

Identifying Latent Declarative Representations of Code for Assisting Repository Migration

Shraddha Surana^{a,*}, Ashwin Srinivasan^a and Michael Bain^b

^aDepartment of Computer Science and Information Systems, BITS Pilani, K. K. Birla Goa Campus, NH 17B, Zuarinagar, 403726, Goa, India

^bSchool of Computer Science and Engineering, UNSW Sydney, Sydney, NSW, 2052, Australia

ARTICLE INFO

Keywords:

software modernization
latent declarative representation
data-flow diagrams
code translation
program comprehension
program synthesis
Fortran

ABSTRACT

Legacy software repositories embed decades of domain knowledge in undocumented code, making understanding and modernization difficult. We treat a program as the implementation of an unobserved, declarative description of its computation and investigate whether making this *latent declarative representation* explicit improves repository-scale porting. ADFD-Migrate approximates the latent representation with an annotated data-flow diagram (ADFD) of processes, data stores, external entities, flows, and behavioral contracts. An LLM infers the source ADFD from bounded repository context, guided by static-analysis coverage checks. Dependency-aware chunking orders bounded process groups for target-language generation. Differences between the source ADFD and a statically recovered target ADFD then guide regeneration. We evaluate ADFD-Migrate on f2x50, a new benchmark of 50 Fortran repositories spanning 1.5k–1.6M lines of code and three complexity tiers, and assess the resulting ports along two dimensions: porting soundness, measured by source-oracle behavioral agreement, and porting completeness, measured by a composite migration outcome index. Against 382 curated Fortran-oracle probes, the generated Python passes 327 (85.6%), with 40 repositories passing every attempted probe. ADFD-Migrate exposes all 382 planned behaviors as runnable targets, compared with 99 and 98 for direct and repository-context translation and 69 and 30 for the static-profile and dependency-chunking ablations. It also achieves a 93.1% mean migration outcome index and a 17–59 percentage-point outcome-index advantage over direct translation on 47 repositories. These results suggest that an inspectable semantic bottleneck can improve the coverage and integration of repository-scale migration while enabling lower-cost generation for many repositories.

Glossary

ADFD Annotated data-flow diagram: the intermediate representation of processes, data stores, external entities, flows, and process contracts.

Behavioral probe A source-oracle test that compares Fortran and generated Python behavior on the same input.

Latent declarative representation The unobserved, language-independent specification of the computation implemented by a program. An inferred ADFD is an explicit, inspectable approximation of this representation.

Migration outcome index The composite measure of porting completeness, combining the generated migration’s semantic alignment, executability, code quality, and implementation rate.

SCC Strongly connected component in the ADFD process-dependency graph.

1. Introduction

Legacy software systems, often written in languages such as Fortran, COBOL, or early C, represent decades

of accumulated domain knowledge embedded in code von Mayrhauser and Vans (1995). When the original developers are unavailable and documentation is sparse, understanding these systems becomes a major challenge for software maintenance and modernization efforts Comella-Dorda, Wallnau, Seacord and Robert (2000). Scientific computing is particularly affected: critical numerical libraries (LAPACK, MINPACK, WRF) remain in active use but are locked in Fortran, limiting integration with modern data science ecosystems built around Python.

Recent advances in large language models (LLMs) have demonstrated impressive capabilities in code understanding and generation Chen, Tworek, Jun, Yuan, Pinto, Kaplan et al. (2021); Rozière, Gehring, Gloeckle, Sootla, Gat, Tan et al. (2023). However, direct LLM-based translation of legacy code, i.e., asking the model to translate an entire repository directly, suffers from several limitations: (1) loss of cross-module structure, (2) hallucinated APIs and dependencies, (3) no mechanism for detecting or correcting errors, and (4) no intermediate representation that a human can inspect and validate Pan, Ibrahimzade, Krishna, Sankar, Wassi, Merler et al. (2024).

We start from the premise that a program is one procedural implementation of a more declarative description of its computation. That description is *latent*: it is not directly observed as a separate artifact in the repository. We use an Annotated Data Flow Diagram (ADFD) as an explicit hypothesis about this latent declarative representation. The

*Corresponding author

 p20220031@goa.bits-pilani.ac.in (S. Surana);

ashwin@goa.bits-pilani.ac.in (A. Srinivasan); m.bain@unsw.edu.au (M. Bain)

ORCID(s): 0000-0002-3009-3178 (S. Surana)

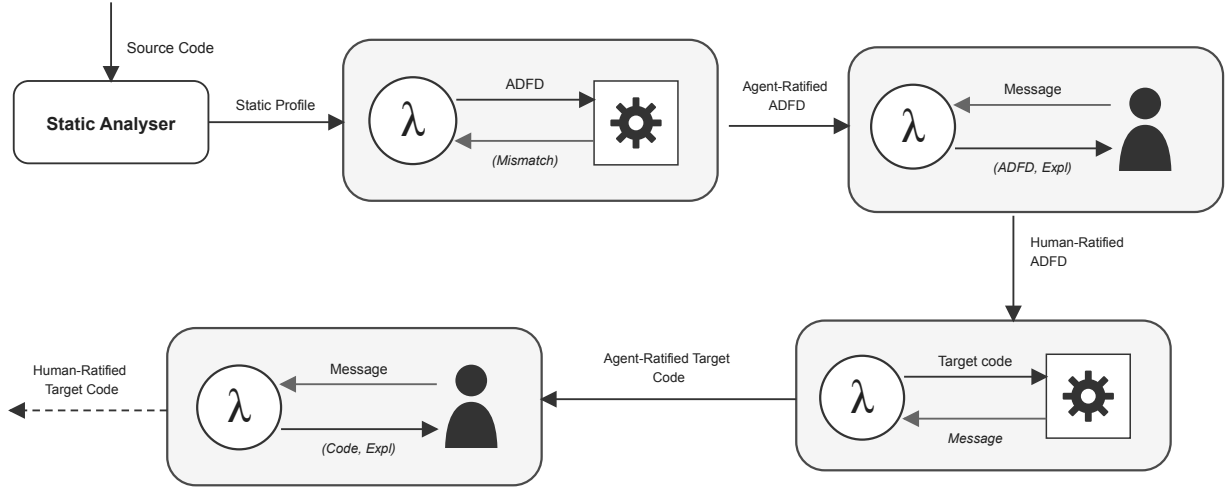


Figure 1: ADFD-mediated porting workflow. The λ and gear symbols denote LLM and deterministic agents. The first diagnostic loop refines a source ADFD; after optional human ratification, the second refines target code. Experiments automate both loops; human exchanges indicate where experts can optionally inspect and correct the intermediate ADFD and target code.

ADFD records what the computation must do through processes, data stores, external entities, flows, and behavioral contracts, without committing to the source language’s implementation details.

Figure 1 shows the ADFD-mediated porting workflow. Static analysis first produces a static source profile. An LLM proposes an ADFD, and a deterministic agent returns mismatches until the ADFD is agent-ratified; a human expert may then correct it before ratification. The ratified ADFD becomes the specification for target-code generation, followed by an analogous diagnostic loop and expert review. Thus, the explicit latent representation is checked before being reused to construct the target.

ADFD-Migrate operationalizes the machine-facing parts of this workflow: static-analysis coverage supplies the mismatch signal during source-ADFD construction, while dependency-aware generation and source–target ADFD comparison provide the target-side feedback loop. The experiments run the two LLM–software-agent interactions automatically, so the human-ratification interactions are inspectable extension points rather than an evaluated treatment. Appendix A maps this workflow to the complete automated system architecture.

A central contribution is the *dependency-aware chunking* strategy used to organize target-code generation. Tarjan’s SCC detection Tarjan (1972) and Kahn’s topological sorting Kahn (1962) partition large ADFDs into bounded, dependency-ordered process groups. Each group is generated with upstream interface summaries, keeping calls tractable while preserving cross-module references.

By reducing per-call complexity, chunking makes smaller open-source models usable for the computationally intensive generation. Low- and medium-tier generation ran on Qwen-3-Coder (30B locally via Ollama, 480B cloud-hosted) and

incurred no metered per-call charges; complex-tier generation and the evaluation-only judge used Anthropic models. Appendix D.7 states the cost-accounting boundary and measured charges.

This paper makes the following contributions:

1. An ADFD-mediated migration framework that treats the declarative specification of the computation implemented by a legacy repository as a latent representation and makes an explicit, inspectable approximation of it as an ADFD for cross-language code porting with an iterative source ADFD construction loop driven by static coverage scoring (Section 3.1.1). This is distinct from call-graph decomposition Ibrahimzada, Ke, Pawagi, Abid, Pan, Sinha et al. (2025), intermediate programming languages Macedo, Tian, Nie, Cogo and Adams (2025), and compiler IRs Szafraniec, Roziere, Leather, Char-ton, Labatut and Synnaeve (2023).
2. A *dependency-aware chunking* algorithm based on Tarjan’s SCC detection and topological sorting that partitions ADFD process *dependency graphs* into bounded, topologically ordered groups while preserving referential integrity (Section 3.1.2).
3. f2x50, a curated benchmark of 50 open-source Fortran repositories across three complexity tiers with converted Python repositories, intermediate ADFDs, and evaluation artifacts (Section 4.1).
4. A correctness-first empirical evaluation across all 50 repositories, comparing ADFD-Migrate against four non-ADFD baselines and ablations: direct file-by-file translation, repository-context direct translation, static-profile ablation, and dependency-chunking ablation. ADFD-mediated decomposition reaches 85.6% behavioral agreement over all planned probes against

Table 1

Comparison of Fortran migration approaches.

Property	f2py	fable	LFortran	Ours
Target language	Py (bind)	C++	Py (AST)	Python
Readable output	✗	Partial	✗	✓
Idiomatic target	✗	✗	✗	✓
Preserves structure	N/A	✗	✗	✓
Inspectable IR	✗	✗	LLVM IR	ADFD
Human checkpoint	✗	✗	✗	✓

21.7% for both direct and repository-context direct translation and 11.0% and 3.7% for the two ablations, and a +17 to +59 pp outcome-index advantage, with the largest gains on complex repositories and with reduced or eliminated API cost (Section 5).

