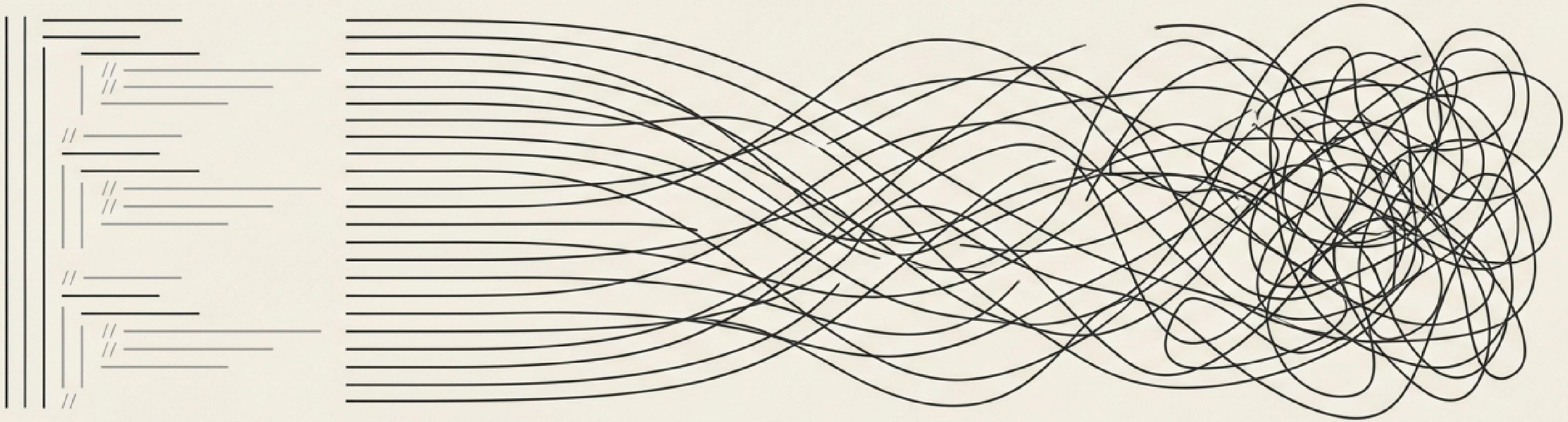
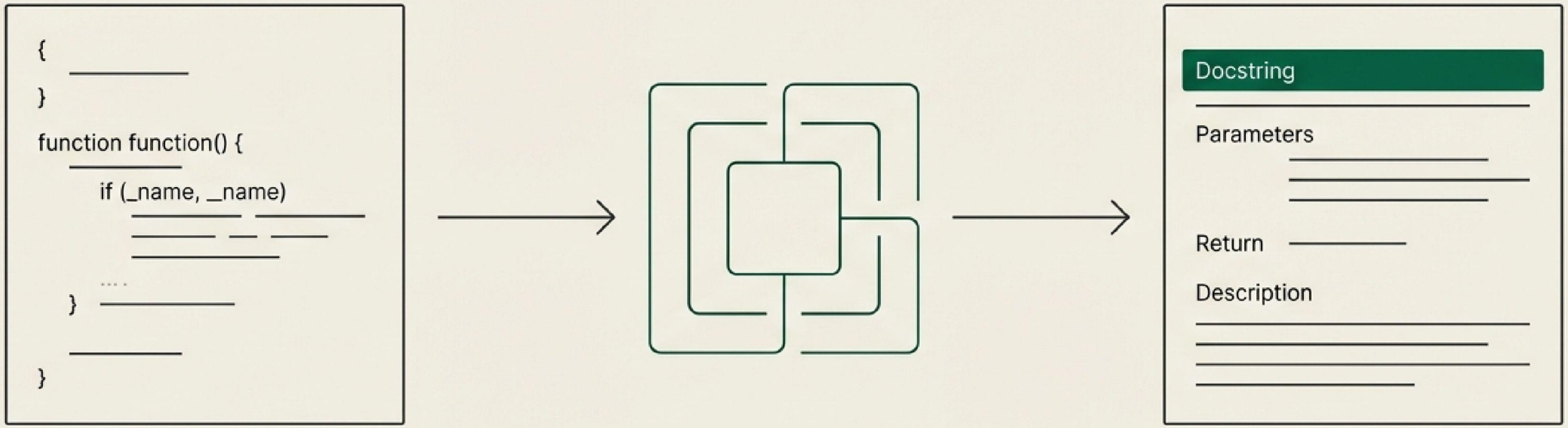
High-quality code documentation is a cornerstone of sustainable software engineering.

