AI for Low-Code for AI

Nikitha Rao
nikitharao@cmu.edu
Carnegie Mellon University
United States

Jason Tsay
jason.tsay@ibm.com
IBM Research
United States

Kiran Kate
kakate@us.ibm.com
IBM Research
United States

Vincent J. Hellendoorn
vhellendoorn@cmu.edu
Carnegie Mellon University
United States

Martin Hirzel
hirzel@us.ibm.com
IBM Research
United States

ABSTRACT

Low-code programming allows citizen developers to create programs with minimal coding effort, typically via visual (e.g. drag-and-drop) interfaces. In parallel, recent AI-powered tools such as Copilot and ChatGPT generate programs from natural language instructions. We argue that these modalities are complementary: tools like ChatGPT greatly reduce the need to memorize large APIs but still require their users to read (and modify) textual programs, whereas visual tools abstract away most or all program text but struggle to provide easy access to large APIs. At their intersection, we propose LowCoder, the first low-code tool for developing AI pipelines that supports both a visual programming interface (LowCoder$_{VP}$) and an AI-powered natural language interface (LowCoder$_{NL}$). We leverage this tool to provide some of the first insights into whether and how these two modalities help programmers by conducting a user study. We task 20 developers with varying levels of AI expertise with implementing four ML pipelines using LowCoder, replacing the LowCoder$_{NL}$ component with a simple keyword search in half the tasks. Overall, we find that LowCoder is especially useful for (i) Discoverability: using LowCoder$_{NL}$, participants discovered new operators in 75% of the tasks, compared to just 32.5% and 27.5% using web search or scrolling through options respectively in the keyword-search condition, and (ii) Iterative Composition: 82.5% of tasks were successfully completed and many initial pipelines were further successfully improved. Qualitative analysis shows that AI helps users discover *how* to implement constructs when they know *what* to do, but still fails to support novices when they lack clarity on what they want to accomplish. Overall, our work highlights the benefits of combining the power of AI with low-code programming.

ACM Reference Format:
Nikitha Rao, Jason Tsay, Kiran Kate, Vincent J. Hellendoorn, and Martin Hirzel. 2024. AI for Low-Code for AI. In *29th International Conference on Intelligent User Interfaces (IUI '24), March 18–21, 2024, Greenville, SC, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3640543.3645203

IUI '24, March 18–21, 2024, Greenville, SC, USA

ACM ISBN 979-8-4007-0508-3/24/03
https://doi.org/10.1145/3640543.3645203

1 INTRODUCTION

Most AI development today involves Python programming with popular libraries such as scikit-learn (sklearn) [28]. Unfortunately, writing code, even in a language as high-level as Python, is hard for *citizen developers* [22]—people who lack formal training in programming but nevertheless write programs as part of their everyday work. This is a fairly common situation for data scientists, among others. AI programming libraries also tend to be large and change regularly. Needing to remember hundreds of AI operators and their arguments slows down even professional developers.

Low-code programming [20] reduces the amount of textual code developers write by offering alternative programming interfaces. In recent years, it has been embraced by software vendors to both democratize software development and increase productivity [32]. Most low-code offerings for building AI pipelines currently favor visual programming [6, 12, 18]. While visual programming helps users navigate complex pipelines, it poorly supports *discoverability* of API components in large APIs due to the large range of options and limited screen space [27]. In parallel, *programming by natural language* (PBNL) has recently soared in popularity. Tools like Copilot [1] and ChatGPT [2] can generate code from natural language prompts in which users describe what they want to accomplish, which is especially helpful in ecosystems with large APIs. However, these tools still generate code, which can be complicated and hard to understand [35], especially without formal training in programming.

At the intersection of these two paradigms, we propose LowCoder, the first low-code tool to combine visual programming with PBNL. We conjecture that the respective strengths of these two low-code techniques can compensate for each other's weaknesses. PBNL uses AI to help users retrieve and use programming constructs based on natural language queries. This does not always return correct programs, necessitating a way to help users understand and fix generated programs. Visual programming complements PBNL by providing a clear, unambiguous representation of the program that users can directly manipulate to experiment with alternatives.

Our goal is to help people who know *what* they want to accomplish (e.g., build an AI pipeline) but face syntactic barriers from the programming language and library (the *how* part), perhaps due to a lack of formal programming training. End-users writing software face similar "design barriers" [22], where it is difficult to even conceptualize a solution. In contrast to other popular low-code domains such as traditional software [31], the domain of developing AI pipelines is particularly difficult in this regard due to its experimental nature where progress has a high degree of uncertainty [39].

We chose to target sklearn [28] because of its pervasive use, because visual programming naturally fits the pipeline structure of sklearn, and because PBNL is particularly useful in aiding recall of operators from the relatively large API of sklearn.

LowCoder's visual programming component, LowCoder$_{VP}$, lets users snap together visual blocks for AI operators into well-structured AI pipelines. It uses Blockly [27] to provide a Scratch-like [31] look-and-feel. The PBNL component, LowCoder$_{NL}$, lets users enter natural language queries and predicts relevant operators, optionally configured with hyper-parameters. It uses a fine-tuned variant of the CodeT5 model [42] that we developed through experiments with a variety of neural models for program generation, ranging from training models from scratch to few-shot prompting large language models [25]. We further noticed that queries usually mention at most a subset of hyper-parameters for each pipeline step, so we developed a novel task formulation tailored to this use case that improved learning outcomes.

We leverage LowCoder to provide some of the first insights into both how and when low-code programming and PBNL help developers with various degrees of expertise. We conduct a user study with 20 participants with varying levels of AI expertise using LowCoder to complete four tasks, half of which with the help of the AI-powered component LowCoder$_{NL}$. Overall, the combination of visual programming along with the natural language interface helped both novice and non-novice users to successfully compose pipelines (85% of tasks) and then further refine their pipelines (72.5% of tasks) when using LowCoder$_{NL}$. Additionally, LowCoder$_{NL}$ helped users discover previously-unknown operators in 75% of the tasks compared to just 32.5% using other methods like web search when LowCoder$_{NL}$ was not available. In addition, despite being trained on a different dataset, LowCoder$_{NL}$ accurately answered real user queries. In summary, this paper makes three main contributions:

(i) Low-Code for AI: We introduce LowCoder, a new low-code tool that combines language models with visual programming to help develop AI pipelines.
(ii) AI for Low-Code: We benchmark various AI models and develop a novel task formulation to develop an AI powered natural language interface to LowCoder.
(iii) User Study: We analyze the trade-offs between the two modalities and study the effects of using language models for low-code programming through a user study involving 20 participants with varying levels of AI expertise.

While we prototyped LowCoder specifically for sklearn pipelines, we hope the general findings will help improve low-code tooling for other API libraries as well.

2 RELATED WORK

Low-code: In adopting a visual programming approach to low-code, we follow a long tradition [7]. We were particularly inspired by Scratch, a popular visual programming environment for children that uses lego-like connected blocks [31]. Our other inspiration came from projectional editors, where the visual programming interface is a projection, or *view*, over an internal domain-specific language (DSL) [38]. Our implementation uses Blockly, a meta-tool for creating block-based visual programming tools [27], and Lale, a DSL for machine-learning pipelines [4].

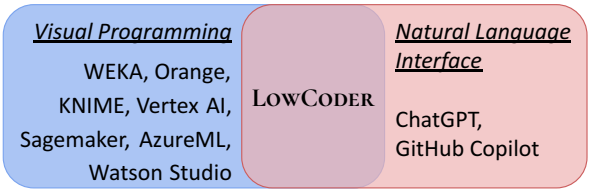

Figure 1: Relationship between LowCoder and other low-code for AI tools. LowCoder is the only low-code tool that supports both visual programming and a natural language interface.

Visual programming for AI: Most low-code interfaces for programming AI pipelines use visual programming. Examples include WEKA [18], Orange [12], and KNIME [6]. Each has a palette of operators that can be dragged onto a canvas, where they can be connected into a boxes-and-arrows style diagram. Commercial low-code visual interfaces follow the same approach, such as Vertex AI, Sagemaker, AzureML, and Watson Studio. A related approach for low-code ML pipeline development is automated machine learning (AutoML) [34], which is also used by many of the same commercial AI interfaces mentioned earlier. These tools tend to have a black-box approach where the user has little control over the AutoML search and may not even see the resulting pipeline. AutoML libraries such as auto-sklearn [16], TPOT [26], and hyperopt [5] provide a Python interface, which is intended for textual code development. There are also natural-language interfaces for professional developers based on large language models such as GitHub Copilot which uses Codex [10] and ChatGPT. Since these support APIs for which there is sufficient publicly available code to use as training data, they cover popular AI libraries such as sklearn. The main difference between these low-code tools for AI and our paper is that we combine the ease-of-use of visual programming with a natural language interface to help users discover and configure operators and, inspired by Scratch [31], our tool encourages liveness [33] through immediate user feedback for each user input into the system. This contrasts with most tools that require explicit training and scoring steps for feedback. Figure 1 summarizes the relationship between LowCoder and other low-code for AI tools.

Using AI for low-code development: The most prominent AI technique for low-code is programming by natural language (PBNL). When Androutsopoulos et al. surveyed natural language interfaces to databases in 1995, it was already a well-established field [3]. Desai et al. treat PBNL as a program synthesis problem targeting a DSL designed for the purpose [13]. The Overnight paper addresses the problem of missing training data for PBNL interfaces by crowdsourcing [41]. And SwaggerBot lets users extend and customize a chatbot from within the chatbot itself [37]. Unlike these works, our paper uses language models for PBNL, uses PBNL for creating AI pipelines, and integrates with a visual programming interface.

Combining low-code techniques: Our work combines visual programming with PBNL. In a similar vein, Rousillon combines visual programming with programming by demonstration [9] and Pumice combines programming by demonstration with PBNL [23].

Like Rousillon and Pumice, our goal in combining techniques is to use strengths of each technique to mitigate weaknesses in the other. However, unlike Rousillon and Pumice, we choose different techniques to combine and target a different domain, namely AI pipelines.

