InstructPipe: Building Visual Programming Pipelines with Human Instructions

ZHONGYI ZHOU, JING JIN, VRUSHANK PHADNIS, XIUXIU YUAN, JUN JIANG, XUN QIAN, JINGTAO ZHOU, YIYI HUANG, ZHENG XU, YINDA ZHANG, KRISTEN WRIGHT, JASON MAYES, MARK SHERWOOD, JOHNNY LEE, ALEX OLWAL, DAVID KIM, RAM IYENGAR, NA LI, RUOFEI DU*, Google Research, USA

Fig. 1. Traditional visual programming requires users to select nodes, conceive pipeline structure, and then create the pipeline with a node-graph editor. In contrast, InstructPipe streamlines this process by instantiating a multi-modal machine learning pipeline directly from human instructions, enabling users to further iterate and interact with the pipeline across diverse applications such as news summarization, image style transfer, and real-time AR effects.

Visual programming offers coding-free experiences to build diverse machine learning (ML) pipelines. However, beginners often struggle with setting up and linking the appropriate nodes entirely from scratch. To address this, we introduce InstructPipe, a visual programming system to start prototyping with a text instruction. InstructPipe features a comprehensive set of 65 nodes to build multi-modal ML pipelines, informed by 236 proposals from 58 contributors. Our system comprises a node selector, a code writer, and a code interpreter, seamlessly integrated into a node-graph editor for human-AI collaboration. Technical evaluations reveal that InstructPipe reduces user interactions by 81.1% compared to traditional methods, based on 48 community-contributed pipelines. Our user study (N=16) showed that InstructPipe empowers novice users to streamline their workflow in creating desired ML pipelines, reduce their learning curve, and spark innovative ideas with open-ended commands.


Additional Key Words and Phrases: Visual Programming; Large Language Models; Visual Prototyping; Node-graph Editor; Graph Compiler; Low-code Development; Deep Neural Networks; Deep Learning; Visual Analytics
1 INTRODUCTION

A visual programming interface provides users with a node-graph editor to program. As opposed to writing code in a code editor, the node-graph allows users to build a pipeline in a visual workspace with nodes and edges. This approach amplifies users’ creativity to prototype diverse applications without requiring to understand programming languages and algorithms. Recent advances in machine learning (ML) further stimulated growing interest in visual programming. Open-sourced ML libraries (e.g., TensorFlow [1], PyTorch [41], and Hugging Face [59]) provide users with various encapsulated modules to accelerate AI project development and experimentation, where researchers and practitioners can easily incorporate “off-the-shelf” models. Recent foundational models, e.g., large language models (LLMs) [3, 6, 54], and findings on Chain-of-Thought [58] further stimulate a community-wise interest in visual programming [60] that allows users to interactively explore the chains.

However, this streamlined approach has its limitations. The utility of a visual programming platform relies heavily on its predefined nodes — users’ creative process will be disrupted even with one missing node. Even worse, compared to various built-in functions in the programming language, visual programming typically provides a small number of pre-defined nodes (e.g., 38 nodes in Visual Blocks [16]). The community addresses this issue in a reactive development cycle by collecting feature requests from users, and then assigning tasks to developers. We believe that the lack of community guidelines makes it challenging to address this issue in a proactive manner. To this end, we raise the following research question:

RQ1. What is the design space of primitive nodes for vision-language pipelines? How does its corresponding implementation impact users’ creative processes?

An answer to RQ1 can effectively guide proactive node implementation that facilitates users’ creation of multimodal ML pipelines. As developers continue to add nodes for enhanced system utility, it will eventually clutter the user interface, causing a high cognitive burden. Even worse, users typically build a pipeline “from scratch”, i.e., selecting nodes, ideating the pipeline structure and finally connecting nodes from a completely empty workspace. A large pool of nodes can easily overwhelm users in this creative process, causing unsatisfactory user experience (i.e., the usability issue). Similar issues also exist when users write code using programming languages (with many built-in functions and libraries), but recent advances in LLMs show that such challenges can be addressed. For example, GitHub Copilot [17] makes it possible to generate code by simply describing users’ requirement in natural language. Therefore, we hypothesized that similar AI assistants can also benefit visual programming interfaces by reducing users’ workload in their node editing experiences. We further elaborate the second research question in the project as follows:

RQ2. How can an LLM-assistant support users to build a pipeline beyond “from scratch”? How does this new experience influence future system design in visual programming?

In this paper, we present our answers to both research questions with a system based on Visual Blocks [16]. To explore RQ1, we gathered 236 ML pipeline proposals from 58 contributors, and distilled a design space that guided our implementation space of 27 new nodes in the system. We further implemented InstructPipe, the core function of
our system allowing users to generate a pipeline through natural language instruction (Figure 1). This is achieved by decomposing the generation process into three steps. The system first scopes the potentially useful nodes, and then generates pseudocode for a target pipeline. InstructPipe finally compiles the pseudocode and renders the pipeline to facilitate further user interaction. We organized two open-ended workshops and found that 95.3% of the pipelines 23 participants created contained at least one new node we implemented. Our technical evaluation suggests that InstructPipe reduces user interactions by 81.1% compared to building pipelines from scratch. Our system evaluation with 16 participants provide statistically significant results for how InstructPipe reduced users’ workload and increased users’ satisfaction in accomplishing the goals.

In summary, this work makes the following contributions:

1. A new visual programming system for prototyping multimodal ML pipelines with a comprehensive node library, and InstructPipe, which allows users to build pipelines from human instructions,
2. A design space of primitive nodes for prototyping multimodal ML pipelines, derived from 236 proposals by 58 participants and the corresponding implementation of 27 new nodes,
3. The implementation of InstructPipe, which leverages a node selector, a code writer, and a code interpreter to compile interactive pipeline directly from text instructions within a node-graph editor,
4. 64 annotated pipelines contributed by 23 participants and a technical evaluation of InstructPipe,
5. A system evaluation of InstructPipe with 16 participants, highlighting its potential in enhancing human-AI collaboration for visual programming platforms.

2 RELATED WORK

2.1 Visual Programming

The operation of computer systems is defined by a computer program. However, “the program given to a computer for solving a problem need not be in a written format” [52]. This future-looking statement, dating back to 1960s, inspired several generations of researchers to design and build visual programming systems.

Today, visual programming systems (e.g., LabView [28], Unity Graph Editor [53], PromptChainer [60], and Visual Blocks [16]) typically feature a node graph editor, providing users with a visual workspace to “write” their program. Recent works further explored the application of visual programming in education [25, 29], authoring support [65, 66], and robotics [13, 21, 22]. Zhang et al. [66] connected the visual programming tool to the concept of teaching by demonstration [69], allowing users to rapidly customize AR affects in video creation. FlowMatic [65] extended traditional visual programming interfaces into 3D virtual environments, providing users with immersive authoring experiences.

