Surfacing Program Structure with Auxiliary Signals for Low-Resource Code Translation

Anonymous ACL submission

Abstract

LLMs are extremely good code translators, except when it matters most: translating from low-resource to high-resource programming languages. The reason may lie in the fact that code translation relies on superficial lexical cues rather than on exploiting the deeper, language-agnostic program structure. We propose *BoJac* (*Bohm–Jacopini*), a novel commenting approach that pushes LLMs to exploit program structure by verbalizing structured-programming principles in a guided natural-language format. By using structure, *BoJac* enhances the ability to translate low-resource-to-high-resource code. Using equivalent code benchmarks across multiple language pairs, we compare *BoJac* against several forms of auxiliary guidance provided alongside the source code: synthetic free-form chain-of-comments, abstract syntax tree, and I/O Traces. To disentangle structural understanding from identifier-level matching, we repeat all experiments under name mangling and extend the same comparison to LoRA Fine-tuning on signal-augmented training sets. Our results show that the strongest gains come from auxiliary signals expressed in natural language, particularly when they explicitly encode structural and computational-level properties of the source code as in *BoJac*. While purely structural representations appear less effective, hybrid approaches such as *BoJac*, which convey structural and paradigm-level information through formalised natural language, emerge as a promising direction.

1 Introduction

LLMs are extremely good at code translation, except when it matters most: translating from low-resource to high-resource Programming Languages (PLs). This shortfall has real consequences. Many widely deployed systems are written in languages that are underrepresented in modern LLM training data (e.g., banking systems written in COBOL),

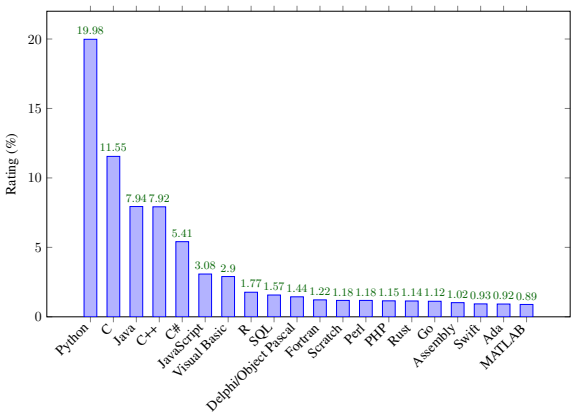

Figure 1: Programming languages popularity. The heavy tail continues further.Ratings indicate search engine queries rather than total use such as number of lines of code commercially deployed, as this is difficult to measure. Source: `www.tiobe.com`.

and translating automatically such code into widely supported targets is a recurring need for migration, interoperability, and modernisation (Nguyen et al., 2013; Yan et al., 2023). In addition, the overall presence of low-resource PLs in the software community is not to be underestimated (see Figure 1).

PLs differ in syntax and idiom, but the translation should be possible since often share the same underlying control-flow and data-flow structures. A competent programmer can often recognise these structures even in unfamiliar languages. Whether LLMs exhibit a similar abstraction capability is less clear. Most evidence of their coding competence comes from a narrow set of high-resource languages, such as Python, JavaScript, Java, and C++, which dominate pretraining corpora (Cassano et al., 2024; Orlanski et al., 2023; Xie et al., 2023).

The Böhm–Jacopini theorem (Böhm and Jacopini, 1966) offers a powerful simplification for making explicit the structure of programs across languages. The theorem shows that any computable function can be expressed through a small set of control-flow primitives, commonly understood as

sequence, selection, and iteration[1]. If LLMs rely on such abstractions, a promising direction can be to investigate what signal, provided alongside the source code, makes them salient during translation. Prior work has explored two broad families of auxiliary signals: natural-language guidance, such as Chain-of-Comments (Cao et al., 2025; Chen et al., 2024; Song et al., 2024), and more formal representations, such as abstract syntax trees (Gong et al., 2024) or I/O Traces. Natural-language signals are easier for LLMs to consume, while formal representations expose structure more explicitly, but neither directly foregrounds the control-flow primitives that structured programming theory identifies as cross-linguistic invariants.

In this paper, we propose *BoJac*, a semi-formal commenting strategy to address the challenge of translating from low-resource to high-resource PLs by exploiting the Böhm–Jacopini theorem. *BoJac* uses a teacher model to annotate low-resource source code in natural language, but constrains comments through a small vocabulary of structural tags such as DEFINE, LOOP, RETURN, CALL, and CONDITION. This yields a scaffold that remains readable as natural language while making the program's control-flow skeleton explicit. We benchmark *BoJac* against a range of auxiliary signals, including self- and teacher-generated chain-of-comments, ASTs, and I/O Traces, across multiple low-to-high resource translation pairs. We further evaluate all settings under name mangling, where identifiers are systematically obfuscated, to test whether performance reflects structural understanding rather than lexical cues reliance, and extend the comparison to parameter-efficient fine-tuning with LoRA.

Our main contributions are:

- We cast low-resource-to-high-resource code-to-code translation as a probe of structural code understanding in LLMs, conducting a systematic analysis across multiple source–target language pairs.
- We introduce *BoJac*, a semi-formal commenting strategy inspired by Böhm-Jacopini structured programming theory, which steers a teacher model to surface programming structure through a small set of control-flow tags embedded in natural-language comments.
- We augment existing code translation benchmarks with a name-mangling protocol that controls for identifier-level cues, isolating structural understanding from lexical pattern matching.

[1]In 1966 Böhm and Jacopini proved that any program can be built from three constructs: sequence, selection, and iteration.

2 Method

We develop our approach, *BoJac*, for code-to-code translation from comparatively lower-resource source languages to higher-resource target languages by using a principled approach based on the Böhm–Jacopini theorem. In this section, we describe our semi-formal principled approach in relation to informal and formal approaches to enhance code-to-code translations. These latter two approaches have never been used in the context to low-resource to high-resource code translation and, for this reason, are presented together with our *BoJac* approach. From the point of view of the tasks, all the methods are based on the enrichment the low-resource source programs with *auxiliary signals* that expose complementary aspects of computational meaning and organisation.

2.1 Informal Auxiliary Signals

Chain-of-Comments (*CoC*) have ben used as auxiliary signals to improve code-to-code traanslation. These signals are informal in the sense that *CoC* augments the source code with natural-language comments describing its main computational steps, *e.g.*, comments such as "iterate over the input list" or "return the accumulated result." These annotations make algorithmic intent explicit by verbalising local operations, control-flow decisions, and intermediate goals. We consider two variants. In the *self-generated* setting *CoC_S*, the same model used for translation first produces the comments and then translates the annotated code. In the *teacher-generated* setting *CoC_T*, the comments are instead produced by a stronger teacher model and provided to the student as part of the input. This distinction allows us to test not only whether linguistic guidance is useful, but also whether its usefulness depends on the quality of the model providing it.

2.2 Formal Auxiliary Signals

Formal languages such as Abstract Syntax Trees (ASTs) and Input/Output traces have been used to guide code generation and, in our case, we have used these formal languages as formal auxiliary signals as follows.

Abstract Syntax Tree (ASTs) provide a formal view of program structure by making the hier-

archical organisation of the source code explicit. In our setup, the model receives the source program together with a linearised AST represented as an *s-expression*, *e.g.*, fragments of the form `(function_declaration ... (for_statement ...))`. Declarations, control-flow constructs, expressions, and variable references are thus encoded through nested symbolic units. This representation exposes structural relations such as block nesting, statement composition, parameter structure, and operator scope without relying on natural-language description. The AST signal therefore tests whether purely symbolic structure can support code translation more effectively than language-based guidance.

Input/Output Traces (I/O Traces) augment the source code with a behavioural signal in the form of input–output examples specifying the expected function behaviour, *e.g.*, `f([3,1,2])` → `[1,2,3]`. In our setting, these examples are expressed in a compact arrow-style notation following the grammar of the source language. This formalism does not encode syntax or control flow explicitly, but provides concrete behavioural constraints that the translation should preserve. The intuition is that such traces may help the model recover program semantics from observed behaviour, especially when the mapping between source and target constructs is not immediately transparent from syntax alone.

2.3 *BoJac*: Hybrid Auxiliary Signals

BoJac is our new methodology to introduce formal directives in a semi-formal format, as if these directives were comments. *BoJac* is based on the solid formal theory of the Böhm–Jacopini theorem, but in *BoJac* we express this theory by using natural language directives expressing language-independent structured-programming primitives. Unlike free-form *CoC* or the extremely constrained-form of AST and I/O Traces, *BoJac* constrains annotations to a fixed inventory of structural natural language directives:

Class	*Semi-formal directive*

declarations	`DEFINE_CLASS`
	`DEFINE_METHOD`
	`DEFINE_FUNCTION`

data-flow and computational operations	`DEFINE_VARIABLE`
	`ASSIGN`
	`CALL`

control-flow and output behaviour	`CONDITION`
	`LOOP`
	`RETURN`

These directives are assigned to semantically relevant program units, while preserving the original code exactly, including identifiers, statement order, formatting, imports, braces, and blank lines.

