AI for Low-Code for AI

Anonymous Author(s)

***Abstract*—Low-code programming allows citizen developers to create programs while minimizing code in a textual language such as Java or Python. This paper introduces LOWCODER, a low-code tool for programming AI pipelines with steps for data preprocessing and predictive models from the popular scikit-learn library. LOWCODER has a visual programming interface, LOWCODER$_{VP}$, that helps developers quickly compare pipeline variants by allowing them to swap out code blocks. In addition, LOWCODER has an AI-driven natural language interface, LOWCODER$_{NL}$, that helps developers discover code blocks when they know *what* they want to do but struggle with the *how*. To develop LOWCODER$_{NL}$, we curate an aligned training dataset of natural language queries and scikit-learn operator invocations from Kaggle notebooks. We benchmark a variety of large language models on this corpus and conclude by proposing a fine-tuned CodeT5 model that can generate partial operator invocations depending on available information. We next evaluate the performance of LOWCODER in a user study involving 20 developers with different levels of AI expertise, contrasting LOWCODER$_{NL}$ with a traditional, keyword-based lookup. We found that LOWCODER is especially useful for (i) Discoverability: 80% of users discovered new operators in some tasks, and (ii) Iterative Composition: 82.5% of tasks were successfully completed and many such pipelines were further successfully improved. Qualitative analysis showed that novices especially struggled when they lacked clarity on *what* they wanted to accomplish. Even so, developers of all backgrounds successfully built and improved AI pipelines by playing around with code blocks, with minimal guidance. Overall, our work demonstrates the promise of combining both an AI-powered natural language interface and a visual interface for helping developers create AI pipelines without writing code.**

I. INTRODUCTION

This paper tackles the problem of simplifying the development of AI pipelines by drastically reducing the programming effort required. Most AI development today involves using popular Python libraries such as scikit-learn (sklearn) [1]. Unfortunately, having to write textual code, even in a language as high-level as Python, poses a high barrier to entry for citizen developers [2]. In our domain, *citizen developers* are people without formal training or experience as AI software developers. Furthermore, these libraries are often large. Needing to remember hundreds of AI operators and their hyper-parameters slows down even professional developers.

Low-code programming [3] reduces the amount of textual code developers write by offering alternatives such as visual programming, programming by demonstration, or programming by natural language (PBNL). The term low-code, initially coined by market research companies [4], is being quickly embraced by software vendors to democratize software development and increase productivity. So far, most low-code offerings favor visual programming, which has been a popular choice for building AI pipelines in prior work [5]–[7]. However, while visual programming is well-suited for helping users navigate complex pipelines, it poorly supports *discoverability* of API components in large APIs due to too many choices and too little screen space.

This paper introduces LOWCODER to overcome this problem by combining visual programming with PBNL. The main idea is for the respective strengths of these two low-code techniques to 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.

The visual programming component of LOWCODER, LOWCODER$_{VP}$, lets users snap together visual blocks for AI operators into well-structured AI pipelines. It uses Blockly [8] to provide a Scratch-like [9] look-and-feel. The PBNL component, LOWCODER$_{NL}$, lets users enter free-flow natural language queries and predicts relevant operators, optionally configured with hyper-parameters. It uses a fine-tuned variant of the CodeT5 model [10] that we developed through experiments with a wide variety of neural models for program generation, ranging from training models from scratch to few-shot prompting (very) large language models [11]. We further noticed that it is common in this domain for queries to 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.

Taking inspiration from projectional editors [12], LOWCODER provides three views: visual, natural language, and a third view showing the effect of pipelines on data, all of which are backed by a common domain-specific language (DSL). For the DSL, we picked Lale [13], which is embedded [14] in Python as its host language. We evaluate LOWCODER holistically by conducting a user study with 20 participants. To understand the effect of the AI model in the context of the full tool, the study contrasts users using LOWCODER$_{NL}$ with those using a simple keyword-based search interface. We find that LOWCODER$_{NL}$ helped users discover previously-unknown operators more easily than other methods, such as web search. In addition, despite being trained on a different dataset, LOWCODER$_{NL}$ accurately answered real user queries. Overall, the combination of visual, natural language, and data views helped users to successfully compose and then further refine their pipelines during the study.

