InstructPipe: Building Visual Programming Pipelines with Human Instructions

ZHONGYI ZHOU*, The University of Tokyo , Japan

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
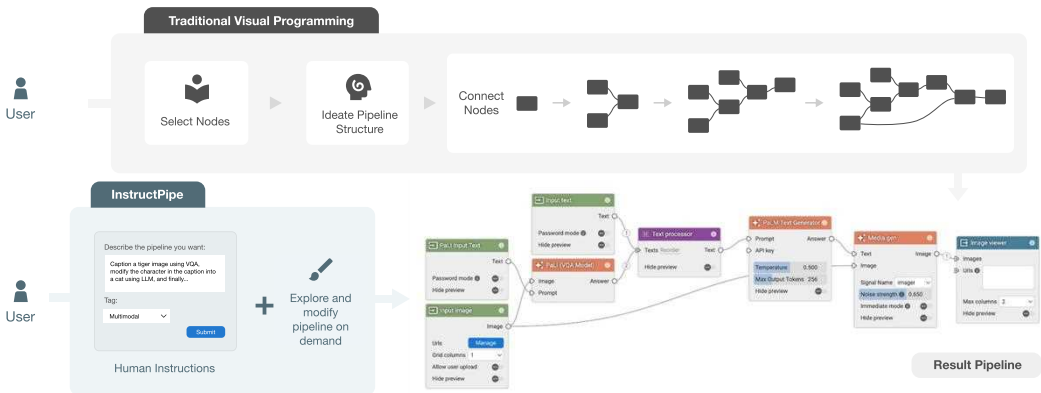
Fig. 1. Existing visual programming requires users to select nodes, conceive pipeline structure, and then create the pipeline starting from an empty workspace. In contrast, InstructPipe facilitates users to build interactive pipelines directly from text instructions.

Visual programming provides beginner-level programmers with a coding-free experience to build their customized pipelines. Existing systems require users to build a pipeline entirely from scratch, implying that novice users need to set up and link appropriate nodes all by themselves, starting from a blank workspace. We present InstructPipe, an AI assistant that enables users to start prototyping machine learning (ML) pipelines with text instructions. We designed two LLM modules and a code interpreter to execute our solution. LLM modules generate pseudocode of a target pipeline, and the interpreter renders a pipeline in the node-graph editor for further human-AI collaboration. Technical evaluations reveal that InstructPipe reduces user interactions by 81.1% compared to traditional methods. 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.

CCS Concepts: • **Computing methodologies** → Visual analytics; *Machine learning*; • **Software and its engineering** → **Visual languages**.

---

*Work conducted during an internship at Google.

†Corresponding author: me [at] duruofei [dot] com; Also contact: zhongyi [at] iis-lab [dot] org

---

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 effectively reduces technical barriers for users to prototype creative applications. Advances in machine learning (ML) further stimulate growing interest in visual programming. Open-sourced ML libraries (*e.g.*, TensorFlow [1], PyTorch [40], and Hugging Face [55]) provide users with various encapsulated modules to accelerate AI project development and experimentation. Meanwhile, this also provides valuable protocols for visual programming developers to create systems for ML applications [13], where ML practitioners can interactively test "off-the-shelf" models on the node-graph editor. Recent foundational models like large language models (LLMs) [3, 6, 50] and findings on Chain-of-Thought [54] further stimulate a community-wise interest in visual programming [14, 56, 58], which provides users interactive experiences to explore AI chains.

Despite the development of visual programming platforms in various domains, we observed that existing systems share one similar characteristic: users usually start a creative process in the workspace *"from scratch"*. This implies that users need to 1) select nodes, 2) ideate the pipeline structure, and finally, 3) connect nodes from *a completely empty workspace*. For users who are unfamiliar with a particular visual programming platform, such processes can easily overwhelm them, degrading their overall programming experience.

Similar issues also exist when users write programs using text-based editors (there exist many built-in functions in a particular programming language and multiple variables in a program), but advances in LLM assistants show that such challenges can be effectively reduced. For example, GitHub Copilot [15] makes it possible for users to generate code by simply describing users' requirements in natural language. Even though the generated code is not absolutely correct, the AI assistance usually finishes a large portion of the task, and programmers may only need to make a few edits to achieve a correct result. To this end, we raise the following questions that motivate our work: *How can we build such an assistant to benefit visual programming users?*

In this paper, we built InstructPipe, an AI assistant for visual programming users to generate a pipeline through natural language instructions (Figure 1). We implemented InstructPipe on Visual Blocks [13], a visual programming system for prototyping ML pipelines. One major technical challenge in implementing InstructPipe lies in the lack of visual programming data. Different from the standard approach to building a copilot for the text-based editor (*i.e.*, training LLMs using large-scale text-based programs online), it is impractical to collect sufficient data for a particular visual programming platform. We addressed this issue by decomposing the generation process into three steps. Using two separate LLM modules, the system first scopes the potentially useful nodes and then generates pseudocode for a target pipeline. InstructPipe then compiles the pseudocode and renders the pipeline on the node-graph editor to facilitate further user interaction. Our technical evaluation suggests that InstructPipe reduces user interactions by 81.1% compared to building pipelines from scratch. Our system evaluation with 16 participants demonstrated that

InstructPipe significantly reduced users' workload in their creative process. Qualitative results further reveal that InstructPipe effectively supports novices' *on-boarding* experience of visual programming systems and allows them to easily prototype a concept for various purposes. As one pioneering work on visual programming copilot, we also observed new challenges caused by humans' cognitive characteristics and proposed future technical directions toward a next-level, open-ended AI prototyping assistant.

In summary, this work offers the following contributions:

(1) InstructPipe, an AI assistant that enables users to *build ML pipelines from human instructions*,
(2) System design and technical development of InstructPipe, which includes two LLM modules and a code interpreter that generate codes for a visual programming pipeline, compile the code, and render the pipeline in an interactive node-graph editor,
(3) A technical and a user evaluations that demonstrate effectiveness of InstructPipe, and the corresponding findings that reveal new challenges for the HCI community.

2 RELATED WORK

2.1 Visual Programming

A computer program defines the operation of computer systems. However, *"the program given to a computer for solving a problem need not be in a written format"* [48]. This future-looking statement, dating back to the 1960s, inspired several generations of researchers to design and build visual programming systems.

Today, visual programming systems (*e.g.*, LabView [26], Unity Graph Editor [49], PromptChainer [56], and Visual Blocks [13]) typically feature a node graph editor, providing users with a visual workspace to "write" their program by *"building blocks"* [19, 45]. Recent work further explored the application of visual programming in education [24, 27], authoring support [62, 63], and robotics [9, 21, 22]. For example, Zhang *et al.* [63] connected the visual programming tool to the concept of *teaching by demonstration* [30, 34, 66], allowing users to rapidly customize AR effects in video creation. FlowMatic [62] extended traditional visual programming interfaces into 3D virtual environments, providing users with immersive authoring experiences.

More recently, findings on LLM Chains [58] and Chain-of-Thought [54] 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 applications. In such a scenario, visual programming provides a great platform for users to focus on high-level exploration. Example research work and industrial products include PromptChainer [56], LangFlow [29], and ComfyUI [8].

This work offers technical contributions of implementing and evaluating an AI assistant that supports users to prototype an AI visual programming pipeline from text-based instructions. Compared to a mainstream user workflow in which people begin their creation from scratch, InstructPipe provides a new experience for beginner-level users to easily prototype a pipeline.

2.2 LLMs and Their Applications in Interactive Systems

Early multimodal ML work uses language models to solve Visual Question Answering (VQA) [2, 4], yet these solutions are limited to simple questions and cannot perform effective reasoning and problem solving [33]. LLMs revolutionized AI's reasoning capability [54, 65] in language, which motivate researchers to build LLM applications in various domains beyond NLP [38, 43, 47]. For instance, LLMs augmented the perceptual and planning intelligence in robotics [43], supported autonomous driving [31], and assisted clinical processes in medical science [28, 44].

(a) InstructPipe's instruction dialog.
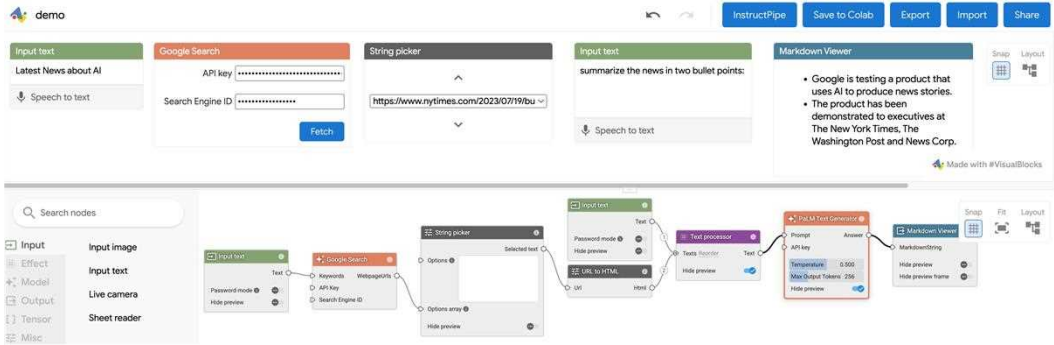
(b) InstructPipe's visual programming interface.

Fig. 2. The user interface of InstructPipe. The user can first click on the "InstructPipe" button on the top-right corner of the interface in (b). A dialog will appear, and the user can input the instruction and select a category tag. InstructPipe then renders a pipeline on (b), in which the user can interactively explore and revise.

The advances in LLMs empower recent interactive systems [39, 41] with enhanced machine intelligence. Recent research leverages LLMs to edit visualization [42, 53], receive AI explanation [57], facilitate communications [32], understand user interface [52], and study simulated social behaviors [38]. The revolution also motivated HCI researchers to design new interfaces for LLMs [25, 46]. Graphologue [25] augmented LLM responses by displaying interactive diagrams along with the response text, which depicts the semantic logic in a paragraph. Sensecape [46] provides users with a workspace to explore long LLM responses in a hierarchical structure.

InstructPipe focuses on utilizing LLMs for generating pipelines in visual programming with human instructions [20, 47]. This work shares a similar vision with Prompt2Model [51] and VisProg [16]. Prompt2Model [51] finetunes a BERT model [10] using data generated by instructions. VisProg [16] produces Python code from instructions with task-dependent few-shot prompting. However, prior arts focus on full AI automation and lack an interactive workspace for human-AI collaboration. In contrast, InstructPipe generates and compiles a pipeline (without finetuning), while rendering the pipeline in a visual programming interface, facilitating an interactive, collaborative, and explainable workflow.

