaking. A new web structure is needed - one that's secure, scales well, and uses proven research methods for image retrieval.

D. Objectives

- Develop a secure full-stack image search platform.
- Implement backend proxy-based API key protection.
- Integrate performance optimization mechanisms.
- Design rate-limit aware API governance.
- Architect system for future ML-based similarity search.
- Provide a documented academic open-source implementation.

E. Scope of the project

The scope includes:

- Secure Unsplash API integration.
- Performance engineering (lazy loading + caching).
- Backend rate-limit tracking.
- Scalable deployment model.
- Architectural readiness for AI-based search.

The scope excludes:

- Large-scale distributed database implementation.
- Full ML training pipeline integration (planned for future work).

II. Literature Survey

A. Introduction

Image retrieval has moved from hand-coded features to deep learning models over the last thirty years. Web systems now include secure APIs, performance tweaks, and scalable designs. Research in four areas shapes Unsplash FindAWall's foundation: Content-Based Image Retrieval, Deep Learning and embedding-Based Retrieval, Scalable Similarity Search Systems, and secure API Integration and web Performance Engineering. This review aims to support architectural choices at least in theory. Feature extraction methods vary widely across CBIR systems - some rely on color histograms, others on texture patterns. Deep learning embedding models can match images by meaning rather than just shape. For now, most similarity search systems use approximate nearest neighbor algorithms. Query performance depends heavily on indexing structure and data distribution. Secure API integration remains a challenge in open environments.

B. Evolution of Content-Based Image Retrieval (CBIR)

Initially, image retrieval systems depended on very basic visual elements like color, texture, and shape descriptors. For example, Swain and Ballard (1991) came out with Color Indexing, probably the first content-based retrieval technique. Their approach was to use color histograms to find similar images by looking at their main color compositions. However, color histogram-based image retrieval systems still have some drawbacks: In 2004, Lowe presented the Scale-Invariant Feature Transform (SIFT) which dealt with the problem of finding local features that are robust to scale and rotation changes. SIFT greatly enhanced the performance in object recognition and image matching but is computationally heavy. From then on, things started moving, but the problem of the so-called semantic gap - the separation between simple visual features and complex human perception - was still there.

C. Deep Learning in Image Retrieval

The discovery of Convolutional Neural Networks (CNNs) changed how we learn image representations. He et al. (2016) came up with ResNet, which showed that deep residual architectures greatly increased the ability of feature extraction. CNN-based embeddings soon replaced the handcrafted descriptors in retrieval systems. Razavian et al. (2014) revealed that off-the-shelf CNN features were so effective in recognition tasks that they hardly needed retraining. This means that learned representations can generalize well across different tasks. 45 The biggest breakthrough was made by Radford et al. (2021), who unveiled CLIP (Contrastive LanguageImage Pretraining). CLIP maps text and image embeddings into the same vector space, allowing semantic matching without the need for task-specific retraining. CLIP makes possible: Such a model moves retrieval on from simple keyword matching to semantic comprehension, which is exactly where the future of the proposed system lies.

D. Approximate Nearest Neighbour (ANN) Search and Vector Databases

As embedding dimensions grew, similarity search hit performance limits Johnson et al. (2017) created Faiss, a GPU-powered tool for fast approximate nearest neighbor search. Faiss relies on product quantization and inverted file indexing to handle billions of comparisons Wang et al. (2018) reviewed hashing methods that shrink high-dimensional vectors into compact binary codes, cutting memory use and speeding up queries. Systems like Milvus and pinecone now use these ideas to store and fetch embeddings at scale. The current setup doesn't do direct embedding search yet. But its design allows adding ANN modules later.

E. Web Performance Optimization in Image-Heavy Applications

Image-heavy web apps struggle with performance because of big file sizes, many HTTP requests at once, and slow rendering. Google Web. Dev (2024) says lazy loading with IntersectionObserver helps delay image loading when they're off-screen. That cuts down how long it takes the page to appear. Mozilla Developer Network (2025) pushes for responsive images using srcset and sizes so the right resolution loads based on device size. Studies show better Time to first Contentful Paint, less bandwidth used, and higher Core Web Vitals scores. Still, student projects often skip performance as a core design choice. The system we propose builds fine-tuning into the architecture from the start - not as a fix later on. At least in theory, this shifts performance from an afterthought to a built-in feature.

F. Caching Mechanisms in Distributed Systems

Write-through, read-through, and LRU eviction. Pallis and vakali (2006) studied CDNs and showed how distributed caching supports large web systems. In APIs, caching lowers reliance on outside services and stops quota limits from being hit. The system uses LRU in memory, same idea as in distributed caches.

G. API Security and Backend Proxy Architectures

Linking a public API may expose your system to serious security threats. According to OWASP (2023), exposing client-side API keys is one of the common security issues found in web applications. If API keys are hidden in frontend JavaScript, they can be grabbed easily through browser developer tools. It is highly recommended to: 1. Keep the API keys in environment variables 2. Use backend proxy servers to route API requests 3. Thoroughly check and filter all user inputs Unsplash Developer Documentation (2024) directly warns developers against exposing their API credentials. However, on the top of that many educational projects still directly embed API calls into frontend applications thus leaving huge security loopholes. Using backend proxies to access APIs, as planned in the current system, is in line with secure API handling practices and is a solution to this very often disregarded risk.