This paper makes three main contributions:

(i) Low-Code for AI: We introduce LOWCODER, a new low-code tool for visual programming of AI pipelines.

(ii) AI for Low-Code: We explore AI models for PBNL and integrate those models with the visual tool.
(iii) User Study: We evaluate our tool with 20 participants covering a spectrum from citizen to pro-developers.

II. Related Work

AI for Low-code for AI: In adopting a visual programming approach to low-code, we follow a long tradition [15]. We were particularly inspired by Scratch, a popular visual programming environment for children that uses lego-like connected blocks [9]. Our other inspiration came from projectional editors, where the visual programming interface is a projection, or *view*, over an internal domain-specific language (DSL) [12]. Our implementation uses Blockly, a meta-tool for creating block-based visual programming tools [8], and Lale, a DSL for AI pipelines [13].

AI for Low-code for AI: Most low-code interfaces for programming AI pipelines use visual programming. Examples include WEKA [7], Orange [6], and KNIME [5]. 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. Our tool uses connecting blocks instead of boxes-and-arrows to encourage clearer structure and reduce clutter. The main difference between these earlier efforts on low-code for AI and our paper is that we add a natural language interface to help users discover and configure operators.

AI for Low-code for AI: 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 [16]. Desai et al. treat PBNL as a program synthesis problem targeting a DSL designed for the purpose [17]. The Overnight paper addresses the problem of missing training data for PBNL interfaces by crowd-sourcing [18]. And SwaggerBot lets users extend and customize a chatbot from within the chatbot itself [19]. Unlike these works, our paper uses large language models for PBNL, uses PBNL for creating AI pipelines, and connects to 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 [20] and Pumice combines programming by demonstration with PBNL [21]. 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 we target a different domain, namely AI.

PBNL for pro-code: Our work uses large language models for low-code. It is inspired by recent advances towards using large language models for pro-code, such as Codex [22], which uses the GPT-3 model for natural language and fine-tunes it on Python code. While GPT-3 is not openly available, there are several other large language models that are, which Xu et al. recently compared empirically [22]. In this paper, we experiment with a transformer model trained from scratch, as well as with fine-tuning CodeT5 [10] and few-shot prompting for CodeGen [11]. Unlike these papers, ours specifically targets low-code, not pro-code in a textual language such as Python.

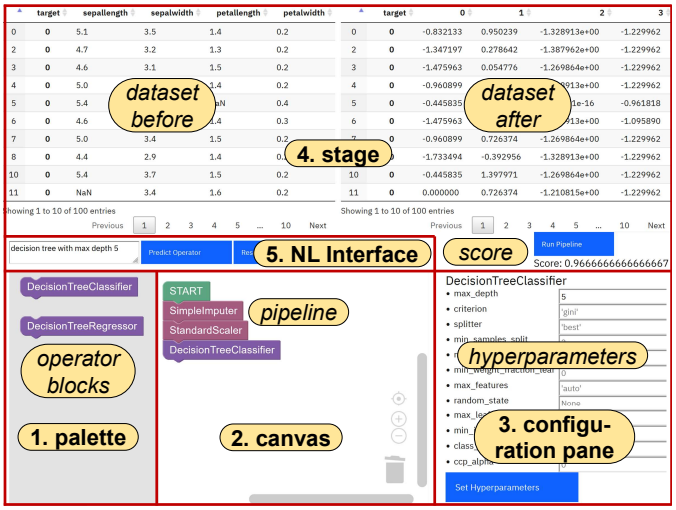

Fig. 1: LowCoder interface.

III. Low-Code for AI

In this work, we explore the intersection of low-code and AI. We accomplish this by implementing and studying LowCoder, a prototype low-code tool for building AI 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 1 highlights the main features and inputs of LowCoder. The tool is available for review at https://bit.ly/aiforlowcodeforai (please do not distribute).

To support multiple low-code modalities, we follow the lead of projectional code editors [12] 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. 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 desiderata is that the DSL is compatible with sklearn [1], one of the most popular libraries 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 [13] satisfies these requirements, and in addition, describes hyper-parameters in JSON schema format [23], which our tool uses extensively as discussed below. LowCoder uses a client-server architecture with a Python Flask back-end server and front-end based on Blockly [8] which is a meta-tool for creating block-based visual programming tools.

