SolMover: Feasibility of using LLMs for Translating Smart Contracts

***Abstract*—Large language models (LLMs) have showcased remarkable skills, rivaling or even exceeding human intelligence in certain areas. Their proficiency in translation is notable, as they may replicate the nuanced, preparatory steps of human translators for high-quality outcomes. Although there have been some notable work exploring using LLMs from code to code translation, there has not been one for smart contracts, especially when a target language is unseen to the LLM. In this work, we aim to introduce our novel framework SolMover consisting of two different LLMs working in tandem in a framework to understand coding concepts and then use that to translate code to an unseen language. We explore the human-like learning capability of LLMs in this paper with a detailed evaluation of the methodology to translate existing smart contracts from solidity to a low-resource one called move. Specifically, we enable one LLM to understand coding rules for the new language to generate a planning task, for the second LLM to follow, which does not have planning capability but does have coding. Experiments show that SolMOver brings significant improvement over gpt-3.5-turbo-1106 and outperforms both Pal2 and Mixtral-8x7B-Instruct. Our further analysis shows us that employing our bug mitigation technique even without the framework still improves code quality for all models.**

***Index Terms*—Smart Contracts, Machine Learning, Machine Translation, Code Transpilation, LLM, Large Language Model**

I. Introduction

Recent advancements in Large Language Models (LLMs) have been transformative, marking significant progress towards being human-like. These models are being touted to have human-level intelligence, especially in comprehending and generating language [1], [2], [3], [4]. Translation is one domain where LLMs excel, showcasing notable proficiency and effectiveness [5], [6], [7], [8], [9]. This advancement resonates with the initial goals and visions set forth by machine translation researchers in the 1960s [10], [11], raising the question of whether LLMs can mirror the translation methodologies employed by humans. This also begs the question, can we take inspiration from how these translations are working, to explore code-to-code translation using LLMs?

In recent times, the proliferation of decentralized ledger technologies, coupled with the smart contracts built upon them, has been particularly notable within the finance industry. For example, it has been reported that in the year 2021, Uniswap's smart contracts facilitated transactions averaging a daily volume of approximately $7.17 billion [12]. Given the surge in smart contract utilization and their pivotal role in diverse applications such as Confidential Computing [13], [14], Decentralized Serverless Architectures [15] it becomes imperative to inquire into the efficacy of Large Language Models (LLMs) in authoring smart contracts derived from user directives, as well as the security robustness of such automatically generated contracts. So it becomes a natural question if we can take a smart contract in one language and use LLMs to translate it.

In this study, we delve into the capabilities and constraints of LLMs within the realm of smart contract code translation. We investigate whether LLMs can mimic human translators in their approach to translating smart contracts—a specialized yet increasingly essential subset of software code.

The primary focus of this work is to explore whether we can use LLMs for smart contract transpilation with specific security guarantees. Specifically, we aim to investigate whether LLMs can effectively understand a smart contract written in solidity, and generate equivalent code in another smart contract language, namely Move in our case. We also look at how we can encode the smart contract concepts into the model for a more universal code transpilation framework that can translate one coding language to another in a universal way.

To address the above, we propose our system SolMover, which is a wordplay on Solidty-move. SolMOver involves multi-step knowledge distillation to understand the solidity code and generate equivalent move code. We convert the translation problem to a multi-step code synthesis problem with a granular task-based approach. SolMover first parses a solidity file for functions and content, generating a high-level task based on the file, this task is then sent to the first part of our fine-tuned LLM to generate sub-tasks. The fine-tuned LLM is fine-tuned on textbooks of move language and specifications along with solidity. This generates sub-tasks based on the previous input. This then we pass on to a smaller codegen model trained specifically on a very small subset of move code. Which generates the translated equivalent mode code for us. We use move-prover for verification of the generated code, as well as multi-prompting techniques to pass on the error message from move-cli to further improve the quality of the generated code. We validate our approach based on our evaluation set, as well as analyzing if the translated code has matched with the input of the specifications.

This work performs a multifaceted exploration of smart-contract translation of a low-resource language (Move) through the use of a combination of concept mining along with error-code-guided prompt engineering for regeneration. We also perform a preliminary evaluation with state-of-the-art models for the same code translation work. Our study aims to answer the following research questions:

RQ1: Can the LLMs learn coding concepts? We try to encode the knowledge of the Move language in our first

LLM. Are the Large Language Models capable of learning programmatic rules from textbooks? Can they associate the semantic rules described in a text with actual coding?

RQ2: Can the concepts be used to generate a granular subtask from a generalized prompt? We explore the possibility of generating task-specific granular sub-tasks based on the inferred prompt from the input contract using knowledge from **RQ1**. Are these sub-tasks useful enough for generating code?

RQ3: Can we generate code in a low resource language the LLMs have not been trained on? We ask the question of our fine-tuned model, which has not been trained on a large plethora of move code, whether can we still generate compilable move code.

RQ4: Mitigating bugs using compiler feedback. Can we use compiler feedback to mitigate the bugs and make the code better? To what extent do prompting techniques help the model mitigate these bugs?

Move [16] was designed to work on the Libra Blockchain [17], [18]. Move's unique feature is crafting unique asset types (called resources) with built-in self-defense [19]. Duplication and silent disposal are out of the question; these assets can only change hands between storage spaces within your program. Move's rigorous type system acts as a bouncer, checking IDs and verifying every transfer. Even with these robust shields, resources remain flexible citizens in your code: they can nestle in data structures, act as arguments, and join the party wherever needed. In contrast to other programming languages Move is not a completely separate language, but rather created based on Rust [20] with different safety features built-in. This makes it even more challenging for translating smart contracts to move.

In summary, the contributions of this paper are listed below:

- We propose SolMover which takes a Solidity code and attempts to translate it to Move
- We demonstrate that even if an LLM has not been explicitly pre-trained on a code corpus, and is unable to generate compilable code, it still can learn to generate compilable code with fine-tuning and the help of our framework
- We demonstrate that LLMs can encode concepts, and generate granular sub-tasks from larger task
- We find that using compiler errors to guide the LLMs to produce better code does produce results, but to a limited degree
- We demonstrate that with a combination of knowledge distillation and fine-tuning using a very small amount of code, code generation for low-resource language is possible

To our knowledge, we are the first to propose and evaluate a system to (1) use concepts and code to generate code and (2) show that it is indeed possible to generate compilable code for low-resource language using off-the-shelf LLMs, just by fine-tuning and daisy chaining.