3 INSTRUCTPIPE

InstructPipe is an AI assistant that enables users to generate a visual programming pipeline by simply providing text-based instructions. We implemented InstructPipe on Visual Blocks [13], a visual programming system for prototyping ML pipelines.
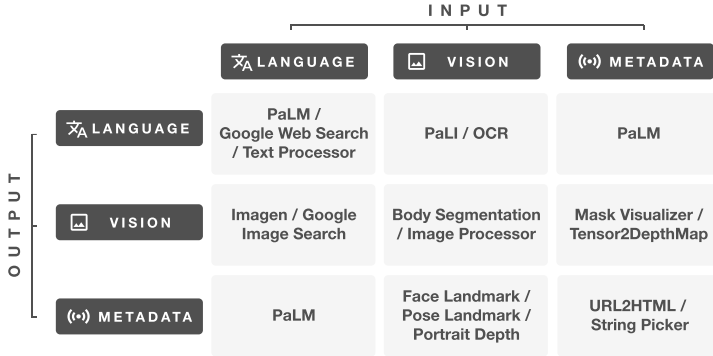
Fig. 3. An overview of major primitive processor nodes supported by InstructPipe. 17 of the 20 processor nodes are categorized by the type of accepted I/O into a 3×3 matrix. Note that "PaLM" represents two nodes in InstructPipe, *i.e.*, a text generation model and a chat model of PaLM [3].

3.1 User Workflow

To generate a pipeline, users can first click the "InstructPipe" button on the top-right corner of the interface (Figure 2b). The system then activates a simple dialog (Figure 2a) 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 the "Submit" button in the dialog, InstructPipe renders a visual programming pipeline on the node-graph editor. Based on the result, users can further refine the pipeline in the visual programming interface.

3.2 Primitive Nodes

InstructPipe supports 27 primitive nodes in Visual Blocks, including seven I/O nodes (*e.g.*, live camera and markdown viewer) and 20 processor nodes. Appendix A provides a description for each of the 27 nodes. Compared to related work [16, 47] that automates ad hoc ML inferences in specific use scenarios, we aim at an open-ended use case with diverse primitive nodes. Figure 3 visualizes 17 (out of 20 in total) processor nodes according to the I/Os of the nodes. The remaining three nodes are "Google Sheet" (which takes a Google Sheet URL as input and outputs the sheet data), "image_mixer" (which combines multiple images), and "virtual_sticker" (which casts a sticker on a person's face on a live camera). "Metadata" in Figure 3 indicates intermediate data used in ML pipelines, which can be a segmentation mask that describes an input image, or a URL that describes a target news that users want to read. As shown in the matrix, InstructPipe contains a wide range of nodes that supports the creation of complex ML pipelines.

4 PIPELINE GENERATION FROM INSTRUCTIONS

InstructPipe leverages LLMs to generate visual programming pipelines from text instructions. There exist two prevailing approaches for customizing LLMs: 1) fine-tuning LLMs [5, 32], and 2) few-shot prompting [12, 38]. For our task, fine-tuning LLMs requires a substantial volume of annotated data comprising pairs of pipelines and prompts. Additionally, a growing list of nodes poses new challenges to scaling this approach with new data annotations. In comparison, few-shot prompting can seem more practical [16, 54, 59], but it is challenging to design an efficient prompt that fits within a reasonable number of tokens. The configuration file of the 27 nodes alone includes 8.2k tokens. Moreover, because of the combinatorial explosion of the 27 nodes in the system, it is
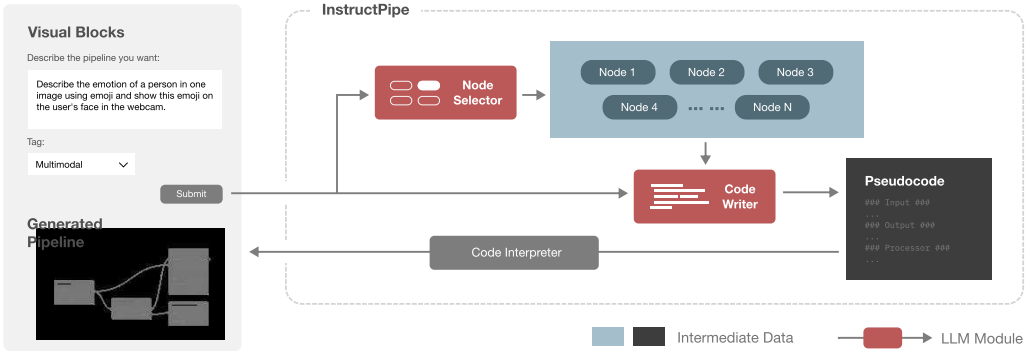
Fig. 4. Workflow of InstructPipe. First, users describe their desired pipeline in natural language and designate it with a language, image, or multi-modal tag. InstructPipe then feeds user instructions into a node selector to identify a relevant set of nodes. Subsequently, both the instructions and the 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 refine and interact with it further within Visual Blocks's node-graph editor.
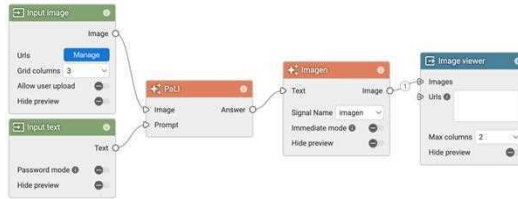
(a) Pipeline.

(b) pseudocode.

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.

not clear how many prompt examples are needed and how we can construct in-context pipeline examples.

To this end, we implement InstructPipe with a two-stage LLM refinement prompting strategy, followed by a pseudocode interpretation step to render a pipeline. Figure 4 illustrates the high-level workflow of the InstructPipe implementation. InstructPipe includes two LLM modules (highlighted in red): 1) a Node Selector (§4.2) and 2) a Code Writer (§4.3). 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 instruction. In the Node Selector, we prompt the LLM with a very brief description

of each node, aiming to filter out unrelated nodes for a target pipeline. The selected nodes and the original user input (the prompt and the tag) are then fed into the Code Writer, which generates pseudocode for the desired pipeline. In Code Writer, we provide the LLM with detailed description and examples of each selected nodes to ensure LLMs have a thorough understanding of each candidate node. Finally, we employ a Code Interpreter to parse the pseudocode and render a visual programming pipeline for the user to interact with.

4.1 Pipeline Representation

The Visual Blocks system represents a pipeline as a Directed Acyclic Graph (DAG) in JSON format[1]. To compress the verbose JSON file, InstructPipe represents pipelines as pseudocode [16, 47], which can be further compiled into a JSON-formatted pipeline. The pseudocode representation is highly token-efficient. Figure 5 shows an example in which the pseudo compresses the JSON-based pipeline representation (2.8k tokens) into a 123-token pseudocode representation. The efficiency does come with a cost: it loses some fine-grained annotations of 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). InstructPipe leaves the task of finetuning these parameters to the user and focuses on generating the graphic structure of a visual programming pipeline.

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 [64]: an algorithm block typically starts with specifying the input/output and then explains the intermediate processor. We highlight 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 pseudocode, we do not annotate the 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.

4.2 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(s) 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 [17]) present a high-level overview of all functions[2]. The 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 the LLM to learn and adapt to the context of the task.

---

[1]Pipeline in JSON files: see supplementary materials for examples.
[2]See an example in the following link: https://numpy.org/doc/1.25/reference/routines.array-manipulation.html

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:
###input###
live_camera: Capture video stream through your device camera.
... // other input nodes
###output###
image_viewer: View images.
... // other output nodes
###processor###
google_search: Use Google to search web that returns a list of URLs based on a given keyword; usually selected with string_picker.
... // other processor nodes

Examples:
Q:
{'description': 'generate a photo and validate whether it is real or generated.', 'tag': 'multimodal'}
A:
['input_text', 'markdown_viewer', 'imagen', 'pali']
... // more in-context examples

You are a programmer responsible for helping the user design an AI pipeline.
Upon receiving a concise description from the user about the desired functionality of the pipeline, you should generate the whole pipeline using pseudo codes.

Guidelines:
1. Respond solely in pseudo codes, without additional commentary.
2. Utilize ONLY the nodes listed below; introducing new nodes is not permitted.
3. Ensure there's a minimum of one line in each pseudocode category: 'input', 'output', and 'processor'.

Below are the nodes you can incorporate into the pipeline:
... // detailed node configurations for each selected node

The following is a full list of nodes you may also use but those not included above are not recommended:
... // a full list of node types supported by InstructPipe

Examples:
Q:
{'description': 'generate a photo and validate whether it is real or generated.', 'tag': 'multimodal'}
A:
... // pipeline pseudo codes
... // more in-context examples

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

Fig. 7. The prompt structure for the Code Writer module. Detailed node configurations, see supplementary materials for examples, are listed in the highlighted region.

4.3 Code Writer

With a pool of selected nodes, the Code Writer module can write pseudocode for rendering a target pipeline. Figure 7 shows the structure of the prompt utilized in this LLM module. Similar to §4.2, the prompt starts with a general introduction and several guidelines. The major difference in 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 the full node configuration in the supplementary materials. Similar to the previous LLM module (§4.2), the prompt design here is also inspired by the documentation of several popular code libraries. Specifically, we gain inspiration from low-level function-specific documentation[3], which typically includes a detailed description and data types in the input/output, followed by one or more examples of 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: a combination of all the nodes causes many 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 pipeline tag (*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: the in-context examples serve 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

---

[3]See an example in numpy.shape: https://numpy.org/doc/1.25/reference/generated/numpy.shape.html#numpy-shape

(a) Before layout optimization.   (b) After layout optimization.

Fig. 8. A comparison of the same generated pipeline before and after layout optimization. (a): Each node is assigned with a default property value when convert pseudocode into a JSON file for rendering the pipeline. Such default values will cause sub-optimal visual effects. (b): Our layout optimization process re-arranges the layout for better presentation of the pipeline.

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 failure in node rendering. Note that we also observed this issue when combining the prompts used in [16]. 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.

4.4 Code Interpreter

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

(1) **Lexical Analysis**: InstructPipe first tokenizes each line of the pseudocode is into output variable id, node id, node type, and node arguments (§4.1).