Section 2 positions the approach against existing migration tools and LLM translation methods. Section 3 defines the ADFD and the porting procedures, Section 4 states the hypothesis and experimental design, Section 5 reports results along the two dimensions of porting effectiveness, and Section 6 discusses the method and threats to validity.

2. Background and Related Work

Legacy Code Modernization Table 1 positions our approach relative to existing Fortran migration tools. `f2py` Peterson (2009) generates Python bindings that call compiled Fortran via C extensions. It does not produce readable Python. `fable` Grosse-Kunstleve, Terwilliger, Sauter and Adams (2012) performs rule-based Fortran-to-C++ translation but the output retains Fortran idioms requiring manual post-editing. `LFortran` LFortran Contributors (2023) compiles to LLVM IR with an experimental AST-to-Python transpiler limited to a subset of Fortran 2018. Our approach translates through a *semantic* intermediate representation that captures *what* the code computes and not simply *how* it is expressed.

LLM-Based Code Translation TransCoder Rozière, Lachaux, Chanussot and Lample (2020) uses unsupervised machine translation between programming languages.

General-purpose LLMs show strong performance on code translation benchmarks Yang, Liu, Yu, Keung, Li, Liu et al. (2024b); Yin, Ni, Nguyen, Wang and Yang (2024) but typically operate at the file or function level. Code Distillation (CoDist) Huang, Qi, Yao, Wang, Gu, Clement et al. (2023) uses a language-agnostic distilled-code pivot, improving over TransCoder-ST by 12.7% on average in snippet benchmarks. Unlike our ADFD, it creates parallel training corpora without addressing repository-wide dependency ordering, interface propagation, or human-inspectable checkpoints. Recent work on long code blocks shows that context management remains central for large translation tasks Chakaravarthy, Roy Choudhury, Kanvar, Katan, Pandian, Raghuvanshi et al. (2026). Saha et al. Saha, Rabbi, Pham, Wang and Yang (2026) investigate natural language

specifications as an intermediate representation across five languages, finding inconsistent gains alone but improvements for some language pairs when combined with source code. Our work instead studies how an explicit ADFD representation can organize repository-scale migration, where dependency structure, interfaces, import paths, and generated artifacts must remain coherent across a repository.

Repository-level translation This is an active frontier. AlphaTrans Ibrahimzada et al. (2025) decomposes Java repositories into program fragments using static analysis, translates them in reverse call order, and validates translated fragments using the source project’s tests. It reports 96.40% syntactically correct fragments and validates run-time behavior and functional correctness for 27.03% and 25.14% of application-method fragments. These results are not directly comparable: AlphaTrans studies ten Java-to-Python projects using project test suites, whereas we use uniform source-oracle probes for Fortran-to-Python. No shared benchmark supports a controlled head-to-head test. InterTrans Macedo et al. (2025) routes through intermediate programming languages to improve code-translation accuracy. CodePlan Bairi, Sonwane, Kanade, C., Iyer, Parthasarathy et al. (2024) uses planning over dependency graphs. SemanticForge Zhang, Zhang, Luo, Ma, Yuan, Gu et al. (2025) constructs knowledge graphs for repository-level generation.

Our approach differs in two ways: (1) we translate through a *semantic* intermediate representation that captures computational intent rather than syntactic structure, and (2) the ADFD is designed for human inspectability. AlphaTrans decomposes by call graph (a syntactic artifact); we decompose by *data flow* (a semantic artifact).

Data Flow Diagrams (DFDs) in Software Engineering DFDs have a long history in structured systems analysis DeMarco (1979); Yourdon (1989). Recent work has revived interest in DFDs as intermediate representations for LLM-driven code generation. Prior work on interactive program synthesis Surana, Srinivasan and Bain (2026) uses DFDs as specifications for human-in-the-loop validation. The present work extends this from interactive code *construction* to automated code *porting*, using the DFD as a language-agnostic intermediate representation rather than just a specification to be implemented.

LLM-Based Coding Agents Autonomous coding agents such as SWE-Agent Yang, Jimenez, Wettig, Lieret, Yao, Narasimhan et al. (2024a), Devin Cognition Labs (2024), AutoCodeRover Zhang, Ruan, Fan and Roychoudhury (2024) and OpenHands Wang, Li, Song, Xu, Tang, Zhuge et al. (2025) couple LLMs with tool use for software engineering tasks. AgenticTyper Pohle (2026) automates JavaScript-to-TypeScript modernization; FreshBrew May, Misra, Luo, Sridhar, Gehring, Ribeiro et al. (2026) benchmarks agents on Java 8-to-17 migration.

For repository-scale *cross-language* porting, these agents usually depend on frontier or task-specific models (AgenticTyper degrades with smaller models), expose no inspectable intermediate artifact, and require replaying long trajectories to debug failures. ADFD-Migrate instead uses bounded chunks that localize errors, support a locally hosted 30B model for many repositories, and let experts inspect the ADFD before generation.

3. Our Approach

We distinguish the latent declarative representation from the explicit artifact used to approximate it. A repository is an observed implementation, whereas its declarative specification of what is computed is unobserved. The source ADFD is an inferred, explicit approximation of that latent declarative representation. It allows different implementations to be mediated by a shared representation while retaining the software-engineering benefits of a declarative DFD: language independence, visual inspectability, and composition from guarded processes.

The ADFD also serves as an *explicit semantic bottleneck* because target-code generation is conditioned on the bounded ADFD rather than directly on the source repository. Information needed for generation is organized into processes, data stores, entities, flows, and behavioral contracts. The resulting artifact can be checked against static analysis, reviewed by a developer, partitioned for repository-scale generation, and compared across source and target languages. We assume ADFDs are sufficiently expressive to approximate the computational structure needed for migration, and specify the approach using two definitions.

Definition 1 (Annotated Data Flow Diagram). An ADFD is a tuple (P, D, E, F, σ) where P is a set of *processes*, D a set of *data stores*, E a set of *external entities*, $F \subseteq (P \cup D \cup E) \times (P \cup D \cup E)$ a set of directed *flows* (i.e., ordered pairs denoting who sends data to whom), and $\sigma : P \rightarrow (\text{Spec}, \text{Pre}, \text{Post}, \text{In}, \text{Out})$ assigns each process a specification, pre/post-conditions, and typed I/O. We denote the set of all possible ADFDs as $\mathcal{A}$.

Definition 2 (ADFD-Mediated Porting of Repositories). Let S denote a source implementation language and T a target implementation language. Let $\mathcal{R}_S$ and $\mathcal{R}_T$ denote the set of all possible repositories in S and T respectively. We use an ADFD as an explicit approximation of a repository's latent declarative representation, and let $\mathcal{A}$ denote the space of ADFD representations. Let $\text{Construct}_S : \mathcal{R}_S \rightarrow \mathcal{A}$ and $\text{Construct}_T : \mathcal{R}_T \rightarrow \mathcal{A}$ denote functions that construct ADFDs from source- and target-language repositories, respectively, and let $\text{Generate}_T : \mathcal{A} \rightarrow \mathcal{R}_T$ denote a function that generates a target repository from an ADFD. The porting function is $\text{Port}_{S \rightarrow T} = \text{Generate}_T \circ \text{Construct}_S$. After generation, source-target ADFD alignment compares $A_S = \text{Construct}_S(R_S)$ with $A_T = \text{Construct}_T(R_T)$. This separate operation evaluates and refines the generated repository.

Procedure 1 CONSTRUCTADFD

```

1: Input:  $R_S$ : a source repository;  $\lambda$ : an LLM;  $k$ : a bound
   on iteration;  $\tau$ : a coverage threshold
2: Output:  $A$ : an Annotated DFD
3:  $\Sigma \leftarrow \text{STATICANALYSIS}(R_S)$ 
4:  $A_{\text{best}} \leftarrow \emptyset$ ;  $s_{\text{best}} \leftarrow 0$ ;  $dir \leftarrow \emptyset$ 
5: for  $i = 1$  to  $k$  do
6:    $A_i \leftarrow \text{INFERADFD}(R_S, \lambda, \Sigma, dir)$ 
7:    $(s_i, gaps_i) \leftarrow \text{COVERAGESCORE}(\Sigma, A_i)$ 
8:   if  $s_i > s_{\text{best}}$  then
9:      $A_{\text{best}} \leftarrow A_i$ ;  $s_{\text{best}} \leftarrow s_i$ 
10:  end if
11:  if  $s_{\text{best}} \geq \tau$  then
12:    break
13:  end if
14:   $dir \leftarrow \text{GAPDIRECTIVES}(gaps_i)$ 
15: end for
16: return  $A_{\text{best}}$ 

```

3.1. Implementation

We implement Construct_S , Construct_T , and Generate_T with ADFDs as the intermediate representation, but the two construction functions are not identical. The source ADFD is inferred by an LLM and checked against a Fortran static-analysis profile. The target ADFD is extracted by Python static analysis, with optional LLM enrichment of flow labels. Target-code refinement then uses structural alignment between the source and target ADFDs. A separate LLM judge assesses semantic alignment only for evaluation.

3.1.1. Constructing and Checking the Source ADFD

Procedure 1 constructs a source ADFD through iterative refinement of the LLM's output. Static analysis of the source code acts as a *repository profile*, containing module structure, call graphs, function/subroutine signatures, and file-to-definition mappings. The construction procedure uses this repository profile together with relevant source-code excerpts to generate an ADFD for the repository. "Goodness" is computed using a *static coverage scoring* function that compares the ADFD against the profile to identify *gaps* (source files and definitions not represented in the ADFD). Targeted directives describing the gaps are then injected into a prompt for the next invocation of the LLM.