BoJac therefore provides a controlled linguistic abstraction of program structure: it is more constrained than free-form natural-language comments, while remaining more readable and model-accessible than purely formal artefacts such as ASTs or I/O Traces. This makes it suitable for code-to-code translation, where the auxiliary signal should highlight control flow, data-flow updates, callable structure, and return behaviour in a form that can be aligned across PLs. A full specification of the tag set and annotation constraints is provided in Appendix B, with examples in Figure 2 and Appendix A.4.

3 Experiments

3.1 Experimental Setup

Lower-Resource Source Languages We consider Go, Rust, Kotlin, MATLAB, OCaml, and Cobol as comparatively lower-resource source languages, and translate them into higher-resource targets: Python, JavaScript, Java, and C++. We use *low-resource* in a relative sense: these languages are not equally scarce, but they are less represented than the target languages in the code corpora, benchmarks, developer communities, and documented resources from which code LLMs typically learn.

Our selection is also constrained by evaluation feasibility. Code-to-code translation requires parallel or functionally equivalent benchmarks across languages, so that outputs can be evaluated through unit tests. We therefore focus on languages for which equivalent benchmark pairs are available, while covering different degrees of resource availability: MATLAB, OCaml, and Cobol represent clearer low-resource cases, whereas Go, Rust, and Kotlin occupy an intermediate position. Further evidence supporting this classification is provided in Appendix D.

Dataset. We evaluate on test-set language pairs from the HumanEval family, RepoTrans, and CodeTrans, which provide functionally equivalent multilingual programs for unit-test-based code-to-code evaluation. We also introduce HumanEval-XNM, a name-mangled extension of HumanEval-X where user-defined identifiers are replaced with neutral placeholders while preserving program logic. Further details are in Appendix 6.

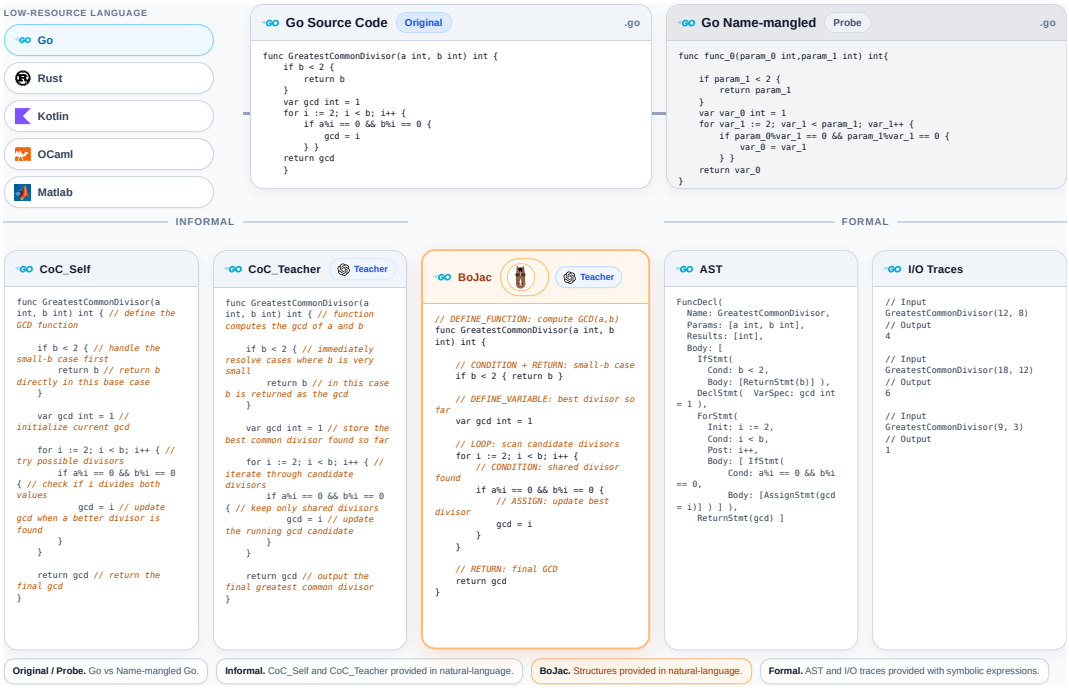

Figure 2: Overview of the auxiliary signal formats used in our code-to-code translation setting. Starting from a low-resource source language, here exemplified with Go, we compare free-form comment-based signals *CoC_Self*, *CoC_Teacher*, our semiformally guided representation *BoJac*, and more formal signals such as ASTs and I/O Traces. The figure also shows the name-mangled version of the source code used to probe reliance on identifier-level cues.

Post-processing and Evaluation Before evaluation, we post-process model outputs to isolate the translated program, removing explanations, Markdown formatting, and other non-code text. For Java and C++, we add only the minimal evaluator-specific boilerplate required for compilation and execution, such as class wrappers, imports/includes, entry points, and assertion harnesses; these components do not encode solution logic. The resulting program is compiled and run against the target-language tests, and is counted as correct only if all tests pass without compilation errors, runtime errors, or timeouts. Further details are provided in Appendix E.

Metrics We evaluate translation quality with *Pass@1*. For each task, the model produces a single candidate translation, which is compiled and executed against the corresponding unit tests. A translation is correct only if it passes all tests without compilation errors, runtime errors, or timeouts. Unlike more lenient Pass@k settings, *Pass@1* reflects a stricter fixed-budget scenario, assessing whether one generated program functionally preserves the behaviour of the source code.

Name-mangling To test whether translation performance relies on human-readable identifiers rather than program structure, we introduce a name-mangling condition in which function names, variable names, and other user-defined identifiers are systematically obfuscated. This transformation preserves syntax and execution semantics while removing lexical cues about the program's purpose. We apply it across all auxiliary-signal settings to assess which representations remain effective when models must rely on structural and behavioural information rather than naming. Following Paul et al. (2025), we use Tree-sitter[2] to parse each program and automatically rename functions and variables while preserving functional equivalence.

Fine-tuning In addition to in-context evaluation, we study whether auxiliary signals are also beneficial during training. To this end, we perform parameter-efficient fine-tuning with LoRA on training instances in which the source code is enriched with each type of auxiliary signal. This setting allows us to compare whether natural-language, structural, and computational signals differ not

[2]Tree-sitter library (https://tree-sitter.github.io/tree-sitter/).

only in their usefulness at inference, but also in the quality of the supervision they provide during adaptation. In this way, we test which form of code enrichment is most effective for learning code translation behaviour.

Models We evaluate code-to-code translation with instruction-tuned general-purpose LLMs from different families and scales, including Qwen2.5-1.5B-Instruct, Gemma-2-2B/9B, Gemma-3-4B, Llama-3.2-3B-Instruct, and gpt-oss-20B. Since our focus is abstraction and transferable program understanding, we do not restrict the evaluation to code-specialised models. This setup tests whether auxiliary signals remain effective across architectures and parameter sizes. For auxiliary-comment generation, we use stronger teacher models, such as GPT-5.1 and gpt-oss-120B, to produce high-quality Chain-of-Comments annotations and compare self-generated with teacher-generated guidance. Full model details are reported in Appendix 5.

4 Results and Analysis

Three findings emerge. Auxiliary signals improve translation across the board, with the gap widening under name-mangling. *BoJac* achieves the strongest results overall, outperforming both formal and informal signals. Its advantage is largest when identifier cues are removed, suggesting that semi-formal scaffolds compensate for lost lexical information most effectively.

4.1 Auxiliary Signals Utility

We analyse auxiliary-signal effectiveness in the original code-to-code setting. Table 1 shows that smaller models benefit more from natural-language guidance than from purely formal representations, generally preferring teacher-generated comments, *CoC_T*, to self-generated one *CoC_S*. This suggests that weaker models benefit from external linguistic guidance rather than deriving explanatory structure autonomously. Unless otherwise stated, *CoC_T* uses GPT-5.1 as teacher; this trend also holds on a subset of examples using the open, less proficient gpt-oss-120B teacher, as detailed in Appendix I.2. More broadly, formal signals such as ASTs and I/O Traces provide only limited and highly unstable gains across models and language pairs. By contrast, *BoJac* is often more consistent: by expressing structural-programming concepts through controlled natural language, it offers a robust compromise between free-form explanation and purely symbolic structure, frequently matching or improving over standard *CoC* forms.