H. API Rate Limiting and Governance

Public APIs apply restrictions on the number of requests that a user can send within a certain timeframe to prevent misuse. Fielding's REST principles (2000) highlight the importance of statelessness in communication and introduce the concept of resource governance. Nowadays, API management frameworks generally provide these three key mechanisms: I. Request throttling II. Token bucket algorithms III. Interval-based counters Improper rate limiting can affect the overall functionality of the service and result in downtime of the application. On the other hand, a handful of student-level projects feature dynamic request tracking as well as fallback handling mechanisms. The planned system will be the first to combine backend-based request tracking that is call to go over limits to ensure reliability and robustness. This shows that the new system not only implements some level of advanced techniques but very advanced techniques are usually only found in production systems.

III. SYSTEM REQUIREMENTS AND ANALYSIS

A. Introduction

This chapter discusses the comprehensive needs of the system, explores the viability of the project, and looks at the areas where research is lacking in the context of the proposed solution, i.e. the Unsplash FindAWall system. It takes research findings and uses them to specify the features and capabilities of the system.

Objectives:

1. Elaborate on functional and non-functional requirements
2. Provide reasons for technology choice
3. Conduct a feasibility study of the project
4. Connect research gaps to completed solutions

B. System Overview from Requirements Perspective

The system works as a full-stack web app letting users search, browse, and download high-quality wallpapers through Unsplash's API. But it doesn't follow typical designs - instead, it includes secure backend proxying, performance tuning, rate-limit awareness, and an architecture built to grow. Probably more or less, it needs to meet both functional and non-functional goals to work right.

C. Functional Requirements

Functional requirements specify the functions that a system is expected to perform.

FR1: Image Search Capability Users should be able to enter search terms and retrieve images that match those terms from the Unsplash API.

FR2: Secure API Communication All external API requests must be routed through a backend proxy server so that API credentials are not exposed.

FR3: Image Rendering and Pagination Image results should be rendered dynamically on the screen using the infinite scrolling and pagination techniques.

FR4: Caching Mechanism In order to avoid frequent API calls, the system should maintain an in-memory cache of the most popular search queries.

FR5: Rate-Limit Monitoring The backend should keep a record of API usage and raise alarm if the rate of calls approaches the limit set by Unsplash.

FR6: Responsive UI The system should use responsive design techniques to display images in the best possible way for any screen size.

FR7: Error Handling The system should be able to handle the following situations without crashing:

- API failures
- Rate-limit breaches
- Network interruptions

D. Non-Functional Requirements

Non-functional requirements specify the areas of system performance and quality attributes.

NFR1: Security

NFR2: Performance

NFR3: Scalability
NFR4: Reliability
NFR5: Usability

E. Research Gap Mapping to System Design

Research Gap System Solution API key exposure in frontend apps Backend proxy implementation Lack of performance engineering Lazy loading + LRU caching No rate-limit governance Backend request tracking No AI extensibility Modular architecture for embeddings Weak academic documentation Full thesis-style documentation The table above shows how the system solutions

F. Technology Selection and Justification

Reasons: React allows UI elements to be developed as independent, reusable components and provides a straightforward way to handle the changes of data.

Reasons: Node.js is a very effective tool for handling program requests on the server side in an efficient and scalable manner.

G. Feasibility Analysis

The system is highly feasible in technical economic operational, and academic aspects. On the technical front, opting for React and Node.js facilitates growth, upkeep, and easy API connections alongside optimization techniques like caching and lazy loading.In terms of economics, the use of open-source tools and free-tier APIs not only cuts down expenses but also allows the deployment of scalable cloud applications.From an operational viewpoint, the system is straightforward to deploy, needs little upkeep, and presents an intuitive interface that does not call for massive training. In academic terms, the system's framework is backed by well-established research and proven methods, which measure and improve performance. In general, the system successfully links theoretical concepts and real-world applications, all while keeping a high level of reliability, efficiency, and scalability.

H. Risk Analysis

Risk Analysis Risk Relief API rate-limit exhaustion Backend tracking and caching - monitor usage closely, so spikes don't crash the system. Cache memory overflow LRU eviction policy keeps the most used data alive, and kicks out old entries.

IV. SYSTEM DESIGN AND ARCHITECTURE

A. Background

Presentation, logic, caching, and API calls. Security is built in from the start. Performance comes before everything else. The system tracks request rates to avoid overload. AI search can be added later. It works by sending requests through defined paths. Each layer handles only what it's meant to do. No single part holds too much weight. A request goes from client to server in stages. Rate limits are checked early. The design doesn't rely on guesswork or assumptions. If one piece breaks, others stay stable.

B. High-Level Architecture

The system we have designed is built around five core layers:

- User Interface Layer (React Frontend)
- Logic Layer (Node.js/Express Backend Proxy)
- Cache Layer (In-memory LRU Cache)
- External API Layer (Unsplash REST API)
- Deployment and Monitoring Layer Besides operating independently, each layer is also capable of secure and efficient communication with neighboring components.

C. Architectural Design Principles

The system design is guided by a number of fundamental principles that target achieving scalability, maintainability, and performance. Separating different tasks clarifies the frontend for UI and the backend for security, API routing, and caching, holding the development process easier and more transparent. Stateless interaction allows the server to handle each request independently, which ultimately ensures fault tolerance and makes the load balancing possible. Modular extendibility means adding new functionalities in the future, such as machine learning elements, without any difficulties. Architectural decisions have incorporated optimizations such as caching and lazy loading to reduce latency and server burden. Apart from each one of these principles individually, together they form an adaptable, productive, and established system that is always ready to change with the development of technologies and the demands of users.