STATICANALYSIS parses the source repository to produce a repository profile Σ containing file-level definitions, module structure, call graphs, function/subroutine signatures, I/O contracts, and file-to-definition mappings. The complete profile is retained for coverage scoring, but is not sent wholesale to the LLM. Files are ranked by static-analysis importance, and the prompt receives a bounded projection of the profile plus excerpts from the highest-ranked source files. For large repositories, ranked files are grouped into at most six call-connected module families; each family is processed with its own projected profile and cross-family interface edges, and the resulting ADFD fragments are merged. The default prompt budget is 120,000

characters. If a projected prompt exceeds its share of this budget, call edges, procedure and I/O contracts, and source excerpts are progressively reduced; source excerpts can be removed while a reduced structural profile is retained. Subsequent coverage loops identify omitted files and definitions against the complete profile and inject targeted directives. INFERADFD prompts λ with this bounded context and any gap directives (*dir*), returning candidate A_i . COVERAGESCORE compares it with the profile and returns a scalar score $s_i \in [0, 1]$ together with a gap set $gaps_i$ listing uncovered source files and definitions. The scalar score is file coverage; uncovered definitions are retained in $gaps_i$ to generate refinement directives.

GAPDIRECTIVES converts the gap set into targeted instructions (e.g., “*The following source files are not represented: [list]. Revise the ADFD to include these.*”), thus preserving the LLM’s freedom to restructure while ensuring omissions are reported. In practice, the first pass usually maps 70–85% of source files, and the second pass typically exceeds the (configurable) 80% file-coverage threshold. Repositories with deeply nested hierarchies, such as M_strings and xtb, may require further refinement loops.

3.1.2. Target-Code Generation and Source–Target ADFD Alignment

Procedure 2 generates target code chunk by chunk in topological order and then invokes a separate source–target ADFD alignment step. The generator LLM λ_g produces code; the evaluation-only judge is not part of this procedure. For each chunk C_j , BUILDCHUNKDFD extracts a *sub-ADFD* from the full ADFD A_S . It needs the complete ADFD because the chunk’s processes may reference data stores and external entities defined globally, and flows may connect to elements outside the chunk. UPSTREAMINTERFACES derives lightweight interface summaries *ifc* from the accumulated context *ctx*, which holds the public signatures of all modules generated by prior chunks. LLMGENERATE receives the sub-ADFD, interface summaries, and target language, and generates code for the chunk. Finally, SIGNATURES parses the output to extract public module signatures appended to *ctx* so that subsequent chunks can reference them correctly.

After each complete pass through all chunks, EXTRACT-TARGETADFD applies Python static analysis to the generated repository to obtain A_T . The function ALIGNDIFF compares A_S and A_T to identify structural differences, including missing and additional processes, flows, and data stores. This comparison is deterministic: it normalizes element names, matches source and target processes and data stores by exact, containment, or token-overlap rules, and compares flows by their normalized endpoint pairs. It also computes count-based coverage for processes, flows, and data stores. The resulting difference set *diff* supplies structural feedback to LLMGENERATE when each chunk is regenerated. The loop terminates when no structural differences remain or when the iteration bound is reached. Appendix C.3 gives the normalization and matching rules in full. Separately,

Procedure 2 GENERATETARGET

```

1: Input:  $T$ : the target language;  $A_S$ : the source ADFD;
    $\lambda_g$ : a generator LLM;  $c_{\max}$ : a bound on processes per
   chunk;  $m$ : a bound on refinement iterations
2: Output:  $R_T$ : a target repository
3:  $C \leftarrow \text{DEPENDENCYCHUNK}(A_S, c_{\max})$ 
4:  $diff \leftarrow \emptyset$ 
5: for  $i = 1$  to  $m$  do
6:    $R_T \leftarrow \emptyset$ ;  $ctx \leftarrow \emptyset$ 
7:   for each chunk  $C_j$  in  $C$  do
8:      $sub \leftarrow \text{BUILDCHUNKDFD}(C_j, A_S)$ 
9:      $ifc \leftarrow \text{UPSTREAMINTERFACES}(ctx)$ 
10:     $code_j \leftarrow \text{LLMGENERATE}(\lambda_g, sub, ifc, T, diff)$ 
11:     $R_T \leftarrow R_T \cup code_j$ 
12:     $ctx \leftarrow ctx \cup \text{SIGNATURES}(code_j)$ 
13:   end for
14:    $\Sigma_T \leftarrow \text{STATICANALYSIS}(R_T)$ 
15:    $A_T \leftarrow \text{EXTRACTTARGETADFD}(R_T, \lambda_g, \Sigma_T, \emptyset)$ 
16:    $diff \leftarrow \text{ALIGNDIFF}(A_S, A_T)$ 
17:   if  $diff = \emptyset$  then
18:     break
19:   end if
20: end for
21: return  $R_T$ 

```

an evaluation-only LLM judge assesses semantic alignment between A_S and A_T for evaluation (Appendix C.7); its score is not used in the refinement loop.

The principal difficulty for target-code generation is scale: real-world ADFDs can contain dozens of interconnected processes. Providing the full ADFD to an LLM as a single prompt usually exceeds the context window and degrades quality Liu, Lin, Hewitt, Paranjape, Bevilacqua, Petroni et al. (2024). We address this with a dependency-aware chunking procedure DEPENDENCYCHUNK that partitions the ADFD process graph into bounded, topologically ordered groups. Note that this is fundamentally different from the text-based chunking used in Retrieval-Augmented Generation systems, which splits documents into fixed-size fragments for retrieval; our chunking operates on a directed process dependency graph using graph-theoretic algorithms to preserve referential integrity and generation order. Given an ADFD and a maximum number of processes per chunk $c_{\max}$, DEPENDENCYCHUNK is a multi-stage process, summarized below.

Stage 1. Dependency Graph Construction.

Extract the process-to-process dependency graph from the ADFD.

Stage 2. SCC Detection. Apply Tarjan’s algorithm Tarjan (1972) (iterative variant for large graphs) to find all strongly connected components in the graph from Stage 1.

Stage 3. Topological Sort. Replace each SCC with a single *super-node* that represents all the processes it

Procedure 3 PORT

- 1: **Input:** T : the target language; R_S : the source repository; λ_g : the generator LLM; $k, \tau, c_{\max}, m$
- 2: **Output:** R_T : target repository
- 3: $A_S \leftarrow \text{CONSTRUCTADFD}(R_S, \lambda_g, k, \tau)$
- 4: $R_T \leftarrow \text{GENERATETARGET}(T, A_S, \lambda_g, c_{\max}, m)$
- 5: **return** R_T

contains, collapsing the mutual dependencies within the SCC. The result is now a DAG. We compute a topological ordering of the DAG with Kahn’s algorithm Kahn (1962), guaranteeing that when code is generated for a chunk C , every chunk it depends on has already been processed.

Stage 4. Greedy Bin-Packing. The topologically sorted sequence of SCCs is packed into chunks using a greedy strategy, i.e., iterate through the SCCs in topological order. When the number of processes in a chunk exceeds the bound $c_{\max}$ a new chunk is started. Because SCCs are atomic units (never split), a single SCC whose process count exceeds $c_{\max}$ simply becomes an oversized chunk on its own, thus preserving dependency integrity at the cost of a larger LLM prompt.

The total chunking complexity is $O(n+e)$, where $n = |P|$ and e is the number of direct process-to-process flows in the ADFD dependency graph.

Once we have the implementations of CONSTRUCTADFD and GENERATETARGET, the porting function is straightforward (see Procedure 3).

As part of GENERATETARGET, the source ADFD is compared with the ADFD extracted from the generated target repository, as described above. Appendix Figure B.1 presents the two partial ADFDs side-by-side for a MINPACK example, providing a visual illustration of the comparison.

4. Empirical Evaluation

The principal goal of the experiment is to test the hypothesis:

Using an LLM to generate an ADFD as a latent representation of code; and then using the LLM to generate the actual code using the ADFD as context results in more effective porting of code repositories than direct LLM-based migration of repositories.

We test this hypothesis on repositories of varying sizes ranging from very simple to very complex, and measure effectiveness along several dimensions that broadly fall into two categories:

1. **Porting soundness.** These measure the behavioral agreement between the source and target repositories on identical inputs.

Table 2

f2x50: 50 Fortran repository benchmark.

Tier	n	LoC Range	Files	Examples
Low	20	1.5k–41k	4–112	ABAQUS, fftpack, tsunami
Medium	20	4.8k–192k	9–405	minpack, stdlib, xtb
Complex	10	29k–1.6M	158–3.6k	hdf5, lapack, pymc2

2. **Porting completeness.** These measure the extent to which source repositories are ported completely into executable and maintainable target repositories.

4.1. Materials

f2x50: Benchmark Corpus To the best of our knowledge, no standard benchmark exists for porting full Fortran repositories to Python. Existing benchmarks such as CodeXGLUE Lu, Guo, Ren, Huang, Svyatkovskiy, Blanco et al. (2021) and HumanEval-X Zheng, Xia, Zou, Dong, Wang, Xue et al. (2023) evaluate task-level or function-level code understanding and generation rather than repository-scale Fortran porting, and recent repository-level benchmarks target Java Ibrahimzada et al. (2025); May et al. (2026). We fill this gap with f2x50, a benchmark designed specifically for repository-scale Fortran-to-Python migration. Table 2 summarizes its three complexity tiers. It comprises 50 curated open-source Fortran repositories from GitHub, selected for diversity in domain, coding style, and structural complexity.

Repositories span diverse scientific domains: numerical methods (minpack, quadpack, SISSO), computational fluid dynamics (CaNS, Incompact3d, Nek5000, openfast), molecular dynamics and quantum chemistry (dftd4, xtb, crest), astrodynamics (Fortran-Astrodynamic-Toolkit), string and time processing (M_strings, M_time, datetime-fortran), wavelet analysis (wavelets), testing frameworks (test-drive), weather modeling (WPS, ccpp-physics), finite element analysis (elmerfem), high-performance I/O (hdf5), linear algebra (lapack, arpack-ng), Bayesian statistics (pymc2), package management (fpm), and neural networks (FKB, neural-fortran, fastGPT).

