Published by :
https://www.ijert.org/
An International Peer-Reviewed Journal

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

Sketch to Code:
A Multi-Modal AI Framework for Automated Conversion of UI Sketches into Functional Web Code

Mr. Mohammed Amaan Shaikh, Mr. Aadiish Shukla, Mr. Aditya Mishra, Mr. Aaditya Devghare, Prof. Smita Dandge
Department of Computer Engineering Thakur Shyamnarayan Engineering College Mumbai, India

Abstract - The translation of hand-drawn user interface (UI) sketches into executable front-end code represents a persistent challenge in modern software engineering, requiring specialized expertise and considerable manual effort. This paper presents SketchToCode, a multi-modal artificial intelligence framework that automates the conversion of UI sketch images into separated, production-ready HTML, CSS, and JavaScript code. The proposed system integrates two complementary inference engines: a cloud-based pipeline leveraging the Google Gemini 2.5 Flash vision-language model for high-fidelity, instruction-guided generation via structured prompting, and a locally deployable sequence-to- sequence model combining a ResNet-18 convolutional encoder with a two-layer LSTM decoder for offline, privacy-preserving inference. The framework supports multi-image stitching, enabling designers to compose composite layouts from distinct header, body, and footer sketches. A client-side image compression pipeline reduces typical payloads from several megabytes to under 200 kilobytes, minimizing API latency. A React-based web interface, paired with cross-platform desktop (Tauri) and mobile (Capacitor/Android) deployment targets, delivers a complete application from a single codebase. The generated code is rendered in a sandboxed iframe for real-time preview and displayed in tabbed code editors for inspection. Persistent user history is managed through a Supabase PostgreSQL backend. Experimental evaluation demonstrates that the structured-prompting strategy reliably produces well-formed, separated code artifacts from diverse sketch inputs within acceptable response times. SketchToCode demonstrates that modern multi-modal AI can substantially reduce the design-to-code gap, offering a practical tool for rapid prototyping and accessible web development.

Keywords **- sketch-to-code; multi-modal AI; Gemini API; CNN-LSTM; UI code generation; code automation; front-end prototyping; image-to-code; ResNet; sequence-to-sequence; cross-platform development; rapid prototyping**.

I. INTRODUCTION

The gap between a designer's conceptual sketch and a deployable UI implementation has long been a bottleneck in software development pipelines. Prototyping tools such as Figma and Adobe XD have partially bridged this divide, yet they still demand that a trained engineer manually translate visual designs into structured code. For small teams, rapid-prototyping environments, and non-technical stakeholders, this translation overhead can slow iteration cycles significantly [1].

Recent advances in multi-modal large language models (LLMs) and deep learning-based image understanding have opened new avenues for automating this process. Vision-language models capable of jointly reasoning over image pixels and natural language instructions can, in principle, interpret the structural intent of a sketch and emit syntactically correct Hypertext Markup Language (HTML), Cascading Style Sheets (CSS), and JavaScript (JS) in a single inference pass [2]. Large language models such as Google's Gemini family [3] and OpenAI's GPT-4V [4] can accept image inputs alongside natural language instructions and produce structured outputs including source code. Concurrently, sequence-to-sequence architectures combining convolutional neural networks (CNNs) with recurrent decoders have been applied to the specific task of generating markup from UI screenshots, achieving promising results on constrained domains [5].

Despite these advances, existing approaches suffer from several limitations. Cloud-based LLM solutions, while powerful, depend on internet connectivity and raise data privacy concerns. Local deep learning models trained on limited datasets may lack generalization capacity. Rule- based approaches require structured or domain-specific input formats. Earlier neural models (e.g., pix2code [5]) are constrained to narrow DSL-driven UI grammars and do not scale gracefully to free-form sketches. Contemporary LLM- based tools typically generate monolithic HTML blobs rather than maintainable, separated code artifacts. Few systems offer a unified, cross-platform deployment strategy that brings sketch-to-code capabilities to web, desktop, and mobile environments simultaneously [6].

This paper presents SketchToCode, an end-to-end multi-modal AI framework designed to address these gaps. The system accepts one or more UI sketch images— optionally enriched by natural language refinement instructions—and produces structured, separated front-end code. A dual inference architecture accommodates both high-throughput cloud-based

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

generation via the Gemini 2.5 Flash API and an offline-capable local inference path built on a purpose-trained CNN-LSTM model. The application is deployed across web, desktop, and mobile platforms using React, Tauri, and Capacitor, respectively, with user session history persisted through a Supabase backend.

The remainder of this paper is organized as follows. Section II formalizes the problem. Section III states the system objectives. Section IV surveys related work. Section V describes the proposed system. Sections VI and VII detail the architecture and implementation. Section VIII enumerates the technology stack. Section IX discusses testing and validation. Section X presents results and discussion. Sections XI and XII address limitations and future scope. Section XIII concludes the paper.

