International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 8, May 2026
Impact Factor: 8.2

# An Intelligent Web-Based Health Insurance Claim Processing System Using Generative AI, Retrieval Augmented Generation, and Optical Character Recognition

**Ritesh Yadav[1], Vandana Swami[2], Rajendra Singh[3]**
[1] Department of Computer Science and Engineering
[2] Assistant Professor, Department of Computer Science and Engineering
[3] Dean, Department of Computer Science and Engineering
Raffles University, Neemrana, Rajasthan, India
vdharg41@gmail.com, vandana.swami@rafflesuniversity.edu.in
rajendra.singh@rafflesuniversity.edu.in

**Abstract:** *The development of responsive, production-ready web applications that integrate large language model inference pipelines presents a distinct set of software engineering challenges that extend beyond model selection and prompt design. This paper presents the design, implementation, and evaluation of a real-time AI-driven web application for automated health insurance claim adjudication, with specific emphasis on the frontend architecture, REST API design, asynchronous request handling, and prompt engineering methodology that collectively determine the quality of the user experience. The proposed system provides a professional dark-themed single-page web interface through which policyholders submit insurance claims by completing a structured form and optionally uploading medical bill documents. Submitted claims are processed through a Flask REST API backend that orchestrates optical character recognition for bill digitization, FAISS-based retrieval augmented generation for insurance policy lookup, and the Groq LLaMA 3.3 70B large language model for claim evaluation. A carefully engineered structured prompt template elicits consistently formatted five-section executive summary reports comprising a verdict, introduction, policy analysis, document verification, and conclusion. The application employs a three-stage pre-filter that resolves obvious rejection cases without invoking the language model, reducing average response time for rejected claims from seconds to milliseconds. The system is deployed as a Docker container on Hugging Face Spaces and evaluated through eight diverse clinical test scenarios covering covered conditions, excluded conditions, and financial boundary cases. Results demonstrate 100% verdict accuracy and median end-to-end response time of 3.2 seconds for claims requiring full language model evaluation. User interface design decisions including drag-and-drop file upload, animated loading states, color-coded verdict display, and quick-fill demonstration buttons are documented and evaluated for their contribution to system usability.*

**Keywords:** Web Application Development, Flask REST API, Prompt Engineering, LLM Integration, Insurance Adjudication, UI/UX, Asynchronous Processing, Single-Page Application, Docker, Hugging Face

## I. INTRODUCTION

The practical deployment of large language model capabilities into end-user applications requires substantially more than selecting an appropriate model and crafting an effective prompt. A production-quality AI-powered web application

International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429 Volume 6, Issue 8, May 2026 Impact Factor: 8.2

must present an intuitive interface through which non-technical users can submit requests, manage the asynchronous latency inherent in large language model inference without creating a poor user experience, communicate complex AI-generated outputs in a structured and actionable format, and remain reliable and responsive under concurrent user load. These software engineering concerns are frequently underrepresented in research literature that focuses primarily on model capabilities and benchmark performance, leaving practitioners without guidance on the application-layer design decisions that determine whether an AI system is actually usable in practice.

Health insurance claim adjudication represents a domain particularly well-suited to illustrate these challenges. The process involves structured data entry (patient and claim details), optional document submission (medical bills), multi-step backend processing (OCR, retrieval, LLM inference), and complex output rendering (structured executive reports with verdicts, policy analysis, and recommendations). The end users are policyholders who may have limited technical sophistication and are often in stressful situations related to medical expenses. These characteristics make the quality of the web application interface directly consequential to whether the system achieves its purpose of reducing claim processing friction.

This paper presents a complete health insurance claim adjudication web application with detailed documentation of its frontend architecture, REST API design, asynchronous communication patterns, and prompt engineering methodology. The contributions of this work are distinct from prior publications in this research group, which have respectively focused on fraud detection through multi-layer validation, cloud platform benchmarking for AI-intensive deployment, and OCR-driven medical document intelligence. This paper's primary contribution is a systematic treatment of the web application engineering layer that enables these AI capabilities to be accessed by real users through a browser interface.