(2) **Graph Generation**: The tokenized results allow us to build a graph structure in a node-graph editor that connects each node as specified by the pseudocode. We then generate JSON-formatted code, which Visual Blocks uses to render the pipeline in the node-graph editor. Note that the JSON code includes far more parameters than those used in InstructPipe for defining the graphic structure. InstructPipe fills texts in the "Input Text" node based on LLM-generated pseudocode and uses default values for the rest nodes. For example, by default, the temperature and the max output tokens for the PaLM node are set to 0.5 and 256, respectively. If users are not satisfied with the default values, they can interactively adjust the parameters in the node-graph editor.

(3) **Graph Rendering and Optimization**: A problem with employing the default parameters is that it results in a chaotic distribution of nodes in the node-graph editor: each node is located in a predefined position, and the edges go across the workspace without a reasonable arrangement. Therefore, InstructPipe traverses the graph with a breadth-first search (BFS) and arranges the nodes based on their depth values in the DAG. As shown in Figure 8, such post-processing can effectively enhance the visualization of a generated pipeline.

Our exploration shows that LLMs may not always generate accurate code as expected. One typical issue is that Code Writer tends to invent illegal nodes out of our library of 27 nodes. To address this, InstructPipe disposes such lines that leverage illegal nodes. If legal nodes use the output of such an illegal node, InstructPipe discards the edge connection between them to prevent bugs when running the pipeline. Another common issue is that the generated code may not have a correct order. For example, the generated code may be "Line 1; Line 3; Line 4; Line 2;", in which

"Line 1 – 4" represents four lines of code, and the correct order should be from "Line 1" to "Line 4". Such ordering issues would cause an input value of the current line to become undefined, because its definition is mistakenly placed afterward. We address this issue by stacking the line that is not ready for execution, similar to graph construction by traversing an adjacency list. The Code Interpreter runs in a loop to interpret the code until all legal lines in the stack become ready.

5 TECHNICAL EVALUATION

Evaluating our system is challenging. Ideally, the evaluation set should cover 1) diverse visual programming pipelines (*i.e.*, the pipeline factor) and 2) a variety of instructions created by different individuals for each pipeline (*i.e.*, the human factor). However, such a combination requires a significant number of user evaluation sessions, which is impractical to deploy. For example, if an evaluation set includes 50 pipelines and 20 instructions for each pipeline from 20 different participants, it requires 1,000 user study sessions. Asking participants to attend multiple pipeline creation sessions seems efficient, but it introduces new learning transfer effects to the study. The learn effects usually require additional user study design with counterbalancing, which eventually increases the total number of required sessions.

To address this issue, we decouple the two factors that cause variations by decomposing the whole evaluation into two evaluation sessions. The major benefit of this design is that it allows us to deploy the evaluation tasks with a reasonable workload while maintaining the rigorousness of the whole evaluation. In our first evaluation session (*i.e.*, technical evaluation), we focus on understanding the technical performance of InstructPipe among a variety of pipelines. In the second evaluation session (*i.e.*, user evaluation), we control the pipeline factor and examine the human factor by recruiting different participants for prototyping controlled pipelines. The first technical evaluation provides crucial data on the "accuracy" of InstructPipe, which guides us to select such representative pipelines with as little bias as possible. More importantly, the user evaluation further allows us to understand how user experiences are affected under two conditions: with and without InstructPipe.

In this section, we focus on explaining the first step of our evaluation, the technical evaluation, and the following user evaluation is covered in the next section.

5.1 Data Collection

To conduct the technical evaluation, we first organized a two-day hybrid workshop with 23 participants, aiming to collect pipelines that Visual Blocks users build for their creative usage. Six attendees claimed that they had prior experience in using Visual Blocks. At the beginning of the workshop, we gave 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. We required the participants to export the JSON file from Visual Blocks and upload it to our Google Drive every time they finished a pipeline. We also emphasized that they should caption their pipelines when they export the files. Such data pairs (pipeline/caption) by the participants constitute important raw data we use for the technical evaluation.

Note that Visual Blocks includes more primitive nodes than the 27 nodes covered by InstructPipe. The workshop is an open-ended creation process, in which participants are free to any node available in Visual Blocks. The InstructPipe feature was not available in the workshop.

5.2 Data Post-Processing

After the workshops, one author carefully examined each collected pipeline and found several critical issues in the raw data:
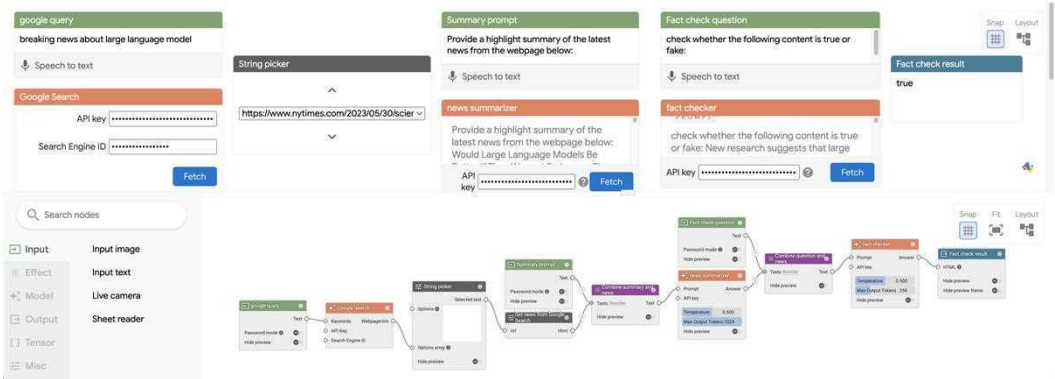
(a) Search news from Google, summarize it and then conduct fact check. Input: a keyword for Google Search; Output: a summarization of the news and a fact-check result.
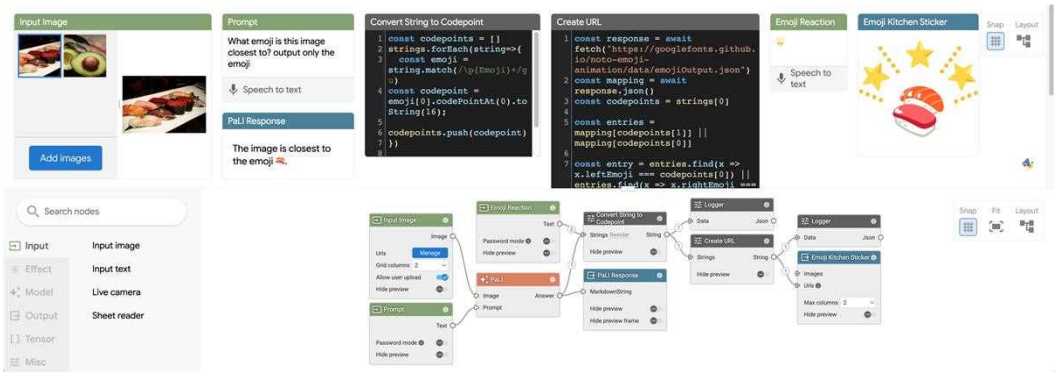
(b) Generating an emoji from a photo. Input: a photo uploaded by the user; Output: an emoji generated from the photo.
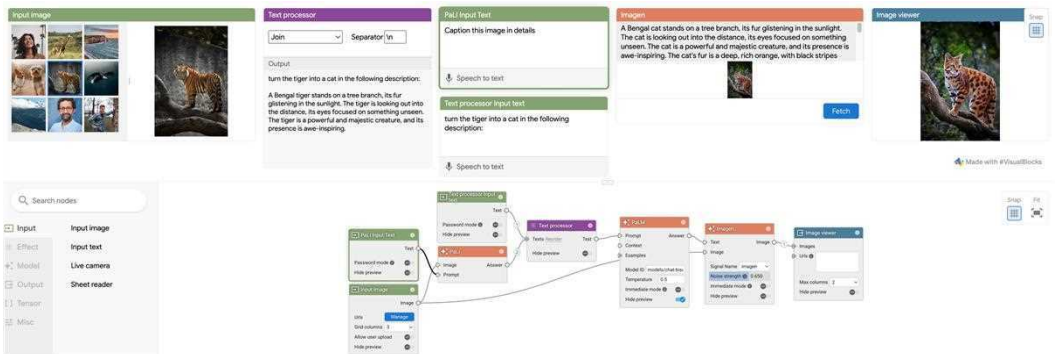
(c) Turning a tiger into a cat. Input: an image of a tiger; Output: an image of a cat.

Fig. 9. Example pipelines participants built in the workshops.

- **Incomplete pipelines**. There exist pipelines uploaded by the participants that were incomplete.
- **Isolated graphs**. There exist pipelines that include at least one isolated subgraph. The isolated subgraph, as opposed to the main graph, is defined as a graph (or a node) that has no connection to the main graph in the pipeline that provides the main functionality of

the pipeline. The "Image viewer" node on the bottom-left corner of Figure 8b We observed that some participants typically would like to explore the system by working on a separate sub-space. While we acknowledge its usefulness, leaving such "redundant" graphs in the raw data for the evaluation would cause issues when we calculate the number of user interactions (*i.e.*, the metric used in the evaluation that will be defined in the next subsection).

- **Low-quality captions**. While we explicitly required the participants to write descriptive captions, we found some captions written by the participants were either empty or low-quality (*e.g.*, *"newsletter"*, *"image editing"* and *"[participant name]-demo"*).

The observation motivated us to post-process the raw data to present more rigorous evaluation results. We first removed incomplete pipelines and the isolated graphs in each pipeline (if there are any). Although the low-quality captions may reflect real-world scenarios on how the users may use our system, such a hypothesis cannot be officially validated. In our data collection process, participants are situated in the scenarios of writing captions instead of instructing an AI assistant. When users generate instruct pipelines, they may be aware of the importance of writing good instructions, and thus, such low-quality captions would not be as usual. More importantly, the technical evaluation focuses on considering the variation of the pipeline, instead of text instructions. In the next user evaluation (§6), we will explicitly situate a new group of participants in instructing the assistant and observe the user behaviors and the variation of human instructions on the same pipelines. Therefore, we were motivated to enhance the quality of the captions for fair technical evaluations. Two authors individually annotated the caption of each pipeline separately by referring to the original captions and pipelines authored by the participants. It is important to note that we finished the workshop and the data annotation task *before* we completed the system implementation. The two authors had no experience using InstructPipe before completing the annotation. We believed this process could effectively enhance the quality of the captions while maintain fairness of the technical evaluation.

As we clarified in §5.1, the workshop is designed to be an open-ended creation process. This indicates that the dataset inevitably includes out-of-scope nodes like "custom scripts" (in which the participants write code to process the input data and return custom outputs; see Figure 9b for an example) and "TFLite model runner" (which call a custom TensorFlow model with a URL input of the model in the TF-Hub). As we explained in §4.1, 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 removed pipelines that contained "out-of-scope" nodes in our technical evaluation.

