IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 1, January 2026

DOI: 10.17148/IJARCCE.2026.15145

Prompt2Extension: A System for Generating Functional Browser Extensions from Natural Language Prompts

Sanyam Jain[1], Mayank Mishra[2], Aditya Palan[3], Devendra Bodkhe[4]

Department of Artificial Intelligence and Data Science, TCET, Mumbai, India[1-4]

Abstract: This paper introduces Prompt2Extension, that allows users to create functional, installable browser extensions using natural language prompting. Even though the power of the large language models (LLMs) has shown a tremendous effect on the software development process, designing user-specific browser extensions in an automated manner is not an easy task. The sophisticated, prompt-centric architecture of our system can convert high-level user requirements into multi file browser extensions with massive scale and using the Gemini 1.5 Pro API of Google. We rely on a complex Chain-of-Thought system prompt that forces the LLM to strategize the required files and permissions, judge the request and explain the logic prior to creating the final code. The formatted JSON object created by this one-shot generation process contains all required files (manifest.json, JavaScript, HTML and CSS) and is bundled into a downloadable .zip file. One of the features of our React-based user interface is the ability of the user to look at the generated code before installation. Experiments have shown that this prompt-driven method can effectively create a broad spectrum of extensions, including simple style manipulators as well as more complex, event-driven extensions that need background scripts. This piece shows how advanced prompt engineering can become in order to turn a specialized field of software development into a far more accessible field by allowing non-programmers to build their own original web tools.

Keywords: Prompt Engineering, Code Generation, Browser Extensions, Natural Language Processing, Gemini, Large Language Models, Fast-API, React, Chain-of-Thought

I. INTRODUCTION

Browser extensions represent a powerful mechanism for personalizing the web, yet their development remains largely inaccessible to non-programmers. The creation of even simple extensions requires proficiency in a triad of web technologies—HTML, CSS, and JavaScript—and an understanding of platform-specific manifest configurations and APIs. This technical barrier effectively excludes a vast population of users from creating bespoke tools to enhance their own productivity and browsing experience. While recent advances in Large Language Models (LLMs) have revolutionized code generation for self-contained scripts, the automated generation of multi-file, platform-compliant software projects like browser extensions remains an open challenge. This paper introduces Prompt2Extension, a system that directly addresses this challenge by translating high-level, natural language descriptions into functional, installable browser extensions. Our core contribution is a prompt-centric architecture that leverages a sophisticated Chain-of-Thought (CoT) methodology to guide a generative model through the entire development lifecycle in a single pass. This approach handles intent analysis, architectural planning (including file scaffolding and permission declaration), and multi-file code generation within a single, coherent process. By doing so, Prompt2Extension significantly lowers the barrier to entry for software creation, offering a compelling model for prompt-driven development and empowering non-technical users to build their own customized web utilities.

II. LITERATURE REVIEW

The automated translation of human intent into executable code is a long-standing goal in artificial intelligence. Historically, this endeavor focused on formal program synthesis from specifications. Early systems like Tellina demonstrated the feasibility of translating natural language into specific formal languages, such as bash shell scripts, using Recurrent Neural Networks to parse commands and arguments [14]. Concurrently, research into flexible web navigation, such as the FLIN system, sought to map user commands to concept-level actions rather than brittle UI components, framing the problem as a ranking task to generalize across different websites [13]. These foundational efforts highlighted both the promise and the complexity of bridging the gap between ambiguous language and structured code. The advent of Large Language Models (LLMs) has fundamentally transformed this landscape, enabling a shift from single-function synthesis to the management of com plex, multi-file software projects. As detailed in comprehensive

International Journal of Advanced Research in Computer and Communication Engineering
Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 1, January 2026
DOI: 10.17148/IJARCCE.2026.15145