LowCoder supports 143 sklearn operators. Sklearn distinguishes between *operators*, which are steps of machine learning pipelines and usually have learned coefficients, and *functions*, which are used separately and cannot be a step in a pipeline. For example, `MinMaxScaler` is an operator whose

learned coefficients are the minimum and maximum values for each feature, while `train_test_split` is a function. Since LOWCODER focuses on pipelines, it only has explicit blocks for operators but not functions. The front-end converts the block-based representation to Lale which is then sent to the back-end. The back-end validates 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.

A. Visual 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 [9] is to encourage a more "tinkerable" 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 the data (with a *transform()* method) and purple for other operators which are primarily estimators such as classifiers and regressors (with a *predict()* method).

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 1 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. Blocks can also be dragged in-between other blocks to insert steps in the middle of a 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.

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. Our tool provides a *stage (4)* with *Before* and *After* tables to give feedback on how the current pipeline affects the given dataset. When a tabular dataset is loaded, it is shown in the *Before* table, which displays the target column on the left and feature columns on the right and supports pagination, search, and sortable columns. When a pipeline that transforms input data is executed, the results of the transformations are shown in the *After* table. For instance, in the example shown in Figure 1, 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 [24] 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. This gives the user immediate feedback on the effect that pipeline changes have on the dataset.

B. Natural Language Interface and Simple Keyword Search

A potential weakness of visual low code tools is that users have trouble discovering the right components to use [2]. 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 (5)* (NL = Natural Language) 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 1. 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 IV 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 [16]. In comparison, visual inputs and representations are unambiguous [25], 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 V). 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). This version is also available for review at https://bit.ly/aiforlowcodeforaikeyword (please do not distribute).

IV. AI for Low-Code

This section discusses the AI that went into LowCoder$_{NL}$.

A. 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 the 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 machine intelligence, particularly, language models that can be trained to translate a natural language query into the corresponding line of code [26]. 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 probably 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 V-A5 show that this is indeed effective.

B. 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. Our data extraction scripts handle a variety of corner cases, such as sklearn calls that are aliased in import statements and invocations spanning across multiple lines. 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. 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. Some interesting observations pertaining to the hyper-parameters in the operator invocations are as follows:

- 18% of the data does not specify any hyper-parameters and make use of the default values only.
- 39% of the data has 1–3 named hyper-parameters.
- 3% of the data has hyper-parameter values that are explicitly mentioned in the NL query.

See Section 2 in the supplementary material for a detailed distribution of hyper-parameters and their values.

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

C. Tasks

Given the NL query, our model aims to generate a line of sklearn code corresponding to an operation invocation that can be used 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 I. Additional examples can be found in Section 1 of the supplementary material.

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.

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 V-A4) show that the model often predicts arbitrary hyper-parameter values, resulting in lines of code that can rarely be used directly by developers.

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) 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 upfront. 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.

To evaluate the feasibility of predicting code using the different task formulations, we train a simple sequence-to-sequence model (detailed in Section IV-D1) and compare the results for the various training tasks in Section V-A4. We find HOI to be the most accurate/reliable formulation for the task

TABLE I: Task formulations highlighting the code components: operator name, hyper-parameter name, hyper-parameter value, mask. 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')`

of generating low-code from NL queries. We therefore proceed to use this task formulation for training the models.

D. Modeling

All tasks from Section IV-C are sequence-to-sequence tasks. We compare and contrast three different deep learning paradigms for this type of task, illustrated in Figure 2. First, we train a standard sequence to sequence transformer *from scratch*. Our relatively small dataset of ca. 70K training samples limits the size of a model that can be trained in this manner. A common alternative is to pretrain a larger model on a large dataset based on a generic task, such as masked token infilling, and/or a combination of objectives. Such models can then be fine-tuned (calibrated) on a smaller dataset to high accuracy. For the second paradigm, we fine-tune one such "medium" sized model, CodeT5 [10]. Finally, Large Language Models (LLMs), with billions of parameters, are often able to generalize to new tasks from just a few examples. These models are typically expensive to train from scratch or fine-tune; we adopt such a model, named CodeGen [11], and query it by means of few-shot prompting. We elaborate on these models below. Note that we use top-k sampling for our top-5 results. (A comparison of results with other decoding strategies can be found in Section 3 and 4 in the supplementary material).