The final 48 pipelines (out of 64 pipelines) are comprised of 23 language pipelines, seven visual pipelines, and 18 multi-modal pipelines. Figure 9 shows three pipelines created by the participants. Figure 9b is an example of the pipelines that include out-of-scope nodes, and therefore is not included in the final 48 pipelines. In the technical evaluation, we ran our generation 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 using the metric that will be introduced below.

5.3 Metric: The Number of User Interactions

Evaluating a generated pipeline is more complex than other tasks that have a universally recognized definition of accuracy. In this project, intuitively, an accurate generation implies that users only need to do very few edits (or even no edits) based on the generated result. This intuition inspires our definition of "the number of user interactions" as the core metric used in our evaluations:

Table 1. The ratio of human interactions in the technical evaluation. Results are reported as mean ± standard deviation.

Overall	Language	Visual	Multimodal
18.9 ± 20.3%	17.4 ± 20.6%	17.6 ± 23.7%	20.8 ± 16.0%

*The Number of User Interactions is defined as the **minimal** number of user interactions needed to **complete** the pipeline from a generated pipeline.*

Note that there are countless ways to modify a generated pipeline toward a complete pipeline in practice. Nevertheless, the **minimal** number of user interactions, representing the "smartest" way(s) to make a modification, is deterministic, and this is an objective metric that can fairly reveal how many minimal efforts users need to spend to achieve their goal. A pipeline is considered **complete** when it satisfies the given instruction. We calculate the number of interactions from 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 interaction for the node deletion.

Our definition of the number of user interactions has two important implications. 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 fulfills the task described in the caption, we consider the pipeline 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 §4.1, the generation of property values is out of the scope of this work.

In the technical evaluation with various pipelines, it is unfair to report an averaged *absolute* value of user interactions because the complexity of the pipelines varies dramatically. For instance, the user may need three edits based on a generated result to complete a large pipeline that requires 20 edits from scratch. In another pipeline, the user also needs to do three edits starting from the generated result, but the whole pipeline only takes three edits to finish. Averaging these *absolute* values does not provide reasonable insights into how accurate the generation is. Therefore, we reported an averaged *ratio* of user interactions required to complete a pipeline "from our generated pipeline" to that "from scratch" as our target metric in the technical evaluation.

5.4 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 §5.3), demonstrating the effectiveness of the InstructPipe support. **Seven** generated pipelines directly satisfied with instructions without user interactions in all six trials, and **38** generated pipelines completed at least once in any of the six trials.

6 USER EVALUATION

While the technical evaluation demonstrates the accuracy of InstructPipe among various real pipelines created by participants, it is still unclear what is the actual user experience when real users go through the entire system workflow. Additionally, there is a lack of subject variation on a controlled group of pipelines in the technical evaluation. Therefore, we conducted an in-person user study of InstructPipe with another group of participants, aiming to provide more insights into our system performance. The study recruitment was in accordance with the ethics board of our institution. We obtained participant consent before the study began.
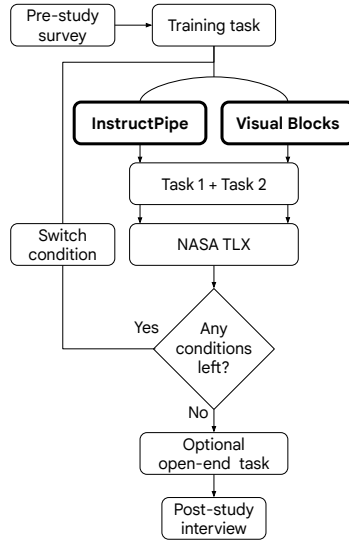
Fig. 10. A flow diagram of the user study. After a training session, participants completed the two tasks in each condition in the sequence determined by the counterbalancing protocol.

6.1 Study Design

In the user evaluation, we aimed to investigate how the interface condition (with InstructPipe and without InstructPipe; the independent variable) affects the user experience and behaviors (the dependent variable). We will refer to these two interface conditions as "InstructPipe" and "Visual Blocks" in the following content. Figure 10 visualizes the complete study flow. In each condition, participants completed the two pipelines with counterbalance (referred to as Task 1 and Task 2 in Figure 10).

We carefully designed this study to ensure a fair study that can be completed with reasonable efforts. The following elaborates how we made two important decisions that affected the rigorousness of our study:

- **Two controlled pipelines**. Our user evaluation focuses on two controlled pipelines. While we acknowledge more pipelines (*e.g.*, four, six, or even more, instead of two in our design) can enhance the solidness of the study, such design also dramatically increases the required groups of the user study session. For example, counterbalancing four controlled pipelines with two interface conditions results in $4! \times 2 = 48$ study orders. If there are four participants for each unique order, the whole study requires $48 \times 4 = 192$ participants, which is far more than a standard number in HCI system papers (*i.e.*, recruiting 10 - 20 participants for evaluation). We believe two pipelines fit the best in our case since they require $2! \times 2 \times 4 = 16$ participants.

- **Pipeline selection**. With the limited number of pipelines we can choose in the study, it becomes highly important how we select the two pipelines. There are two critical factors we considered in the pipeline selection process: representativeness and diversity. The representativeness implies that the selected pipelines should represent the averaged performance of InstructPipe. The diversity further suggests that the selected pipelines should provide various experiences to simulate the actual use scenarios in which InstructPipe may work well in some pipelines but not always. By following this guideline, we first selected four candidates, and the final decision was made after a pilot study with one participant to

test the level of pipeline difficulty. The two 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 in these two pipelines are 27.8% and 5.2%, respectively. See Appendix C for more details of the pipelines.

6.2 Procedure

The study starts with a 10-15 minute hands-on training of both conditions. The training included 1) all the Visual Blocks interactions needed to complete the subsequent steps of the experiment and 2) all the nodes that participants need to use for creating the pipelines they are assigned in the main sessions. 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 were asked to build pipelines under the given conditions. We verbally described the pipelines 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.

During the task, participants were allowed to consult with us for technical help. If participants were unable to make progress, we provided hints. We provided such support to ensure that each participant spent a reasonable amount of time on each task and had sufficient time for the following tasks. As shown in the results (§6.5.2), we provided far more support in the "Visual Blocks" condition. Therefore, we argued that this is a fair moderation in the study because it favors our comparative conditions (*i.e.*, it makes it easier for users to finish the pipeline under the "Visual Blocks" condition). As an optional extension to the study, eight participants were offered an open-ended pipeline creation, where participants prototype their own ideas with InstructPipe. This optional section was offered based on the progress of the participant in the previous sections and time constraints so that the study duration was controlled within the time we guaranteed in our recruitment process.

After trying all pipeline-condition combinations, participants answered open-ended questions in a semi-structured interview. The interview script is available in the appendix §D. 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.

6.3 Participants

We recruited 16 participants from an internal participant pool at Google. A diverse variety of job profiles were represented, but no software engineers were included. See Table (Table 2) for a full breakdown. Note that we intentionally recruited novice users, as we envision them as intended users of InstructPipe. The criteria for selecting the novice users was based on self-evaluation ratings of the below prompts:

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

6.4 Metrics

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

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

| ID	Job Title	Self-identified Gender	Age Group	Programming Experience	Machine Learninig Skill	LLM Usage
P1	Product Manager	Woman	25 - 34	Beginner	Beginner	At least once a month
P2	Image Tuning Engineer	Man	35 - 44	Intermediate	Beginner	At least once a week
P3	Program Manager	Woman	45 - 54	No experience	No experience	At least once a week
P4	Hardware Engineer	Man	35 - 44	Intermediate	No experience	At least once a month
P5	Technical Program Manager	Man	35 - 44	Beginner	No experience	At least once a day
P6	Senior Hardware Engineer	Man	35 - 44	Beginner	No experience	At least once a month
P7	Technical Program Manager	Woman	18 - 24	Beginner	Beginner	Never used it
P8	Technical program manager	Man	25 - 34	No experience	No experience	Multiple hours every day
P9	Solutions Engineer	Man	25 - 34	Beginner	No experience	At least once a month
P10	Program Manager	Man	55 - 64	Beginner	Beginner	At least once a month
P11	Program Manager	Woman	35 - 44	No experience	No experience	Never used it
P12	Lab Manager	Man	35 - 44	Intermediate	Beginner	At least once a week
P13	Partner Development Manager	Man	25 - 34	Beginner	Beginner	At least once a week
P14	Hardware Engineer	Man	25 - 34	Beginner	Beginner	At least once a week
P15	Global Supply Manager	Man	25 - 34	Beginner	No experience	At least once a month
P16	Global Supply Manager	Woman	55 - 64	No experience	No experience	At least once a week
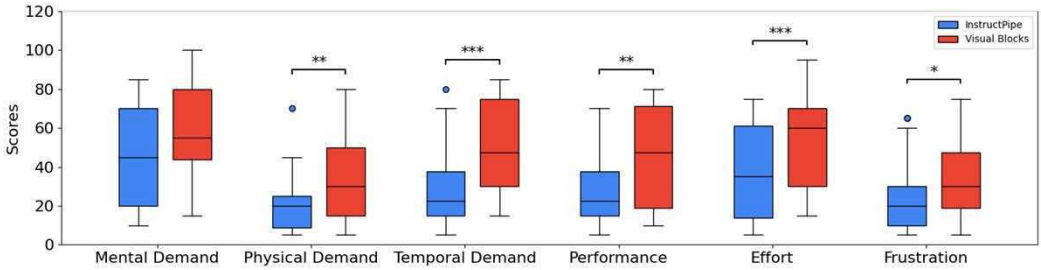
Fig. 11. Raw-TLX results. The statistic significance is annotated with *, **, or *** (representing $p<.05$, $p<.01$, and $p<.001$, respectively).

Table 3. Task completion time and the number of human interactions in the user study (N=16).

System	Time (secs)			# Interactions		
	Median	IQR	p	Median	IQR	p
InstructPipe	**203.5**	**156.25**	p <.001	**5.0**	**4.25**	p <.001
Visual Blocks	304.5	124.25		16.0	6.0	

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

6.4.2 The Number of User Interactions. We used the number of user interactions (introduced in §5.3) to measure the user's objective workload. Unlike the results in §5.4, we report an absolute value here because all the pipelines are controlled in the system evaluation.

6.4.3 Perceived Workload. The raw task load index (Raw-TLX) questionnaire was used to measure participant's perceived workload [18]. This questionnaire was a subset of the NASA-TLX (part I). Participants filled out the questionnaire after each condition (InstructPipe or Visual Blocks).

