InstructPipe: Generating Visual Blocks Pipelines with Human Instructions and LLMs

Zhongyi Zhou
Google
Tokyo, Japan
The University of Tokyo
Tokyo, Japan
zhongyi.zhou.work@gmail.com

Jing Jin
Google
Mountain View, CA, USA
jingjin@google.com

Vrushank Phadnis
Google
Mountain View, CA, USA
vrushankphadnis@gmail.com

Xiuxiu Yuan
Google
Mountain View, CA, USA
xiuxiuyuan@google.com

Jun Jiang
Google
Sunnyvale, CA, USA
junjiang@google.com

Xun Qian
Google
Mountain View, CA, USA
xunqian@google.com

Kristen Wright
Google
Mountain View, CA, USA
kristenwright@google.com

Mark Sherwood
Google
Mountain View, CA, USA
marksherwood@google.com

Jason Mayes
Google
Mountain View, CA, USA
jasonmayes@google.com

Jingtao Zhou
Google
Mountain View, CA, USA
jingtaozhou@google.com

Yiyi Huang
Google
Sunnyvale, CA, USA
yiyih@google.com

Zheng Xu
Google
Seattle, WA, USA
xuzheng@google.com

Yinda Zhang
Google
Mountain View, CA, USA
yindaz@google.com

Johnny Lee
Google
Redmond, WA, USA
johnnylee@google.com

Alex Olwal
Google
Mountain View, CA, USA
olwal@acm.org

David Kim
Google
Zurich, Switzerland
kidavid@google.com

Ram Iyengar
Google
Mountain View, CA, USA
tenheadedram@gmail.com

Na Li
Google
Palo Alto, CA, USA
linazhao@google.com

Ruofei Du*
Google
San Francisco, CA, USA
ruofei@google.com

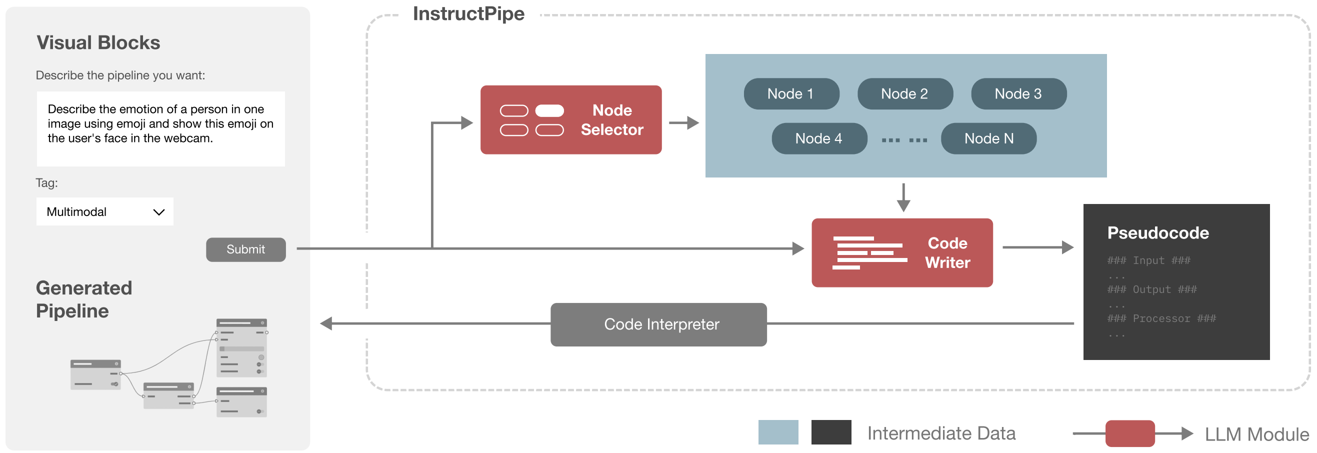


Figure 1: Workflow of InstructPipe. First, users describe their desired pipeline in natural language and designate it with a language, image, or multi-modal tag. InstructPipe then feeds user instructions into a node selector to identify a relevant set of nodes. Subsequently, both the instructions and 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.

*Corresponding author: me [at] duruofei [dot] com; Also contact: zhongyi.zhou.work [at] gmail [dot] com

CHI '25, Yokohama, Japan

ACM ISBN 979-8-4007-1394-1/25/04
https://doi.org/10.1145/3706598.3713905

Abstract

Visual programming has the potential of providing novice programmers with a low-code experience to build customized processing pipelines. Existing systems typically require users to build pipelines from scratch, implying that novice users are expected to set up and link appropriate nodes from a blank workspace. In this paper, we introduce InstructPipe, an AI assistant for prototyping machine learning (ML) pipelines with text instructions. We contribute two large language model (LLM) modules and a code interpreter as part of our framework. The LLM modules generate pseudocode for a target pipeline, and the interpreter renders the pipeline in the node-graph editor for further human-AI collaboration. Both technical and user evaluation (N=16) shows that InstructPipe empowers users to streamline their ML pipeline workflow, reduce their learning curve, and leverage open-ended commands to spark innovative ideas.

CCS Concepts

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

Keywords

Visual Programming; Large Language Models; Visual Prototyping; Node-graph Editor; Graph Compiler; Low-code Development; Deep Neural Networks; Deep Learning; Visual Analytics

ACM Reference Format:
Zhongyi Zhou, Jing Jin, Vrushank Phadnis, Xiuxiu Yuan, Jun Jiang, Xun Qian, Kristen Wright, Mark Sherwood, Jason Mayes, Jingtao Zhou, Yiyi Huang, Zheng Xu, Yinda Zhang, Johnny Lee, Alex Olwal, David Kim, Ram Iyengar, Na Li, and Ruofei Du. 2025. InstructPipe: Generating Visual Blocks Pipelines with Human Instructions and LLMs. In *CHI Conference on Human Factors in Computing Systems (CHI '25), April 26–May 01, 2025, Yokohama, Japan.* ACM, New York, NY, USA, 22 pages. https://doi.org/10.1145/3706598.3713905

1 Introduction

A *visual programming* interface provides users with a node-graph editor to program through interaction with visual elements. As opposed to writing code in a code editor, the node graph allows users to design pipelines by configuring nodes and connecting them with edges in a visual workspace. This alternative user interface approach often accelerates experimentation and exploration in the prototyping phases of creative applications, and can make advanced technology more accessible to beginners. Advances in machine learning (ML) further stimulate growing interest in visual programming. Open-source ML hubs (*e.g.*, TF-Hub [1], PyTorch-Hub [57], and Hugging Face [83]) contribute large numbers of encapsulated modules that accelerate AI project development and experimentation, and such libraries provide important resources for an ML-based visual programming platform. Recent advancements in large language models (LLMs) [3, 8, 77] and findings on Chain-of-Thought [81] have further stimulated community-wide interest in visual programming [4, 19, 84, 86], suggesting further potential in the interactive exploration of AI chains.

Despite the development of visual programming platforms in various domains, we observed that existing systems share one similar characteristic: users usually initiate 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 within *a completely empty workspace*. As was also highlighted in existing literature in programming tools [92, 95], such processes can easily overwhelm users, especially those who are unfamiliar with a particular visual programming platform. Providing pipeline templates may reduce on-boarding efforts [9, 21], but the templates inherently lack flexibility and are not easily adaptable to users' specific needs. Similar issues also arise 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 [23] enables 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 [12, 38]. To this end, we raise the following question that motivates our work: *How can we build visual programming assistants to accelerate the design and prototyping of ML pipelines?*

This paper introduces InstructPipe, a visual programming AI assistant that enables ML pipeline generation and design through natural language instructions. InstructPipe facilitates node connection and selection, allowing users to focus on more creative tasks like parameter tuning and interactive analysis within the visual programming workspace. We focus our AI assistant exploration on ML-based pipelines, and therefore implemented InstructPipe as an extension to Visual Blocks [18], a visual programming system for prototyping ML pipelines. One major technical challenge in implementing InstructPipe lies in the lack of visual programming data, making it impractical to finetune a dedicated code-LLM similar to how developers build text-editor-based copilots [12, 23, 38]. We addressed this issue by decomposing the generation process into three steps (Figure 1). InstructPipe's first LLM module scopes the potentially useful nodes, while the second LLM module generates pseudocode for a pipeline. InstructPipe then parses the pseudocode and renders the pipeline in the node-graph editor to facilitate further user interaction. Our technical evaluation suggests that InstructPipe reduces the necessary user interactions by 81.1% when users select and connect nodes, compared to building them from scratch. This can potentially streamline the development process, and allows users to focus on more novice-friendly interactions like parameter-tuning and human-in-the-loop verification. 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 concepts for various purposes. In our experiments, we also observed new challenges caused by human cognitive characteristics, and proposed future technical directions towards open-ended AI prototyping assistants.

In summary, we contribute:

(1) InstructPipe, a visual programming AI assistant that enables users to generate ML pipelines from human instructions by automating node selection and connection,

(2) System design and technical development of InstructPipe. The system consists of two LLM modules and a code interpreter, which generate the specification for the visual programming pipeline, compile the code, and render the pipeline in an interactive node-graph editor,
(3) Technical and user evaluations that characterize the effectiveness of InstructPipe, and contribute 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"* [73]. 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 [39], Unity Graph Editor [76], PromptChainer [84], ComfyUI [13] and Visual Blocks [18]) typically feature a node graph editor, providing users with a visual workspace to "write" their program using *"building blocks"* [28, 68, 89]. Recent work further explored the application of visual programming in education [9, 35, 40], XR creativity support [88, 91, 93], and robotics [14, 30, 31]. For example, Zhang *et al.* [93] connected the visual programming tool to the concept of *teaching by demonstration* [44, 49, 99], allowing users to rapidly customize AR effects in video creation. FlowMatic [91] extended traditional visual programming interfaces into 3D virtual environments, providing users with immersive authoring experiences.

Advancements in AI have introduced many repositories of advanced ML models [33, 66], and an increasing number of researchers are exploring AI chains [41, 86]. This progress has motivated HCI researchers to design and build a range of visual programming interfaces to support the AI development process [13, 43, 84]. For example, ChainForge is a web-based platform for developers to explore various LLM-related configuration and designs in a wide range of applications [4]. Visual Blocks enables creation and interaction of advanced ML pipelines that can leverage state-of-the-art computer vision and computer graphics models in the browser [18].