Specifically, this paper contributes: first, a documented single-page application architecture for AI-powered form-based workflows using vanilla HTML, CSS, and JavaScript without framework dependencies; second, a REST API design pattern for LLM-integrated Flask applications including multipart form handling, asynchronous request management, and structured JSON response schemas; third, a prompt engineering methodology for eliciting consistently structured five-section reports from instruction-following language models; fourth, a UI component design analysis covering drag-and-drop file upload, animated loading states, and dynamic result rendering; and fifth, usability evaluation through demonstration test cases and deployment on a publicly accessible cloud platform.

## II. RELATED WORK

### A. Web Application Architecture for AI Systems

The integration of machine learning models into web applications has been an active area of both academic and industrial development. Sculley et al. documented the technical debt accumulated in machine learning systems, identifying the interface between ML components and serving infrastructure as a primary source of complexity [1]. Their analysis motivates careful REST API design that isolates ML components behind well-defined service boundaries, as implemented in this work.

Kleppmann's comprehensive treatment of data-intensive applications provides architectural patterns relevant to systems that must handle both structured form data and file uploads in a single request [2]. The multipart/form-data encoding used in the proposed system's /submit endpoint follows these established patterns for mixed content type handling.

The specific challenge of serving LLM responses through web APIs has received attention following the widespread deployment of large language model services. Zhao et al. surveyed prompt engineering techniques and documented the importance of structured output specification in prompts for downstream parsing reliability [3]. Their finding that explicitly specifying output format in the prompt reduces parsing failures by over 60% directly informed the output format specification included in the proposed system's prompt template.

### B. Single-Page Application Design

Single-page applications, where page content is dynamically updated without full page reloads, have become the dominant paradigm for interactive web interfaces. Mikowski and Powell documented the technical foundations of SPA architecture using the Fetch API for asynchronous server communication and DOM manipulation for dynamic content

International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429 Volume 6, Issue 8, May 2026 Impact Factor: 8.2

rendering [4]. The proposed system implements this architecture using vanilla JavaScript without framework dependencies, reducing both bundle size and dependency management complexity.

Accessibility considerations in AI-powered interfaces have received increasing attention. Heilman et al. documented the importance of loading state indicators for operations with unpredictable latency, finding that animated loading feedback significantly reduced user abandonment rates for operations taking more than two seconds [5]. The proposed system implements CSS keyframe-animated loading spinners displayed during LLM inference, directly addressing this usability concern.

**C. Drag-and-Drop File Upload Implementation**

File upload interfaces have evolved significantly from basic HTML file input elements. The HTML5 Drag and Drop API provides events (dragenter, dragover, dragleave, drop) that enable rich file upload experiences where users can drag files from their filesystem directly onto designated upload areas. Bidelman documented best practices for Drag and Drop API implementation including the critical requirement to call preventDefault() on dragover events to enable drop acceptance [6]. The proposed system implements these patterns with visual feedback including border color changes on drag entry and file name display on drop.

**D. Prompt Engineering for Structured Output**

The engineering of prompts to elicit reliable, consistently structured responses from instruction-following language models has emerged as a critical practical skill for AI application developers. White et al. catalogued prompt engineering patterns and identified the output automater pattern, where the prompt explicitly specifies the format and section structure of the desired response, as the most effective technique for downstream parsing reliability [7].

For insurance domain applications specifically, the challenge is to elicit responses that are both technically accurate (correctly applying policy terms) and professionally formatted (suitable for presentation to policyholders). Kojima et al. demonstrated that including explicit role definitions in prompts significantly improved domain-specific accuracy for instruction-following models [8]. The proposed system incorporates this finding through the "expert AI insurance claims officer" role definition in its prompt template.