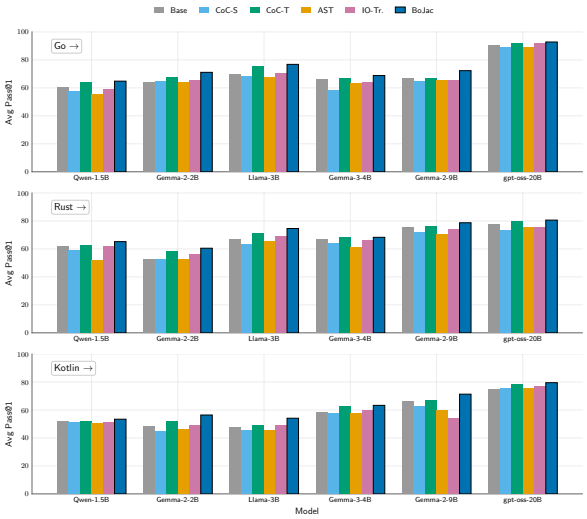

Figure 3: Average Pass@1 by source language for selected models. *BoJac* consistently outperforms both the baseline and *CoC_T*, indicating that structured natural-language guidance is robust across source languages.

Source Languages. Since HumanEval-X provides algorithmically equivalent problems across the studied languages, we analyse performance by fixing the source language and averaging Pass@1 over the four targets: Python, Java, JavaScript, and C++. This source-language view abstracts away from target-specific variation and isolates how auxiliary signals behave when translating from Go, Rust, and Kotlin. Numerical results are reported in Table 8, with a compact visualisation in Figure 3.

Overall, Go tends to be translated more accurately than Rust and Kotlin, consistently with its comparatively higher-resource status and stronger representation in code corpora. Across all source languages and models, *BoJac* provides the most consistent improvements. The gains are stable for Go, particularly clear for Rust, and especially useful for Kotlin, where performance is more variable across models. This suggests that guided structural comments help models recover transferable program logic when lexical or language-specific cues are weaker. Compared with *CoC_T*, *BoJac* is more robust because it constrains natural-language guidance through language-agnostic programming primitives.

4.2 Robustness to Name-mangling

Table 2 reports robustness under identifier obfuscation on a subset of language pairs. The *Base*

Go →	Python						Java						JavaScript						C++					
Model	Base	CoC_S	CoC_T	AST	IO-Tr.	BoJac	Base	CoC_S	CoC_T	AST	IO-Tr.	BoJac	Base	CoC_S	CoC_T	AST	IO-Tr.	BoJac	Base	CoC_S	CoC_T	AST	IO-Tr.	BoJac
Qwen2.5-1.5B	68.9	61.4	70.7	70.1	68.9	**73.2**	52.4	54.2	59.5	57.3	47.5	**62.2**	75.0	73.2	**78.2**	45.7	74.4	76.2	46.3	40.2	48.2	**48.8**	45.6	47.8
Gemma-2-2B	58.9	67.7	69.5	58.6	63.4	**71.7**	50.6	54.2	60.1	58.5	54.2	**63.4**	88.1	88.1	90.6	**93.2**	92.7	92.1	57.9	47.5	50.6	44.5	50.6	**57.3**
Llama-3.2-3B	74.4	70.1	73.2	68.2	71.3	**77.8**	52.4	57.3	68.9	68.2	58.5	**69.5**	91.5	90.9	**94.5**	67.7	88.1	92.1	61.5	54.2	65.9	65.9	64.0	**67.7**
Gemma-3-4B	60.9	52.4	62.4	53.5	60.9	**64.0**	58.9	49.8	60.1	52.4	50.6	**62.2**	84.5	88.6	88.1	87.9	**90.9**	87.9	60.9	42.2	57.6	58.5	54.2	**60.9**
Gemma-2-9B	77.4	76.2	77.4	65.2	74.4	**79.3**	35.8	32.5	34.6	39.6	35.8	**42.7**	86.5	87.5	89.2	90.1	92.1	**93.9**	68.9	61.4	67.7	67.1	58.5	**73.2**
gpt-oss-20B	91.5	90.9	93.2	89.1	**94.5**	**94.5**	93.3	91.5	94.5	89.1	91.5	**95.2**	96.3	93.2	**97.2**	95.7	96.3	96.3	80.9	80.9	84.1	80.9	84.7	**85.1**

Rust →	Python						Java						JavaScript						C++					
Model	Base	CoC_S	CoC_T	AST	IO-Tr.	BoJac	Base	CoC_S	CoC_T	AST	IO-Tr.	BoJac	Base	CoC_S	CoC_T	AST	IO-Tr.	BoJac	Base	CoC_S	CoC_T	AST	IO-Tr.	BoJac
Qwen2.5-1.5B	62.2	54.2	63.4	50.6	63.4	**65.8**	52.4	**62.2**	59.7	49.8	55.5	60.1	82.0	74.4	79.0	65.5	80.1	**83.5**	49.3	46.3	47.5	40.2	47.5	**51.6**
Gemma-2-2B	64.2	61.4	69.6	70.1	73.2	**74.4**	51.2	55.5	60.0	47.5	56.5	**62.2**	40.2	40.2	**42.7**	39.6	42.3	41.5	54.2	51.6	60.9	54.2	51.6	**64.0**
Llama-3.2-3B	70.1	71.3	**74.4**	69.5	71.7	**74.4**	60.4	61.9	69.5	62.2	64.0	**73.7**	71.7	60.1	75.6	73.2	74.4	**78.5**	64.0	59.7	65.9	57.9	65.5	**71.7**
Gemma-3-4B	69.5	65.7	**71.2**	66.7	69.5	68.9	63.9	63.9	65.5	60.1	67.7	**69.5**	78.5	73.7	**79.3**	68.9	74.4	78.5	54.1	52.4	**56.5**	47.5	53.5	**56.5**
Gemma-2-9B	75.0	74.4	77.6	73.2	77.7	**78.8**	65.2	64.0	67.7	64.0	61.9	**69.5**	92.7	77.7	87.2	85.4	**93.2**	92.1	68.9	71.3	73.2	57.9	64.0	**74.4**
gpt-oss-20B	78.8	75.6	80.1	76.2	78.1	**81.8**	67.7	66.4	69.5	68.9	66.4	**71.7**	91.5	91.5	**96.3**	90.1	92.7	95.7	71.7	60.9	71.7	65.6	64.0	**73.2**

Kotlin →	Python						Java						JavaScript						C++					
Model	Base	CoC_S	CoC_T	AST	IO-Tr.	BoJac	Base	CoC_S	CoC_T	AST	IO-Tr.	BoJac	Base	CoC_S	CoC_T	AST	IO-Tr.	BoJac	Base	CoC_S	CoC_T	AST	IO-Tr.	BoJac
Qwen2.5-1.5B	60.3	55.5	57.3	54.2	58.5	**59.7**	40.8	40.8	41.5	40.2	41.8	**42.7**	89.3	90.9	**91.1**	90.2	88.1	90.2	15.9	17.2	18.5	15.9	16.6	**20.9**
Gemma-2-2B	48.8	42.6	46.7	36.4	47.5	**50.6**	40.2	38.4	42.7	43.2	40.2	**47.5**	84.1	80.1	85.9	**87.4**	82.5	86.6	20.1	18.2	31.1	16.6	25.0	**40.9**
Gemma-3-4B	56.7	55.6	**62.2**	50.5	54.2	55.6	49.5	44.7	51.5	50.2	55.3	**57.5**	77.5	79.3	81.1	82.5	78.3	**84.5**	50.6	50.6	54.2	47.5	52.4	**55.5**
Llama-3.2-3B	56.7	56.7	60.9	52.4	55.5	**62.2**	48.8	45.7	46.3	43.2	49.5	**51.6**	42.6	40.2	45.7	45.7	**48.1**	45.7	42.7	40.1	43.9	39.6	43.2	**56.7**
Gemma-2-9B	67.7	62.2	69.5	65.5	71.2	**73.2**	51.2	47.5	45.7	40.2	51.2	**54.2**	88.4	85.4	89.1	82.5	87.5	**92.1**	56.7	54.2	63.4	49.5	5.6	**65.9**
gpt-oss-20B	82.3	83.0	88.4	**90.2**	89.2	**90.2**	76.9	79.4	82.3	73.0	78.1	**82.5**	**93.1**	**93.1**	92.7	91.5	**93.8**	**93.1**	47.5	46.2	50.6	45.6	46.9	**52.4**

Table 1: Pass@1 accuracy on Code-To-Code under different auxiliary signals on HumanEval-X. Green cells indicate improvements over the *Base* setting, while red cells indicate drops.

column gives direct translation on name-mangled code, while red values show the drop relative to the corresponding *Base* result on the original dataset. Name-mangling consistently reduces Pass@1, confirming that code-to-code translation benefits from semantic cues in variable and function names. The effect is stronger for smaller models, suggesting greater reliance on lexical information and weaker recovery of program logic from structure alone.