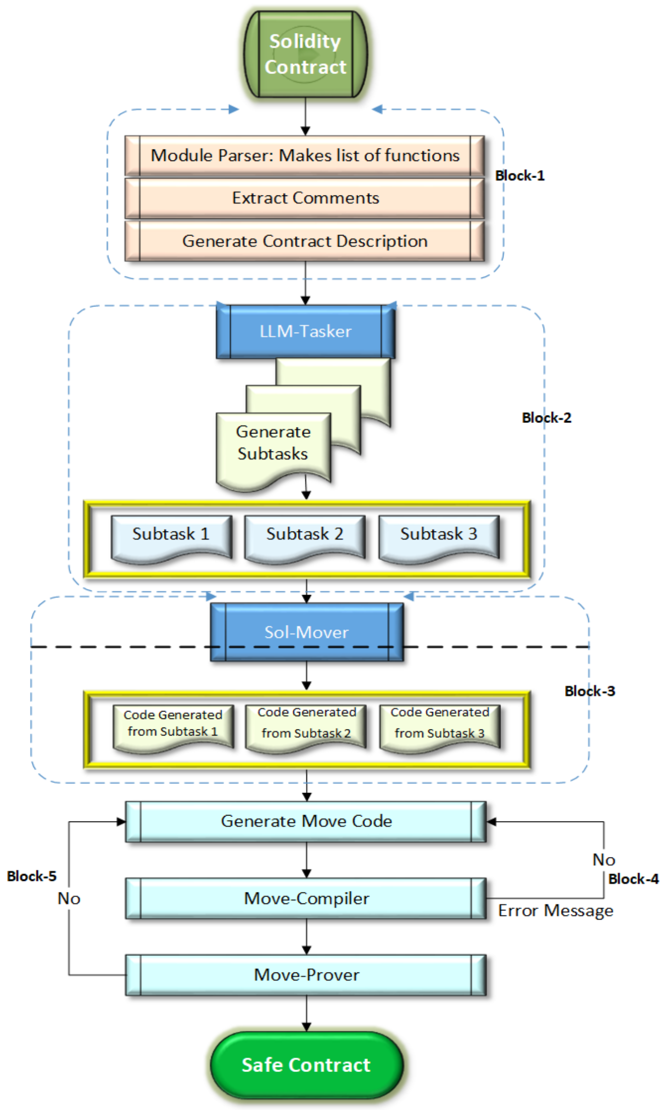

Fig. 1. Framework for **SolMover**. Five stages: (1) Task Creation: The parser parses the solidity file to generate the initial prompt for the task based on comments and keywords used in the smart contract. (2) Concept Mining: We use retrieval retrieval-augmented approach to mine concepts from move programming guides and books to generate subtasks based on the initial prompt. (3) Code Generation: We use the subtasks generated using concept distillation to generate code for each subtask. These are then compiled into a single move file, which will be our candidate-translated code. (4) The translated code is compiled, if it cannot be compiled, the error is sent back to Sol-Mover with a modified prompt incorporating the error for regeneration. (5) The compiled code is passed through Move Prover to see if it can be proved, if it cannot be it again is sent to Sol-Mover with the error message incorporated in the prompt for regeneration.

II. SolMover

In this section, we introduce the SolMover framework.

As depicted in figure 1, SolMover consists of multiple modular blocks. *In block 1* the Solidity smart contract given for translation is analyzed. We use a technique similar to Karanjai et al [21] to extract the function and comments from the solidity file. The module parser and comment extractor work in tandem to extract comments from the solidity code and use a prompt template similar to [21] to generate the initial Prompt for the task. *Block 2* consists of the LLM tasker, which uses Move Books, whitepapers, and tutorials (with code examples) as its source for Retrieval Augmented Search to generate subtasks based on the prompt generated in *Block 1*. Each of these subtasks is then sent to our second LLM (Sol-Mover) fine-tuned on very limited Move code[22]. These subtasks are then collated to make the final Move Code to be part of *Block 4*. In *Block 4* we use the move-cli to try to compile the translated code. This step logs the successful compilation. And if a code does not compile it logs the error message and sends it back to Sol-Mover again to compile. Sol-mover tries to do it five times and logs the translation as a failure after the fifth try. *Block 5* tries to take the compilable code and verify correctness by running it through Move Prover [23]. Move Prover tries to verify the contract formally proving that the smart contract is correct within its specification. In *Block 5* we log the incorrectness specification and again pass it to Sol-Mover to rectify the error by re-prompting with the error. This is done again five times and if the correctness holds, the program is marked as a safe contract. Else marked as compilable but not yet formally proved.

A. Code Understanding

Since we have converted our code translation work to a code generation task, we first need to understand what the original input code is doing. To do that we developed a parser that takes a solidity file and extracts comments, functions, and generates a seed candidate prompt from the solidity input file that the

Comments are essential for understanding what a solidity smart contract is supposed to do. Explanatory comments embedded within the code act as beacons, revealing its core logic and its central purpose. These are essential to understanding a coder's original intentions and ensuring the code's accurate execution. By parsing the solidity code using techniques similar to [24].

Topic refers to what the solidity code does. Mostly these are derived from the initial comment and the used functions. As we can see in Code Snippet in Listing 1.

```
pragma solidity ^0.8.0;

// The sample hotel and vending Solidity smart contract below allows one to rent a hotel room.
// It allows someone to make a payment for a room if the room is vacant. After payment is made to the contract the funds are sent to the owner.
// This smart contract can be expanded to unlock the door or dispense a key code after payment is made.
//
// Think of this contract like a vending machine. You input funds, validations pass, and you get something in return.
// It is the same concept as a gumball machine.

contract HotelRoom {
    //create an emun with 2 status so we can keep track of our hotel room
    enum Statuses {
        Vacant,
        Occupied
    }
    Statuses currentStatus;
```

Listing 1. Input Solidity Code Example

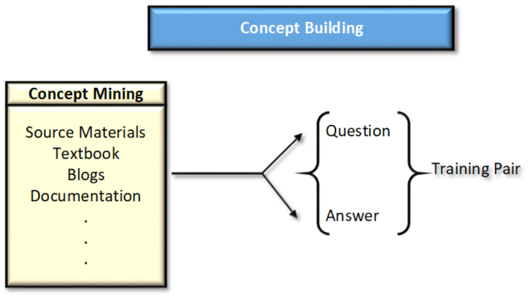

Fig. 2.

B. Concept Mining

We take cues from how human translation works [25], the concept mining requires the LLM to first produce output that that aligns with those concepts and knowledge beneficial for the transpilation work.

The concept mining involves fine-tuning an LLM on textbook data for move. we follow Gunasekar et al [26] to prepare the dataset and to fine-tune the model and [27], [28] for finding relevant text. We prepare the dataset as shown in Figure 2.

Where we use Move Books [29], [30], blogs on move [31], tutorials with code snippets [32] as concepts and use code samples from this repository [33]. We intentionally do not use any code corpus for fine-tuning the model.

For preparing the dataset we use scripts from [34] modified to work for our files.

1) Architecture: We use Retrieval Augmented search in our architecture as described by [27] for finding out our concepts. Since our task is different than their task, our implementation is different.