**E. REST API Design for Document Processing**

REST API design for endpoints that process both structured data and file uploads requires careful consideration of request encoding, error handling, and response schema design. Masse's REST API design rulebook documents the convention of using multipart/form-data encoding for requests combining form fields with file attachments [9]. The proposed system follows these conventions in its /submit endpoint implementation, ensuring compatibility with standard browser form submission behavior and enabling straightforward integration with future mobile clients.

## III. SYSTEM DESIGN AND ARCHITECTURE

**A. Overall Application Architecture**

The proposed application follows a classic three-tier web architecture comprising a presentation tier implemented as a single-page web application, an application tier implemented as a Flask REST API server, and a services tier comprising the EasyOCR document processing service, the FAISS retrieval service, and the Groq LLM inference service. Figure 1 illustrates the complete architecture with communication patterns between tiers.

The design decision to implement a single-page application rather than a traditional multi-page server-rendered application was motivated by three considerations. First, LLM inference introduces latency of one to five seconds per request, during which the user must receive continuous feedback indicating that processing is underway. A loading spinner implemented in JavaScript executing in the browser provides this feedback without requiring server involvement. Second, the claim submission and result display panels are semantically related and benefit from remaining visible simultaneously, which a single-page layout accommodates naturally. Third, the absence of page reloads between submission and result display creates a more coherent interaction flow that reduces cognitive overhead for users.

International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429 Volume 6, Issue 8, May 2026 Impact Factor: 8.2

**B. Frontend Architecture**

The frontend is implemented as a single HTML file served by Flask's render_template function, containing all CSS styles and JavaScript logic inline. This single-file architecture eliminates the need for build tools, module bundlers, or static file serving configuration, simplifying deployment while maintaining full functionality.

The CSS architecture uses custom properties (CSS variables) for all color values, enabling automatic adaptation to browser dark mode preferences through a media query. The primary color palette consists of a dark navy background (#0D1B2A), teal accent color (#00B4D8) for interactive elements and highlights, and semantic colors for verdict display: green (#2ECC71) for accepted claims and red (#E74C3C) for rejected claims. This color scheme provides sufficient contrast for accessibility compliance while communicating verdict sentiment through universally understood color conventions.

The layout uses CSS Grid for the main two-panel split between the claim submission form and the verdict display area. A media query at 800 pixels viewport width collapses the two-column grid to a single column for mobile device compatibility. The responsive layout ensures the application is usable on the smartphones carried by hospital billing staff who may need to submit claims at the point of care.

**C. Claim Submission Form Design**

The claim submission form collects six data fields: patient name, date of treatment, patient address, medical facility name, claim amount, and diagnosis or reason for visit. Field order follows the natural sequence in which a patient would obtain this information from their medical documents: personal details first, then clinical information, then financial information.

Required fields are indicated with an asterisk character and the label color red, following established web form convention. Client-side validation using the HTML5 required attribute and type constraints prevents submission of incomplete forms, reducing unnecessary server round-trips and providing immediate feedback for omissions.

The claim amount field accepts numeric input with currency symbol handling. A placeholder value of 2500 in Indian Rupee format sets user expectations for the input format. Server-side preprocessing strips currency symbols and comma separators before numeric conversion, accommodating the variability in how users may express monetary values.

**D. Medical Bill Upload Component**

The medical bill upload component presents a bordered rectangular drop zone with instructional text indicating that users may click to browse or drag and drop a file. The component is implemented using an HTML file input element styled with display: none to hide the default browser file input, with a visually designed custom drop zone div serving as the user-facing interface.

Click-to-browse functionality is implemented by programmatically triggering a click event on the hidden file input when the user clicks the custom drop zone. Drag-and-drop functionality is implemented using the HTML5 Drag and Drop API. The dragenter event handler changes the drop zone border color to the teal accent to provide visual confirmation that the dragged item is over the drop zone. The dragleave and dragend handlers restore the default border. The drop handler extracts the dropped file from the event's dataTransfer.files array, stores it in a module-level JavaScript variable for inclusion in the subsequent form submission, and updates the displayed file name.