This work contributes the technical system, implementation and evaluation of a novel AI assistant that enables the use of text-based instructions in visual programming of ML pipelines. Compared to typical workflows in which people manually build their pipelines, InstructPipe has the potential to accelerate ML pipeline prototyping in visual programming.

2.2 Interactive Systems with LLMs

The advances in LLMs bring many research directions for HCI researchers. Researchers have started designing new LLM interfaces, to advance beyond the currently dominant chatbot interface (*e.g.*, OpenAI ChatGPT, Google Gemini). For example, Graphologue [36] augmented LLM responses with interactive diagrams that visualize response texts in a structured format. Sensecape [71] provides users with a workspace to explore long LLM responses in a hierarchical structure.

Many HCI researchers integrated LLMs in conventional interactive systems and demonstrated that such enhanced machine intelligence can provide new user experiences [20, 46, 56, 60, 78]. This research principle is widely applied in many downstream HCI applications, including visualization [65, 80], explainable AI [79, 85], and social science [45, 55]. For example, Chen et al. [11] utilized LLMs to bridge low-level sensor information with high-level human requests. Experiments showed that such connection allows users to *"construct their personalized contexts [for an intelligent system] more quickly, accurately, and naturally"*. To interface human intention with machine operations, researchers typically utilized LLMs by following the ReAct (reasoning and acting) paradigm [87]. For example, Park et al. [55] simulated human behaviors in an artificial social system by leveraging LLMs as intelligent agents that perceive the environment, plan their behaviors, and act in the environment. Automated Visualization (AutoViz) researchers employ LLMs for data analysis and reasoning for presenting the visualization [48, 50, 63]. For example, LIDA features four modules in the visualization pipeline to 1) summarize a structured dataset, 2) explore the user's goal, 3) generate code for visualization, and 4) render visualization [16]. ChartGPT further constructs a dedicated dataset for chart visualization, and finetunes an LLM for fully automating the data visualization pipeline [74].

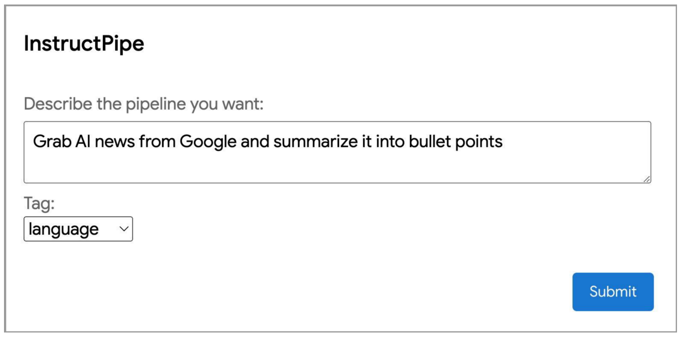
InstructPipe extends the application of ReAct-like LLM frameworks [16, 74] to visual programming and demonstrates its effectiveness to support rapid prototyping with lower user workload. Additionally, introducing visual programming to the ReAct framework showcases an interface solution for human-AI collaboration. That being said, our work values partially correct AI generation, though the previous literature considers it as a complete generation failure [16, 24]. We leverage visual programming as a platform to integrate partially complete AI generations with human interactions, enabling even novices to intuitively collaborate with AI in their creative processes.

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 [18, 98], a visual programming system for prototyping ML pipelines.

3.1 User Workflow

To generate a pipeline, users first click the "InstructPipe" button in the top-right corner of the interface (Figure 2b). The system then activates a simple dialog (Figure 2a) in which users provide a description and a tag for their desired pipeline. The tag can be "language", "visual", or "multimodal", and helps guide the pipeline generation. After users click the "Submit" button, InstructPipe generates a visual pipeline in the node-graph editor. More specifically, InstructPipe generates a directed acyclic graph (DAG) of a visual programming pipeline. This implies that it uses default node parameters (e.g., the "temperature" or "max_tokens" value of an LLM node). Therefore, after the generation, the user needs to 1) finish the graphic structure if necessary, and 2) perform parameter tuning as well as human-in-the-loop evaluation of the pipeline quality interactively in the visual programming platform.

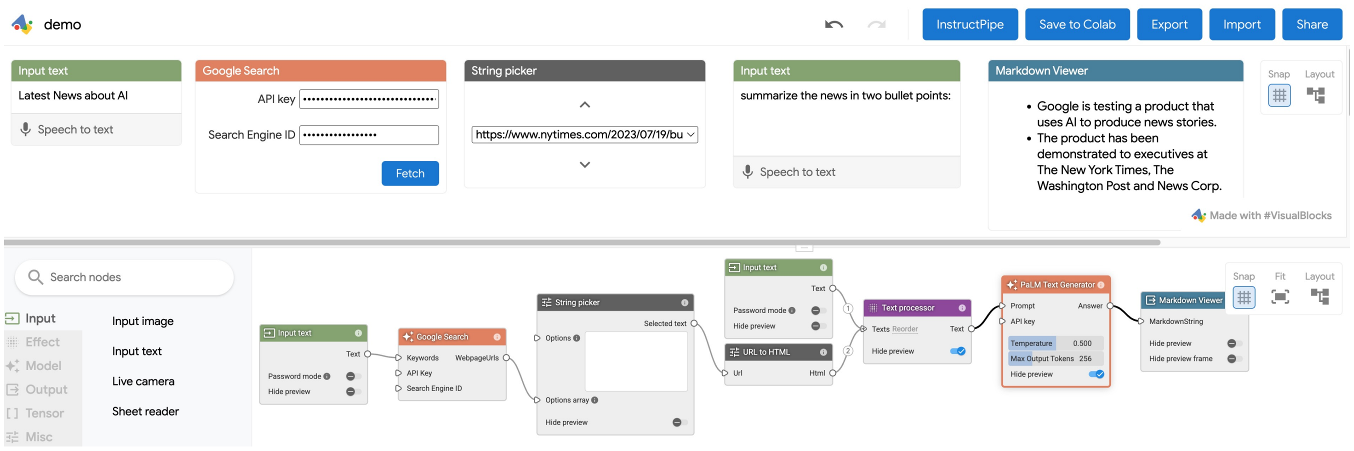

InstructPipe

Describe the pipeline you want:

Grab AI news from Google and summarize it into bullet points

Tag: language

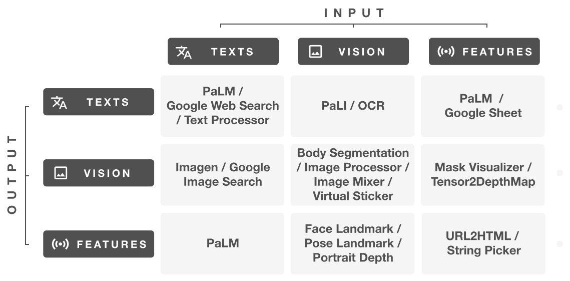
(a) InstructPipe's instruction dialog.



- Google is testing a product that uses AI to produce news stories.
- The product has been demonstrated to executives at The New York Times, The Washington Post and News Corp.

(b) InstructPipe's visual programming interface.

Figure 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.


INPUT
OUTPUT

	TEXTS	VISION	FEATURES
TEXTS	PaLM / Google Web Search / Text Processor	PaLI / OCR	PaLM / Google Sheet
VISION	Imagen / Google Image Search	Body Segmentation / Image Processor / Image Mixer / Virtual Sticker	Mask Visualizer / Tensor2DepthMap
FEATURES	PaLM	Face Landmark / Pose Landmark / Portrait Depth	URL2HTML / String Picker

Figure 3: The distribution of 20 primitive processor nodes supported by InstructPipe. Note that "PaLM" represents two nodes in InstructPipe, *i.e.*, a text generation model and a chat model of PaLM [3].

As we will show in our evaluation, this new human-AI collaboration approach reduces users' workload on the technical portion of the visual programming tasks (selecting and connecting nodes) and thus provides a more novice-friendly experience for technical visual programming platforms.

3.2 Primitive Nodes

InstructPipe supports 27 primitive nodes in Visual Blocks. We achieved this node library of InstructPipe by filtering out nodes without explicit definition of their functions[1]. For example, 'TFLite model runner' is an implicitly defined node: the user needs to input a tensorflow hub link to define its functionality. As we mentioned previously, InstructPipe focuses on generating a DAG and leaves the parameter-tuning task to users. Adding such implicit nodes without a clear definition of the functionality can easily confuse our AI assistant in the generation process, and thus, we decide to exclude these nodes in the node library of InstructPipe.

The 27 nodes in our library include three input nodes, four output nodes and 20 processor nodes. The following shows an example node in each category, and we leave the full node library description in Appendix A:

- **"live camera" (an input node)**: Capture video stream through your device camera
- **"markdown viewer" (an output node)**: Render Markdown strings into stylized HTML.

[1]Note that Visual Blocks is a system that is actively being updated, and there are more nodes now.

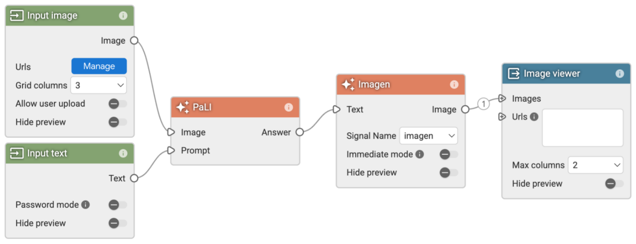
- **"imagen" (a processor node)**: Generate an image based on a text prompt.

We distributed 20 processor nodes based on the data type of its I/O edges and visualized it in Figure 3. For example. "Google Web Search" takes "Texts" information as input and output new "Texts", and "OCR" takes an image (vision-based information) as input and output "Texts". "Features" in Figure 3 indicates a wide range of intermediate data formats used in ML pipelines, including segmentation masks, pose landmarks, URLs and etc. As shown in the matrix, InstructPipe contains a wide range of nodes that support the creation of complex ML pipelines. Compared to related work that automates ad hoc ML inferences in specific use scenarios [24, 72], InstructPipe makes one more step towards the open-ended assistants with a more diverse set of primitive nodes. Further extending our node library can effectively empower the capability of our AI assistant, which we leave as critical future work. In the current implementation of InstructPipe, we focus on demonstrating its capability based on our focused node library and explore what new experiences this AI assistant can bring to our users.