6.5 Results

6.5.1 ***InstructPipe 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 scales ($p < .001$).

Figure 11 further visualizes the results of users' perceived workload in six sub-scales. The Wilcoxon signed ranks test revealed significant differences on five sub-scales, all but "Mental Demand" (see §7.2 for more explanations and discussion). 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 the 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]

6.5.2 ***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 that motivates this project.

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

P1's comment matches with our observation of participants' behaviors during the study. In the Visual Blocks condition, we observed that people were more easily stuck in their creative purposes, which required our support[4]. Typical supports include 1) guiding participants if we notice they go too far away from the correct pipeline and 2) reminding them of an important node for the pipeline, although we introduced all the necessary nodes in our training session.

To this end, participants commented that InstructPipe is a good onboarding 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, and 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.

6.5.3 ***Integration into The Existing Workflow****.* InstructPipe is a feature available in Visual Blocks. In the interviews, participants particularly expressed their strong appreciation of this design as an AI assistant that enhances, instead of completely replacing, the existing user workflow (*i.e.*, the creative process purely using the node-graph editor):

---

[4]In the whole study, we encouraged the users to talk to us while keeping their focus on their task. While we offer explicit hints only when we consider a participant is stuck in an issue, it is still difficult to strictly define what conversation is " providing hints" and count them quantitatively. Therefore, we cannot fairly report quantitative data for "the number of hints" in both conditions. Here, we faithfully report this finding based on our observation and experience.

"[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]

While most participants shared similar opinions with P6, P15 raised one concern about such integration. The concern lies in the mental experience of the state that transits from the prompting task to the node-graph editing process:

"Rather than fixing it (the generated pipeline) 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]

6.5.4 ***Use Scenarios: Accessible ML Prototyping and Education****.* In the open-ended session, we observed participants could efficiently 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 an 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 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]

6.5.5 ***Limitations and Future Directions****.* Across the study sessions, we consistently observed a specific user behavior pattern: participants typically paused their pace when a generated pipeline appeared in the workspace. At these times, some participants used soliloquy, as in saying "Let me see", while others kept a focused stare on the workspace. These human behaviors strongly indicate that the participants were engaging in deep, contemplative thought.

This observation suggests that participants typically need to perceive a pipeline that appears in the workspace. Such a sense-making process brings new challenges to their creative process:

"[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]

Additionally, we observed that several participants spent more time crafting their prompts than the others. P15 is the participant who spent the most time writing the prompt. The following comments explain the reason, and how the prompting process causes extra mental workload to him/her/them.

"I'm a relatively visual thinker ... Getting the prompt right requires me to think in a way that is a lot more like precise and like getting it figured out without working it out live ... [When writing prompts,] you're just putting them (every detail in a whole pipeline) all out [in one short prompt]" [P15]

In addition to the lack of the original visual thinking experience in visual programming, P13 also warned that such simplification of the creative process into prompting experience 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]

7   DISCUSSION AND FUTURE DIRECTIONS

7.1   Human-AI Collaboration in Prototyping Open-ended ML Pipelines

Our technical evaluation (§5.4) shows that InstructPipe reduces the number of user interactions to 18.9 % (±20.3%). There are two implications from the results.

- InstructPipe automates *most* pipeline components with a single prompt.
- InstructPipe is *not* able to automate *all* the pipeline creation processes.

Such takeaways *differ from existing findings* that show LLMs can achieve full automation of ML inference [16, 47]. The main reason is that existing work built their ad hoc solutions for target use scenarios, respectively. In contrast, InstructPipe covers a wide range of mainstream ML models (§3.2) and aims for an open-ended use case. Our results show that LLMs (*i.e.*, GPT-3.5-turbo) still fail to write robust code with prompt engineering techniques. As we mentioned in §4.4, the main attribute of such failure is that LLMs tend to invent nodes (*i.e.*, the node type in the pseudocode) that do not exist in our node library, which causes execution error. Note that we explicitly instructed the LLM that *"introducing new nodes is not permitted"* in the system prompt for the Code Writer module (Figure 7). We also observed similar issues when we combined all the official prompts of four scenarios in VisProg [16].

While the execution error in existing work indicates the failure of the full solution, InstructPipe renders the executable components of the generation pipeline in a node-graph editor. This facilitates a collaboration protocol between humans and AI in which humans do not need to start a project from scratch. Results in §6.5.3 revealed that a wide range of participants appreciated this design of human-AI collaboration. We believe the main reason behind this observation is that most non-expert users prefer working on a task from a template (although sometimes it is not perfect) rather than starting the whole project from scratch. The visual programming system provides non-experts with a novice-friendly interface to work with AI in a visual platform. In this manner, *"users can do what matters most – building software by letting AI do the redundant work"* [61].

7.2   Three Attributes to Mental Workload

While most participants are positive about our integration of InstructPipe into the existing user workflow of Visual Blocks, results from the user evaluation also reveal several challenges in such human-AI collaboration. Results in §6.5.1 show that InstructPipe failed to significantly reduce novice users' mental demand. We summarized its major causes into three aspects.

- **Instruction**. P15's comment in §6.5.5 summarizes the first aspect that causes mental burden. Although the "instruction-to-pipeline" process is fast and seems effortless, the process of framing a prompt is one factor that may overwhelm users, especially those who are more accustomed to visual thinking. In essence, an instructor of an LLM system (the user of

InstructPipe) should be clear about the problem they want to solve and preferably what pipeline they want, which causes a mental burden to the user. Zamfirescu-Pereira et al.'s study shows that non-experts may not prompt LLMs well [60], which further explains why non-expert participants might find the instruction process mentally demanding.

- **Perception**. The integration of LLM supports into the visual programming interface enables a "multimodal programming" experience [11], in which, users can program through both verbal and visual approaches. Despite the enhanced flexibility of the system, it also causes additional perceptual burden because users need to *switch their "brain mode" between "visual thinking" and "text-based thinking"*. This finding is aligned with Dual Coding Theory (DCT) [36], in which Paivio hypothesized that human brains process information using two different channels: verbal and visual. Interestingly, our results seemingly contradict with multiple psychological findings based on DCT that show a combination of verbal and visual information actually helps humans' memory process [35, 37]. For example, people feel it easier to remember a new word if they learn the word using a vocabulary card with a figure that explains the texts. Based on these existing theories on humans' mind processes, we make the following explanation on the finding in our project: the cause of mental workload does not imply that people dislike a multimodal workspace, but that *users require a transparent interface that connects humans' mental model to the AI reasoning process, both verbally and visually*. When users frame the instruction in the InstructPipe dialogue, there is a lack of feedback system from the AI assistant that visualizes how AI interprets the instructions in real time. As a result, humans need to either first estimate the AI prediction or make no expectations when they perform instruction. Even for experts, it is hard to accurately predict AI generation, and thus, the generated pipeline is usually unexpected. This causes a sudden increase of cognitive workload because users need to perceive an unexpected pipeline from scratch. That being said, when users frame the instructions, the current interface design is purely based on texts, and thus the visual channel in the user's brain is off. When the generation is complete, the user suddenly needs to switch on a sleeping (visual) channel, which causes a mental workload. One possible approach to addressing this issue is to expose users to the multimodal information *while* they are performing instructions. For example, future work can investigate methods to visualize a generated pipeline on-the-fly when users are framing and typing their instructions. In this manner, users' mental models are continuously synchronized with the AI assistant before the final generation, which should eliminate users' cognitive burdens when the final generation result appears.

- **Debugging**. When a rendered pipeline does not match with users' expectations, users then are required to debug the generated results (see P13's comment §6.5.5). In essence, debugging is one professional programming skill, which explains why it mentally overwhelms beginner-level users. InstructPipe made the first step to facilitate users to 1) perceive the results and 2) take actions on the results, *i.e.*, two important elements in a debugging process, on the node-graph editor. Results in our study reveal that this first step is insufficient to support users so that users still feel like they are debugging the codes as professional programmers. Future work should investigate how to further support users' debugging process with visual programming interfaces. For example, when execution fails, future work can consider visualizing messages that provide cues on the possible issues and provide actionable guidance on the interface for users to fix the issues easily.

On the other hand, it is important to note that we distill the attributes mainly from a subset of participants who explicitly mentioned that they realized an increase in mental workload when using InstructPipe. Those participants typically spent more time writing prompts and working on the
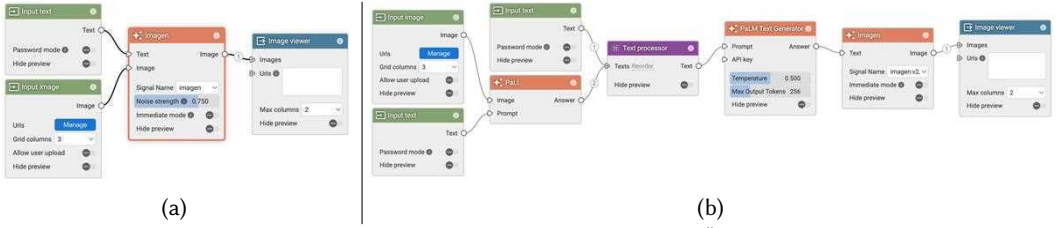
Fig. 12. A comparison of InstructPipe generated by two instructions: (a) *"Edit an image by updating the image caption"*; (b) *"Caption a tiger image using VQA, modify the character in the caption into a cat using LLM, and finally generate a cat image based on the updated caption"*. See Figure 9c for the complete pipeline.

node-graph editor than the average performance (*e.g.*, P15 spent the most time ideating and writing the instructions). Therefore, we argue that the aforementioned issues do not represent the average user experience, and the issues are distilled from explicit comments on their negative experience. In fact, we observed that most participants were relatively decisive and could quickly write their instructions and modify a generated result in the node-graph editor. This observation aligns with our qualitative results in Figure 11: the median score of "Mental Demand" in the InstructPipe condition is lower than the baseline condition.

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

As we explained above, one reason that caused mental workload is that non-experts found it challenging to instruct LLM. More interestingly, we found that *even we, the inventors of InstructPipe, failed to write optimal instructions*. As mentioned in §5.2, two authors annotated captions for the pipelines offline, and we observed multiple imperfections, especially for the complex ones. For instance, the two captions of Figure 9c are *"Describe the image and turn it into a cat image"* and *"Edit an image by updating the image caption"*. Neither caption explicitly describes the detailed pipeline flow clearly, and therefore, all the six evaluation trials (§5) were incomplete (see Figure 12a 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 12b). 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.