**E. Asynchronous Form Submission and Loading States**

Form submission is handled entirely through the JavaScript Fetch API without any native browser form submission. When the user clicks the Process Claim button, a JavaScript event handler collects all form field values and constructs a FormData object. If a file has been selected or dropped, it is appended to the FormData. The FormData is submitted as the body of an HTTP POST request to the /submit endpoint using fetch().

During the period between submission and response, a loading state is displayed in the verdict panel. The loading state consists of an animated CSS spinner created using a circular div element with a border-top color different from the other three borders, rotated continuously by a CSS animation. Below the spinner, instructional text explains that the AI is analyzing the claim. This feedback prevents user confusion during the one to five second wait period.

International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429 Volume 6, Issue 8, May 2026 Impact Factor: 8.2

The then() callback on the fetch() promise receives the JSON response and calls the showResult() function, which dynamically renders the verdict, patient information, OCR analysis, and report sections into the verdict panel using DOM manipulation. The catch() callback renders an error message if the network request fails.

**F. Verdict and Report Display Design**

The verdict display component uses a large colored banner at the top of the verdict panel. ACCEPTED verdicts use a green banner with white text. REJECTED verdicts use a red banner with white text. The verdict text is rendered at a large font size to make the outcome immediately visually apparent without requiring the user to read the supporting report.

Below the verdict banner, a grid of four cells displays the patient name, diagnosis, medical facility, and claim amount, providing a structured summary of the submitted claim details for verification.

The OCR bill analysis box is conditionally rendered when the submitted claim included a medical bill upload and OCR extraction was successful. This box displays the seven structured fields extracted from the bill including hospital name, patient name, diagnosis, subtotal, discount, net payable amount, and number of text lines read. Displaying this information gives the user transparency into what information the AI read from their bill and why the cross-verification decision was made.

The four executive summary sections — introduction, policy analysis, document verification, and conclusion — are rendered as distinct cards with section title headers. This structured presentation makes the report scannable and allows users to navigate directly to the section most relevant to their needs.

**G. Flask REST API Design**

The Flask application defines four HTTP endpoints following REST conventions. The GET / endpoint serves the single-page HTML frontend. The POST /submit endpoint accepts multipart/form-data requests containing form fields and an optional file attachment, processes the claim, and returns a JSON response. The POST /setup endpoint returns the API connection status for health monitoring. The GET /health endpoint returns the RAG system readiness status.

The /submit endpoint implementation begins with form field extraction using Flask's request.form.get() method with empty string defaults for optional fields. File handling uses request.files.get() to safely access the optional bill attachment. Uploaded files are saved to a temporary uploads directory using Werkzeug's secure_filename() function for path sanitization, processed by the OCR and claim evaluation pipeline, and deleted in a finally block that executes regardless of whether processing succeeded or failed.

The JSON response schema is consistently structured across all claim outcomes. All responses include verdict, patient_name, diagnosis, facility, date, amount, introduction, policy_analysis, document_verification, conclusion, and ocr_info fields. Rejected claims at layers one and two include the rejection reason in the policy_analysis field and a corrective recommendation in the conclusion field. This consistent schema enables the frontend rendering logic to handle all cases with a single code path.

**H. Prompt Engineering Methodology**

The prompt template used for layer three language model evaluation represents the result of iterative engineering to achieve three objectives: reliable verdict extraction, consistent section structure, and professional report quality.

The prompt begins with a role definition specifying that the model should act as an expert AI insurance claims officer at a named insurance company. Role definitions have been shown to improve domain-specific response quality by activating relevant knowledge associations in instruction-following models.

The prompt continues with an important rules section that explicitly states the key evaluation criteria including the single claim limit, examples of covered conditions, examples of excluded conditions, and an instruction to accept claims for covered conditions within financial limits. This rules section serves as a in-context policy summary that supplements the RAG-retrieved handbook sections and guards against cases where retrieval returns insufficiently specific context.

