DIDUP: Dynamic Iterative Development for UI Prototyping

Jenny Ma*, Karthik Sreedhar*, Vivian Liu, Sitong Wang, Pedro Alejandro Perez, Lydia B. Chilton

jenny.ma@columbia.edu, ks4190@columbia.edu, chilton@cs.columbia.edu

Columbia University

Abstract

Large language models (LLMs) are remarkably good at writing code. A particularly valuable case of human-LLM collaboration is code-based UI prototyping, a method for creating interactive prototypes that allows users to view and fully engage with a user interface. We conduct a formative study of GPT Pilot, a leading LLM-generated code-prototyping system, and find that its inflexibility towards change once development has started leads to weaknesses in failure prevention and dynamic planning; it closely resembles the linear workflow of the waterfall model. We introduce DIDUP, a system for code-based UI prototyping that follows an iterative spiral model, which takes changes and iterations that come up during the development process into account. We propose three novel mechanisms for LLM-generated code-prototyping systems: (1) adaptive planning, where plans should be dynamic and reflect changes during implementation, (2) code injection, where the system should write a minimal amount of code and inject it instead of rewriting code so users have a better mental model of the code evolution, and (3) lightweight state management, a simplified version of source control so users can quickly revert to different working states. Together, this enables users to rapidly develop and iterate on prototypes.

1 Introduction

Large language models (LLMs) are remarkably good at writing code as evidenced by numerous tools including Devin, CoPilot, Gemini, and GPT Pilot [CognitionAI 2024; GitHub 2024; Google 2024; Pythagora 2024]. Programming is a complex task that requires both high-level, system design planning and low-level understanding of how features can be implemented. Prior work has found that programmers tend to be caught up in the implementation details, debugging rather than thinking at the high-level [Myers and Rosson 1992]. These existing systems suggest that LLMs can greatly enhance developer productivity and effectiveness.

Author's Contact Information: Jenny Ma*, Karthik Sreedhar*, Vivian Liu, Sitong Wang, Pedro Alejandro Perez, Lydia B. Chiltonjenny.ma@columbia.edu, ks4190@columbia.edu, chilton@cs.columbia.eduColumbia University.

A particularly valuable case of human-LLM collaboration is creating code-based user interface (UI) prototypes – prototypes that enable users to view and fully interact with a UI. Code-based UI prototypes are valuable for testing data-driven applications that require an interactive component. It is not a task that can be fully automated, but rather is an iterative and exploratory process where users adapt their designs as they incrementally build and test their prototypes. Automatic code generation can be valuable to alleviate tedious aspects of the development process, but the user must remain in control in order to guide the system.

In software development, there are two main paradigms of writing code: (1) the classic waterfall model, and (2) the iterative spiral model. The waterfall model is a linear workflow where each phase of development must be completed before the next phase begins, and stages are traversed sequentially [Royce 2021]. It is simple to realize in practice, but limited because of its inflexibility towards changes based on emerging requirements. Conversely, the spiral model is an iterative approach to software development where small features and ideas are continuously designed, implemented and evaluated as they are discovered [Boehm 1988]. Prototyping is fundamentally an iterative process, and thus any tool for prototyping should adopt a spiral development process.

2 Background and Approach

GPT-Pilot is the leading open-source code prototyping tool powered by LLMs; of the tools we tried [CognitionAI 2024; Pythagora 2024], it produced the most working prototypes. GPT Pilot has a 4-step workflow: First, the user inputs a project goal, and the system asks questions to formalize the specification. Second, GPT Pilot creates an implementation plan, breaking down the project into subtasks. The user is not consulted on this. Third is the execution phase. For each task, GPT Pilot requires human approval before progressing to the next task; if there are bugs it will attempt to debug it. Fourth, after development, there is a documentation phase. While GPT Pilot does have some human input, it is essentially a waterfall model; it lacks sufficient interaction to recover from failures. It also lacks a continuous mechanism that allows the user to elaborate on the prototype and continue improving it.

We propose three novel mechanisms that can guide the architecture for code-based UI prototyping systems that follow an iterative spiral model: (1) **Adaptive Planning**: systems must support continual updates in designs and plans based on feedback and implementation. This allows for flexibility in development and for users to elaborate on initial project plans. (2) **Code Injection**: when executing sequential tasks, LLMs often rewrite prior working code, which is confusing for users because they don't know what was removed or added. By injecting the minimal amount of code necessary to a target location, code injection allows for safe code modifications, providing users with a clear mental model of the code's evolution. (3) **Lightweight State Management**: errors inevitably occur, and

Jenny Ma*, Karthik Sreedhar*, Vivian Liu, Sitong Wang, Pedro Alejandro Perez, Lydia B. Chilton

in situations where the machine cannot fix them, it provides a simplified version of source control. Users can quickly revert to different working states and rapidly prototype explorations.

3 System

We introduce DIDUP, a web application to assist code-based UI prototypes. It employs a dynamic, iterative approach to development that allows for human direction at every step. DIDUP is a Flask application that runs on the web, featuring a Python backend and a Typescript frontend. It is designed for users familiar with front-end development who want to prototype faster and helps users author web applications in HTML, CSS, and Javascript.

We walk through the system with a motivating example. The user first inputs the prototype goal: creating a UI that visualizes Yelp restaurants as a card-swiping UI (Tinder) to help users choose restaurants to eat at. The system outputs a detailed specification of the prototype idea and generates synthetic data to populate the interface. The user can regenerate, approve, or modify the outputs.