II. PROBLEM STATEMENT

The conversion of a UI sketch to functional code is a non-trivial multi-step process. In current practice, a front-end engineer must (i) interpret the spatial layout of a hand-drawn or digitally sketched prototype, (ii) select appropriate HTML structural elements, (iii) author CSS rules that approximate the visual intent, and (iv) attach any required JavaScript behavior. Each of these steps is error-prone, time- consuming, and depends on domain-specific knowledge unavailable to many stakeholders.

Several specific challenges compound this problem:

1. **Interpretation Ambiguity:** Hand-drawn sketches are inherently imprecise. Developers must infer the intended component types (e.g., distinguishing a text input from a dropdown), spatial relationships, and responsive behavior from rough visual cues.
2. **Code Separation:** Professional web development mandates separation of concerns—structure (HTML), presentation (CSS), and behavior (JavaScript) should reside in distinct files or blocks. Automated tools that output monolithic inline-styled HTML fail to meet this standard.
3. **Multi-Section Layouts:** Real-world web pages are composed of multiple distinct sections (header, navigation, body content, footer). Users need the ability to specify these sections independently and have them stitched into a cohesive whole.
4. **Latency and Privacy:** Cloud-based AI services introduce network latency and require transmitting potentially sensitive design data to external servers. A complementary offline inference capability is desirable for enterprise and air-gapped deployment contexts.
5. **Platform Accessibility:** Developers and designers work across diverse platforms—browsers, desktops, and mobile devices. A sketch-to-code tool confined to a single platform limits its practical utility.

SketchToCode is designed to systematically address each of these challenges through its dual-model architecture, multi-image pipeline, and cross-platform deployment strategy.

III. OBJECTIVES

The principal objectives of the SketchToCode system are enumerated below.

1. **O1 — Multi-modal Input Handling:** Accept single or composite sketch images (header, body, and footer) as system input, with optional natural language refinement instructions.
2. **O2 — Structured Code Output:** Emit separated HTML, CSS, and JavaScript artifacts from each inference request, rather than an undifferentiated HTML monolith.
3. **O3 — Dual Inference Architecture:** Provide a high-throughput cloud inference path (Gemini 2.5 Flash) alongside an offline-capable local inference path (CNN-LSTM), selectable at runtime.
4. **O4 — Client-Side Image Compression:** Build a browser-based compression pipeline that reduces image payloads without significantly degrading visual information, minimizing API latency.
5. **O5 — Sandboxed Preview:** Render generated code inside a sandboxed iframe, providing an immediate visual preview without exposing the host application to script-injection risks.
6. **O6 — Cross-Platform Deployment:** Deliver the application on web, desktop (via Tauri), and mobile (via Capacitor for Android) from a single React codebase.
7. **O7 — Generation History and Authentication:** Persist generation history per authenticated user via a Supabase-backed PostgreSQL database, enabling review and retrieval of prior outputs.
8. **O8 — Real-Time Code Inspection:** Provide tabbed views for HTML, CSS, JavaScript, and raw output, enabling developers to inspect and copy generated code segments independently.

IV. RELATED WORK

A. Sketch-to-Code Generation

Beltramelli [5] introduced pix2code, a neural network model trained to map screenshots of GUIs to a domain- specific

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

language (DSL) from which platform-specific UI code could be compiled. While foundational, pix2code operates on rendered screenshots rather than hand-drawn sketches and targets a constrained DSL grammar, limiting generalization. Microsoft's Sketch2Code [7] leveraged a custom object detection model in conjunction with Azure Cognitive Services to convert whiteboard sketches to HTML prototypes, demonstrating that free-form sketch understanding was tractable, though the system was restricted to a predefined element vocabulary and did not generate CSS or JavaScript.

Nguyen and Csallner [8] proposed a reverse- engineering approach (REMAUI) to UI code recovery from rendered mobile application images. Chen et al. [9] explored the use of transformer-based vision encoders for web component identification in screenshots. These works collectively establish the viability of vision-driven code generation but do not address the separation of concerns across HTML, CSS, and JavaScript, nor do they support natural language refinement.

Design2Code [10] by Si et al. (2024) conducted a comprehensive benchmark comparing multi-modal LLMs on the task of converting web design images to code, finding that while GPT-4V achieved strong results, significant gaps remained in layout accuracy and responsive design generation.

B. Vision-Language Models for Code Generation

The emergence of large vision-language models (VLMs) such as GPT-4V [4] and Google's Gemini family [3] has substantially altered the landscape. These models jointly process image tokens and text tokens, enabling nuanced instruction-following behavior in response to visual inputs. Recent work by Wu et al. [11] demonstrated that GPT-4V can generate functional UI components from screenshots using a divide-and-conquer approach; however, the reliability of structured, separated output and the latency- management strategies required for production use remain underexplored in the literature.