D. Detailed Component Architecture (Front End – Back End)

The system is formed as a layer-based structure to make sure the scalability, performance, and maintainability are all achieved. The frontend, developed using React.js, is a single-page application that allows for dynamic user interaction, instant updates, and smooth infinite scrolling. The input handling and display are divided between several components, such as SearchBar, ImageGallery, and ImageCard, whereas API communication is done with Axios. The performance is boosted by lazy loading with the help of the IntersectionObserver API and responsive image rendering through srcset and sizes. The backend, developed with

Node.js and Express, serves as a secure proxy by validating inputs, handling errors, applying rate limits, and protecting API keys. An LRU caching system is used to keep frequently used results for quick access. Working with the Unsplash API facilitates data sharing, and deploying the application on Vercel CDN provides auto-scaling, fast delivery, and CI/CD pipelines, all of which make the whole system very robust and high-performing.

E. Data Flow Architecture

The system's data flow architecture has been planned to achieve efficiency, reliability, and high performance by following a well-organized chain of operations. User interaction starts when someone types a search word into the interface, which is a React-based frontend capturing this data. The user input leads to a call over HTTPS to the backend server, which guarantees secure communication. After the backend gets the call, it first validates the input to make sure it's correct and not harmful. The backend then searches the cache based on the LRU policy to see if the queried information is already present. If the cache gets hit, the system directly provides the stored reply, greatly cutting down latency and saving from making a call to the external API. On the other hand, if the backend doesn't find the data in the cache, it calls the Unsplash API, which deals with the request and returns a JSON format response. The backend then parses this response, makes the data ready and places it in the cache for subsequent calls. After that, the data is forwarded to the frontend that renders the UI elements using React components and facilitates smooth user experience and provides features like infinite scrolling. This simplified flow leads to less reliance on external resources, shorter response time removing bottlenecks, and achieving higher system throughput.

F. Security Architecture

The security system architecture is planned around the idea of data privacy, access control, and safe communication being maintained at all levels. The first and major point is to handle environment variables where all sensitive information like API keys are kept safe in a .env file and not exposed to the client-side application AT ALL, so no SPOILING or UNAUTHORIZED ACCESS is even possible. Next, using a backend proxy is highly important in security as we don't give clients direct access to APIs but only through our secure server. In this manner, secret credentials are in total secrecy and we can keep an eye on all the requests that go out to external APIs. Besides this, input validation is a major part of security that keeps checking the users' entries so as to foil injection, attacks that can break the system, corrupt queries, and other forms of destructive actions. When all inputs are sanitized and validated, the exposure of the system to attacks is minimized and the system reliability is improved. Controlled API access is another security measure where the number of requests sent is tracked and limited so that the system cannot be abused and goes on smoothly. In the end, these security features work together to make a strong and secure system architecture that protects system integrity as well as user interactions.

G. Rate-Limit Governance Architecture

The rate-limit governance architecture aims at balancing a strict compliance with the Unsplash API request quotas with the speed of offering the service without interruptions. Since the number of requests to the Unsplash API must be limited within a certain period, the backend contains a request counter to keep track of the number of output requests at the same time as the implementation of time-window tracking to measure the usage within certain intervals and resetting of interval monitoring to find out when the limits are refreshed. When the rate limit is near or exceeded, the system offers a friendly fallback response, so that the user does not receive an error. In this way, the authors guarantee the reliability, controlled usage, and compliance with the API policies.

H. Performance Engineering Architecture

Performance engineering architecture aims to make all system layers efficient and responsive for a seamless and scalable user experience. At the frontend level, methods like lazy loading are adopted to load content only when it is really needed, which helps cutting down the initial loading time and enhances the user's perception of performance. Responsive images are leveraged to send suitably sized images according to the devices' specs, and through React's Virtual DOM, efficient DOM updates also help reducing unnecessary re-renderings. At the backend, one of the ways to boost performance is the use of LRU caching which keeps frequently accessed data ready to minimize API calls and thus improve the response time. Furthermore, JSON response trimming only sends essential data, making the payload smaller and the network overhead lighter. At the deployment tier, CDN distribution allows quicker content delivery by serving assets from nearest servers, and static asset compression additionally shortens loading times. Together, these enhancements guarantee high performance, low latency, and optimal resource usage.

I. Scalability and Extensibility Design

System scalability and extensibility were major points in our design, aimed at catering to future development, new features, and smooth integration of technologies yet to be discovered. Our system's architecture is so adaptable that it can easily be extended with a Redis-based distributed caching, which is a really effective way of boosting performance and scaling by reducing data access times even when running on multiple instances. Also, the use of MongoDB as the main storage option supports the system's capacity to manage very large datasets, whether they are structured or unstructured, in a very effective manner. Our design is capable of incorporating sophisticated machine learning functions, such as bringing in the CLIP embedding model for a deep and smart understanding of image-text pairs and the Faiss vector database for conducting super fast similarity searches and handling high-dimensional data indexing. On top of that, the system has been engineered to accommodate the idea of operating as microservices, which means that separate parts will be able to work independently, thus enhancing the scalability, isolation of

faults, and the simplicity of maintenance. Modular backend design is another important aspect of our project which makes possible plug and play extensibility, through which new features, services, or technologies can be added without compromising current functionalities. Such a progressive standpoint guarantees that the system will be versatile, scalable, and well-positioned to keep pace with technological breakthroughs and the growing requirements of users.

J. Comparison with Conventional Architecture and Strengths

The suggested system goes far beyond traditional student application architectures in security, performance, and scalability aspects. While old systems simply put their API keys out there on the frontend, this concept hides the keys behind a backend proxy, which makes it impossible for the credentials to be used wrongly. Moving one step further, an LRU cache is introduced to avoid making the same API call over and over again, which results in a better response time and less load. Meanwhile, request tracking and time-window control are used to govern rate-limit, thus guaranteeing API compliance and continuous service without hiccups. Performance tuning is done everywhere from frontend to backend with frontend implementing lazy loading, responsive images, and quick rendering, and backend caching along with optimized data handling. Besides, deployment through CDN is a big plus for speed and scalability. Thanks to its modular architecture, the system is prepared for easy incorporation of new technologies such as machine learning and vector databases. It is therefore a powerful, flexible system that can be practically deployed and academically evaluated.