User studies on AI tools: There are a few studies that aim to evaluate whether developers perform better on programming tasks when working with AI tools. Vaithilingam et al. had developers use GitHub Copilot on three programming tasks and found that while neither task success rate nor completion time improved while using Copilot, developers preferred using it compared to the standard code completion [35]. Similarly, Xu et al. had developers perform several programming tasks with and without the use of a natural language to code generation model and found no significant differences with regards to code quality, task completion time and program correctness [43]. Wang et al. interviewed several data scientists to better understand their perceptions of automated AI and found that they had mixed feelings [40]. However, nearly all of them felt that the future of data science involved collaboration between humans and AI systems. Unlike other work which tends to focus on how AI supports software development by experienced developers, our paper focuses on AI tools in the context of low-code systems where developers have varying expertise levels in both building software and AI.

3 LOW-CODE FOR AI: LOWCODER TOOL DESIGN

This work explores the intersection of visual programming and language models in an effort to understand the benefits and limitations of using the combination in low-code programming. We accomplish this by implementing and studying LowCoder, a prototype low-code tool for building ML pipelines with sklearn operators for tabular data that includes both visual programming (VP) and natural language (NL) modalities, which complement each other by mitigating the limitations of either modality separately. Building this tool provided us with the opportunity to examine the impact of both modalities on users. Figure 2 highlights the main features and inputs of LowCoder.

To support multiple low-code modalities, we follow the lead of projectional code editors [38] by adopting the model-view-controller pattern. Specifically, we treat visual programming as a read-write view, PBNL as a write-only view, and let users inspect data in a read-only view [20]. The tool keeps these three views in sync by representing the program in a domain-specific language (DSL). The domain for the DSL is AI pipelines. A corresponding, practical desideratum is that the DSL is compatible with sklearn [28], the most popular library for building AI pipelines, and is a subset of the Python language, in which sklearn is implemented, which also enables us to use AI models pretrained on Python code. The open-source Lale library [4] satisfies these requirements, and in addition, describes hyper-parameters in JSON schema format [29], which our tool also uses. The current version of our tool supports 143 sklearn operators. LowCoder$_{VP}$ uses a client-server architecture with a Python Flask back-end server and front-end based on the Blockly [27] meta-tool for creating block-based visual programming tools. The front-end converts the block-based representation to Lale which is then sent to the back-end. The back-end validates

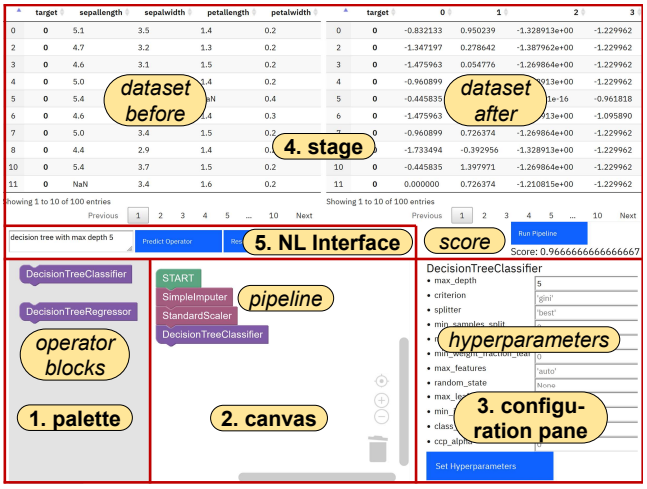

Figure 2: LowCoder interface with labeled components, described in the text.

the given Lale pipeline using internal schemas, then evaluates the pipeline against a given dataset. The results of this evaluation (including any error messages) are returned to the front-end and presented to the user.

3.1 Visual Programming Interface

LowCoder$_{VP}$ is our block-based visual programming interface for composing and modifying AI pipelines. One goal that this tool shares with other block-based visual tools such as Scratch [31] is to encourage a highly interactive experience. The block visual metaphor allows for blocks that correspond to sklearn operators to be snapped together to form an AI pipeline. The shape of the blocks suggest how operators can connect. Their color indicates how they affect data: red for operators that transform data (with a *transform()* method) and purple for other operators that make predictions, such as classifiers and regressors (with a *predict()* method).

Figure 2 illustrates the interface. A *palette (1)* on the left side of the interface contains all of the available operator blocks. Blocks can be dragged-and-dropped from the palette to the *canvas (2)*. For ease of execution, our tool only allows for one valid pipeline at a time, so blocks must be attached downstream of the pre-defined *Start* block to be considered part of the active pipeline. Figure 2 shows an example of blocks defining a pipeline where the `SimpleImputer`, `StandardScaler`, and `DecisionTreeClassifier` blocks are connected to the *Start* block and each other. Input data are transformed by the first two operators (`SimpleImputer` and `StandardScaler`) and then sent to `DecisionTreeClassifier` for training and then scoring. Blocks not attached to the *Start* block are disabled but can be left on the canvas without affecting the execution of the active pipeline. Selected operator blocks also display a *hyper-parameter configuration pane (3)* on the right. The pane lists each hyper-parameter for an operator along with a description (when hovering over the hyper-parameter name) and default values along with input boxes to modify each hyper-parameter.

Our tool provides a *stage (4)* with *Before* and *After* tables to give immediate feedback with every input on how the current pipeline

affects the given dataset. When a tabular dataset is loaded, the *Before* table displays its target column on the left and feature columns on the right. When a pipeline that transforms input data is executed, the *After* table shows the results of the transformations. At any time, a pipeline can be executed on the given dataset by pressing the "Run Pipeline" button. Executing a pipeline will attempt to train the given pipeline on the training portion of the given dataset and then return a preview of all data transformations on the training data in a second table. For instance, in the example shown in Figure 2, executing the pipeline with `SimpleImputer` and `StandardScaler` transforms data from the *Before* table by imputing missing values and standardizing all feature values in the *After* table. If training is successful, then the trained pipeline is scored against the test set and the score (usually accuracy) is displayed. LowCoder$_{VP}$ also encourages liveness [33] by executing the pipeline when either the active pipeline is modified or hyper-parameters are configured. For example, adding a `PCA` operator and setting the `n_components` hyper-parameter to 2 for the prior example will reduce the feature columns in the *After* table to 2. Hence, users receive immediate feedback on the effect of pipeline changes on the dataset without requiring separate training or scoring steps. This liveness encourages a high degree of interactivity [31].

3.2 Natural Language Interface

A potential weakness of visual low code tools is that users have trouble discovering the right components to use [22]. For instance, the palette of LowCoder$_{VP}$ contains more than a hundred operator blocks. Rather than requiring users to know the exact name of the operator or scroll through so many operators, we provide LowCoder$_{NL}$, which allows users to describe a desired operation in the *NL interface (labeled component 5 in Figure 2)* text box and press the "Predict Pipeline" button. The tool then infers relevant operator(s) and any applicable hyper-parameters using an underlying natural-language-to-code translation model and automatically adds the most relevant operator to the end of the pipeline. The palette is also filtered to only display any relevant operator(s) such as in Figure 2. Pressing the "Reset Palette" button will undo filtering (so the palette shows all available operators again) without clearing the active pipeline or canvas. Depending on the NL search, the automatically added operator may either have hyper-parameters explicitly defined or potentially relevant hyper-parameters highlighted. As an example, the NL search *"PCA with 2 components"* will automatically add the `PCA` operator where the `n_components` hyper-parameter is set to 2 and may highlight other hyper-parameters such as `random_state` for the user to consider setting. Section 4 describes the design and implementation of this model in detail. A potential weakness of natural language low-code tools is that the generated programs can be incorrect, due to a lack of clarity, or ambiguity, in the query, or a lack of context for the model providing inferences [3]. In comparison, visual inputs and representations are unambiguous [20], requiring no probabilistic interpretation, so users can easily understand and manipulate the results returned by LowCoder$_{NL}$.

To ground our evaluation of LowCoder$_{NL}$, we also provide a version of the tool without a trained language model to users in our study (described in Section 5). In this setting, the *NL interface (5)* text box becomes a simple substring keyword search that matches the query against operator names. For example, inputting *"classifier"* filters the palette to only display sklearn operators that contain *'classifier'* in the name such as `RandomForestClassifier` (but notably not all classifiers such as `SVC`).

4 AI FOR LOW-CODE: USING LANGUAGE MODELS FOR LOW-CODE

This section discusses the language modeling for LowCoder$_{NL}$.

4.1 Data Collection

Our goal is to make a large API accessible through a low-code tool by allowing users to describe *what* they want to do when they do not know *how*. More specifically, we want to enable users to build sklearn pipelines in a low-code setting, using a natural language interface that can be used as an *intelligent search* tool. This problem can be solved using language models that can be trained to translate a natural language query into the corresponding line of code [15]. However, such models heavily rely on data to learn such behaviour and would need to be trained on an aligned dataset of natural language queries and the corresponding sklearn line(s) of code demonstrating how a user would want to use such an intelligent search tool. Naturally, we cannot collect such a dataset without this tool, creating a circular dependency. To overcome this challenge, we curate a *proxy dataset* using 140K Python Kaggle notebooks that were collected as part of the Google AI4Code challenge.[1] From these notebooks, we extracted aligned Natural Language (NL) & Code cells related to machine learning and data science tasks. While the distribution of the NL in the markdown cells is not completely representative of the NL queries that users would enter in the low-code setting, they provide the model with a broad range of such examples. Results in Section 5.1.5 show that this is indeed effective.

4.2 Data Preprocessing

We first filter out notebooks that do not contain any sklearn code. This leaves 84,783 notebooks – evidently, many notebooks involve sklearn. We further filter out notebooks with non-English descriptions in all of the markdown cells, resulting in 59,569 notebooks. We then create a proxy dataset by extracting all code cells containing sklearn code and pairing these with their preceding NL cell to get a total of 211,916 aligned NL-code pairs. We remove any duplicate NL-code pairs, leaving 102,750 unique pairs. For each code cell, we then extract the line(s) of code corresponding to an sklearn operation invocation statement.