Selection criteria: (1) publicly available on GitHub under an open-source license; (2) compilable with a standard Fortran compiler; (3) more than 1,000 LoC; and (4) no public Python translation available, ensuring that no LLM could have been trained on a Python version of these repositories, making them authentic test cases. Repositories were stratified by structural complexity considering inter-module coupling, use of advanced Fortran features, and codebase scale. Because stratification weighs coupling and language features as well as size, the LoC ranges of adjacent tiers overlap (Table 2). The benchmark is publicly available at <https://github.com/ShraddhaSurana/iProg/tree/main/f2x50>.

4.2. Method

The steps below were used to test the hypothesis. The ADFD construction, generation, and alignment steps are the ones specified in Section 3. Table 3 lists the settings used.

For each repository class (low, medium, complex):

Fix the generator LLM per repository (Table 3).

For each repository in that class:

1. Curate the behavioral probes from the Fortran source.
2. Construct an ADFD using the construction method of Section 3.1.1.
3. Port the code to Python using the ADFD (Section 3.1.2, Procedure 2).
4. Extract an ADFD from the ported code and compare it with the source ADFD using the alignment process of Section 3.1.2. Where the two differ, regenerate the code with the differences as feedback and compare again. The ADFDs are *aligned* when the comparison reports no missing or additional processes, flows, or data stores.
5. Construct the ported code *without* an ADFD, once for each baseline or ablation run on that repository.
6. Run the fixed probes against every port and against the same Fortran oracles, and compute the migration outcome index for every port; record both.

Some additional details are relevant. Four design choices in these steps control what a difference in outcome can be attributed to. These are:

(i) *Generator assignment*. The generator is assigned by what a model could handle, not by tier label: the local 30B model ported all 20 low-tier repositories and the 4 medium-tier ones it completed without excessive stubbing or timeouts, the 480B model the other 16 medium-tier repositories, and Claude Sonnet 4.5 the 10 complex-tier ones. Whichever generator a repository was assigned produced both its ADFD-Migrate port and all of its baseline ports, so a difference in outcome for that repository is attributable to the method rather than the model.

(ii) *Probe provenance*. Probes are curated from the *source* side only: public, deterministic Fortran routines callable on small self-contained inputs, with expected values taken from the original Fortran compiled with `gfortran` or wrapped with `f2py`. They are fixed before any port is produced, so the same probe set is applied to every port of a repository.

(iii) *Baselines and ablations (step 5)*. The two baselines are direct file-by-file translation and repository-context direct translation; the two ablations are the static-profile ablation (no ADFD) and the dependency-chunking ablation (no ADFD). All four cover the same 47 repositories: all low- and medium-tier repositories plus 7 complex-tier repositories (`pymc2`, `c_cpp-physics`, `Nek5000`, `petsc`, `hdf5`, `fpm`, and `openfast`). The remaining three largest complex-tier repositories – `lapack`, `cp2k`, and `e1merfem` – could not be tried across all four comparisons due to cost (e.g., running repository-context direct translation on `lapack` alone cost > \$300). Together these comparisons separate the contribution of the declarative representation from that of context and dependency ordering alone.

Table 3

Experimental settings.

Setting	Value
Generator (low)	Qwen-3-Coder 30B (local)
Generator (medium)	Qwen-3-Coder 30B (4), 480B (16)
Generator (complex)	Claude Sonnet 4.5 (API)
Judge model	Claude Opus 4.6 (evaluation only)
Temperature	0.3 (generation), 0.2 (judge)
ADFD iterations k	3–5
Coverage threshold τ	0.80
Chunk size $c_{\max}$	3 processes
Refinement iterations m	3–5

(iv) *Measures (step 6)*. The two measures are defined in Section 4.2.1: behavioral agreement is oracle-based and uses the same probes, oracles, and inputs for every method, whereas the outcome index is not oracle-based and is computed from each port directly.

Appendix C gives the remaining details needed to replicate the experiment: model configurations and timeouts, the prompts for ADFD inference, target generation, and each baseline, the computation of the structural and semantic alignment scores, the probe inputs and tolerances, and the outcome-index formula; Appendix F reports representative probe outputs and the computational environment. No developer inspected or ratified an ADFD in any run reported here; every port was produced end-to-end without human intervention (Appendix D.8).

4.2.1. Evaluation

Step 6 records two measures, corresponding to the two dimensions of effectiveness stated above.

Porting soundness: behavioral agreement. For each probe, the harness executes the Fortran oracle and the corresponding generated Python on identical inputs and compares the outputs by numeric tolerance, bit-exact array, or exact scalar comparison. A probe *passes* if the outputs agree, and *fails* otherwise. A probe also fails if no corresponding Python behavior can be located, imported, and invoked: failing to produce runnable target code for a selected source behavior is a failure of the porting method. Behavioral agreement is therefore reported over all *planned* probes. Failures of this second kind are additionally tallied as *missing functionality*, because they diagnose a different defect (absent or unusable generated code) from a wrong numerical result, and a method’s runnable-only rate is reported alongside as a diagnostic.

Porting completeness: migration outcome index. The index combines ADFD semantic alignment (LLM-judged), executability (syntax validity and import success), code quality (docstring and type-hint coverage), and implementation rate (the fraction of generated functions containing real logic rather than stubs). It measures whether a method yields a complete, importable, maintainable migration and

is *not* a correctness percentage; correctness claims rest on behavioral agreement. Appendix C.7 gives the formula and component definitions.

5. Results

The hypothesis is supported on both dimensions defined in Section 4: the ADFD-mediated ports reproduce the source repositories' behavior on nearly four times as many of the planned probes as direct translation. We report the two dimensions in turn, with the relevant baselines inside each comparison. Supporting diagnostics follow; their detailed tables are in Appendix D, and repository-level scores in Appendix E.

5.1. Porting Soundness: Behavioral Agreement

ADFD-Migrate against the Fortran oracles: The executable Fortran-vs-Python harness planned 382 behavioral probes across all 50 repositories, with at least four probes per repository. Every planned probe was runnable against the ADFD-Migrate ports: 327 passed and 55 failed, with no missing functionality, no execution errors, and no skipped checks, for a behavioral agreement rate of 85.6%; 40 of 50 repositories passed every probe. The checks are targeted probes rather than exhaustive system tests, but they are the direct correctness evidence in this study: each check compares source and generated behavior on the same inputs. The tier breakdown is in Appendix D.1 and the per-mode counts in Appendix C.4; the probes cover numerical, bit-exact, scalar/string, boolean, and duration-normalized comparisons.

ADFD-Migrate against direct translation, same probes: To test whether the intermediate ADFD representation improves porting, we ran the same 382 curated behavioral probes against the two baselines and the two ablations, using the same Fortran oracles and the same input vectors. The verifier does not require a comparison method to use the same file layout as ADFD-Migrate: for each probe, it searches the method's generated *src/* tree by routine name, so a routine is still tested if the method relocates it or places it in a different module. The results are summarized in Table 4.

Counting every planned probe, ADFD-Migrate agrees with the Fortran oracles on 85.6% of the 382 probes, against 21.7% for both direct file-by-file translation (83/382) and repository-context direct translation (83/382), a factor of 3.9, and against 11.0% for the static-profile ablation (42/382) and 3.7% for the dependency-chunking ablation (14/382). The gap is driven by runnability: all 382 probes are runnable against ADFD-Migrate, but only 99 and 98 respectively against the two baselines, and only 69 and 30 respectively against the two ablations. The remaining probes fail as *missing functionality*, which can arise when parts of a large repository were not generated because of input size constraints, the relevant routine or wrapper is absent, the generated file has an import or syntax defect, or the located routine has an incompatible calling convention.

The last two columns of Table 4 show where the advantage comes from. Restricted to the probes a method

could actually run, the two baselines remain comparable to ADFD-Migrate: 85.6%, 83.8%, and 84.7%, with some probes passing only for a baseline and some only for ADFD-Migrate. So when a baseline does produce runnable code for a behavior, that code is about as likely to be right. The two ablations trail on this measure (60.9% and 46.7%), indicating that, unlike the baselines, the routines they do manage to run are also less often correct. What separates all four comparisons from ADFD-Migrate is primarily how often they produce runnable code at all, and this is why the over-all-382 rates diverge so sharply: their high (or, for the ablations, middling) runnable-only rates are computed over a much smaller share of the intended behaviors – roughly a quarter for the baselines and a fifth or less for the ablations. Because a port that omits a behavior, or produces code that cannot be imported or called, has not delivered that behavior to whoever needs the migration, we treat those probes as failures. The agreement rate over all 382 planned probes is therefore the one we report, and on this measure the hypothesis is supported.

5.2. Porting Completeness: Migration Outcome Index

Table 5 reports the migration outcome index together with its principal components and the paired direct-translation baseline. ADFD-Migrate averages 93.1% across all 50 repositories and exceeds direct translation in every tier by 16.9–58.5 percentage points. The gap grows with repository complexity, which is consistent with the claim that an explicit approximation of the latent declarative representation is most useful when file-wise translation loses repository-wide structure.

The stronger repository-context direct baseline improves the plain direct mean from 0.654 to 0.709, but remains 21.5 percentage points below ADFD-Migrate and loses on 46 of 47 pairs. The static-profile ablation and the dependency-chunking ablation also trail ADFD-Migrate on all 47 available pairs each. These comparisons separate the ADFD contribution from simply supplying more repository context, static metadata, or dependency groups. Full baseline and ablation distributions, representative cases, paired confidence intervals, and tests are reported in Appendix D.2.