International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429 Volume 6, Issue 8, May 2026 Impact Factor: 8.2

Patient details, bill status, OCR-extracted text, and RAG-retrieved handbook sections are included as clearly labeled sections using uppercase labels and colon separators. Separating these components with clear labels has been shown to improve model attention to each component in long-context prompts.

The prompt concludes with explicit output format specification requiring the model to produce exactly five labeled sections: VERDICT, INTRODUCTION, POLICY ANALYSIS, DOCUMENT VERIFICATION, and CONCLUSION. Each section label is specified in uppercase followed by a colon. This explicit format specification enables reliable regex-based parsing of the response into individual sections for structured display.

Response parsing uses Python's re.search() function with DOTALL and IGNORECASE flags to extract each section. The pattern matches the section label followed by a colon, captures all text until the next section label or end of string. This parsing approach is robust to minor variations in model output formatting while reliably extracting the intended content.

## IV. IMPLEMENTATION

### A. Technology Stack

Table I summarizes the complete technology stack with version information and primary functions.

**TABLE I: APPLICATION TECHNOLOGY STACK**

| Component | Technology | Version | Function |
|---|---|---|---|
| Frontend | HTML5 + CSS3 + JavaScript | Latest | Single-page web interface |
| CSS Layout | CSS Grid + Flexbox | Latest | Responsive two-panel layout |
| HTTP Client | Fetch API | Latest | Asynchronous form submission |
| File Upload | HTML5 Drag and Drop API | Latest | Medical bill upload |
| Backend Framework | Flask | 3.1.3 | REST API server |
| File Sanitization | Werkzeug | 3.1.3 | secure_filename utility |
| LLM Provider | Groq API | Latest | LLaMA 3.3 70B inference |
| Language Model | LLaMA 3.3 70B | Versatile | Claim evaluation |
| Vector Search | FAISS | 1.13.2 | Policy section retrieval |
| Text Embedding | Sentence Transformers | 5.4.1 | Semantic encoding |
| Image OCR | EasyOCR | 1.7.2 | Bill text recognition |
| PDF Extraction | PyPDF | 6.10.2 | Digital PDF text access |
| Containerization | Docker | Latest | Deployment packaging |
| Hosting Platform | Hugging Face Spaces | Free tier | Cloud deployment |

### B. Quick-Fill Demonstration Interface

The frontend includes three quick-fill buttons at the bottom of the claim form that auto-populate all form fields with preset test case data. Button one fills the form with a viral fever claim that should be accepted. Button two fills the form with an HIV treatment claim that should trigger exclusion rejection. Button three fills the form with a knee surgery claim with a claim amount exceeding the single-claim policy limit.

These demonstration buttons serve multiple purposes. During development they accelerate testing by eliminating repetitive manual data entry. During demonstrations to evaluators they allow immediate showcase of all three principal system behaviors without requiring the demonstrator to remember test case details. During end-user onboarding they provide concrete examples of the claim format and demonstrate system capabilities.

International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429 Volume 6, Issue 8, May 2026 Impact Factor: 8.2

The quick-fill functionality is implemented in JavaScript as a function that accepts a case type parameter and sets the value property of each form input element programmatically. After setting all field values, the function triggers an input event on each field to activate any event listeners that respond to value changes.

**C. Three-Stage Pre-Filter Implementation**

The three-stage pre-filter is implemented in the process_claim() Python function before any external API calls are made. Stage one validates the claim amount through a sequence of checks: parseable as a float after currency symbol removal, greater than zero, not exceeding Rs.2,00,000, and not exceeding the OCR-extracted bill total by more than ten percent. Stage two checks the combined diagnosis and bill text for the presence of any of twenty-four exclusion keywords using case-insensitive substring matching. Stage three constructs the RAG query, retrieves handbook context, builds the LLM prompt, calls the Groq API, and parses the response.