More recently, findings on LLM Chains [62] and Chain-of-Thought [58] further stimulated researchers to build visual programming tools that chain LLM modules. Developers want to explore various ways to chain an LLM module for various application, and in such scenario, visual programming provides a great platform for users to focus on the high-level exploration. Example research work and industrial products include PromptChainer [60], LangFlow [31], and ComfyUI [12].

InstructPipe differs from related work in two aspects. First, InstructPipe extends PromptChainer [60, 62] and Visual Blocks [16], which individually focus on language and multimedia nodes. We provided users with LLMs, multimodal models, search, and low-level processing nodes for more versatile creation. As we will show in Figure 8, new nodes can effectively boost users’ creative processes. Second, InstructPipe further allows users to generate a pipeline using text-based instruction, providing users with new experience beyond building a pipeline from scratch.
Early multimodal ML work uses language models to solve Visual Question Answering (VQA) [2, 4], but these solutions are limited to simple questions and cannot perform effective reasoning and problem solving [35]. LLMs revolutionized AI’s reasoning capability [58, 68] in language, which motivated researchers to build LLM applications in various domains beyond NLP [39, 47, 51]. For instance, LLMs augmented the perceptual and planning intelligence in robotics [47], supported autonomous driving [32] and assisted clinical processes in medical science [30, 48].

The advances in LLMs empower recent interactive systems [40, 42] with enhanced machine intelligence. Recent research leverages LLMs to edit visualization [46, 57], receive AI explanation [61], facilitate communications [34], understand user interface [56], and study simulated social behaviors [39]. The revolution also motivated HCI researchers to design new interfaces for LLMs [26, 50]. Graphologue [26] augmented LLM response by displaying interactive diagrams on the side of the response text, which visualizes the semantic logics in a paragraph. Sensecape [50] provides users with a workspace to explore long LLM response in a hierarchical structure.

InstructPipe focuses on utilizing LLMs for generating pipelines in visual programming with human instructions. This work shares similar vision with Prompt2Model [55] and VisProg [18]. Prompt2Model [55] finetunes a BERT model [14] using data generated by instructions. VisProg [18] produces Python code from instructions with task-dependent few-shot prompting. However, both prior arts lack an interactive workspace that facilities novices to use. In contrast, InstructPipe generates and compiles a pipeline (without fine-tuning), while rendering the pipeline in a visual programming interface, facilitating an interactive, collaborative, and explainable workflow.

3 DESIGN SPACE

To understand what pipelines people would like to build with visual programming, we conducted an online survey to gather proposals of multimodal ML pipeline. These results informed a new design space of primitive nodes for visual programming with language and vision models.

3.1 Online Survey

We distribute an online survey (Appendix D) through internal communication channels, email lists, and social media. In two weeks, we collected 236 pipeline proposals from 58 respondents, of whom 32 (55%) had no prior experience with Visual Blocks. Using the affinity diagram approach, two researchers organized participants’ responses and classified the proposals into three categories: language, vision, and multi-modal pipelines. We list selected proposals in Appendix C to inspire future researchers with innovative ideas.

3.2 Observations

We followed the design space analysis methods [7] and held iterative discussion sessions. Through this process, we discerned that the existing 38 nodes in Visual Blocks were insufficient for satisfying the diverse requirements to meet participants’ need. We then categorized the missing nodes into the following three classes:

- **input nodes** (e.g., text input, Google Sheets reader)
- **output nodes** (e.g., markdown viewer, Colab output)
- **processor nodes** (e.g., large language models, vision-language models)
Input and output nodes serve as protocols that extend the creative usage of a pipeline. Processor nodes decide the intelligence of data processing. For example, with a Google Sheets reader node, a Large language model node, and a Colab output node, one can create a pipeline of "Visualize the user preferences data from a Google Sheet in a bar chart".

3.3 Node Design Matrix
Informed by the set of nodes and the classification derived from the online survey, we crafted a design space specifically focused on processor nodes, aiming to facilitate the prototyping of language and vision pipelines. As shown in Figure 2, we categorize intermediate nodes based on their input/output types including language, vision, and metadata. "Metadata", defined as the data of the data, represents intermediate features commonly used in machine learning pipelines, such as text vectors and image landmarks.

4 INSTRUCTPIPE
InstructPipe is a visual programming system that evolves from Visual Blocks in two ways. Drawing from our design space, we first add a comprehensive set of nodes with enhanced intelligence to support creativity. While the increase in nodes amplifies the system’s capabilities, it also adds complexity, which negatively affects user experience according to Hick’s Law [43]. To counteract this challenge, we further introduce a dialog UI for crafting pipelines with text-based instructions. Together, we envision InstructPipe can accelerate on-boarding and amplify creative expression.

4.1 Enhanced Node Library for Multimodal ML Pipelines
Informed by the design space, we implement new nodes that enhance the intelligence of visual programming. Figure 2(b) summarizes our node implementation. As listed in Appendix A, we integrate 27 new nodes into Visual Blocks to satisfy people’s proposed pipelines. We elaborate six key nodes below:

- **PaLM** [11]: a large language model (LLM) that generates text given a text prompt.
- **Google Text Search**: given a text query, output a list of URLs returned from Google.
- **Google Image Search**: given a text query, output the first image from Google.
- **PaLI** [8, 9]: a Vision-Language Model that generates text from a joint of image and text inputs.
- **Imagen** [45]: a text-to-image diffusion model.
- **OCR** [49]: a text recognition model from an image query.
4.2 Instruction-driven Pipeline Generation with Further Human Interaction

InstructPipe enables users to generate a pipeline by simply providing text-based instructions. Different from the node implementation space that aims for an enhanced utility, the goal here is to simplify the user interaction for better usability. To generate a pipeline, users can first click on the “InstructPipe” button on the top-right corner of the interface (Figure 3b). The system then activates a simple dialog (Figure 3a) in which users can 1) provide a description of their target pipeline and 2) tag the pipeline. The tag can be “language”, “visual”, or “multimodal”. After users click “Submit” at the bottom-right corner of the dialog, InstructPipe renders a visual programming pipeline with the node-graph editor. Based on the result, users can further refine the pipeline with the visual programming interface.