surveys on the topic, the focus has evolved towards LLM-based autonomous agents capable of handling the entire software development lifecycle (SDLC) [1]. This new paradigm has given rise to competing philosophies in agent design. One prominent approach, exemplified by AGENTOCCAM, champions architectural simplicity. It posits that agent performance is best maximized not by creating complex, hand-crafted control flows, but by carefully refining the agent's observation and action spaces to align with the intrinsic capabilities of the underlying LLM [5]. By simplifying the agent's interaction with the environment, the full reasoning power of the model can be more effectively leveraged. In contrast, other systems employ more intricate, multi-stage methodologies to tackle the complexities of web automation. The STEWARD system, for instance, operates as an end-to-end tool that accepts natural language instructions and reactively plans a sequence of actions on live websites, achieving high efficiency through a combination of LLM-driven planning and direct browser automation [6]. Similarly, AutoWebGLM utilizes a sophisticated multi-stage training regimen—including curriculum learning, reinforcement learning from its own errors, and rejection sampling—to enhance the web navigation proficiency of a smaller, open-source model [7]. These systems demonstrate powerful capabilities in executing tasks within a web environment. However, a critical distinction must be made: these advanced agents are designed to produce ephemeral, agentic behaviors—a sequence of actions to achieve a goal. They are not designed to produce a durable, user-owned software artifact. Our work addresses a different class of problem: the generation of a complete, installable browser extension. This task introduces a unique set of challenges not central to web automation agents, including the need to generate a valid manifest file (manifest.json), correctly declare permissions, structure code across multiple files (background scripts, content scripts, popup HTML/CSS/JS), and package the final output for installation. This shifts the focus from dynamic action planning to static code architecture and platform compliance. Underpinning all these advancements is the field of prompt engineering. As systematic surveys have shown, the method of structuring the input prompt has a profound impact on the reasoning, planning, and code generation quality of LLMs [3][4]. Techniques range from providing few-shot examples to more complex approaches like Chain-of-Thought (CoT), which explicitly instructs the model to "think step by step" before providing a final answer. The effectiveness of prompt engineering is not just theoretical; case studies demonstrate its impact on enhancing autonomous learning and enabling complex task completion [11][12]. Some research even formalizes automatic prompt engineering as a mathematical optimization problem, seeking the optimal prompt structure to maximize performance on a given task [4]. Our system, Prompt2Extension, is built upon this principle, positing that a single, meticulously crafted system prompt can guide an LLM to manage the entire architectural planning and code generation process for a multi-file project. The final piece of the research context is evaluation. As LLM-based systems become more complex, so too must the methods for assessing them. Recent surveys on agent evaluation call for rigorous benchmarks that test not just task completion but also agent capabilities like tool use, planning, and reasoning in realistic environments [8]. To this end, benchmarks like Web-Bench have been introduced specifically to evaluate LLMs on real-world, multi-file web development workflows, including browser extension generation [2]. Complementary frameworks like DeepEval provide open-source tools for comprehensive, metric-based evaluation of LLM-generated code, assessing fac tors like correctness, hallucination, and relevancy [10]. These tools highlight a critical need in the field: robust, project-scale evaluation. In summary, while the literature demonstrates significant progress in LLM-powered web agents and prompt engineering, a research gap exists in the area of generative tooling for user-owned, platform-compliant software artifacts. This is further highlighted in recent roadmaps for the field which identify context management and dependency resolution as key open challenges [9]. Furthermore, understanding the human-in-the loop component, including user behavior and the cognitive costs of interacting with AI assistants, is critical for designing the next generation of generative tools [15]. Prompt2Extension is designed to fill this gap, leveraging advanced prompt engineering to transform the specialized task of browser extension development into an accessible, on-demand process for non-programmer

III. SYSTEM ARCHITECTURE AND PROPOSED FRAMEWORK

The architecture of Prompt2Extension is deliberately minimalist, embodying a client server model designed to offload the cognitive burden of extension generation entirely onto a single, powerful LLM. This design philosophy stands in direct contrast to traditional multi-stage pipeline approaches that rely on a cascade of specialized components for parsing, planning, and code generation. Our decision to centralize intelligence within a single, comprehensive system prompt is strategic: it prioritizes flexibility and capitalizes on the emergent, holistic reasoning capabilities of state-of-the-art models like Gemini 1.5 Pro. By trading bespoke, brittle engineering for advanced prompt craft, the system can adapt to a wider range of user requests without modification to its own codebase, treating the LLM as the core, reconfigurable engine.