- It enhances code comprehensibility, facilitates team collaboration, and simplifies long-term maintenance.
- However, the manual creation and upkeep of documentation are often tedious, time-consuming, and neglected.
- This leads to documentation that is outdated, incomplete, or entirely absent, creating a significant drag on development velocity and project health.



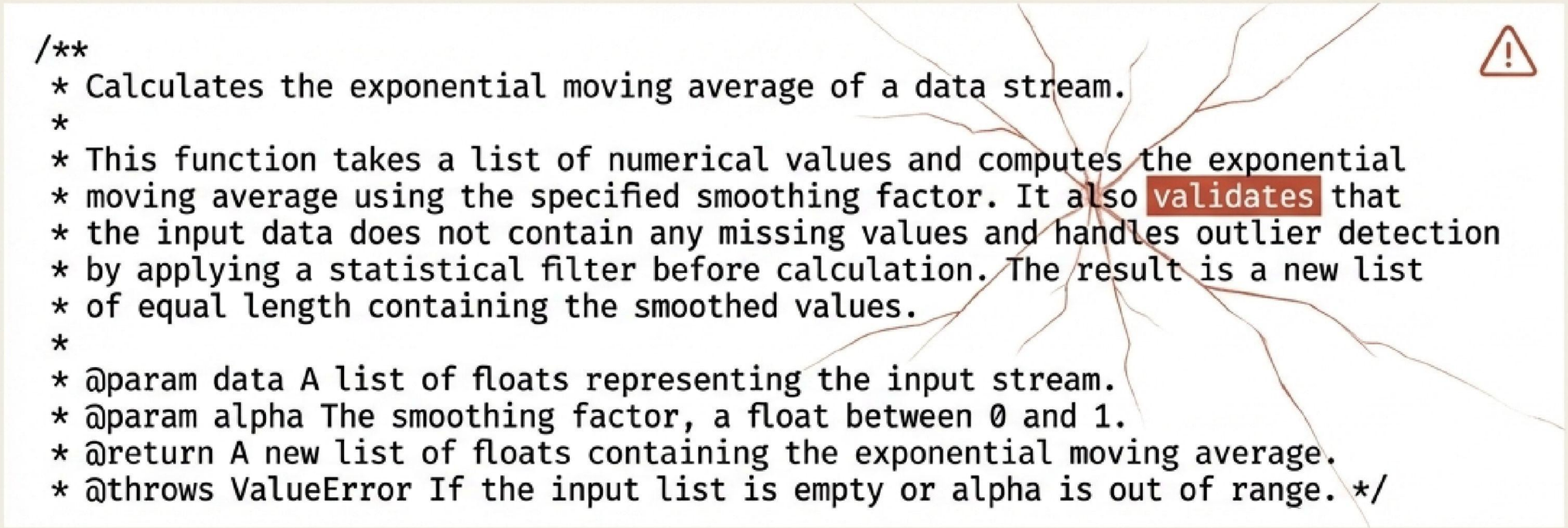
Large Language Models offer a powerful new path to automating code documentation.

- The advent of LLMs has enabled the automated generation of code documentation from source code, and vice versa.
- These models can significantly accelerate development by handling the tedious work of writing initial docstrings and explanations.



But this power comes with a critical flaw: LLMs are prone to “hallucinations.”

- LLM outputs are not infallible. They can be inconsistent, subtly inaccurate, or contain plausible-sounding but factually incorrect statements.
- The Core Problem: Fabrication. An LLM might generate a docstring that describes functionality the code does not possess.



```
/**
 * Calculates the exponential moving average of a data stream.
 *
 * This function takes a list of numerical values and computes the exponential
 * moving average using the specified smoothing factor. It also validates that
 * the input data does not contain any missing values and handles outlier detection
 * by applying a statistical filter before calculation. The result is a new list
 * of equal length containing the smoothed values.
 *
 * @param data A list of floats representing the input stream.
 * @param alpha The smoothing factor, a float between 0 and 1.
 * @return A new list of floats containing the exponential moving average.
 * @throws ValueError If the input list is empty or alpha is out of range. */
```