The important takeaway is while natural languages are proven to be one promising communication media that connects humans and AI systems [7, 52], *instructions may not be the best format to facilitate such connection*. We believe the reason is that instructions are still not intuitive to humans: AI typically requires flawless and unambiguous instructions while humans tend to express their intention using ambiguous natural languages in conversations. We encourage future work to investigate alternative interaction mediums beyond instructions to further enhance user experience in human-AI collaboration.

7.4   Online InstructPipe

In this project, we made the following assumption: *visual programming and its LLM support are both offline*. Note that InstructPipe does have connection to the Internet, and the "offline" here implies that the system does not dynamically update its node library with online resources: every node is pre-defined by the system developers. This statement shows one major limitation of InstructPipe

we observed in the open-ended session of our user evaluation: the 27 nodes covered by InstructPipe are insufficient to facilitate all the creative instructions from participants. For example, P9 prompted InstructPipe using *"giving me [an] image person walking [a] dog with webcam of myself"*. After some clarification, we found that P9 wanted to generate a video in which P9 is walking a dog by taking the webcam as the input of P10's visual appearance. Such a pipeline is infeasible in our system because it requires a text-driven video generation model. While an extension of InstructPipe with such a text-to-video model as a primitive node can solve this specific issue, we argue that, in practice, users may request many more nodes for their customized uses. Therefore, it is rather important for researchers to investigate a generalizable solution for issues of this category.

In this project, InstructPipe enables users to prototype an ML pipeline with human instruction, and we focus our exploration on a fixed node library. An extension of InstructPipe to a system that covers a wide range of possible *online* functions is still under-explored. With a next generation of InstructPipe that can find ML models, define nodes and implement the node dynamically with human instruction, we believe the aforementioned issues (*e.g.*, the one raised by P10) can be effectively addressed. In the past several years, the community has already established ML libraries and API services for various models (*e.g.*, by Hugging Face [23]). Such ecosystems provide crucial resources for AI-oriented visual programming systems to dynamically define their primitive nodes, and thus provide more powerful support for users. We encourage future work to investigate how to build a model selector (similar to the Node Selector in Figure 4) that can intelligently select the correct online API to be called in a node. We believe such "online InstructPipe" would provide an unprecedented user experience in which researchers can brainstorm with the system, and the system will automatically return a pipeline with state-of-the-art ML models to accelerate researchers' creative workflow.

8   LIMITATIONS

As discussed in the main content, we notice multiple limitations of InstructPipe. In our current implementation, InstructPipe focuses on generating the graph structure in the pipeline (§4.1), and therefore the system is not able to generate the property value of the nodes. In some nodes like "tf-model-runner" that exist in Visual Blocks (but not covered by InstructPipe), such property value plays a critical role that directly defines the functionality of the node. Additionally, when InstructPipe detects an undefined generated node, it directly disposes of the line without leveraging such information for other purposes (§4.4). In the evaluation, we were not able to conduct a large-scale user study with thousands of user study sessions to understand the performance of InstructPipe with both variations from the pipeline factors and the human factors (§5). Our quantitative results show that InstructPipe fails to significantly reduce participants' mental workload (fig. 4). Qualitative results further validate this finding and reveal several issues that cause the mental burden when users use InstructPipe (§7.2).

We also observe two major limitations of the project in addition to those discussed in the main content of the paper. First, our user study participants only engaged InstructPipe in one hour, and therefore, it is unclear whether users will still frequently use and appreciate InstructPipe supports or similar AI assistants in general in a long-term manner. Additionally, all participants are non-experts. Hence, we cannot verify the effectiveness of InstructPipe for users with other levels of expertise. Second, InstructPipe currently cannot detect harmful data or misuse of AI. We believe such a feature is crucial, especially when future work builds "online InstructPipe"(§7.4), which greatly enhances the generalizability of ML pipeline prototyping capability. Future work must study effective methods to eliminate potential harmful uses when AI assistants become increasingly open-ended.

9   CONCLUSION

In this paper, we introduce InstructPipe, an AI assistant that facilitates users to build ML visual programming pipelines with text instructions. We design and implement InstructPipe by decomposing the task into three modules: a node selection module, a code writer, and a code compiler. Results in our technical and system evaluations demonstrate that InstructPipe provides users' satisfactory "on-boarding" experience of visual programming systems and allows them to rapidly prototype an idea. We further discuss the issues we observed concerning LLMs in visual programming, related to both human factors and technical implementations. We hope that InstructPipe will engage a diverse community to easily develop creative machine learning pipelines.

(10,369 words)

REFERENCES

[1] 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[2] Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. 2017.   Neural Module Networks. arXiv:1511.02799 [cs.CV]

[3] Rohan Anil, Andrew M. Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, Siamak Shakeri, Emanuel Taropa, Paige Bailey, Zhifeng Chen, Eric Chu, Jonathan H. Clark, Laurent El Shafey, Yanping Huang, Kathy Meier-Hellstern, Gaurav Mishra, Erica Moreira, Mark Omernick, Kevin Robinson, Sebastian Ruder, Yi Tay, Kefan Xiao, Yuanzhong Xu, Yujing Zhang, Gustavo Hernandez Abrego, Junwhan Ahn, Jacob Austin, Paul Barham, Jan Botha, James Bradbury, Siddhartha Brahma, Kevin Brooks, Michele Catasta, Yong Cheng, Colin Cherry, Christopher A. Choquette-Choo, Aakanksha Chowdhery, Clément Crepy, Shachi Dave, Mostafa Dehghani, Sunipa Dev, Jacob Devlin, Mark Díaz, Nan Du, Ethan Dyer, Vlad Feinberg, Fangxiaoyu Feng, Vlad Fienber, Markus Freitag, Xavier Garcia, Sebastian Gehrmann, Lucas Gonzalez, Guy Gur-Ari, Steven Hand, Hadi Hashemi, Le Hou, Joshua Howland, Andrea Hu, Jeffrey Hui, Jeremy Hurwitz, Michael Isard, Abe Ittycheriah, Matthew Jagielski, Wenhao Jia, Kathleen Kenealy, Maxim Krikun, Sneha Kudugunta, Chang Lan, Katherine Lee, Benjamin Lee, Eric Li, Music Li, Wei Li, YaGuang Li, Jian Li, Hyeontaek Lim, Hanzhao Lin, Zhongtao Liu, Frederick Liu, Marcello Maggioni, Aroma Mahendru, Joshua Maynez, Vedant Misra, Maysam Moussalem, Zachary Nado, John Nham, Eric Ni, Andrew Nystrom, Alicia Parrish, Marie Pellat, Martin Polacek, Alex Polozov, Reiner Pope, Siyuan Qiao, Emily Reif, Bryan Richter, Parker Riley, Alex Castro Ros, Aurko Roy, Brennan Saeta, Rajkumar Samuel, Renee Shelby, Ambrose Slone, Daniel Smilkov, David R. So, Daniel Sohn, Simon Tokumine, Dasha Valter, Vijay Vasudevan, Kiran Vodrahalli, Xuezhi Wang, Pidong Wang, Zirui Wang, Tao Wang, John Wieting, Yuhuai Wu, Kelvin Xu, Yunhan Xu, Linting Xue, Pengcheng Yin, Jiahui Yu, Qiao Zhang, Steven Zheng, Ce Zheng, Weikang Zhou, Denny Zhou, Slav Petrov, and Yonghui Wu. 2023. PaLM 2 Technical Report. arXiv:2305.10403 [cs.CL]

[4] Stanislaw Antol, Aishwarya Agrawal, Jiasen Lu, Margaret Mitchell, Dhruv Batra, C. Lawrence Zitnick, and Devi Parikh. 2015. VQA: Visual Question Answering. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*.

[5] Tim Brooks, Aleksander Holynski, and Alexei A. Efros. 2023. InstructPix2Pix: Learning to Follow Image Editing Instructions. arXiv:2211.09800 [cs.CV]

[6] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language Models are Few-shot Learners. *Advances in Neural Information Processing Systems* 33 (2020), 1877–1901. https://doi.org/10.48550/arXiv.2005.14165

[7] Weihao Chen, Chun Yu, Huadong Wang, Zheng Wang, Lichen Yang, Yukun Wang, Weinan Shi, and Yuanchun Shi. 2023. From Gap to Synergy: Enhancing Contextual Understanding through Human-Machine Collaboration in Personalized Systems. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology* (San Francisco, CA, USA) *(UIST '23)*. Association for Computing Machinery, New York, NY, USA, Article 110, 15 pages. https://doi.org/10.1145/3586183.3606741

[8] ComfyUI. 2023. ComfyUI. https://github.com/comfyanonymous/ComfyUI

[9] Chandan Datta, Chandimal Jayawardena, I Han Kuo, and Bruce A MacDonald. 2012. RoboStudio: A visual programming environment for rapid authoring and customization of complex services on a personal service robot. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2352–2357. https://doi.org/10.1109/IROS.2012.6386105

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805 [cs.CL]

[11] Griffin Dietz, Nadin Tamer, Carina Ly, Jimmy K Le, and James A. Landay. 2023. Visual StoryCoder: A Multimodal Programming Environment for Children's Creation of Stories. In *Proceedings of the 2023 CHI Conference on Human*

Factors in Computing Systems (Hamburg, Germany) *(CHI '23)*. Association for Computing Machinery, New York, NY, USA, Article 96, 16 pages. https://doi.org/10.1145/3544548.3580981

[12] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, Lei Li, and Zhifang Sui. 2023. A Survey on In-context Learning. arXiv:2301.00234 [cs.CL]

[13] Ruofei Du, Na Li, Jing Jin, Michelle Carney, Scott Miles, Maria Kleiner, Xiuxiu Yuan, Yinda Zhang, Anuva Kulkarni, Xingyu Liu, Ahmed Sabie, Sergio Orts-Escolano, Abhishek Kar, Ping Yu, Ram Iyengar, Adarsh Kowdle, and Alex Olwal. 2023. Rapsai: Accelerating Machine Learning Prototyping of Multimedia Applications Through Visual Programming. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems* (Hamburg, Germany) *(CHI '23)*. Association for Computing Machinery, New York, NY, USA, Article 125, 23 pages. https://doi.org/10.1145/3544548.3581338

[14] Ruofei Du, Na Li, Jing Jin, Michelle Carney, Xiuxiu Yuan, Kristen Wright, Mark Sherwood, Jason Mayes, Lin Chen, Jun Jiang, Jingtao Zhou, Zhongyi Zhou, Ping Yu, Adarsh Kowdle, Ram Iyengar, and Alex Olwal. 2023. Experiencing Visual Blocks for ML: Visual Prototyping of AI Pipelines. In *Adjunct Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology (UIST)*. ACM. https://doi.org/10.1145/3586182.3615817

[15] GitHub. 2023. GitHub Copilot · Your AI pair programmer. https://github.com/features/copilot

[16] Tanmay Gupta and Aniruddha Kembhavi. 2023. Visual Programming: Compositional Visual Reasoning Without Training. In *2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. arXiv. https://doi.org/10.48550/arXiv.2211.11559

[17] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2

[18] Sandra G Hart and Lowell E Staveland. 1988. Development of NASA-TLX (Task Load Index): Results of empirical and theoretical research. In *Advances in psychology*. Vol. 52. Elsevier, 139–183.