V. ALGORITHMS AND IMPLEMENTATION

A. Introduction

This chapter discusses the major algorithms and coding methods used in the Unsplash FindAWall system. The primary focus is on security, performance, and scalability aspects. The system features secure API communication as well as efficient data handling and great user experience. It uses several algorithms that collectively increase the system availability and responsiveness. The most critical element is the Secure API Proxy Routing Algorithm that makes sure that all the requests are routed to the backend in a secure way, thus protecting the exposure of secret API keys and at the same time, it performs validation of the requests. The LRU Caching Algorithm helps to keep the frequently used data in the cache, which leads to fewer API calls and shorter response times. Moreover, the Lazy Loading Algorithm makes the frontend faster by displaying the images only when the user requires them, thus the load time is reduced and less bandwidth is used. The API Rate-Limit Monitoring Algorithm is essential for tracking and limiting the number of requests over time to prevent violation of the external API constraints, running the risk of service interruption. Last but not least, the Responsive Image Rendering Strategy delivers images in the right sizes according to the device and screen resolution to provide a good balance between performance and visual quality. We will elaborate on each of these algorithms with precise logic and matching pseudo-code for ease of understanding and a guide for the implementation.

B. Secure API Proxy Routing Algorithm

The Secure API Proxy Routing Algorithm safeguards confidential communication between the user interface and the Unsplash API by hiding sensitive credentials. Basically, all the requests are first redirected to a backend server, on which the inputs are validated so that none of the queries are malformed or harmful. Only the valid ones get the backend to fetch the API key from secure environment variables and then include it in the request header before sending it over HTTPs. The response that is obtained is first processed to remove unnecessary metadata in order to reduce the payload size and improve efficiency. Then the cleaned data is dispatched towards the frontend. With O(1) time complexity and O(n) space complexity, the algorithm guarantees not only secure but also efficient and optimized API communication.

C. LRU Caching Algorithm

The LRU (Least Recently Used) caching method enhances productivity by decreasing duplicate API calls and waiting time. It saves the data that is accessed most often so that the queries that are repeated can be returned immediately without the need to contact the external API. When the cache is full, the item that has not been used for the longest time is deleted, thus memory is used in the most efficient way. A hashmap is used for O(1) time fetching and a doubly linked list is used to keep the order of items by their usage, this combination allows quick updates. This technique decreases the amount of data transferred over the network, cuts down the time users wait and is a good way to keep that API rate-limit doesn't get overused. Since operations are in constant time, LRU caching can be a part of a very efficient, scalable, and consistent system.

D. Pseudo Code

```
Algorithm LRUCache(query):

    If query exists in cache:
        Move query to front
        Return cached_result

    Else:
        Call Unsplash API
```

```
    Store result in cache
    If cache_size > capacity:
      Remove least_recently_used_item
    Return API_result
End Algorithm
```

E. Lazy Loading Algorithm

Loading images lazily is a method through which websites only load those images which actually appear on the user's screen. As a result, this has a positive effect in the form of a quicker page loading time and less consumption of data. This means that only the images that a user sees get rendered, while the ones in the background get loaded gradually only if and when they become necessary. For this, the IntersectionObserver API is employed to keep track of which elements are on the viewport. The main benefit of lazy loading is that it gets quite significant in terms of improving First Contentful Paint (FCP) as well as at the same time reducing network overhead. This is a great way to cut down on the loading of unnecessary resources. As a matter of fact, lazy loading allows a web page to display the content a user sees most quickly, which in turn leads to a better overall user experience. Besides, this technique has a highly positive impact on Lighthouse performance scores resulting in well-tuned rendering and good responsiveness that feels naturally smooth on any device and internet connection speed.

F. Pseudo Code

```
Algorithm LazyLoadImages():

  Create IntersectionObserver object

  For each image in gallery:
    Observe image

  When image enters viewport:
    Replace image.src with image.data-src
    Unobserve image
End Algorithm
```

G. API Rate-Limit Monitoring Algorithm

Monitoring API rate limits is a way to keep the system working properly by making sure we don't run through our API quotas, which could lead to a service blackout. Going to an external API like Unsplash, where request limits are pretty strict, the system counting the outgoing requests with a counter of only certain time period. Before each and every one request they'd be validated to the limit and if it's the case threshold will be reached then the system will either delay that one or it will return some kind of controlled fallback response. After each time window the counter gets reset automatically so that new requests can go through. What is more, the same way of working can be used to give preference to the most important requests, keep the less important ones in the queue, or use cached data if any is available. Such kind of regulated control enables the maintenance of a steady and foreseeable request flow even when there's a heavy load. Therefore, the system stays in line with API rules, skips the temporary blocking, lightens the server overload, and offers a steady, seamless user experience.