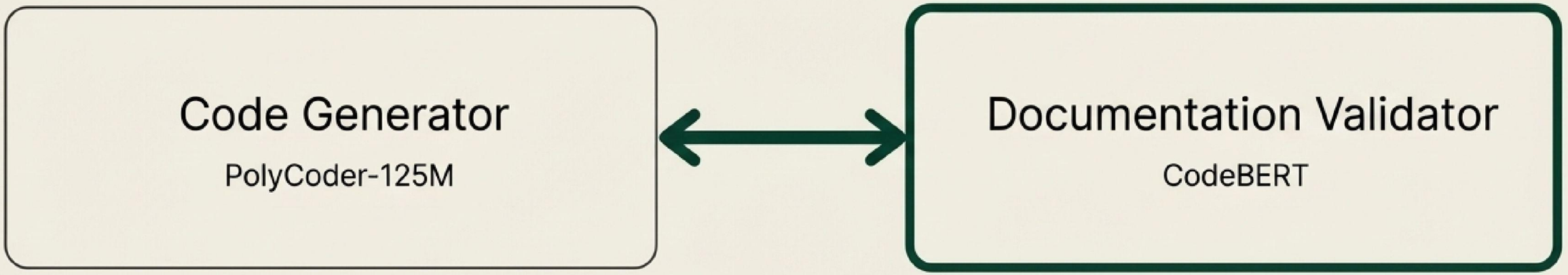
The true challenge is not generation, but ensuring the factual correctness of AI outputs.

- This research addresses the critical need for an automated validation mechanism.
- The goal is to programmatically ensure the semantic consistency and factual correctness of AI-generated code documentation.
- We need a safety guardrail to make AI-assisted development tools genuinely trustworthy.

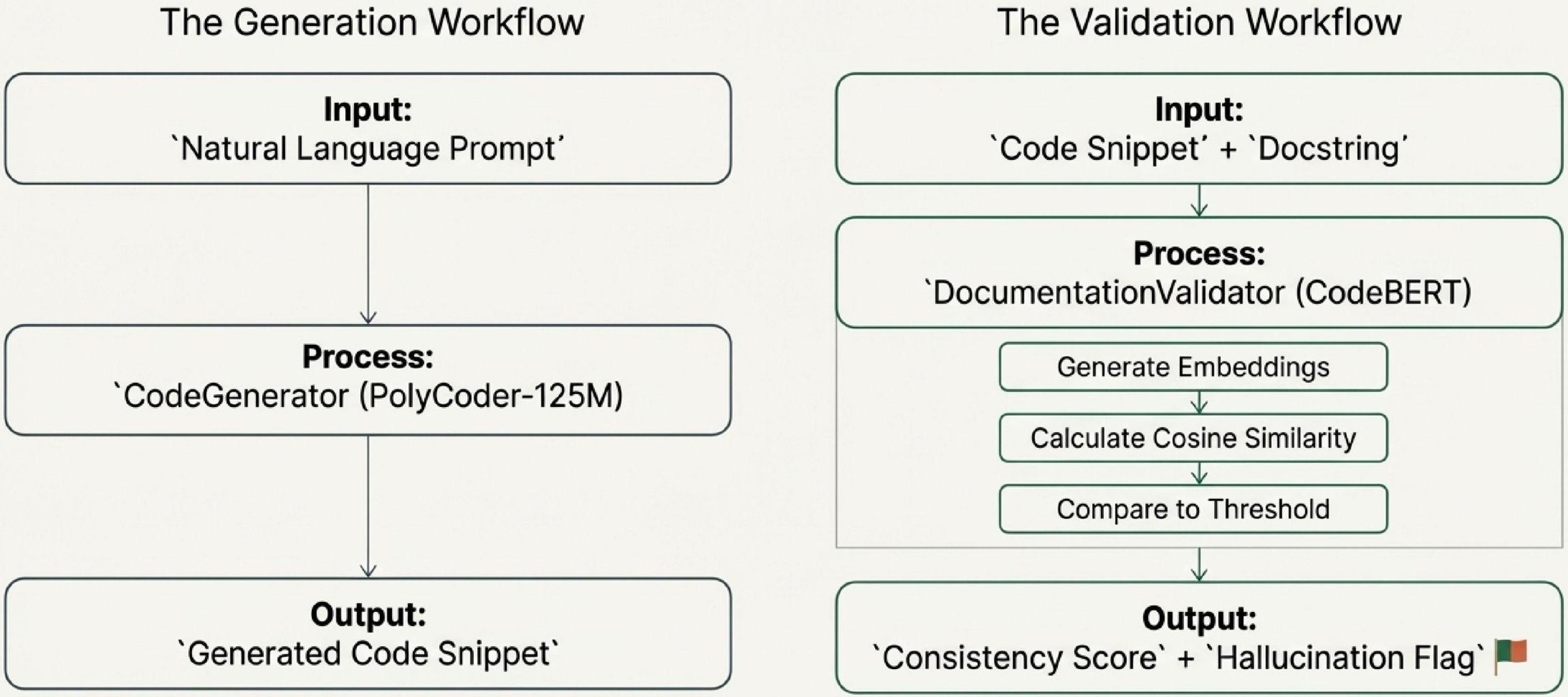
Our Solution: An Integrated System for AI-Powered Generation and Validation.