4 Pipeline Generation from Instructions

InstructPipe leverages LLMs to generate visual programming pipelines from text instructions. There are two prevailing approaches for LLM-customization, fine-tuning [46, 62], and few-shot prompting [24, 55]. Fine-tuning would require a substantial volume of annotated data, with pairs of pipelines and prompts, and it is hard to achieve for a specific visual programming platform. Additionally, a growing list of nodes would consistently require 1) new data annotation and 2) retraining the model, making this approach less sustainable. In comparison, few-shot prompting is a more practical approach for prototyping an interaction concept to understand the new experience it would bring to the community [24, 81, 87]. One major challenge of applying LLMs in visual programming AI assistants lies in designing efficient prompts that fit within a reasonable number of tokens. Even though we focus our exploration on 27 nodes, the node configuration file alone includes 8200 tokens. Further formulating pipeline examples as in-context few-shot examples would result in a combinatorial explosion, causing an overwhelming number of tokens in the prompt.

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 1 illustrates the high-level workflow of the InstructPipe implementation. InstructPipe leverages two LLM modules (highlighted in red); a *Node Selector* (section 4.2), and a *Code Writer* (section 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 descriptions and examples of each selected node to ensure the LLM has extensive context for 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.



Grid columns 3
Allow user upload
Hide preview
Password mode
Hide preview
Signal Name imagen
Immediate mode
Hide preview
Max columns 2
Hide preview

(a) Pipeline.

```
###Input###
input_image_1: input_image();
input_text_1: input_text(text="caption this image in detail");

###Output###
image_viewer_1: image_viewer(images=imagen_1_out);

###Processor###
pali_1_out = pali_1: pali(image=input_image_1,
prompt=input_text_1);
imagen_1_out = imagen_1: imagen(text=pali_1_out);
```

(b) Pseudocode.

Figure 4: 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.

4.1 Pipeline Representation

The Visual Blocks system takes JSON-format data as input and renders a directed acyclic graph (DAG) in the visual programming workspace. Therefore, the ultimate goal of InstructPipe is to generate the JSON file; however, directly generating the long JSON file is computationally expensive. For example, the JSON file for rendering the pipeline in Figure 4a contains approximately 2.8k tokens. To address this issue, we utilize the pseudocode representation of a DAG, and define this token-efficient data format as the output data format of our LLM module. Figure 4b shows the corresponding pseudocode representation of the pipeline in Figure 4a, and the it only contains 123 tokens. The pseudocode representation simply stores the DAG information of a visual programming pipeline without other information such as node parameters (*e.g.*, the "max_tokens" configuration of an LLM module) and the layouts of the nodes. This indicates that InstructPipe leaves the task of node parameter tuning to the user, which we believe is a more novice-friendly task, and focuses on providing technical assistance on selecting and connecting nodes.

In the following content, we provide detailed explanation on the pseudocode design and implementation. As we mentioned above, Figure 4 provides an example of a pipeline (Figure 4a) and its corresponding pseudocode (Figure 4b). The syntax design is inspired by TypeScript, and the overall structure is inspired by how academic papers present pseudocode [94] in an algorithm block. In

```
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
```

Figure 5: 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.

Figure 4b, we highlight the first line under the processor module (*i.e.*, the operation of the PaLI node) in different colors, representing four different components in the programming language. "pali_1" is the unique *node ID*. The green symbol after the colon, *i.e.*, "pali", specifies the *node type*. In this example, node ID "*pali_1*" is a "pali" node. The arguments in brackets, *i.e.*, "image=input_image_1, prompt=input_text_1", specifies the input variables (or input edges in the graph) of this node. "pali_1_out" represents the *output variable name*. For input nodes, the output variable name is the same as the *node id*, so we do not annotate the output variable with a separate name (*e.g.*, "input_image_1: input_image()" instead of "input_image_1 = input_image_1: input_image()"). Note that InstructPipe generates texts (*i.e.*, the node parameter) 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 parameter input as a special case.

4.2 Node Selector

Node Selector filters out unrelated nodes by providing the LLM with a short description of each node. Figure 5 shows the prompt we use in Node Selector. Followed by a general task description and guidelines, we list all node types with a short description that explains the function of each. Several nodes come with recommendation(s) when the users interact with Visual Blocks, and we also include such node recommendations in the prompt. The main intuition of this prompt design is based on how existing

```
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 pseudocode.

Guidelines:
1. Respond solely in pseudocode, 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 LLM2Pipeline

Examples:
Q:
{'description': 'generate a photo and validate whether it is real or
generated.', 'tag': 'multimodal'}
A:
... // pipeline pseudocode

... // more in-context examples
```

Figure 6: The prompt structure for the Code Writer module. Detailed node configurations, see the appendix for examples, are listed in the highlighted region.

open-source libraries (*e.g.*, Numpy [25]) 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&As as few-shot examples to support the LLM to learn and adapt to the context of the task.

4.3 Code Writer

With a pool of selected nodes, the Code Writer module can write pipeline rendering pseudocode. Figure 6 shows the structure of the prompt utilized in this LLM module. Similar to section 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 pseudocode, for how this node connects to other nodes. We include a detailed explanation of the full node configuration in Section B.1.2. Similar to the previous LLM module (section 4.2), the prompt design here is also inspired by the documentation of existing software libraries. Specifically, we gain inspiration from low-level function-specific documentation[3], which typically includes 1) a detailed description, 2) data types in

[2]See an example in the following link: https://numpy.org/doc/1.25/reference/routines.array-manipulation.html

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

the input/output, followed by 3) one or more examples of a few lines of code for how developers can use the function.

The prompt also includes a Q&A list 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 options, and it is impossible to design a dedicated list of few-shot examples in each possible case. Therefore, we only created an example list for each pipeline tag (*i.e.*, "language", "visual", and "multimodal") and intended to utilize these few-shot pipelines to teach LLMs example use cases in each category. This implies that in-context pipelines may include nodes that were not selected for the prompt. This can potentially lead to LLM hallucinations [32], *i.e.*, utilizing the nodes that do not exist in our node library. We mitigated this issue by adding specific prompts that explicitly show a list of supported nodes (*i.e.*, the contents start with "the following is a full list of ..." in Figure 6). However, LLM hallucination is a community-wide challenge, and we also find that our approach cannot eliminate this issue in visual programming. Therefore, InstructPipe conducts a sanity check for the Code Writer outputs and directly disposes of the line of pseudocode with such hallucinated nodes. This can ensure that the generated code is in a valid data format for rendering the pipeline in Visual Blocks.

4.4 Code Interpreter

After our LLM modules generate the pseudocode, InstructPipe employs a code interpreter to parse the generated pseudocode and compile a JSON-formatted pipeline with an automatic layout. Since we incorporated standard approaches to achieving such conversion from the pseudocode to the JSON file, which we do not intend to claim as our main contributions, we briefly summarize our implementation into the following three steps for simplicity and elaborate low-level implementation details at Appendix B.2:

1. **Lexical Analysis**: InstructPipe first tokenizes each line of the pseudocode into output variable id, node id, node type, and node arguments (section 4.1).
2. **Graph Generation with Default Node Parameters**: We generated a DAG based on the tokenized results and applied predefined default node parameters in each generated node. 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. **Layout Optimation**: When pseudocode is converted into a JSON file, default node parameters will cause sub-optimal visual effects (Figure 11a). InstructPipe conducts a layout optimization process using the breadth-first search (BFS) algorithm, which re-arranges the layout for better presentation of the pipeline (Figure 11b).

5 Technical Evaluation

InstructPipe contributes a framework for generating specifications for visual programming pipelines based on text prompts from users. To characterize the system's performance, we designed a technical evaluation to assess the accuracy of the generated graphs for a variety of prompts.

5.1 Data Collection

To compute the accuracy of our generated pipelines, we need to collect a corpus with pairs of instructions and their corresponding ground-truth pipelines. Therefore, we organized a two-day hybrid workshop with 23 participants, aiming to collect real pipelines that Visual Blocks users would build for their creative usage. The 23 participants (F: 6; M: 17) are composed of five software engineers, four research scientists, four students, three designers, two project managers, and two engineering managers. Six attendees claimed that they had prior experience in using Visual Blocks. As this was a data collection study rather than a user study, where each participant here served as a data creator and annotator, we did not restrict participation to individuals who self-identified as novices. The workshop began with a 15-minute Visual Blocks tutorial walking the participants through the nodes and the pipeline-building process. After the tutorial, attendees created pipelines independently. Once they finished creating the pipelines, participants were required to caption their pipelines and upload them. We utilized this corpus of data pairs (caption/pipeline) as the data set for the technical evaluation.

The workshop was an open-ended creation process in which participants were free to use any node available in Visual Blocks with more than the 27 nodes covered by InstructPipe. *The InstructPipe feature was not available in the workshop.* After the workshop, we post-processed our collected data and achieved 48 pipelines (23 language pipelines, seven visual pipelines and 18 multi-modal pipelines) for our technical evaluations. The post-processing procedure details are presented in Appendix C.1.

5.2 Metric: The Number of User Interactions

To quantify the efficacy of InstructPipe based on our goal of accelerating and streamlining pipeline creation, we defined the metric *Number of User Interactions* as follows:

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

This definition is mainly inspired by Graph Edit Distance (GED) in graph theory [22]. Note that there are countless ways to modify a generated pipeline toward a complete pipeline in practice. Nevertheless, the **minimal** number of user interactions is deterministic, and this is an objective metric that can fairly estimate the amount of effort 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 across two types of events: 1) adding/deleting a node, and 2) adding/deleting an edge between nodes. In the technical evaluation, we report the average *ratio* of user interactions required to complete a pipeline "from our generated pipeline" compared to "from scratch" as our target metric. For example, if it takes 3 interactions to complete a pipeline from our generated results and takes 10 interactions to complete from scratch, then the ratio of interactions is 30%. Appendix C.2 contains further discussion of this metric.

5.3 Experiment Setups and Results

We ran our generation algorithm on the pipeline captions six times (three times for each caption × two captions for each pipeline), and

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%

computed an averaged performance among the six trials for each pipeline.

Table 1 summarizes the results of the technical evaluation. Compared to building a pipeline from scratch, InstructPipe allows the user to complete a pipeline with **18.9%** of the user interactions, demonstrating the potential of InstructPipe to require more than 5X fewer interactions. **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. Therefore, we conducted an in-person user study of InstructPipe with another group of participants, aiming to provide more insights into our system performance as well as explore new user experiences brought by InstructPipe. The study recruitment was in accordance with the ethics board of Google. We obtained participant consent before the study began.

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 7 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 7).

