Kanvas AI A No-Code Web Development Platform for Automating UI Design, Generation and Collaboration

Maheshwaran S[1], J. Lin Eby Chandra[2]
[1,2]*Department of Computer Science and Engineering Jaya Engineering College, Anna University, Chennai, India*

***Abstract*—Traditional web development imposes a significant barrier to entry, requiring proficiency across HTML, CSS, JavaScript, and backend frameworks before a functional application can be produced. Existing no-code platforms reduce this barrier but typically operate as closed ecosystems, fragmenting the workflow across separate tools for templating, visual editing, code generation, and conversational support. This paper introduces Kanvas AI, a unified no-code web development platform that consolidates four complementary modules into a single Django-based environment: an HTML Auto-Generator that produces tailored templates from user inputs, a GrapesJS-based drag-and-drop visual editor, a Code Generator and Error Solver that produces and repairs code snippets from natural language descriptions, and an NLP-powered conversational assistant named K.ai. The platform follows a five-layer modular architecture spanning presentation, application, AI/NLP services, external integrations, and data persistence, and supports clean export of user output as HTML, CSS, or JSON. We describe the system architecture, module-level design, and report unit and integration testing across all four modules. The deployed prototype on a public cloud environment validates the architecture's portability beyond local development and demonstrates that an integrated no-code workflow can meaningfully reduce context switching while preserving full ownership of user output.**

***Index Terms*—no-code development, visual editor, GrapesJS, Django, natural language processing, code generation**

I. INTRODUCTION

The global market for low-code and no-code development platforms surpassed USD 22 billion in 2024 and is projected to grow at a compound annual rate exceeding 25% through 2030 [1]. This growth reflects a broader structural shift in software production: as the demand for digital products outpaces the supply of trained developers, organizations increasingly rely on tools that allow non-technical users to construct applications through visual interfaces rather than code. Within this category, no-code web development specifically targets the rapid construction of marketing pages, dashboards, internal tools, and lightweight applications without requiring proficiency in HTML, CSS, JavaScript, or backend frameworks.

Despite the maturity of commercial offerings such as Webflow, Framer, and Shopify, three structural limitations persist. First, existing platforms are typically siloed across the development lifecycle: a user constructs visual layouts in one tool, generates copy in another, troubleshoots through documentation in a third, and exports finished output to a fourth. This fragmentation imposes a context-switching cost that disproportionately affects first-time builders. Second, most commercial platforms operate as closed ecosystems where exported assets, hosting, and extensibility are tied to the vendor, creating long-term dependency and limiting portability. Third, while AI-assisted features are increasingly available, they are typically deployed as standalone enhancements rather than as integrated reasoning partners that span template generation, visual editing, code production, and error resolution.

This paper introduces Kanvas AI, a unified no-code web development platform designed to address these limitations. Kanvas AI consolidates four complementary modules into a single Django-based environment: an HTML Auto-Generator that produces tailored templates from structured user inputs; a drag-and-drop visual editor built on GrapesJS that supports component-level customisation on a live canvas; a Code Generator and Error Solver that produces working snippets from natural language descriptions and resolves common coding issues; and an NLP-

powered conversational assistant named K.ai that provides contextual guidance across the platform.

The contributions of this work are fourfold:

- We present a unified architectural design for a no-code web development platform that integrates template generation, visual editing, AI-assisted code production, and conversational support within a single Django framework, eliminating the workflow fragmentation typical of commercial alternatives.
- We describe a GrapesJS-based visual editor extended with a four-zone workspace, viewport switching, and direct export to HTML, CSS, and JSON, ensuring that users retain full ownership of generated output and avoid platform lock-in.
- We introduce K.ai, an NLP-powered conversational assistant that leverages web-scraped knowledge sources and rule-based natural language understanding to answer queries about web development and platform features in real time.
- We deploy and validate the platform on a public cloud environment, demonstrating the portability of the architecture beyond local development setups, and report unit and integration testing results across all four modules.

The remainder of this paper is organised as follows. Section 2 surveys related work in no-code development platforms, visual editor frameworks, and AI-assisted coding tools. Section 3 describes the methods, covering system architecture, component design, and implementation. Section 4 reports unit and integration testing results. Section 5 discusses the findings in the context of existing platforms. Section 6 concludes with limitations and future directions.