We discard any code cells that do not include sklearn operation invocation statements but include other sklearn code, leaving a final total of 79,372 NL-Code pairs. We separate these into train/validation/test splits resulting in 64,779 train samples, 7,242 validation samples, and 7,351 test samples. See Section B in the appendix for more details.

4.3 Tasks

Given the NL query, our model aims to generate a line of sklearn code corresponding to an operation invocation that can be used

[1] https://www.kaggle.com/competitions/AI4Code

Table 1: Task formulations highlighting the code components: mask, operator name, hyper-parameter name, hyper-parameter value. The Hybrid Operator Invocation setting does not mask 'balanced' as it appears in the query.

Task Formulation	Code for the NL query: *Random forest with balanced class weight*
Operator Name	`RandomForestClassifier`
Complete Operator Invocation	`RandomForestClassifier ( n_estimators = 100 , class_weight = 'balanced' )`
Masked Operator Invocation	`RandomForestClassifier ( n_estimators = MASK , class_weight = MASK )`
Hybrid Operator Invocation	`RandomForestClassifier ( n_estimators = MASK , class_weight = 'balanced' )`

to build the next step of the pipeline. We consider a range of formulations of the task with different levels of details, as illustrated in Table 1. Additional examples can be found in Section A of the appendix.

4.3.1 Operator Name Generation. The simplest task is generating only the operator name from the NL query. This alone can significantly help a developer with navigating the extensive sklearn API. We process the aligned dataset to map the query to the name(s) of operator(s) invoked in the code cell, discarding any other information such as hyper-parameters.

4.3.2 Complete Operator Invocation Generation. At the other extreme, we task the model with synthesizing the complete operation invocation statement, including all the hyper-parameter names and values. Preliminary results (discussed in Section 5.1.4) show that the model often makes up arbitrary hyper-parameter values, resulting in lines of code that can rarely be used directly by developers.

4.3.3 Masked Operator Invocation Generation. In this scenario, we mask out all the hyper-parameter values from the invocation statement, keeping only their names. The goal of this formulation is to ensure that the model learns to predict the specific invocation signature, even if it is unaware of the values to provide for the hyper-parameters.

4.3.4 Hybrid Operator Invocation Generation (HOI). Manual inspection of the NL-code pairs revealed that the queries sometimes explicitly describe a subset of the hyper-parameter names and values to be used in the code. When this is the case, the model has the necessary context to predict at least those hyper-parameter values. Supporting this form of querying enables users to express the most salient hyper-parameters up-front. Therefore, we formulated a new hybrid task, where we keep the hyper-parameter values if they are explicitly stated in the NL query and mask them otherwise. This gives the model an opportunity to learn the hyper-parameter names and values if they are explicitly stated in the description, and unburdens it from making up values that it lacks the context to predict by allowing it to generate placeholders (masks) for them.
Evaluation: To evaluate the feasibility of predicting code using the different task formulations, we train a simple sequence-to-sequence model (detailed in Section 4.4.1) and compare the results for the various training tasks in Section 5.1.4. We find HOI to be the most accurate/reliable formulation for our setting. We therefore proceed to use this task formulation for training the models.

4.4 Modeling

All tasks from Section 4.3 are sequence-to-sequence tasks. We compare and contrast three different deep learning paradigms for this type of task, illustrated in Figure 3: 1) train a standard sequence-to-sequence transformer *from scratch*, 2) fine-tune (calibrate) a pre-trained "medium" sized model, 3) query a Large Language Model (LLM) by means of few-shot prompting [30]. We elaborate on these models below. Note that we use top-k sampling for our top-5 results. (Appendix C and D shows a comparison of results with other decoding strategies.)

4.4.1 Transformer (from scratch). We train a sequence-to-sequence Transformer model [36] with randomly initialized parameters on the training data. Our relatively small dataset of ca. 70K training samples limits the size of a model that can be trained in this manner. We use a standard model size, with 6 encoder and decoder layers and 512-dimensional attention across 8 attention heads and a batch size of 32 sequences with up to 512 tokens each. We use a sentence piece tokenizer (trained on Python code) with a vocabulary size of 50K tokens. The model uses an encoder-decoder architecture that jointly learns to encode (extract a representation of) the natural language sequence and decode (generate) the corresponding sklearn operator sequences.

4.4.2 Fine-tuning CodeT5. CodeT5 is a pretrained encoder-decoder transformer model [42] that has shown strong results when fine-tuned on various code understanding and generation tasks [24]. CodeT5 was pretrained on a corpus of six programming languages

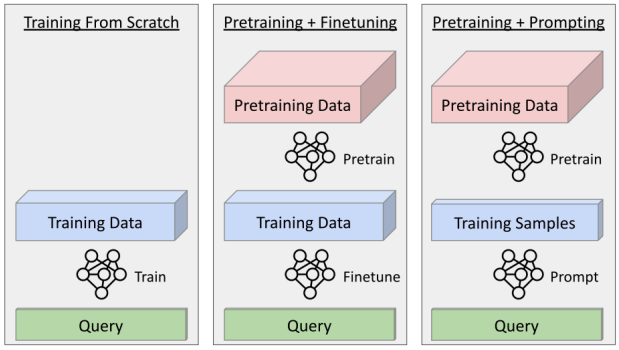

Figure 3: Overview of the "trifecta" of training approaches used in contemporary deep learning: smaller models are directly trained from scratch on downstream task data; medium sized models (100M-1B parameters) are pretrained with a generic training signal and then fine-tuned on task data; large models (>1B parameters) are only pretrained on very large datasets and are prompted with examples from the training data as demonstration followed by the query.

```
NL: Build a simple linear support vector classification
Code: SVC(kernel='linear', random_state=MASK)

NL: PCA with 2 components
Code: PCA(n_components=2)

NL: Put the column median instead of missing values
Code: SimpleImputer(missing_values=MASK, strategy='median')

NL: [Enter query here]
Code:
```

Figure 4: Example of a few (3) shot prompting template for querying a large language model in our study.

from the CodeSearchNet dataset [21] and fine-tuned on several tasks from the CodeXGLUE benchmark[24] in a multi-task learning setting, where the task type is prepended to the input string to inform the model of the task. We fine-tune CodeT5 on the HOI generation task by adding the 'Generate Python' prefix to all NL queries. We experiment with different size CodeT5 models: codet5-small (60M parameters), base (220M), and large (770M).

4.4.3 Few-Shot Learning With CodeGen. Lastly, we explore large language models (LLMs) that are known to perform well in a task-agnostic few-shot setting [8]. More specifically, we look at CodeGen, a family of LLMs that are based on standard transformer-based autoregressive language modeling [25]. Pretrained CodeGen models are available in a broad range of sizes, including 350M, 2.7B, 6.1B and 16.1B parameters. These were all trained on three different datasets, starting with a large, predominantly English corpus, followed by a multi-lingual programming language corpus, and concluding with fine-tuning on just Python data, which we use in this work. The largest model trained this way was shown to be competitive with Codex [10] on a Python benchmark [25].

Models at this scale are expensive to fine-tune and are instead commonly used for inference by means of "few-shot prompting" [30]. LLMs are remarkably capable of providing high-quality completions given an expanded prompt containing examples demonstrating the task [8]. We prompt our model with 5 such NL-code examples. Figure 4 illustrates an example prompt with 3 such pairs. The model does in-context learning on the examples in the prompt and completes the sequence task, which results in generating the HOI code.

5 EVALUATION

This section describes the evaluations for the language modeling that enables LowCoder$_{NL}$ along with the user studies that we conducted to analyze the benefits and challenges of using low-code for developing AI pipelines using LowCoder.

5.1 Modeling

5.1.1 Experimental Setup. All of our models are implemented using PyTorch transformers and the HuggingFace interface. We use the latest checkpoints of the CodeT5 [42] and CodeGen [25] models. Our models were trained on a single machine with multiple 48 GB NVIDIA Quadro RTX 8000 GPUs until they reached convergence on the validation loss. We clip input and output sequence lengths to 512 tokens, but reduce the latter to 64 when using the model in LowCoder to reduce inference time. We find in additional experiments that since few predictions are longer than this threshold, this incurs no significant decrease in accuracy, but speeds up inference by 34%. We use a batch size of 32 for training and fine-tuning all of our Transformer and CodeT5 models, except for CodeT5-large, for which we used a batch size of 64 to improve stability during training.

5.1.2 Test Datasets. To ensure a well-rounded evaluation, we look at two different test datasets.
(i) Test data (from notebooks) - We use the NL-code pairs from the Kaggle notebooks we created in Section 4.2 containing 7,351 samples. These are noisy – some samples contain vague and underspecified Natural Language (NL) queries, such as - *"Data preprocessing"*, *"Build a model"*, *"Using a clustering model"*. Others contain multiple operator invocation statements corresponding to a single NL query, even though the NL description only mentions one of them, e.g., *"Model # 2 - Decision Trees"* corresponds to `DecisionTreeClassifier()` and `confusion_matrix(y_true, y_pred)`. Furthermore, these samples were collected from Kaggle notebooks, so the distribution of the NL queries collected from the markdown cells are not necessarily representative of NL queries that real users may enter into LowCoder$_{NL}$.
(ii) Real user data - We log all the NL queries that users searched for in LowCoder during the user studies along with the list of operators that the model returned. This gives us a more accurate distribution of NL queries that developers use to search for operators in LowCoder$_{NL}$. We obtained a total of 218 samples in this way, which we then manually annotated to check whether (i) the predictions were accurate, that is, if the operators in any of the predictions matches the inferred intent in the query and (ii) the NL query was clear, with an inter-rater agreement of 97.7% and a negotiated agreement [17] of 100%. (See Appendix E for details on annotation guidelines.)

5.1.3 Test Metrics. We use both greedy (top-1) and top-K (top-5) decoding (see Section C in appendix) when generating the operator invocation sequences for each NL query. We evaluate the models' ability to generate just the operator name as well as the entire operator invocation (including all the hyper-parameter names and values) based on the hybrid formulation.