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 1, January 2026

DOI: 10.17148/IJARCCE.2026.15145

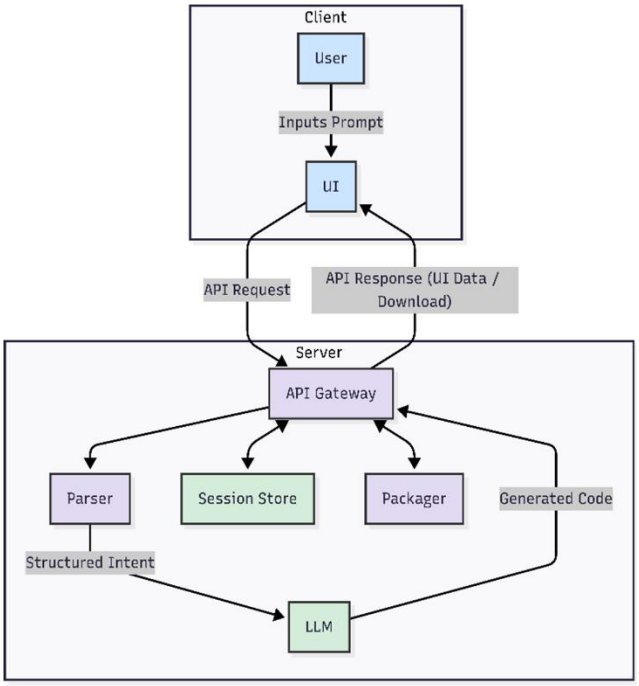

Fig. 1 System Architecture of Prompt2Extension. This diagram illustrates the high-level client- server model, where a user's natural language prompt is transmitted via an API request to the server, which orchestrates the LLM to generate and package the final browser extension.

3.1 React Client-Side Architecture

It is a single-page modern application that is developed with the help of the React framework. Its main duties shall be to offer a clean user interface and deal with the backend communication.

1. User Interface (UI): The UI shows a plain text field where the user is supposed to type their natural language prompt. It also has controls to start the generation process and, when it completes, there are download buttons to generate the final extension and to review the generated code.

2. API Communication: The React client makes a POST request to a single end point on the FastAPI server whenever the user makes a prompt. It then processes the special response by the server, which has two data elements:
a. Response Body: The received installable .zip file is a Blob that is ready to be downloaded.
b. Custom HTTP header (X-Generated-Files): The generated code of a given generated file, which is a JSON string. The client uses this header to fill the "Inspect the Code" feature without having to make a second API call.

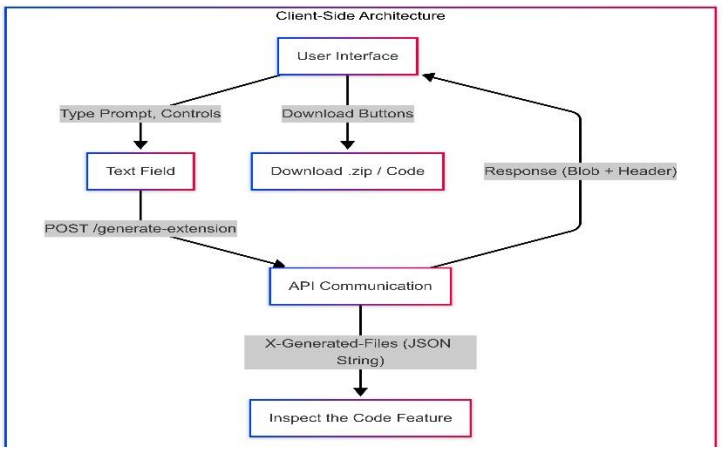

Fig. 2 Client-Side Architecture. The diagram details the user interaction flow within the React application, from inputting a prompt to initiating the API call and handling the dual-modality response containing the downloadable '.zip'file (Blob) and the code for inspection (HTTP Header).

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 1, January 2026

DOI: 10.17148/IJARCCE.2026.15145

3.2 Server-side Architecture (FastAPI)

The server-side is a performance-intensive, asynchronous Python FastAPI-based application. It is a lean orchestrator, whose rationale lies in the interaction with the LLM.

1. API Endpoint (/generate-extension): The server has one endpoint where the prompt of user is received. This endpoint has the controls of the whole generation workflow throughout.

2. Prompt Engineering Module (System-PROMPT):is the mind of the system. It is not a program in itself but a highly crafted block of text that is used as the master set of instructions to the AI. It has a Chain-of-Thought approach to it, and the LLM is taught to generate a plan in the form of <thinking> tags, then generate the final code. All of the parsing and planning is part of this prompt because it directs the internal processing of the LLM.