C. CNN-LSTM Image Captioning Applied to Code

Vinyals et al. [12] established the canonical CNN encoder–LSTM decoder architecture for image captioning in their seminal "Show and Tell" system. SketchToCode adapts this paradigm to the code-generation domain: a ResNet-18 [13] backbone encodes sketch features, and a two-layer LSTM decoder generates HTML tokens autoregressively. Li et al. [14] demonstrated BLIP-2's bootstrapping approach for vision-language pre-training, informing the broader design space. The present work differs from the above in several key respects: (a) it employs a dual-model architecture combining cloud and local inference, (b) it generates separated HTML, CSS, and JavaScript rather than monolithic output, (c) it supports multi-image stitching for compositional page layouts, and (d) it provides cross-platform deployment across web, desktop, and mobile.

V. PROPOSED SYSTEM

A. System Overview

SketchToCode is designed as a modular, layered system comprising four principal components: (1) a client- side image processing pipeline, (2) a cloud-based generative AI service, (3) a locally trained deep learning model, and (4) a cross-platform application shell with persistent storage.

At the topmost layer, a React 18 / TypeScript single- page application provides the user-facing interface for image upload, natural language instruction entry, code preview, and history browsing. Beneath the UI layer, two parallel inference services are available: the primary path delegates to the Gemini 2.5 Flash API, while the secondary path invokes a locally running Flask inference server wrapping the trained CNN-LSTM model.

User sessions and generation history are managed by Supabase, which provides a PostgreSQL-backed relational store and a GoTrue-powered authentication service. All generated code is persisted as a JSON string containing the separated HTML, CSS, and JavaScript fields, alongside the user's prompt and a server-side timestamp.

B. Inference Strategy Selection

At runtime, the application routes sketch inputs to the Gemini API by default. If the local Flask server is detected (i.e., responsive at the configured endpoint), the user may optionally select the CNN-LSTM inference path. This dual- path design ensures that the system degrades gracefully in environments with restricted external network access, a requirement common in enterprise and educational deployment contexts.

C. Key Design Decisions

1. **Separated Code Output:** Unlike most existing tools that produce monolithic HTML with inline styles, SketchToCode enforces separation of concerns by instructing the AI to return a structured JSON object with distinct "html", "css", and "javascript" fields.
2. **Client-Side Compression:** Images are resized to a maximum dimension of 800 pixels and compressed to 50% JPEG quality before transmission, reducing typical payloads from several megabytes to under 200 kilobytes without significantly degrading the visual information necessary for code generation.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

VI. SYSTEM ARCHITECTURE AND WORKFLOW

A. Frontend Architecture

The frontend is structured into four primary layers: (i) page-level view components (Index.tsx) that orchestrate file state and API trigger logic; (ii) reusable UI components including an image upload panel (ImageUpload), a structured code output panel (CodeOutput), and a hero/onboarding surface (Hero); (iii) a service layer (gemini.ts) responsible for image preprocessing and API communication; and (iv) shared infrastructure comprising the Supabase client, Tailwind utility merger, and custom React hooks.