5 IMPLEMENTATION

Applying LLMs to generate pipelines in a visual programming system with a large number of total nodes poses non-trivial challenges. There exist two prevailing approaches for customizing LLMs: 1) fine-tuning LLMs [5, 34], and 2) few-shot prompting [15, 39]. Fine-tuning LLMs requires a substantial volume of annotated data, comprising pairs of pipelines and prompts. Additionally, with a growing list of nodes, it poses new challenges to scale this approach with new annotations. Few-shot prompting can seem more practical [18, 58, 63], but it is challenging to design efficient prompt that fits within the token limits. If we provide a full list of 65 nodes, and their corresponding input/output descriptions, the raw text of the node configuration alone can have 19k tokens. Moreover, because of the combinatorial explosion of the large number of nodes in the system, it is not clear how many prompt examples are needed and how we can construct these pipeline prompt examples.
Fig. 4. Workflow of InstructPipe. First, users describe their desired pipeline in natural language and designate it with a language, image, or multi-modal tag. InstructPipe then feeds user instructions into a node selector to identify a relevant set of nodes. Subsequently, both the instructions and the relevant nodes with their description are input into a code writer to produce pseudocode. Finally, a code interpreter parses the pseudocode, rectifies errors, and compiles a JSON-formatted pipeline, allowing users to further refine and interact with it within Visual Blocks’s node-graph editor.

To this end, we implement InstructPipe by utilizing a two-stage LLM refinement prompting approach, followed by a pseudocode interpretation step to render a pipeline. Figure 4 illustrates the high-level workflow of InstructPipe implementation. InstructPipe includes two LLM modules (highlighted in red): 1) a Node Selector (§5.2) and 2) a Code Writer (§5.3). Given a user instruction and a pipeline tag, we first devise the Node Selector to identify a list of potential nodes that would be used according to the user’s description. In the Node Selector, we prompt the LLM with a very brief description of each node, aiming to filter out unrelated nodes for the target pipeline. The selected nodes and the original user input (the prompt and the tag) are then fed into the Code Writer and generate pseudocodes that represent a pipeline. In Code Writer, we provide LLM with detailed description and examples of each selected nodes to ensure LLMs have a thorough understanding of each candidate. Finally, we employ a Code Interpreter to parse the pseudocode and render a visual programming pipeline for the user to interact with.

5.1 Pipeline Representation

The Visual Blocks system represents a pipeline as a Directed Acyclic Graph (DAG) in JSON format. To compress the verbose JSON file, InstructPipe represents pipelines as pseudocode [18, 51], which can be further compiled into a JSON-formatted pipeline. Pseudocode representation is highly token-efficient (e.g., as shown in Figure 5, it...

---

Fig. 5. A pair example of pipeline and pseudocode. In the first line of code under "processor", pali_1_out, pali_1, pali and image=input_image_1, prompt=input_text_1 represents output variable id, node id, node type, and node arguments, respectively.

---

1Pipeline in JSON files: see supplementary materials for examples.
You are an assistant tasked with aiding the user in constructing an AI pipeline.

For this assignment, select a small set of nodes to fulfill the user's pipeline request.

Guidelines:
1. In your selection, include at least one node from each category: "input", "processor", and "output".
2. Ensure you incorporate all necessary nodes. Opting for a few additional nodes, if required, is acceptable.
3. Limit your selection to a maximum of 10 nodes.

Below are the nodes you may select from:

<table>
<thead>
<tr>
<th>Node Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong></td>
</tr>
<tr>
<td>live_camera: Capture video stream through your device camera.</td>
</tr>
<tr>
<td>google_search: Use Google to search web that returns a list of URLs based on a given keyword; usually selected with string picker.</td>
</tr>
</tbody>
</table>

Examples:

Q:  

A:  

---

Fig. 6. The prompt structure for the Node Selection module. Each node description is formatted as `{node types}: {short descriptions of the nodes}; [recommended node(s)]`. The node recommendation is optional.

compresses the pipeline from 2.8k tokens to 123 tokens). The efficiency does come with a cost: it loses some fine-grained annotations on each node like property values (e.g., layout of the nodes, parameters of a segmentation model, and the degree of blurring parameters in an "image processor" node). This implies that InstructPipe leaves the task on fine-tuning these parameters to the user.

Figure 5 provides an example of a pipeline (Figure 5a) and its corresponding pseudocode (Figure 5b). The syntax design is inspired by TypeScript. The structure is inspired by how academic papers present pseudocode [67]: an algorithm block typically starts with specifying the input/output and then explains the intermediate processor. We highlighted the first line under the processor module (i.e., the operation of the PaLI node) in Figure 5b in four different colors, representing four different components in the programming language. "pali_1_out" represents the output variable name of the node. "pali_1" is the unique ID of the node. The green symbol after the colon, i.e., "pali", specifies the type of the node. In this example, the node with the ID of "pali_1" is a "pali" node. The rest part in the bracket, i.e., "image=input_image_1, prompt=input_text_1", defines the arguments of the nodes. In the input pseudocodes, we do not annotate output variable (i.e., there are no red colors in the highlighted line under the input module) because all the input nodes only export one value, and the output variable name is automatically annotated as the same symbol as the node id (i.e., "input_text_1"). Note that InstructPipe generates the text input (i.e., the property value) in the "input_text" node. Therefore, the argument in "text=\"caption this image in detail\"" does not indicate that the "input_text" node accepts input edges, but accepts the node property input.

---
5.2 Node Selector

Node Selector aims to filter unrelated nodes by providing an LLM with a short description of each node. Figure 6 shows the prompt we use in Node Selector. Followed by a general task description and guidelines, we list all the node types with a short description that explains the function of the node. Several nodes come with recommendation when the users interact with Visual Blocks, and we also include such node recommendations information in the prompt. The main intuition of this prompt design is based on how the existing open-source libraries (e.g., Numpy [19]) presents a high-level overview of all functions. These documentation typically presents a list of supported functions (in each category), followed by a short description, so that developers can quickly find their desired functions. At the end of the prompt, we provide a list of Q&A as few-shot examples to support LLM to learn and adopt to the context of the task.

5.3 Code Writer

With a pool of selected nodes, the Code Writer module is able to write pseudocodes for rendering a target pipeline. Figure 7 shows the structure of the prompt utilized in this LLM module. Similar to §5.2, the prompt starts with a general introduction and several guidelines. The major difference of the prompt design in this stage lies in the granularity of each node introduction. We provide a detailed configuration for each selected node with additional information including 1) input data types, 2) output data types, and 3) an example, represented in pseudocodes, showing how this node connects to other nodes. We put a detailed explanation of full node configuration in the supplementary materials. Similar to the previous LLM module (§5.2), the prompt design here is also inspired by the documentation of several popular code libraries. Nevertheless, we gain inspiration from low-level function-specific documentation, which typically includes a detailed description and data types in the input/output, followed by one or more examples on how developers can use this function with a few lines of codes.