We assume that we have a textbook database containing a large selection of texts comprising different texts and code snippets as described in Section II-B. We create D number of text snippets based on splitting these documents. Since all of the available documents are available in HTML format, they were split using the presence of **H** or **Heading** available in HTML code (or in the absence of bigger text) and accompanying smaller text chunks, in hopes to split concept heading and related description. We use Dense Passage Retriever (DPR) [35], to split each of the text segments into segments of equal length as a primary unit for search and retrieve in hopes of getting better retrieval as suggested by [35]. This also has the added benefit of making our framework scalable, supporting very large file sizes. Thus we get M text fragments as the retrieval database $C = \{c_1, c_2, ..., c_M\}$.

Let $X = \{x_1, x_2, ..., x_k\}$ be the query keyword, a searcher $\mathbf{R} : (X, C) \rightarrow C$ searches and finds the most similar text fragment c_s in C. The generator $\mathbf{G}$ predicts the following text

3

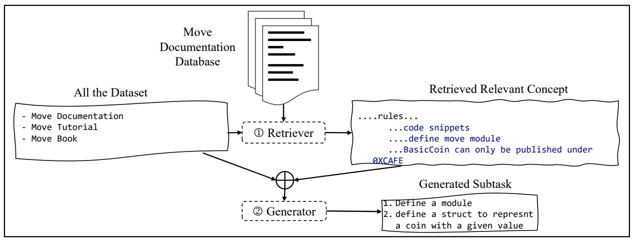

Fig. 3.

token(s) $Y = \{x_{k+1}, ..., x_{k+n}\}$, where $n = 1$ in the token-level text prediction task, based on context and retrieved text.

Formally, $P(Y) = \prod_{i=1}^{n} P(x_{k+i} | c_s, x_{1:k+i-1})$.

For searching BM25 [36] is used similarly to ElasticSearch. BM25 is a term-based searching method, which uses bag-of-words. It computes the lexical similarity between the query and the document fragments. The DPR models [35] are made of two bidirectional transformers E_C and E_Q. E_C encodes each split text in the search database C and indexes them. The representation of `[CLS]` token is taken as output, where we compute the similarity by $sim(q, c) = E_C(c)^T E_Q(q)$. For the training we follow [35], [28], by adopting batch negatives we calculate the loss as shown by [37]. However, we do not ignore the hard negatives like [28] and instead follow [35].

$$\begin{aligned} &L(q, c^+, c_1^-, c_2^-, ..., c_m^-) \\ &= -\,log \frac{e^{sim(q,c^+)}}{e^{sim(q,c^+)} + \sum_{i=1}^{m} e^{sim(q,c_i^-)}} \end{aligned} \quad (1)$$

C. Concept Selection

Concept Selection is the final step for our sub-task generation. Even though keywords, topics, and relevant text demonstration are beneficial for subtask generation for our translation task, not all results will be useful. For example, the LLM may generate subtasks based on trivial or noisy content which will in turn guide the Sol-Mover LLM to generate useless unused code snippets in the best case, and wrong code in the worst case. Hence we limit the subtask generation based on concepts gleaned only from the text parts where suitable code snippets have also been found. Which is also passed to the Sol-Mover as part of the subtask prompts. This helps the LLM produce subtasks as seen in Figure 3

D. Sol-Mover Code Generator

Sol-Mover is our Move Code generator which takes the Sub Tasks generated by the previous LLM as seen in Figure 3 and generates code in Move. Sol-Mover is an instruction-trained model based on Alpaca[38]. Alpaca is an open-source instruction following LLM fine-tuned on LLAaMA [39] model with 52k Self Instruct data with user-shared conversations collected from ShareGPT [40]. We use langchain to build a prompting framework that takes each of the sub-tasks along with any code snippet example found and generates code using the fine-tuned model. The generated response then is compiled into a single file based on the task sequence to rebuild the translated code for the original task. This is then sent to move-cli for compilation. All the compilation errors are logged and added as memory to the langchain agent and sent back to the Sol-Mover for program rectification. We run this process iterative-ly five times. If the candidate's response passes the compilation stage we log it as successful code translation. However, we still do not guarantee its correctness and send it to move-prover, which is an automatic formal verification system for Move. The errors suggested by the prover are again sent to Sol-Mover for five iterations for correction. We discuss in detail the training part of both Sol-Mover and our sub-task creation LLM in Section III.

E. Dataset for Solidity as Input Candidate

In our study, we faced the limitation of not being able to utilize pre-existing datasets like HumanEval [**?**], which are predominantly focused on Python programming. Consequently, we initiated the creation of a novel dataset for our assessment. Our approach to verifying code accuracy involved a meticulous selection of source files from GitHub, all under open-source licenses with permissive terms.

Our strategy included the development of a scraper via GitHub APIs, aimed at harvesting Solidity code from a variety of projects. The corpus collection was guided by multiple criteria:

- Eligible projects must have at least 50 GitHub stars, indicating sufficient interest, popularity, and likelihood of code written for a human audience.
- The presence of "Solidity" as a programming tag was mandatory for further refinement.
- An ample presence of comments in the projects was essential. This facilitates the development of scalable prompts and the establishment of an experimental pipeline.

Following the collection, these projects underwent further processing to remove non-Solidity files.

a) Compilation of the raw code corpus.: Utilizing GitHub, an invaluable resource for a diverse array of publicly available source code, we replicated top-ranking repositories marked with the Solidity language tag and having a minimum of 50 stars. From these, we extracted all Solidity-authored files, forming the foundational dataset for training.

III. Technical Specification

We give details of our training regimen and model selection for both of our models for Concept understanding and sub-task generation. As well as fine-tuned Alpaca for code generation.

A. Retrieval Augmented LLM for Concept Understanding

To address one of the prime premises of code-to-code translation for very low-resource languages like Move, our framework solves a key bottleneck, premise selection [41]. Most of the existing LLM-based knowledge distillation [27] and code completion [36] frameworks generate the next tactic (step) based on the current step as input. However generating coherent plans and subtasks critically depends on premises, such as rules and code examples from an associated description.

Incorporating all possible contexts is too large to fit into LLM's input prompt given the limited context window, and circumventing by using LangChain's agent-based memory does not yield much better results. Existing methods must memorize the association between the task state and the next relevant task based on the original task. If the context has been used in the training data to solve a similar problem, then this works, however for low-resource languages like Move that is not the case. On top of that, since Move is based on Rust, the existing pre-trained Rust code pollutes the premise by providing the wrong candidates. It also does not generalize for truly novel scenarios, that require code generation based on tasks unseen in training data.

Our potential solution complements the memorization of LLM with explicit context selection for better concept finding. The context is extracted from books where they are defined and used. It enables us to find relevant context by augmenting LLMs with retrieval.