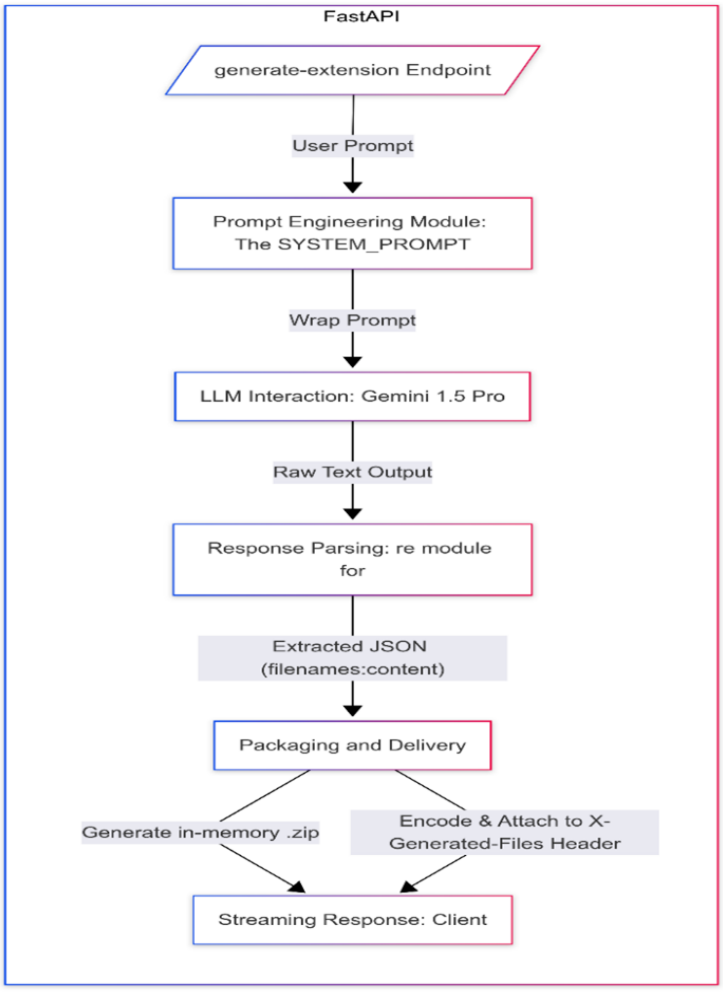

Fig. 3 Server-Side Architecture. This flowchart illustrates the workflow on the FastAPI server, beginning with the user prompt being wrapped by the Chain-of-Thought 'SYSTEM-PROMPT', interaction with the Gemini 1.5 Pro API, response parsing, and final packaging for delivery to the client.

3. LLM Interaction (Gemini 1.5 Pro): FastAPI server wraps the simple prompt issued by the user in the large SYSTEM-PROMPT and sends the entire text to the Gemini 1.5 Pro API through one call.

4. Response Parsing: The server parses the raw text input the LLM provided and uses Python's re module (regular expressions), which is sure to locate and extract the JSON string in the 'json-output' tags. This renders the system robust towards any unwanted text or notes that the AI may produce beyond the stipulated output block.

5. Packaging and Delivery: The server does both of the following things when the JSON is extracted and converted into a dictionary of filenames and their contents: It relies on the zip file library of Python to generate the installable .zip file in memory.

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 1, January 2026

DOI: 10.17148/IJARCCE.2026.15145

IV. METHODOLOGY

The methodology introduces a streamlined approach for generating multi-file software projects using a Large Language Model (LLM). This technique circumvents traditional, multi-stage pipelines of parsing and generation by employing a single, comprehensive prompt that guides the LLM through a structured reasoning and generation process. The entire operation, from user intent to a downloadable software artifact, is executed within a single API call.

A. Chain-of-Thought for Architectural Planning

Our methodology is centered on a structured reasoning paradigm enforced by a single, comprehensive Chain-of-Thought (CoT) system prompt. The critical insight driving this approach is that the primary failure mode in generating multi-file applications is not syntactic error within a single file, but architectural incoherence between files. To mitigate this, our CoT prompt mandates a preliminary planning phase where the LLM must externalize its architectural strategy before generating code. This process, which includes explicit steps for goal analysis, file scaffolding, and permission declaration, forces the model to commit to a consistent blueprint. For example, by requiring the LLM to first declare its intent to use a background script, it is far more likely to correctly request the necessary permissions in the manifest.json and place event listeners in the appropriate file. This preemptive planning directly addresses the inter-file dependency challenges unique to browser extension development, leading to a marked improvement in the coherence and functional correctness of the generated artifact.