We present the “AI Code Documentation Validator & Generator,” a system with two primary functionalities:

- **Code Generation:** Leverages a transformer-based model (PolyCoder-125M) to produce source code from natural language prompts.
- **Documentation Validation:** Critically, it features a component that programmatically assesses the semantic consistency between a code snippet and its documentation.



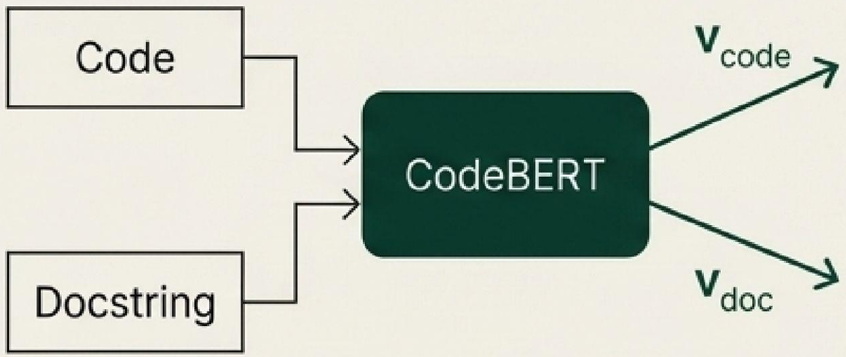
The system operates via two distinct, synergistic workflows.

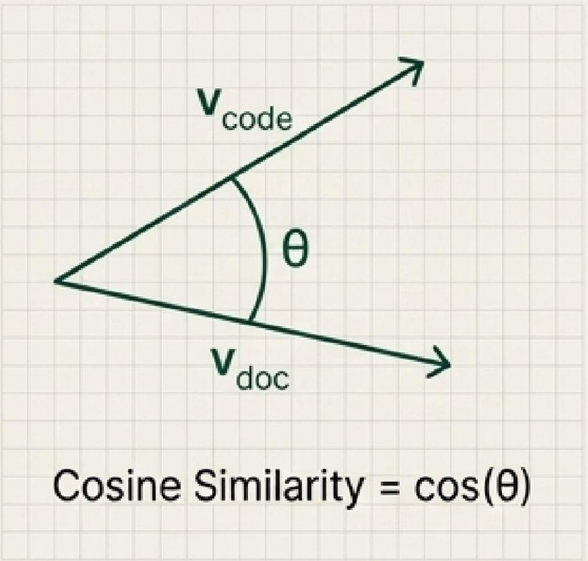


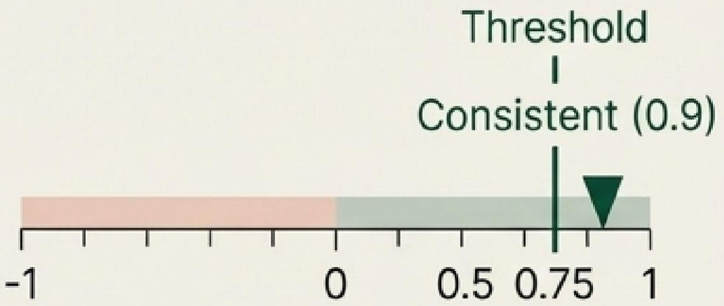
The validator uses semantic embeddings to programmatically measure alignment.