We also need to limit the context retrieval to a small number of contexts for it to be effective and useful. As we had discussed in Section II-B1 our retriever builds upon DPR but has the following two changes relevant to our purpose. First, not all context is available while searching. We first extract comments and functions that limit the accessible context based on them as keyword input in the initial prompt. That reduces the retriever's task on our data by almost 75%, simplifying the task. Secondly, DPR benefits from having hard negatives in training, i.e. irrelevant contexts that are hard to distinguish. We instead use in-file hard negatives which samples negative contexts defined in the prompt itself.

1) Experimental Setup: The training is in two stages. In the first stage, we train the retriever and use it to retrieve 100 context states for all contexts for each task. Second, we train the sub-task generator, taking as input the retrieved context. We evaluate the task generator by combining it with the best-first search to generate valid sub-tasks. Training takes 7 days on a single NVIDIA A100 GPU with 80GB memory.

B. Sol-Mover Code Generation Training

For the code generator, we use Alpaca and fine-tune it using code examples as part of the instruction training dataset created as described in Section II-D. We used a very very limited set of code samples as a corpus for fine tuning. These include Fungible Tokens [42], [43], [44], [45], [46], NFT [47], [48], DeFi [49], [50], [51] as an example for creation of the dataset. The function of this fine-tuning is to create an association in the Alpaca model to have code association for the Move code.

IV. Experiments

A. Experimental Setup

a) Models.: We experiment with four LLMs, encompassing both open-source and closed-source state-of-the-art models.

- gpt-3.5-turbo-1106: Employing the innovative RLHF methodology developed by [52], OpenAI's robust yet closed-source language model offers exceptional capabilities. We interact with this model through the official API provided by OpenAI.
- Alpaca [38]: Building upon the LLaMA model [**?**], this open-source, instruction-following language model was further honed using a comprehensive dataset of 52,000 examples generated through the Self-Instruct approach [**?**].
- Mixtral [53]: Mixtral-8x7B-Instruct, a free-to-use language model, excels at following your instructions and unleashing its creative potential. Built upon the powerful Mixtral-8x7B architecture [53], it's been meticulously trained on a custom dataset to understand and respond to diverse prompts and requests. This versatile model can generate text in various formats, translate languages, answer your questions informatively, and even write different kinds of creative content.
- Palm2 [54]: Google's PaLM 2, pushing the boundaries of language understanding, shines in tasks demanding advanced reasoning and expertise. Whether it's grappling with code and math, navigating multilingual nuances, or crafting accurate translations, PaLM 2 excels with its impressive size and cutting-edge techniques.

For Alpaca we use the 7B version and perform inference on a single NVIDIA V100 32GB GPU.

b) Comparative Methods.: For our code generation task, we consider only the single-candidate method. Since this is a framework that utilizes a daisy-chained response, only the primary candidate is being considered.

Within single candidate method, we consider:

- **Baseline**: Zero-shot smart contract translation with temperature set to 0 (default for remainder of experiments in this paper).
- **5-Shot** [55]: Prompting the model with five exemplars of superior quality, specifically selected from the sub-task domain and prepended to the test input, has been demonstrated to yield optimal overall performance, as per [55]. Further augmenting the number of examples, however, does not appear to produce any appreciable enhancement in outcomes.

c) Metrics and Benchmark: Since this work looks at code translation problems for a very low resource language Move. There are no existing code generation or testing benchmarks we can use. Furthermore, the nature of the Move smart contract makes it very difficult to have a training and testing separate evaluation dataset available for systematic evaluation. It also makes it hard to rely on GitHub to try to generate exact code examples for comparison, since this is not a code generation but rather a Code Translation Task.

For that purpose, we adopt the following evaluation criteria.

[HTML]DAE8FC LLMs	Compilable Code?	Performance
gpt-3.5-turbo-1106	Y	Mixed
Solmover (Two LLM Combined)	Y	Mixed
Mixtral-8x7B-Instruct	N	Mixes different languages and generate unusable code
LLama2	N	Mixes different languages and generate unusable code
Palm2	N	For Move only generates code snippets and plan

TABLE I
CODE TRANSLATION CAPABILITY OF FOUR LLMS

[HTML]CBCEFB LLM	Total Translation Task	Successful Compilation(SC)	SC After Error Feedback	SC after Move Prover Feedback
gpt-3.5-turbo-1106	734	204	229	229
Solmover (Two LLM Combined)	734	313	397	401

TABLE II
SUCCESSFUL TRANSLATIONS

- Code Compilability of the translated smart contract
- Code Correctness of the translated smart contract
- Bug mitigation of first candidate translation

Based on this we evaluate the effectiveness of our approach.

V. RESULTS

We have run the experiments on all four LLMs defined in Section IV. However, both Palm2 and Mixtral did not produce any tangible output for any of our candidate translations as we can see from Table I.

Henceforth we will only report the results found in our SolMover framework and gpt-3.5-turbo-1106.

A. *Successful Smart Contract Translation*

We ran a total of 734 solidity contracts from our collected dataset in Section II-E through both gpt-3.5-turbo-1106 and SolMover. Each model was allocated an identical quantity of tasks totaling 734. The performance is evaluated based on successful compilation (SC), improvements post-error feedback, and further enhancements after Move Prover feedback. The results are detailed in Table II.

As we can see from Figure 4.

- Initial SC rates were 204 for `gpt-3.5-turbo-1106` and 313 for `Solmover`.
- Post error feedback, SC rates improved to 229 for `gpt-3.5-turbo-1106` and significantly to 397 for `Solmover`.
- Subsequent to Move Prover feedback, `gpt-3.5-turbo-1106` remained static at 229 SCs, whereas `Solmover` exhibited a marginal increase to 401 SCs.

In this comparative analysis, the combined language model `Solmover` exhibited superior performance over `gpt-3.5-turbo-1106` in a translation task involving 734 items. `Solmover` outperformed in successful compilations initially, after error feedback, and after Move Prover feedback, with the latter showing no improvement at the final stage. This indicates that the integration of two language models in `Solmover` significantly enhances its ability to learn from feedback and improve its output, making it more adept at handling complex code translation tasks.