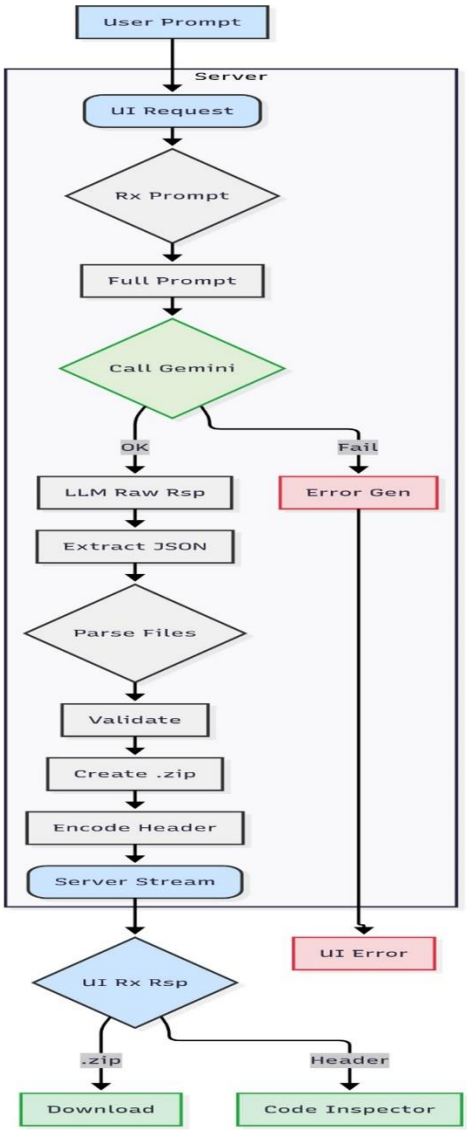

Fig. 5 End-to-End Methodology Flow. This flowchart provides a comprehensive view of the entire process, starting from the user prompt and detailing the server's internal steps, including the Gemini API call, JSON extraction, and the dual delivery of the downloadable '.zip' package and the inspectable code via the HTTP header.

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 1, January 2026

DOI: 10.17148/IJARCCE.2026.15145

B. Single-Pass Structured Data Generation

Following the completion of the thought process, the LLM is instructed to produce the final output as a single, valid JSON object encapsulated within "json-output" tags to facilitate robust parsing. The structure of the JSON object maps filenames as keys to string literals containing the full source code for that file. This single-pass generation method is highly efficient, as it consolidates the creation of an entire multi-file project into a single API transaction, reducing latency and ensuring inter-file consistency.

C. Backend Processing and Artifact Delivery

The complete response from the LLM is transmitted to a FastAPI server for processing and delivery. The server first parses the LLM output to extract the JSON object from within the designated tags. Subsequently, it employs a parallel delivery mechanism: the parsed code data is sent to the React client via a custom HTTP header to populate an interface for code review, while concurrently, the server packages the files into a '.zip' archive which is sent as the primary response body. This dual-modality delivery provides the user with both immediate inspectibility and a downloadable artifact from a single operation.

V. IMPLEMENTATION DETAILS

The implementation of the system, Prompt2Extension, is a high-performance web application with a clear client-server architecture. The technology stack was selected for a responsive user experience, asynchronous AI communication, and logical separation between the frontend and backend.

A. Development Environment and Tools

The deployment takes advantage of a set of modern and powerful technologies each selected for its specific contribution to the system's performance and architecture.

1. Backend API: The server is developed in Python 3.11 and FastAPI. FastAPI was selected for its high-performance asynchronous capabilities, which are essential for efficiently managing I/O-bound operations like API requests to the generative model without blocking the main thread.
2. Frontend Application: The frontend is a single-page application built with React 18. This framework was chosen for its component-based architecture and efficient state management, which is ideal for creating a responsive user interface that must handle dynamic states such as loading, user input, and the final code display.
3. Generative AI Core*:* The core of the system is Google's Gemini 1.5 Pro, accessed via the official google-generative ai Python library. This model was selected for its large context window and advanced reasoning capabilities, which are critical for interpreting the nuances of user prompts and generating complex, coherent, multi-file code structures in a single pass.
4. Packaging*:* Python's standard zip-file library is used to package the generated files into a downloadable archive, providing a simple and robust method for artifact delivery.