5.3. Supporting Analysis: Mechanism and Practicality

The supporting analyses clarify what the representation does and where it remains insufficient. Source ADFDs were constructed for all 50 repositories and partitioned into 1–42 dependency-ordered chunks. Near-perfect syntax validity indicates that bounded generation is reliable, while import success (72.5–79.8% by tier) remains the main integration weakness. A conservative deterministic reconciliation pass raised mean import success from 79.7% to 82.3% (Appendix D.3).

Representation-level alignment is useful but not equivalent to correctness. High outcome-index repositories can still contain formula-level, formatting, numerical-constant,

Table 4

Cross-method behavioral comparison on the same 382 planned Fortran-vs-Python probes. *Runnable*: probes for which the generated Python target could be located, imported, invoked, and compared with the Fortran oracle. A probe fails either because the generated code returned a wrong output or because no runnable target existed for it at all (*missing functionality*: missing generated target, non-generated repository, import/syntax defect, or incompatible interface). The last two columns give the same pass counts as a rate over all 382 planned probes, which is the reported agreement rate, and over only the probes that ran.

Method	Planned	Runnable	Pass	Failures		Behavioral agreement	
				Wrong output	No runnable target	Over all 382	Over runnable only
ADFD-Migrate	382	382	327	55	0	85.6%	85.6%
Direct translation	382	99	83	16	283	21.7%	83.8%
Repo-context direct	382	98	83	15	284	21.7%	84.7%
Static profile (no ADFD)	382	69	42	27	313	11.0%	60.9%
Dependency chunking (no ADFD)	382	30	14	16	352	3.7%	46.7%

Table 5

Cross-tier migration outcome results. Sem. = ADFD semantic-alignment score; Syn. = syntax validity; Imp. = import success. Direct comparison uses 20 low-, 20 medium-, and 7 complex-tier repositories. Δ is the ADFD-minus-direct difference in percentage points.

Tier	<i>n</i>	Sem.	Syn.	Imp.	ADFD outcome	Direct <i>n</i>	Direct outcome	Δ
Low	20	95.2%	99.9%	77.2%	93.0%	20	76.1%	+16.9
Medium	20	98.3%	98.1%	72.5%	91.9%	20	65.3%	+26.6
Complex	10	99.0%	100.0%	79.8%	95.7%	7	37.2%	+58.5
All	50	97.2%	99.2%	75.9%	93.1%			

or numerical-kernel defects; the four detailed failure modes in Appendix D.4 motivate adding executable contracts and source-oracle feedback to the refinement loop. Sensitivity analysis preserves a positive ADFD–baseline mean gap for 98.9% of 10,000 sampled weightings (Appendix D.5).

Finally, chunking enabled a local 30B model to port all low-tier repositories with no metered generation charge. The ten complex repositories cost about \$117, and the ADFD workflow cost about 54% of file-wise direct translation on the seven complex repositories with recorded paired costs. Appendix D.6 reports model, construction, chunking, and error-taxonomy details; Appendix D.7 gives the cost accounting.

6. Discussion

6.1. Additional Questions and Answers

We now turn to some questions that require additional clarification.

How is data flow recovered, given pointers and aliasing?

The ADFD is inferred by an LLM and constrained by a static profile (Section 3.1.1), so no alias analysis is run. Fortran arguments may not overlap unless declared as pointers or targets, so the profile’s call graph can be trusted as written, and two names for one array are the same data store regardless. A pointer-heavy language such as C would first need points-to analysis in its adapter, which needs to be created for each source language.

Is data flow sufficient, and where does control flow go?

Control flow is not discarded; it is captured *inside* the processes. The diagram carries only what crosses procedures, which is what chunking and alignment need. Fidelity to control flow therefore rests on process specification and contract text, so control-flow-heavy repositories are more challenging: for example, quadpack (1/12 probes) and roots-fortran (0/7) recovered the correct processes but not the correct stopping logic (Appendix D.4).

Why not generate the target tests from the source test suite?

The probes do this in a stronger form: their expected values come from executing the original Fortran, and they are fixed before any port exists (Section 4.2.1). The existing suites are themselves Fortran programs, unevenly present across the corpus, so translating one is a migration task carrying the defect being measured, and a failure would not separate a faulty port from a faulty test.

6.2. Limitations and Threats to Validity

Construct: The 382 probes are targeted case-level checks, not exhaustive system tests, so behavioral agreement bounds correctness from below rather than certifying it. Exhaustive equivalence testing is not currently feasible across this corpus: many repositories need large scientific inputs, MPI/HDF5 or compiler-specific runtimes, and project-specific output oracles that do not exist uniformly. The outcome index is a separate matter: it encodes a preference for complete, importable, maintainable draft migrations,

and sensitivity analysis supports its comparative use without making it a correctness percentage. The two ablations show the ADFD contributes beyond static metadata and dependency-only chunking, but a component-by-component ablation remains future work. Human expert verification of the ported code has not been possible: the benchmark spans scientific domains (e.g., quantum chemistry, finite element analysis, weather modeling) that require specialized expertise the authors do not have and this expertise has been difficult to find, so probe-based and LLM-judged evaluation are the only correctness signals reported here.

Internal: Temperature 0.3 makes generation non-deterministic; we mitigate this by evaluating 50 diverse repositories rather than repeated runs of a few. The Claude Opus 4.6 judge is independent of the Qwen generators but shares a model family with the complex-tier Sonnet 4.5 generator, so its scores may carry family bias; no correctness claim depends on it. Prompt effort favors the baselines and ablations, whose prompts were hand-refined to their best obtainable results while ADFD-Migrate ran with fixed prompts and no per-repository tuning (Section 4.2).

External: Only Fortran-to-Python porting was evaluated, and the approach suits numerically oriented software where data-flow structure dominates control-flow complexity; heavy I/O, event-driven, or GUI logic may benefit less. The ADFD and chunking algorithms are language-agnostic, but construction needs language-specific analyzers, generation needs target-language instructions, and import reconciliation is Python-specific, so a pair such as COBOL-to-Java requires new adapters. The three model configurations (Qwen-3-Coder 30B, Qwen-3-Coder 480B, and Claude Sonnet 4.5) give some evidence of model-agnosticism.

7. Conclusion

ADFD-Migrate starts from the premise that a program implements an unobserved declarative description of its computation. It makes an inferred ADFD approximation of this latent declarative representation explicit and inspectable, then uses dependency-aware chunking to generate a target repository in bounded, ordered units. On the 50-repository f2x50 benchmark, it passes 327 of 382 source-oracle behavioral probes and exposes 3.9 $\times$ as many planned behaviors as runnable targets as either direct baseline. Its 17–59 percentage-point outcome-index advantage is consistent across repository tiers, although behavioral failures show that process-level alignment cannot guarantee formula-level fidelity.

Three insights emerge. First, an ADFD can make a useful approximation of the latent declarative representation explicit, inspectable, and language-agnostic. Second, dependency-aware chunking enables smaller open-source LLMs to generate code. Third, structuring the problem via semantic decomposition offers an alternative to relying only on model scale. Future work will integrate failed source-oracle probes into chunk refinement, strengthen domain-specific numerical contracts, evaluate smaller models and

additional language pairs, and develop richer domain-aware repair beyond import reconciliation.

Data Availability Statement

The f2x50 benchmark corpus catalog and reproduction scripts are available at <https://github.com/ShraddhaSurana/iProg/tree/main/f2x50>. The inferred ADFDs, generated Python, metrics, manifests, and analysis outputs are available at <https://doi.org/10.5281/zenodo.22054047>.

CRediT authorship contribution statement

Shraddha Surana: Conceptualization, Methodology, Software, Investigation, Formal analysis, Data curation, Writing - original draft, Writing - review and editing. **Ashwin Srinivasan:** Conceptualization, Supervision, Writing - review and editing. **Michael Bain:** Writing - review and editing.

Declaration of competing interests

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Funding

This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

A. Full ADFD-Migrate Architecture

Figure A.1 maps the porting workflow of Figure 1 to the implemented system, expanding its three steps (convert source code to an ADFD, inspect or revise the ADFD, generate target code from it) into the automatic components used in the experiments.

Recovering the source representation. The left-hand stage implements $A_S = \text{Construct}_S(R_S)$. A deterministic static-analysis agent parses R_S to produce the repository profile Σ_S : files, definitions, module relationships, call edges, procedure signatures, and input/output information. The profile is used in two ways. A bounded projection of it, together with ranked source excerpts, conditions the LLM that infers a candidate source ADFD. The complete profile is retained by the coverage agent, which checks which source files and definitions that candidate represents; if coverage is below threshold τ , the uncovered elements become focused gap directives for the next inference pass. The best candidate is retained, and the loop stops at the threshold or after the configured iteration bound. Static analysis therefore constrains and checks the explicit ADFD approximation produced by the LLM.