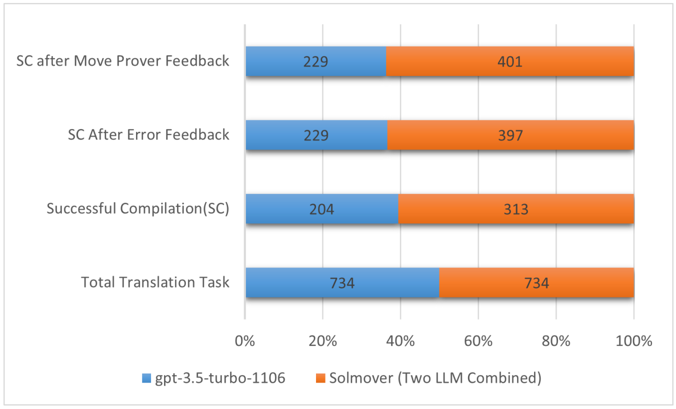

Fig. 4. Successful Translations

[HTML]CBCEFB LLM	Total Translation Task	Incomplete Translation (IC)	IC After Error Feedback
gpt-3.5-turbo-1106	734	187	110
Solmover (Two LLM Combined)	734	201	134

TABLE III
BUG MITIGATION

B. *Reducing Bugs using Iterative Error Feedback*

As we described in Section II-D we try to mitigate bugs introduced by the LLMs for code generation by iterative prompting with error code as a guide. This compiler error-based prompting iterative loop runs at a maximum of five times before marking the translation case as unsuccessful. We run this experiment on both of the LLMs to decouple the concept mining part from the bug mitigation part, and to see if just iterative prompting alone can help achieve better performance on state-of-the-art commercial LLMs too. We report the result in the Table III. The focus here is on incomplete translations (IC) before and after error feedback is applied.

Here we first log the incomplete translations without any kind of compiler feedback. Then we log how both compiler feedback and move provers feedback affect the LLM's bug mitigation capabilities.

As we can see from Figure 5 the results indicate that both models benefited from error feedback, with `gpt-3.5-turbo-1106` showing a more substantial reduction in Incomplete Completions. Producing more compilable code after the feedback loop. Despite `Solmover` starting with more Incomplete Completions(ICs), it did not reduce its ICs as effectively as `gpt-3.5-turbo-1106` post-feedback.

In the comparison of incomplete translations for two language models, `gpt-3.5-turbo-1106` and `Solmover (Two LLM Combined)`, both began with a similar number of tasks. `gpt-3.5-turbo-1106` initially reported fewer incomplete translations and also showed greater improvement after receiving error feedback. This suggests that while `Solmover` had a higher starting point of incomplete tasks, it was less responsive to feedback in reducing these errors compared to `gpt-3.5-turbo-1106`.

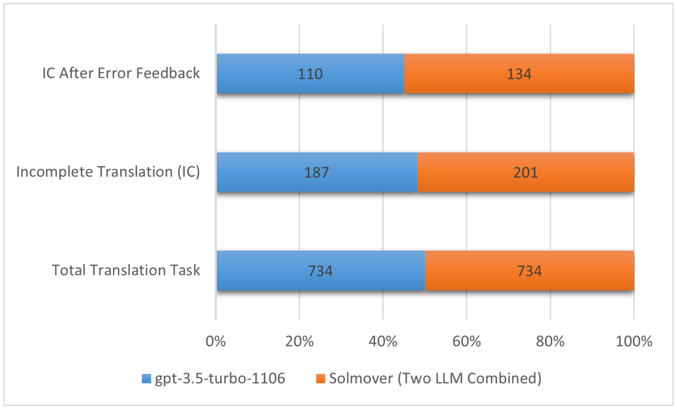

Fig. 5. Bug Mitigation

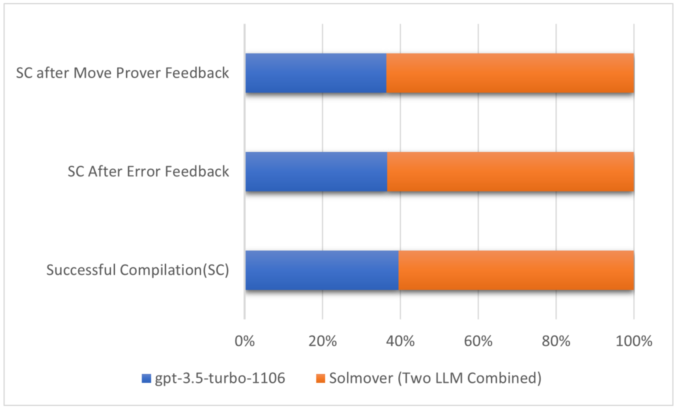

Fig. 6. Contract Correctness

[HTML]CBCEFB LLM	Successful Compilation(SC)	SC After Error Feedback	SC after Move Prover Feedback
gpt-3.5-turbo-1106	204	229	229
Solmover (Two LLM Combined)	313	397	401

TABLE IV
CONTRACT CORRECTNESS

- Pre-feedback, `gpt-3.5-turbo-1106` had 187 ICs, while `Solmover` had slightly more with 201 ICs.
- Post-error feedback, ICs reduced to 110 for `gpt-3.5-turbo-1106` and to 134 for `Solmover`.

C. Correctness

Our measure of code correctness here is based on the theorem prover for move, move prover. When move prover can prove a contract, we mark it as safe. Otherwise, if it produces an error, we mark it as correctness not proved. We also take the same iterative approach as the compiler error feedback loop to see if that has any effect on the different compilers. The results are detailed in Table 6.

As we can see from Figure 6 that `Solmover` not only had a higher success rate in initial compilations but also showed greater improvement upon receiving feedback. Moreover, it continued to improve even after the Move Prover feedback, in contrast to the `gpt-3.5-turbo-1106`, which plateaued.

- The `gpt-3.5-turbo-1106` had a baseline SC of 204, which increased to 229 after error feedback and remained unchanged after Move Prover feedback.
- The `Solmover` model, on the other hand, started with a higher SC of 313, which then improved to 397 after error feedback and slightly increased to 401 after Move Prover feedback.

The comparative assessment of `gpt-3.5-turbo-1106` and `Solmover (Two LLM Combined)` reveals that the latter outperforms the former in all stages of code compilation. Initially, `Solmover` starts with a higher number of successful compilations. It continues to improve significantly upon receiving error feedback and demonstrates a modest increase even after Move Prover feedback, suggesting a robust ability to learn and adapt. In contrast, `gpt-3.5-turbo-1106` exhibits improvement after error feedback but shows no further enhancement post Move Prover feedback, indicating a potential limit to its adaptability.

VI. Discussion & Analysis

In this section, we conduct analyses to understand the SolMover framework. We try to answer the research question (RQ) we started with and analyze the results reported.

A. Concept Distillation

One of the unique aspects of the framework is to try to encode concepts into an LLM using retrieval-augmented search methods. As we see from Figure 2 our search method is able to produce sub-tasks. If we look closely at the results in Fig. 6. Contract Correctness and Fig. 5. Bug Mitigation, we will notice that the iterative method has been applied to both SolMover and the gpt-3.5-turbo-1106 model. However, the increases in performance and decrease in noncompilable code synthesis are primarily observed in our framework. prompting us to look at our translation recipe of generating code using sub-tasks, and generation of the said sub-tasks using Concept Distillation.