5.1.4 Task Comparison. We first train a series of randomly initialized 6-layer Transformer models from scratch on each task formulation from Section 4.3. We compare the model's ability to correctly generate the operator name and the operator invocation based on the formulation corresponding to the training task using top-1 and top-5 accuracy as shown in Figure 5. We find that the hybrid formulation of the operation invocation task, while challenging, is indeed feasible and allowed the model to achieve reasonably strong performance when generating the entire operation invocation statement. Contrary to the other task formulations, a model trained with the HOI signal also achieved comparable performance to the model trained solely on operator names when evaluated purely on operator name prediction (ignoring the generated hyper-parameter string). These results highlight that the hybrid representation helps the model learn by unburdening it from inferring values that it lacks the context to predict.

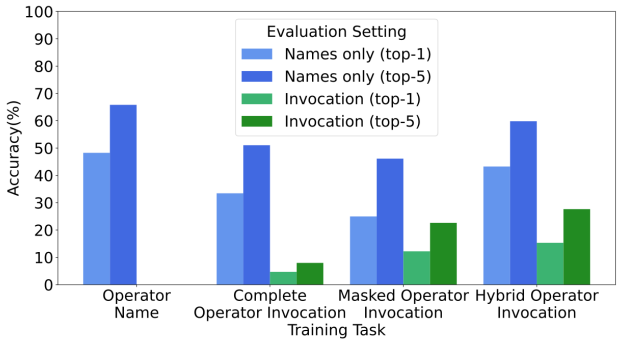

Figure 5: Accuracy of Transformer models trained from scratch on various task formulations. 'Invocation' test results refer to the specific invocation formulation of the training task, while 'Names only' just considers whether the generated code starts with the correct operator name. Only the Hybrid Operator Invocation setting yields useful quality on both tasks.

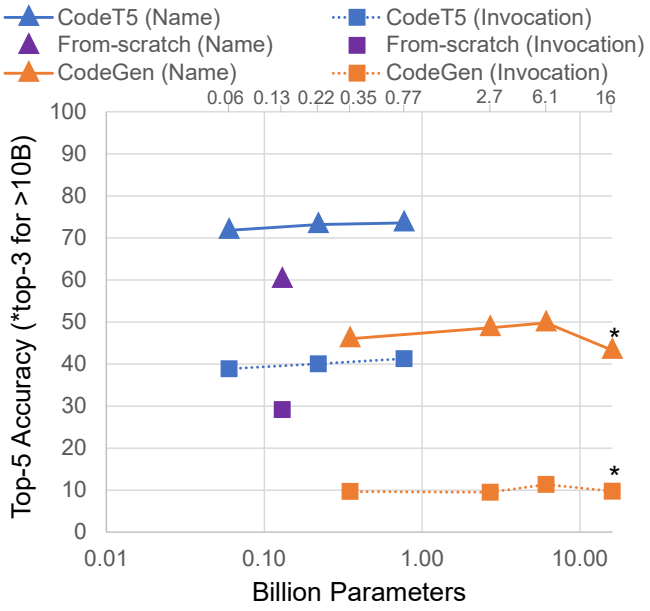

Figure 6: Accuracy vs. model size based on top-5 sampling. (*The 16B CodeGen uses top-3 due to memory constraints.) We compare the three modeling paradigms, namely training transformer from scratch, finetuning CodeT5, and fewshot prompting CodeGen, on both Operator Name generation and Hybrid Operator Invocation generation.

5.1.5 Model Comparison. We next evaluate the performance of the trifecta of modeling strategies from Section 4.4 on the task of Hybrid Operation Invocation (HOI) generation. We benchmark across different model sizes and compare the performance for both operator name and operator invocation generation using top-5 accuracy in Figure 6. (See Section D in the appendix for additional results and ablation studies.) The results show that the 0.77B parameter fine-tuned CodeT5 is the best performing model with an accuracy of 73.57% and 41.27% on the test data for the operation name and operation invocation generation respectively. The 0.22B parameter fine-tuned CodeT5 model has comparable performance, but its inference time is approximately 2–3 seconds faster than the 0.77B fine-tuned CodeT5 model, making it more desirable for integration with the tool.

5.1.6 Performance in Practice. Up to this point, all our evaluations have been based on the proxy dataset from Kaggle. To get a better idea of the model's performance in the real world, we further evaluate the performance of the fine-tuned 0.22B parameter CodeT5-base from the tool on real user data that was collected during the user studies. The distribution of NL queries collected from the user studies represents the "true" distribution of queries that can be expected from users in a low-code setting. Out of the 218 samples that were collected, we found only one sample in which a user explicitly specified a hyper-parameter value in their query. We therefore only compute the accuracy of the operation name generated rather than the entire operation invocation (as they would use default values anyway and so the scores remain the same except for that one sample).

Out of 218 query requests, the fine-tuned CodeT5-base model that was used in our tool answered 150 queries correctly, which would suggest an overall accuracy of 68.8%. However, 33 of these requests targeted actions that are not supported by the sklearn API, such as dropping a column (commonly the territory of the Pandas library). Disregarding such unsupported usage, LowCoder$_{NL}$ answered 141 out of 185 queries correctly for an overall accuracy of **76.2%**. For 33 additional samples, neither annotator could infer a reasonable ground truth since the prompt was unclear (e.g.: "empty"). Leaving these out, i.e., when the prompt is both clear *and* the operator is supported by the tool, LowCoder$_{NL}$ was accurate in over **90%** (137/152) of completions (refer to Appendix F for additional results).

5.2 User Study

We conducted a user study with 20 participants with varying levels of AI expertise to create AI pipelines using LowCoder across four tasks, replacing LowCoder$_{NL}$ with a simple keyword search in half the tasks. We collect and analyze data to investigate the following research questions:

- RQ1: How do LowCoder$_{NL}$ and other features help participants discover previously-unknown operators?
- RQ2: Are participants able to compose and then iteratively refine AI pipelines in our tool?
- RQ3: What are the benefits and challenges of integrating language models with visual programming for low-code?

5.2.1 Study Methodology. We recruited 20 participants within the same large technology company via internal messaging channels. We expect that citizen developers without formal programming training may also have varying levels of AI expertise and intentionally solicited participants of all backgrounds. Potential participants filled out a short pre-study survey to self-report experience in the following: machine learning, data preprocessing, and sklearn using a 1 (no experience) to 5 (expert) scale. Participants include a mix of roles including developers, data scientists, and product managers

working in a variety of domains such as AI, business informatics, quantum computing, and software services. 25% of the participants are female and the remaining 75% are male. 40% of the participants self-reported being novices in machine learning by indicating a 1 or 2 in the pre-study survey.

The study design is within-subjects [11] where each participant was exposed to two conditions: using LowCoder with (*NL condition*) and without (*keyword condition*) the natural language (NL) interface powered by LowCoder$_{NL}$. The keyword condition used a simple substring filter for operator names. Each participant performed four tasks (described below) total across the two conditions. For each participant, the order of the conditions and the order of the tasks were shuffled such that there is a uniform distribution of the order of conditions and tasks.

As our study included machine learning novices, we gave each participant a short overview of the basics of machine learning with tabular datasets and data preprocessing. We avoided using specific terms or names of operators in favor of more general descriptions of data-related problems.

We then gave each participant an overview of LowCoder. To mitigate potential biasing or priming, the tool overview used a fifth dataset from the UCI repository [14]. To avoid operators that were potentially useful in user tasks, the overview used both a non-sklearn operator that was not available in the study versions of the tool as well as sklearn's DummyClassifier that generates predictions without considering input features. Participants were allowed to use external resources such as web search engines or documentation pages. Nudges were given by the study administrators after five minutes if necessary to help participants progress in a task. Nudges were in the form of reminders to use tool features such as the NL interface, external resources, or to include missing steps such as data preprocessing or classifiers. Nudges did not mention specific operator names nor guidance on specific actions to take.

For each version of the tool, study administrators would describe the unique features of the particular version and then have participants perform tasks using two out of four sample datasets. After performing tasks using both versions of the tool and all four sample datasets, participants were asked to provide open-ended feedback and/or reactions for both LowCoder and the comparison between the NL and keyword conditions.

5.2.2 Tasks Description. Each participant performed a total of four tasks. For each task, participants were instructed to create AI pipelines with data preprocessing and classifier steps on a sample dataset with as high a score (accuracy on the test set) as possible during a time period of five to ten minutes. Each sample dataset was split beforehand into separate train and test sets. Tasks were open-ended with no guidance on what preprocessing steps or classifiers should be used.

There were four sample datasets in total and each participant was exposed to all four. The sample datasets are based on public tabular datasets from the UCI Machine Learning Repository [14], as follows:

- Dataset A modifies the Iris dataset to include missing values in the form of not-a-number (NaN) for 30% of values.
- Dataset B is the Covertype dataset and demonstrates features with differing scales.
- Dataset C is the Digits dataset and demonstrates relatively higher dimensionality.
- Dataset D is a modified version of the Mushroom dataset that only contains categorical features.
- The tutorial uses the Abalone dataset.

While Datasets A and D require a specific data preprocessing step in order to successfully create a pipeline, B and D do not technically require preprocessing to proceed. The specific datasets and train/test splits used are also available as artifacts.

Table 2: Incidence of tasks where participants find previously-unknown operators per condition (40 tasks for all, 16 tasks by novices, and 24 by non-novices). Note that rows may not sum to 100% as participants can use multiple methods to discover operators for a given task or not discover operators at all.