In this setting, *BoJac* is generally more effective than teacher-generated *CoC_T*. Since *CoC_T* is generated over name-mangled code, free-form comments can become less reliable: without meaningful identifiers, the teacher may infer incorrect or vague task-level interpretations. By contrast, *BoJac* constrains the signal through language-agnostic programming primitives, making comments more abstract and structurally grounded. This helps explain why *BoJac* better mitigates name-mangling degradation. Section I.1 further supports this interpretation, showing that *BoJac* annotations remain more faithful to program structure than unguided *CoC* generated by GPT-5.1.

4.3 Fine-tuning

Beyond in-context prompting, we test whether auxiliary signals remain useful when incorporated into training data. We use parameter-efficient fine-tuning with LoRA to adapt the model while updating only a small number of parameters, and ask which signal provides the most effective supervision for low-resource-to-high-resource code translation.

Based on the prompting results, we restrict fine-tuning to the two strongest natural-language signals: teacher-generated Chain-of-Comments (*CoC_T*) and *BoJac*. We exclude formal signals such as ASTs and I/O Traces, which were less competitive in the in-context setting and would substantially increase the number of runs. This focuses the comparison on whether structured natural-language supervision improves adaptation, and whether the constrained *BoJac* signal is preferable to free-form teacher comments. We fine-tune on a subset of source–target language pairs under three conditions: original translation pairs without annotation, source programs enriched with *CoC_T*, and source programs enriched with *BoJac* comments anchored on semiformal structural directives. Each example contains the low-resource source program, the target-language declaration, and the corresponding high-resource implementation. The target implementation is kept fixed across conditions; only the source-side representation changes, isolating the effect of the auxiliary signal. At evaluation time, all fine-tuned models follow the same protocol as

HumanEval-X	Go → JS			Rust → Python			Kotlin → Java		
Model	Base	CoC_T	BoJac	Base	CoC_T	BoJac	Base	CoC_T	BoJac
Qwen2.5-1.5B	72.5 (-2.5)	73.7	**79.5**	58.1 (-4)	55.6	**62.2**	19.9 (-20)	25.00	**30.7**
Gemma-2-2B	39.6 (-19)	52.5	**59.5**	41.6 (-23)	47.5	**54.2**	15.9 -24	8.7	**10.4**
Llama-3.2-3B	84.7 (-7)	67.1	**76.9**	53.5 (-4)	57.6	**59.5**	39.6 (-9)	42.7	**49.5**
Gemma-3-4B	62.5 (-16)	74.4	**78.7**	54.8 (-15)	60.9	**64.0**	34.2 (-15)	30.7	**40.3**
Gemma-2-9B	42.6 (-44)	60.6	**65.9**	63.9 (-11)	56.7	**67.7**	32.5 -19	46.3	**49.5**
gpt-oss-20B	88.1 (-3)	85.9	**91.5**	67.8 (-11)	76.2	**81.8**	68.9 (-8)	**75.0**	73.2

Table 2: Name-mangled code-to-code translation results on a subset of language pairs. We report Pass@1 for baseline translation, teacher-generated Chain-of-Comments (*CoC_T*), and *BoJac*. Bold indicates the best method for each model–source pair. Red values indicate the drop with respect to the corresponding *Base* result on the original dataset.

Pair	Model	Original	CoC_T	BoJac
Rust → Java	Qwen2.5-1.5B	59.1 (+7)	61.6 (+9)	63.4 (+11)
	Gemma-2-2B	52.4 (+1)	57.3 (+6)	61.4 (+10)
Kotlin → Python	Qwen2.5-1.5B	66.7 (+6)	62.2 (-1)	64.0 (+4)
	Gemma-2-2B	47.5 (-1)	51.6 (+3)	56.7 (+8)
Rust → C++	Qwen2.5-1.5B	57.9 (+13)	63.3 (+16)	68.8 (+21)
	Gemma-2-2B	58.1 (+3)	56.4 (+2)	62.2 (+8)

Table 3: Pass@1 results for LoRA fine-tuning under three training conditions: original source code, teacher-generated Chain-of-Comments (*CoC_T*), and *BoJac*-enriched source code.

in-context prompting and are tested on HumanEval-X. Details of the fine-tuning setup are reported in Appendix 4.3.

Very Low-Resource Languages. Table 4 reports results on very low-resource source languages, including OCaml, MATLAB, and COBOL. Overall, the same trend observed in the main experiments holds in this more challenging setting: the strongest auxiliary signals are those mediated through natural language. *CoC_T* often improves over the *Base* setting, showing that teacher-generated comments provide useful guidance when the source language is less familiar to the model. However, *BoJac* is the most consistent signal, achieving the best result in most settings. This suggests that, for very low-resource languages, models benefit not only from natural-language explanations, but especially from comments constrained by semi-formal programming primitives that expose transferable program structure.

4.4 Error Analysis

To understand how auxiliary signals help, we classify every failed translation into four categories: *compile errors* (the generated code does not compile), *runtime failures* (it compiles but crashes or times out), *wrong output* (it runs but fails at least one assertion), and *harness mismatches* (failures attributable to the evaluation scaffold's expected signature rather than to the translated program). Figure 4 reports this breakdown on Rust → Java for a small model (Qwen2.5-1.5B), its fine-tuned variant, and a larger model (gpt-oss-20B), comparing the *Base*, *CoT_T*, and *BoJac* conditions. Errors shift markedly with model capability: the small model's failures concentrate in compile and wrong-output errors, indicating difficulty in producing valid, semantically faithful Java, whereas the larger model's residual errors collapse almost entirely into wrong output, reflecting subtle semantic mistranslations rather than failures of basic competence. Across all three models, *BoJac* reduces errors in every category, not merely the total: it lowers compile and runtime failures while also shrinking the harder wrong-output bucket, suggesting that surfacing structural and computational properties of the source program helps both the syntactic and the semantic dimensions of translation.

5 Background and Related Work

Transversal Code-Generation Tasks. Recent work argues that code LLMs should be evaluated on tasks requiring syntactic and semantic program understanding (Pei et al., 2023). Functional equivalence studies analyse models' limitations in distinguishing semantics-preserving from semantics-altering transformations (Maveli et al., 2025), while termination prediction requires reasoning about control flow, loops, and potentially unbounded execution (Sultan et al., 2026). Relatedly, few-shot grammar generation studies examine how LLMs infer structural rules underlying text and code gen-

OCaml →	Python						C++						Java					
Model	Base	CoC_S	CoC_T	AST	IO-Tr.	BoJac	Base	CoC_S	CoC_T	AST	IO-Tr.	BoJac	Base	CoC_S	CoC_T	AST	IO-Tr.	BoJac
Gemma-3-4B	47.8	42.8	55.6	46.1	47.1	**57.6**	30.7	25.0	31.1	30.2	30.7	**34.2**	17.1	7.5	10.4	9.7	16.6	**20.9**
gpt-oss-20B	92.2	90.9	**94.5**	82.1	91.8	93.2	65.8	62.2	67.7	62.7	64.9	**71.3**	47.8	36.4	39.6	44.3	40.3	**52.4**

MATLAB →	Python					
Model	Base	CoC_S	CoC_T	AST	Ex.Tr.	BoJac
Gemma-3-4B	75.6	75.9	78.1	73.4	75.7	**80.9**
gpt-oss-20B	85.4	84.0	91.4	84.9	84.9	**92.7**

COBOL →	Python					
Model	Base	CoC_S	CoC_T	AST	Ex.Tr.	BoJac
Gemma-3-4B	55.5	29.5	**57.5**	42.5	**57.5**	**57.5**
gpt-oss-20B	85.6	59.6	85.6	82.2	82.9	**87.5**

Table 4: Pass@1 accuracy on Code-To-Code under different auxiliary signals on very-low resource source-languages.Green cells indicate improvements over the *Base* setting, while red cells indicate drops.