RQ1 & RQ2 *Can the LLMs learn coding concepts? & Can the concepts be used to generate a granular subtask from a generalized prompt?*
We were able to empirically observe an increase in successful code translation for the model prompted by sub-tasks generated by the concept. Leading us to believe LLMs can encode associations resembling concepts. However we will not actually call it ”concept”, rather context association.

B. Smart Contract Translation

One of the primary goals of this paper was to explore whether can we translate smart contract code from one language (solidity) to another smart contract language (move). This is made harder by the fact that there are very few move codes available in GitHub and not suitable for training a large language model. We also looked at different ways we can improve the quality of the translated code by reducing bugs introduced by the LLM. Our results are depicted in Fig. 4. Successful Translations give us confirmation that we can successfully translate an existing smart contract code written in solidity to Move, and also beat the eixsting state of the art.

RQ3 *Can we generate code in a low-resource language the LLMs have not been trained on?*
Yes. Our empirical result suggests that without being trained in a specific language, an LLM can still translate an existing known language smart contract to a low-resource one with the help of sub-tasks and minimal fine-tuning.

Our iterative method of using both compiler and prover feedback also decreases bugs in the translated code and increases code correctness as we can see from Fig. 6. Contract Correctness and Fig. 5. Bug Mitigation. Though we also see that even with the feedback, there is negligible improvement hen it comes to prover feedback.

RQ4 *Mitigating bugs using compiler feedback*
Yes. Our empirical result suggests that iterative prompting with compiler feedback does help reduce the number of uncompilable codes. Both of the models, with higher effectiveness in our framework.

VII. Related Work

The related work can be categorized broadly as just Code Transpilers and LLM based solutions.

Rule-based Transpilers. A rule-based transpiler, like skilled interpreters, bridges the gap between programming languages. Instead of relying on guesswork, they follow predefined rules, ensuring accurate and predictable code transformations. This enables collaboration and code reuse, allowing users to seamlessly move their Python code to Java [56] or Java to Python [57]. While these tools offer precision and flexibility, they're not mind-reading magicians. Setting up the rules can require some programming expertise, and complex language features might need manual adjustments.

Statistical ML-based Transpilers. Statistical ML-based transpilers employ machine learning to translate between programming languages. Unlike rule-based methods, they learn from vast datasets of paired translations, identifying patterns and relationships to predict translations for unseen code. A phrase-based approach to code migration was introduced by Nguyen et al. [58], while Karaivanov et al. [59] enhanced this methodology by incorporating the grammatical structure of the target language and custom rules. Aggarwal et al. [60] applied a similar approach to convert Python2 to Python3, utilizing sentence alignments. Additionally, bidirectional transpilers, such as the one proposed by Schultes [61] for Swift [62] and Kotlin [63], have been explored. Ling et al.'s CRustS transpiler [64] stands out for its ability to reduce unsafe expressions in the target language through the implementation of extra transformation rules.

Transformer and Other ML-based Code Translation Tools. Transformer models, known for their magic in natural language translation, are now transforming code translation too! The encoder-decoder architecture introduced by Vaswani et al. [65] in Transformers has had a profound impact on natural language (NL) translation, capturing intricate contextual relationships among words. This breakthrough influence extended to the realm of software engineering (SE), sparking a renewed interest in crafting specialized language models tailored for the unique challenges of code translation tasks. Different Transformer flavors like CodeBERT [66]and CodeGPT [67]are being brewed, some focusing on understanding code, others on generating fluent target language. Wang et al. [68] introduced CodeT5, an encoder-decoder Transformer incorporating code semantics derived from developer-assigned identifiers, resulting in more precise and domain-specific translations. Ahmad et al. [69] explored a unified Transformer model, PLBART, trained through denoising autoencoding, enabling versatile performance across both natural language (NL) and programming languages (PLs). Researchers are even experimenting with incorporating developer hints and exploring alternative architectures like tree-to-tree models. Chen et al. [70] pushed beyond the traditional sequence-to-sequence paradigm by introducing a tree-to-tree neural network tailored specifically for program translation.

LLM-based Methods Using Compiler/unit testing. Roziere et al. [71] introduced an unsupervised code translation method that utilizes self-training and automated unit tests to ensure the equivalence of source and target code. It is noteworthy that they employ unit tests to construct a synthetic parallel dataset for model refinement; however, these tests are not integrated into the loss calculation during training. In a different context, Wang et al. [72] leverage RL with compiler feedback for code generation, again distinct from a supervised learning setting. Pan et al [73] did a comprehensive study on the existing LLMs for code translation. On the other side, Orlanski et al [74] explored the area of how low-resouce coding languages are being affected in the LLM era.

VIII. Conclusion

Here we have proposed SolMover, a composite two-LLM-based framework that encodes textbook knowledge in one to generate a granular subtask for the other to generate code based on that. We have evaluated the framework for the code translation task, involving an extremely low-resource code "Move" as a target language for smart contract creation. We have shown how our framework can take existing solidity smart contracts and translate them to generate move smart contracts. Our contributions include introducing a way to encode concepts into an LLM, and showing how it can help LLM translate code into a non-trained source language. We also evaluate our iterative compiler error feedback loop to show it can help mitigate bugs in the translated code.

References

[1] P. Liang, R. Bommasani, T. Lee, D. Tsipras, D. Soylu, M. Yasunaga, Y. Zhang, D. Narayanan, Y. Wu, A. Kumar *et al.*, "Holistic evaluation of language models," *ArXiv preprint*, vol. abs/2211.09110, 2022. [Online]. Available: https://arxiv.org/abs/2211.09110

[2] S. Bubeck, V. Chandrasekaran, R. Eldan, J. Gehrke, E. Horvitz, E. Kamar, P. Lee, Y. T. Lee, Y. Li, S. Lundberg *et al.*, "Sparks of artificial general intelligence: Early experiments with gpt-4," *ArXiv preprint*, vol. abs/2303.12712, 2023. [Online]. Available: https://arxiv.org/abs/2303.12712

[3] H. Wu, W. Wang, Y. Wan, W. Jiao, and M. R. Lyu, "Chatgpt or grammarly? evaluating chatgpt on grammatical error correction benchmark," *ArXiv preprint*, vol. abs/2303.13648, 2023. [Online]. Available: https://arxiv.org/abs/2303.13648

[4] S. R. Moghaddam and C. J. Honey, "Boosting theory-of-mind performance in large language models via prompting," *ArXiv preprint*, vol. abs/2304.11490, 2023. [Online]. Available: https://arxiv.org/abs/2304.11490

[5] W. Jiao, W. Wang, J. tse Huang, X. Wang, and Z. Tu, "Is chatgpt a good translator? a preliminary study," in *ArXiv*, 2023.