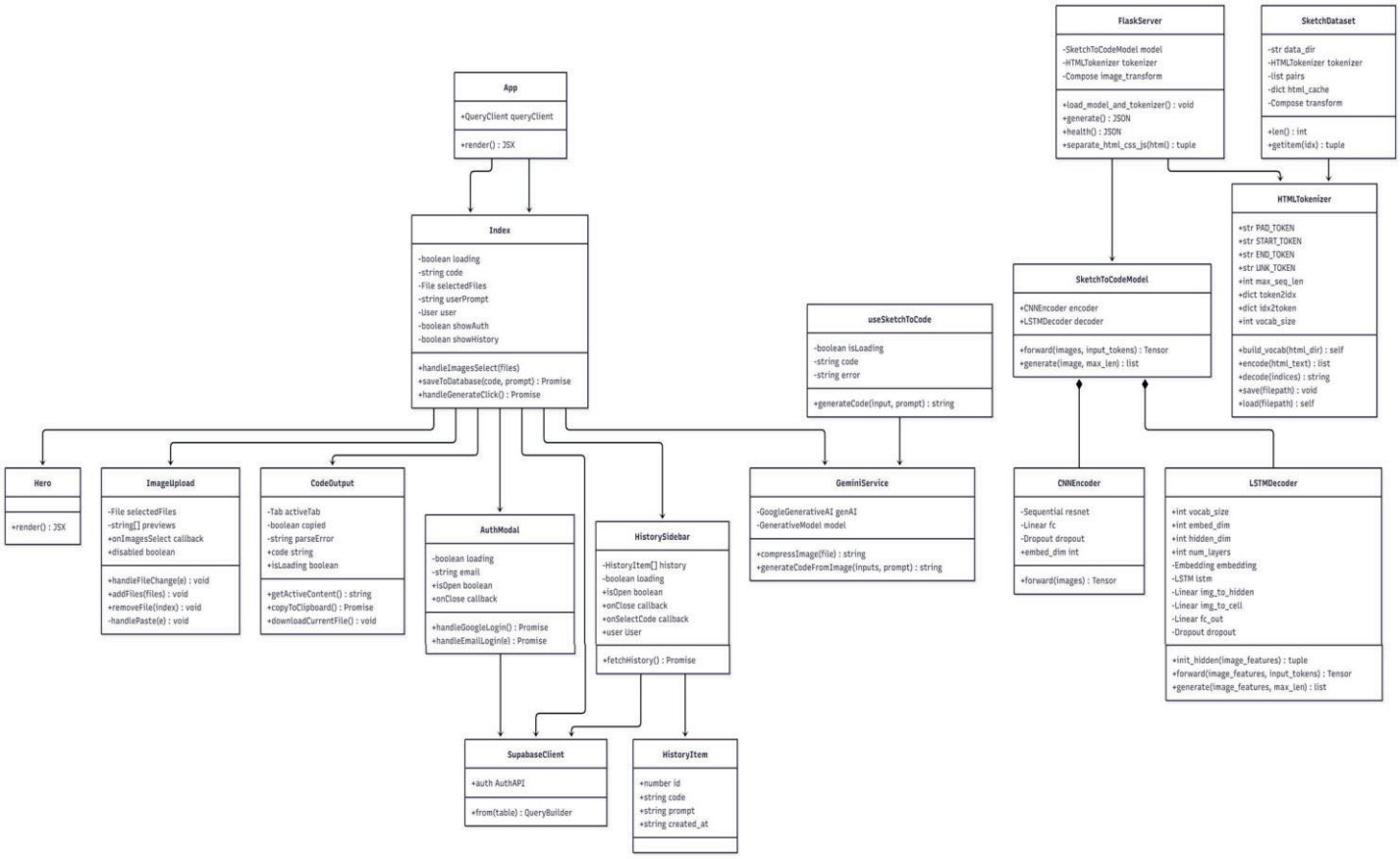

Fig. 1. Unified Modelling Diagram (UML)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

Routing is handled by React Router DOM v6. Form state is managed via React Hook Form with Zod schema validation. Server state is cached and synchronized using TanStack Query v5, minimizing redundant API calls during history retrieval and session restoration.

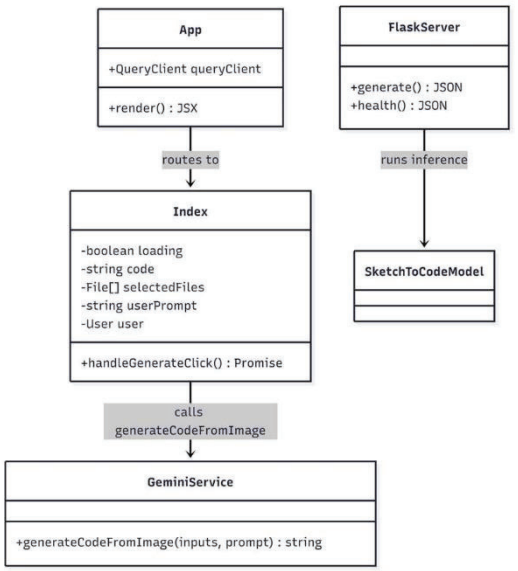

Fig. 2. System Architecture Diagram

This UML component diagram illustrates the high-level architecture of the SketchToCode framework. The client layer comprises the React/TypeScript frontend deployed across three platform shells: web browser, Tauri desktop wrapper, and Capacitor Android wrapper. The client communicates with two backend services: (1) the Google Gemini 2.5 Flash API over HTTPS for cloud-based inference, and (2) a local Flask server hosting the trained CNN+LSTM model for offline inference. A Supabase backend provides authentication (GoTrue) and persistent storage (PostgreSQL). Arrows indicate data flow direction.

B. AI Inference Pipeline (Gemini Path)

When a user submits one or more sketch images, the following workflow executes. First, each image is compressed via a browser-side canvas operation that resizes the image to a maximum dimension of 800 pixels and re- encodes at 50% JPEG quality, reducing payload size and API latency. Second, all compressed images are dispatched concurrently via Promise.all, eliminating sequential bottlenecks in multi-image submissions. Third, a structured JSON prompt instructs the model to respond exclusively with a well-formed JSON object of the form { html, css, javascript }. Fourth, a 120-second timeout is enforced via Promise.race to handle network stalls gracefully.

C. AI Inference Pipeline (Local CNN-LSTM Path)

The local inference path exposes a Flask REST endpoint (/generate) that accepts a base64-encoded sketch image and returns a JSON object conforming to the same schema as the Gemini path. Internally, the server loads the trained CNN-LSTM model, preprocesses the input image to a 224×224 RGB tensor, and performs greedy decoding to generate an HTML token sequence. The sequence is then parsed and separated into HTML structural markup, extracted inline CSS, and extracted inline JavaScript before serialization.