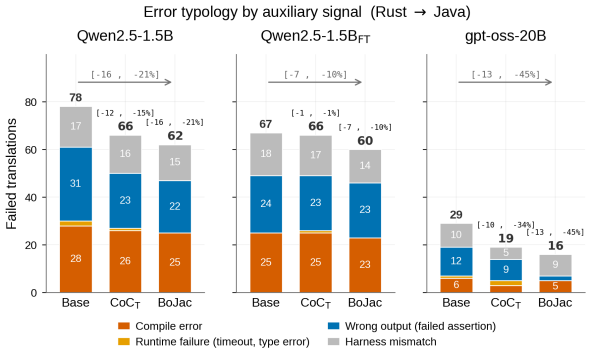

Figure 4: Error typology on Rust → Java across auxiliary signals (BASE, *CoC_T*, *BoJac*) for 3 models (gpt-oss-20B, Qwen2.5-1.5B and its finetuned version with different auxiliary signals). Bar height is the number of failed translations; segments give the four error categories. *BoJac* reduces errors across all categories for every model.

eration (Tang et al., 2025). More broadly, work on LLMs as compilers supports viewing code generation as part of program understanding rather than pure next-token prediction (Chae et al., 2024; Jiang et al., 2025).

Auxiliary Representations for Code Generation. Most LLM-based code-to-code translation work focuses on high-resource source–target language pairs and relies on large-scale parallel or synthetic datasets (Chae et al., 2024; Jana et al., 2024; Eniser et al., 2025; Zhang et al., 2025b). In contrast, we ask whether auxiliary representations can mitigate data scarcity by exposing program structure and behaviour more explicitly. Formal signals, such as compiler IR, can support multilingual generation and cross-lingual robustness (Paul et al., 2024), but LLMs may struggle to exploit purely compiler-oriented abstractions (Zhang et al., 2025a). Other methods use natural-language or semi-structured intermediates, including pseudocode, comments, structured reasoning traces, and execution traces (Cao et al., 2025; Chen et al., 2024; Li et al., 2025; Lavon et al., 2025; Ni et al., 2024). This motivates our comparison of formal, natural-language, and hybrid auxiliary signals for code translation.

Low-Resource Programming Languages. Recent work shows that code LLM performance varies substantially across programming languages, and that targeted transfer can partially reduce gaps for lower-resource PLs (Cassano et al., 2024), although no single strategy is uniformly effective (Giagnorio et al., 2025). We build on this line by studying low-resource-to-high-resource code translation as a probe of program understanding, comparing formal, natural-language, and hybrid auxiliary signals under standard and name-mangled conditions. Related work on GRPO for code functional equivalence similarly frames multilingual program understanding as a semantics-centred, cross-language problem (Wu et al., 2025).

6 Conclusion

Our code-to-code translation results reveal a consistent pattern: models benefit more from auxiliary representations expressed in natural language than from purely formal signals such as ASTs or I/O traces. This suggests that, even for code, natural-language scaffolding remains an effective interface for eliciting latent program knowledge. Crucially, *BoJac* shows that this interface can be made more structured: by encoding programming primitives in a semi-formal natural-language format, it exposes syntactic and computational information that models can exploit both at inference time, through in-context learning, and during adaptation, through fine-tuning. More broadly, these findings point to semi-formally guided representations as a promising direction for improving models' grasp of structured programming principles and core aspects of computational thinking.

Limitations

We translate from a limited number of source languages into four mainstream targets. While this spans a range of programming paradigms, it excludes other relevant low-resource programming languages, including Ruby and Swift, languages following substantially different paradigms such as Prolog, and lower-level languages such as Assembly. These languages may pose distinct challenges for code-to-code translation, either because of limited representation in training corpora, paradigm-specific abstractions, or closer interaction with machine-level details. Investigating whether structural signals such as *BoJac* transfer effectively to these settings is a direction we leave to future work.

Our evaluation also relies on a limited set of parallel equivalent-code examples, which constrains the statistical power of our analysis, so the trends we report should be read as indicative rather than definitive.

We assess translations primarily through execution-based correctness (pass@k), which captures functional equivalence but not idiomaticity, readability, or maintainability, and more formal methods—semantic equivalence checking beyond test-suite pass rates, or static analysis of generated code—would give a fuller picture of quality.

Finally, our method augments the input under LoRA fine-tuning; we do not explore reinforcement learning from execution feedback (e.g. GRPO) or larger-scale supervised fine-tuning, both of which could further improve code-to-code translation and interact with structural signals in ways we leave to future work.

References

Corrado Böhm and Giuseppe Jacopini. 1966. Flow diagrams, turing machines and languages with only two formation rules. *Commun. ACM*, 9(5):366–371.

Liuwen Cao, Hongkui He, Hailin Huang, Jiexin Wang, and Yi Cai. 2025. Rethinking-based code summarization with chain of comments. In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 3043–3056, Abu Dhabi, UAE. Association for Computational Linguistics.

Federico Cassano, John Gouwar, Francesca Lucchetti, Claire Schlesinger, Anders Freeman, Carolyn Jane Anderson, Molly Q Feldman, Michael Greenberg, Abhinav Jangda, and Arjun Guha. 2024. Knowledge transfer from high-resource to low-resource programming languages for code llms. *Preprint*, arXiv:2308.09895.

Hyungjoo Chae, Yeonghyeon Kim, Seungone Kim, Kai Tzu iunn Ong, Beong woo Kwak, Moohyeon Kim, Seonghwan Kim, Taeyoon Kwon, Jiwan Chung, Youngjae Yu, and Jinyoung Yeo. 2024. Language models as compilers: Simulating pseudocode execution improves algorithmic reasoning in language models. *Preprint*, arXiv:2404.02575.

Yijie Chen, Yijin Liu, Fandong Meng, Yufeng Chen, Jinan Xu, and Jie Zhou. 2024. Comments as natural logic pivots: Improve code generation via comment perspective. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 7040–7051, Bangkok, Thailand. Association for Computational Linguistics.

Ole-Johan Dahl, Edsger W. Dijkstra, and C. A. R. Hoare. 1972. *Structured Programming*. Academic Press, London, UK.

Edsger W. Dijkstra. 1968. Letters to the editor: go to statement considered harmful. *Commun. ACM*, 11(3):147–148.

Hasan Ferit Eniser, Hanliang Zhang, Cristina David, Meng Wang, Maria Christakis, Brandon Paulsen, Joey Dodds, and Daniel Kroening. 2025. Towards translating real-world code with llms: A study of translating to rust. *Preprint*, arXiv:2405.11514.

Alessandro Giagnorio, Alberto Martin-Lopez, and Gabriele Bavota. 2025. Enhancing code generation for low-resource languages: No silver bullet. *Preprint*, arXiv:2501.19085.

Linyuan Gong, Mostafa Elhoushi, and Alvin Cheung. 2024. Ast-t5: Structure-aware pretraining for code generation and understanding. *Preprint*, arXiv:2401.03003.

Prithwish Jana, Piyush Jha, Haoyang Ju, Gautham Kishore, Aryan Mahajan, and Vijay Ganesh. 2024. *CoTran: An LLM-Based Code Translator Using Reinforcement Learning with Feedback from Compiler and Symbolic Execution*. IOS Press.

Hailong Jiang, Jianfeng Zhu, Yao Wan, Bo Fang, Hongyu Zhang, Ruoming Jin, and Qiang Guan. 2025. Can large language models understand intermediate representations in compilers? *Preprint*, arXiv:2502.06854.

Mohammad Abdullah Matin Khan, M Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. 2023. xcodeeval: A large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. *Preprint*, arXiv:2303.03004.

Boaz Lavon, Shahar Katz, and Lior Wolf. 2025. Execution guided line-by-line code generation. *Preprint*, arXiv:2506.10948.

Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2025. Structured chain-of-thought prompting for code generation. *ACM Trans. Softw. Eng. Methodol.*, 34(2).

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, and 47 others. 2024. Starcoder 2 and the stack v2: The next generation. *Preprint*, arXiv:2402.19173.

Nickil Maveli, Antonio Vergari, and Shay B Cohen. 2025. What can large language models capture about code functional equivalence? In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 6880–6918, Albuquerque, New Mexico. Association for Computational Linguistics.

Greg Michaelson. 2020. Programming paradigms, turing completeness and computational thinking. *The Art, Science, and Engineering of Programming*, 4(3).

Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. 2013. Lexical statistical machine translation for language migration. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, page 651–654, New York, NY, USA. Association for Computing Machinery.

Ansong Ni, Miltiadis Allamanis, Arman Cohan, Yinlin Deng, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2024. Next: Teaching large language models to reason about code execution. *Preprint*, arXiv:2404.14662.

Gabriel Orlanski, Kefan Xiao, Xavier Garcia, Jeffrey Hui, Joshua Howland, Jonathan Malmaud, Jacob Austin, Rishabh Singh, and Michele Catasta. 2023. Measuring the impact of programming language distribution. *Preprint*, arXiv:2302.01973.