[6] S. Agrawal, C. Zhou, M. Lewis, L. Zettlemoyer, and M. Ghazvininejad, "In-context examples selection for machine translation," in *Findings of the Association for Computational Linguistics: ACL 2023*, A. Rogers, J. Boyd-Graber, and N. Okazaki, Eds. Toronto, Canada: Association for Computational Linguistics, 2023, pp. 8857–8873. [Online]. Available: https://aclanthology.org/2023.findings-acl.564

[7] B. Zhang, B. Haddow, and A. Birch, "Prompting large language model for machine translation: A case study," *ArXiv preprint*, vol. abs/2301.07069, 2023. [Online]. Available: https://arxiv.org/abs/2301.07069

[8] D. Vilar, M. Freitag, C. Cherry, J. Luo, V. Ratnakar, and G. Foster, "Prompting palm for translation: Assessing strategies and performance," *ArXiv preprint*, vol. abs/2211.09102, 2022. [Online]. Available: https://arxiv.org/abs/2211.09102

[9] H. Lu, H. Huang, D. Zhang, H. Yang, W. Lam, and F. Wei, "Chain-of-dictionary prompting elicits translation in large language models," *ArXiv preprint*, vol. abs/2305.06575, 2023. [Online]. Available: https://arxiv.org/abs/2305.06575

[10] Y. Bar-Hillel, "A demonstration of the nonfeasibility of fully automatic high quality translation," *Advances in computers*, vol. 1, pp. 158–163, 1960.

[11] E. Macklovitch, "The future of mt is now and bar-hillel was (almost entirely) right," in *Proceedings of the Fourth Bar-Ilan Symposium on the Foundations of Artificial Intelligence. url: http://rali. iro. umontreal. ca/Publications/urls/bisfai95. ps*, 1995.

[12] J. Benson, "Uniswap trading volume exploded by 450% to $7 billion. here's why," 2021. [Online]. Available: https://decrypt.co/63280/uniswap-trading-volume-exploded-7-billion-heres-why

[13] K. Rabimba, L. Xu, L. Chen, F. Zhang, Z. Gao, and W. Shi, "Lessons learned from blockchain applications of trusted execution environments and implications for future research," in *Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP '21. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3505253.3505259

[14] R. Karanjai, Z. Gao, L. Chen, X. Fan, T. Suh, W. Shi, and L. Xu, "Dhtee: Decentralized infrastructure for heterogeneous tees," in *2023 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, 2023, pp. 1–3.

[15] R. Karanjai, L. Xu, N. Diallo, L. Chen, and W. Shi, "Defaas: Decentralized function-as-a-service for emerging dapps and web3," in *2023 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, 2023, pp. 1–3.

[16] S. Blackshear, E. Cheng, D. L. Dill, V. Gao, B. Maurer, T. Nowacki, A. Pott, S. Qadeer, D. Russi, S. Sezer, T. A. K. Zakian, and R. Zhou, "Move: A languagewith programmable resources," 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:201681125

[17] (2019). [Online]. Available: https://librenotlibra.info/docs/LibraWhitePaper_en_US\protect\discretionary{\char\hyphenchar\font}{}{}1.pdf

[18] (2023). [Online]. Available: https://mitsloan.mit.edu/shared/ods/documents?PublicationDocumentID=5859

[19] J. Girard, Y. Lafont, and L. Regnier, *Advances in Linear Logic*, ser. Lecture note series / London mathematical society. Cambridge University Press, 1995. [Online]. Available: https://books.google.com/books?id=ROEf2h5FvD4C

[20] N. D. Matsakis and F. S. Klock II, "The rust language," in *ACM SIGAda Ada Letters*, vol. 34, no. 3. ACM, 2014, pp. 103–104.

[21] R. Karanjai, E. Li, L. Xu, and W. Shi, "Who is smarter? an empirical study of ai-based smart contract creation," in *2023 5th Conference on Blockchain Research Applications for Innovative Networks and Services (BRAINS)*, 2023, pp. 1–8.

[22] (2023) sui · github. [Online]. Available: https://github.com/MystenLabs/sui/tree/main/sui_programmability/examples/fungible_tokens

[23] J. E. Zhong, K. Cheang, S. Qadeer, W. Grieskamp, S. Blackshear, J. Park, Y. Zohar, C. Barrett, and D. L. Dill, "The move prover," in *Computer Aided Verification*, S. K. Lahiri and C. Wang, Eds. Cham: Springer International Publishing, 2020, pp. 137–150.

[24] R. Karanjai, E. Li, L. Xu, and W. Shi, "Who is smarter? an empirical study of ai-based smart contract creation," in *2023 5th Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*. IEEE, 2023, pp. 1–8.

[25] D. Gile, *Basic concepts and models for interpreter and translator training*, ser. Benjamins Translation Library. Amsterdam: John Benjamins, 2009, no. 8.

[26] S. Gunasekar, Y. Zhang, J. Aneja, C. C. T. Mendes, A. Del Giorno, S. Gopi, M. Javaheripi, P. Kauffmann, G. de Rosa, O. Saarikivi *et al.*, "Textbooks are all you need," *arXiv preprint arXiv:2306.11644*, 2023.

[27] J. Zhang, A. Muhamed, A. Anantharaman, G. Wang, C. Chen, K. Zhong, Q. Cui, Y. Xu, B. Zeng, T. Chilimbi *et al.*, "Reaugkd: Retrieval-augmented knowledge distillation for pre-trained language models," 2023.

[28] S. Lu, N. Duan, H. Han, D. Guo, S.-w. Hwang, and A. Svyatkovskiy, "Reacc: A retrieval-augmented code completion framework," *arXiv preprint arXiv:2203.07722*, 2022.

[29] (2023) Introduction - the move book. [Online]. Available: https://move-language.github.io/move/

[30] (2022) The move language - the move book. [Online]. Available: https://move-book.com/

[31] (2023) Introduction - move patterns: Design patterns for resource based programming. [Online]. Available: https://www.move-patterns.com/

[32] (2023) move · github. [Online]. Available: https://github.com/move-language/move/tree/main/language/documentation/tutorial

[33] (2023) Sui basics - sui move by example. [Online]. Available: https://examples.sui.io/basics/index.html

[34] (2023) hf-codegen. [Online]. Available: https://github.com/sayakpaul/hf-codegen/blob/main/data/prepare_dataset.py

[35] V. Karpukhin, B. Oğuz, S. Min, P. Lewis, L. Wu, S. Edunov, D. Chen, and W.-t. Yih, "Dense passage retrieval for open-domain question answering," *arXiv preprint arXiv:2004.04906*, 2020.

[36] S. Robertson, H. Zaragoza *et al.*, "The probabilistic relevance framework: Bm25 and beyond," *Foundations and Trends® in Information Retrieval*, vol. 3, no. 4, pp. 333–389, 2009.

[37] A. v. d. Oord, Y. Li, and O. Vinyals, "Representation learning with contrastive predictive coding," *arXiv preprint arXiv:1807.03748*, 2018.