We carefully designed the experiment to create a fair study that could be completed with reasonable effort. In the following content, we elaborate on how we made two important decisions related to the study's rigor:

6.1.1 ***Two controlled pipelines with full counterbalancing***. Our user evaluation focuses on two controlled pipelines with full counterbalancing. While we acknowledge that more pipelines (*e.g.*, four, six, or more) could enhance generalizability, such designs would also inevitably increase the size of the required user groups, even without fully counterbalancing. For example, fully counterbalancing four controlled pipelines requires 12× more participants. Partially counterbalancing with four pipelines using the Latin Square design still requires us to double the number of participants. Additionally, novice participants are likely to progressively gain experiences within the study, and such learning effects will weaken the design of partial counterbalancing. We believe that two pipelines with full counterbalancing are a reasonable experiment setup in this work, and future work could consider extending and scaling up these experiments.

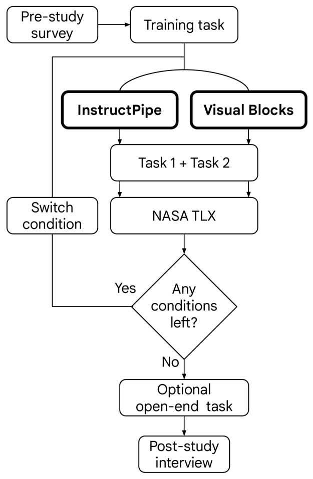


Figure 7: A flow diagram of the user study. After a training session, participants completed the two tasks in each condition in the sequence determined by the counterbalancing protocol.

6.1.2 ***Pipeline selection***. Given the fixed number of pipelines we can evaluate with users and the potential bias introduced by few-shot prompts [96], it is important how we select the two pipelines for user study. There are two critical factors that we considered: representativeness and diversity. Representativeness implies that the selected pipelines should represent the average performance of InstructPipe. Diversity suggests that the selected pipelines should provide various experiences to simulate the actual use scenarios in which the performance of InstructPipe may vary. Following this guideline, we 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 resulting 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 Section D.2 for more detail on the pipelines.

6.2 Procedure

Each study session takes 55 - 65 minutes in total. The study started with 10-15 minutes of hands-on training for 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 will need to use for pipeline creation in the main session. Participants were also encouraged to experiment with building a pipeline independently, and to ask questions.

After the training, participants progressed to a formal study session where they were asked to build and complete pipelines under the given conditions. We verbally described the pipelines to participants as below, and participants could not see 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.

A pipeline is considered complete when the aforementioned functions run in the user's visual programming workspace. For example, we consider the "real-time multi-model pipeline" as complete when the pipeline registers the sunglasses on the user's face, with real-time tracking and following of the head movement.

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 many more hints in the baseline condition, and we made this decision to ensure every novice-level participant can finish their tasks within a reasonable amount of time. Appendix D.3 contains more details and discussions of the assistance we provided in the study. As an optional extension to the study, eight participants were offered an open-ended pipeline creation, where participants prototyped their own ideas using 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 conducting all pipeline-condition combinations, participants answered open-ended questions in a semi-structured interview. The interview script is available in the appendix D.1. Participants provided their general impression of each condition, listed pros and cons, identified potential future use cases, and critiqued the user interface for future improvements. We transcribed the interviews and conducted the open coding analysis on the qualitative data [69, 70]. More specifically, we categorized the quotes based on our observations and then refined the code for presentation.

6.3 Participants

We recruited 16 participants from our internal participant pool, which is specifically designed for UX research within our institution. Importantly, none of the participants was involved in our project, and the authors in charge of the study did not personally know any of the participants. We screened participants on their self-reported programming experience and machine learning skills. All of the 16 selected participants rated their "Programming Experience" and "'Machine Learning Skill' as "Intermediate" or below (See Table 3 for a full breakdown). We intentionally recruited novice users, as we envision them as intended users of InstructPipe.

6.4 Metrics

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

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

6.4.2 The Number of User Interactions. We used the number of user interactions (introduced in section 5.2) to measure the user's objective workload. Unlike the results in section 5.3, 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 [26]. This questionnaire was a subset of the NASA-TLX (part I). Participants filled out the questionnaire after each condition (InstructPipe or Visual Blocks).

Table 2: Task completion time and the number of human interactions in the user study (N=16). We use $***$ to denote $p < .001$.

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

6.5 Results

6.5.1 ***InstructPipe Reduces Users' Workload.*** Table 2 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 8 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 except "Mental Demand" (see section 7.2 for more explanations and discussion). Furthermore, the test indicates that all participants unanimously considered that 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 the potential for InstructPipe to dramatically 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 visual programming systems, which validates our statements and motivation in section 1.

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

P1's comment matches 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. Typical support included 1) guiding participants if they went 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]

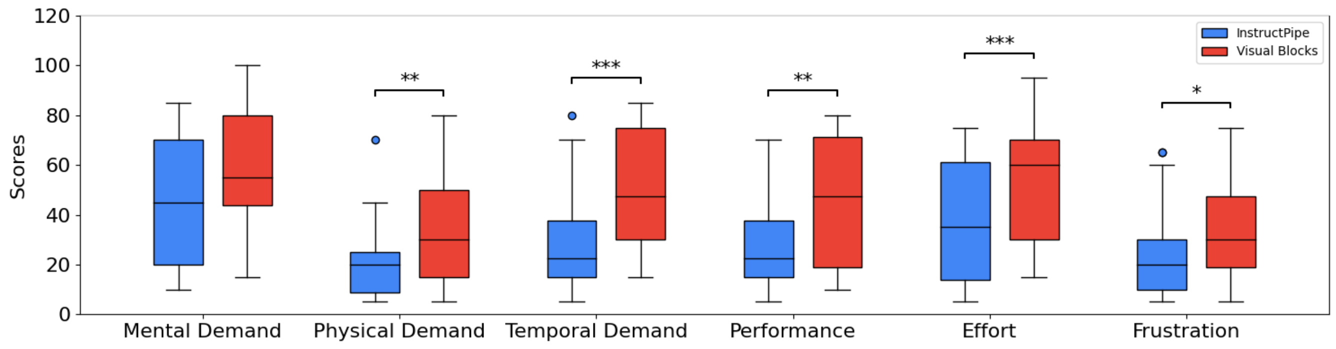


Figure 8: Raw-TLX results. The statistic significance is annotated with *, **, or *** (representing p<.05, p<.01, and p<.001, respectively).

> *"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:

> *"[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, like P6, appreciated the integration of the AI assistant into the standard visual programming workflow, P15 expressed a concern about this approach. In visual programming, users typically rely on visual thinking to construct pipelines, but the new prompt-based method introduces a shift toward text-based reasoning. This blend of cognitive processes could potentially increase users' mental workload:

> *" [the participant is talking about s/he wants to fix an unsuccessful generation by changing the prompt instead of performing visual programming here] ... 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 in the pipeline ... [this is very overwhelming]"* [P15]

6.5.4 ***Use Scenarios: Accessible ML Prototyping and Education***. In the open-ended session, we observed that 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) is going to facilitate prototype building for PMs (Product Managers) ... 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:

> *"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 suggest that InstructPipe led participants to engage in deeper, contemplative thought.

The observation suggests that participants needed time to perceive the generated pipelines as they appeared in the workspace. Such sense-making processes bring new challenges to the 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 others. P15 spent the most time writing the prompt. The following comments provide insights into how the prompting process caused extra mental workload:

> *"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

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

Our technical evaluation (section 5.3) shows that InstructPipe reduces the number of user interactions to 18.9 % (±20.3%) There are two key 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 early-stage findings* that show LLMs can achieve full automation of ML inference [24, 72]. The main reason is that existing work built their ad hoc solutions for target use scenarios, respectively. In contrast, InstructPipe covers a larger range of ML models (section 3.2) and aims for an open-ended use case. Our results show that LLMs (we used GPT-3.5-turbo in the study) still fail to write robust code with prompt engineering techniques. This aligns with the latest research findings that show that even the latest LLMs still fail to formulate a whole working pipeline [62, 82].

While LLMs cannot generate a fully executable pipeline, our work shows that AI can successfully render a certain portion of a pipeline for users. Both technical and user evaluations highlight the important values here. We believe such values provide useful takeaways for HCI researchers to explore more human-AI collaboration approaches and designs in visual programming.

7.2 Three Attributes to Mental Workload

Results in section 6.5.1 show that InstructPipe failed to significantly reduce novice users' mental demand. We summarized its major causes into three aspects.

7.2.1 ***Instruction.*** P15's comment in section 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. InstructPipe requires its users to 1) be clear about the problem they want to solve, and 2) be able to clearly articulate the desired pipeline. Such requirements cause a mental burden to the user [6]. We believe that our results can reinforce the existing knowledge on how non-experts may not prompt LLMs well [51, 90] in the visual programming domain.

7.2.2 ***Perception.*** The integration of LLM support into the visual programming interface enables a "multimodal programming" experience [17], in which, users can program through both verbal and visual approaches. However, this flexibility increases perceptual burden as users switch between visual and text-based thinking [53]. Interestingly, our results seemingly contradict psychological findings based on the Dual Coding Theory (DCT) that show a combination of verbal and visual information actually helps humans' memory process[4] [52, 54]. Therefore, we believe that the mental workload stems not from dislike of multimodal workspaces, but from the lack of a transparent interface that aligns users' mental models with AI reasoning both verbally and visually. That being said, a next-generation copilot should visualize a pipeline (*i.e.*, visual info) while the user is prompting the system (*i.e.*, verbal info), constituting and interfacing multimodal processing in humans' brains.

7.2.3 ***Debugging.*** When a rendered pipeline does not match users' expectations, users need to debug (see P13's comment section 6.5.5). Specifically, users need to *"invest extra effort to review and understand the generated content"* [95] and then solve the issues caused by the AI assistant. In essence, debugging is a professional programming skill, which understandably can be mentally overwhelming for beginner-level users. While InstructPipe visualizes generated code in the visual programming platform, our results suggest that future systems should provide more guidance for beginners to better proceed with their programming tasks.

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

As we discussed above, non-experts found it challenging to instruct LLM. More interestingly, we found that *even we, the inventors of InstructPipe, failed to write optimal instructions.* For instance, the two captions of Figure 13c 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 (section 5) were incomplete (see Figure 9a 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 9b). 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 [11, 78], *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 intentions using ambiguous natural languages in conversations. We encourage future work to investigate alternative interaction

[4]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.