Condition	Participant	Method of Discovery		
		LowCoder$_{NL}$	Web search	Palette
NL	All	30 (75.0%)	5 (12.5%)	5 (12.5%)
	Novice	8 (50.0%)	2 (12.5%)	4 (25.0%)
	Non-Novice	22 (91.7%)	3 (12.5%)	1 (4.2%)
Keyword	All	*Not available in this condition.*	13 (32.5%)	11 (27.5%)
	Novice		3 (18.8%)	5 (31.3%)
	Non-Novice		10 (41.7%)	6 (25.0%)

5.2.3 Data Collection and Analysis. To answer our research questions, for each participant, we collect and analyze both quantitative and qualitative data. For quantitative data, we report on the incidence of participants discovering a previously-unknown operator (RQ1) and the incidence of completing the task and iterating or improving the pipeline (RQ2). We consider an operator 'previously-unknown' if the participant found and used the operator without using the exact or similar name. For example, using an NL query such as *"deal with missing values"* to find the `SimpleImputer` operator is considered discovering a previously-unknown operator while a query such as *"simpleimpute"* is not. We report discovery using the following methods: through LowCoder$_{NL}$, generic web search engine (Google), and scrolling through the palette. Participants may discover multiple unknown operators during the same task, possibly using different methods. For each participant's task, we consider it 'complete' if the composed pipeline successfully trains against the dataset's training set and returns a score against the test set. We consider the pipeline iterated if a participant modifies an already-complete pipeline. More specifically, we consider the following forms of iteration: a preprocessing operator block is added or swapped, a classifier block is swapped, or hyper-parameters are tuned. We report each of these as separate types of pipeline iteration. Participants may perform multiple types of iteration during the same task. Both sets of quantitative metrics are counted per task (80 tasks total for 20 participants, 40 tasks per condition).

We use qualitative data to answer RQ3. This data focuses on the participants' actions in LowCoder, commentary while using the tool and performing tasks, and answers to open-ended questions after the study. Specifically, the same two authors that administered the user study analyzed the notes generated by the study along with the audio and screen recordings when the notes were insufficient, using discrete actions and/or quotations as the unit of analysis. The

first round of analysis performed open coding [11] on data from 16 studies to elicit an initial set of 73 themes. The two authors then iteratively refined the initial themes through discussion along with identifying 13 axial codes which are summarized in Figure 7. The same authors then performed the same coding process on a hold-out set of 4 studies. No additional themes were derived from the hold-out set of studies, suggesting saturation.

5.2.4 Study Results. We answer RQ1 and RQ2 using quantitative data collected from observing participant actions per task and answer RQ3 through open coding of qualitative data.

RQ1: How do LowCoder$_{NL}$ and other features help participants discover previously-unknown operators?

A known limitation of visual programming is discoverability [27]. Table 2 reports how often participants discovered previously-unknown operators during their tasks. 80% of the participants discovered an unknown operator across 63.8% of all 80 tasks in the study. Participants discovered unknown operators in 82.5% of the 40 NL condition tasks compared to 45% of the 40 keyword condition tasks. The odds of discovering an unknown operator are significantly greater in the NL condition than keyword ($p \ll 0.001$) using Barnard's exact test. We examine the methods of discovery in more detail, noting that LowCoder$_{NL}$ is only available in the NL condition whereas web search and scrolling through the operator palette are available in both conditions. Participants were not able to use the keyword search to discover unknown operators due to needing at least part of the exact name. Using LowCoder$_{NL}$, participants discovered unknown operators in 75% of tasks in the NL condition as opposed to an average of 22.5% using web search engines (12.5% in the NL condition and 32.5% in the keyword condition) and an average of 20% by scrolling through the operator palette (12.5% in the NL condition and 27.5% in the keyword condition). Within the NL condition, the odds of an unknown operator being discovered are significantly greater using LowCoder$_{NL}$ as opposed to both web search ($p \ll 0.001$) and scrolling ($p \ll 0.001$). When splitting on the experience of the participant, we find statistically greater chances of novices discovering operators in the NL condition using LowCoder$_{NL}$ as opposed to web search (p=0.013) but not scrolling (p=0.086). Non-novices were significantly more likely to discover operators using LowCoder$_{NL}$ compared to web search or scrolling ($p \ll 0.001$, $p \ll 0.001$). Results do not change if considering web searches or scrolling across all 80 tasks. These results suggest that LowCoder$_{NL}$ is particularly helpful in discovering previously-unknown operators, especially compared to web search, but novices still face some challenges. We discuss these challenges in RQ3.

RQ2: Are participants able to compose and then iteratively refine AI pipelines in our tool?

Machine learning development is intensely iterative [39] and tools should support this. Table 3 reports how often participants iterated on pipelines. Participants completed 82.5% of the 80 tasks in the study and further iterated their pipelines in 72.5% of the tasks. Splitting on condition, the NL condition has 85% task completion and 72.5% further iteration while the keyword condition has 80% task completion and 72.5% iteration rate. Swapping classifiers was the most common form of iteration at 48.8%, followed by adding or swapping preprocessors at 43.8% and setting hyper-parameters at 30%. Comparing novices to non-novices, both types of participants

Table 3: Incidence of tasks where participants complete and iterate on preprocessors, classifiers, and hyper-parameters.

Iteration Type	*Total Tasks* (80)	*Novice* (32)	*Non-Novice* (48)
Task Completion	66 (82.5%)	21 (65.6%)	45 (93.8%)
Swap Classifier	39 (48.8%)	11 (34.4%)	28 (58.3%)
Add/Swap Preprocessors	35 (43.8%)	15 (46.9%)	20 (41.7%)
Set Hyper-parameters	24 (30.0%)	4 (19.0%)	20 (41.7%)
All Iterations	58 (72.5%)	20 (62.5%)	38 (79.2%)

are mostly successful in iterating pipelines with no significant differences in iteration rate using Barnard's exact test (p=0.109). This result holds when iterating preprocessors (p=0.664) but not classifiers (p=0.038) nor hyper-parameters (p=0.005). Non-novices are more likely to complete the task than novices (p=0.002). Regardless of experience, both novices and non-novices are able to iteratively refine their pipelines, but novices face some challenges compared to non-novices regarding actually completing the task. These challenges are discussed in the next research question.

RQ3: What are the benefits and challenges of integrating language models with visual programming for low-code?

Figure 7 shows our 13 axial codes for answering RQ3. These codes broadly represent three overarching themes regarding combining visual programming and language models for low-code:

1) *Discovery* of machine learning operators relevant for the task at hand, 2) *Iterative Composition* of the operators in the tool, and 3) *Challenges* that participants, particularly novices, face regarding working with machine learning and/or using low-code tools. We also collect *Feedback* from participants to inform future development of LowCoder. Due to space limitations, we only report on a selection of the 13 axial codes and 73 codes derived from open coding (refer to Appendix G for the full list of codes).

For the first category of **Discovery**, our analysis derived two axial codes related to the participants' goal while attempting to discover operators: 1) *Know "What" Not "How"* where participants have a desired action in mind but do not know the exact operator that performs that action (19 out of 20 participants experienced this axial code) and 2) *Know "What" And "How"* where participants have a particular action and operator in mind (18/20). We dive deeper into *Know "What" Not "How"* which includes the code where participants *Discover a previously-unknown operator using NL* (16/20). We found in RQ1 that LowCoder$_{NL}$ was helpful in finding unknown operators compared to other methods. The qualitative data suggests that participants were able to find unknown operators using LowCoder$_{NL}$ during cases where they have an idea of the action to perform but do not know the exact operator name for a variety of reasons. For example, when discovering `SimpleImputer` with LowCoder$_{NL}$, P11 noted that they *"never used SimpleImputer but had an idea of what I wanted to do, even though I generally remove NaNs in Pandas."* Another example is P16 who *"preferred the [NL version of LowCoder], even when I was doing Google searches, they... didn't give me options, your tool at least returns some options that I can try out and swap out."* As a novice, P16 had difficulties finding the names of useful operators from web search results as opposed to the LowCoder$_{NL}$ which directly returned actionable operators. Challenges regarding general web search is also an axial code.

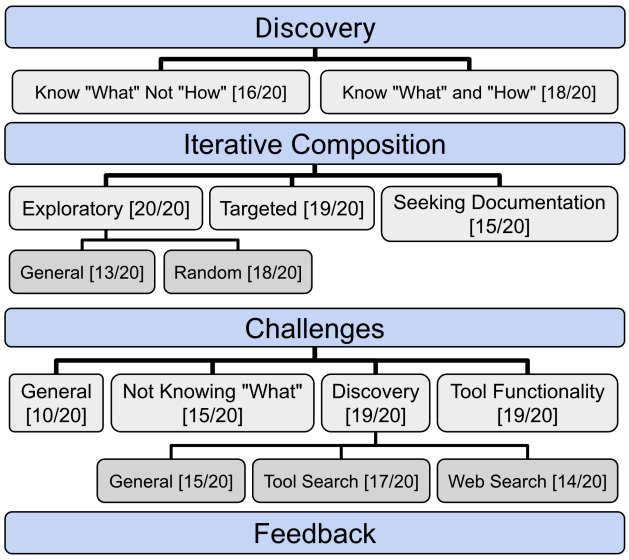

Figure 7: Axial codes from our qualitative analysis.

For the second category of **Iterative Composition**, we derived four axial codes related to participant behaviors while attempting to compose and iterate on pipelines: 1) *General Exploratory* (13/20) iteration, 2) Exploratory iteration but where participants will select operators or hyper-parameters seemingly at *Random* (18/20), 3) *Targeted* (19/20) iteration where participants select operators or hyper-parameters with a particular intent, and 4) *Seeking Documentation* (15/20) where participants search for documentation to inform iteration decisions. For both forms of Exploratory iteration and Targeted iteration, we find examples of participants iterating classifiers, preprocessors, and hyper-parameters. For the axial code of seemingly *Random* iteration, participants, especially (but not exclusively) novices, when unsure of how to proceed, tended to try out arbitrary preprocessors or classifiers. This was more common for more difficult tasks that required particular data preprocessing to proceed. For example, non-novice P9 remarked *"I'm not familiar enough with it, so do I Google it or brute force it? [...] I don't even know what to Google to figure this out... I guess I'll do some light brute-forcing"* and proceeded to swap in and out preprocessors from the palette. In contrast, the axial code of *Targeted* (19/20) iteration has codes that reflect particular intentions that participants derived from observations within the tool, such as *Noticing error messages* (10/20) or *Making use of data tables in task* (14/20). As an example of the data tables case, P11 realized through the *Before* data table that the given dataset had *"too many columns"* and added the `IncrementalPCA` operator along with setting its `n_components` hyper-parameter to 5. Upon seeing the change in data in the *After* data table, they remarked, *"Wow... I really like that I can see all the hyper-parameters that I can play with"* and proceeded to tune various hyper-parameters.