The validation process is a three-step method:

1. **Generate Semantic Embeddings:** The CodeBERT model creates high-dimensional vector representations of both the code and the docstring, projecting them into a shared semantic space.
2. **Calculate Cosine Similarity:** The system calculates the cosine of the angle between the two embeddings. The resulting score (from -1 to 1) quantifies their semantic proximity. A score near 1 indicates strong alignment.
3. **Detect 'Fabrication' Hallucinations:** The similarity score is compared against a pre-defined threshold of 0.75. If the score falls below this, the system flags the documentation as a potential fabrication.







The validator precisely distinguishes between accurate and fabricated documentation.

A qualitative test provides clear, reproducible evidence of the system's capability.

Case 1: Consistent Documentation

```
def add(a, b):
    return a + b

'Adds two numbers together.'
```

0.991

CONSISTENT

Case 2: Fabricated Documentation

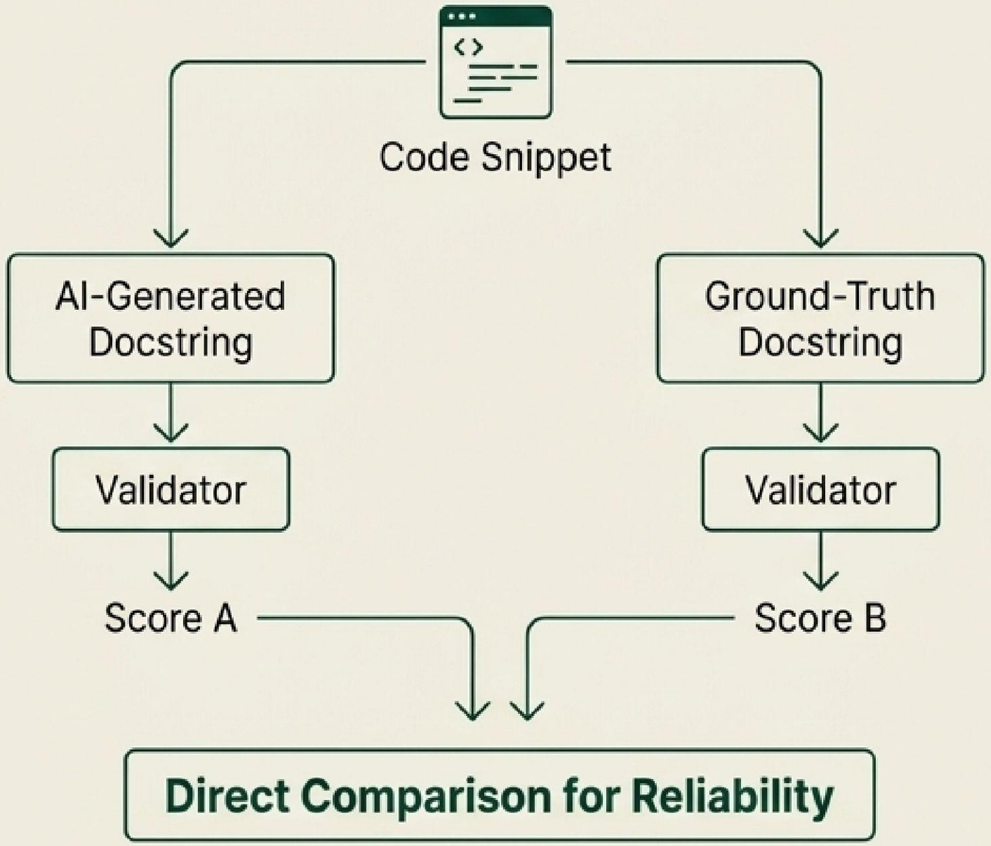
```
def multiply(x, y):
    return x * y

'Divides two numbers.'
```

0.412

FABRICATION DETECTED

Robustness was confirmed through automated batch evaluation against ground-truth data.

- **Experimental Setup:** An automated script (`batch_eval.py`) systematically assesses the validator's performance against the **CodeSearchNet (CSN)** dataset.
- **Evaluation Procedure:** For each code snippet, the process calculates two scores:
  1. Similarity between the code and the **AI-generated docstring**.
  2. Similarity between the same code and its **ground-truth (human-authored) docstring**.
- **Outcome:** This dual-validation approach allows for direct, scalable, and reproducible comparison, confirming the metric's reliability. The full results are logged in a structured `batch_eval.json` file.



The prototype was implemented using a modern, API-first technology stack.

A functional, end-to-end prototype was realized with a backend service and a lightweight frontend for interaction.

FastAPI

Exposing AI functionality as a high-performance RESTful API.

Streamlit

Creating an interactive web-based UI for demonstration.