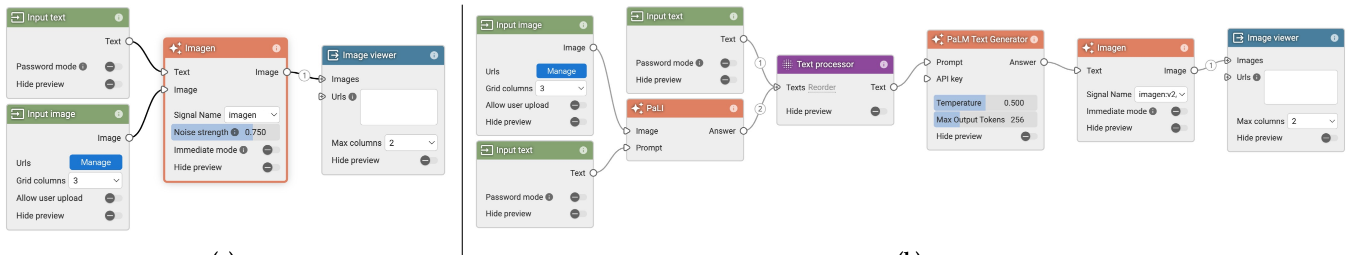


Figure 9: A comparison of InstructPipe generated by two instructions: (a) *"Edit an image by updating the image caption"*; (b) *"Caption a tiger image using VQA, modify the character in the caption into a cat using LLM, and finally generate a cat image based on the updated caption"*. See Figure 13c for the complete pipeline.

mediums beyond instructions to further enhance user experience in human-AI collaboration.

8 Limitations and Future Work

8.1 Assisting Humans to Prompt AI Copilot in Visual Programming

InstructPipe introduces a novel user interaction technique for visual programming, along with its set of challenges – prompting AI is not easy [90]. While the latest research has explored prompt writing assistants [7, 42, 47], creating such assistants in visual programming poses unique challenges, as discussed in section 7.2, and requires further dedicated investigation. Despite these challenges, the visual programming workspace offers a unique opportunity – it provides an interactive and visual medium for delivering AI-generated information. For example, a prompt writing assistant could provide "a pipeline preview" in real time via a lightweight LLM. Visualizing estimated outcomes, such as unexpected pipeline results (as illustrated in Figure 9a), highlights the potential of these tools to guide users in refining their instructions effectively.

8.2 Node Parameter Tuning

InstructPipe focuses on generating the graph structure in the pipeline (section 4.1), and InstructPipe is not able to generate node parameters. The latest research in AI agents shows great potential for distributing a systematic task among multiple LLMs and letting them solve the problem collaboratively [37]. We encourage future work to extend such distributed AI agent approaches to generate suitable node parameters to further reduce users' workload in tuning them.

8.3 A Larger and Dynamic Node Library

InstructPipe is an AI assistant prototype on a small-scale library with 27 nodes. Similar to other tool-calling LLM systems [15, 64], InstructPipe cannot generate any out-of-scope node, and thus, there is a limited scope of pipelines that InstructPipe can generate. Future work should investigate a scale-up problem by creating an assistant that supports large-scale nodes [61]. What new technical challenge will emerge? Will such a large-scale library provide practical human value? If yes, what are the concrete new user experience it opens up in visual programming?

Additionally, future work should explore a dynamic solution of the node library, in which an AI assistant can define necessary nodes in visual programming on the fly. HuggingGPT [66] is a pioneering project that shares a similar vision as this goal, but existing investigations show that the accuracy of such open-ended generation is still unsatisfactory [58, 62]. How can we design an interface to bridge such imperfect AI and human users in visual programming copilot? What will be the interaction paradigm in an interactive system with a dynamic node library?

8.4 Refining System Component Design

InstructPipe provides a system contribution, and we verified the usefulness of InstructPipe via two evaluations that assess InstructPipe as a whole system. One important future direction would be to verify (or even challenge) each technical component of our system, as elaborated below:

Pseudocode. We designed the pseudocode order based on how algorithm papers present their algorithm blocks. Is this design the best approach among all the possible candidates? If not, how can we further improve the design of pseudocode language?

Prompt Design. We leveraged the in-context learning capability of LLMs in our prompt design. Prior work shows that few-shot examples cause bias effects in practice [97], and thus, we encourage future work to mitigate this bias by collecting a large dataset and finetuning LLM on the dataset.

Divide-and-conquer at Scale. We adopt the strategy of divide-and-conquer [67] with a two-stage LLM pipeline. Despite its effectiveness with a small node library and simple graphs, its effectiveness is unknown when generating complex graphs. Exploring agent-based approaches [29, 34] with Retrieval Augmented Generation (RAG) would be a promising future direction to manage complex graph generation in a divide-and-conquer manner [67]. We encourage future work to contribute high-quality datasets as well as an interactive LLM system with RAG that provides users with better experiences from AI agents.

8.5 Evaluation Metrics and Long-term Evaluation

In the technical evaluation, we assessed the performance of AI assistants based on the number of user interactions. Existing related metrics, predominantly from the code synthesis literature [2, 27], largely focuses on two categories: correctness-based metrics [5, 10] that rely on test case verification, and similarity-based metrics [75]. *Very little research* falls outside these two categories [59]. Our metric incorporates human factors by objectively estimating user effort through graph theory, addressing a gap in the visual programming literature where human-centric considerations are crucial. While our work advances metric development in this domain, further formal research is essential to establish comprehensive standards for visual programming evaluation.

In the user evaluation, we conducted a lab study to understand the user experience of InstructPipe. As future work, we plan to conduct longer-term studies and gather more realistic insights than those we obtained from the lab study. This is critical for us to understand the long-term usefulness of our assistant for beginners, as well as collect feedback to inform our system design.

8.6 Responsible AI

InstructPipe currently cannot detect harmful data or misuse of AI. We believe such safety features are crucial, especially in the context of the potential for future dynamic node libraries, which would greatly enhance 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

This paper introduces InstructPipe, an AI assistant that empowers users to accelerate their design of ML visual programming pipelines using 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 suggest that InstructPipe provides users' satisfactory "on-boarding" experience of visual programming systems and allows them to rapidly prototype an idea, potentially with more than 5X fewer interactions. 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 inspire the community to continue work in accelerated human-AI collaboration for increased expressivity and creativity, for machine learning pipelines, and beyond.

Acknowledgments

We would like to thank Gong Xuan, Fengyuan Zhu, Kevin Zhang and Karl Rosenberg for their feedback and discussion on our early-stage prototype, as well as Koji Yatani and Takeo Igarashi for the feedback on the paper draft. We also thank our reviewers for their insightful feedback.

References

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

[2] Aida Amini, Saadia Gabriel, Peter Lin, Rik Koncel-Kedziorski, Yejin Choi, and Hannaneh Hajishirzi. 2019. Mathqa: Towards interpretable math word problem solving with operation-based formalisms. *arXiv preprint arXiv:1905.13319* (2019).

[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] Ian Arawjo, Chelse Swoopes, Priyan Vaithilingam, Martin Wattenberg, and Elena L. Glassman. 2024. ChainForge: A Visual Toolkit for Prompt Engineering and LLM Hypothesis Testing. In *Proceedings of the CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '24)*. Association for Computing Machinery, New York, NY, USA, Article 304, 18 pages. doi:10.1145/3613904.3642016

[5] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).

[6] Alan F Blackwell. 1996. Metacognitive theories of visual programming: what do we think we are doing?. In *Proceedings 1996 IEEE symposium on visual languages*. IEEE, 240–246.

[7] Stephen Brade, Bryan Wang, Mauricio Sousa, Sageev Oore, and Tovi Grossman. 2023. Promptify: Text-to-Image Generation through Interactive Prompt Exploration with Large Language Models. 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 96, 14 pages. doi:10.1145/3586183.3606725

[8] 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. doi:10.48550/arXiv.2005.14165

[9] Liuqing Chen, Shuhong Xiao, Yunnong Chen, Yaxuan Song, Ruoyu Wu, and Lingyun Sun. 2024. ChatScratch: An AI-Augmented System Toward Autonomous Visual Programming Learning for Children Aged 6-12. In *Proceedings of the CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '24)*. Association for Computing Machinery, New York, NY, USA, Article 649, 19 pages. doi:10.1145/3613904.3642229

[10] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[11] 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. doi:10.1145/3586183.3606741

[12] Shrestha Basu Mallick Chris Perry. 2023. *AI-powered coding, free of charge with Colab.* Retrieved Sep 10, 2024 from https://blog.google/technology/developers/google-colab-ai-coding-features/

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

[14] 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. doi:10.1109/IROS.2012.6386105

[15] Fernanda De La Torre, Cathy Mengying Fang, Han Huang, Andrzej Banburski-Fahey, Judith Amores Fernandez, and Jaron Lanier. 2024. LLMR: Real-time Prompting of Interactive Worlds using Large Language Models. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '24)*. Association for Computing Machinery, New York, NY, USA, Article 600, 22 pages. doi:10.1145/3613904.3642579

[16] Victor Dibia. 2023. LIDA: A Tool for Automatic Generation of Grammar-Agnostic Visualizations and Infographics using Large Language Models. arXiv:2303.02927 [cs.AI] https://arxiv.org/abs/2303.02927

[17] 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. doi:10.1145/3544548.3580981

[18] 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. doi:10.1145/3544548.3581338

[19] 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. doi:10.1145/3586182.3615817

[20] K. J. Kevin Feng, Q. Vera Liao, Ziang Xiao, Jennifer Wortman Vaughan, Amy X. Zhang, and David W. McDonald. 2024. Canvil: Designerly Adaptation for LLM-Powered User Experiences. arXiv:2401.09051 [cs.HC] https://arxiv.org/abs/2401.09051

[21] James Fogarty, Jodi Forlizzi, and Scott E. Hudson. 2001. Aesthetic information collages: generating decorative displays that contain information. In *Proceedings of the 14th Annual ACM Symposium on User Interface Software and Technology* (Orlando, Florida) *(UIST '01)*. Association for Computing Machinery, New York, NY, USA, 141–150. doi:10.1145/502348.502369

[22] Xinbo Gao, Bing Xiao, Dacheng Tao, and Xuelong Li. 2010. A survey of graph edit distance. *Pattern Analysis and applications* 13 (2010), 113–129.

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

[24] 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. doi:10.48550/arXiv.2211.11559

[25] 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. doi:10.1038/s41586-020-2649-2

[26] 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.

[27] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. 2021. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938* (2021).

