Detailed prompts, Future Mapping and Error Tables

I. Deatiled prompts

In this research we used Large language models (LLMs) like GPT-4, Deepseek, Qwen and Claude for programming translation tasks. These models are trained on large-scale code repositories, enabling them to solve real-world programming challenges with limited human intervention. One of the most effective ways to harness the power of LLMs is through prompt engineering—the practice of carefully crafting inputs to guide the model's output. For programming tasks, prompt design is critical. The effectiveness of a prompt can mean the difference between compiling, working code and syntactic or logical errors. In this context, we used four core prompting techniques in the experimentation: zero-shot, few-shot, chain-of-thought, and few-shot + chain-of-thought prompting.

Prompt engineering plays an important role in optimizing LLM performance for specialized tasks like code translation. Our prompt selection was guided by established research on LLM capabilities [1], [2] and progressed systematically from basic to more sophisticated techniques. Table I summarizes the four prompting strategies employed in our study.

TABLE I
Prompting Strategies and Input Design

Prompt	Input Design
Zero-Shot	Direct instruction to translate Solidity to Algorand Python without additional context or examples.
Few-Shot	Includes multiple Solidity-to-Algorand Python examples before generating the target translation.
Chain-of-Thought	Guides the model through a structured reasoning process, outlining key architectural and syntactic differences before translation.
Few-Shot + CoT	Combines example-driven learning and structured reasoning with language mapping documentation to enhance translation accuracy and maintain fidelity.

We selected these prompting techniques based on their demonstrated effectiveness in complex reasoning and code generation tasks.

Zero-shot prompting served as our baseline, testing the models' inherent knowledge of both languages without additional context. This approach, while straightforward, helps establish the fundamental capabilities of each model.

In zero-shot prompting, no examples are provided to the model. The prompt contains only the task description, possibly with instructions or constraints, and the model is expected to generate the required code based solely on its pertaining.

Prompt

** Zero shot Example Task:**
Translation of Solidity smart contract into an equivalent Algorand Python smart contract.
"Translate the Solidity smart contract into an equivalent Algorand Python smart contract"

Solidity Smart Contract code to translate

Few-shot prompting leverages the models' ability to learn from examples, providing concrete translation patterns between Solidity and Algorand Python. This approach is particularly valuable for Algorand, which may be underrepresented in the models' training data. Our few-shot prompts included the mapping table (Table II), and two short corresponding smart contract examples that demonstrate basic constructs before presenting the translation task.

Few-shot prompting provides the model with several examples of input-output code pairs. This helps the model infer the desired format, language, or style of the output and adapt to specific problem types.

Prompt

Few Shot Example Task:
Translation of Solidity smart contract into an equivalent Algorand Python smart contract.
**Example 1:
Solidity Contract and its equivalent Algorand Python contract.
**Example 2:
Solidity Contract and its equivalent Algorand Python contract.
Now, translate the solidity smart contract into an equivalent Algorand Python smart contract.

Solidity Smart Contract code to translate

Chain-of-Thought (CoT) prompting exploits the models' reasoning capabilities by decomposing the translation process into discrete steps. This technique has shown particular effectiveness for complex tasks requiring multi-step reasoning [1]. We guided each model through a structured analysis of the input contract's features before generating Algorand code.

Chain-of-thought (CoT) prompting helps the model by guiding it through intermediate reasoning steps before writing the final code. In programming, this could mean first explaining the logic or algorithm and then translating that into code.

Prompt

Chain of thought Example Task:
Translation of Solidity smart contract into an equivalent Algorand Python smart contract.
Translate the Solidity smart contract into an equivalent Algorand Python smart contract following Step-by-Step Reasoning:
step 1. Identify Key Solidity Features
- What are the contract's functions? - What are the state variables? - Does it use inheritance, mappings, or payable functions?
step 2. Find Algorand Equivalents
- Convert Solidity contract structure to Algorand python using algopy class.
- Replace 'payable' functions with 'Txn.amount()'.
- Convert Solidity's 'msg.sender' to 'Txn.sender()'.
- Replace Solidity mappings with 'Boxmap' or 'Box Storage'.
step 3. Write Algorand python Code
- Implement contract logic in algorand python. - Ensure transaction security and permissions.
step 4. Verify and Optimize
- Ensure contract logic follows Algorand python constraints.
- Ensure the correct import statements and data types.
- Don't mix up algorand python and python data types.
- Optimize state storage and fees.
——————————————————
Solidity Smart Contract code to translate
——————————————————

Few-Shot + Chain-of-Thought Prompting integrates both example-based learning and structured reasoning. Research suggests this combined approach can improve performance on complex tasks.Similar to the few-short learning, the prompt includes the mapping table (Table I), and two short corresponding smart contract examples.

This method combines the strengths of few-shot prompting and chain-of-thought prompting. The model is shown multiple examples that include step-by-step reasoning followed by implementation.