Hugging Face `transformers`

Loading and running the CodeBERT and PolyCoder-125M models.

torch

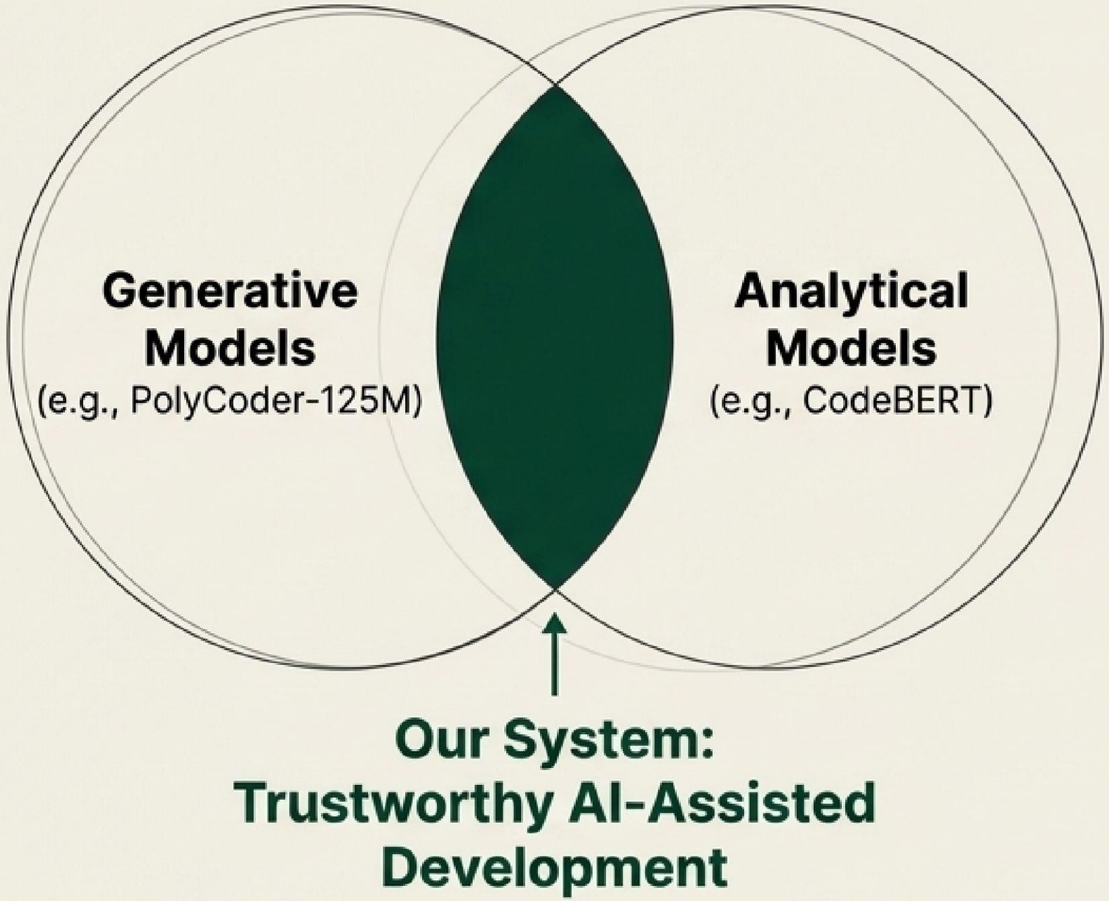
Providing the core deep learning framework for all model operations.

CodeSearchNet (CSN)

Serving as the ground-truth dataset for rigorous evaluation.

Our contribution lies in the synergistic integration of generation and semantic analysis.

- While prior work has focused on either code generation or semantic analysis in isolation, our system's novelty is in their combination.
- We pair a generative model (**PolyCoder-125M**) with an analytical model (**CodeBERT**) to create a holistic workflow.
- This integration is a direct countermeasure to the risk of hallucination, providing an automated quality assurance mechanism missing from purely generative tools.



This system provides a necessary blueprint for building safer, more trustworthy AI developer tools.

- By programmatically flagging semantic inconsistencies, we can build a new class of AI-powered tools that developers can rely on.
- This integrated approach of generation with validation demonstrates a practical and essential method for implementing safety guardrails for AI-assisted software development.
- The future of productive AI tooling depends on our ability to build in mechanisms for trust and verification from the ground up.