[28] 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.

[29] Dong Huang, Qingwen Bu, Jie M Zhang, Michael Luck, and Heming Cui. 2023. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010* (2023).

[30] 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.

[31] 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.

[32] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, and Ting Liu. 2023. A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions. arXiv:2311.05232 [cs.CL] https://arxiv.org/abs/2311.05232

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

[34] Md Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 2024. MapCoder: Multi-Agent Code Generation for Competitive Problem Solving. *arXiv preprint arXiv:2405.11403* (2024).

[35] 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. doi:10.1145/3544549.3585878

[36] 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).

[37] Haolin Jin, Linghan Huang, Haipeng Cai, Jun Yan, Bo Li, and Huaming Chen. 2024. From LLMs to LLM-based Agents for Software Engineering: A Survey of Current, Challenges and Future. arXiv:2408.02479 [cs.SE] https://arxiv.org/abs/2408.02479

[38] Jupyter-ai. 2024. *Jupyter AI*. Retrieved Sep 10, 2024 from https://github.com/jupyterlab/jupyter-ai

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

[40] 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. doi:10.1145/3386527.3406725

[41] LangChain. 2023. LangChain. https://www.langchain.com/

[42] LangChain. 2024. *Promptim: an experimental library for prompt optimization*. Retrieved Nov 26, 2024 from https://blog.langchain.dev/promptim/

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

[44] 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. doi:10.1145/1095034.1095071

[45] Ruibo Liu, Ruixin Yang, Chenyan Jia, Ge Zhang, Denny Zhou, Andrew M Dai, Diyi Yang, and Soroush Vosoughi. 2023. Training socially aligned language models on simulated social interactions. *arXiv preprint arXiv:2305.16960* (2023).

[46] 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. doi:10.1145/3544548.3581566

[47] Qianou Ma, Hua Shen, Kenneth Koedinger, and Sherry Tongshuang Wu. 2024. How to teach programming in the ai era? using llms as a teachable agent for debugging. In *International Conference on Artificial Intelligence in Education*. Springer, 265–279.

[48] Rishab Mitra, Arpit Narechania, Alex Endert, and John Stasko. 2022. Facilitating Conversational Interaction in Natural Language Interfaces for Visualization. In *2022 IEEE Visualization Conference (VIS)*. IEEE. doi:10.48550/arXiv.2207.00189

[49] 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. doi:10.1145/22627.22349

[50] Arpit Narechania, Arjun Srinivasan, and John Stasko. 2021. NL4DV: A Toolkit for Generating Analytic Specifications for Data Visualization from Natural Language Queries. *IEEE Transactions on Visualization and Computer Graphics (TVCG)* (2021). doi:10.1109/TVCG.2020.3030378

[51] Sydney Nguyen, Hannah McLean Babe, Yangtian Zi, Arjun Guha, Carolyn Jane Anderson, and Molly Q Feldman. 2024. How Beginning Programmers and Code LLMs (Mis)read Each Other. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '24)*. Association for Computing Machinery, New York, NY, USA, Article 651, 26 pages. doi:10.1145/3613904.3642706

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

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

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

[55] 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]

[56] 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. doi:10.1145/3526113.3545616

[57] 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).

[58] Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. 2023. Gorilla: Large language model connected with massive apis. *arXiv preprint arXiv:2305.15334* (2023).

[59] Debalina Ghosh Paul, Hong Zhu, and Ian Bayley. 2024. Benchmarks and Metrics for Evaluations of Code Generation: A Critical Review. *arXiv preprint arXiv:2406.12655* (2024).

[60] 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]

[61] Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. 2023. Toolllm: Facilitating large language

models to master 16000+ real-world apis. *arXiv preprint arXiv:2307.16789* (2023).

[62] Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2023. ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs. arXiv:2307.16789 [cs.AI] https://arxiv.org/abs/2307.16789

[63] Subham Sah, Rishab Mitra, Arpit Narechania, Alex Endert, John Stasko, and Wenwen Dou. 2024. Generating Analytic Specifications for Data Visualization from Natural Language Queries using Large Language Models. Presented at the NLVIZ Workshop, IEEE VIS 2024. arXiv:2408.13391 [cs.HC] https://arxiv.org/abs/2408.13391

[64] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2024. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems* 36 (2024).

[65] 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. doi:10.1109/TVCG.2022.3148007

[66] Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. 2024. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face. *Advances in Neural Information Processing Systems* 36 (2024).

[67] Douglas R Smith. 1985. The design of divide and conquer algorithms. *Science of Computer Programming* 5 (1985), 37–58.

[68] 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. doi:10.1145/3563657.3596089

[69] Anselm Strauss, Juliet Corbin, et al. 1990. *Basics of qualitative research*. Vol. 15. sage Newbury Park, CA.

[70] Anselm L Strauss. 1987. *Qualitative analysis for social scientists*. Cambridge university press.

[71] 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).

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

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

[74] Yuan Tian, Weiwei Cui, Dazhen Deng, Xinjing Yi, Yurun Yang, Haidong Zhang, and Yingcai Wu. 2024. Chartgpt: Leveraging llms to generate charts from abstract natural language. *IEEE Transactions on Visualization and Computer Graphics* (2024).

[75] Ngoc Tran, Hieu Tran, Son Nguyen, Hoan Nguyen, and Tien Nguyen. 2019. Does BLEU score work for code migration?. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 165–176.

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

[77] 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). doi:10.5555/3295222.3295349

[78] 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. doi:10.1145/3544548.3580895

[79] Xinru Wang, Hannah Kim, Sajjadur Rahman, Kushan Mitra, and Zhengjie Miao. 2024. Human-LLM Collaborative Annotation Through Effective Verification of LLM Labels. In *Proceedings of the CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '24)*. Association for Computing Machinery, New York, NY, USA, Article 303, 21 pages. doi:10.1145/3613904.3641960

[80] 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. doi:10.1109/TVCG.2022.3209357

[81] 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]

[82] Chao Wen, Jacqueline Staub, and Adish Singla. 2024. Program Synthesis Benchmark for Visual Programming in XLogoOnline Environment. arXiv:2406.11334 [cs.AI] https://arxiv.org/abs/2406.11334

[83] 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

[84] 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. doi:10.1145/3491101.3519729

[85] 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]

[86] 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. doi:10.1145/3491102.3517582

[87] 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]

[88] Hui Ye, Jiaye Leng, Pengfei Xu, Karan Singh, and Hongbo Fu. 2024. ProInterAR: A Visual Programming Platform for Creating Immersive AR Interactions. In *Proceedings of the CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '24)*. Association for Computing Machinery, New York, NY, USA, Article 610, 15 pages. doi:10.1145/3613904.3642527

[89] Zhengyan Yu, Hun Namkung, Jiang Guo, Henry Milner, Joel Goldfoot, Yang Wang, and Vyas Sekar. 2024. SEAM-EZ: Simplifying Stateful Analytics through Visual Programming. In *Proceedings of the CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '24)*. Association for Computing Machinery, New York, NY, USA, Article 1041, 23 pages. doi:10.1145/3613904.3642055

[90] 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. doi:10.1145/3544548.3581388

[91] 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. doi:10.1145/3379337.3415824

[92] Lei Zhang, Jin Pan, Jacob Gettig, Steve Oney, and Anhong Guo. 2024. VRCopilot: Authoring 3D Layouts with Generative AI Models in VR. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology* (Pittsburgh, PA, USA) *(UIST '24)*. Association for Computing Machinery, New York, NY, USA, Article 96, 13 pages. doi:10.1145/3654777.3676451

[93] 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*.

[94] 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.

[95] Zheng Zhang, Jie Gao, Ranjodh Singh Dhaliwal, and Toby Jia-Jun Li. 2023. VISAR: A Human-AI Argumentative Writing Assistant with Visual Programming and Rapid Draft Prototyping. *arXiv preprint arXiv:2304.07810* (2023).

[96] Zihao Zhao, Eric Wallace, Shi Feng, Dan Klein, and Sameer Singh. 2021. Calibrate Before Use: Improving Few-shot Performance of Language Models. In *Proceedings of the 38th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 12697–12706. https://proceedings.mlr.press/v139/zhao21c.html

[97] Zihao Zhao, Eric Wallace, Shi Feng, Dan Klein, and Sameer Singh. 2021. Calibrate Before Use: Improving Few-shot Performance of Language Models. In *Proceedings of the 38th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 12697–12706. https://proceedings.mlr.press/v139/zhao21c.html

[98] Zhongyi Zhou, Jing Jin, Vrushank Phadnis, Xiuxiu Yuan, Jun Jiang, Xun Qian, Jingtao Zhou, Yiyi Huang, Zheng Xu, Yinda Zhang, Kristen Wright, Jason Mayes, Mark Sherwood, Johnny Lee, Alex Olwal, David Kim, Ram Iyengar, Na Li, and Ruofei Du. 2024. Experiencing InstructPipe: Building Multi-modal AI Pipelines via Prompting LLMs and Visual Programming. In *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI EA '24)*. Association for Computing Machinery, New York, NY, USA, Article 402, 5 pages. doi:10.1145/3613905.3648656

[99] 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. doi: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 (section 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.

```
{
  "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"
  ]
}
```

(a) Body segmentation

```
{
 "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"
 ]
}
```

(b) PaLI

Figure 10: Examples of node configuration used in Code Writer. The configuration is structured in a JSON format.

B System Implementation

B.1 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.

B.1.1 Node Selector. Please see our supplementary file (node_select.txt) for the full prompt we use in this stage.

B.1.2 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 6 shows the structure of the prompt utilized in this LLM stage. Figure 10 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.

B.2 Code Interpreter

Here, we provide more low-level implementation details on Code Interpreter. The Code Interpreter parses generated pseudocode into a visual programming pipeline for visualization at the Visual Blocks workspace. Figure 12 shows the data type definition of graphs, nodes, and edges in the system. The example JSON file to be parsed into the Typescript interface is available at the Visual Blocks website[5]. Our code defines a visual programming pipeline into an array of serialized nodes, $G(N)$. When the user adds a new node to a pipeline, it adds a new "SerializedNode", containing the edge definition between this new node and other nodes in the current workspace, to the current "SerialedGraph". This mechanism ensures that nodes can be incrementally added in the order they appear in the pseudocode order while maintaining the integrity of the graph by clearly defining dependencies and data flow between nodes. Algorithm 1 further shows how InstructPipe parses code and incrementally adds nodes to formulate a final serialized graph.