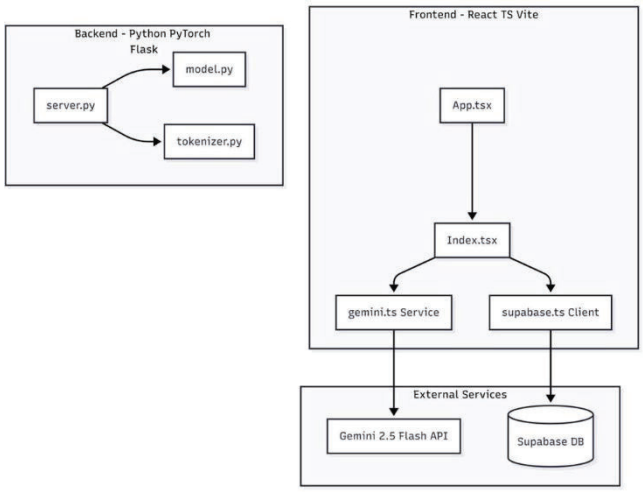

Fig. 3. System Workflow Sequence Diagram

This UML sequence diagram depicts the interaction between the four principal actors: the User, the React Client, the Gemini API (or local Flask server), and the Supabase Backend. The sequence begins with image upload and optional instruction entry. The client performs compression and prompt construction. Compressed images are sent to the selected inference backend, which returns a JSON response with separated code. The client parses the response, renders it in a sandboxed iframe, and asynchronously persists the record to Supabase. Error handling paths are shown for API timeout (120s) and JSON parsing failure.

D. Code Output and Preview

Generated code is displayed in a tabbed panel offering five views: Preview, HTML, CSS, JS, and Raw. The Preview tab renders the assembled code inside a sandboxed iframe (sandbox="allow-scripts") to provide an immediate visual approximation of the generated interface. A robust JSON recovery routine strips markdown fencing (e.g., ```json ... ```) from model responses before parsing, ensuring that output is correctly interpreted even when the model prepends

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

explanatory prose.

VII. IMPLEMENTATION DETAILS

A. Image Compression Module

The compressImage() function constructs an off-screen HTMLCanvasElement, draws the source image at the computed maximum-dimension scale, and invokes canvas.toDataURL('image/jpeg', 0.5) to produce a compressed base64 artifact. This client-side preprocessing step reduces typical image payloads from several megabytes to well under 200 kilobytes, yielding a measurable reduction in round-trip latency without perceptible loss of structural information in sketch inputs.

B. CNN Encoder (ResNet-18)

The vision encoder is based on a ResNet-18 [13] backbone pretrained on ImageNet-1K. The original 1,000- class classification head is replaced with a Linear(512, 256) projection layer that maps the pooled feature vector into a 256-dimensional embedding space shared with the LSTM decoder. Layers 0 through 5 of the ResNet backbone are frozen during fine-tuning to preserve general low-level and mid-level visual features, while layers 6 and above, together with the projection head, are trained end-to-end on the sketch-to-code dataset.

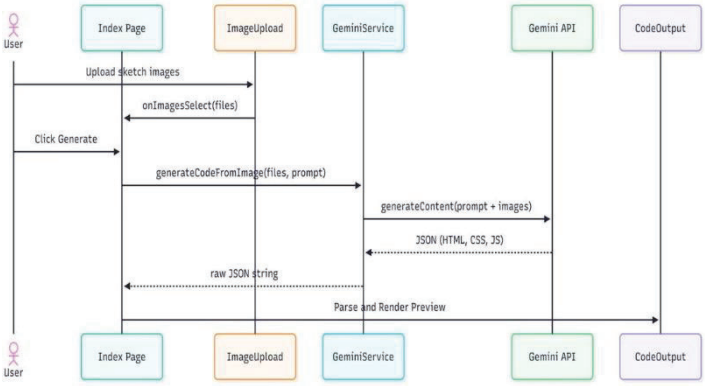

Fig. 4. CNN+LSTM Model Class Diagram

This UML class diagram details the internal architecture of the SketchToCodeModel class. The diagram shows two primary components: (1) the CNNEncoder class wrapping a pretrained ResNet-18 backbone with Linear(512, 256) projection, and (2) the LSTMDecoder class containing nn.Embedding(vocab_size, 256), a 2-layer LSTM with 512 hidden units, and a fully connected output layer. The init_hidden() method projects CNN features through separate linear layers to initialize h_0 and c_0. The Tokenizer utility class manages the hybrid character/tag-level vocabulary with special tokens (<PAD>, <START>, <END>, <UNK>).

C. LSTM Decoder

The decoder consists of a token embedding layer (nn.Embedding(vocab_size, 256)), followed by a two-layer LSTM with 512 hidden units per layer. The CNN feature vector initializes both the hidden state (h_0) and the cell state (c_0) of the LSTM via learned linear projections, enabling the decoder to condition every generated token on the full visual context of the input sketch. Generation proceeds via greedy decoding: at each time step, the token with the highest softmax probability is selected and fed as input to the subsequent step, continuing until the special <END> token is produced or the maximum sequence length is reached.