The prompt also includes a list of Q&A as few-shot examples. However, providing few-shot examples in this stage is non-trivial. The reason lies in the dynamics of the node selection pool: combination of all the nodes causes a large number of possible selections, and it is impossible to design a list of few-shot examples for each possibility. Therefore, we only created an example list for each of the pipeline tags (i.e., “language”, “visual” and “multimodal”) and intended to utilize these few-shot examples to cover most of the use cases. The intuition behind this design is that the in-context examples aim to list possible creative ideas of the pipelines in each tag, which can be condensed into few-shot examples, rather than traversing all the possible combinations. This implies that the in-context examples may include nodes that are not selected in the prompts. According to our preliminary tests, such out-of-scope nodes cause negative effects on the generation results: LLMs tend to also “invent” new nodes that do not exist in our system, causing node rendering failure. Note that we also observed this issue when combining the prompts used in [18]. We eliminate this issue by using the prompt contents between the node configurations and the in-context examples (i.e., the contents start with “the following is a full list of ...” in Figure 7). This aims to inform LLMs of the exceptions for invention.

5.4 Code Interpreter

Finally, InstructPipe employs a code interpreter to parse the generated pseudocode, correct errors, and compile a JSON-formatted pipeline with automatic layout. We delineate the graph compilation and rendering procedure below:

---

2 See an example in the following link: https://numpy.org/doc/1.25/reference/routines.array-manipulation.html
3 See an example in numpy.shape: https://numpy.org/doc/1.25/reference/generated/numpy.shape.html#numpy-shape
Lexical Analysis: Using a series of regular expressions, the pseudocode is tokenized into node type, node name, and node parameters.

Semantic Analysis: Type checking is performed on each tokenized node to validate input/output connections between nodes and to identify potential cyclic graphs. If all integrity checks are met, memory allocation for the node and its corresponding edges is performed to form a node graph.

Graph Generation: With the node graph, we generate JSON-formatted code with an auto-filling process. Note that InstructPipe only fills node parameters for text input, while default values are used for other nodes (e.g., image blurring parameter).

Graph Rendering and Optimization: Finally, the generated graph is traversed using a breadth-first search algorithm, and the nodes are laid out with grid alignment in the node-graph editor within Visual Blocks.

6 DATA COLLECTION

We organized a two-day hybrid workshop using the latest iteration of Visual Blocks with two objectives: 1) to assess the efficacy of our new design and implementation space (§3), and 2) to collect data for technical evaluation of InstructPipe.

6.1 Workshop

We invited all 58 survey respondents to participate in our hybrid workshop. A total of 23 participants attended at least one day. Only 6 of the attendees had prior experience with Visual Blocks.

Participants spent two hours per day in the workshop, commencing with a 15-minute tutorial walking the participants through the nodes and the pipeline building process using Visual Blocks. After the tutorial, attendees created pipelines independently. To encourage collaboration and open dialogue, we encouraged participants to post questions, share their screenshots, and report findings through a shared online chat.

6.2 Results

Participants built 64 pipelines in two workshops, including 61 pipelines (95.3%) that use at least one node from our new implementation. This demonstrates that our node implementation (guided by our design spaces) can effectively facilitate the creative use of Visual Blocks users. Figure 8 shows three selected pipelines participants built in the workshops.

7 TECHNICAL EVALUATION

We further utilized the dataset for the technical evaluation. Different from our system evaluation (§8) that evaluates the full system by recruiting a small group of participants, the technical evaluation focuses on assessing the technical performance of our pipeline generation algorithm.

7.1 Procedure

We first conducted post-processing on the data collected from the workshop: 1) removing incomplete pipelines, 2) deleting the nodes that are isolated from the main pipelines. We also noticed a certain portion of the captions are either empty or low-quality (e.g., "newsletter", "image editing" and "[participant name]-demo"). Therefore, two authors re-wrote captions of each pipeline separately, and annotated the tag for each pipeline together.

Workshop participants consisted of 9 in-person attendees and 14 remote participants via Google Meet. Refer to Appendix §B for details.
Fig. 8. Example pipelines participants built in the workshops.

(a) Search news from Google, summarize it and then conduct fact check.

(b) Generating an emoji from a photo.

(c) Turning a tiger into a cat.
Table 1. The ratio of human interactions in the technical evaluation. Results are reported as mean ± standard deviation.

<table>
<thead>
<tr>
<th></th>
<th>Overall</th>
<th>Language</th>
<th>Visual</th>
<th>Multimodal</th>
</tr>
</thead>
<tbody>
<tr>
<td>18.9 ± 20.3%</td>
<td>17.4 ± 20.6%</td>
<td>17.6 ± 23.7%</td>
<td>20.8 ± 16.0%</td>
<td></td>
</tr>
</tbody>
</table>

We further removed pipelines that utilized out-of-scope nodes like "custom scripts" (in which the participants write codes to process the input data and return custom outputs; see Figure 8b for an example) and "TFLite model runner" (which call a TensorFlow model according to the user’s URL input of a model in the TF-Hub). As we explained in §5.1, InstructPipe focuses on generating the graphic structure of a pipeline without considering the property value within each node. Different from the majority of nodes in Visual Blocks, nodes like “custom scripts” and “TFLite model runner” derive their intrinsic functionality from the property values. Without this information, the entire pipeline’s rationale is obscured. Therefore, we consider these nodes as “out-of-scope” nodes in our technical evaluation, and we left it as future work.

The final 48 pipelines (out of 64 pipelines) are composed of 23 language pipelines, 7 visual pipelines, and 18 multi-modal pipelines. We ran our algorithm on the pipeline captions six times (three times for each caption × two captions from two authors for each pipelines) and evaluated the generation results.

7.2 Metric: Number of User Interactions

Evaluating the metric of a generated pipeline is more complex than in other tasks that have a universally recognized definition of accuracy. In our evaluation, we measure the accuracy of a generated pipeline by counting the minimal number of user interactions needed to complete the pipeline, so that it satisfies the given pipeline caption (or instructions). The number of interactions is defined as the sum of two events: 1) adding or deleting a node and 2) adding or deleting an edge between nodes. Note that when a node with edges connected is deleted, our system will auto-remove these edges. In such instances, we only register one single interaction for the node deletion.

There are two important implications in our definition of the number of user interactions. First, a complete pipeline after user interaction does not need to be the same as the corresponding pipeline in the dataset. As long as it fulfils the task described in the caption, we consider the pipeline is complete. Second, our definition does not consider interactions of modifying property values of a node, e.g., typing in a text box or selecting a value in a drop-down box. We argue that such interactions are highly node-dependent and are hard to quantify objectively. More importantly, as we explain in §5.1, the generation of property values are out of scope of this work.