Algorithm 1: Code Interpreter

```
1  Input: $C$: the generated texts (i.e., pseudocode) in the string format.
2  Output: $G(N)$: a visual programming pipeline (SerializedGraph) that mainly stores an array of SerializedNode (Figure 12).
3  Variables: $T$: a dictionary of parsed tokens that contains output_variable_id, node_id, node_type, node_arguments; $e$: the incoming edges of a new node, in the format of SerializedIncomingEdge; $n$: a new node in the format of SerializedNode.
4  $G : SerializedNode[] = []$ // Initialize $G$ as an empty array
5  $lines = line\_parser(C)$ // Parse $C$ into lines of code with no pseudocode order changed.
6  for line in lines do
       /* Example: */
       /* $pali\_1\_out = pali\_1 : pali(image = input\_image\_1, prompt = input\_text\_1)$ */
       /* -> */
       /* 'pali_1_out', 'pali_1', 'pali' and ['image = input_image_1', 'prompt = input_text_1'] */
7      $T = tokenizer(line)$
8      $e : incomingEdges = create\_incoming\_edges(T)$ // create incoming edges for the new node
9      $n : SerializedNode = create\_node(T, e)$ // create a new SerializedNode with the incomingEdges and the parsed dictionary
10     $G.add\_serializednode(n)$ // add the new SerializedNode to the graph
11 $G = optim\_layout(G)$ // Perform the UI layout optimization, as shown in Figure 11
12 return $G$
```

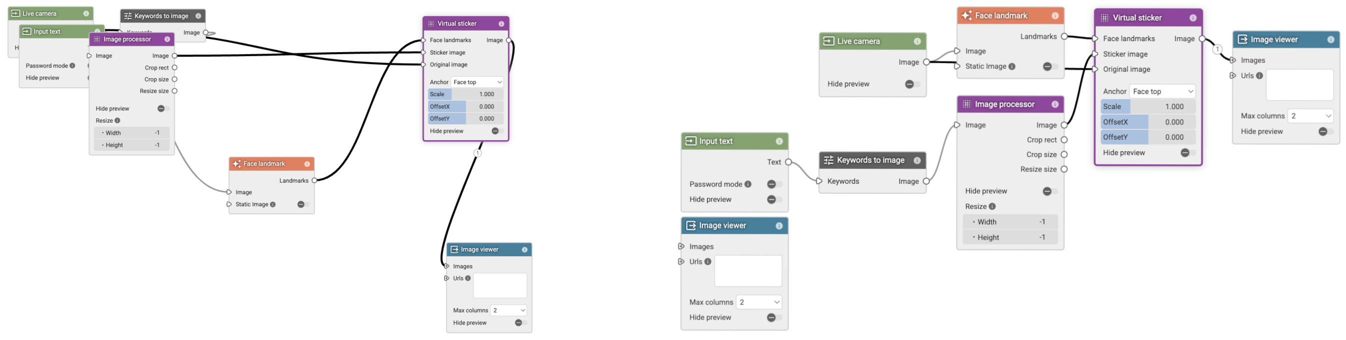
C Technical Evaluation

C.1 Data Post-Processing

After the workshops, one author carefully examined each collected pipeline and found several critical issues in the raw data:

- **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 (*e.g.*, the "Image viewer" node on the bottom-left corner of Figure 11b). 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"*).

[5]https://visualblocks.withgoogle.com/. The JSON file is available for data structure exploration by 1) entering an example project and 2) clicking on the "Export" button on the top-right corner.



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

Figure 11: A comparison of the same generated pipeline before and after layout optimization.

```
/** A serialized graph.  */
export declare interface SerializedGraph {
  nodes: SerializedNode[];

  /** other properties */
}

/** A serialized node.  */
export declare interface SerializedNode {
  /** The id of the node, e.g., pali_1. */
  id: string;

  /** The node spec id, e.g., pali. */
  nodeSpecId: string;

  /**
   * Serialized incoming edges that
   * connect to this node.
   */
  incomingEdges?: {
    [inputId: string]: SerializedIncomingEdge[]
  };

  /** other properties */
}

/** A serialized incoming edge. */
export declare interface SerializedIncomingEdge {
  /** The id of the source node. */
  sourceNodeId: string;

  /** The id of the output in the source node. */
  outputId: string;
}
```

Figure 12: The definition of a graph, a node and an edge in the system using the Typescript language. Only the core properties of graphic structure definition are presented in this figure.

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).

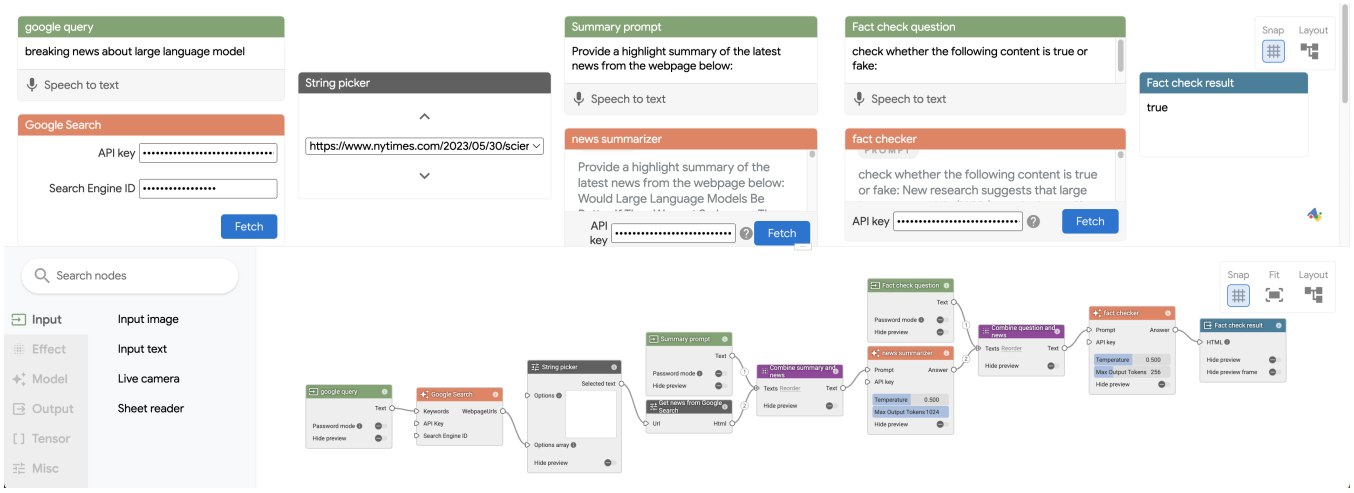
To further enhance the annotation quality, two authors individually annotated the caption of each pipeline separately by referring to the original captions and pipelines authored by the participants. It is important to note that we finished the workshop and the data annotation task *before* we completed 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 maintaining the fairness of the technical evaluation.

As we clarified in section 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 13b for an example) and "TFLite model runner" (which call a custom TensorFlow model with a URL input of the model in the TF-Hub).

We removed the pipelines that contain node(s) out of our focus 27 nodes, and selected all the remaining pipelines as our final evaluation set. We argue that this post-processing is critical for reporting a fair accuracy value since InstructPipe can only generate pipelines based on its known node library. The final 48 pipelines (out of 64 pipelines) are comprised of 23 language pipelines, seven visual pipelines, and 18 multi-modal pipelines. Figure 13 shows three pipelines created by the participants. Figure 13b is an example of the pipelines that include out-of-scope nodes, and therefore are 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 pipeline) and evaluated the generation results using the metric that will be introduced below.

C.2 Evaluation Metric: The Number of User Interactions

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 the node parameters, *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


google query: breaking news about large language model

Summary prompt: Provide a highlight summary of the latest news from the webpage below:

Fact check question: check whether the following content is true or fake:

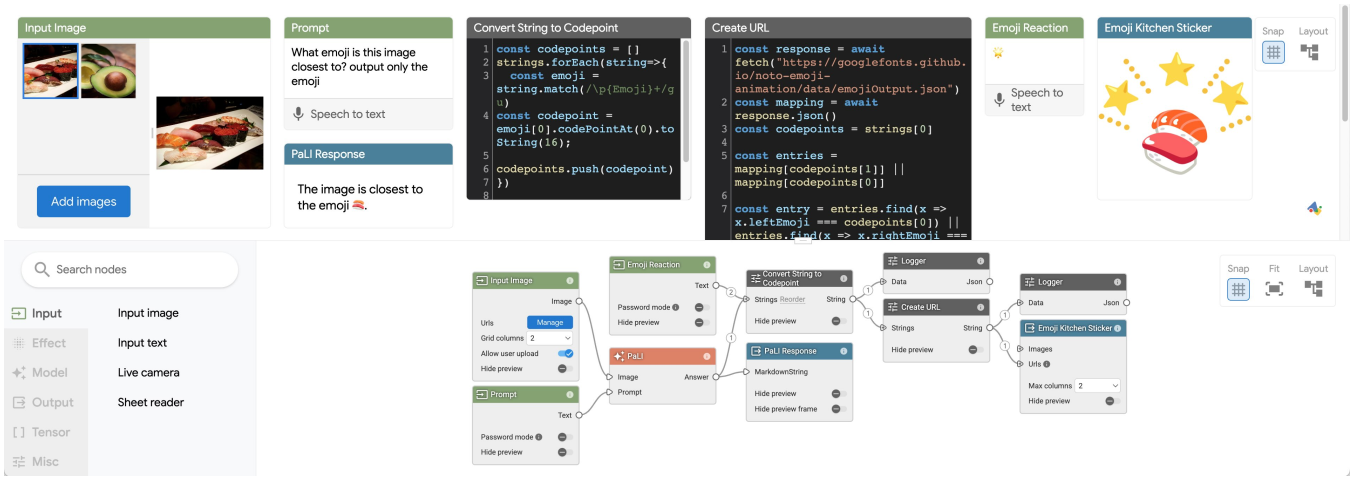
String picker: https://www.nytimes.com/2023/05/30/scier

news summarizer: Provide a highlight summary of the latest news from the webpage below: Would Large Language Models Be

fact checker: check whether the following content is true or fake: New research suggests that large

Fact check result: true

(a) Search news from Google, summarize it, and then conduct a fact check. Input: a keyword for Google Search; Output: a summarization of the news and a fact-check result.


Prompt: What emoji is this image closest to? output only the emoji

PaLI Response: The image is closest to the emoji 🍣.