[19] Thomas T Hewett. 2005. Informing the design of computer-based environments to support creativity. *International Journal of Human-Computer Studies* 63, 4-5 (2005), 383–409.

[20] Yushi Hu, Otilia Stretcu, Chun-Ta Lu, Krishnamurthy Viswanathan, Kenji Hata, Enming Luo, Ranjay Krishna, and Ariel Fuxman. 2023. Visual Program Distillation: Distilling Tools and Programmatic Reasoning into Vision-Language Models. arXiv:2312.03052 [cs.CV]

[21] Justin Huang and Maya Cakmak. 2017. Code3: A system for end-to-end programming of mobile manipulator robots for novices and experts. In *Proceedings of the 2017 ACM/IEEE International Conference on Human-Robot Interaction*. 453–462.

[22] Justin Huang, Tessa Lau, and Maya Cakmak. 2016. Design and evaluation of a rapid programming system for service robots. In *2016 11th ACM/IEEE International Conference on Human-Robot Interaction (HRI)*. IEEE, 295–302.

[23] HuggingFace. 2022. Spaces. https://huggingface.co/docs/transformers/preprocessing

[24] Peiling Jiang. 2023. Positional Control in Node-Based Programming. In *Extended Abstracts of the 2023 CHI Conference on Human Factors in Computing Systems* (Hamburg, Germany) *(CHI EA '23)*. Association for Computing Machinery, New York, NY, USA, Article 231, 7 pages. https://doi.org/10.1145/3544549.3585878

[25] Peiling Jiang, Jude Rayan, Steven P Dow, and Haijun Xia. 2023. Graphologue: Exploring Large Language Model Responses with Interactive Diagrams. *arXiv preprint arXiv:2305.11473* (2023).

[26] Jeffrey Kodosky. 2020. LabVIEW. *Proc. ACM Program. Lang.* 4, HOPL, Article 78 (jun 2020), 54 pages. https://doi.org/10.1145/3386328

[27] Anastasia Kovalkov, Avi Segal, and Kobi Gal. 2020. Inferring Creativity in Visual Programming Environments. In *Proceedings of the Seventh ACM Conference on Learning @ Scale* (Virtual Event, USA) *(L@S '20)*. Association for Computing Machinery, New York, NY, USA, 269–272. https://doi.org/10.1145/3386527.3406725

[28] Tiffany H Kung, Morgan Cheatham, Arielle Medenilla, Czarina Sillos, Lorie De Leon, Camille Elepaño, Maria Madriaga, Rimel Aggabao, Giezel Diaz-Candido, James Maningo, et al. 2023. Performance of ChatGPT on USMLE: Potential for AI-assisted medical education using large language models. *PLoS digital health* 2, 2 (2023), e0000198.

[29] LangFlow. 2023. LangFlow. https://github.com/logspace-ai/langflow

[30] Yang Li and James A. Landay. 2005. Informal Prototyping of Continuous Graphical Interactions by Demonstration. In *Proceedings of the 18th Annual ACM Symposium on User Interface Software and Technology* (Seattle, WA, USA) *(UIST '05)*. Association for Computing Machinery, New York, NY, USA, 221–230. https://doi.org/10.1145/1095034.1095071

[31] Xiwen Liang, Yangxin Wu, Jianhua Han, Hang Xu, Chunjing XU, and Xiaodan Liang. 2022. Effective Adaptation in Multi-Task Co-Training for Unified Autonomous Driving. In *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), Vol. 35. Curran Associates, Inc., 19645–19658. https://proceedings.neurips.cc/paper_files/paper/2022/file/7c319b62e2257b34cb0e1040ced2e007-Paper-Conference.pdf

[32] Xingyu Liu, Vladimir Kirilyuk, Xiuxiu Yuan, Alex Olwal, Peggy Chi, Xiang Chen, and Ruofei Du. 2023. Visual Captions: Augmenting Verbal Communication With On-the-fly Visuals. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems (CHI)*. ACM, 20 pages. https://doi.org/10.1145/3544548.3581566

[33] Zixian Ma, Jerry Hong, Mustafa Omer Gul, Mona Gandhi, Irena Gao, and Ranjay Krishna. 2023. CREPE: Can Vision-Language Foundation Models Reason Compositionally?. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 10910–10921.

[34] B. A. Myers. 1986. Visual Programming, Programming by Example, and Program Visualization: A Taxonomy. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Boston, Massachusetts, USA) *(CHI '86)*. Association for Computing Machinery, New York, NY, USA, 59–66. https://doi.org/10.1145/22627.22349

[35] Allan Paivio. 1969. Mental imagery in associative learning and memory. *Psychological review* 76, 3 (1969), 241.

[36] Allan Paivio. 1991. Dual coding theory: Retrospect and current status. *Canadian Journal of Psychology/Revue canadienne de psychologie* 45, 3 (1991), 255.

[37] Allan Paivio, James M Clark, et al. 2006. Dual coding theory and education. *Pathways to literacy achievement for high poverty children* (2006), 1–20.

[38] Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. 2023. Generative Agents: Interactive Simulacra of Human Behavior. arXiv:2304.03442 [cs.HC]

[39] Joon Sung Park, Lindsay Popowski, Carrie Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. 2022. Social Simulacra: Creating Populated Prototypes for Social Computing Systems. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology* (Bend, OR, USA) *(UIST '22)*. Association for Computing Machinery, New York, NY, USA, Article 74, 18 pages. https://doi.org/10.1145/3526113.3545616

[40] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[41] Zhenhui Peng, Xingbo Wang, Qiushi Han, Junkai Zhu, Xiaojuan Ma, and Huamin Qu. 2023. Storyfier: Exploring Vocabulary Learning Support with Text Generation Models. arXiv:2308.03864 [cs.HC]

[42] Leixian Shen, Enya Shen, Yuyu Luo, Xiaocong Yang, Xuming Hu, Xiongshuai Zhang, Zhiwei Tai, and Jianmin Wang. 2023. Towards Natural Language Interfaces for Data Visualization: A Survey. *IEEE Transactions on Visualization and Computer Graphics* 29, 6 (2023), 3121–3144. https://doi.org/10.1109/TVCG.2022.3148007

[43] Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. 2023. Progprompt: Generating situated robot task plans using large language models. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 11523–11530.

[44] Karan Singhal, Shekoofeh Azizi, Tao Tu, S. Sara Mahdavi, Jason Wei, Hyung Won Chung, Nathan Scales, Ajay Tanwani, Heather Cole-Lewis, Stephen Pfohl, Perry Payne, Martin Seneviratne, Paul Gamble, Chris Kelly, Nathaneal Scharli, Aakanksha Chowdhery, Philip Mansfield, Blaise Aguera y Arcas, Dale Webster, Greg S. Corrado, Yossi Matias, Katherine Chou, Juraj Gottweis, Nenad Tomasev, Yun Liu, Alvin Rajkomar, Joelle Barral, Christopher Semturs, Alan Karthikesalingam, and Vivek Natarajan. 2022. Large Language Models Encode Clinical Knowledge. arXiv:2212.13138 [cs.CL]

[45] Zefan Sramek, Arissa J. Sato, Zhongyi Zhou, Simo Hosio, and Koji Yatani. 2023. SoundTraveller: Exploring Abstraction and Entanglement in Timbre Creation Interfaces for Synthesizers. In *Proceedings of the 2023 ACM Designing Interactive Systems Conference* (Pittsburgh, PA, USA) *(DIS '23)*. Association for Computing Machinery, New York, NY, USA, 95–114. https://doi.org/10.1145/3563657.3596089

[46] Sangho Suh, Bryan Min, Srishti Palani, and Haijun Xia. 2023. Sensecape: Enabling Multilevel Exploration and Sensemaking with Large Language Models. *arXiv preprint arXiv:2305.11483* (2023).

[47] Dídac Surís, Sachit Menon, and Carl Vondrick. 2023. Vipergpt: Visual inference via python execution for reasoning. *arXiv preprint arXiv:2303.08128* (2023).

[48] William Robert Sutherland. 1966. *The on-line graphical specification of computer procedures*. Ph.D. Dissertation. Massachusetts Institute of Technology.

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

[50] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. *Advances in Neural Information Processing Systems* 30 (2017). https://doi.org/10.5555/3295222.3295349

[51] Vijay Viswanathan, Chenyang Zhao, Amanda Bertsch, Tongshuang Wu, and Graham Neubig. 2023. Prompt2Model: Generating Deployable Models from Natural Language Instructions. arXiv:2308.12261 [cs.CL]

[52] Bryan Wang, Gang Li, and Yang Li. 2023. Enabling Conversational Interaction With Mobile UI Using Large Language Models. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems* (Hamburg, Germany) *(CHI '23)*. Association for Computing Machinery, New York, NY, USA, Article 432, 17 pages. https://doi.org/10.1145/3544548.