The third category is the variety of **Challenges** that participants faced while using LowCoder and performing the machine learning tasks, where we derive six axial codes: 1) *General* challenges (10/20) faced by participants that are not particular to our tool or tasks, 2) *Not Knowing "What"* (15/20) where participants experienced difficulties due to knowing neither "what" nor "how" to begin, 3) *General Discovery* challenges (15/20), 4) Discovery challenges around using *Web search* (14/20), 5) Discovery challenges when using *Tool search* (17/20) or specifically using LowCoder$_{NL}$, and 6) *Tool Functionality* (19/20) which describes challenges participants faced using (or not using) LowCoder features. We dive deeper into the axial code of *Not Knowing "What"* and note its contrast to the *Know "What" Not "How"* axial code where participants may have intentions but not know how to execute them or the *Exploratory* iteration axial code where participants may not have specific intentions but know how to iterate. All novices (8/8) and most non-novices (7/12) experienced this challenge. The primary code is that participants *Did not know "what" they wanted to do* (11/20). One possible cause of this lack of progression is choice paralysis, for example on P17's first task, *"first things first, I don't even know where to begin... right now it's super overwhelming, I guess I'll start throwing stuff in there."* We also describe the axial code of *Tool search* (17/20) where participants had difficulties forming search queries for LowCoder$_{NL}$.

Participants noted that despite the interface being intended for general natural language, the interface still *Needed a specific vocabulary* (8/20). As P19, a novice, described it, *"I get the idea of how it's supposed to work but it's hit and miss... even if I use very layman's terms... it expects a non-naive explanation of what needs to be done."* Part of this challenge may be due to a mismatch in the natural language in Kaggle notebooks used to train LowCoder$_{NL}$ and the language used by novices.

6 REFLECTION OF PRACTICAL AND SOCIETAL IMPACT

Our results show that the integration of LowCoder$_{VP}$ with LowCoder$_{NL}$ was helpful with aspects like operator discovery (RQ1) or iteratively composing pipelines (RQ2), even for novice participants. Through our work, we hope to help with the democratization of AI by supporting users with varying levels of AI expertise. LowCoder is especially useful for citizen developers who have an idea of *what* they would like to do but do not fully know *how* to accomplish that, perhaps due to a lack of formal programming training. In fact, our qualitative analysis (RQ3) reveals that a number of our participants (including all novices who participated) struggled with knowing *what* to do. End-users writing software face similar "design barriers" [22], where it is difficult for a non-programmer to even conceptualize a solution. In contrast to other popular low-code domains such as traditional software [31], the domain of developing machine learning pipelines is particularly difficult in this regard due to its experimental nature, where progress has a high degree of uncertainty [39]. This uncertainty then requires an abundance of judgment calls that rely heavily on prior machine learning experience [19] that novices lack. Some participants in our studies echo this, identifying that some ML knowledge is necessary to use our tool. That suggests that our low-code approach may be best-suited for citizen developers who have some domain knowledge but lack programming training, such as statisticians for the low-code domain

of machine learning. A further improved low-code machine learning tool could thus be made more suitable towards novice citizen developers by guiding them to discover the *what* along with the *how*, i.e., by helping developers acquire the necessary ML knowledge.

Assisting novices without domain knowledge may then require low-code approaches that are orthogonal to both visual programming and language models. One such approach, suggested by a study participant, is to provide suggestions in the form of templates or recipes for pipelines. These suggestions could also be contextual to the given dataset or active pipeline, for example automatically suggesting encoders when detecting categorical features. Ko et al. [22] also suggest templates as a possible solution for design barriers. A related suggestion made by a number of our study participants is data visualization and summarization for the given dataset, such as plots, charts, confusion matrices, etc. These visualizations could themselves inform contextual suggestions – a histogram detecting a non-standard distribution may suggest the need for a `StandardScaler`. These contextual suggestions may also help in guiding developers in *what* to do, making for a more generally useful low-code tool for both citizen and experienced developers alike. Additionally, some visual programming languages risk vendor lock-in; we avoid that problem by backing LowCoder$_{VP}$ with a pre-existing, open-source DSL with the Lale library.

Threats to Validity: The user study for LowCoder has several limitations. The study focused on relatively small, public tabular datasets and sklearn operators and may not be indicative of other machine learning tasks such as deep learning on large datasets. Participants also all come from the same large technology company and may not be representative of general users. However, we did intentionally elicit participation from a variety of groups and experience levels to mitigate this. As our user study has a within-subjects design, there may be potential learning effects between tasks and conditions. In fact, we observed some cases of this (8/20), with some participants explicitly mentioning selecting particular operators due to the previous task. We mitigated this learning effect by randomizing the order of tasks and conditions, as well as by having two tasks (A and D) require the use of preprocessing operators that were not applicable to other tasks.

7 CONCLUSION

We developed LowCoder, a low-code tool that combines visual programming (via a block-based editor, LowCoder$_{VP}$) with programming by natural language (via a language model, LowCoder$_{NL}$) to help developers of all backgrounds create AI pipelines. We used LowCoder to provide some of the first insights into whether and how the integration of visual programming and language models help programmers by conducting user studies across four tasks with (NL condition) and without (keyword condition) LowCoder$_{NL}$. Overall, LowCoder helped developers compose (85% of tasks) and iterate (72.5% of tasks) over ML pipelines. Furthermore, LowCoder$_{NL}$ helped users discover previously-unknown operators in 75% of tasks, compared to just 22.5% (12.5% in the NL condition and 32.5% in the keyword condition) using web search. Our qualitative analysis showed that language models helped users discover *how* to implement various parts of the pipeline when they know *what* to do. However, they failed to support novices when they lacked clarity on what they want to accomplish, which may suggest a worthwhile target for improving AI-based program assistants. Our work demonstrates the promise of combining both a language model powered natural language interface and a visual interface for low-code programming.

8 DATA AVAILABILITY

The implementation of LowCoder, datasets for training and evaluating LowCoder$_{NL}$, results of additional experiments, as well as the material from the user study, including the full set of (axial) codes and anonymized quantitative and qualitative data, are available at: https://doi.org/10.5281/zenodo.7042296.

REFERENCES

[1] 2021. GitHub Copilot. https://github.com/features/copilot

[2] 2022. ChatGPT. https://openai.com/blog/chatgpt/

[3] Ion Androutsopoulos, Graeme D. Ritchie, and Peter Thanisch. 1995. Natural Language Interfaces to Databases – An Introduction. *Natural Language Engineering* 1, 1 (1995), 29–81. https://doi.org/10.1017/S135132490000005X

[4] Guillaume Baudart, Martin Hirzel, Kiran Kate, Parikshit Ram, Avraham Shinnar, and Jason Tsay. 2021. Pipeline Combinators for Gradual AutoML. In *Advances in Neural Information Processing Systems (NeurIPS)*. https://proceedings.neurips.cc/paper/2021/file/a3b36cb25e2e0b93b5f334ffb4e4064e-Paper.pdf

[5] J. Bergstra, D. Yamins, and D. D. Cox. 2013. Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures. In *International Conference on Machine Learning (ICML)*. I–115–I–123.

[6] Michael R. Berthold, Nicolas Cebron, Fabian Dill, Thomas R. Gabriel, Tobias Kötter, Thorsten Meinl, Peter Ohl, Kilian Thiel, and Bernd Wiswedel. 2009. KNIME - the Konstanz Information Miner: Version 2.0 and Beyond. *ACM SIGKDD Explorations Newsletter* 11, 1 (Nov. 2009), 26–31. https://doi.org/10.1145/1656274.1656280

[7] Marat Boshernitsan and Michael Downes. 2004. *Visual Programming Languages: A Survey*. Technical Report UCB/CSD-04-1368. University of California, Berkeley. https://digitalassets.lib.berkeley.edu/techreports/ucb/text/CSD-04-1368.pdf

[8] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Conference on Neural Information Processing Systems (NeurIPS)*. 1877–1901. https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

[9] Sarah E. Chasins, Maria Mueller, and Rastislav Bodik. 2018. Rousillon: Scraping Distributed Hierarchical Web Data. In *Symposium on User Interface Software and Technology (UIST)*. 963–975. https://doi.org/10.1145/3242587.3242661

[10] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harri Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, Will Guss, Alex Nichol, Igor Babuschkin, Suchir Balaji, Shantanu Jain, Andrew Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. https://arxiv.org/abs/2107.03374

[11] John W. Creswell. 2013. *Research design: Qualitative, quantitative, and mixed methods approaches* (4th ed.). SAGE publications.

[12] Janez Demsar, Blaz Zupan, Gregor Leban, and Tomaz Curk. 2004. Orange: From Experimental Machine Learning to Interactive Data Mining. In *European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD)*. 537–539. https://doi.org/10.1007/978-3-540-30116-5_58

[13] Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, Sailesh R, and Subhajit Roy. 2016. Program Synthesis Using Natural Language. In *International Conference on Software Engineering (ICSE)*. 345–356. https://doi.org/10.1145/2884781.2884786

[14] Dheeru Dua and Casey Graff. 2017. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml

[15] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 1536–1547. https://aclanthology.org/2020.findings-emnlp.139/

[16] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. 2015. Efficient and Robust Automated Machine Learning. In *Conference on Neural Information Processing Systems (NIPS)*. 2962–2970. http://papers.nips.cc/paper/5872-efficient-and-robust-automated-machine-learning

[17] D Randy Garrison, Martha Cleveland-Innes, Marguerite Koole, and James Kappelman. 2006. Revisiting methodological issues in transcript analysis: Negotiated coding and reliability. *The Internet and Higher Education* 9, 1 (2006), 1–8.