1) Transformer (from scratch): We train a sequence-to-sequence Transformer model [27] with randomly initialized parameters on the training data. 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.

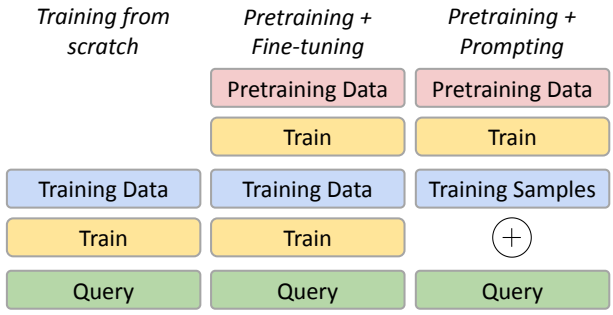

Fig. 2: Overview of the "trifecta" of training approaches used in contemporary deep learning: smaller models are directly trained from scratch on downstream task data; medium sized models (100M-1B parameters) are pretrained with a generic training signal and then fine-tuned on task data; large models (>1B parameters) are only pretrained on very large datasets and are prompted with examples from the training data as demonstration followed by the query.

2) Fine-tuning CodeT5: CodeT5 is a pretrained encoder-decoder transformer model [10] that has shown strong results when fine-tuned (calibrated) on various code understanding and generation tasks [28]. It uses a novel identifier-aware objective during pretraining and leverages natural language/code pairs to learn cross-modal alignment. CodeT5 was pretrained on a corpus of six programming languages from the CodeSearchNet dataset [29] and fine-tuned on several tasks from the CodeXGLUE benchmark [28] 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), codet5-base (220M) and codet5-large (770M).

3) Few-Shot Learning With CodeGen: Lastly, we explore large language models that are known to perform well in a task-agnostic few-shot setting [30]. More specifically, we look at CodeGen, a family of large language models that are based on standard transformer-based autoregressive language modeling [11]. 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 [22] on a Python benchmark [11].

Models at this scale are expensive to fine-tune and are instead commonly used for inference by means of "few-shot prompting" [31]. Large language models are remarkably capable of providing high-quality completions given an expanded prompt containing examples demonstrating the task [30]. We prompt our model with 5 such NL-code examples. Figure 3 illustrates an example prompt with 3 such pairs. The model learns from the examples in the prompt and completes the sequence task which results in generating the HOI code.

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:

Fig. 3: Few (3) shot prompting

V. Evaluation

This section describes the evaluations for the AI modeling that enables LowCoder$_{NL}$ along with the user studies that holistically evaluate LowCoder.

A. Modeling

1) Experimental Setup: All of our models are implemented using Pytorch transformers and the HuggingFace interface. We use the latest checkpoints of the CodeT5 [10] and CodeGen [11] 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. Since few predictions are longer than this threshold, this incurs no significant decrease in accuracy, but speeds up inference by 34% (details in Section V-A5). 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.

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 IV-B 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 [32] of 100%. (See Section 5 in supplementary material for details on annotation guidelines).

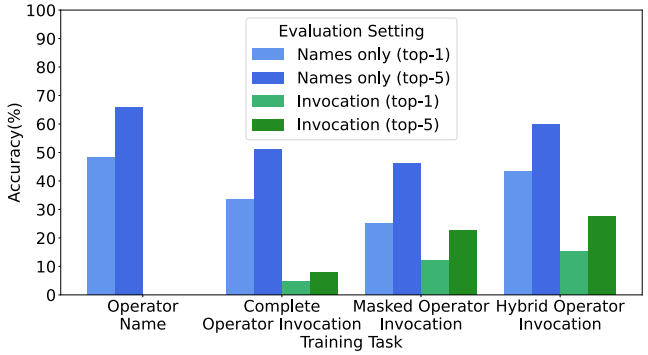

Fig. 4: Accuracy of transformers trained from scratch on various task formulations. 'Invocation' test results refer to the specific invocation formulation corresponding to the training task, while 'Names only' only considers whether the generated code starts with the correct operator name. Only the Hybrid Operator Invocation setting yields useful quality at both tasks.