Indraneil Paul, Goran Glavaš, and Iryna Gurevych. 2024. IRCoder: Intermediate representations make language models robust multilingual code generators. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15023–15041, Bangkok, Thailand. Association for Computational Linguistics.

Indraneil Paul, Haoyi Yang, Goran Glavaš, Kristian Kersting, and Iryna Gurevych. 2025. Obscuracoder: Powering efficient code lm pre-training via obfuscation grounding. *Preprint*, arXiv:2504.00019.

Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2023. Can large language models reason about program invariants? In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 27496–27520. PMLR.

Demin Song, Honglin Guo, Yunhua Zhou, Shuhao Xing, Yudong Wang, Zifan Song, Wenwei Zhang, Qipeng Guo, Hang Yan, Xipeng Qiu, and Dahua Lin. 2024. Code needs comments: Enhancing code LLMs with comment augmentation. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 13640–13656, Bangkok, Thailand. Association for Computational Linguistics.

Oren Sultan, Jordi Armengol-Estape, Pascal Kesseli, Julien Vanegue, Dafna Shahaf, Yossi Adi, and Peter O'Hearn. 2026. Llms versus the halting problem: Revisiting program termination prediction. *Preprint*, arXiv:2601.18987.

Weizhi Tang, Yixuan Li, Chris Sypherd, Elizabeth Polgreen, and Vaishak Belle. 2025. HyGenar: An LLM-driven hybrid genetic algorithm for few-shot grammar generation. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 13640–13665, Vienna, Austria. Association for Computational Linguistics.

Yanli Wang, Yanlin Wang, Suiquan Wang, Daya Guo, Jiachi Chen, John Grundy, Xilin Liu, Yuchi Ma, Mingzhi Mao, Hongyu Zhang, and Zibin Zheng. 2025. Repotransbench: A real-world multilingual benchmark for repository-level code translation. *Preprint*, arXiv:2412.17744.

Niklaus Wirth. 1971. Program development by stepwise refinement. *Commun. ACM*, 14(4):221–227.

Haoyuan Wu, Rui Ming, Jilong Gao, Hangyu Zhao, Xueyi Chen, Yikai Yang, Haisheng Zheng, Zhuolun He, and Bei Yu. 2025. On-policy optimization with group equivalent preference for multiprogramming language understanding. *Preprint*, arXiv:2505.12723.

Yiqing Xie, Atharva Naik, Daniel Fried, and Carolyn Rose. 2023. Data augmentation for code translation with comparable corpora and multiple references. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 13725–13739, Singapore. Association for Computational Linguistics.

Weixiang Yan, Yuchen Tian, Yunzhe Li, Qian Chen, and Wen Wang. 2023. CodeTransOcean: A comprehensive multilingual benchmark for code translation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 5067–5089, Singapore. Association for Computational Linguistics.

Jipeng Zhang, Jianshu Zhang, Yuanzhe Li, Renjie Pi, Rui Pan, Runtao Liu, Zheng Ziqiang, and Tong Zhang. 2025a. Bridge-coder: Transferring model capabilities from high-resource to low-resource programming language. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 10865–10882, Vienna, Austria. Association for Computational Linguistics.

Longhui Zhang, Bin Wang, Jiahao Wang, Xiaofeng Zhao, Min Zhang, Hao Yang, Meishan Zhang, Yu Li, Jing Li, Jun Yu, and Min Zhang. 2025b. Function-to-style guidance of llms for code translation. *Preprint*, arXiv:2507.11083.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2024. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. *Preprint*, arXiv:2303.17568.

A Auxiliary Signals

We report illustrative examples of the auxiliary signals used in our experiments, shown as they are integrated with the source code. These examples are intended to clarify the format of each signal and the type of information it provides to the model at inference time.

A.1 Chain-of-Comments

```
def gcd(a, b):          # compute the greatest common divisor
    while b:            # repeat until b becomes zero
        a, b = b, a % b # apply Euclid's update rule
    return a            # return the final gcd
```

A.2 Abstract Syntax Tree

Source code + AST

```
def gcd(a, b):
    while b:
        a, b = b, a % b
    return a
\\
(function_definition
  (identifier 'gcd')
  (parameters
    (identifier 'a')
    (identifier 'b'))
  (block
    (while_statement
      (identifier 'b')
      (block
        (assignment
          (identifier 'a')
          (identifier 'b')
          (binary_expression
            (identifier 'a')
            '%'
            (identifier 'b')))))
    (return_statement
      (identifier 'a'))))
```

A.3 I/O Traces

```
def gcd(a, b):
    while b:
        a, b = b, a % b
    return a
\\
gcd(10, 15) -> 5
gcd(14, 21) -> 7
gcd(9, 28)  -> 1
```

A.4 BoJac

```
# DEFINE_FUNCTION: compute the greatest common divisor
def gcd(a, b):
    # LOOP: repeat until b becomes zero
    while b:
        # ASSIGN: apply Euclid's update rule
        a, b = b, a % b
    # RETURN: output the final gcd
    return a
```

B *BoJac* Annotation Specification

BoJac annotations are generated under a constrained commenting protocol. Given an input program, the annotator must preserve the original code exactly and may only insert comments using the native comment syntax of the source language. The comments are restricted to the following tag inventory: DEFINE_CLASS, DEFINE_METHOD, DEFINE_FUNCTION, DEFINE_VARIABLE, CONDITION, LOOP, ASSIGN, CALL, and RETURN. Tags are assigned only to semantically meaningful units: declarations, control-flow constructs, value updates, function or method calls, and returned expressions. The annotator is instructed not to rename identifiers, reorder statements, reformat the code, remove imports or delimiters, or add comments to trivial syntax. This yields a semi-formal representation in which each comment has the form: `<native-comment-marker> TAG: concise structural description`.

```
// DEFINE_FUNCTION: return the positive elements of the input
     slice
func GetPositive(l []int) []int {                  //
     DEFINE_FUNCTION: return the positives of the input slice

    // DEFINE_VARIABLE: res accumulates positive integers
    res := make([]int, 0)
    // LOOP: iterate over the elements of l
    for _, x := range l {
        // CONDITION: check whether the current element is
     positive
        if x > 0 {
            // CALL: append x to the result slice
            res = append(res, x)
        }
    }
    // RETURN: output the slice of positive integers
    return res
}
```

B.0.1 Origins of the *BoJac* Tag Vocabulary

The Böhm-Jacopini theorem (Böhm and Jacopini, 1966) did not merely establish a foundational result on control-flow expressiveness; it inspired an active line of research on structured programming : a programming methodology centred on the discipline of expressing computation through a small, well-defined set of control-flow constructs. In *BoJac* we leverage the fact that most of these constructs are shared across programming languages (Michaelson, 2020).

Influential contributions to this tradition include Dijkstra's critique of unrestricted `goto` (Dijkstra, 1968), Wirth's stepwise refinement methodology (Wirth, 1971), and the structured-programming texts of Dahl, Dijkstra, and Hoare (Dahl et al., 1972), which established the modern vocabulary of structured constructs: sequencing, selection, iteration, procedure definition, procedure call, and explicit return.

The *BoJac* tag vocabulary — DEFINE, LOOP, CONDITION, CALL, RETURN — draws directly on this lineage. LOOP and CONDITION correspond to the iteration and selection primitives of the Böhm-

Jacopini result. DEFINE, CALL, and RETURN cover the procedural extensions that became standard in subsequent structured-programming work and underpin the modular control flow of every imperative or multi-paradigm language we consider. We also extend DEFINE to DEFINE CLASS to cover class declarations in object-oriented contexts, since the modular boundary it marks generalises naturally from procedures to classes. The five tags are thus not an arbitrary design choice but a minimal vocabulary identified by half a century of research on what is essential for expressing structured computation. We deliberately do not include tags for paradigm-specific constructs (e.g., pattern matching, monadic binding, ownership transfer), since these vary across our source and target languages and would compromise the language-independence that motivates *BoJac* in the first place.

C Prompts for Comment Generation

Chain-of-Comments are generated with a lightweight language-specific annotation prompt. Given a source program, the model is instructed to preserve the code exactly and to add only concise inline comments using the native comment syntax of the source language. The prompt avoids any formal tag inventory, in contrast to *BoJac*, and instead elicits free-form explanatory comments aligned with the local program structure.