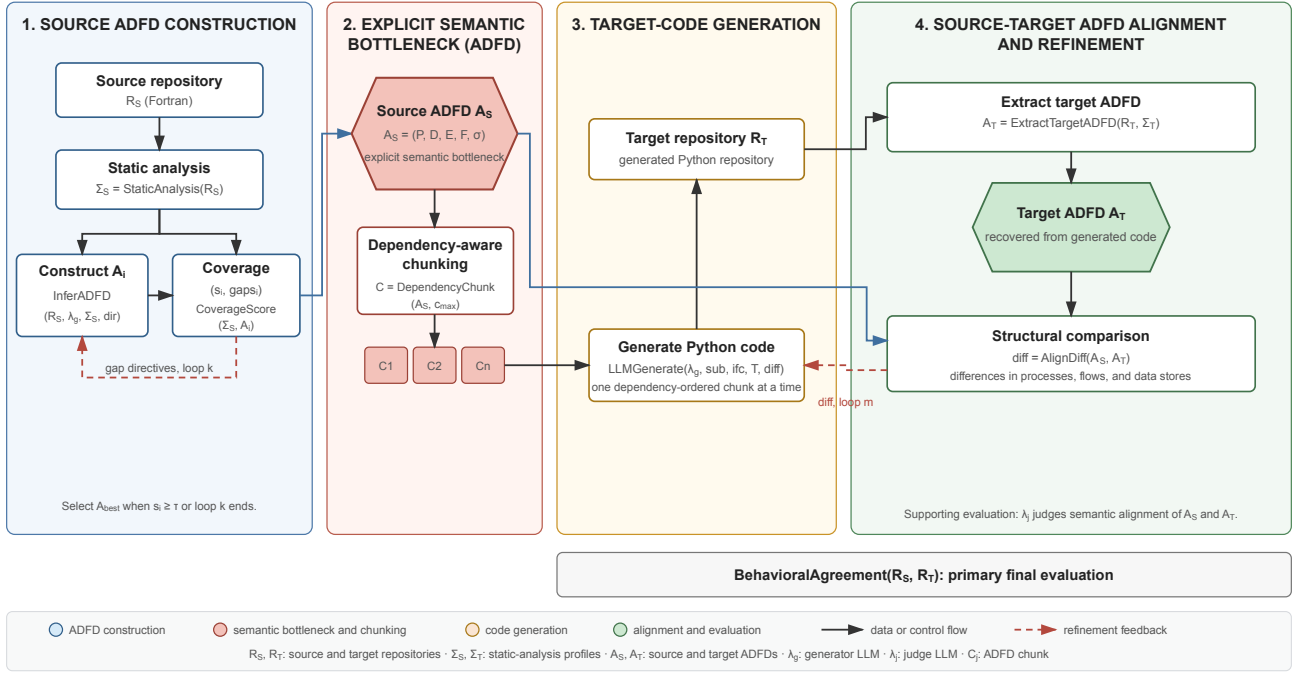


Figure A.1: Full ADFD-Migrate system architecture. Static-analysis coverage refines the inferred source ADFD; dependency-aware chunks guide generation; source–target ADFD differences guide regeneration. The LLM judge supplies supporting representation-level evidence. Behavioral agreement is the primary evaluation.

Using the ADFD as a semantic bottleneck. The middle stage begins from $A_S = (P, D, E, F, \sigma)$, rather than translating the source files independently. A deterministic dependency agent projects the process-to-process flows, collapses strongly connected components, and topologically orders the resulting graph. It then packs that order into chunks $C_1, \dots, C_n$, subject to the process bound c_{max} while keeping each strongly connected component intact. For each chunk, the generator receives its sub-ADFD, the contracts of referenced data stores and external entities, and interface summaries from already generated chunks. This is the operational role of the latent-representation claim: the inferred ADFD provides a language-independent, repository-level account of the computation that mediates between the observed Fortran repository and the generated Python repository.

Checking and refining the target. After each generation pass, deterministic Python analysis produces profile Σ_T and target ADFD A_T . Differences between A_S and A_T guide the next bounded regeneration pass. This loop tests whether the generated repository realizes the computational structure made explicit in A_S . The evaluation-only LLM judge supplies a post-generation semantic score and is not part of this loop.

Evaluation boundary. The bottom path distinguishes system checks from the final experimental criterion. Coverage and source–target ADFD alignment measure whether the representation is complete enough to guide generation

and whether its structure is preserved; they cannot establish behavioral correctness. The primary test therefore executes matched source-oracle probes against R_S and R_T . The ADFD is the proposed mechanism for improving porting, structural and semantic alignment diagnose that mechanism, and observable Fortran–Python agreement evaluates the ported program.

B. Worked ADFD Example: MINPACK

Figure B.1 shows one five-process numerical cluster from minpack, rather than its complete ADFD. The source artifact contains 16 processes, 72 flows, 10 data stores, and four external entities and is generated in three chunks; the refined target ADFD contains 32 extracted processes and 48 flows. Panel (a) maps representative MINPACK routines to ADFD processes, with the Jacobian as a data store and the user application and callback as external entities. Panel (b) uses process names extracted from the generated Python, enabling direct comparison of computational roles and flows.

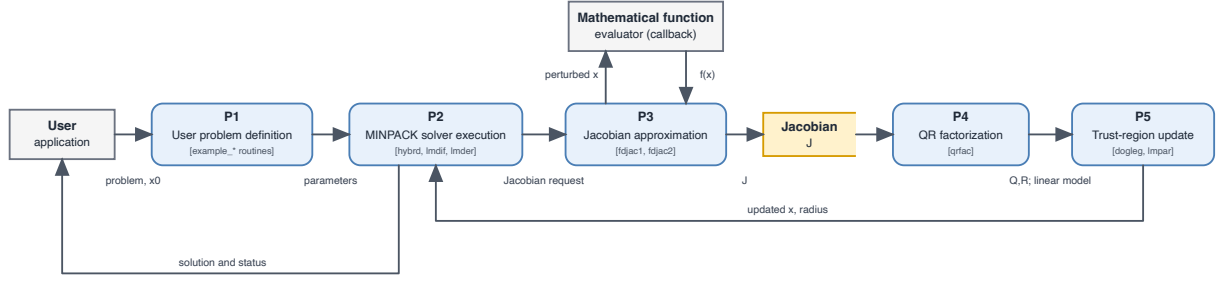
C. Experimental Details

This appendix provides the detailed experimental configuration, baseline and ablation descriptions, probe curation methodology, and outcome index definition referenced in Section 4.2.

C.1. Model Configurations

Generation models. Three generator configurations were used. Qwen-3-Coder 30B, hosted locally via Ollama, ported

(a) Five-process source-ADFD cluster inferred from Fortran MINPACK



(b) Corresponding excerpt from the generated-Python ADFD

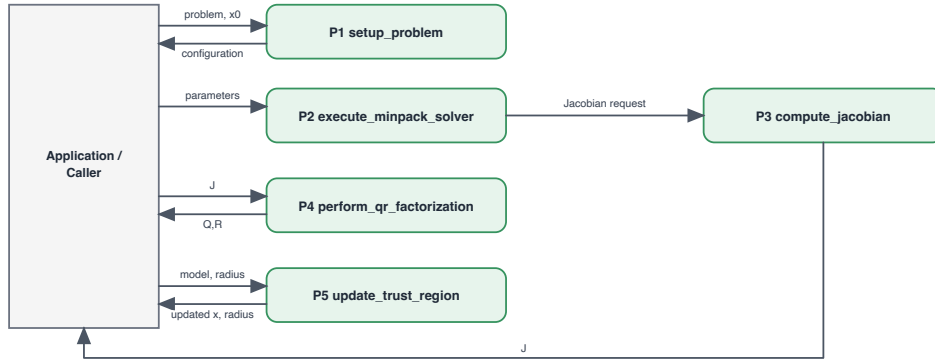


Figure B.1: Worked MINPACK ADFD excerpt using Gane–Sarson notation. Rounded rectangles are processes, the open-ended rectangle is a data store, square-cornered rectangles are external entities, and labeled arrows are flows. P1–P5 identify corresponding computational roles in the source and generated-Python ADFDs.

all 20 low-tier repositories and 4 of the 20 medium-tier ones (wavelets, SISSO, quadpack, MPM3D-F90); Qwen-3-Coder 480B via cloud-hosted Ollama ported the other 16 medium-tier repositories, on which the 30B model produced excessive stubs or timed out; Anthropic Claude Sonnet 4.5 via API ported all 10 complex-tier repositories. Each repository’s generator was also used for all of its baseline and ablation ports.

All generation models used temperature 0.3, JSON response format, and 1200 s timeout. The non-zero temperature allowed bounded refinement retries to explore alternatives after a repeated structural defect. We did not tune temperature; 0.3 is a fixed setting, not an empirical optimum.

Judge model. Claude Opus 4.6 (extended thinking; temperature 0.2; 600 s timeout) was used only for semantic evaluation. It is independent of the Qwen generators, but shares the Anthropic family with the complex-tier generator. Target-ADFD label enrichment also uses the generator, so the score may retain family and construction bias Panickssery, Bowman and Feng (2024). This score is supporting evidence only.

C.2. Baseline and Ablation Descriptions

Direct baseline. Translates each Fortran file to Python independently without an ADFD-mediated intermediate representation.

Repository-context baseline. Additionally supplies neighboring modules and dependency summaries while retaining independent file generation.

Static-profile ablation (no ADFD). A pure static-analysis baseline: it supplies the same repository profile used to construct the source ADFD (module structure, call graphs, function/subroutine signatures, and file-to-definition mappings) but omits the ADFD itself, isolating the contribution of the declarative representation on top of static analysis alone.

Dependency-chunking ablation (no ADFD). Chunks the source call graph using strongly connected component detection and topological ordering, the same dependency-ordering mechanism ADFD-Migrate uses, but applied directly to the call graph instead of an ADFD process graph. This isolates the contribution of the declarative representation on top of dependency-aware chunking alone.

C.3. Alignment Scoring

The source ADFD A_S and the ADFD A_T extracted from the generated code are compared twice: a deterministic structural comparison that drives the refinement loop, and an LLM-judged semantic comparison used only for evaluation.

Structural alignment (ALIGNDIFF). Names are lower-cased and non-alphanumeric characters become underscores. Normalized names match by equality, substring containment, or Jaccard overlap $|a \cap b|/|a \cup b| \geq 0.5$ between their underscore-delimited token sets. Thus, `solve_linear_system` and `linear_system_solver` match at 2/4. Flows are normalized (source, target) pairs. Unmatched processes, flows, and data stores on either side (i.e. missing or extra) form *diff*, which Procedure 2 passes to LLMGENERATE, and their empty union is the alignment criterion of Section 4.2.

The comparison also reports rename-insensitive count-coverage and topology-similarity summaries, combined into a composite structural alignment score. These are diagnostic only: the refinement loop reacts to the missing and extra sets, never to the composite score, so a port is never regenerated merely for renaming an element.