B. Modular Code Structure

The project is structured into two decoupled sections: a 'frontend/' directory for the React application and a root directory for the Python backend. The frontend's state is managed by the main 'App.jsx' component, handling user prompts, loading states, and the final code display. The backend logic is consolidated into a single 'app.py' file, which initializes the FastAPI server, houses the master 'SYSTEM-PROMPT', communicates with the Gemini API, and handles the final '.zip' packaging and HTTP response delivery.

VI. EXPERIMENTAL SETUP AND RESULTS

A. Experimental Setup

In order to test the performance of the system, we utilized a special dataset of 5 test prompts that represent standard extension use cases of different difficulty.

Task	Complexity	Core Technology	Prompt Description
T1	Very Easy	CSS	Make all websites dark grey.
T2	Simple	Content Script	An extension that reads the selected text aloud.
T3	Medium	Popup UI	A button that copies the current page's URL to the clipboard.
T4	Complex	Background Script	Add a right-click menu option to search selected text on Wikipedia.
T5	Complex	Popup + Storage	Create a simple notepad in the popup that saves notes.

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 1, January 2026

DOI: 10.17148/IJARCCE.2026.15145

Metrics for evaluation included Functional Correctness (Success Rate), where a test passed if the '.zip' file was installable and functional; Architectural Correctness, assessing if the LLM chose the correct file structure; and Efficiency, measured as the time from submission to download

B. Results

With the final Chain-of-Thought prompt, the system scored a 100% success rate (5 out of 5) on the test prompts. On all prompts, the LLM was able to analyze the requirements correctly using its '(thinking)' block, resulting in the selection of the correct architecture. Examples include its ability to correctly determine the presence of a background script and the 'contextMenus' permission for the Wikipedia task, and its ability to correctly use a simpler popup-only architecture for the URL-copying task. The mean generation time was 8-15 seconds.

VII. DISCUSSION

The 100% success rate achieved across our benchmark tasks provides strong validation for our prompt-centric architectural philosophy. The system's ability to correctly distinguish between a content-script-only task (T2) and a more complex task requiring a background script and contextMenus permission (T4) is particularly telling. This demonstrates that the mandated CoT planning phase [3] effectively forces the LLM to reason about the appropriate architecture, a key challenge that differentiates our work from the action-oriented web agents previously discussed [1][6]. While naive prompts often failed by placing all logic in a single popup script, the structured reasoning process was the critical factor in achieving architectural correctness and, consequently, functional success. This finding suggests that for complex, structured code generation tasks, the intellectual investment in prompt engineering can yield a higher return on reliability than fine-tuning a model on a narrow dataset.

VIII. FUTURE SCOPE

Although the existing system has a good basis, there are various innovative ways in which research and development can be done in the future to broaden the potential of the system and solve the existing constraints. Our future work areas have been identified as the following: 1. The Interactive Refinement Loop: The present system is a one-shot generator. The biggest improvement would be to introduce a human-in-the-loop feedback, conversational mechanism. This would enable users to iteratively improve the generated code with corrective feedback (e.g., Change the highlight color to blue). This feedback would be interpreted by the system, and only the relevant parts of the code would be recreated, so that the process of one command would become a conversational ones. 2. Increasing Complex Task Decomposition: We intend to come up with an advanced planning module to increase the rate of success of complex tasks. Rather than compute all of the code directly, the system might first break down a more com plex prompt into a multi-step plan expressed in a high-level language (e.g., "1. Create the popup UI. 2. Write script to access data. 3. Display data in the popup."). The LLM would then produce the code of each step in series, a process theory that has its roots in chain-of-thought and agent-based planning. 3. Adding Multimodal Prompting: One of the future directions is to move beyond text-only prompts on the one hand, and add a Vision Language Model (VLM) on the other. This would allow the users to provide a visual context. As an illustration, a user may add a screenshot of a webpage, outline an item, and add a cue such as "Hide this part". The VLM would examine the image to find the appropriate element and its CSS element so that the user experience would be more user-friendly and powerful. 4. Automated Code Validation and Testing: To enhance the trustworthiness of the generated extensions, we recommend that an automated testing module be implemented. This would then invoke a headless browser instance, install the extension and execute a simple functional test to identify errors post code generation. When a test is failed, the system might perform a self-correcting step and restart the process of feeding the error message back into the LLM in order to produce a modified version of the code.