[18] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. 2009. The WEKA Data Mining Software: An Update. *SIGKDD Explorations Newsletter* 11, 1 (Nov. 2009), 10–18. http://doi.acm.org/10.1145/1656274.1656278

[19] C Hill, R Bellamy, T Erickson, and M Burnett. 2016. Trials and tribulations of developers of intelligent systems: A field study. In *Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 162–170.

[20] Martin Hirzel. 2023. Low-Code Programming Models. *Communications of the ACM (CACM)* 66, 10 (Oct. 2023), 76–85. https://doi.org/10.1145/3587691

[21] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).

[22] Amy J. Ko, Brad A. Myers, and Htet Htet Aung. 2004. Six Learning Barriers in End-User Programming Systems. In *Symposium on Visual Languages – Human Centric Computing (VL/HCC)*. https://doi.org/10.1109/VLHCC.2004.47

[23] Toby Jia-Jun Li, Marissa Radensky, Justin Jia, Kirielle Singarajah, Tom M. Mitchell, and Brad A. Myers. 2019. PUMICE: A Multi-Modal Agent That Learns Concepts and Conditionals from Natural Language and Demonstrations. In *Symposium on User Interface Software and Technology (UIST)*. 577–589. https://doi.org/10.1145/3332165.3347899

[24] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *ArXiv* abs/2102.04664 (2021).

[25] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. A Conversational Paradigm for Program Synthesis. https://arxiv.org/abs/2203.13474

[26] Randal S. Olson and Jason H. Moore. 2016. TPOT: A Tree-based Pipeline Optimization Tool for Automating Machine Learning. In *Workshop on Automatic Machine Learning (AutoML)*. 66–74. https://proceedings.mlr.press/v64/olson_tpot_2016.html

[27] Erik Pasternak, Rachel Fenichel, and Andrew N. Marshall. 2017. Tips for Creating a Block Language with Blockly. In *Blocks and Beyond Workshop (B&B)*. https://doi.org/10.1109/BLOCKS.2017.8120404

[28] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[29] Felipe Pezoa, Juan L. Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. 2016. Foundations of JSON Schema. In *International Conference on World Wide Web (WWW)*. 263–273. https://doi.org/10.1145/2872427.2883029

[30] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2018. Language Models are Unsupervised Multitask Learners. (2018). https://d4mucfpksywv.cloudfront.net/better-language-models/language-models.pdf

[31] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. 2009. Scratch: Programming for All. *Communications of the ACM (CACM)* 52, 11 (Nov. 2009), 60–67. https://doi.org/10.1145/1592761.1592779

[32] Apurvanand Sahay, Arsene Indamutsa, Davide Di Ruscio, and Alfonso Pierantonio. 2020. Supporting the understanding and comparison of low-code development platforms. In *Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 171–178. https://doi.org/10.1109/SEAA51224.2020.00036

[33] Steven L. Tanimoto. 2013. A perspective on the evolution of live programming. In *International Workshop on Live Programming (LIVE)*. 31–34. https://doi.org/10.1109/LIVE.2013.6617346

[34] Chris Thornton, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2013. Auto-WEKA: Combined Selection and Hyperparameter Optimization of Classification Algorithms. In *Conference on Knowledge Discovery and Data Mining (KDD)*. 847–855. https://doi.org/10.1145/2487575.2487629

[35] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. 2022. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In *Conference on Human Factors in Computing Systems (CHI)*. Article 332. https://doi.org/10.1145/3491101.3519665

[36] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[37] Mandana Vaziri, Louis Mandel, Avraham Shinnar, Jérôme Siméon, and Martin Hirzel. 2017. Generating Chat Bots from Web API Specifications. In *Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*. 44–57. http://doi.acm.org/10.1145/3133850.3133864

[38] Markus Voelter and Sascha Lisson. 2014. Supporting Diverse Notations in MPS' Projectional Editor.. In *Workshop on The Globalization of Modeling Languages (GEMOC)*. 7–16. https://hal.inria.fr/hal-01074602/file/GEMOC2014-complete.pdf#page=13

[39] Zhiyuan Wan, Xin Xia, David Lo, and Gail C. Murphy. 2019. How does Machine Learning Change Software Development Practices? *Transactions on Software Engineering (TSE)* (2019). https://doi.org/10.1109/TSE.2019.2937083

[40] Dakuo Wang, Justin D. Weisz, Michael Muller, Parikshit Ram, Werner Geyer, Casey Dugan, Yla Tausczik, Horst Samulowitz, and Alexander Gray. 2019. Human-AI Collaboration in Data Science: Exploring Data Scientists' Perceptions of Automated AI. *Proc. ACM Hum.-Comput. Interact.* 3, CSCW, Article 211 (nov 2019), 24 pages. https://doi.org/10.1145/3359313

[41] Yushi Wang, Jonathan Berant, and Percy Liang. 2015. Building a semantic parser overnight. In *Annual Meeting of the Association for Computational Linguistics (ACL)*. 1332–1342. https://www.aclweb.org/anthology/P15-1129.pdf

[42] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 8696–8708. https://aclanthology.org/2021.emnlp-main.685/

[43] Frank F. Xu, Bogdan Vasilescu, and Graham Neubig. 2022. In-IDE Code Generation from Natural Language: Promise and Challenges. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 2, Article 29 (mar 2022). https://doi.org/10.1145/3487569

A TASK FORMULATIONS

Table 4 contains additional examples of NL queries and the corresponding code based on the task formulation.

B ADDITIONAL DETAILS ABOUT THE DATA

We find that the NL query corresponds to a single sklearn operator invocation in the code cell in 62% of the data, whereas in the remaining 38% it has multiple sklearn operator invocation statements. Table 5 shows the distribution of hyper-parameters in hybrid operator invocations, based on whether the hyper-parameters were named, and whether the hyper-parameter values were masked or valued (based on whether they appear in the NL query).

C DECODING TECHNIQUES

We experiment with different decoding techniques to generate our output hybrid operation invocation sequence. We describe them below:

(i) **Greedy decoding:** At each time step, greedy decoding chooses the token with the highest conditional probability. Since the model weights are fixed, the output is deterministic, always yielding the same generation for a given prompt. We use this to generate a single output sequence.

(ii) **Top K sampling:** At each time step, only consider the top k most probable tokens (according to the model). Renormalize their probabilities and select a token corresponding to these probabilities. The output of this approach is no longer deterministic, but instead explores multiple, predominantly high-probability, completion paths. For our experiments, we set the value of k to 5 and generate a total of 5 output sequences.

(iii) **Nucleus sampling:** Nucleus sampling is similar to top k sampling, but rather than fixing the number of most-probable tokens to consider at each time step, it samples a variable

Table 4: Examples of NL-Code pairs for different task formulations.

NL query	Operator Name	Complete Operator Invocation	Masked Operator Invocation	Hybrid Operator Invocation
Split X, y data into training set and testing set	train_test_split	train_test_split(X, y, test_size=0.2)	train_test_split(MASK, MASK, test_size=MASK)	train_test_split(X, y, test_size=MASK)
PCA with 2 components	PCA	PCA (n_components=2, random_state=42)	PCA (n_components=MASK, random_state=MASK)	PCA (n_components=2, random_state=MASK)
Replace missing data with the mean value	SimpleImputer	SimpleImputer (strategy='mean')	SimpleImputer (strategy=MASK)	SimpleImputer (strategy='mean')
Encoding categorical features	OneHotEncoder	OneHotEncoder()	OneHotEncoder()	OneHotEncoder()
Standardisation of Data	StandardScaler	StandardScaler()	StandardScaler()	StandardScaler()
K-Means with 4 clusters	KMeans	KMeans (n_clusters=4, random_state=42)	KMeans (n_clusters=MASK, random_state=MASK)	KMeans(n_clusters=4, random_state=MASK)
Build Decision Tree with max_depth = 7	DecisionTreeClassifier	DecisionTreeClassifier (criterion='gini', max_depth=7)	DecisionTreeClassifier (criterion=MASK, max_depth=MASK)	DecisionTreeClassifier (criterion=MASK, max_depth=7)
Random forest with balanced class weight	RandomForestClassifier	RandomForestClassifier (n_estimators=100, class_weight='balanced')	RandomForestClassifier (n_estimators=MASK, class_weight=MASK)	RandomForestClassifier (n_estimators=MASK, class_weight='balanced')

Table 5: Distribution of hyper-parameters in hybrid operator invocations.

Parameter Type	Distribution of Parameters (%)		
	0	1-3	4+
Total	18.49	61.51	19.99
Named	54.82	39.16	6.01
Masked	18.99	61.61	19.39
Valued	96.97	3.02	0.01

number of tokens whose cumulative conditional probabilities reaches or exceeds a defined probability (p) value.[2] In contexts where just one or two tokens are highly probable or where many tokens are similarly plausible, this allows the model to switch between sampling more greedily or uniformly respectively. Once the samples are chosen, the probabilities are again redistributed among them and a token is selected according to these probabilities. In our experiments, we set the p value to be 0.9 and generate a total of 5 output sequences.

D MODELING RESULTS

We benchmark the performance of all the models and record the inference time across several different variables in Table 6, namely:

(i) **Learning strategy:** we look at three different learning strategies for training the models, which include: training a sequence to sequence Transformer from scratch, fine-tuning a CodeT5 model, and few-shot prompting CodeGen, a large language model.

(ii) **Model size:** we compare a range of sizes for both the CodeT5 and CodeGen models.

(iii) **Decoding:** we compare the different decoding strategies that include, greedy (top-1), topk (top-5) and nucleus (top-5).

(iv) **Tool supported operations:** Recall that since sklearn pipelines can only contain operators but not functions, LowCoder only exposes blocks for operators. Therefore, we also contrast the performance of the model on all the test data points (total of 7,351 samples) along with a filtered set of test data points that only include operators supported by the tool (resulting in 3,941 samples).

We also perform additional experiments in an effort to reduce the inference time of the fine-tuned CodeT5 model with top-5 decoding for integration with the tool. We reduce the output sequence length from 512 tokens to 64 and find a negligible decline in accuracy (73.20% for 512 tokens vs. 72.81% for 64 tokens) with significantly lower inference time (2.37s vs. 1.56s per sample). Note that the inference time is not proportional to the number of tokens as the model learns to stop generating new tokens when it hits the end token.