3580895

[53] Yun Wang, Zhitao Hou, Leixian Shen, Tongshuang Wu, Jiaqi Wang, He Huang, Haidong Zhang, and Dongmei Zhang. 2023. Towards Natural Language-Based Visualization Authoring. *IEEE Transactions on Visualization and Computer Graphics* 29, 1 (2023), 1222–1232. https://doi.org/10.1109/TVCG.2022.3209357

[54] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. arXiv:2201.11903 [cs.CL]

[55] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2019. Huggingface's Transformers: State-of-the-Art Natural Language Processing. *ArXiv Preprint ArXiv:1910.03771* (2019). https://arxiv.org/pdf/1910.03771

[56] Tongshuang Wu, Ellen Jiang, Aaron Donsbach, Jeff Gray, Alejandra Molina, Michael Terry, and Carrie J Cai. 2022. PromptChainer: Chaining Large Language Model Prompts through Visual Programming. In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA) *(CHI EA '22)*. Association for Computing Machinery, New York, NY, USA, Article 359, 10 pages. https://doi.org/10.1145/3491101.3519729

[57] Tongshuang Wu, Marco Tulio Ribeiro, Jeffrey Heer, and Daniel S. Weld. 2021. Polyjuice: Generating Counterfactuals for Explaining, Evaluating, and Improving Models. arXiv:2101.00288 [cs.CL]

[58] Tongshuang Wu, Michael Terry, and Carrie Cai. 2022. AI Chains: Transparent and Controllable Human-AI Interaction by Chaining Large Language Model Prompts. In *CHI Conference on Human Factors in Computing Systems*. ACM. https://doi.org/10.1145/3491102.3517582

[59] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. arXiv:2210.03629 [cs.CL]

[60] J.D. Zamfirescu-Pereira, Richmond Y. Wong, Bjoern Hartmann, and Qian Yang. 2023. Why Johnny Can't Prompt: How Non-AI Experts Try (and Fail) to Design LLM Prompts. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems* (Hamburg, Germany) *(CHI '23)*. Association for Computing Machinery, New York, NY, USA, Article 437, 21 pages. https://doi.org/10.1145/3544548.3581388

[61] Beiqi Zhang, Peng Liang, Xiyu Zhou, Aakash Ahmad, and Muhammad Waseem. 2023. Practices and Challenges of Using GitHub Copilot: An Empirical Study. In *Proceedings of the 35th International Conference on Software Engineering and Knowledge Engineering (SEKE2023)*. KSI Research Inc. https://doi.org/10.18293/seke2023-077

[62] Lei Zhang and Steve Oney. 2020. FlowMatic: An Immersive Authoring Tool for Creating Interactive Scenes in Virtual Reality. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology* (Virtual Event, USA) *(UIST '20)*. Association for Computing Machinery, New York, NY, USA, 342–353. https://doi.org/10.1145/3379337.3415824

[63] Yongqi Zhang, Cuong Nguyen, Rubaiat Habib Kazi, and Lap-Fai Yu. 2023. PoseVEC: Authoring Adaptive Pose-aware Effects Using Visual Programming and Demonstrations. In *ACM Symposium on User Interface Software and Technology*.

[64] Zhenyu Zhang, Xuxi Chen, Tianlong Chen, and Zhangyang Wang. 2021. Efficient lottery ticket finding: Less data is more. In *International Conference on Machine Learning*. PMLR, 12380–12390.

[65] Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, et al. 2022. Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625* (2022).

[66] Zhongyi Zhou and Koji Yatani. 2022. Gesture-Aware Interactive Machine Teaching with In-Situ Object Annotations. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology* (Bend, OR, USA) *(UIST '22)*. Association for Computing Machinery, New York, NY, USA, Article 27, 14 pages. https://doi.org/10.1145/3526113.3545648

APPENDIX

A A FULL LIST OF 27 NODES IN INSTRUCTPIPE

The following content shows 27 nodes InstructPipe covers in the generation process and their corresponding short description used in the Node Selector (§4.2):

A.1 Input Nodes

(1) **live_camera**: Capture video stream through your device camera.
(2) **input_image**: Select images to use as input to your pipeline. You can also upload your own images.
(3) **input_text**: Add text to use as input to your pipeline.

A.2 Output Nodes

(1) **image_viewer**: View images.
(2) **threed_photo**: Create a 3D photo effect from depthmap tensors.
(3) **markdown_viewer**: Render Markdown strings into stylized HTML.
(4) **html_viewer**: Show HTML content with styles

A.3 Processor Nodes

(1) **google_search**: Use Google to search the web that returns a list of URLs based on a given keyword; usually selected with string_picker.
(2) **body_segmentation**: Segment out people in images; usually selected with mask_visualizer.
(3) **tensor_to_depthmap**: Display tensor data as a depth map.
(4) **portrait_depth**: Generate a 3D depth map for an image; usually selected with tensor_to_depthmap, threed_photo.
(5) **face_landmark**: Detect faces in images. Each face contains 468 keypoints; usually selected with landmark_visualizer, virtual_sticker.
(6) **pose_landmark**: Generate body positional mappings for people detected in images; usually selected with landmark_visualizer.
(7) **image_processor**: Process an image (crop, resize, shear, rotate, change brightness or contrast, add blur or noise).
(8) **text_processor**: Reformat and combine multiple text inputs.
(9) **mask_visualizer**: Visualize masks.
(10) **string_picker**: Select one string from a list of strings; usually used with google_search.
(11) **image_mixer**: Combine images and text into one output image. Requires two image inputs.
(12) **virtual_sticker**: Use face landmarks data to overlay virtual stickers on images.
(13) **palm_textgen**: Generate Text using a large language model.
(14) **keywords_to_image**: Search for images by keywords.
(15) **url_to_html**: Crawl the website by a given URL.
(16) **image_to_text**: Extract text from an image using OCR service.
(17) **pali**: Answer questions about an image using a vision-language model.
(18) **palm_model**: Generate text using a large language model based on prompt and context.
(19) **imagen**: Generate an image based on a text prompt.
(20) **input_sheet**: Read string data from Google Sheets.

B SYSTEM PROMPTS USED IN LLM MODULES

Here we provide more details about the prompts we utilized in InstructPipe. The original txt files are also attached in the supplementary zip file.

```
{
  "nodeSpecId": "body_segmentation",
  "description": "Segment out people in images.",
  "category": "processor",
  "inputSpecs": {
    "image": {
      "type": "image"
    }
  },
  "outputSpecs": {
    "segmentationResult": {
      "type": "masks",
      "recommendedNodes": [
        "mask_visualizer"
      ]
    }
  },
  "examples": [
    "live_camera_xhjtec:
live_camera();\nbody_segmentation_xctd1p_out =
body_segmentation_xctd1p:
body_segmentation(image=live_camera_xhjtec);\nmask_visualizer_frjz
ga_out = mask_visualizer_frjzga:
mask_visualizer(image=live_camera_xhjtec,
segmentationResult=body_segmentation_xctd1p_out);\n"
  ]
}
```

```
{
  "nodeSpecId": "pali",
  "description": "Answer questions about an image using a
vision-language model.",
  "category": "processor",
  "inputSpecs": {
    "image": {
      "type": "image"
    },
    "prompt": {
      "type": "string"
    }
  },
  "outputSpecs": {
    "answer": {
      "type": "string"
    }
  },
  "examples": [
    "input_image_f1ohfa: input_image();\ninput_text_04ejnm:
input_text(text=\"What is the person in the image
doing?\");\npali_2pzuwn_out = pali_2pzuwn:
pali(image=input_image_f1ohfa,
prompt=input_text_04ejnm);\nmarkdown_viewer_6wqe86:
markdown_viewer(markdownString=pali_2pzuwn_out);\n"
  ]
}
```

(a) Body segmentation   (b) PaLI

Fig. 13. Examples of node configuration used in Code Writer. The configuration is structured in a JSON format.

B.1 Node Selector

Please see our supplementary file (node_select.txt) for the full prompt we use in this stage.

B.2 Code Writer

The prompt in Code Writer is dynamic, which is dependent on the nodes chosen in Node Selector. Therefore, we cannot provide all the possible prompts in the supplementary materials. Here, we will focus on providing examples of two detailed node configurations utilized in InstructPipe. Figure 7 shows the structure of the prompt utilized in this LLM stage. Figure 13 provides two examples of node configurations (*i.e.*, "Body segmentation" and "PaLI") that InstructPipe may chose into the highlighted line(s). Each configuration includes keys of "nodeSpecId" (*i.e.*, node types), "description", "category" and "examples". For those nodes that support input and output edges, "inputSpecs" and "outputSpecs" specify the sockets and their corresponding valid data types. For example, the output socket name of "Body segmentation" is "segmentationResult", and its data type is "masks". Some nodes (*e.g.*, "Body segmentation") include recommended node(s) (*e.g.*, "Mask visualizer" for "Body segmentation"), and our configuration also contains such information in the dictionary.
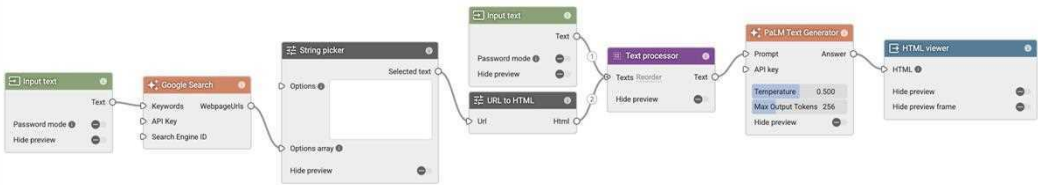
C USER STUDY PIPELINES

Fig. 14. Text-based pipeline. The "String picker" node provides users a drop-down menus to select one URL from a list of URLs returned by "Google Search". "PaLM Text Generator" is an LLM used to summarize the full HTML page.
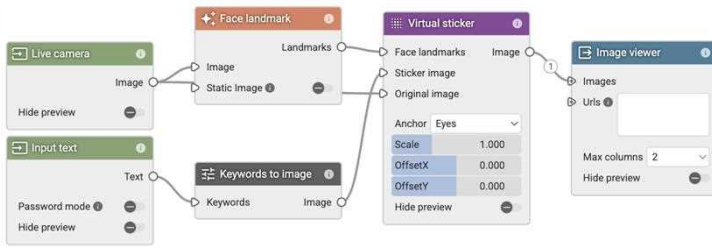
Fig. 15. Real-time multimodal pipeline. The "Keyword to image" node is used to search sunglasses image, and the "Virtual sticker" node anchors the sunglasses onto users' face.

Figure 14 and Figure 15 visualize two pipelines we required the participants to complete in our user study. Figure 15 is a multimodal pipeline that allows participants to interact with AR effects in real time. Our technical evaluation shows that InstructPipe can generate this pipeline accurately: the averaged ratio of human interactions = 5.2%. Figure 15 is a text-based pipeline that provides participants with a summary of the news searched from Google. The technical evaluation reveals that InstructPipe cannot generate this pipeline accurately: the averaged ratio of human interactions = 27.8%.

Note that even though InstructPipe may be able to complete the pipeline structure in Figure 15 from users' instruction, we observed that participants still need to fine-tune their keywords to get an ideal pair of sunglasses. Additionally, the default anchor value is "Face top", so participants need to use the drop-down menu on the "Virtual sticker" node to change the value to "Eyes". This further motivates us to use the metric of "Time" in addition to the number of user interactions in our study. Our demo video also covers the workflows of these two pipelines.

D   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.