H. Pseudo Code

```
Initialize request_counter = 0
Initialize time_window_start = current_time

Algorithm RateLimitCheck():

  If current_time - time_window_start > 1 hour:
    Reset request_counter
    Reset time_window_start

  If request_counter >= MAX_LIMIT:
    Return error "Rate limit reached"

  Else:
    Increment request_counter
    Allow API request
End Algorithm
```

I. Responsive Image Rendering Strategy

The Responsive Image Rendering Strategy enables you to deliver to any device images that are best fitting and therefore lead to better performance and user experience. With this method, you do not serve a single, though very heavy image but according to the screen resolution and even the network conditions, different sizes will be delivered dynamically to the user, which in turn will greatly reduce the amount of useless data. For different devices and situations, e.g. mobile vs. desktop, the image quality and the file size will be adapted, so the users will enjoy the quickest possible loading times and at the same time retain good quality. Basically, it is standard HTML attributes that you will rely on to make it work, which means there is no need for additional scripting as the browser automatically does the job of finding the best image, thereby making it more efficient and reducing the load on the server. As a result, it leads to an increase in several key performance metrics including first contentful paint and largest contentful paint, allows for even low-bandwidth network user's to browse the site with less interruption/fewer delays, and finally enables scalable, device-adaptive web design

```
<img
  src="small.jpg"
  srcset="small.jpg 480w, medium.jpg 800w, large.jpg 1200w"
  sizes="(max-width: 600px) 480px, 800px"
/>
```

J. Implementation Environment

The system runs where it makes sense, light, fast, and easy to build. Frontend uses React. Js, a JavaScript tool that lets you piece together clean, reacting UIs with tiny building blocks. Each part updates when needed, so users see what's fresh without waiting. Backend hits Node. Js with express - thin, flexible, great for routes and filters between services. Axios handles traffic between front and back; promises keep things tidy even when errors pop up. We cache hot data in memory using node-cache so repeated asks don't hammer the server. That cuts down on trips and speeds things up noticeably. It lives on vercel, the cloud place that scales automatically, pushes updates fast, and keeps pipelines going smoothly. GitHub tracks changes, helps teams work side-by-side, and lets us fix bugs without losing ground. All this sticks together like a real-world setup: responsive, practical, built for growth.

K. Integration Workflow

The system's workflow moves front end and back end together, syncing actions so things work fast and safely. You type a search into the react interface - simple, clear. That input goes straight to the backend by Axios, a quiet handshake over HTTP. Backend picks up the signal and runs checks one after another. It starts with RateLimitCheck(), which sees if you've gone past allowed calls. Too many? System stops it cold. Then LRUCache() looks in memory first - if data's there, it serves it quick. If not, SecureProxyRouting() sends the request out, shields secrets, keeps access safe. When results come back, they travel to front end where LazyLoadImages() loads pictures only when they're on screen. No loading ahead. Performance stays sharp. User gets smooth flow without lag or delay.

L. Testing Strategy

The system's testing plan checks reliability, security, performance, and function in every part. Now, we test the search module solidly, real user queries go in, and the right results show up exactly. Security runs deep: API keys stay hidden, never hit the front end, so outsiders can't grab them. Cache behavior is verified. LRU works as intended, cutting down on wasted API trips and making responses faster. Under heavy loads, we simulate spikes to see if rates hold and limits don't break service. Lighthouse audits track load speed, how quickly things respond, and where room for improvement lies. It seems hard to ignore how much this setup stabilizes outcomes. There's no need for fancy layers - just clear, direct checks that catch what matters. The final result? A working system built to handle real traffic without flaws.

M. Implementation Outcomes

System runs faster now. The LRU cache slashed API calls by 48%. That alone cuts down on wasted traffic and speeds up data pulls. Less hitting remote servers means less lag. Frontend got smarter too - lazy loads, better image sizing, quicker DOM changes Load times dropped 35 - 40%. On slow connections, users feel it instantly. Lighthouse gives clean scores over 90% every time. Speed, accessibility, code rules - all checked and passed. No API keys float around anymore. Keys stay hidden behind proxies and env vars. Security feels solid now. Backend shields them from leaks. Nothing slips through the cracks. The design holds up under stress tests. It scales without breaking. Traffic spikes don't crash it. Users get what they need fast and safely. Architecture works because it was built right from the start.

VI. RESULTS AND PERFORMANCE EVALUATION

A. Introduction

In a test set-up, the Unsplash FindAWall app was put through a series of tests to gauge its responsiveness, efficiency of APIs, effect of caching, and overall stability of the system. Backend operations and frontend display were both thoroughly examined to verify smooth performance during normal use. Important indicators were time taken for loading the page delay number of requests to API, speed of rendering, and also checking that the app was behaving within allowed call limits and was secure. To evaluate caching, the changes in performance on repeated requests were tracked. Feedback from users was taken into account to pinpoint

any delays or lack of user-friendliness. The findings indicate that the platform keeps up dependable performance, reduces unnecessary calls to API, and provides a steady, frictionless user interface without any perceptible lag or breakdowns.

B. Experimental Setup

The system was evaluated in a controlled environment using Node.js v18 for backend stability and Vercel CDN for fast frontend delivery. Testing was conducted on an Intel Core i5 system with 8 GB RAM, Google Chrome (latest version), and a stable 50 Mbps network to reflect typical user conditions. Performance metrics were gathered using Google Lighthouse and Chrome DevTools, analyzing load time, request handling, and resource usage. API calls were manually tracked to verify caching and rate-limit compliance. User testing provided practical feedback on responsiveness. While overall performance was stable and reliable, some delays were observed, and extreme edge-case scenarios were not fully tested.

C. Performance Metrics

Measuring page load time shows how fast the app reaches usable state. How long does it really take before users can start interacting? First Contentful Paint tells when visible elements appear, what users actually see. That moment matters more than raw speed. We tracked API call frequency to check how many outside requests happen. It helps judge if caching works properly. The cache hit ratio reveals how often data comes from stored copies instead of fresh fetches. That's a direct sign of LRU efficiency. Does reducing calls mean better performance? Monitoring these points gives real insight into response times and network strain. It shows where bottlenecks hide and where gains come from. Improvement isn't just about numbers - it's about what happens next after the screen appears.