Prompt

Few Shot + Chain of Thought Example Task:
Translation of Solidity smart contract into an equivalent Algorand Python smart contract.
**Example 1:
Solidity smart contract and its equivalent Algorand Python code.
**Example 2:
Solidity smart contract and its equivalent Algorand Python code.
Now, translate the solidity smart contract into an equivalent Algorand Python smart contract following step by step reasoning.
Step-by-Step Reasoning:
step 1. Identify Key Solidity Features
- What are the contract's functions? - What are the state variables? - Does it use inheritance, mappings, or payable functions?
step 2. Find Algorand Equivalents
- Convert Solidity contract structure to Algorand python using algopy class.
- Replace 'payable' functions with 'Txn.amount()'.
- Convert Solidity's 'msg.sender' to 'Txn.sender()'.
- Replace Solidity mappings with 'Boxmap' or 'Box Storage'.
step 3. Write Algorand python Code
- Implement contract logic in algorand python. - Ensure transaction security and permissions.
step 4. Verify and Optimize
- Ensure contract logic follows Algorand python constraints.
- Ensure the correct import statements and data types.
- Don't mix up algorand python and python data types.
- Optimize state storage and fees.
——————————————————
Solidity Smart Contract code to translate
——————————————————

II. Comprehensive Detailed Feature Mapping Table

We present a mapping between Solidity and Algorand Python language constructs to bridge the architectural differences between these blockchain platforms. Created through analysis of documentation, example contracts, and developer resources, this mapping identifies equivalent constructs across contract structure, data types, function modifiers, operations, and state management (Table II). This mapping serves dual purposes: it provides a structured reference for evaluating translation correctness and forms a key component of our prompting strategies. By incorporating this mapping into our prompts, we aimed to improved LLMs' understanding of platform differences and improve translation accuracy. To our knowledge, no existing studies focus specifically on Solidity-to-Algorand Python translation, making this mapping a novel contribution.

TABLE II
Feature Mapping: Solidity vs Algorand Python

Category	Solidity	Algorand Python
Contract Structure	contract X { ... } is inheritance	class X(algopy.ARC4Contract): Multiple base classes
Basic Types	uint256, uint8 address string bytes, bytes32 bool mapping	UInt64 Account String Bytes bool GlobalState
Complex Types	Arrays Structs Enums	Box, BoxRef Multiple boxes Constants
Function Modifiers	public view pure payable	@algopy.arc4.abimethod() @algopy.arc4.abimethod (readonly=True) @algopy.subroutine Check Txn.amount in function

Operations	require(x, msg)
	+=, -=, *=, /=
	revert("msg")
		assert x, msg
		Direct operators
		assert False, "msg"

Global Variables	msg.sender msg.value block.timestamp address(this)	Txn.sender Txn.amount Global.latest_timestamp Global.current _application_address
Data Storage	State Variables Mappings Arrays Local Variables	GlobalState GlobalState with mapping Box, BoxRef Method variables

Special Functions	constructor()
	fallback()
	receive()
		__init__() and create()
		approval_program()
		Handle in approval _program()

Visibility	public private internal external	@algopy.arc4.abimethod() @algopy.subroutine @algopy.subroutine @algopy.arc4.abimethod()

Events	event X(...)
	emit X(...)
		log(...)
		log(Bytes(...))

III. Comprehensive Detailed Error Table

Throughout our translation experience, we encountered recurring error patterns that represent the key challenges in cross-blockchain translation. Table III shows a preliminary categorization of these errors, which we found consistent across all models. State management discrepancies emerged as a primary challenge, with conceptual differences between Ethereum's global state and Algorand's key-value approach leading to persistent translation errors, particularly with complex data structures. Type system mismatches also proved problematic, as Solidity's static typing system differs significantly from Algorand Python's approach, resulting in frequent type errors that required manual correction.

Transaction handling represented another significant challenge area. Algorand's transaction model differs fundamentally from Ethereum's, causing consistent errors in payment verification and asset transfers. We also observed frequent API knowledge gaps, with all models occasionally attempting to use non-existent Algorand methods or incorrectly implementing Algorand-specific features. These patterns highlight areas where LLMs currently lack sufficient understanding of blockchain-specific concepts rather than general programming knowledge. Notably, approximately 60-70% of errors were related to syntax issues from the platform-specific constructs related to Algorand blockchain.

TABLE III
COMPREHENSIVE TABLE OF ERRORS IN ALGORAND PYTHON SMART CONTRACT EXECUTION

Category	**Error Message**
Attribute Errors	"BoxMap[String, VotingRound]" has no attribute "exists" "String" has no attribute "encode"
Name Errors	Name "gtxn" is not defined
Type System Errors	Expression has type "Any" "UInt64" not callable "Bytes" has no attribute "to uint64"
Argument Type Errors	Argument 1 to "join" has incompatible type "reversed[str]" Argument 1 to "join" has incompatible type "list[String]"
Logical Errors	assert not self.rounds.exists(vote_id) asset_transfer.xfer_asset == self.assetid.value
Transaction Errors	assert paymenttx.type_enum == 1 assert payment_tx.receiver == Txn.application_address
Unsupported Operations	Value of type Module is not indexable
Return Type Mismatch	Returning Any from function declared to return "Bytes"
Global/Local State Issues	GlobalState[Account] has no attribute "exists" Expression has type "Any" when accessing global state
Indexing & Boundaries Issues	Replace operation out of bounds (`index + len(new_value) <= len(original)`)
Decorator & Function Typing Issues	Untyped decorator makes function untyped Function is untyped after decorator transformation
Incompatible Type Assignments	Incompatible types in assignment (`Account` vs. `Bytes`)
Operand Type Issues	Unsupported operand types for == (`Account` and `Bytes`)

IV. CONCLUSION

This report explored LLMs for translating Solidity contracts to Algorand Python. While no model produced consistently compilable code, few-shot prompting with feature mappings significantly reduced development effort. LLMs showed strengths in translating basic structures but struggled with platform-specific features like state management and transaction handling. While LLMs show great promise in code generation, their susceptibility to introducing security vulnerabilities demands rigorous auditing, its convenience masks hidden risks that could turn into costly exploits if left unchecked. A hybrid approach proved most practical, where LLMs generate initial translations for developers to refine. Future purpose-built models with blockchain-specific training could dramatically improve translation quality. These findings aim to stimulate research that lowers barriers to entry for platforms like Algorand and enhances cross-blockchain interoperability.

REFERENCES

[1] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[2] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.