3) Test Metrics: We use both greedy (top-1) and top-K (top-5) decoding 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.

4) Task Comparison: We first train a series of randomly initialized 6-layer Transformer models from scratch on each task formulation from Section IV-C. 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 4. 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.

5) Model Comparison: We next evaluate the performance of the various model types described in Section IV-D on the task of Hybrid Operation Invocation (HOI) generation. We benchmark the performance of the trifecta of modeling strategies (see Section IV-D) across different model sizes and compare the performance for both operator name and operator invocation generation using top-5 accuracy in Table II. (See Section 4 in the supplementary material for additional results and ablation studies.)

TABLE II: Top-5 Accuracy scores based on top-K sampling of various models. (*uses top-3 due to memory constraints)

Paradigm	***From Scratch***	***Fine-tuning***			***Few-shot Prompting***			
Model	**Transformer**	**CodeT5**			**CodeGen**			
Size	*6 layers*	*Small*	*Base*	*Large*	*350M*	*2.7B*	*6.1B*	*16B**
Operator Name	60.13%	71.78%	73.20%	**73.57%**	46.02%	48.63%	49.79%	43.27%
Hybrid Operator Invocation	29.16%	38.86%	40.04%	**41.27%**	9.66%	9.48%	11.32%	9.73%

We find that the fine-tuned CodeT5-large is the best performing model with an accuracy 73.57% and 41.27% on the test data for the operation name and operation invocation generation respectively. The fine-tuned CodeT5-base model has comparable performance, but its inference time is approximately 2–3 seconds lower than the fine-tuned CodeT5-large model, making it more desirable for integration with the tool.

We also perform additional experiments in an effort to reduce the inference time of the fine-tuned CodeT5-base 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.

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 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$_{\text{NL}}$ 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$_{\text{NL}}$ was accurate in over 90% (137/152) of completions (refer to Section 6 in supplementary material for additional results).

B. User Study

To evaluate LowCoder, we conducted a user study with 20 participants using two versions of our tool to create AI pipelines across four sample datasets. We collect and analyze data to investigate the following research questions:

RQ1: How do LowCoder$_{\text{NL}}$ 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 using low-code for AI?

1) Study Methodology: We recruited participants within the same large technology company via internal messaging channels and intentionally solicited participants of all ranges of machine learning experience. Potential participants filled out a quick 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, cybersecurity, business informatics, quantum computing, and software services. Five out of 20 (20%) participants are female. Eight out of 20 (40%) 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 [33] 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$_{\text{NL}}$. The keyword condition used a simple substring filter for operator names. Each participant performed four tasks total across the two conditions. 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 public tabular datasets from the UCI Machine Learning Repository [34]. Two of the tasks (A and D) require a specific data preprocessing step in order to successfully create a pipeline while two (B and D) technically do not require preprocessing to proceed. 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 [34]. 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 natural language 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. Artifacts used in the user study including task datasets and tutorial slides are included in the supplementary material.

2) 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).

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 [33] 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 5. 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.

TABLE III: Incidence of tasks where participants find previously-unknown operators.

Method of Discovery	***Total Tasks*** **(80)**	***Novice*** **(32)**	***Non-Novice*** **(48)**
LOWCODER$_{NL}$	30 (37.5%)	8 (25.0%)	22 (45.8%)
Web Search	18 (22.5%)	5 (15.6%)	13 (27.1%)
Scrolling Through Palette	16 (20.0%)	9 (28.1%)	7 (14.6%)
All Methods	51 (63.8%)	16 (50.0%)	35 (72.9%)

3) Study Results: **RQ1: How do LOWCODER$_{NL}$ and other features help participants discover previously-unknown operators?** Table III 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. Using the NL interface powered by LOWCODER$_{NL}$, participants discovered unknown operators in 37.5% of tasks as opposed to 22.5% using web search engines or 20% by scrolling through the operator palette. We note that the participants were not able to use the keyword search to discover unknown operators due to needing at least part of the exact name. The odds of an unknown operator being discovered is significantly greater using LOWCODER$_{NL}$ as opposed to both web search (p=0.032) and scrolling (p=0.014) using Barnard's exact test. When splitting on the experience of the participant, we find statistically greater chances of discovering operators using LOWCODER$_{NL}$ as opposed to web search or scrolling for non-novices (p=0.039, p=0.001) but not novices (p=0.206, p=1.000) per task. These results suggest that LOWCODER$_{NL}$ is particularly helpful in discovering previously-unknown operators for non-novices but novices face some challenges. We discuss these challenges in RQ3.