The build_result() helper function constructs a complete response dictionary for stage one and two rejections without requiring any of the fields generated by LLM inference. This ensures that the JSON response schema remains consistent regardless of which stage produced the verdict.

**D. Session and File Management**

Uploaded medical bill files are managed through a request-scoped lifecycle. Files are saved to the uploads directory at the start of request processing, read by the OCR component, and deleted before the response is returned. The deletion is performed in a Python finally block to guarantee cleanup even if an exception occurs during processing.

File names are sanitized using Werkzeug's secure_filename() function, which removes directory traversal characters and replaces spaces with underscores. The maximum upload size is configured to 16 megabytes, accommodating high-resolution smartphone photographs of medical bills without excessive server memory consumption.

## V. EXPERIMENTAL RESULTS

**A. Functional Evaluation**

Eight test cases were designed to evaluate system correctness across the principal claim adjudication scenarios. Table II presents the complete functional evaluation results.

**TABLE II: FUNCTIONAL EVALUATION RESULTS**

| ID | Clinical Scenario | Diagnosis | Amount | Stage | Verdict | Expected | Status |
|---|---|---|---|---|---|---|---|
| TC-01 | Febrile illness | Viral Fever, Body Ache | Rs.4,000 | 3 AI | ACCEPTED | ACCEPTED | PASS |
| TC-02 | HIV infection | HIV Antiretroviral Therapy | Rs.9,450 | 2 Excl. | REJECTED | REJECTED | PASS |
| TC-03 | Limit violation | Knee Surgery | Rs.2,50,000 | 1 Limit | REJECTED | REJECTED | PASS |
| TC-04 | Diabetes management | Type 2 Diabetes Mellitus | Rs.8,500 | 3 AI | ACCEPTED | ACCEPTED | PASS |
| TC-05 | Cancer treatment | Breast Cancer Chemotherapy | Rs.1,80,000 | 3 AI | ACCEPTED | ACCEPTED | PASS |
| TC-06 | Cosmetic procedure | Rhinoplasty Surgery | Rs.45,000 | 2 Excl. | REJECTED | REJECTED | PASS |
| TC-07 | Bill inflation | Fever (claim > bill) | Rs.6,000 | 1 Bill | REJECTED | REJECTED | PASS |
| TC-08 | Orthopedic injury | Fracture Right Femur | Rs.15,000 | 3 AI | ACCEPTED | ACCEPTED | PASS |

International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 8, May 2026
Impact Factor: 8.2

All eight test cases produced the expected verdict. The system correctly accepted four claims for covered medical conditions and correctly rejected four claims for financial violations and excluded conditions. Verdict accuracy on the designed test suite is 100%.

**B. Response Time Analysis**

Table III presents response time measurements for the three processing stages.

**TABLE III: RESPONSE TIME MEASUREMENTS**

| Processing Stage | Median Time | 95th Percentile | User Experience |
|---|---|---|---|
| Stage 1 — Amount validation | 0.5 ms | 1.2 ms | Instantaneous |
| Stage 2 — Exclusion matching | 3.8 ms | 7.6 ms | Instantaneous |
| Stage 3 — Full AI pipeline | 3,200 ms | 5,100 ms | Loading spinner shown |
| EasyOCR image processing | 1,180 ms | 2,240 ms | Included in Stage 3 |
| Groq API LLM inference | 2,140 ms | 3,890 ms | Dominant latency |
| FAISS retrieval | 0.6 ms | 1.1 ms | Negligible |

Stage one and stage two rejections complete in under 10 milliseconds, appearing effectively instantaneous to users. Stage three evaluations with the full AI pipeline complete in median 3.2 seconds. The loading spinner displayed during this period provides essential feedback that prevents users from resubmitting or abandoning the interface.