Additionally, it is unfair to simply report an average number of user interactions required among different pipelines because the complexity of the pipelines in the dataset vary dramatically. Therefore, in the technical evaluation, we reported the ratio of the minimal number of user interactions required to complete a pipeline “from our generated pipeline” to that “from scratch” as our target metric.

7.3 Results

Table 1 summarizes the results of the technical evaluation. Compared to building a pipeline from scratch, InstructPipe allows the user to finish a pipeline with **18.9%** of the user interactions (as defined in §7.2), demonstrating the effectiveness of the InstructPipe support. **Seven** generated pipelines directly satisfied with instructions without any user interactions in all of the six trials, and **38** generated pipelines completed at least once in any of the six trials. This shows that the LLM is still not able to fully automate the pipeline generation process with task descriptions, and human-AI collaborations are still essential for visual programming systems using LLMs.
8 SYSTEM EVALUATION

We conducted an in-person usability study of InstructPipe with novice users. The study recruitment was in accordance with the [Anon.] ethics board, and participant consent was obtained before the study began.

8.1 Procedure

The study was designed as a within-subjects repeated measures experiment with two conditions: with InstructPipe and without InstructPipe. We will refer to these two conditions as “InstructPipe” and “Visual Blocks” in the following content. Figure 9 visualizes the complete study flow. In each condition, participants completed the two pipelines with counterbalance (referred as Task 1 and Task 2 in Figure 9). The pipelines were carefully selected to ensure a fair study as well as provide users a diverse experience of InstructPipe. Additionally, to ensure non-experts can explore the pipeline as well as successfully finish the pipelines, the pipelines should have decent level of complexity. We first selected four candidates, and the final decision was made after a pilot study with one participant for testing the level of pipeline difficulty. The resultant pipelines are composed of eight nodes with seven edges and six nodes with six edges, respectively. Using the instructions from two authors, the averaged ratio of human interactions of these two pipelines are 27.8% and 5.2%, respectively. In the study, we verbally described the pipeline to participants as below, and participants are not allowed to read our scripts:

- Text-based pipeline: get the latest news about New York using Google Search and compile a high-level summary of one of the results.
- Real-time multimodal pipeline: create a virtual sunglasses try-on experience using your web-camera.

The pipelines were curated from the hybrid workshop and were validated in a pilot study before being finalized. All participants were provided with a 10-15 minute hands-on training of both conditions. The training included all the Visual Blocks interactions needed to complete the subsequent steps of the experiment. Participants were also encouraged to try building a pipeline independently and to ask questions.

After the training, participants progressed to an unmoderated session where they worked through the prescribed condition independently. However, participants were allowed to consult with the research team for technical help. If participants were unable to make progress, we provided hints. This was in line with the goal of the user study: to provide the participants with an experience of creating an end-to-end pipeline in both conditions so that they could evaluate the relative performance differences. As an optional extension to the study, eight participants were offered a free-form open-ended pipeline creation, where the participants implemented their own ideas into a pipeline with InstructPipe. Note that the optional section was offered based on progress of the participant in the previous sections and time constraints.

One is shown in Figure 3b and other is in our supplementary material.
After trying all pipeline-condition combinations, participants answered open-ended questions in a semi-structured interview. The interview script is available in the appendix §E. Participants provided their general outlook of each condition, listed pros and cons, identified potential future use cases, and critiqued the user interface for future improvements. In total, participants spent 55-65 minutes in the study.

8.2 Participants

16 participants (4 females) were recruited from an internal participant pool at [Anon.]. Our sample size was informed by prior work on early stage usability testing [16]. A diverse variety of job profiles were represented, but no software engineers were included. See Table (Table 2) for a full breakdown.

Table 2. Participant demographics for the user study, showing various demographic characteristics and skills relevant to InstructPipe.

As a reminder, we intentionally recruited novice users, as we envision them to be the intended users of InstructPipe. Our criteria was based on self-evaluation ratings of the below prompts:

- Please provide an self-evaluation of your programming experience
- Please provide an self-evaluation of your machine learning (ML) skillset

8.3 Metrics

In addition to the qualitative data from the interview, we measured the following quantitative data.

8.3.1 Task Completion Time. Back-end logs were used to collect timestamps for starting and ending events. Then, completion times for each condition was calculated per task for each participant.

8.3.2 The Number of User Interactions. We used the number of user interactions (introduced in §7.2) to measure the user’s objective workload. Different from the results in §7.3, we report an absolute value here because all the pipelines are controlled in the system evaluation.
Table 3. Task completion time and the number of human interactions in the user study (N=16).

<table>
<thead>
<tr>
<th>System</th>
<th>Time (secs)</th>
<th># Interactions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Median</td>
<td>IQR</td>
</tr>
<tr>
<td>InstructPipe</td>
<td>203.5</td>
<td>156.25</td>
</tr>
<tr>
<td>Visual Blocks</td>
<td>304.5</td>
<td>124.25</td>
</tr>
</tbody>
</table>

8.3.3 Perceived Workload. The raw task load index (Raw-TLX) questionnaire was used to measure participant’s perceived workload [20]. This questionnaire was a subset of the NASA-TLX (part I). Participants filled out the questionnaire after each condition (InstructPipe or Visual Blocks).

8.4 Results

InstructPipe Significantly Reduces Users’ Workload. Table 3 shows the results of two objective metrics measured in the study. The Wilcoxon signed ranks test found significant differences on both scale (both $p < .001$).

Figure 10 further visualizes the results of users’ perceived workload in six sub-scales. The Wilcoxon signed ranks test revealed significant difference on five sub-scales, all but “Mental Demand” (see the end of this subsection and §9.1 for more qualitative results and explanations). Furthermore, the test indicates that all participants unanimously vote InstructPipe provides lower or equal workload on the subscales of “Physical Demand”, “Temporal Demand”, “Performance” and “Effort” ($W = 0$). These quantitative results, with both objective and subjective metrics, demonstrate that InstructPipe can effectively reduce users’ workload during the pipeline creation process.

Users’ qualitative feedback is also aligned with our quantitative results. Participants complimented that InstructPipe is “helpful”[P16] and “obviously easier (to use) than [Visual Blocks]”[P1]. P11 and P6 further elaborated how InstructPipe enhances user experience when the user builds a visual programming pipeline:

“I feel like I can talk in natural language, and it (InstructPipe) can write the code for me.” [P11]

“The pipeline generated by [InstructPipe] could be pretty close to what I want ... Or maybe sometimes not, but that’s okay. I got most of the blocks there, and then it’s up to me to figure out how to connect them.” [P6]

On-boarding Support of Visual Programming. P1, P5, and P9 explicitly mentioned that there is a “learning curve” in the visual programming system for beginners, which validates our statements in §1.