II. RELATED WORK

2.1 No-Code Development Platforms

Mendez and Tran [1] characterise the rise of no-code as a response to the persistent gap between digital demand and developer supply, identifying three structural shifts that have accelerated adoption: the maturation of browser-based runtimes, the proliferation of cloud-hosted backends, and the increasing accessibility of pre-trained AI models. Cisneros and Rodriguez-Echeverria [2] survey no-code and low-code tools for IoT applications, observing that despite domain-specific variation, most platforms converge on a common architecture consisting of a visual editor, a template library, an export pipeline, and a runtime environment.

Commercial platforms such as Webflow, Framer, and Shopify exemplify this architecture at scale. Webflow targets professional designers with a CSS-grid-based visual editor and custom code injection; Framer emphasises animation and prototyping; Shopify focuses on e-commerce templates and storefront customisation. All three operate as closed ecosystems: exported assets are tied to vendor hosting, custom integrations require platform-specific APIs, and end-user output cannot be migrated without substantial rework. Kanvas AI differs from these systems in two respects. First, it operates as an open Django-based platform with no proprietary export format, allowing users to download HTML, CSS, and JSON output directly. Second, it integrates AI-assisted code generation and conversational support as first-class modules rather than as standalone enhancements.

2.2 Visual Editor Frameworks

Visual editors for web development have a long history, spanning early WYSIWYG tools such as Dreamweaver to modern component-based systems. GrapesJS [3] is an open-source visual editor framework that exposes a programmable canvas, a component model, and a styling system through JavaScript APIs. It is widely adopted as the foundation for self-hosted no-code platforms because it does not impose a runtime or hosting dependency. Kanvas AI builds on GrapesJS for the visual editor module, extending it with a four-zone workspace consisting of a toolbar, components panel, live canvas, and styling panel, viewport switching between desktop and mobile, and direct integration with the platform's template generation and export pipelines.

2.3 AI-Assisted Code Generation and Error Resolution

Zeydan and Ergun [4] argue that the future of software development will be characterised by hybrid workflows in which AI-assisted tools handle boilerplate and repetitive code, freeing human developers to focus on design and architectural decisions. Da Silva et al. [5] conduct a systematic mapping study of no-code programming platforms and identify code generation and error resolution as the two most consistently requested AI capabilities. Despite this demand, most commercial no-code

platforms deploy code assistance as a separate documentation feature rather than as an integrated module that interacts directly with the visual editor and template generator. Kanvas AI addresses this gap by implementing the Code Generator and Error Solver as a first-class module within the Django application layer, enabling natural language code requests and automated error resolution within the same workflow as visual editing.

2.4 Conversational Assistants for Development

Ahrens and Irgens [6] survey the democratisation of software development and observe that conversational assistants have emerged as a key mechanism for lowering the technical barrier to entry. Beyond commercial chat interfaces, integrated assistants embedded within development platforms allow users to ask context-aware questions without leaving their workflow. K.ai, the conversational assistant integrated into Kanvas AI, is designed for this scenario: it uses rule-based natural language understanding combined with web-scraped knowledge sources to answer queries about web development concepts, platform features, and specific user tasks. The assistant is available as a floating launcher on every page of the platform, reducing the friction of moving between documentation, search engines, and the editor.

III. METHODS

This section presents the methodological design of Kanvas AI, covering the layered system architecture, the decomposition of the application layer into four modules, the component-level structure of each module, and the implementation choices that realise the design.

3.1 System Architecture

Kanvas AI follows a five-layer modular architecture designed to separate user-facing presentation, application logic, AI services, external integrations, and data persistence into distinct subsystems.

The Presentation Layer handles all browser-side rendering through HTML, CSS, JavaScript, and Ajax. It exposes four user-facing surfaces: the Landing Page, the No-Code Editor, the Templates browser, and the Code Tools interface. All four surfaces share a common shell and floating chatbot launcher, providing consistent navigation across the platform.

The Application Layer, built on the Django framework, orchestrates the four core modules through routing, views, ORM access, and session management. The four modules contained within this layer — the HTML Auto-Generator, the Drag and Drop Visual Editor, the Code Generator and Error Solver, and the K.ai Chatbot Service — each correspond to a distinct functional concern and communicate with each other through the shared Django session and database.