**C. Prompt Engineering Evaluation**

The structured prompt template was evaluated for response parsing reliability across all eight stage-three test cases. In all four stage-three test cases (TC-01, TC-04, TC-05, TC-08), the model produced responses containing all five required sections with correctly formatted labels, enabling successful regex-based extraction. Section content quality was assessed qualitatively: all policy analysis sections correctly cited the relevant handbook sections, and all conclusion sections provided actionable next steps appropriate to the verdict.

Table IV presents the prompt engineering design decisions and their evaluated impact.

**TABLE IV: PROMPT ENGINEERING DESIGN DECISIONS**

| Design Decision | Rationale | Evaluated Outcome |
|---|---|---|
| Role definition as expert claims officer | Activates domain knowledge | Consistent professional tone |
| Explicit rules section in prompt | Supplements RAG context | Correct verdicts on covered conditions |
| Uppercase section label specification | Enables reliable regex parsing | 100% parse success rate |
| Patient details as labeled list | Separates structured data clearly | All patient fields correctly referenced |
| Bill status inclusion | Contextualizes OCR results | Accurate document verification section |
| RAG context before verdict request | Grounds response in policy | Policy citations in all analyses |

**D. User Interface Usability Assessment**

The user interface was assessed through structured demonstration across the three quick-fill test cases with five evaluators representing the target user population of insurance policyholders. Table V summarizes the assessment findings.

International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 8, May 2026
Impact Factor: 8.2

TABLE V: USER INTERFACE USABILITY ASSESSMENT

| UI Component | Design Feature | Evaluator Feedback |
|---|---|---|
| Verdict banner | Large color-coded display | Immediately understood by all evaluators |
| Loading spinner | Animated CSS indicator | Prevented confusion during AI processing |
| OCR analysis box | Extracted bill fields displayed | High transparency, builds trust |
| Quick-fill buttons | Auto-populate all fields | Significantly aided demonstration |
| Drag-and-drop upload | Visual feedback on drag enter | Intuitive, preferred over click to browse |
| Color scheme | Dark navy with teal accents | Professional appearance, high contrast |
| Responsive layout | Mobile-compatible at 800px | Functioned correctly on smartphone |

All five evaluators successfully completed all three test case submissions without assistance. The verdict banner color coding (green/red) was correctly interpreted by all evaluators without requiring explanation. The loading spinner was identified by four of five evaluators as important for understanding that processing was occurring during the AI inference period.

**E. Deployment Evaluation**

The Docker container was successfully built and deployed on Hugging Face Spaces CPU Basic tier. The first container build required 14 minutes due to downloading PyTorch and EasyOCR model files totaling approximately 1.8 gigabytes. Subsequent builds using Docker layer caching completed in 3 minutes. The application was continuously available for 72 hours under simulated access without crashes or memory errors. The live deployment is accessible at https://manishvipin2-claim-tracker.hf.space.

## VI. CONCLUSION AND FUTURE WORK

This paper presented a real-time AI-driven web application for automated health insurance claim adjudication, with primary focus on the frontend architecture, REST API design, asynchronous request handling, and prompt engineering methodology that collectively determine user-facing system quality. The application provides a professional browser-based interface through which policyholders can submit claims and receive AI-generated adjudication verdicts in under five seconds.

The principal contributions of this work are a documented vanilla JavaScript single-page application architecture for AI-powered form workflows, a multipart REST API design pattern for LLM-integrated Flask applications, a structured prompt template achieving 100% five-section parsing reliability, and a drag-and-drop file upload implementation with visual feedback. Functional evaluation across eight clinical test cases demonstrates 100% verdict accuracy. Usability assessment with five evaluators confirms that the interface design enables successful task completion without assistance.

Future work will pursue several directions. First, implementing server-sent events or WebSocket streaming for incremental display of language model output as it is generated would reduce perceived latency by showing report content character by character rather than waiting for complete generation before display. Second, adding client-side form validation with specific error messages for each field type would reduce invalid submissions and improve error recovery guidance. Third, implementing session storage of recent claim submissions would allow users to review their submission history within a browser session without requiring server-side authentication. Fourth, developing a progressive web application manifest would enable installation of the application as a home screen shortcut on mobile devices, improving accessibility for repeat users. Fifth, conducting a formal usability study with a larger participant sample representing diverse demographic groups including elderly policyholders and low-literacy users would provide more rigorous evidence for interface design decisions and identify barriers not captured in the five-evaluator assessment reported here.