“There is a learning curve to it (using the visual programming system) for sure, because you have to like read each node carefully” [P1]

To this end, participants commented that InstructPipe is a good on-boarding tool in visual programming systems, especially for non-experts, to get familiarized with the system by having a ready solution.
“[InstructPipe] lets you know these nodes exist [when the pipeline appears after the instruction]. It’s like a super speedy tutorial.” [P7]

“If you don’t have experience in visual programming, you will appreciate [InstructPipe] much more ... With [InstructPipe], the structure is there. I feel less worried about making mistakes. It’s, like, giving you examples. It’s easier than starting from scratch.” [P5]

Anecdotally, three participants asked for InstructPipe during the Visual Blocks condition.

Use Scenarios: Accessible Prototyping and Education. In the open-ended session, we observed participants could easily utilize InstructPipe to prototype a pipeline for various daily life or business purposes. For example, P14 tried InstructPipe with “summarize real estate price increase in San Diego California over 2023”. Compared to using LLM chatbots, InstructPipe helps the user quickly build a more explainable pipeline in which the user can track (or modify) the information resources. P4 prototyped an interactive VQA app by “Describe the product in the camera”. P13 further shared his thoughts on how this rapid and accessible prototyping experience can support future business:

“It (InstructPipe)’s going to facilitate the prototype building for PMs (Product Managers) ... I was a PM ... Back then ... My biggest fear is to code ... I have lots of ideas, but my challenge is how to translate idea into the technical world and see a prototype. I think that this app expedites me in that process a lot.” [P13]

Another emerging theme was regarding educating kids on programming as explained in the following:

“With [InstructPipe], I don’t need to teach them (kids) to code in order for them to build something ... Some kids like to code, some kids like to create stuff, but don’t want to be bored with learning the syntax of coding ... Using [InstructPipe], I can see kids can build, like, customized chat-bots or interactive Wikipedia.” [P13]

Limitations and Feature Requests. Participants also pointed out the limitations of InstructPipe. As reported in §8.4, InstructPipe failed to significantly reduce novice users’ mental demand. P13 shared their experience on how InstructPipe affects their mental workload:

“[Using InstructPipe] is a little mentally demanding ... I have to debug ... If it doesn’t help (generating an almost 100% correct pipeline), I have to go through all the nodes ... I don’t like debugging.” [P13]

Multiple participants asked for a function that allows them to edit their original prompt if a generated pipeline does not satisfy their requirement.

“Rather than fixing it on my own [on the node-graph editor], I would have gone back and changed my prompt ... I’m pretty sure I could have gotten it closer ... because I just spent so much time figuring out what the prompt should be. That’s kind of like already where my brain was and I knew that something was wrong there (the prompt), but I would have to switch over to the other mode (visual programming) of figuring out what was wrong on the pipeline.” [P15]

On the contrary, P13 held a different opinion of the instruction iteration and warned that it may sacrifice users’ “hands-on” experiences:

“I’m very hands-on with techs. I would like to understand what’s going on [rather than prompting LLMs to generate everything for me]. I want to like think for myself and then compile all the information myself.” [P13]

9 DISCUSSION
We first revisit our research questions raised in §1, and then share two insights in the project.
RQ1: What is the design space of primitive nodes for vision-language pipelines? How does its corresponding implementation impact users’ creative processes?

This work contributes a design matrix and its corresponding implementation, as is shown in Figure 2. With our implementation, we found that participants can easily build various creative pipelines. Some of the ideas in the pipelines are even similar to the pipelines used in the latest academic papers. For example, Figure 8c is similar to the data generation pipeline used in InstructPix2Pix [5]. Figure 8a utilizes two different LLMs playing different roles (i.e., a text summarizer and a fact checker) in the pipeline. A scale-up version of such role-play LLMs can further stimulate social interactions among AI agents [39, 40]. Adding another Vision-Language Model to each agent can further empower the AI agent with visual intelligence to perceive our physical world [36]. Visual programming provides developers an interactive platform to explore this future [16], as long as the platform provides sufficient nodes in the pipelines. Our design matrix provides a taxonomy of multimodal ML models which guide the implementation of such foundational platforms. Beyond the vision-language domain, we encourage future extensions of the matrix towards a more complete multimodal domain with speech, ubiquitous devices, and beyond.

RQ2: How can an LLM-assistant support users to build a pipeline beyond “from scratch”? How does this new experience influence future system design in visual programming?

We presented our technical solution in §5. After using InstructPipe, participants frequently expressed their desire to iterate on their instruction, which is aligned with prior observation of user behaviors in using coding tools [33]. This phenomenon reveals strong signals that instruction-based approaches may be the key element for Natural User Interfaces (NUIs) [24]. For years, HCI researchers have been pursuing this dream, but ended with “natural user interfaces are not natural” [37]. We believe that InstructPipe brought the community one step forward in this journey by introducing a new programming schema: initializing from natural languages and further editing in a visual space with a node-graph editor. This enables a more non-expert-friendly workspace compared to existing LLM coding support through a no-code interface and a validation module in visual programming.

Meanwhile, InstructPipe also introduces new challenges to the community. As P15 pointed out in §8.4, switching between visual thinking and text-based instruction ideation may increase users’ mental workload. Additionally, an iteration on the instructions may eliminate users’ hands-on experience in visual programming. These findings are related to the existing research that discussed “levels of interaction” in GUI generators [38] and the research question on how to better accommodate “humans’ expression of intents” [10] in UIs empowered with generative AI. Overall, this work opens up a wide range of future directions in visual programming with LLMs that remain under-explored, and we encourage future work to further conduct in-depth to guide future system design.

9.1 Instructing LLMs Poses Challenges for Both Novices and, Potentially, Experts

In §8.4, we show that InstructPipe did not significantly reduce users’ mental demand. The cognitive load largely stems from the need for precise language when instructing LLMs, as suggested by P15’s comments: “you’re just putting them (every detail in a whole pipeline) all out [in one short prompt]”. This insight is aligned with Zamfirescu-Pereira et al.’s finding that non-experts may not prompt LLMs well [64]. Despite that participants expressed desire to turn ideas into pipeline, InstructPipe is designed to work with instructions and writing instructions is nontrivial.