RQ2: Are participants able to compose and then iteratively refine AI pipelines in our tool? Table IV 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. 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 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 iterat-

TABLE IV: Incidence of tasks participants commplete and iterate on preprocessors, classifier, 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%)

ing 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 using low-code for AI? Figure 5 shows our 13 axial codes for answering RQ3. These codes broadly represent three overarching themes regarding working with low-code and machine learning: 1) *Discovery* of machine learning operators relevant for the task at hand, 2) *Iterative Composition* of the operators in the tool mostly through the visual metaphor of blocks, 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 Section 7 in the supplementary material 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 the 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. We note that challenges regarding general web search is also an axial code.

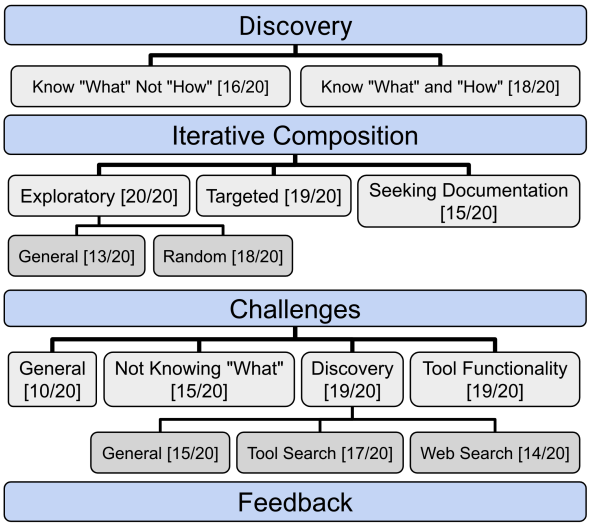

Fig. 5: Axial codes from qualitative analysis.

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. We note that 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$_{\text{NL}}$, 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$_{\text{NL}}$. 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$_{\text{NL}}$ and the language used by novices.

VI. DISCUSSION

Our results show that both the LOWCODER$_{\text{VP}}$ and LOWCODER$_{\text{NL}}$ components were helpful with aspects like discovering operators (RQ1) or iteratively composing pipelines (RQ2), especially for novice participants. This is useful for developers who have an idea of *what* they would like to do but do not fully know *how* to accomplish that. In fact, our qualitative analysis (RQ3) reveals that a number of our participants (including all novices who participated) struggled with knowing *what* to do. In contrast to other popular low-code domains such as traditional software [9], the domain of developing machine learning systems is particularly difficult in this regard due to its highly experimental nature where progress has a high degree of uncertainty [35]. This uncertainty then requires an abundance of judgment calls that rely heavily on prior machine learning experience [36] that novices lack. Some participants in our studies echo this, identifying that some ML knowledge is necessary to use our tool. A further improved low-code machine learning tool could thus be made more suitable towards citizen developers by guiding them to discover the *what* along with the *how* by helping developers acquire the necessary ML knowledge.

A potential extension, offered 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. A related tool suggestion raised by a number of our study participants is to implement data visualization and summarization tools 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 citizen developers and pro-developers alike.

Threats to Validity: The user study for LOWCODER has several limitations. The study focused on relatively small, public tabular datasets and scikit-learn 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.

VII. CONCLUSION

This paper describes LOWCODER, an AI-driven low-code tool combining a visual programming (VP) interface with programming by natural language (NL) that assists users with a variety of skill levels in creating AI pipelines. We found through user studies that the tool is particularly useful for helping users, even machine learning novices, discover previously-unknown operators and compose and iterate AI pipelines. Particularly for novices, we find challenges such as a lack of guidance for "what" actions should be performed for a given dataset. However, even novices are often able to use both the VP and NL modalities of LOWCODER to create AI pipelines and improve upon them. The combination of the two modalities allows for unique benefits such as exploratory and targeted styles of iteration.