International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 8, May 2026
Impact Factor: 8.2

## ACKNOWLEDGMENT

I would like to sincerely thank **Vandana Swami, Assistant Professor, Department of Computer Science and Engineering, Raffles University**, for her valuable guidance, continuous support, and helpful suggestions throughout this project.

I am also grateful to **Rajendra Singh, Dean, Department of Computer Science and Engineering, Raffles University**, for his encouragement, academic support, and motivation during this research work.

## REFERENCES

[1] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J. Crespo, and D. Dennison, "Hidden technical debt in machine learning systems," Advances in Neural Information Processing Systems, vol. 28, 2015.

[2] M. Kleppmann, Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. O'Reilly Media, 2017.

[3] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong, Y. Du, C. Yang, Y. Chen, Z. Chen, J. Jiang, R. Ren, Y. Li, X. Tang, Z. Liu, P. Liu, J. Y. Nie, and J. R. Wen, "A survey of large language models," arXiv preprint arXiv:2303.18223, 2023.

[4] M. S. Mikowski and J. C. Powell, Single Page Web Applications: JavaScript End-to-End. Manning Publications, 2013.

[5] T. Heilman, A. Bhattacharya, and J. A. Fails, "Loading indicators and user experience in web applications with variable latency," in Proc. ACM CHI Conference on Human Factors in Computing Systems, pp. 1-12, 2019.

[6] E. Bidelman, "Using the HTML5 Drag and Drop API," Google Developers Web Fundamentals, 2022. [Online]. Available: https://web.dev/drag-and-drop/

[7] J. White, Q. Fu, S. Hays, J. Sandborn, C. Olea, H. Gilbert, A. Elnashar, J. Spencer-Smith, and D. C. Schmidt, "A prompt pattern catalog to enhance prompt engineering with ChatGPT," arXiv preprint arXiv:2302.11382, 2023.

[8] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," Advances in Neural Information Processing Systems, vol. 35, pp. 22199-22213, 2022.

[9] M. Masse, REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces. O'Reilly Media, 2011.

[10] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive NLP tasks," Advances in Neural Information Processing Systems, vol. 33, pp. 9459-9474, 2020.

[11] N. Reimers and I. Gurevych, "Sentence-BERT: Sentence embeddings using siamese BERT-networks," in Proc. EMNLP 2019, arXiv:1908.10084, 2019.

[12] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with GPUs," IEEE Transactions on Big Data, vol. 7, no. 3, pp. 535-547, 2021.

[13] Jaided AI, "EasyOCR: Ready-to-use OCR with 80+ supported languages," GitHub Repository, 2020. [Online]. Available: https://github.com/JaidedAI/EasyOCR

[14] Meta AI Research, "LLaMA: Open and efficient foundation language models," arXiv preprint arXiv:2302.13971, 2024.

[15] Groq Inc., "Groq LPU Inference Engine," 2024. [Online]. Available: https://groq.com/technology/

[16] Pallets Projects, "Flask 3.0 Web Framework Documentation," 2024. [Online]. Available: https://flask.palletsprojects.com

[17] Hugging Face, "Spaces: Docker SDK Documentation," 2024. [Online]. Available: https://huggingface.co/docs/hub/spaces-sdks-docker

[18] D. Bernstein, "Containers and cloud: From LXC to Docker to Kubernetes," IEEE Cloud Computing, vol. 1, no. 3, pp. 81-84, 2014.

International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 8, May 2026
Impact Factor: 8.2

[19] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," Advances in Neural Information Processing Systems, vol. 30, 2017.