The AI/NLP Services layer sits beneath the Application Layer and provides two services: a Query Parser that interprets user input from the chatbot, and a Response Generator that crafts contextual replies. The External Integrations layer, also beneath the Application Layer, manages two outbound services: a Web Scraping module that retrieves supplementary information from external sources, and a Code Snippet Repository that supplies pre-built code templates to the Code Generator.

The Data Persistence Layer, managed through SQLite and Django's ORM, stores user projects, templates, blog content, and cached generated code. Communication between layers is handled through well-defined interfaces and standardized data formats, ensuring that each subsystem can evolve independently without disrupting the platform as a whole.

3.2 Decomposition of Modules

The Application Layer is decomposed into four modules, each responsible for a distinct functional concern.

3.2.1 HTML Auto-Generator

The HTML Auto-Generator produces complete, responsive HTML templates from structured user inputs. Users specify the project type (portfolio, landing page, blog, dashboard, store, or restaurant), a stylistic vibe (minimal, bold, dark, warm, playful, or corporate), and any additional requirements through a guided form. The module merges these inputs with a library of pre-defined template structures, applies layout and styling rules, and produces a single HTML document that opens directly within the visual editor for further refinement.

3.2.2 Drag and Drop Visual Editor

The Drag and Drop Visual Editor, built on GrapesJS, presents a four-zone workspace consisting of a toolbar, a components panel, a live canvas, and a

styling panel. Users compose pages by dragging components from the components panel onto the canvas, then customise each component through styling controls for colour, spacing, typography, and layout. The editor supports viewport switching between desktop and mobile, layer management, file naming, and direct export of completed designs as HTML, CSS, or JSON.

3.2.3 Code Generator and Error Solver

The Code Generator and Error Solver module addresses two complementary concerns within a single subsystem. The Code Generator accepts natural language descriptions of desired functionality and produces working code snippets in the language and framework specified by the user. The Error Solver accepts broken code or error messages as input, identifies the underlying issue using a combination of pattern matching and rule-based analysis, and returns a corrected version. Both functions are accessible through a unified interface that gathers user specifications, routes the request to the appropriate sub-component, and presents the result.

3.2.4 K.ai Chatbot Service

The K.ai Chatbot Service provides conversational support across the platform through an NLP-powered assistant. K.ai interprets user queries using rule-based natural language understanding, retrieves relevant information from internal knowledge sources and external web references through the web scraping module, and returns contextual responses formatted for in-place display. The assistant is available as a floating launcher on every page and supports queries about web development concepts, platform features, and specific user tasks.

3.3 Component Design

This subsection details the internal structure of each module by describing the principal classes, their responsibilities, and the runtime interaction patterns that connect them.

3.3.1 HTML Auto-Generator

The HTML Auto-Generator is composed of three principal classes. The HTMLTemplateGenerator class orchestrates the template generation process and exposes the public generateTemplate method. It maintains a theme attribute that determines the stylistic direction of the output. The ContentDataFetcher class is responsible for retrieving dynamic content data from internal sources or user inputs; it maintains a sourceURL attribute and exposes the fetchData and validateContent methods. The TemplateMerger class amalgamates the fetched data with predefined HTML templates; it holds a baseTemplate attribute and exposes the mergeDataWithTemplate and applyStyles methods. The HTMLTemplateGenerator collaborates with both ContentDataFetcher and TemplateMerger through one-to-one associations.

At runtime, the client invokes generateTemplate on the HTMLTemplateGenerator. The generator delegates to ContentDataFetcher to obtain the content data, receives the validated data in return, and forwards it to TemplateMerger together with the base template. TemplateMerger applies the styling rules, returns the merged HTML, and the generator returns this final document to the client. The entire interaction completes within a single request-response cycle, allowing the generated template to open directly within the visual editor.

3.3.2 K.ai Chatbot Service

The K.ai Chatbot Service is composed of four principal classes organised around a central controller. The ChatbotController class maintains the session state through a sessionId attribute and exposes the interactWithUser, retrieveExternalData, and craftResponse methods. The UserInterface class manages the input and output rendering, maintaining an inputBuffer attribute and exposing the displayInterface, getUserInput, and renderMessage methods. The DataFetcher class retrieves supplementary information from web sources through the fetchData, scrapeWebSource, and parseResults methods, parameterised by a queryString attribute. The ResponseGenerator class crafts the final response through the generateResponse, applyNLP, and formatOutput methods, parameterised by an intent attribute. The ChatbotController aggregates the other three classes through one-to-one associations.