Another observation is that even we, the inventors of InstructPipe, failed to write optimal instructions. As mentioned in §7.1, two authors annotated captions for the pipelines, and we observed multiple imperfections, especially for the complex ones. For instance, the two captions of Figure 8c are “Describe the image and turn it into a cat image” and “Edit an image by updating the image caption”. Neither captions explicitly describe the detailed pipeline flow clearly, and
therefore all the six evaluation trials (§7) were incomplete (see Figure 11a for one example). The average ratio of user interactions is 45.8%, more than twice the average value for our multimodal pipelines (20.8%). To further understand the cause of the failure, another author improved the instruction into “Caption a tiger image using VQA, modify the character in the caption into a cat using LLM, and finally generate a cat image based on the updated caption.” The resulting pipeline is significantly improved but still not perfect (Figure 11b). The user only needs to turn “Imagen” into another mode so that it also accepts the input “image” node. Revisiting the improved instruction, we instructed InstructPipe with “generate a cat image based on THE updated caption,” which actually missed the input image. To solve this issue, we encourage future work to explore an “idea2instruction” module that frames users’ ambiguous ideas, which are often more intuitive to humans to propose into actionable instructions for generating pipelines.

9.2 Online InstructPipe

In this project, we made the following assumption: visual programming and its LLM support are both offline. That being said, the system has no access to the Internet for exploring in-the-wild ML models online, and the node can only be pre-defined by the developers of the system.

What if there is an online InstructPipe that can find ML models, define nodes, implement the node dynamically with human instruction? The community has already established ML libraries and API services for the models (e.g., by Hugging Face [23]). Technically, future work needs to investigate how to build a model selector (similar to the Node Selector in Figure 4) that can intelligently select the correct API to be called in a node. This may effectively eliminate the biggest pain point of existing visual programming systems, as we mentioned in §1, i.e., the lack of node support. Additionally, combining online InstructPipe with the “idea2instruction” module in §9.1 can bring unprecedented user experience. For example, we envision a future in which researchers can brainstorm with the system and, the system will automatically return an AI pipeline with state-of-the-arts ML models to accelerate researchers’ creative workflow.

10 LIMITATIONS AND FUTURE DIRECTIONS

We discuss the limitations of our study and implementation, and provide future plans to address them where applicable.

Limitations of user study. As our user study participants only engaged InstructPipe in one hour, it is unclear whether users will still frequently use and appreciate InstructPipe supports in a long-term manner. Furthermore, all participants are all non-experts, hence we are not able to verify the effectiveness of InstructPipe for users with other levels of expertise.

Capabilities of code interpreter. Despite our code interpreter (§5.1) fixes simple bugs, it lacks advanced features in modern compilers such as syntax tree and code optimization. If the code writer happens to generate completely
irrelevant code, the interpreter may have to drop it. Future work may make code writer and interpreter into a loop to produce better working pipelines with “validation and reflection”.

Incomplete implementation of language-vision models. While we have added 27 new primitive nodes informed by the proposals, we still miss integration of recent open-vocabulary models like SAM [27] for image segmentation and CLIP [44] for outputting metadata from a joint input of image and text.

AI Ethics. InstructPipe currently cannot detect harmful data or misuse of AI. Future work should study effective methods to eliminate potential harmful uses.

11 CONCLUSION
In this paper, we introduced InstructPipe, a visual programming system powered by a wide range of multi-model nodes and an LLM support that facilitates users to build pipelines with instruction. The node implementation in InstructPipe was informed by our design guideline, which is based on 236 pipeline proposals from 58 participants. We implemented our instruction-driven pipeline generation support by decomposing the task into three modules: a node selection module, a code writer and a code compiler. To evaluate our design space, we organized a workshop of our system, and found that 95.3% of the pipelines created by the participants utilized at least one node from our new implementation space. Results in our technical and system evaluations demonstrate that InstructPipe provided users’ satisfactory “on-boarding” experience of visual programming systems and allow them to rapidly prototype an idea. We further discussed the issues we observed concerning LLMs in visual programming, related to both human factors and technical implementations. We hope that InstructPipe will inspire both novice and expert users through its accessible interface to visual programming, and engage a broader and more diverse community to develop creative machine learning pipelines.

(9033 Words)

REFERENCES


[53] Unity. 2023. Unity’s Graph Editor. https://docs.unity.cn/Packages/com.unity.visualscripting@1.7/manual/vs-interface-overview.html#the-graph-editor


APPENDIX

A  NEW NODES IMPLEMENTED IN INSTRUCTPIPE AND VISUAL BLOCKS

Drawing upon §3 Design Space, we implement the full list of 27 new nodes that have been integrated into the most recent iteration of Visual Blocks. These additions were formulated based on insights garnered from the community survey, as detailed below:

(2) **Google Text Search** (a Search node): given a text query, output a list of URLs returned from Google.
(3) **Google Image Search** (a Search node): given a text query, output the first image from Google.
(4) **PaLI** [8, 9] (a Search node): Pathways Language and Image model that generates text from a joint of image and text inputs.
(5) **Imagen** [45]: a text-to-image diffusion model.
(6) **OCR** [49]: a text recognition model from an image query.
(7) **Colab Output**: given input code, output to a Colab node to execute the code.
(8) **URL to HTML**: given an URL as text, fetch the webpage as text output.
(9) **URL to Image**: given an URL as text, fetch the data as an image output.
(10) **Markdown Viewer**: given a text in MarkDown language, visualize it as a webpage.
(11) **Text Processor**: given multiple text, join them together or format them in a certain way.
(12) **String Picker**: given a list of text and an ID, fetch one of them as output.
(13) **HTML Viewer**: given a text in HTML language, visualize it as a webpage.
(14) **Hand Pose Detection**: given an image, return landmarks of the hands using MediaPipe.
(15) **Hand Gesture Detection**: given an image, recognize the gesture using MediaPipe.
(16) **Face Landmark Detection**: given an image, return landmarks of the face.
(17) **Body Segmentation**: run a deployed MediaPipe body segmentation model.
(18) **Object Detection**: given an image, output the detected object class name and their bounding boxes.
(19) **Bounding Box Visualizer**: visualizing bounding boxes of machine learning models.
(20) **Landmark Visualizer**: visualizing pose landmarks of machine learning models.
(21) **Mask Visualizer**: visualizing segmentation mask of machine learning models.
(22) **Virtual Sticker**: given landmarks, an original image, and a sticker image, overlay sticker image onto the original image upon the given landmark position.
(23) **Text Toxicity**: given a text input, output the toxicity level of the language.
(24) **Text Processor**: given (a) text input(s), output a reformatted text.
(25) **Google Sheets Reader**: given a Google Sheets URL and range of data, fetch data and output as text.
(26) **Google Sheets Writer**: given a Google Sheets URL and range of data, output the input text directly to the Google Sheets.
(27) **Custom API**: Given an API URL, query it via GET or POST methods and fetch the result.

Nodes marked with * are out of the scope of InstrutPipe.
B DETAILS OF VISUAL BLOCKS WORKSHOP

B.1 Workshop Organization

Our workshop participants are derived from different backgrounds, which diversifies the creation and design of multi-modal ML pipeline, including three students, four researchers, seven software engineers, two designers, and seven product or program managers. When each participant was recruited, they filled out a consent form that stated that all feedback through the workshop, both online and offline, would be recorded, documented and reported.