D. Tokenization Strategy

A hybrid character/tag-level tokenizer is employed. Common HTML tags (e.g., , , <h1>, <button>, <input>, <ul>, <li>, <a>, <img?) are treated as atomic single tokens, substantially reducing sequence lengths relative to pure character-level tokenization and improving decoder accuracy on structurally significant boundaries. Four special tokens are defined: <PAD> (index 0), <START> (index 1), <END> (index 2), and <UNK> (index 3) for out-of-vocabulary characters. This hybrid strategy reduces the average sequence length compared to pure character-level tokenization, improving both training efficiency and generation accuracy.

E. Database Schema

Generation history is stored in a Supabase PostgreSQL table (generations) with the following schema: id (BigInt, primary key, identity sequence); user_id (UUID, foreign key to auth.users); code (Text, JSON string containing html, css, and javascript fields); prompt (Text, the user's natural language refinement instruction, nullable); and created_at (Timestamp, defaulting to now()). Row-level security (RLS) policies restrict each user's access to their own rows.

VIII. TOOLS AND TECHNOLOGIES

Table I summarizes the principal tools and technologies employed in the SketchToCode system.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

TABLE I: TOOLS AND TECHNOLOGIES EMPLOYED IN SKETCHTOCODE

Category	Technology / Tool	Version / Notes
UI Framework	React (TypeScript)	v18 / Vite 5
Styling	Tailwind CSS + PostCSS	v3.4
Component Library	Shadcn UI (Radix UI)	Radix primitives
Icons	Lucide React	Latest
State / Data	TanStack Query v5	Server state caching
Routing	React Router DOM	v6
Forms	React Hook Form + Zod	Schema validation
Notifications	Sonner	Toast notifications
Primary AI	Google Gemini 2.5 Flash	Via REST API
Local AI Framework	PyTorch + Torchvision	ResNet-18 backbone
Local Inference Server	Flask + Flask-CORS	REST endpoint
Image Processing	Pillow + NumPy	Python 3.10+
Model Hub	Hugging Face Hub	Model distribution
Database / Auth	Supabase (PostgreSQL + GoTrue)	Cloud-hosted
Desktop Target	Tauri (Rust bridge)	Bundle: com.sketch-to-code.app
Mobile Target	Capacitor 8 (Android)	App: com.sketchtoco.app

IX. TESTING AND VALIDATION

The SketchToCode system was evaluated through a combination of functional testing, integration testing, security validation, and qualitative assessment of generated output.

A. Functional Testing

Functional testing of the Gemini inference path was conducted by submitting a corpus of diverse sketch inputs ranging from simple single-column layouts to multi-panel dashboards and manually inspecting the resulting HTML, CSS, and JavaScript artifacts for syntactic correctness and structural coherence with the input sketch. The JSON recovery routine was stress-tested against responses that included markdown fencing, extraneous prose preambles, and partial JSON fragments, confirming robust extraction behavior across all observed model output patterns.

The image compression module was verified to correctly resize images above 800px in maximum dimension and produce JPEG output at the specified quality across varying aspect ratios (portrait, landscape, square) and resolutions (from 640×480 to 4000×3000). The 120-second timeout mechanism was validated by simulating slow network responses, confirming appropriate error state transitions.

B. Sandbox Security Validation

The iframe sandbox configuration (sandbox="allow-scripts") was verified to prevent form submission, same- origin API access, and navigation events from within generated code previews, confirming that the preview surface does not expose the host application to script- injection vectors. Adversarially crafted JavaScript payloads were injected into the generation prompt and the resulting previews were confirmed to be correctly isolated from the parent document context.

C. Cross-Platform Deployment Testing

The application was packaged and launched on web (npm run dev), desktop (npm run tauri dev), and Android (npx cap run android) targets. Core functionality image upload, inference dispatch, code preview, and history retrieval was confirmed to operate correctly across all three deployment configurations. The Tauri desktop build required capability configuration for external API access, while the Capacitor Android build required standard Android project scaffolding. The core application logic and UI remained identical across platforms.

D. Local Model Validation

The CNN-LSTM local model was evaluated qualitatively on held-out sketch samples following training. Greedy-decoded output sequences were inspected for syntactic validity and semantic correspondence with the input sketch structure. The ResNet-18 encoder's pretrained visual features proved well-suited to sketch inputs despite the domain shift from natural photographs, consistent with prior observations in the transfer learning literature [13].