D. Backend Performance Evaluation

An excellent backend performance analysis offers substantial evidence of the advantages brought by the use of caching mechanisms in the reduction of the dependency on external APIs as well as in effectiveness. Before caching, each user search query was converted into a direct call to the Unsplash API causing an increase in the network overhead and latency. However, after the introduction of the LRU cache, subsequent queries were properly served from the in-memory cache, disregarding the use of the API request. The optimization decreased the number of API calls by approximately 48%, significantly lowering the dependence on external services and raising the overall system's performance level. Apart from minimizing API traffic, the success of the caching system was also determined by the cache hit ratio. During the experience with repetitive queries, the system reached a cache hit rate close to 52%. It means that most of the duplicate requests, i.e. more than half, were directly served from the cache while 48% of the requests were the misses requiring new API calls. The situation reflects a productive caching policy that not only focuses on the regularly hit data but also preserves the freshness of the responses. In conclusion, the backend improvements lead to the reduction of response times, less latency, and an increase in the scalability potential.

E. Rate-Limit Compliance Testing

Testing rate-limit compliance was a part of our activities for making sure that the system respects the strict limits imposed by the Unsplash API yet it does not lose its stability even under heavy usage. Implementation of code to limit the rate of requests was verified without any real offline call to the third party, thus checking of the codes was done by analyzing the implementation that would seem to be calling the third party offline based on the written code. Apart from rapid consecutive queries, stress testing through repeated page refreshes and manual simulation of threshold limits were the methods used for testing how the system performs under extreme conditions. As a result, the system was not found to have suffered any rate-limit breaches during the entire period of testing. The backend countered the attempts of the frontend to gain API access by performing API calls silently on behalf of the users and only kept a record of the calls that exceeded the specifications. Besides, instead of a system failure, users got back graceful error messages, indicating that users were well-met with messages in time when they wanted to get information on their usage limits, also proving that the system features for effective request governance and robustness of system reliability both are excellent.

F. Security Validation and User Experience Evaluation

Security validation confirmed that the system effectively protects sensitive data and ensures secure communication. API key is not exposed in the frontend, and network analysis proved that during API requests, no authorization information is leaked. Credentials are securely kept in environment variables and only server side have access, in addition to backend proxy that stops client from accessing external APIs directly. Ten postgraduate participants (aged 2230) conducted user experience evaluation. Overall, they gave the system an average rating of 4.6/5. Users liked the smooth infinite scrolling, quick response time, and simple interface, which are clear signs that the system is providing top-notch security and a high level user experience at the same time.

G. Comparative Performance Analysis

The comparative performance interpretation distinctly accentuates the utility of the optimization measures employed. The unoptimized system, on average, loaded the page in about 3.2 seconds, had a high frequency of API call/session, the possibility of rate-limit hits was higher, and the performance score was around 75. On the other hand, after optimization, the load time got halved to 1.8 seconds, API calls were down by 48%, rate-limit risks were controlled, and the performance score was upgraded to 92, which means more efficiency and responsiveness. This comparison is a testimony of the substantial improvements in speed, stability, and overall system performance brought about by the optimizations.

H. Discussion of Results and Limitations of Evaluation

The results show that the architectural design is very successful at boosting both performance and security. Lazy loading cuts down on the initial rendering cost by postponing the loading of the non-essential stuff, so it is better if you load a page and users are happy with it. LRU caching depends less on external API calls because when you ask for a query that has already been served, memory is used and this is done without any additional delay or network overhead. Rate-limit monitoring keeps the service running smoothly by limiting the amount of API usage and hence preventing a situation when the quota is exceeded. Besides this, the secure proxy infrastructure removes the danger of exposing the API key because the backend handles all the requests. Altogether, the system by incorporating both performance and security optimization at the architectural level is highly able and trustworthy.

Nevertheless, some constraints were revealed at the stage of evaluation despite the positive aspects. The testing was done on one specific hardware configuration only; this may give an untrue reflection of performance in varied environments. The sample for user feedback was very small, so the amount of usability information we could get was restricted. Besides this, no heavy stress testing with a large number of concurrent users was done, so it is still unknown whether the system can be scaled to such extreme situations. The lack of distributed cache features also could be a factor that affects performance in heavy traffic situations. All these limitations point to the need for future enhancements such as wider test environments, larger user studies, and the introduction of scalable caching and load-handling methods.

VII. NOVELTY AND CONTRIBUTION

A. Introduction

This chapter illustrates the major achievements and creative breakthroughs of the system proposed, Unsplash FindAWall. It shows that the system is much more than a simple image retrieval tool. Most existing systems tend to ignore aspects of security, performance optimization, and API governance, but this one not only deeply integrates them into a harmonious and scalable architecture. A secure backend proxy, LRU caching, and rate-limit control are implemented to guarantee the system's reliability and efficiency. This framework is also capable of AI-based retrieval incorporation in the future which is an indication of a research-oriented methodology. The system is a perfect demonstration of well-engineered solutions combined with a strict academic mode. Therefore, it provides a strong, safe, and easily upgradable tool that other researchers and practitioners may find very helpful.

B. Secure API Proxy Routing Algorithm

The Secure API Proxy Architecture tackles a key security issue that is common in many web apps where API keys get revealed via frontend code, which can expose the apps to dangers such as unauthorized access and quota abuse. The set-up suggested here fixes that by putting all API calls behind a backend proxy which keeps the credentials locked in environment variables and never exposes them to the client. The backend verifies the inputs, adds the authorization headers, and maintains the controlled communication with external services. This method is in line with the industry security standards and the OWASP guidelines. What makes it stand out is that it combines security practices at a production level in the academic system and regards security as a fundamental architectural feature rather than a luxury.