Data Availability: Our replication package (https://doi.org/10.5281/zenodo.7042297) includes code used to implement LOWCODER and define LOWCODER$_{\text{NL}}$. It includes the training and evaluation datasets for LOWCODER$_{\text{NL}}$ as well as analysis scripts, the full set of (axial) codes, the task datasets, ML tutorial slides, other supplementary material, and anonymized quantitative and qualitative data from the user study.

References

[1] 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, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[2] A. J. Ko, B. A. Myers, and H. H. Aung, "Six learning barriers in end-user programming systems," in *Symposium on Visual Languages – Human Centric Computing (VLHCC)*, Dec. 2004. [Online]. Available: https://doi.org/10.1109/VLHCC.2004.47

[3] A. Sahay, A. Indamutsa, D. Di Ruscio, and A. Pierantonio, "Supporting the understanding and comparison of low-code development platforms," in *Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2020, pp. 171–178. [Online]. Available: https://doi.org/10.1109/SEAA51224.2020.00036

[4] A. C. Bock and U. Frank, "Low-code platform," *Business & Information Systems Engineering (BISE)*, vol. 63, pp. 733–740, 2021. [Online]. Available: https://doi.org/10.1007/s12599-021-00726-8

[5] M. R. Berthold, N. Cebron, F. Dill, T. R. Gabriel, T. Kötter, T. Meinl, P. Ohl, K. Thiel, and B. Wiswedel, "KNIME - the Konstanz information miner: Version 2.0 and beyond," *ACM SIGKDD Explorations Newsletter*, vol. 11, no. 1, pp. 26–31, Nov. 2009. [Online]. Available: https://doi.org/10.1145/1656274.1656280

[6] J. Demsar, B. Zupan, G. Leban, and T. Curk, "Orange: From experimental machine learning to interactive data mining," in *European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD)*, 2004, pp. 537–539. [Online]. Available: https://doi.org/10.1007/978-3-540-30116-5_58

[7] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA data mining software: An update," *SIGKDD Explorations Newsletter*, vol. 11, no. 1, pp. 10–18, Nov. 2009. [Online]. Available: http://doi.acm.org/10.1145/1656274.1656278

[8] E. Pasternak, R. Fenichel, and A. N. Marshall, "Tips for creating a block language with Blockly," in *Blocks and Beyond Workshop (B&B)*, 2017. [Online]. Available: https://doi.org/10.1109/BLOCKS.2017.8120404

[9] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y. Kafai, "Scratch: Programming for all," *Communications of the ACM (CACM)*, vol. 52, no. 11, pp. 60–67, Nov. 2009. [Online]. Available: https://doi.org/10.1145/1592761.1592779

[10] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2021, pp. 8696–8708. [Online]. Available: https://aclanthology.org/2021.emnlp-main.685/

[11] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "A conversational paradigm for program synthesis," 2022. [Online]. Available: https://arxiv.org/abs/2203.13474

[12] M. Voelter and S. Lisson, "Supporting diverse notations in MPS' projectional editor." in *Workshop on The Globalization of Modeling Languages (GEMOC)*, 2014, pp. 7–16. [Online]. Available: https://hal.inria.fr/hal-01074602/file/GEMOC2014-complete.pdf#page=13

[13] G. Baudart, M. Hirzel, K. Kate, P. Ram, A. Shinnar, and J. Tsay, "Pipeline combinators for gradual AutoML," in *Advances in Neural Information Processing Systems (NeurIPS)*, Dec. 2021. [Online]. Available: https://proceedings.neurips.cc/paper/2021/file/a3b36cb25e2e0b93b5f334ffb4e4064e-Paper.pdf

[14] P. Hudak, "Modular domain specific languages and tools," in *International Conference on Software Reuse (ICSR)*, 1998, pp. 134–142. [Online]. Available: https://doi.org/10.1109/ICSR.1998.685738