A user query initiates a sequence of interactions across these components. The user submits a query through the UserInterface, which forwards it to the ChatbotController. The controller invokes the NLP engine to parse the query and produce an interpreted intent, then queries the DataFetcher to retrieve relevant external data. The retrieved data is passed to the ResponseGenerator, which crafts a formatted reply. The controller returns the response to the UserInterface for display. The entire interaction is

asynchronous: the user can continue interacting with the platform while the chatbot processes their query in the background.

3.3.3 Code Generator and Error Solver

The Code Generator and Error Solver is decomposed into five classes split across two functional groups. The Code Generation group contains the CodeGenerator class, which maintains a language attribute and exposes the generateCode and validateInput methods; the UserInputHandler class, shared with the Error Resolution group, which maintains an inputData attribute and exposes the gatherInputData and parseSpecifications methods; and the CodeTemplateRepository class, which maintains a templateList attribute and exposes the accessTemplate and loadByLanguage methods. The Error Resolution group contains the ErrorSolver class, which maintains an errorLog attribute and exposes the identifyError and resolveError methods; and the SolutionProvider class, which maintains a knownFixes attribute and exposes the proposeSolution and automatedFix methods. The SolutionProvider is related to the ErrorSolver through aggregation, and both groups share the UserInputHandler.

At runtime, the module supports two distinct flows. In the code generation flow, the user submits a request to the UserInputHandler, which gathers specifications and invokes the CodeGenerator. The generator accesses the CodeTemplateRepository, retrieves the appropriate template for the requested language, produces the snippet, and returns it through the UserInputHandler to the user. In the error resolution flow, the user submits an error to the UserInputHandler, which invokes the ErrorSolver. The solver consults the SolutionProvider, retrieves a proposed fix, applies it, and returns the corrected output through the UserInputHandler to the user. Both flows are handled by the same Django view, branching internally based on the request type.

3.4 Implementation

3.4.1 Technology Stack

Kanvas AI is implemented as a Django application using Python 3.9 on the backend. The frontend is composed of HTML, CSS, and JavaScript, with GrapesJS providing the visual editor framework and Ajax handling asynchronous communication with the Django views. SQLite serves as the development database, accessed through Django's Object-Relational Mapping (ORM) layer. The AI/NLP subsystem combines rule-based natural language understanding for K.ai with a web scraping module that retrieves supplementary information from external knowledge sources.

3.4.2 Module Implementation

The HTML Auto-Generator is implemented as a Django view that accepts a structured form submission and produces a complete HTML document. The view loads a base template from a Python module, substitutes user-specified content into designated placeholders, applies a random colour theme from a curated set, and returns the assembled document as the response. Generated templates open directly within the visual editor for further refinement, eliminating the export-import round-trip typical of standalone template generators.

K.ai is implemented as a combination of a JavaScript front-end and a Django back-end. The front-end captures user input through a textarea, formats it as a chat bubble, and dispatches it to the back-end through an Ajax request. The back-end parses the query, retrieves matching content from a curated knowledge base or through web scraping when necessary, and returns a formatted response. The response is rendered as a chat bubble in the front-end, accompanied by a timestamp. A floating launcher is rendered on every page through a shared template include, ensuring K.ai is available regardless of which surface the user is currently on.

The Code Generator and Error Solver share a unified Django view that branches on the request type. For code generation, the view accepts a natural language description and produces a code snippet by combining the description with a language-specific template from the repository. For error resolution, the view accepts broken code or an error message, identifies the underlying issue through rule-based analysis, and returns a corrected version. Both flows return their output to the client through a JSON response that the front-end renders as a formatted code block.

The visual editor is implemented by integrating GrapesJS into a Django template and configuring it with a custom component library that includes basic blocks, grids, models, tables, images, badges, widgets, and menus. The editor exposes layer management, styling controls for colour, spacing, and typography, viewport switching between desktop and mobile, file naming, and export buttons for HTML, CSS, and

JSON. The editor state is serialised to JSON for persistence in the user's project record.

3.4.3 Deployment

The platform is deployed on a public cloud environment using Render's free hosting tier. The deployment configuration uses a build script that installs Python dependencies, runs database migrations, and collects static files through Django's collectstatic command. The application is served by Gunicorn behind Render's load balancer. The deployment validates the portability of the architecture beyond local development setups and confirms that the platform operates correctly under production-style hosting constraints. The live instance is accessible at https://kanvas-ai.onrender.com.