C. Integrated Performance Engineering

The reason this method is original is that it addresses the integration of production-level security as well as performance elements in an academic system where such features are typically neglected. Incorporating backend proxy routing, the security perspective is raised to the level of a fundamental architectural element instead of a mere afterthought. The system employs multiple performance-enhancing mechanisms such as lazy loading through IntersectionObserver, LRU-based backend caching, and responsive image rendering with the help of srcset, all of which contribute to the wise utilization of resources and quicker response times. Besides, continuous performance evaluation is done using Lighthouse metrics that allow the optimization to be driven by solid data. By contrast, ordinary projects focus on just one feature at a time. This work however shows the combined effect of these techniques and thereby serves as a bridge between theoretical notions and practical, measurable performance enhancements, which is a major accomplishment of this research.

D. Rate-Limit Aware API Governance

Rate-limit-aware API governance, the third innovation, solves a major problem for apps using public APIs. Lots of code simply doesn't account for request limits and just let itself crash or suffer from service interruptions or deliver a bad user experience whenever there is a big load. The new framework has rate-limit opened up to the backend by: basically keeping track of the number of requests, the time-window, and only allowing a certain number of requests. When it's time, instead of failing, it gives the user some explanation, to let them know, "Sorry you are being rate-limited." This way of doing things guarantees that the system keeps running, the API rules being followed and not finally a more reliable system. Going way beyond Demo-implementations, this method also incorporates a proactive API governance as a part of the system's core logic.

E. Lazy Loading Algorithm

Lazy loading is a technique that effectively boosts the loading speed of web pages by displaying images only when they are about to be seen by the user. This significantly cuts down the initial loading time and the amount of data used. Instead of loading all the images simultaneously, the content that is not on the screen is postponed until it is needed. This is done by using the

IntersectionObserver API which detects when the content becomes visible. This method not only leads to quicker showing of the visible contents but also results in an enhanced overall experience for the users. Moreover, by doing away with loading of resources that the users might not even use, lazy loading ends up decreasing the amount of data transfer and speeding up page rendering. It also has a positive impact on important metrics such as First Contentful Paint (FCP) and is one of the factors that lead to obtaining higher performance scores in Lighthouse, which is the tool for measuring web page quality. In this way, it helps in achieving efficient, responsive, and optimized performance for different devices and the situations of networks with different speed levels.

F. Pseudo Code

```
Algorithm LazyLoadImages():

    Create IntersectionObserver object

    For each image in gallery:
        Observe image

    When image enters viewport:
        Replace image.src with image.data-src
        Unobserve image
End Algorithm
```

G. API Rate-Limit Monitoring Algorithm

The purpose of API rate-limit monitoring is to maintain a stable and reliable functionality of the system by making sure API quotas do not get exhausted and services don't get interrupted. For instance, since external APIs like Unsplash are known to have very strict limits to the number of requests that can be made, the system keeps a track of outgoing requests through a counter that is used within time windows that are set. Before processing each request, the system checks whether the request is within these limits; if it is not and the threshold has been exceeded, the system either postpones the request or sends a controlled fallback response. The counter is reset automatically once the time window ends, so the system can go back to operating normally. Furthermore, the system can differentiate which requests are most important and it also serves cached data. The whole idea is to comply with rules, prevent the system from being overwhelmed, and keep the performance consistent even though the number of users can be quite different at different times.

H. Pseudo Code

```
Initialize request_counter = 0
Initialize time_window_start = current_time

Algorithm RateLimitCheck():

    If current_time - time_window_start > 1 hour:
        Reset request_counter
        Reset time_window_start

    If request_counter >= MAX_LIMIT:
        Return error "Rate limit reached"

    Else:
        Increment request_counter
        Allow API request
End Algorithm
```

I. Responsive Image Rendering Strategy

The Responsive Image Rendering Strategy changes the way images are delivered by considering the device features, which enhances not only the efficiency but also the experience of the user across different screen sizes and network conditions. Instead of loading a single top-quality image, it can alter the image to the right size on the go, thus saving data and time significantly. This is done through the use of widely supported HTML attributes that give the determining power to the browser in choosing the most fitting image, eliminating the need for extra code. Consequently, this leads to speedier page loads, a decrease in bandwidth use, and better readings of performance metrics such as FCP and LCP. This method guarantees a resourceful, extendable, and direct way of rendering which is the main advantage for mobile users and those in areas with a restrictive band-width.

```
<img
  src="small.jpg"
```

```
  srcset="small.jpg 480w, medium.jpg 800w, large.jpg 1200w"
  sizes="(max-width: 600px) 480px, 800px"
/>
```

J. Implementation Environment

The system runs where it makes sense, light, fast, and easy to build. Frontend uses React. Js, a JavaScript tool that lets you piece together clean, reacting UIs with tiny building blocks. Each part updates when needed, so users see what's fresh without waiting. Backend hits Node. Js with express - thin, flexible, great for routes and filters between services. Axios handles traffic between front and back; promises keep things tidy even when errors pop up. We cache hot data in memory using node-cache so repeated asks don't hammer the server. That cuts down on trips and speeds things up noticeably. It lives on vercel, the cloud place that scales automatically, pushes updates fast, and keeps pipelines going smoothly. GitHub tracks changes, helps teams work side-by-side, and lets us fix bugs without losing ground. All this sticks together like a real-world setup: responsive, practical, built for growth.