[38] R. Taori, I. Gulrajani, T. Zhang, Y. Dubois, X. Li, C. Guestrin, P. Liang, and T. B. Hashimoto, "Stanford alpaca: An instruction-following llama model," https://github.com/tatsu-lab/stanford_alpaca, 2023.

[39] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar *et al.*, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[40] ShareGPT, "Sharegpt: Share your wildest chatgpt conversations with one click." 2023, available at: https://sharegpt.com/.

[41] J. Urban, "Mptp–motivation, implementation, first experiments," *Journal of Automated Reasoning*, vol. 33, pp. 319–339, 2004.

[42] (2023) fungible tokens. [Online]. Available: https://github.com/MystenLabs/sui/tree/main/sui_programmability/examples/fungible_tokens

[43] (2023) move/language/documentation/examples/experimental/basic-coin at main · move-language/move · github. [Online]. Available: https://github.com/move-language/move/tree/main/language/documentation/examples/experimental/basic-coin

[44] (2023) Diem. [Online]. Available: https://github.com/0LNetworkCommunity/libra-legacy-v6/blob/main/language/diem-framework/modules/Diem.move

[45] (2023) Token. [Online]. Available: https://github.com/starcoinorg/starcoin-framework/blob/main/sources/Token.move

[46] (2023) Gas. [Online]. Available: https://github.com/0LNetworkCommunity/libra-legacy-v6/blob/main/language/diem-framework/modules/0L/GAS.move

[47] (2023) Nft. [Online]. Available: https://github.com/MystenLabs/sui/tree/main/sui_programmability/examples/nfts

[48] (2023) starcoin-framework/sources/merklenft.move at main · starcoinorg/starcoin-framework · github. [Online]. Available: https://github.com/starcoinorg/starcoin-framework/blob/main/sources/MerkleNFT.move

[49] (2023) Defi. [Online]. Available: https://github.com/MystenLabs/sui/tree/main/sui_programmability/examples/defi

[50] (2023) move/language/documentation/examples/experimental/coin-swap at main · move-language/move · github. [Online]. Available: https://github.com/move-language/move/tree/main/language/documentation/examples/experimental/coin-swap

[51] (2023) Github - elements-studio/starswap-core: The swap project on starcoin such as uniswap a sushiswap. [Online]. Available: https://github.com/Elements-Studio/starswap-core

[52] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray *et al.*, "Training language models to follow instructions with human feedback," *Advances in Neural Information Processing Systems*, vol. 35, pp. 27 730–27 744, 2022.

[53] (2023) Can you feel the moe? mixtral available with over 100 tokens per second through together platform! [Online]. Available: https://www.together.ai/blog/mixtral

[54] R. Anil, A. M. Dai, O. Firat, M. Johnson, D. Lepikhin, A. Passos, S. Shakeri, E. Taropa, P. Bailey, Z. Chen *et al.*, "Palm 2 technical report," *arXiv preprint arXiv:2305.10403*, 2023.

[55] A. Hendy, M. Abdelrehim, A. Sharaf, V. Raunak, M. Gabr, H. Matsushita, Y. J. Kim, M. Afify, and H. H. Awadalla, "How good are gpt models at machine translation? a comprehensive evaluation," *ArXiv preprint*, vol. abs/2302.09210, 2023. [Online]. Available: https://arxiv.org/abs/2302.09210

[56] "`py2java`: Python to Java Language Translator," https://pypi.org/project/py2java/.

[57] T. Melhase *et al.*, "`java2python`: Simple but Effective Tool to Translate Java Source Code into Python," https://github.com/natural/java2python.

[58] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Lexical Statistical Machine Translation for Language Migration," in *Proceedings of the 9 Joint Meeting on Foundations of Software Engineering*, 2013, pp. 651–654.

[59] S. Karaivanov, V. Raychev, and M. Vechev, "Phrase-based Statistical Translation of Programming Languages," in *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, 2014, pp. 173–184.

[60] K. Aggarwal, M. Salameh, and A. Hindle, "Using Machine Translation for Converting Python 2 to Python 3 Code," PeerJ PrePrints, Tech. Rep., 2015.

[61] D. Schultes, "SequalsK – A Bidirectional Swift-Kotlin-Transpiler," in *2021 IEEE/ACM 8 International Conference on Mobile Software Engineering and Systems (MobileSoft)*. IEEE, 2021, pp. 73–83.

[62] "Swift: The Powerful Programming Language that is Also Easy to Learn," https://developer.apple.com/swift/.

[63] "Kotlin Programming Language: Concise. Cross-platform. Fun," https://kotlinlang.org/.

[64] M. Ling, Y. Yu, H. Wu, Y. Wang, J. R. Cordy, and A. E. Hassan, "In Rust We Trust: A Transpiler from Unsafe C to Safer Rust," in *Proceedings of the ACM/IEEE 44 International Conference on Software Engineering: Companion Proceedings*, 2022, pp. 354–355.

[65] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention Is All You Need," *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 30, 2017.

[66] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*. Online: Association for Computational Linguistics, Nov. 2020, pp. 1536–1547.

[67] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang *et al.*, "CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation," *arXiv preprint arXiv:2102.04664*, 2021.

[68] 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 *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, 2021, pp. 8696–8708. [Online]. Available: https://aclanthology.org/2021.emnlp-main.685

[69] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified Pre-training for Program Understanding and Generation," in *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Online: Association for Computational Linguistics, Jun. 2021, pp. 2655–2668. [Online]. Available: https://aclanthology.org/2021.naacl-main.211

[70] X. Chen, C. Liu, and D. Song, "Tree-to-Tree Neural Networks for Program Translation," *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 31, 2018.

[71] B. Roziere, J. Zhang, F. Charton, M. Harman, G. Synnaeve, and G. Lample, "TransCoder-ST: Leveraging Automated Unit Tests for Unsupervised Code Translation," in *International Conference on Learning Representations (ICLR)*, 2022. [Online]. Available: https://openreview.net/forum?id=cmt-6KtR4c4

[72] X. Wang, Y. Wang, Y. Wan, F. Mi, Y. Li, P. Zhou, J. Liu, H. Wu, X. Jiang, and Q. Liu, "Compilable Neural Code Generation with Compiler Feedback," in *Findings of the Association for Computational Linguistics: ACL 2022*, 2022, pp. 9–19.

[73] S. G. Haugeland, P. H. Nguyen, H. Song, and F. Chauvel, "Migrating monoliths to microservices-based customizable multi-tenant cloud-native apps," in *2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 2021, pp. 170–177.

[74] G. Orlanski, K. Xiao, X. Garcia, J. Hui, J. Howland, J. Malmaud, J. Austin, R. Singh, and M. Catasta, "Measuring the impact of programming language distribution," in *International Conference on Machine Learning*. PMLR, 2023, pp. 26 619–26 645.