IV. RESULTS

We evaluate the platform through unit testing of each module and integration testing of the complete development workflow. Test cases are derived from the functional requirements specified in Section 3 and exercise both typical use cases and edge conditions.

4.1 Unit Testing

Unit tests for the HTML Auto-Generator verify that the module produces responsive templates aligned with user specifications, handles oversized image and text inputs gracefully, and renders correctly across desktop and mobile viewports. Unit tests for the K.ai Chatbot verify that the assistant interprets a representative sample of user queries correctly, requests clarification when intent is ambiguous, and requires confirmation before executing potentially destructive operations. Unit tests for the Code Generator and Error Solver verify that the module generates syntactically correct code for representative requests in multiple languages, avoids false-positive syntax error flags on correct input, and proposes valid fixes for common error patterns such as null pointer exceptions.

4.2 Integration Testing

Integration tests exercise the complete development workflow, from initial template generation through visual editing, code assistance, and final export. A representative scenario begins with a user requesting a new landing page through the HTML Auto-Generator, refining the result in the visual editor, asking K.ai for help adding a contact form, generating the form code through the Code Generator, and exporting the final result as HTML and CSS. Integration tests verify that data flows correctly between modules, that intermediate states are preserved across user actions, and that the final export reflects all edits made within the platform.

4.3 Deployment Validation

Beyond functional testing, we validate the platform's behaviour under deployment conditions on Render's free hosting tier. The deployed instance correctly serves the landing page, all module interfaces, and the static assets generated through collectstatic. The platform handles concurrent sessions without state leakage and recovers gracefully from cold-start latency typical of free-tier hosting. These results confirm that the architecture supports operation beyond a local development environment.

4.4 Test Outcomes

A consolidated summary of unit and integration test outcomes is presented in Table 1 (after the references). All tests passed under standard operating conditions. The deployed instance on Render's free tier passed all integration tests; cold-start latency was observed as an operational concern but does not affect functional correctness.

V. DISCUSSION

The results reported in Section 4 confirm that the integrated four-module architecture operates reliably across both functional and deployment dimensions. Three observations emerge from these results that are worth situating against the commercial and academic landscape surveyed in Section 2.

First, the integration testing confirms that consolidating template generation, visual editing, code assistance, and conversational support inside a single Django framework eliminates the export-import round-trips that fragment the workflow on commercial platforms such as Webflow, Framer, and Shopify. A user who begins with an auto-generated template, edits it visually, asks K.ai for guidance, and exports the result performs the entire workflow without leaving the platform — a result that directly addresses the workflow-fragmentation limitation identified at the outset.

Second, the successful direct export to HTML, CSS, and JSON validates the design choice of operating as an open Django-based platform rather than a closed ecosystem. Unlike vendor-bound exports, Kanvas AI's output is portable to any standard hosting environment, supporting the long-term ownership of user output that closed commercial platforms cannot offer.

Third, the deployment validation on a free-tier public cloud environment demonstrates that the architecture is not dependent on developer-specific tooling or local environments. The cold-start latency observed on the free tier is a property of the hosting plan rather than the architecture, and the integration tests pass cleanly under production-style hosting constraints. This portability is a non-trivial differentiator for an academic prototype: many no-code research systems are validated only on local development setups, leaving the question of cloud operability unanswered.

The principal limitation observed during testing is the rule-based design of the K.ai NLP layer, which constrains the assistant's ability to handle queries that fall outside its curated knowledge base. Section 6 discusses how this limitation, along with several others, motivates the future-work directions.

VI. CONCLUSION AND FUTURE WORK

6.1 Conclusion

This paper presented Kanvas AI, a no-code web development platform that integrates four core modules into a single Django-based environment: an HTML Auto-Generator, a drag-and-drop visual editor built on GrapesJS, a Code Generator and Error Solver, and an NLP-powered chatbot named K.ai. Together, these modules address the central problem identified at the outset: that traditional web development creates a significant barrier for individuals without coding expertise, and that existing no-code platforms fragment the workflow across separate tools and impose vendor lock-in on user output.

The implementation demonstrates that a unified no-code environment can meaningfully reduce the time and skill required to produce functional web pages. Users can begin with an auto-generated template, refine it visually on the canvas, request code snippets in natural language, and resolve errors through automated suggestions, all without leaving the platform. Outputs can be exported as clean HTML, CSS, or JSON, ensuring users retain full ownership of their work and avoid platform lock-in.