E ANNOTATION GUIDELINES FOR REAL USER DATA

Two authors manually annotate the real user data by looking at the NL query and the predictions returned by LowCoder$_{NL}$. More specifically, we look at the following criteria during annotation (multiple annotations are possible per query):

- Accurate prediction: At least one of the predictions returned by the model matches the inferred user intent in the query. Inferred intent was determined by annotator domain knowledge.

[2]This approach is also called "top-p sampling".

Table 6: Accuracy scores for Hybrid Operator Invocation task across different model variations (*due to memory constraints)

Model	Size	Decoding	Test notebooks				
			Time (s)	Accuracy in k (%)			
				OpName		OpInvocation	
				all	tool	all	tool
Transformer	6 layers	greedy (n=1)	0.20	43.23	47.39	16.81	14.13
		topK (n=5)	1.11	60.13	63.74	29.16	28.21
		nucleus (n=5)	1.21	59.62	62.47	29.99	27.88
Fine-tuned CodeT5	small	greedy (n=1)	0.76	57.43	64.19	23.88	21.44
		topK (n=5)	1.09	71.78	77.01	38.86	39.25
		nucleus (n=5)	1.52	71.20	76.50	39.57	39.81
	base	greedy (n=1)	1.55	59.37	65.66	25.84	23.97
		topK (n=5)	2.37	73.20	78.07	40.04	39.93
		nucleus (n=5)	3.15	73.35	77.94	41.19	40.97
	large	greedy (n=1)	3.66	60.01	64.96	26.90	24.56
		topK (n=5)	5.89	73.57	77.47	41.27	40.73
		nucleus (n=5)	6.50	73.22	77.16	40.27	39.63
CodeGen	350M	greedy (n=1)	4.39	19.65	25.09	1.94	3.35
		topK (n=5)	5.05	46.02	57.59	9.66	15.25
		nucleus (n=5)	5.42	43.85	55.16	9.65	15.19
	2.7B	greedy (n=1)	6.52	22.67	30.55	2.41	4.21
		topK (n=5)	8.59	48.63	60.62	9.48	15.81
		nucleus (n=5)	9.56	47.33	59.27	9.31	15.20
	6.1B	greedy (n=1)	8.09	26.01	35.70	3.51	6.06
		topK (n=5)	10.24	49.79	61.94	11.32	17.94
		nucleus (n=5)	10.43	48.35	60.39	11.03	17.20
	16.1B	greedy (n=1)	10.99	24.60	34.66	3.44	5.40
		topK (n=3*)	14.27	43.27	55.41	9.73	13.98
		nucleus (n=3*)	14.25	41.34	53.89	10.01	14.62

- NL unclear: The NL query is unclear (e.g. *"empty"*, *"numbers"*, *"variable"*) when neither annotator could infer the intent from the query alone.
- Partially correct: The predictions are partially correct. This usually happened if the NL query requests multiple operators or intents such as *"normalize features and run linear regression"*.
- Not supported by tool: The NL query requests targeted actions that are not supported by the sklearn.
- No output returned by model: These are the cases where the model fails to return any usable predictions.

Table 7 has the distribution of data per task for all the different properties we look at when manually annotating the real user data.

F MODEL EVALUATION ON REAL USER DATA

We evaluate the performance of LowCoder$_{NL}$ on the annotated real user data. Here is a summary of the findings:

- Total accuracy = 150/218 = 68.80%
- Percentage of data where the NL query is clear = 178/218 = 81.65%
- Percentage of data where the NL query is not clear = 40/218 = 18.35%
- Accuracy of model when the NL query is clear = 145/178 = 81.46%
- Accuracy of model when the operator is supported by tool = 141/185 = 70.81%
- Accuracy of model when the NL query is clear and operator is supported by tool = 137/152 = 90.13%

G USER STUDY RESULTS

The following is the full listing of codes and axial codes from the qualitative analysis, broken down by high-level category (which corresponds to 1st level axial codes): 1) Discovery (Table 8), Iterative Composition (Table 9), Challenges (Table 10), and Feedback (Table 11).

Table 7: Distribution of various properties annotated manually for real user data.

Task	Total	Accurate prediction	NL unclear	Partially correct	Not supported by tool	No output returned
A	43	30	5	3	3	0
B	41	27	4	7	16	3
C	53	38	9	3	6	4
D	81	55	22	11	8	6
All	218	150	40	24	33	13

Table 8: Full codes and axial codes from qualitative analysis for Discovery category.

1st Level Axial	2nd Level Axial	Code	Participant Count (20)
Discovery	Know "What" Not "How"	Discovered operator they didn't know about using NL	16
Discovery	Know "What" Not "How"	Discovered useful operator by browsing toolbox	7
Discovery	Know "What" Not "How"	Google error message to find solution	1
Discovery	Know "What" Not "How"	Google for how to do something find general term to use in tool	13
Discovery	Know "What" Not "How"	Keyword result does not match what they want	5
Discovery	Know "What" Not "How"	Liked NLP version	13
Discovery	Know "What" Not "How"	NL search using ML terms	6
Discovery	Know "What" Not "How"	Scrolling through toolbox for something they recognize or relevant name	10
Discovery	Know "What" Not "How"	Search same as hint	1
Discovery	Know "What" and "How"	Keyword close to operator name but not exact match	6
Discovery	Know "What" and "How"	Keyword search using ML term	10
Discovery	Know "What" and "How"	Searched for exact operator name	14

Table 9: Full codes and axial codes from qualitative analysis for Iterative Composition category.

1st Level Axial	2nd Level Axial	3rd Level Axial	Code	Participant Count (20)
Iterative Composition	Exploratory	General	Liked visual blocks	5
Iterative Composition	Exploratory	General	Used score to determine how to refine	14
Iterative Composition	Exploratory	Random	Randomly refining data processing	9
Iterative Composition	Exploratory	Random	Randomly refining hyperparameters	4
Iterative Composition	Exploratory	Random	Randomly refining model	14
Iterative Composition	Exploratory	Random	Randomly scroll through toolbox	17
Iterative Composition	Exploratory	Random	Searched for exact operator name	1
Iterative Composition	Exploratory	Random	Unclear goal but pipeline worked	4
Iterative Composition	Targeted		Google for hyperparameter values	4
Iterative Composition	Targeted		Intentionally refine model	5
Iterative Composition	Targeted		Intentionally refine preprocessing	5
Iterative Composition	Targeted		Intentionally tuning hyperparameters	10
Iterative Composition	Targeted		Liked hyperparameter pane	2
Iterative Composition	Targeted		Made use of data tables in task	14
Iterative Composition	Targeted		Noticed error message	10
Iterative Composition	Targeted		Used canvas to store blocks	4
Iterative Composition	Targeted		Used score to determine how to refine	1
Iterative Composition	Seeking Documentation		Google for operator documentation	6
Iterative Composition	Seeking Documentation		Hard to figure out what operator does	10
Iterative Composition	Seeking Documentation		Hover over hyperparameters to learn more	6

Table 10: Full codes and axial codes from qualitative analysis for Challenges category.

1st Level Axial	2nd Level Axial	3rd Level Axial	Code	Participant Count (20)
Challenges	General		Gave up on task	3
Challenges	General		Ignoring/misinterpreting error messages	4
Challenges	General		Needed nudge to do something	8
Challenges	General		Task clarification	3
Challenges	Not Knowing "What"		Did not know "what" they wanted to do	11
Challenges	Not Knowing "What"		Learning effect - used exact same operator/pipeline as previous task	8
Challenges	Not Knowing "What"		Nudge to prevent invalid pipeline	2
Challenges	Not Knowing "What"		Tool easier to use when knowing "what" to do	3
Challenges	Not Knowing "What"		Unfamiliar with aspect of scikit-learn or machine learning	4
Challenges	Not Knowing "What"		Used operator that did not work as intended	4
Challenges	Discovery	General	Knew "what" they wanted to do but did not know the term	15
Challenges	Discovery	General	Overwhelmed by choices	2
Challenges	Discovery	Google Search	Chose not to Google search	3
Challenges	Discovery	Google Search	Google didn't help find sklearn operator (found pandas/numpy solution)	5
Challenges	Discovery	Google Search	Google something at a very high level "classification/preprocessing"	6
Challenges	Discovery	Google Search	Google something but not able to parse results	10
Challenges	Discovery	Google Search	Had difficulty articulating Google search	3
Challenges	Discovery	Tool Search	Had difficulty articulating search inside tool	5
Challenges	Discovery	Tool Search	NL returned unsupported operator	9
Challenges	Discovery	Tool Search	NL search is unclear or vague	10
Challenges	Discovery	Tool Search	Needed specific vocabulary	8
Challenges	Discovery	Tool Search	No results from keyword filter	9
Challenges	Discovery	Tool Search	No results returned from NLP	3
Challenges	Tool Functionality		Did not understand aspect of tool	5
Challenges	Tool Functionality		Didn't use certain tool features	6
Challenges	Tool Functionality		Learning curve required for tool	3
Challenges	Tool Functionality		NLP returned wrong results	8
Challenges	Tool Functionality		Non-deterministic results from NL search	2
Challenges	Tool Functionality		Non-deterministic scoring	4
Challenges	Tool Functionality		Problem with tool	4
Challenges	Tool Functionality		They didn't understand what NL search really did	5
Challenges	Tool Functionality		Wanted to do something tool doesn't support	11

Table 11: Full codes and axial codes from qualitative analysis for Feedback category.

1st Level Axial	Code	Participant Count (20)
Feedback	Comparison with other tool	4
Feedback	Did not like NLP version	3
Feedback	Did not like some feature of the tool	6
Feedback	Do not like keyword version	2
Feedback	Liked data table feature	8
Feedback	Liked feature of tool	12
Feedback	Liked keyword version	6
Feedback	No preference between keyword and NLP	2
Feedback	Suggestion for tool	13
Feedback	Wanted both NL and keyword	3