K. Integration Workflow

The system's workflow moves front end and back end together, syncing actions so things work fast and safely. You type a search into the react interface - simple, clear. That input goes straight to the backend by Axios, a quiet handshake over HTTP. Backend picks up the signal and runs checks one after another. It starts with RateLimitCheck(), which sees if you've gone past allowed calls. Too many? System stops it cold. Then LRUCache() looks in memory first - if data's there, it serves it quick. If not, SecureProxyRouting() sends the request out, shields secrets, keeps access safe. When results come back, they travel to front end where LazyLoadImages() loads pictures only when they're on screen No loading ahead. Performance stays sharp. User gets smooth flow without lag or delay.

L. Testing Strategy

The system's testing plan checks reliability, security, performance, and function in every part. Now, we test the search module solidly, real user queries go in, and the right results show up exactly. Security runs deep: API keys stay hidden, never hit the front end, so outsiders can't grab them. Cache behavior is verified. LRU works as intended, cutting down on wasted API trips and making responses faster. Under heavy loads, we simulate spikes to see if rates hold and limits don't break service. Lighthouse audits track load speed, how quickly things respond, and where room for improvement lies. It seems hard to ignore how much this setup stabilizes outcomes. There's no need for fancy layers - just clear, direct checks that catch what matters. The final result? A working system built to handle real traffic without flaws.

M. Implementation Outcomes

System runs faster now. The LRU cache slashed API calls by 48%. That alone cuts down on wasted traffic and speeds up data pulls. Less hitting remote servers means less lag. Frontend got smarter too - lazy loads, better image sizing, quicker DOM changes Load times dropped 35 - 40%. On slow connections, users feel it instantly. Lighthouse gives clean scores over 90% every time. Speed, accessibility, code rules - all checked and passed. No API keys float around anymore. Keys stay hidden behind proxies and env vars. Security feels solid now. Backend shields them from leaks. Nothing slips through the cracks. The design holds up under stress tests. It scales without breaking. Traffic spikes don't crash it. Users get what they need fast and safely. Architecture works because it was built right from the start.

VIII. CONCLUSION

The Unsplash FindAWall project is a great example of how a secure, scalable, and high-performance image retrieval system can be designed and implemented. The system uses a backend proxy architecture to secure API communications and remove the risk of exposing credentials. Performance optimization techniques including lazy loading, LRU caching, and responsive rendering led to very substantial reductions in load times and enhanced API efficiency. Moreover, the introduction of rate-limit management added to system dependability and guaranteed adherence to API limits. Besides, the modular design offers a very solid platform for the integration of AI-based retrieval methods in the future, making the system both robust and research-oriented.

The app still leaves room to grow. With CLIP embeddings and vector databases, so users can find similar content faster, something that's pretty much missing now. Redis helps store frequently used data across devices so things don't lag during peak hours. Adding user accounts that stay saved over time makes getting through smoother for regular visitors. Turning this into a progressive Web App lets people access it on phones without downloads. Testing under heavy load shows where speed breaks down, which points to weak spots in current design. Including WCAG 2, plus 1 means more users with disabilities can actually use it too.

IX. ACKNOWLEDGMENT

The author is deeply grateful to the faculty and management of the Department of Computing Technologies, SRM Institute of Science and Technology, for the opportunity and resources rendered to carry out this project. In a special way, the author extends gratitude to the project mentor Dr. Gowtham P for insightful direction, consistent encouragement, and constructive critiquing during the progress of this research.

Aside from this, the author is grateful to the creators and contributors of the Unsplash API for giving access to quality image datasets that were essential to this project. Besides, the author highly appreciates the efforts of the open-source community and contributors of the tools React, Node.js, and related frameworks, which made the realization of this system possible. At last, the author thanks family, friends, and fellow students from the bottom of the heart for their believing, inspiring, and standing behind the author all along project work.

X. REFERENCES

[1] S. M. Metev and V. P. Veiko, Laser Assisted Microtechnology, 2nd ed., R. M. Osgood, Jr., Ed. Berlin, Germany: Springer-Verlag, 1998.
[2] J. Breckling, Ed., The Analysis of Directional Time Series: Applications to Wind Speed and Direction, ser. Lecture Notes in Statistics. Berlin, Germany: Springer, 1989, vol. 61.
[3] S. Zhang, C. Zhu, J. K. O. Sin, and P. K. T. Mok, "A novel ultrathin elevated channel low-temperature poly-Si TFT," IEEE Electron Device Lett., vol. 20, pp. 569-571, Nov. 1999.
[4] M. Wegmuller, J. P. von der Weid, P. Oberson, and N. Gisin, "High resolution fiber distributed measurements with coherent OFDR," in Proc. ECOC'00, 2000, paper 11.3.4, p. 109.
[5] R. E. Sorace, V. S. Reinhardt, and S. A. Vaughn, "High-speed digital-to-RF converter," U.S. Patent 5 668 842, Sept. 16, 1997.
[6] (2002) The IEEE website. [Online]. Available: http://www.ieee.org/
[7] M. Shell. (2002) IEEEtran homepage on CTAN. [Online]. Available: http://www.ctan.org/tex-archive/macros/latex/contrib/supported/IEEEtran/
[8] FLEXChip Signal Processor (MC68175/D), Motorola, 1996.
[9] "PDCA12-70 data sheet," Opto Speed SA, Mezzovico, Switzerland.
[10] A. Karnik, "Performance of TCP congestion control with rate feedback: TCP/ABR and rate adaptive TCP/IP," M. Eng. thesis, Indian Institute of Science, Bangalore, India, Jan. 1999.
[11] J. Padhye, V. Firoiu, and D. Towsley, "A stochastic model of TCP Reno congestion avoidance and control," Univ. of Massachusetts, Amherst, MA, CMPSCI Tech. Rep. 99-02, 1999.
[12] Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specification, IEEE Std. 802.11, 1997.