The platform has been deployed on a public cloud environment, validating the architecture's ability to operate beyond a local development setup. Testing across the four modules confirms that the system performs reliably under standard use cases, and integration testing verifies that the modules work cohesively as a unified development workflow. In achieving these outcomes, Kanvas AI fulfils its stated objective of democratising web development and provides a working foundation for further research into AI-assisted no-code tools.

6.2 Future Work

While the current version of Kanvas AI is functionally complete, several directions are identified for future enhancement. First, the current K.ai chatbot relies on rule-based NLP and web scraping; integrating a large language model would significantly improve the quality of code generation, error resolution, and conversational support, enabling the system to handle more complex and ambiguous user queries. Second, the platform presently supports single-user editing; future iterations could introduce multi-user collaboration on the same canvas through WebSocket-based synchronisation, allowing teams to design and build together remotely. Third, a community-driven component marketplace would enable users to publish, share, and reuse custom components and templates, fostering ecosystem growth and reducing time-to-launch for new projects. Fourth, adding Git-style versioning would allow users to track changes, revert to earlier states, and branch their projects, bringing professional development workflows into the no-code environment. Fifth, the deployed instance currently runs on a free-tier cloud service with cold-start latency; migrating to a production-grade hosting environment with caching, content delivery network integration, and database scaling will be necessary for serving a larger user base. Together, these directions extend Kanvas AI from a working academic prototype into a platform with the potential for sustained real-world adoption.

Declarations

Funding

No funding was received for conducting this study.

Conflicts of Interest

The authors declare no conflict of interest.

Data Availability

The Kanvas AI source code and deployment configuration are publicly available at the project repository. A live deployment is available at https://kanvas-ai.onrender.com.

Author Contributions

Maheshwaran S: Conceptualization, system design, implementation, deployment, writing. J. Lin Eby Chandra: Supervision, methodology validation, manuscript review and editing.

Use of Generative AI

The authors used a generative AI assistant (Anthropic's Claude) to support language refinement, structural organization, and proofreading of the manuscript. All technical content, system design, implementation, testing, and conclusions are the original work of the authors. AI-generated suggestions were reviewed, edited, and verified by the authors, who retain full intellectual responsibility for the manuscript. No AI tool was listed as an author, and no part of the research or results was generated by AI without author oversight.

ACKNOWLEDGEMENTS

The authors thank the Department of Computer Science and Engineering, Jaya Engineering College, for the support extended throughout this project.

REFERENCES

[1] M. Mendez and K. Tran, "The rise of no-code: empowering businesses to develop applications without coding," in Proc. 53rd Hawaii Int. Conf. System Sciences, 2020.

[2] R. Cisneros and R. Rodriguez-Echeverria, "No-code/low-code tools for developing IoT applications," Journal of Systems Architecture, vol. 108, p. 101776, 2020.

[3] GrapesJS, "GrapesJS: free and open source web builder framework," 2024. [Online]. Available: https://grapesjs.com

[4] M. Zeydan and E. Ergun, "The future of software development: no-code/low-code platforms," in Proc. Int. Congress on Cybernetics and Systems, vol. 1, pp. 271–281, 2021.

[5] C. Da Silva, P. Ferreira, and J. Barata, "No-code programming platforms: a systematic mapping study," in Proc. 8th Int. Conf. Model-Driven Engineering and Software Development, pp. 255–262, 2020.

[6] D. Ahrens and C. Irgens, "Democratizing software development: a systematic mapping study on no-code platforms," Journal of Software: Evolution and Process, vol. 32, no. 5, p. e2286, 2020.

[7] D. Casanueva, W. Fuertes, and J. C. Anacleto, "A systematic mapping study on no-code and low-code platforms for business process management," Journal of Systems and Software, vol. 179, p. 110909, 2021.

[8] Django Software Foundation, "Django: the web framework for perfectionists with deadlines," 2024. [Online]. Available: https://www.djangoproject.com

[9] D. Jurafsky and J. H. Martin, Speech and Language Processing, 3rd ed. Upper Saddle River, NJ, USA: Prentice Hall, 2023.

[10] R. Mitchell, Web Scraping with Python: Collecting More Data from the Modern Web, 2nd ed. Sebastopol, CA, USA: O'Reilly Media, 2018.