Convert String to Codepoint:

```
const codepoints = []
strings.forEach(string=>{
  const emoji =
string.match(/\p{Emoji}+/gu)
const codepoint =
emoji[0].codePointAt(0).toString(16);

codepoints.push(codepoint)
})
```

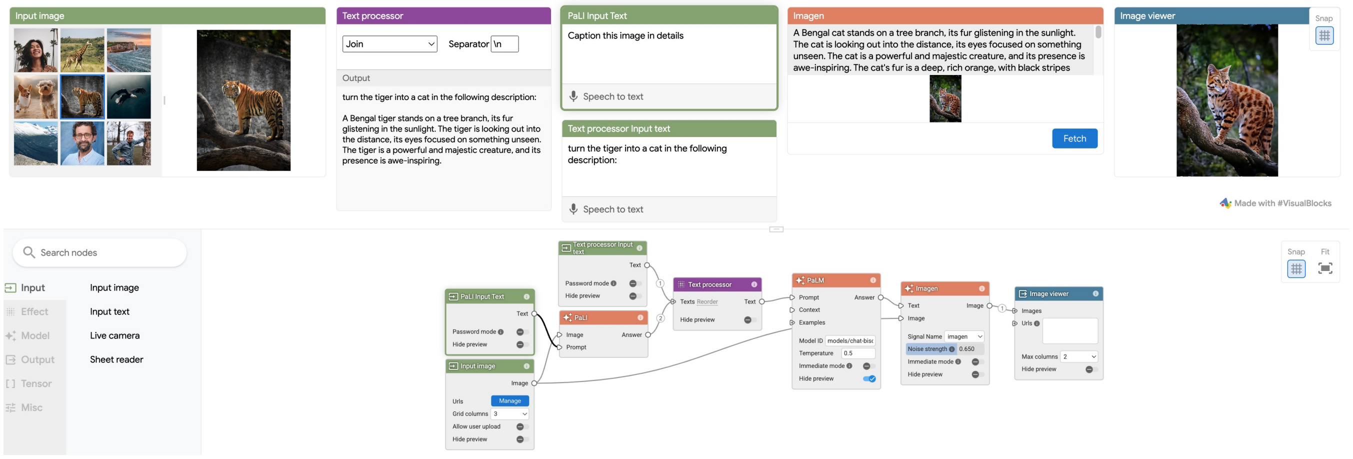
Create URL:

```
const response = await
fetch("https://googlefonts.github.io/noto-emoji-animation/data/emojiOutput.json")
const mapping = await
response.json()
const codepoints = strings[0]

const entries =
mapping[codepoints[1]] ||
mapping[codepoints[0]]

const entry = entries.find(x =>
x.leftEmoji === codepoints[0]) ||
entries.find(x => x.rightEmoji ===
```

(b) Generating an emoji from a photo. Input: a photo uploaded by the user; Output: an emoji generated from the photo.


Text processor: Join, Separator \n

Output: turn the tiger into a cat in the following description:

A Bengal tiger stands on a tree branch, its fur glistening in the sunlight. The tiger is looking out into the distance, its eyes focused on something unseen. The tiger is a powerful and majestic creature, and its presence is awe-inspiring.

PaLI Input Text: Caption this image in details

Text processor Input text: turn the tiger into a cat in the following description:

Imagen: A Bengal cat stands on a tree branch, its fur glistening in the sunlight. The cat is looking out into the distance, its eyes focused on something unseen. The cat is a powerful and majestic creature, and its presence is awe-inspiring. The cat's fur is a deep, rich orange, with black stripes

(c) Turning a tiger into a cat. Input: an image of a tiger; Output: an image of a cat.

Figure 13: Example pipelines participants built in the workshops.

objectively. More importantly, as we explain in section 4.1, the generation of node parameters 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 to make 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.

D User Evaluation

D.1 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.

D.2 User Study Pipelines

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 without further human interaction, and the average ratio of additional human interactions is 27.8%. While the generated diagram (with error) is not deterministic, we observed that InstructPipe commonly generates the pipeline in Figure 14 without "URL to HTML" or "PaLM Text Generator" nodes. The error implies that the LLM may misinterpret 1) the data from the "selected text" port of the "String picker" node is the texts on the web instead of the web URL and 2) that "Text processor" has the LLM capability to process the texts instead of simply combining two texts.

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.3 Assistant Provided to the Participants in the User Evaluation

In the user evaluation, our goal is to make the interface condition (either InstructPipe or Visual Blocks) as the only independent variable that changes our dependent variables (section 5.2). Similar to user evaluations of other early-stage HCI research, we had to improvise for lacking system affordances. As an example, we would include help menus and error recovery models in the future versions of our system, but at this early stage, we relied on in-person help to nudge and assist our user study participants. We took actions (*i.e.*, assistants) in the user evaluation to ensure the study is under an appropriate amount of control as well as maintain the fairness of our study.

Here, we elaborate on two examples of assistants we provided in the user study.

In the InstructPipe condition, one participant started their "instructions" by dragging a text box into the visual programming workspace and began typing. When noticing this issue, we kindly asked the participant whether s/he wanted to write instructions or build a pipeline from scratch. S/he then noticed this issue and clicked on the "InstructPipe" button to write prompts. Note that we explicitly taught every participant how to use InstructPipe

Table 3: 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

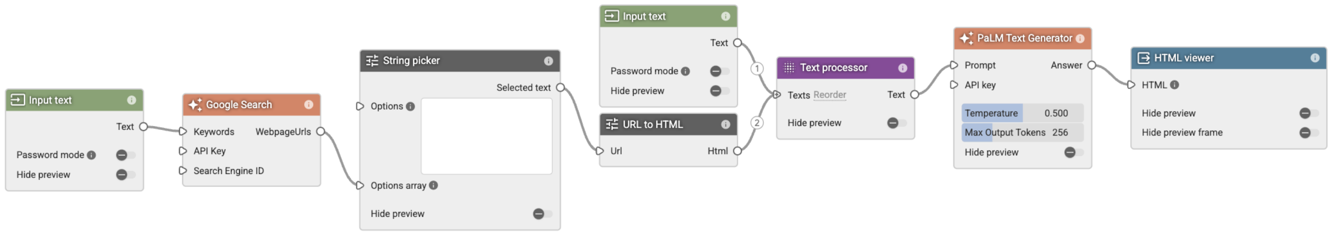


Figure 14: Text-based pipeline. The "String picker" node provides users a drop-down menus to select one URL from a list of URLs returned by "Google Search". "PaLM Text Generator" is an LLM used to summarize the full HTML page.

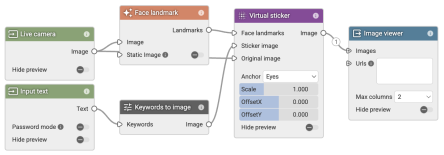


Figure 15: Real-time multimodal pipeline. The "Keyword to image" node is used to search a sunglasses image, and the "Virtual sticker" node anchors the sunglasses onto the user' face.

and asked participants themselves to go through the instruction processes in the training task (Figure 7).

In the Visual Blocks condition, one participant first dragged a "Virtual sticker" into the workspace when s/he wanted to build the multimodal pipeline as required (Figure 15). After a while, s/he asked us for the meaning of "landmarks" on the first input port of the "Virtual sticker" node (Figure 15). We then answered this question and provided a hint on the "Face landmark" node (Figure 15) that could produce the "Face landmarks" required by the "Virtual sticker". While we had explained all the nodes that the participants need to use in the study in the training task (Figure 7), we consider such technical questions reasonable because all of our participants are non-experts. Programming itself is a difficult skill, and it is quite common that people may forget some of the knowledge that they have just learned. Instead of being silent and keeping the participants stuck on a technical issue, we believed offering technical help was an important action we must take to ensure the data quality we collected in the study.

These anecdotes in the user evaluation reveal several limitations of the visual programming system: some designs may not be very intuitive to non-experts. Since the goal of our user evaluation is understanding the benefits of InstructPipe compared to Visual Blocks (without AI assistants), we made our best efforts to take action to prevent the effects caused by other factors from influencing our data. Meanwhile, we also encourage future work to further explore the system design so that future users can more easily use our assistant in visual programming.

Table 4: The counterbalance sheet of the user evaluation. Each cell is in the format of "Interface / Pipeline". "Instruct" and "VB" mean the "InstructPipe" and "Visual Blocks" conditions, respectively. "Search" and "Tryon" represent the "text-based pipeline" (Figure 14) and the "real-time multimodal pipeline" (Figure 15), respectively.

ID	Step 1	Step 2	Step 3	Step 4
P1	Instruct / Tryon	Instruct / Search	VB / Tryon	VB / Search
P2	VB / Tryon	VB / Search	Instruct / Tryon	Instruct / Search
P3	Instruct / Search	Instruct / Tryon	VB / Search	VB / Tryon
P4	VB / Search	VB / Tryon	Instruct / Search	Instruct / Tryon
P5	VB / Tryon	VB / Search	Instruct / Tryon	Instruct / Search
P6	Instruct / Tryon	Instruct / Search	VB / Tryon	VB / Search
P7	VB / Search	VB / Tryon	Instruct / Search	Instruct / Tryon
P8	Instruct / Search	Instruct / Tryon	VB / Search	VB / Tryon
P9	Instruct / Tryon	Instruct / Search	VB / Tryon	VB / Search
P10	VB / Tryon	VB / Search	Instruct / Tryon	Instruct / Search
P11	Instruct / Search	Instruct / Tryon	VB / Search	VB / Tryon
P12	VB / Search	VB / Tryon	Instruct / Search	Instruct / Tryon
P13	VB / Tryon	VB / Search	Instruct / Tryon	Instruct / Search
P14	Instruct / Tryon	Instruct / Search	VB / Tryon	VB / Search
P15	VB / Search	VB / Tryon	Instruct / Search	Instruct / Tryon
P16	Instruct / Search	Instruct / Tryon	VB / Search	VB / Tryon

D.4 Counter-Balancing and The Replication Number

Table 4 presents how we perform counterbalance in the user evaluation. We counterbalanced both the interface factor ("InstructPipe" and "Visual Blocks") and the pipeline factors to reduce the learning effects. We then replicated the order four times so that we collected multiple data from different participants in each unique study order. This helps strengthen the power of the data we collected in the study. Note that, in the group of P5 - P8, we flipped the orders within P5 and P6 as well as P7 and P8, but this does not cause a difference in the counterbalance process.