[15] M. Boshernitsan and M. Downes, "Visual programming languages: A survey," University of California, Berkeley, Tech. Rep. UCB/CSD-04-1368, 2004. [Online]. Available: https://digitalassets.lib.berkeley.edu/techreports/ucb/text/CSD-04-1368.pdf

[16] I. Androutsopoulos, G. D. Ritchie, and P. Thanisch, "Natural language interfaces to databases – an introduction," *Natural Language Engineering*, vol. 1, no. 1, pp. 29–81, 1995. [Online]. Available: https://doi.org/10.1017/S135132490000005X

[17] A. Desai, S. Gulwani, V. Hingorani, N. Jain, A. Karkare, M. Marron, S. R, and S. Roy, "Program synthesis using natural language," in *International Conference on Software Engineering (ICSE)*, 2016, pp. 345–356. [Online]. Available: https://doi.org/10.1145/2884781.2884786

[18] Y. Wang, J. Berant, and P. Liang, "Building a semantic parser overnight," in *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2015, pp. 1332–1342. [Online]. Available: https://www.aclweb.org/anthology/P15-1129.pdf

[19] M. Vaziri, L. Mandel, A. Shinnar, J. Siméon, and M. Hirzel, "Generating chat bots from web api specifications," in *Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, 2017, pp. 44–57. [Online]. Available: http://doi.acm.org/10.1145/3133850.3133864

[20] S. E. Chasins, M. Mueller, and R. Bodik, "Rousillon: Scraping distributed hierarchical web data," in *Symposium on User Interface Software and Technology (UIST)*, 2018, pp. 963–975. [Online]. Available: https://doi.org/10.1145/3242587.3242661

[21] T. J.-J. Li, M. Radensky, J. Jia, K. Singarajah, T. M. Mitchell, and B. A. Myers, "PUMICE: A multi-modal agent that learns concepts and conditionals from natural language and demonstrations," in *Symposium on User Interface Software and Technology (UIST)*, 2019, pp. 577–589. [Online]. Available: https://doi.org/10.1145/3332165.3347899

[22] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. Ponde, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. Guss, A. Nichol, I. Babuschkin, S. Balaji, S. Jain, A. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[23] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč, "Foundations of JSON schema," in *International Conference on World Wide Web (WWW)*, 2016, pp. 263–273. [Online]. Available: https://doi.org/10.1145/2872427.2883029

[24] S. L. Tanimoto, "A perspective on the evolution of live programming," in *International Workshop on Live Programming (LIVE)*, 2013, pp. 31–34. [Online]. Available: https://doi.org/10.1109/LIVE.2013.6617346

[25] M. Hirzel, "Low-code programming models," May 2022. [Online]. Available: https://arxiv.org/abs/2205.02282

[26] Z. Feng, D.Guo, D. Tang, N. Duan, M. G. X. Feng, L. Shou, B. Qin, T. Liu, and D. Jiang, "Codebert: A pretrained model for programming and natural languages." in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings*, 2020.

[27] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Łukasz Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017.

[28] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *ArXiv*, vol. abs/2102.04664, 2021.

[29] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "CodeSearchNet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

[30] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901. [Online]. Available: https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

[31] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2018. [Online]. Available: https://d4mucfpksywv.cloudfront.net/better-language-models/language-models.pdf

[32] D. R. Garrison, M. Cleveland-Innes, M. Koole, and J. Kappelman, "Revisiting methodological issues in transcript analysis: Negotiated coding and reliability," *The Internet and Higher Education*, vol. 9, no. 1, pp. 1–8, 2006.

[33] J. W. Creswell, *Research design: Qualitative, quantitative, and mixed methods approaches*, 4th ed. SAGE publications, 2013.
[34] D. Dua and C. Graff, "UCI machine learning repository," 2017. [Online]. Available: http://archive.ics.uci.edu/ml
[35] Z. Wan, X. Xia, D. Lo, and G. C. Murphy, "How does Machine Learning Change Software Development Practices?" *IEEE Transactions on Software Engineering*, p. 1, 2019.
[36] C. Hill, R. Bellamy, T. Erickson, and M. Burnett, "Trials and tribulations of developers of intelligent systems: A field study," in *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, sep 2016, pp. 162–170.