After the user approves the project specification, DIDUP moves on to the planning stage. DIDUP breaks down the project into tasks, where each task is the next-smallest testable iteration of the previous task. In our example, the task list is broken down to: (1) Create the basic HTML structure, (2) Implement swiping functionality, (3) Build the bookmark display, and (4) Handle the unbookmark click event. The user can regenerate, modify, or approve this task list.

The system executes tasks sequentially. In our example, the system implements task 1) Create the HTML structure. The system generates the initial code, and the UI is rendered on DIDUP for the user to interact with. If there are failures, the user can debug by iteratively prompting the system in the right direction to regenerate code, redoing the task, or debugging the code by hand.

Once the user confirms that it works, they can proceed to task 2) Implement swiping functionality. The user does not modify the task and prompts the system to generate the code. Instead of rewriting the existing code to implement the task, the system performs **code injection** – it writes the minimal amount necessary to complete the task and injects it. DIDUP writes 2 code snippets: 1) javascript functionality for swiping cards and 2) event listeners for swipes, and injects it at the appropriate lines in the file. Code injection minimizes the risk of global errors and the inadvertent deletion of good code, ensuring the task remains modularized.

Once the user tests and approves task 2, they can proceed with tasks 3 and 4, to allow users to create, see, and undo bookmarks, in similar fashion. DIDUP thus finishes the original plan.

After using the app with the bookmark feature, the user gets an idea for a new feature - the ability to search within their bookmarks. DIDUP allows the user to add this task as task 5 as a part of **adaptive planning**, the ability to add, update, or remove tasks continuously during project development. To do this, the user adds a fifth step: 5) Implement search in the bookmarks tab, and the system generates code for this new feature.

While testing their prototype, the user realizes another feature they want to add - they want to mark a bookmarked restaurant as "visited." In the future, when they're looking for a new restaurant to try, they don't look at the restaurants they've already been to. DIDUP adds a field to the datastore of bookmarked restaurants

to have a binary variable for "visited," and generates code so that the UI has a button on every restaurant "card" in the bookmarked section that the user can mark as visited.

The user tests the prototype and sees the feature is fully functioning, but realizes that they've made a mistake - they want to be able to mark *any restaurant as "visited"*, not just the bookmarked ones. Making this change requires modifying the underlying data representation to take "visited" out of bookmarks and onto all restaurants. Thus, it seems safest to do a rollback to the version of the code at task 5 (with bookmark's implemented, but no favorites).

To support rollback, our system introduces **lightweight state management**. DIDUP saves the state of the UI code at every task to allow users to quickly return to a previous state without remnants of previous changes. The user rolls back to step 5. After the rollback, the user can use adapting planning to execute the new task of allowing users to mark any restaurant as visited, and create a new card list of all visited restaurants. The user can now remember all their past restaurant visits, not just the ones that were bookmarked.

This process allowed the user to prototype a working UI and add features as they tested prototypes. By incorporating adaptive planning, code injection, and lightweight state management, DIDUP allows users to rapidly iterate on code-based UI prototypes. It streamlines the typically tedious process of switching between writing code, designing features, managing states, and frequent testing.

4 Evaluation

We conducted a small user study to evaluate DIDUP's ability to build prototypes compared to GPT Pilot. We tasked two users with creating specific UIs, using both systems. In both cases, GPT Pilot became fixated on using MongoDB to implement a database for the prototype, which the user was not familiar with and did not want to use. Repeated prompting from the user to not use MongoDB did not work. In one case, the user was able to successfully create their application only after restarting the entire process and specifying that MongoDB should not be used in the initial project description input. The other case was ultimately unsuccessful because GPT Pilot entered an infinite loop during the debugging process, and there was no way to roll back to a previous working state.

When using DIDUP, both users created working UI prototypes easily. Both easily added new features and backtracked to make major adjustments. Furthermore, the prototypes they created were more complete. DIDUP-generated UIs had significantly more grouping and stylistic elements than GPT Pilot-generated UIs.

5 Future Work

In the future, we plan to increase the number of participants and formalize the evaluation to more accurately understand how effectively DIDUP's development process can support users in code-based UI prototyping. Currently, our system is limited to front-end code generation in javascript, CSS, and HTML; we will expand this to full-stack development. Additionally, we can improve upon adaptive planning mechanisms to allow full regenerations of plans if design directions change, or allow for additions of complex features that add plans, or groups of multiple tasks to the initial design. We can also employ a tree-structure for lightweight version control so users can compare existing prototypes against each other.

References

B. W. Boehm. 1988. A spiral model of software development and enhancement. *Computer* 21, 5 (1988), 61–72. https://doi.org/10.1109/2.59

CognitionAI. 2024. Devin: The LLM Programmer. https://github.com/OpenDevin/OpenDevin.

GitHub. 2024. GitHub Copilot: Your AI pair programmer. https://github.com/features/copilot.

Google. 2024. Gemini: A Large Language Model. https://github.com/kyegomez/Gemini.

Brad A. Myers and Mary Beth Rosson. 1992. Survey on user interface programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Monterey, California, USA) *(CHI '92).* Association for Computing Machinery, New York, NY, USA, 195–202. https://doi.org/10.1145/142750.142789

Pythagora. 2024. GPT Pilot: An Advanced AI System. https://github.com/Pythagora-io/gpt-pilot.

Winston Royce. 2021. Managing the development of large software systems: concepts and techniques. *Ideas That Created the Future* (2021). https://api.semanticscholar.org/CorpusID:110643636