Semantic alignment (S_{sem}). The judge receives both ADFDs as text: each process with its name, specification, and declared inputs and outputs; the flow list with endpoint names and labels; and the data-store and external-entity names. It is instructed that names will differ across languages and must not be penalized on that basis, that one source process may legitimately map to several target processes or the reverse, and that each aligned pair is scored on [0, 1] with 1.0 identical functionality, 0.5 partial overlap, and 0.0 unrelated. It returns the aligned pairs with per-pair scores, the unmatched processes on each side, and one overall score. S_{sem} is that overall score clamped to [0, 1].

C.4. Probe Curation

Probes were selected from the *source* side: routines were chosen from the Fortran repository for being public, deterministic, and executable with small self-contained inputs. Each probe's expected behavior is defined entirely by the Fortran oracle, not by any generated output. Probe selection was therefore independent of which migration method produced the Python under test.

Where a generated repository exposes a source behavior under a different name or module (e.g., `string_to_value` realized as `atoi_int32`), the harness resolves the target by name-based search over the generated `src/` tree and, where needed, repository-specific driver adapters; this resolution step is applied identically to baseline and ADFD-Migrate outputs.

Every probe is curated against an executable oracle: either the original Fortran routine compiled with `gfortran` or a reference `f2py` stub (a Python-callable wrapper compiled directly from the original Fortran via `f2py` Peterson (2009)), run on the same inputs as the generated Python.

Comparison modes and tolerances. Each probe records its inputs, one of five comparison modes, and numeric tolerances (absolute and relative) where applicable. Of 382 probes, 205 use numeric tolerance, 93 array equality, 57 scalar/string equality, 23 Boolean equality, and four equality after unit normalization. Numeric agreement uses `numpy.allclose`: $|x_{py} - x_f| \leq atol + rtol |x_f|$ elementwise with matching shapes for arrays. Per-probe tolerances are 10^{-14} for closed-form functions, 10^{-12} for BLAS kernels, 10^{-10} or 10^{-8} for composite routines and table lookups, and 10^{-6} for iterative solvers; exact modes use zero tolerance. Inputs are small, self-contained, and written into the probe definition, so oracle and port receive identical values.

C.5. Prompts

ADFD inference. INFERADFD runs two LLM passes per module family: pass 1 produces the structure, validated without requiring per-process detail, and pass 2 fills in specifications, conditions, and typed I/O, validated against the expected source-file list. The per-family fragments then receive disjoint ID prefixes and are merged. The system prompt casts the model as “a principal software architect building a reconstruction-grade data flow diagram” (pass 2 adds “and QA engineer refining a DFD for cross-language repository reconstruction”) and requires strict JSON with no markdown.

Each user prompt is a JSON payload: the task statement, the repository name and source language, the constraints, the bounded profile projection, and, from the second construction iteration onward, the gap directives and a summary of the previous best ADFD. The constraints fix the schema (each process carries an id, name, type, specification, pre- and post-conditions, typed inputs and outputs, and the list of source files it covers, which is what makes coverage scoring possible; each flow carries an id, source, target, and label) and require stable ID prefixes, flows referencing only existing IDs, at least one flow per process, explicit flow labels, and specifications of at least 40 characters.

Target generation. System prompt: “You are a principal Python engineer specializing in porting numerical and scientific Fortran libraries to Python. Given an Annotated Data Flow Diagram (ADFD), generate a complete, well-structured Python implementation.” Requirements instruct the model to use NumPy/SciPy where appropriate, preserve numerical precision (`np.float64` by default), include type hints and docstrings, respect the ADFD pre/post-conditions, and return only JSON of the form `{"files": [{"path": "...", "content": "..."}]}`. For chunked generation the prompt additionally states the chunk index, the total chunk count, and that the chunk may import from earlier chunks listed in `upstream_modules` but not the reverse.

Direct baseline. System prompt: “You are an expert Fortran-to-Python translator. Given a Fortran source file, produce the equivalent Python code that faithfully reproduces the same functionality.” Rules instruct the model to preserve computational logic exactly, use idiomatic Python,

include type hints and docstrings, use `snake_case`, and return only valid Python. The user prompt supplies the file name and its full source.

Repository-context baseline. One prompt per source file: “Translate one Fortran source file to Python using repository context... Do not use, infer any declarative intermediate representation. Use the repository context for module names, interfaces, dependencies, shared constants, and naming.” The prompt gives the repository name, source path, expected output path, a compact repository context as JSON, and the full Fortran source, and requires one primary Python file at the expected path with complete implementations, imports, and preserved public routine names.

Static-profile ablation (no ADFD). One prompt per repository: “You are translating an open-source Fortran repository to Python. Use the repository-level static-analysis profile below. Generate a Python repository that preserves the observable behavior, public routines, module structure, and dependencies as much as possible.” The compacted profile follows as JSON, file summaries ordered by static importance and fitted to the same character budget as the ADFD prompts.

Dependency-chunking ablation (no ADFD). One prompt per chunk: “Translate this call-graph/SCC chunk of a Fortran repository to Python. Use only graph chunking, static context, and upstream interface summaries.” The prompt lists the chunk’s nodes and the upstream interface summaries as JSON and asks for complete Python files preserving callable interfaces. Chunks come from the same SCC detection and topological ordering as ADFD-Migrate (Stages 1–3 of DEPENDENCYCHUNK), applied to the source call graph rather than to an ADFD process graph.

Baseline and ablation generation harness. All four share one assembly system prompt (“You are a principal Python engineer specializing in repository-scale Fortran-to-Python migration for scientific software”) that fixes the same JSON file-list output schema, path conventions, and engineering requirements as ADFD-Migrate. They therefore differ only in the context they receive. The complete prompt texts are in the replication package.

C.6. Why Not Reference-Based Metrics

Text-similarity metrics such as BLEU Papineni, Roukos, Ward and Zhu (2002) and CodeBLEU Ren, Guo, Lu, Zhou, Liu, Tang et al. (2020) are inapplicable because no reference Python translation of these repositories exists, which is a selection criterion of f2x50 (Section 4.1). Even given references, cross-language ports legitimately differ in syntax and idiom, and prior work reports poor correlation between these metrics and functional correctness Macedo et al. (2025); Ibrahimzada et al. (2025). The same property rules out training-data contamination: there is no Python version for a model to have memorized.

C.7. Migration Outcome Index

For comparison with baselines, we use a weighted migration outcome index:

$$\text{Outcome} = (0.6 \cdot S_{\text{sem}} + 0.3 \cdot S_{\text{exec}} + 0.1 \cdot S_{\text{qua}}) \times r_{\text{impl}}$$

Component definitions.

- S_{sem} : LLM-judged ADFD semantic-alignment score (Appendix C.3).
- S_{exec} : Executability score, calculated as $(0.6 \cdot F_{\text{syn}} + 0.4 \cdot F_{\text{imp}})$, where F_{syn} is the fraction of generated .py files parseable by `ast.parse` and F_{imp} is the fraction that execute cleanly when loaded via `importlib` after the generated `requirements.txt` has been installed.
- S_{qua} : Code quality, calculated as $\frac{(C_{\text{doc}} + C_{\text{type}})}{2}$ over all generated function definitions, where C_{doc} is the fraction whose body opens with a string literal and C_{type} the fraction carrying a return annotation or at least one annotated parameter.
- r_{impl} : Implementation rate, the proportion of generated functions whose bodies contain real logic. A function counts as a stub when, after an optional docstring, its body is exactly one of: `pass`, `...`, a `raise` statement, a bare `return`, or `return None`.

For fully implemented ports $r_{\text{impl}} \approx 1$ and the multiplier has no effect; for scaffold-only conversions it penalizes the score proportionally.

C.8. Generated Tests

Generated `pytest` suites execute the generated Python, but their expected outcomes are generated from the target-side code or from generic testing patterns rather than from an independent Fortran oracle. They expose syntax, import, packaging, and runtime failures and provide regression scaffolding.

C.9. Structural Comparison

In addition to the ADFD alignment in Appendix C.3, a structural score compares AST-level attributes (function count, class count, identifier overlap, and call-graph edges) between the generated Python and the original Fortran. It is *not* included in the outcome index and does not drive refinement, because cross-language porting legitimately changes identifiers and module boundaries (for example, replacing Fortran `COMMON` blocks with Python classes). It is recorded as a diagnostic only.

D. Supporting Results and Diagnostics

D.1. Behavioral Agreement by Tier

Table D.1 decomposes the 382 source-oracle probes by repository tier.

Table D.1: Executable Fortran-vs-Python behavioral validation by tier.

Tier	Repos	Checks	Pass	Fail	Rate
Low	20	142	116	26	81.7%
Medium	20	127	110	17	86.6%
Complex	10	113	101	12	89.4%
All	50	382	327	55	85.6%

D.2. Extended Baseline and Ablation Comparisons

Table D.2 gives the full outcome-index distributions for ADFD-Migrate and the four non-ADFD alternatives compared in Section 5.2.

Table D.2: Migration outcome comparison with non-ADFD alternatives. All four comparisons share a common 47-repository subset.

Method	<i>n</i>	Mean	Median	Range
ADFD-Migrate	47	0.924	0.939	0.771–0.989
Direct translation	47	0.654	0.764	0.000–0.980
Repository-context direct	47	0.709	0.729	0.397–0.924
Static profile (no ADFD)	47	0.498	0.508	0.000–0.741
Dependency chunking (no ADFD)	47	0.535	0.517	0.153–0.881

Table D.3 shows representative repository-level differences.

Table D.3: Representative repository-level migration outcome comparisons.