C.1 Chain-of-comments

```
ROLE:
You are an expert <LANGUAGE> code annotator.

TASK:
Add concise inline comments to the following <LANGUAGE> code.

CONSTRAINTS:
- Preserve the code exactly.
- Add only inline comments.
- Do not rename, reorder, or reformat code.
- Do not add text outside the code.
- Do not use markdown fences.

INPUT:
<CODE>

OUTPUT:
Return only the commented code.
```

C.2 BoJac

BoJac annotations are generated with a constrained prompting template. Unlike *CoC*, the model is not asked to produce free-form explanatory comments, but to insert semi-formal comments from a fixed tag inventory. The prompt specifies the role of the annotator, the allowed structural tags, and strict preservation constraints: the original program must remain unchanged and the model may only add native inline comments. In our implementation, the prompt also includes a language-specific annotated example before the target program, so that the model can follow the same commenting style.

```
ROLE:
You are an expert multilingual code annotator.

TASK:
Insert semi-formal inline comments into the following
      <LANGUAGE> program.
The comments must expose language-independent
      structured-programming primitives.

TAG SET:
- Declarations:
  DEFINE_CLASS, DEFINE_METHOD, DEFINE_FUNCTION
- Data-flow and computational operations:
  DEFINE_VARIABLE, ASSIGN, CALL
- Control-flow and output behaviour:
  CONDITION, LOOP, RETURN

ANNOTATION PRINCIPLES:
- Use the most specific applicable tag.
- Comment only semantically meaningful program units.
- Prefer fewer, informative comments over redundant comments.
```

```
- Do not describe language-specific syntax unless it is
      semantically relevant.
- Do not invent behaviour that is not explicitly present in
      the code.

CONSTRAINTS:
- Preserve the original code exactly.
- Do not rename identifiers.
- Do not reorder or reformat code.
- Do not remove imports, braces, semicolons, spacing, or
      blank lines.
- Add only comments using the native comment syntax of
      <LANGUAGE>.
- Do not use markdown fences.
- Do not add explanations outside the code.

REFERENCE EXAMPLE:
<LANGUAGE-SPECIFIC BOJAC EXAMPLE>

INPUT:
<CODE>

OUTPUT:
Return only the inline-commented code.
```

C.3 Prompt for Code-to-Code Translation

We use a constrained prompt that asks the model to translate a source program into a target programming language while preserving the required target interface. The prompt specifies both the source and target languages, provides the source program, and supplies the exact target function declaration to be used in the generated solution. This constraint is important because translated programs are evaluated automatically: preserving the expected function name and argument signature ensures that the output can be executed by the benchmark harness. The model is instructed to return only valid target-language code, without markdown formatting or explanatory text.

```
ROLE:
You are an expert <TARGET_LANGUAGE> and <SOURCE_LANGUAGE>
      software engineer.

TASK:
Translate the following <SOURCE_LANGUAGE> code into clear,
      idiomatic <TARGET_LANGUAGE> code.

TARGET INTERFACE:
Use exactly this function declaration:
<FUNCTION_DECLARATION>

CONSTRAINTS:
- Do not rename the function.
- Do not rename the arguments.
- Preserve the semantics of the source program.
- Return only valid <TARGET_LANGUAGE> code.
- Do not include markdown fences.
- Do not include explanations or comments outside the code.

INPUT:
<SOURCE_LANGUAGE> code:
<SOURCE_CODE>

OUTPUT:
<TARGET_LANGUAGE> code:
```

Name Mangling. To test whether models rely on lexical cues encoded in identifiers, we construct a name-mangled version of HUMANEVAL-X. Using Tree-Sitter, we parse the equivalent programs in the benchmark and automatically replace semantically meaningful variable, function, and method names with neutral placeholders, while preserving the original program logic. In this setting, models must infer the translation from control flow, operations, and data dependencies rather than from identifier semantics. An example of name-mangled Kotlin-to-Python translation is shown in Figure 5.

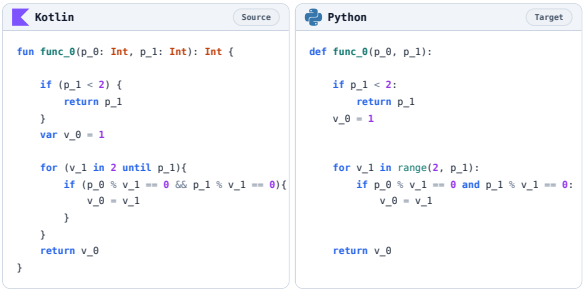

Figure 5: Name-mangled Kotlin-to-Python translation. Without semantically meaningful names, the model must infer logic from control flow, operations, and data dependencies rather than lexical cues.

D Low-Resource Language Classification

We classify source languages as comparatively lower-resource based on three forms of evidence. First, prior work shows that programming-language distribution affects code-generation and translation performance, with lower-resource languages often yielding weaker results (Orlanski et al., 2023). Related work on knowledge transfer for low-resource programming languages explicitly targets underrepresented languages such as OCaml and shows that additional validated data can improve performance (Cassano et al., 2024). Second, corpus-level statistics from The Stack v2 show substantially larger volumes for Python, JavaScript, Java, and C++ than for languages such as Go and Rust (Lozhkov et al., 2024). Third, developer-ecosystem indicators, including Stack Overflow and GitHub activity, reflect a similar imbalance.

Importantly, resource availability is not only a matter of raw code volume. LLMs may also benefit from surrounding documentation, tutorials, comments, library examples, and community discussions. Languages with smaller ecosystems may therefore be disadvantaged both in terms of code exposure and in terms of natural-language explanations of code behaviour. This is directly relevant to our setting, where we study whether auxiliary natural-language and structured-natural-language signals can help models translate programs from

comparatively less represented source languages.

E Output Processing and Evaluation

E.1 Output Filtering

Model outputs are not always directly executable. Even when instructed to return only code, models may include Markdown fences, natural-language explanations, documentation comments, or additional commentary. Before evaluation, we therefore apply a lightweight filtering step to isolate the candidate implementation. This step removes non-code material external to the translated program, while leaving the generated program logic unchanged. Example E.1 shows a representative raw output from Llama-3.2-3B on Kotlin-to-JavaScript translation. The model returns executable JavaScript, but also includes a natural-language documentation block.

```
/**
 * Checks if an array is sorted in ascending order.
 *
 * @param {number[]} arr - The input array to check.
 * @returns {boolean} True if the array is sorted, false
     otherwise.
 */
const isSorted = (arr) => {
  if (arr.length === 0) return true;

  const sortedArr = [...arr].sort((a, b) => a - b);
  let minIndex = Infinity;
  for (let i = 0; i < arr.length; i++) {
    if (i < minIndex) {
      minIndex = i;
    }
  }

  const left = arr.slice(0, minIndex);
  const right = arr.slice(minIndex);
  return JSON.stringify(left) ===
      JSON.stringify(sortedArr.slice(0, minIndex)) &&
         JSON.stringify(right) ===
      JSON.stringify(sortedArr.slice(minIndex));
}
```

For evaluation, only the executable candidate implementation is retained. Explanatory paragraphs, Markdown formatting, and other non-executable text are discarded. The filtering step is not intended to repair incorrect code, complete missing logic, or normalise the algorithmic content of the translation.

E.2 Evaluation Harness Construction

For compiled target languages CPP and Java, the filtered implementation is inserted into a minimal language-specific harness. This harness is required only for compilation and test execution; it does not encode solution logic and is not part of the program-synthesis task. It provides the boilerplate required by the target language, such as imports, includes, a `Main` class, a `main` method, and the corresponding unit tests. For Java (Example E.2), the extracted implementation is placed inside a `Main` class together with the test body and any helper methods required by the evaluation suite:

```
import java.util.*;
import java.util.stream.*;
import java.math.*;

public class Main {

    // --- TRANSLATED CODE ---
    // Candidate implementation generated by the model.
    <TRANSLATED_CODE>

    // --- ASSERT HELPERS ---
    // Helper methods required by the test suite.
    <ASSERT_HELPERS>

    public static void main(String[] args) {
        // Unit tests for the corresponding task.
        <ASSERT_MAIN_BODY>

        System.out.println("SUCCESS_MARKER");
    }
}
```

For C++ (Example E.2), the extracted implementation is combined with the required headers, helper functions, and test body inside a single executable file:

```
#include <iostream>
#include <cassert>
#include <string>
#include <vector>
#include <algorithm>
#include <cmath>
#include <map>
#include <set>
#include <unordered_map>
#include <unordered_set>
using namespace std;

// --- TRANSLATED HELPERS ---
// Candidate implementation generated by the model.
<TRANSLATED_CODE>

// --- ASSERT HELPERS ---
// Helper functions required by the test suite.
<ASSERT_HELPERS>

int main() {
    // Unit tests for the corresponding task.
    <ASSERT_MAIN_BODY>

    cout << "SUCCESS_MARKER" << endl;
    return 0;
}
```

F Models

Family	Model	Scale	Use
Qwen2.5	Qwen2.5-1.5B-Instruct	1.5B	Eval.+Fine-tuning
Gemma-2	Gemma-2-2B	2B	Eval.+Fine-tuning
Gemma-2	Gemma-2-9B	9B	Eval.
Gemma-3	Gemma-3-4B	4B	Eval.
Llama-3.2	Llama-3.2-3B-Instruct	3B	Eval.
GPT-oss	gpt-oss-20B	20B	Eval.
GPT-oss	GPT-oss-120B	120B	Annotation
GPT-5	GPT-5.1	//	Annotation

Table 5: Instruction-tuned models used for code-to-code translation evaluation. We select general-purpose LLMs across families and parameter scales to test whether auxiliary signals improve transferable program understanding beyond model-specific effects. All models were used as annotators for the *CoC_S* approach.

