CODE TRANSLATION WITH TRANSFORMERS

PRESENTED BY:
Tejas Yalamanchili and
Shrinivas Venkatesan

Motivation

Manual Rewriting of Code is a cumbersome, error-prone, and expensive process

Python might be the easiest to code in but might not offer low-level features of C++

Legacy, rule-based conversions using ASTs are brittle and cannot handle library mappings

Why Transformers?

Reason #1:

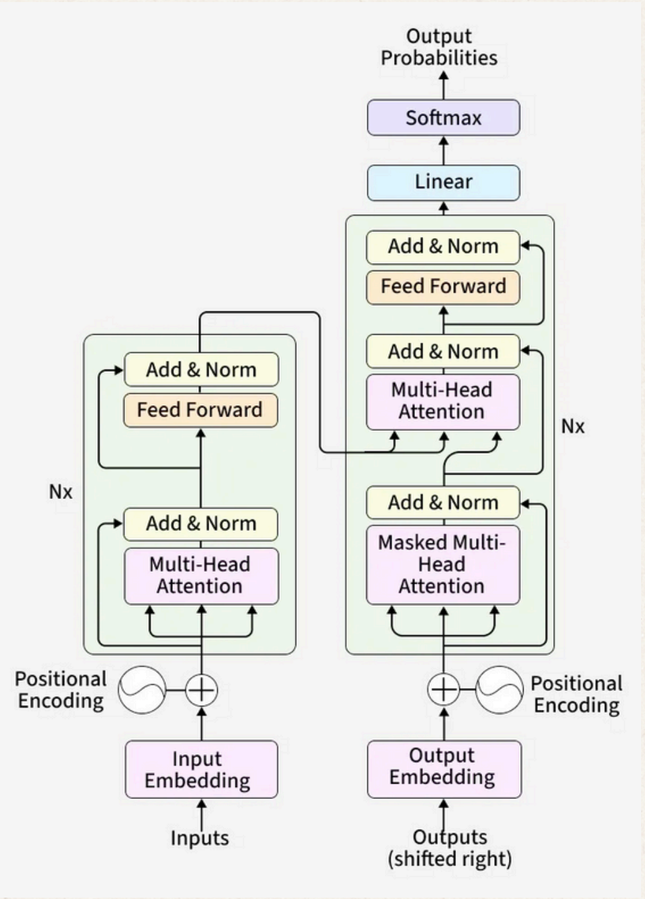
Treats code translation as a Machine Translation (NMT) problem (similar to English → French)

Reason #2:

Attention Mechanism allows the model to map dependencies between distant code parts (a variable definition at the top and its usage at the bottom)

Reason #3:

Can convert "Pythonic" patterns (like list comprehensions or dynamic typing) into performant, "native" C++ structures (like std::vector or std::transform) rather than doing a literal line-by-line translation



Methodology

Step #1: Data Collection

- Code snippets (functions or classes) , with paired translations
- Target Language dependency and library mapping

Step #2: Retrieval

- Embed source code chunks with CodeBERT
- Store vectors in an index; retrieve top-k similar examples at translation time

Step #3: Translator

- Feed source chunk + retrieved examples into a generative mode
- Output target code with the required style and constraints

Step #4: Verification Loop

- Compile and check for any issues
- If any errors, feed the logs back for a fix(repeat a few times)

Step #5: Outputs

- Final translated code
- Also give a few metrics such as test pass rate and runtime parity checks

Why This would work

Problem with generation only

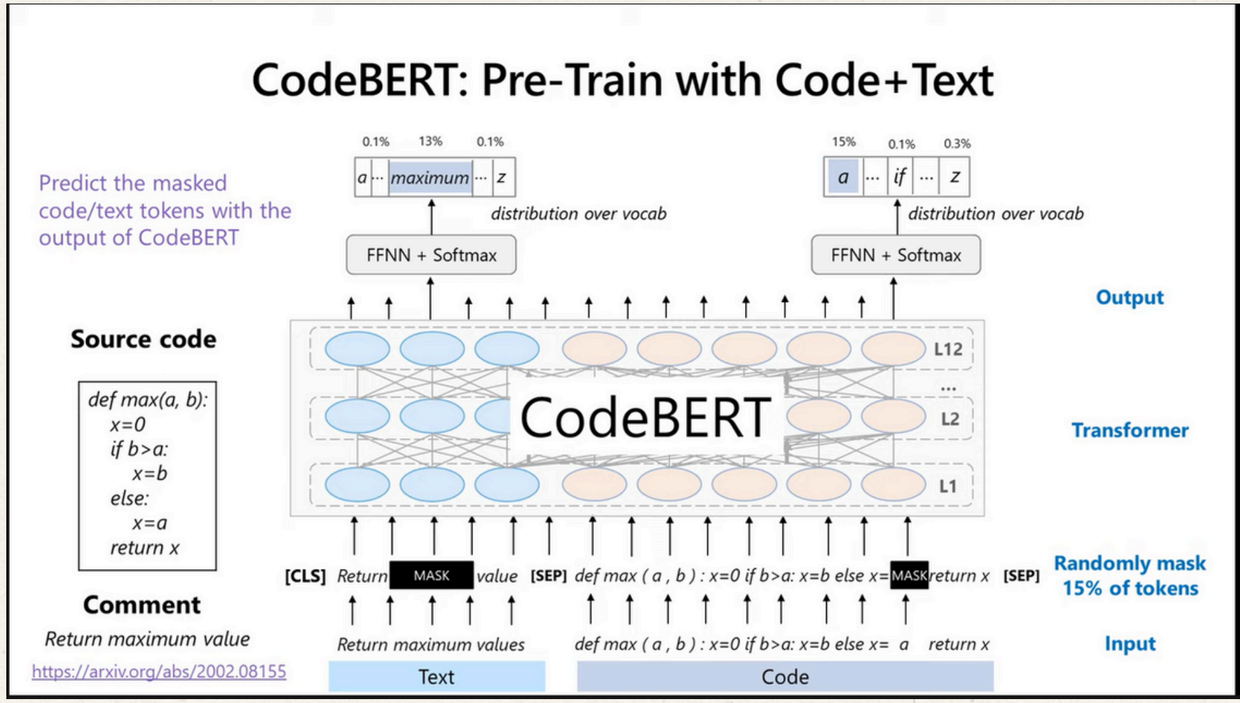
- Misses project-specific patterns and APIs
- Hallucinates Libraries
- Breaks subtle semantics

Reason to add CodeBERT

- It pulls closest real examples
- Anchors the model to correct target-language style and architecture
- Reduces ambiguity for framework heavy code

Reason to add a Verification Loop

- Translation must satisfy real toolchains and contstraints
- Tests give objective feedback
- Repair loop helps by converting failures into incremental patches



Problems We Might Face

Problem #1: Data and Coverage

- Few high-quality paired examples for niche libraries
- Domain or framework mismatch between retrieved examples and current code

Problem #2: Semantic Mismatches

- Dynamic vs static Typing, different memory model, and concurrency model differences.
- Even the exceptions, numeric types and ordering might differ from language to language

Problem #3: Evaluation

- Passing compilation and some tests doesn’t necessarily guarantee identical behviour
- There might be many edge cases, therefore the test cases need to be comprehensiv

Problem #4: Scale

- Apart from the Translator and BERT models, embedding cost and repeated verify/repair loops can add latency and also increase compute cost

Q & A