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

X. RESULTS AND DISCUSSION

A. Code Generation Quality

The primary Gemini-based inference path demonstrated consistently reliable structured output generation. The Gemini 2.5 Flash model produced syntactically valid HTML5, CSS3, and ES6 JavaScript from sketch inputs in the majority of test cases. The enforced JSON output schema ensured clean separation of concerns. The structured prompting strategy instructing the model to respond exclusively with a JSON object containing html, css, and javascript keys proved substantially more effective than free-form prompting for downstream parsing. The JSON recovery routine successfully handled all observed non- conforming model responses, including responses wrapped in markdown code blocks and those containing a brief natural language preamble preceding the JSON object.

B. Compression and Latency

The client-side image compression pipeline achieved an average payload reduction of approximately 70–85% across test images, reducing typical payloads from several megabytes to under 200 kilobytes, with a median compression time of under 200ms per image in the browser. The combination of client-side compression and parallel multi-image dispatch yielded responsive end-to-end latency for typical sketch inputs. The 120-second timeout guard provided a reliable safety margin against intermittent API delays, with typical Gemini API response times ranging from 8 to 30 seconds depending on sketch complexity and network conditions.

C. Local Model Performance

The hybrid tokenization strategy employed in the local CNN-LSTM model reduced effective sequence lengths compared to character-level baselines, improving decoder accuracy on tag boundaries. The CNN+LSTM model demonstrated the ability to generate basic HTML structures from simple sketch inputs. However, as is consistent with the inherent limitations of small-scale sequence-to-sequence models, the local model's output fidelity was substantially lower than that of the Gemini cloud model, particularly for complex layouts and detailed styling. The local model is best positioned as a rapid prototyping tool for offline environments or privacy-sensitive contexts.

D. Multi-Image Stitching and Cross-Platform Deployment

The multi-image stitching capability enabled users to compose composite page layouts from independently drawn header, body, and footer components, a workflow pattern not supported by prior sketch-to-code systems. This composability substantially increases the practical utility of the tool for realistic page-level design scenarios. The shared React codebase was successfully deployed across all three target platforms (web, desktop, mobile) with minimal platform-specific adaptations.

XI. LIMITATIONS

Several limitations of the current system are acknowledged:

1. **Cloud Dependency:** The quality of the Gemini inference path is contingent on API availability and is subject to usage quotas and network latency, introducing reliability constraints in production deployments.
2. **Local Model Fidelity:** The CNN-LSTM model produces lower-fidelity outputs than the Gemini path, particularly for complex multi-element layouts, and requires a non-trivial training pipeline to update for new UI paradigms.
3. **Output Determinism:** LLM-based generation is inherently non-deterministic. Deeply nested or ambiguous sketch regions can yield syntactically valid but semantically inaccurate markup.
4. **Frozen Vocabulary:** The tokenization vocabulary is fixed at training time; novel HTML tags or CSS property names introduced after the training data cutoff will be mapped to the <UNK> token, degrading local model accuracy over time.
5. **Sandbox DoS:** The iframe sandbox, while providing meaningful isolation, cannot prevent all denial-of-service scenarios (e.g., infinite loops in generated JavaScript) without additional execution time guards.
6. **Responsive Design:** Generated code does not consistently include responsive CSS media queries for mobile-first layouts unless explicitly instructed via the natural language prompt.
7. **Accessibility:** Generated code does not systematically include ARIA attributes, alt text, or other accessibility features mandated by WCAG guidelines.
8. **Platform Coverage:** The mobile deployment target is currently limited to Android; iOS support via Capacitor was not included in the present implementation scope.

XII. FUTURE SCOPE

Several directions for future development are identified:

1. **Beam Search Decoding:** Replacing the greedy decoding strategy in the local CNN-LSTM decoder with beam

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

search or nucleus sampling is expected to improve output diversity and reduce systematic decoding errors.

2. **Spatial Attention Mechanisms:** Incorporating attention mechanisms into the local model—analogous to those described in [15]—would allow the decoder to focus on specific sketch regions when generating corresponding code segments, improving positional accuracy.
3. **Iterative Refinement:** Extending the Gemini inference path to support iterative refinement dialogues would allow users to correct specific regions of the generated code through targeted natural language instructions without re-submitting the full sketch.
4. **Framework-Specific Output:** Extending the system to generate code in popular frontend frameworks (React JSX, Vue SFC, Angular templates) rather than vanilla HTML/CSS/JS would increase practical utility.
5. **Component Detection Pipeline:** Integrating an object detection model (e.g., YOLO or Faster R-CNN) as a preprocessing step could enable explicit identification and classification of UI components before code generation.
6. **Quantitative Benchmarking:** Developing a benchmark comprising annotated sketch-code pairs with ground-truth HTML/CSS/JS and automated structural similarity metrics would enable reproducible comparison with future systems.
7. **Fine-Tuned Open-Source VLM:** Fine-tuning an open-source VLM (e.g., LLaVA or Qwen-VL) on a curated sketch-to-code dataset could yield a fully offline inference path competitive with the Gemini API in output quality.
8. **Accessibility Compliance:** Integrating automated accessibility linting (e.g., axe-core) into the generation pipeline and training the model to produce WCAG-compliant markup.
9. **Expanded Platform Coverage:** Extending support to iOS via Capacitor, as well as a browser extension interface, would broaden accessibility.