G Dataset

Task	Dataset	# Test Examples	Paper / Repository
Go → Python	HumanEval-X	164	(Zheng et al., 2024), repo
Go → Java	HumanEval-X	164	//, //
Go → JavaScript	HumanEval-X	164	//, //
Go → C++	HumanEval-X	164	//, //
Rust → Python	HumanEval-X	164	//, //
Rust → Java	HumanEval-X	164	//, //
Rust → JavaScript	HumanEval-X	164	//, //
Rust → C++	HumanEval-X	164	//, //
Kotlin → Python	HumanEval-X	164	//, repo
Kotlin → Java	HumanEval-X	164	//, //
Kotlin → JavaScript	HumanEval-X	164	//, //
Kotlin → C++	HumanEval-X	164	//, //
OCaml → Python	NicheTrans	308	(Yan et al., 2023), repo
OCaml → C++	NicheTrans	308	//, //
OCaml → Java	NicheTrans	308	//, //
Matlab → Python	RepoTransBench	2,294	(Wang et al., 2025), repo
Cobol → Python	HumanEval-Cobol	146	// , repo

Table 6: Code-to-code translation tasks and evaluation datasets used in our experiments.

H Fine-tuning Setup

For supervised adaptation, we sample 1K examples from XCODEEVAL (Khan et al., 2023). We remove duplicated code pairs and repeated solutions, while preserving distinct valid implementations for the same task. The same split is used across all conditions: original uncommented code-to-code, source code enriched with *CoC_T*, and source code enriched with *BoJac*. The target implementation is fixed; only the source-side representation changes.

We fine-tune Qwen-2.5-1.5B and Gemma-2-2B with LoRA in bf16 precision. Table 7 reports the main settings.

Setting	Value
Models	Qwen-2.5-1.5B, Gemma-2-2B
Precision	bf16
Epochs	3
Learning rate	2×10^{-5}
Warmup ratio	0.05
Batch size / device	2
Gradient accumulation	8
Effective batch size	64
Max sequence length	4096
Checkpoint / eval	Each epoch
Best model metric	Eval loss
DeepSpeed	ZeRO-2
Gradient clipping	1.0
LoRA r / α / dropout	16 / 32 / 0.05
LoRA targets	Attention and MLP projections

Table 7: Fine-tuning hyperparameters.

I Results Analysis

I.1 Qualitative Analysis: *BoJac* vs. *CoC*

We qualitatively compare teacher-generated Chain-of-Comments (*CoC*) with *BoJac* on name-mangled code. Free-form *CoC* tends to describe programs at the application level, relying on task-specific interpretations such as filtering strings, matching queries, or computing particular outputs. This makes the comments informative, but also sensitive to lexical cues and identifier names. Under name mangling, where such cues are removed, even a strong teacher model may infer vague, imprecise, or unsupported semantic assumptions.

BoJac, instead, guides comment generation through semi-formal scaffolds grounded in language-agnostic programming primitives, such as `DEFINE_FUNCTION`, `LOOP`, `CONDITION`, `CALL`, and `RETURN`. As a result, it captures the logical organisation of the program rather than its surface task description. Since its annotations are anchored to control-flow and data-flow patterns, *BoJac* remains more stable under identifier obfuscation and more faithful to the structure of the original program. Figure 6 illustrates this contrast on a name-mangled Go example.

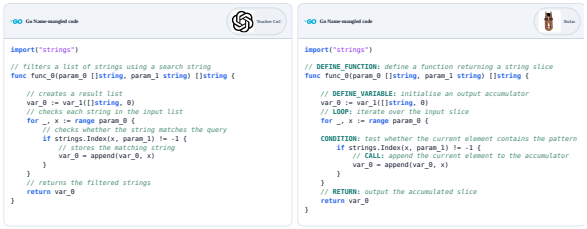

Figure 6: Name-mangled Go code annotated with Chain-of-Comments and *BoJac*. Unlike free-form comments, *BoJac* anchors annotations to structural roles, making them less dependent on lexical cues and more faithful to program logic.

Model	Base	CoC_S	CoC_T	AST	IO-Tr.	BoJac
Go →						
Qwen2.5-1.5B	60.6	57.2	64.2	55.5	59.1	**64.8**
Gemma-2-2B	63.9	64.4	67.7	63.7	65.2	**71.1**
Llama-3.2-3B	70.0	68.1	75.6	67.5	70.5	**76.8**
Gemma-3-4B	66.3	58.2	67.0	63.1	64.2	**68.8**
Gemma-2-9B	67.2	64.4	67.2	65.5	65.2	**72.3**
gpt-oss-20B	90.5	89.1	92.2	88.7	91.8	**92.8**
Rust →						
Qwen2.5-1.5B	61.5	59.3	62.4	51.5	61.6	**65.2**
Gemma-2-2B	52.5	52.2	58.3	52.9	55.9	**60.5**
Llama-3.2-3B	66.5	63.2	71.3	65.7	68.9	**74.6**
Gemma-3-4B	66.5	63.9	68.1	60.8	66.3	**68.3**
Gemma-2-9B	75.5	71.8	76.4	70.1	74.2	**78.7**
gpt-oss-20B	77.4	73.6	79.4	75.2	75.3	**80.6**
Kotlin →						
Qwen2.5-1.5B	51.6	51.1	52.1	50.1	51.2	**53.4**
Gemma-2-2B	48.3	44.8	51.6	45.9	48.8	**56.4**
Llama-3.2-3B	47.7	45.7	49.2	45.2	49.1	**54.1**
Gemma-3-4B	58.6	57.5	62.2	57.7	60.0	**63.3**
Gemma-2-9B	66.0	62.3	66.9	59.4	53.9	**71.3**
gpt-oss-20B	75.0	75.4	78.5	75.1	77.0	**79.5**

Table 8: Source-language view of code-to-code translation results. For each source language, we average Pass@1 across the four target languages: Python, Java, JavaScript, and C++. Bold indicates the best auxiliary-signal setting for each model.

I.2 Open-Model Teacher for Chain-of-Comments

To test whether the gains from teacher-generated Chain-of-Comments depend on a closed teacher model, we additionally generate *CoC_T* annotations with gpt-oss-120B on a subset of Go translation tasks and small models. Table 9 shows that external comment supervision remains more effective than self-generated comments even when the teacher is an open, less proficient model than GPT-5.1. Results are displayed in Table 9.

Go →	Python					Java					JavaScript					C++				
Model	Base	CoC_S	CoC_T*	CoC_T**	BoJac	Base	CoC_S	CoC_T*	CoC_T**	BoJac	Base	CoC_S	CoC_T*	CoC_T**	BoJac	Base	CoC_S	CoC_T*	CoC_T**	BoJac
Gemma-2-2B	58.9	67.7	69.5	69.5	**71.7**	50.6	54.2	57.9	60.1	**63.4**	88.1	88.1	86.6	90.6	**92.1**	57.9	47.5	50.6	50.6	**57.3**
Gemma-2-9B	77.4	76.2	78.1	77.4	**79.3**	35.8	32.5	40.2	34.6	**42.7**	86.5	87.5	88.1	89.2	**93.9**	68.9	61.4	65.2	67.7	**73.2**
Llama-3.2-3B	74.4	70.1	69.5	73.2	**77.8**	52.4	57.3	64.0	68.9	**69.5**	91.5	90.9	93.2	**94.5**	92.1	61.5	54.2	60.4	65.9	**67.7**

Table 9: Pass@1 on Go translation tasks for small models. *CoC_T** denotes teacher-generated Chain-of-Comments produced with gpt-oss-120B, while *CoC_T*** denotes those produced with GPT-5.1. Green and red cells indicate improvements and drops with respect to the Base setting, respectively; bold indicates the best auxiliary signal for each model–target pair. Green cells indicate improvements over *Base* setting while red indicate drops