B.2 Contributed Pipeline

A committee of ten ML experts (eight of them opted out of the workshop, nine or them had a Ph.D. or a senior title) reviewed all the community-created pipelines in five days and voted for three best pipelines in text, visual, and multi-modal categories. We elaborate more pipelines contributed from the participants in Figure 12.

C SELECTED PROPOSALS FROM ONLINE SURVEY

Given that some proposals violate anonymous or IP confidential policy, we provide a curated list of selected proposals from our line survey of Visual Blocks from 58 contributors, to inspire future research in visual programming and human-AI collaboration.

Proposals that are achievable with InstructPipe and our iterated version of Visual Blocks:

(1) Grab yesterday’s meeting transcription and do the following: Create a meeting summary email, outline TODOs for discussed in the meeting, and tell me whether there were unresolved discussion points that need follow up.

(2) Paraphrase text in the style of a famous author (e.g. Shakespeare) and generate a spoken version to a given beat/rhythm.

(3) Process the my NASA-TLX data in the sheet, and show me how to plot it in python.

(4) Detect when I am not paying attention to meetings.

(5) Blur my background when it detects two people in the view.

(6) In a video conference call, detect my presence and hand gestures that trigger key strokes that unmute/mute me, disables/enables video, raise hand, send thumbs-up emoji etc.

(7) Change the style of the image so it looks like taken by a photographer.

(8) Translate the Japanese menu into English and show some pictures on the ones that are hard to translate.

(9) Check fake news.

(10) Decode sign language from webcam.

(11) Analysis and summarization of Reddit/Twitter threads/news.

(12) Summarize my current Search (text, videos, images, etc) for diverse use cases (e.g. trip planning, learn about a topic) and organize them spatially, like a mind palace.

(13) Grab current stock information and summarize it in three sentences.

(14) Take a photo of a medical record (like physical exam); explain the terms and suggest each item is normal or not; support multi-languages.

(15) Generate a picture based on my language inputs.

(16) Generate a wallpaper according to the theme of the exhibition.

(17) Tell me which animal/plant/mountain/dish it is in the video/photo, and find the summary/details of them.

(18) Summarize what’s in my fridge and tell me what to cook.
(a) Given a place, number of days to stay, and number of people, output a trip plan.

(b) Change given image into Picasso style.

(c) Write me a paragraph about my trip to San Francisco with two images of the highlights.

(19) Take today trends and make 5 Headlines.
(20) Compare A and B and summarize the similarities and differences between them.
(21) Suggest makeup / lipstick colors / hairstyles / clothing styles for this person, for formal / casual / blablabla occasions.
(22) Find the (similar) product of this object in the webcam video/photo.
(23) Provide more information about specific pieces of text, e.g. defining a word that the user is pointing to.
(24) Summarize my coding progress into a presentation.
(25) Analyze the quality of responses to a specific set of questions.
(26) Identification of elements of an image.
(27) Automatically generate a design doc according to the related meeting notes.
(28) Recognize the emotion of someone in webcam.

Proposals for future work:

(1) Analysis my calendar and draw my route today in google map. (Missing a Maps node)
(2) Calendar management: Given my calendar and my weekly / daily goals - tell if the calendar is too full, suggest which meetings to reduce/expand to match the goals. 2 - Leave for office - Look at map images from last week, and current image from last two-three hours - suggest when to leave for office to save time. (Missing a Calendar node, but achievable from text input)
(3) In a video conference call, detect my presence and hand gestures that trigger key strokes that unmute/mute me, disables/enables video, raise hand, send thumbs-up emoji etc. (Missing integration with video conferencing software)
(4) Automatic photoshop if humans are detected in the image: Touch up faces of people in the image by removing shadows, increasing contrast etc. (Missing ControlNet to remove shadows)
(5) Answer questions about things that the user has seen/stored (memory). (Missing access to user memory)
(6) Identify my cat from the webcam video and add a cool sunglasses to his face. (Missing real-time landmarks model on cat)

D ONLINE SURVEY

[ Consent Form ]
1. Have you ever used Visual Blocks? (no worries if you haven’t, it’s great to learn what you want from your unique perspective!)
2. Have you contributed pipelines to Visual Blocks codebase?
3. Suppose we have a intelligent ML pipeline maker that can handle language, images, and webcam input, what is the top pipeline that you want to make with language models as the main input, preferably with a real-world use case? Describe concisely in one sentence.
4. What is the top pipeline that you want to make with image as the main input, preferably with a real-world use case? You may combine with language.
5. What is the top pipeline that you want to make with webcam as major input, preferably with a real-world use case?
6. What would be another pipeline that you want to make using ML with a real-world use case?
7. (optional) please feel free to list as many pipelines as you could in the last brainstorming box :) Thank you for your contribution!
8. Would you like to be invited to attend a workshop of Visual Blocks?

E SEMI-STRUCTURED INTERVIEW SCRIPT

[ Introduction ] ( Start timing! 60 min max. )
Hello, my name is X.

First, I would like to thank you for your participation and completing the consent form. Today, you will be a participant in a user study regarding machine learning and visual programming. Your data will be kept anonymous.

Additionally, as a researcher I have no position on this topic and ask that you be as open, honest, and detailed in your answers as possible. Do you have any questions before we begin?

Basically, visual programming borrows the metaphor of block building and allows novice users to develop digital functionalities without writing codes.

[Show Visual Blocks]

Here, each block is called a node, and each node takes in specific inputs, then returns the desired outputs. What you can do is to connect a series of nodes together as a pipeline to achieve a high-level goal.

We are going to walk you through our Visual Blocks system and ask you to actually use Visual Blocks in two conditions to create a few applications.

[ Tutorial ] (Start timing! 10 min max.)

Before we get started, let us do a tutorial of our system.

[ Study and TLX ] (Start timing! about 30 min)

[Leverage the counter-balanced sheet and give user a task]

[Think aloud. Have a short discussion with the user. What’s the user’s plan to achieve this given functionality?]

[ Interview ] (Start timing! about 15 min)

1. What’s your impression of Visual Blocks / InstructPipe [counterbalanced]? Do you need many edits / operations to make it work?

2. Are there any pipelines you come up with in work scenarios / casual scenarios?

3. What works with InstructPipe? In what specific scenarios will InstructPipe be very helpful?

4. What does not work with InstructPipe? Would you give me an example?

5. Do you have any suggestions to improve the design of both systems?

6. Which kinds of technologies would be interesting to add?

7. What applications do you want to start with InstructPipe? And what applications do you want start without it?

That’s all for our user study. Thank you for your participation and we will compensate for your time.