IX. CONSLUSION

In conclusion, Prompt2Extension demonstrates a significant step forward in prompt driven software engineering. By successfully translating abstract natural language prompts into complete, installable browser extensions, this work validates a method ology that leverages a single, sophisticated Chain-of-Thought prompt to orchestrate the entire development process. Our approach effectively bridges the gap between the ephemeral actions of web automation agents and the generation of durable, user-owned software, addressing the unique challenges of code structure, packaging, and platform compliance. The impact of this research extends beyond the domain of browser extensions. It serves as a powerful proof-of-concept for a new paradigm of software creation where the barrier to entry is not technical proficiency but clarity of intent. This democratization of development empowers educators, researchers, and domain experts to build their own custom tools, transforming software from a specialized discipline into a universally accessible medium for problem-solving. Prompt2Extension is therefore not merely a tool, but a compelling vision of a future where the creation of personalized software is as intuitive as describing an idea.

International Journal of Advanced Research in Computer and Communication Engineering
Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 1, January 2026
DOI: 10.17148/IJARCCE.2026.15145

REFERENCES

[1]. Jiang X, Qian J, Wang T, Zhang K, Zhang Y, Wang Y, Chen Y, Tang J (2024) A Survey on Code Generation with LLM-based Agents. arXiv preprint arXiv:2402.13217
[2]. Xu K, Mao Y, Guan X, Feng Z (2024) Web-Bench: A LLM Code Benchmark Based on Web Standards and Frameworks. arXiv preprint arXiv:2405.02116
[3]. Sahoo P, Singh AK, Saha S, Jain V, Mondal S, Chadha A (2024) A Systematic Survey of Prompt Engineering in Large Language Models: Techniques and Applications. arXiv preprint arXiv:2402.07927
[4]. Li W, Wang X, Li W, Jin B (2024) A Survey of Automatic Prompt Engineering: An Optimization Perspective. arXiv preprint arXiv:2403.14733
[5]. Yang K, Liu Y, Chaudhary S, Fakoor R, Chaudhari P, Karypis G, Rangwala H (2024) AGENTOCCAM: A SIMPLE YET STRONG BASELINE FOR LLM BASED WEB AGENTS. arXiv preprint arXiv:2406.01464
[6]. Tang B, Shin KG (2024) STEWARD: Natural Language Web Automation. arXiv preprint arXiv:2402.15946
[7]. Lai H, Liu X, Iong IL, Yao S, Chen Y, Shen P, Yu H, Zhang H, Zhang X, Dong Y, Tang J (2024) AutoWebGLM: A Large Language Model-based Web Navigating Agent. arXiv preprint arXiv:2405.15286
[8]. Yehudai A, Eden L, Li A, Uziel G, Zhao Y, Bar-Haim R, Cohan A, Shmueli Scheuer M (2024) Survey on Evaluation of LLM-based Agents. arXiv preprint arXiv:2405.00824
[9]. Jin H, Chen H, Lu Q, Zhu L (2024) Towards Advancing Code Generation with Large Language Models: A Research Roadmap. arXiv preprint arXiv:2401.07021
[10]. DeepEval Contributors (2024) DeepEval: An Open-Source Framework for Evaluating LLM-Generated Code. GitHub Repository. https://github.com/confident ai/DeepEval
[11]. Mzwri K, Turcs´anyi-Szabo M (2025) The Impact of Prompt Engineering and a Generative AI-Driven Tool on Autonomous Learning: A Case Study. Education Sciences 15(2):199
[12]. Hill PA, Narine LK, Miller AL (2024) Prompt Engineering Principles for Generative AI Use in Extension. The Journal of Extension 62(3):20
[13]. Mazumder S, Riva O (2021) FLIN: A Flexible Natural Language Interface for Web Navigation. In: Proceedings of the 2021 Conference of the North Ameri can Chapter of the Association for Computational Linguistics: Human Language Technologies, pp 3132–3138
[14]. Lin XV, Wang C, Pang D, Vu K, Zettlemoyer L, Ernst MD (2017) Program Synthesis from Natural Language Using Recurrent Neural Networks. University of Washington
[15]. Kim D, Patel P, Bernstein MS (2024) Modeling User Behavior and Costs in AI-Assisted Programming. arXiv preprint arXiv:2402.13840