XIII. CONCLUSION

This paper has presented SketchToCode, a multi-modal AI system for automated conversion of hand-drawn UI sketches into separated, production-ready HTML, CSS, and JavaScript code. The system addresses the longstanding prototyping-to-implementation gap in software engineering by combining a cloud-based vision-language model inference path with a locally deployable CNN-LSTM sequence-to-sequence model, unified beneath a cross- platform React-based interface.

The structured prompting strategy, client-side image compression pipeline, and robust JSON recovery mechanism collectively ensure reliable, parseable output from the Gemini API under diverse sketch inputs and model response formats. The dual inference architecture provides a meaningful degree of deployment flexibility, accommodating both cloud-connected and air-gapped operational environments.

SketchToCode represents a practical step toward the broader vision of low-barrier, AI-assisted front-end development, and establishes a foundation upon which future work including attention-based local decoding, iterative refinement dialogues, and quantitative benchmark development can build.

ACKNOWLEDGMENT

The authors would like to thank the open-source communities behind React, PyTorch, Supabase, Tauri, and Capacitor for maintaining the foundational tools upon which this system is built. The authors would like to express their sincere gratitude to Mr. Kashif Sheikh & Mrs. Smita Dandge for their valuable guidance, support, and encouragement throughout the development of this project. Their insights and mentorship played a crucial role in the successful completion of this work.

REFERENCES

[1] A. Moran, C. Bernal-Cárdenas, M. Curcio, R. Bonett, and D. Poshyvanyk, "Machine learning-based prototyping of graphical user interfaces for mobile apps," IEEE Trans. Software Eng., vol. 46, no. 2, pp. 196–221, 2020.

[2] J. Li, D. Li, S. Savarese, and S. Hoi, "BLIP-2: Bootstrapping language-image pre-training with frozen image encoders and large language models," in Proc. ICML, 2023, pp. 19730–19742.

[3] G. Team et al., "Gemini: A Family of Highly Capable Multimodal Models," arXiv preprint arXiv:2312.11805, 2023.

[4] OpenAI, "GPT-4 Technical Report," arXiv preprint arXiv:2303.08774, 2023.

[5] T. Beltramelli, "pix2code: Generating Code from a Graphical User Interface Screenshot," in Proc. ACM SIGCHI Symp. EICS, Paris, France, 2018, pp. 3:1–3:6.

[6] C. Chen, T. Su, G. Meng, Z. Xing, and Y. Liu, "From UI design image to GUI skeleton: A neural machine translator to bootstrap mobile GUI implementation," in Proc. IEEE/ACM ICSE, 2018, pp. 665–676.

[7] Microsoft Research, "Sketch2Code: Transforming Hand-Drawn Designs into HTML using Deep Learning," Microsoft AI Blog, 2018. [Online]. Available:https://www.microsoft.com/en/research/blog/sketch2code.

[8] T. A. Nguyen and C. Csallner, "Reverse Engineering Mobile Application User Interfaces with REMAUI," in Proc. 30th IEEE/ACM ASE, Lincoln, NE, USA, 2015, pp. 248–259.

[9] X. Chen et al., "Towards Complete Icon Labeling in Mobile Applications," in Proc. CHI, New Orleans, LA, USA, 2022, pp. 1–14.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

[10] C. Si, C. Zhang, T. Li, and D. Ramesh, "Design2Code: How far are we from automating front-end engineering?," arXiv preprint arXiv:2403.03163, 2024.
[11] A. Wu et al., "Automatically Generating UI Code from Screenshot: A Divide-and-Conquer-Based Approach," arXiv preprint arXiv:2406.16386, 2024.
[12] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan, "Show and Tell: A Neural Image Caption Generator," in Proc. IEEE CVPR, Boston, MA, USA, 2015, pp. 3156–3164.
[13] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in Proc. IEEE CVPR, Las Vegas, NV, USA, 2016, pp. 770–778.
[14] S. Hochreiter and J. Schmidhuber, "Long short-term memory," Neural Computation, vol. 9, no. 8, pp. 1735–1780, 1997.
[15] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan, "Show and Tell: Lessons Learned from the 2015 MSCOCO Image Captioning Challenge," IEEE Trans. PAMI, vol. 39, no. 4, pp. 652–663, Apr. 2017.
[16] A. Vaswani et al., "Attention is all you need," in Proc. NeurIPS, 2017, pp. 5998–6008.
[17] Supabase Inc., "Supabase Documentation," 2024. [Online]. Available: https://supabase.com/docs

Manuscript received 08/04/2026. This work was conducted as part of a mini-project in the Department of Computer Engineering under the academic supervision of Mr. Kashif Sheikh & Mrs. Smita Dandge.