Repository	Direct	ADFD	Δ (pp)
CaNS	42.1%	94.2%	+52.1
fftpack	61.4%	92.0%	+30.6
toml-f	60.0%	85.5%	+25.5
F-A-Toolkit	83.9%	95.7%	+11.8
arpack-ng	38.5%	96.3%	+57.8
dftd4	44.2%	95.3%	+51.1

Across all available pairs, ADFD-Migrate improves over direct translation by 0.270 mean outcome-index points (95% bootstrap confidence interval [0.206, 0.341]), over the static-profile ablation by 0.426 [0.374, 0.476], over the dependency-chunking ablation by 0.389 [0.343, 0.435], and over repository-context direct by 0.215 [0.181, 0.250]. Wilcoxon signed-rank tests are consistent with positive improvements ($p < 0.001$).

D.3. Generated Tests and Import Reconciliation

Generated tests produced 1,232 executions: 1,205 passed, one failed, and 26 were skipped because target-side defects prevented execution. Among the 1,206 non-skipped tests, the low, medium, and complex tiers passed 452/453, 509/509, and 244/244 respectively (99.9% overall). These tests provide runtime sanity checks but are not independent evidence of Fortran equivalence.

Of 972 generated modules, 770 imported successfully and 202 failed. Missing dependencies (71, 35%) and Import

Error (47, 23%) dominate, followed by circular-import symptoms (22, 11%), ModuleNotFoundError (21, 10%), missing-symbol NameError (21, 10%), AttributeError (13, 6%), and SyntaxError (7, 3%). A conservative AST-based reconciliation pass modified 37 files in copied outputs and improved 12 repositories, raising mean module-level import success from 79.7% to 82.3%. It repairs verifiable module paths, re-exports, package initializers, and standard-library imports, but does not synthesize logic or repair behavioral defects.

D.4. Outcome Index Versus Behavioral Agreement

Outcome index and behavioral agreement are not monotonic. The mean outcome index is 93.5% for the 40 repositories with no failed probes, 90.7% for the two with 1–2 failures, 92.1% for the four with 4–5 failures, and 91.7% for the four with 7–11 failures. The worst-agreement cases show what representation-level measures miss:

- roots-fortran (97.0% outcome) has a formula-level error in bisection termination that recurs across its Brent, Ridders, and Illinois solvers;
- Incompact3d (91.6%) uses incorrect Padé stencil coefficients;
- quadpack (81.1%) disagrees in the dqk15 and dqk21 numerical kernels; and
- ccpp-physics (96.6%, 7/11) differs in Fortran diagnostic formatting and side effects.

These defects motivate richer ADFD contracts with boundary behavior, formatting conventions, numerical examples, and source-oracle probes inside the refinement loop. They are also the kind of localized, contract-level issue that the human ratification checkpoint in Figure 1 is designed to let a domain expert catch before generation; that interaction is not evaluated in this paper.

D.5. Sensitivity of the Migration Outcome Index

Table D.4 summarizes alternative aggregation rules. In 10,000 random Dirichlet weightings, 98.9% preserve a positive mean ADFD–baseline gap; the mean sampled gap is 0.143 and the median is 0.146.

Table D.4: Sensitivity of the migration outcome comparison.

Configuration	ADFD	Direct	Gap	ADFD > Direct
Paper weights	0.926	0.660	0.263	95.7%
Drop implementation gate	0.941	0.761	0.179	91.5%
Geometric mean	0.923	0.633	0.288	95.7%
Weakest-link minimum	0.826	0.534	0.285	80.9%
Add structural 15%	0.897	0.681	0.213	89.4%

D.6. Model Scale, ADFD Construction, and Generation

The local 30B model ported all 20 low-tier repositories with a 93.0% mean outcome index, compared with 76.1% for the same model under direct translation. Within the medium tier, the four repositories completed by 30B average

83.6%, and the 16 completed by 480B average 94.0%; this is descriptive because model selection depended on successful completion.

Source ADFDs contain 1–53 processes in the low tier (mean 19.1), 9–132 in the medium tier (mean 27.4), and 22–85 in the complex tier (mean 41.1). With $c_{\max} = 3$, these become 1–42 chunks. Large SCCs are never split, preserving dependency correctness at the cost of occasional oversized chunks. Table D.5 records the generated-code error categories used in diagnosis.

Table D.5: Generated-code error taxonomy.

ID	Category	Severity	Detection
E1	API hallucination	Major	AST/import check
E2	Type mismatch	Major	Static pattern
E3	Control-flow divergence	Major	LLM judge
E4	Interface mismatch	Minor	ADFD contract diff
E5	Numerical precision	Minor	Behavioral probe
E6	Syntax	Critical	ast.parse
E7	Import failure	Critical	Runtime import test

D.7. Cost Analysis

Recorded metered generation charges were zero for the low and medium tiers under the local and cloud-hosted Ollama configurations. The ten complex repositories incurred about \$117. For the seven complex repositories also run under the plain direct baseline, ADFD-Migrate cost about 54% as much (\$86 versus \$159). `lapack` was generated in 18 process chunks rather than one call per 3,587 files, approximately 200× fewer generation calls (`lapack` is not included in the evaluation analysis as it could not be run for all baselines/ablations). On the seven complex repositories with repository-context direct costs, that baseline cost \$479 versus about \$78 for ADFD-Migrate.

D.8. Human-in-the-Loop Potential

The explicit ADFD provides an artifact-level checkpoint that a developer can inspect before target-code generation. The evaluated pipeline is fully automatic, but the same representation supports RATIFY, REFUTE, REVISE, and REJECT interactions in the two-way intelligibility protocol implemented by Dhaani (<https://shraddhasurana.github.io/dhaani/>). This extends interactive DFD-based synthesis Surana et al. (2026) from code construction to automated porting.

E. Repository-Level Results

Tables E.1, E.2, and E.3 report the repository-level measurements underlying the aggregate results in Section 5. Here, $\#P$ is the number of ADFD processes and Beh. is the number of passing behavioral probes over attempted probes.

F. Reproducibility Details

F.1. Representative Behavioral Outputs

Table F.1 gives representative Fortran-oracle comparisons. The full set of probe inputs, tolerances, and outputs is included in the replication package.

Table F.1: Representative outputs from the ADFD-Migrate executable Fortran-vs-Python harness. *Max. abs.* is largest absolute difference between the Fortran oracle and generated Python.

Repository / routine	Input	Common output	Max. abs.
tsunami / num_tiles	n=6	[3, 2]	0
lapack / dcopy	x=[5, -1, 0, 2.5, 7]	[5, -1, 0, 2.5, 7]	0
minpack / enorm	x=[1e-20, 3, 4, 1e19]	1.0e19	0
cp2k / kahan_	a=[1, 2, 3], b=[4, 5, 6]	32.0	0
dot_product			
arpack-ng / compute_	r=[-3, -4, 0, 0]	5.0	0
residual_			
norm			
FKB / gaussian	x=[-2, -0.5, 0, 1.25]	[0.01832, 0.77880, 1.0, 0.20961]	0
stdlib / gauss_	n=3, [-1, 1]	[-0.77460, 0, 0.77460]	4.4×10^{-16}
legendre_			
points			

F.2. Computational Environment

Experiments ran on a MacBook Pro (Mac16,5) with an Apple M4 Max (16-core CPU, integrated GPU, and 48 GB unified memory) under macOS 26.5.1 (Darwin 25.5.0). Qwen-3-Coder 30B inference used local Ollama acceleration; Qwen-3-Coder 480B and Anthropic inference were remote. Configuration manifests record models, sampling parameters, token use, system information, and stage timings.

Acknowledgements

During part of this work, AS was a visiting Professorial Fellow at UNSW, and a Visiting Professor at the Centre for Health Informatics at Macquarie University. He is a member of the Anuradha and Prashant Palakurthi Centre for AI Research (APPCAIR) at BITS Pilani. The authors would like to thank Santonu Sarkar and Soumyadip Bandyopadhyay for their comments on the paper.

Declaration of generative AI and AI-assisted technologies in the manuscript preparation process

During preparation of this manuscript, the authors used AI-assisted drafting tools to help organize empirical material, integrate tables, and improve wording. The authors reviewed and edited the content and take full responsibility for the manuscript.

Table E.1

Repository-level results for the low-complexity tier.

Repository	LoC	Files	#P	Chunks	Syntax	Import	Semantic	Beh.	Outcome	Time	Model
fortran2018-ex.	3.1k	75	19	7	100%	100%	100%	4/4	100.0%	25 min	30B
roots-fortran	5.2k	4	1	1	100%	100%	95%	0/7	97.0%	13 min	30B
forpy	13.8k	9	3	1	100%	100%	95%	4/4	97.0%	2 min	30B
tsunami	3.1k	33	31	11	100%	100%	95%	7/7	97.0%	214 min	30B
IO-Fortran-Lib.	40.9k	34	6	2	100%	70%	100%	4/4	96.4%	21 min	30B
functional-fortran	6.6k	27	10	4	100%	67%	100%	7/7	96.0%	13 min	30B
M_time	14.7k	58	37	13	100%	100%	92%	4/4	95.2%	51 min	30B
fortranlib	28.4k	38	12	4	100%	94%	92%	2/4	94.5%	350 min	30B
CaNS	11.3k	40	40	9	98%	87%	98%	1/10	94.2%	222 min	30B
fastGPT	2.1k	19	14	5	100%	53%	100%	18/18	94.1%	25 min	30B
ABAQUS	1.9k	5	6	2	100%	100%	88%	6/6	92.8%	12 min	30B
M_strings	35.4k	112	26	9	100%	88%	93%	4/4	92.3%	107 min	30B
fftpack	4.5k	70	18	6	100%	90%	92%	4/8	92.0%	106 min	30B
M_args-main	4.1k	16	12	4	100%	50%	100%	4/4	91.8%	68 min	30B
datetime-fortran	3.2k	19	20	7	100%	35%	100%	12/12	91.2%	50 min	30B
fortran-utils	13.1k	96	14	5	100%	77%	95%	7/7	90.8%	99 min	30B
test-drive	4.3k	5	53	18	100%	71%	92%	8/8	90.3%	71 min	30B
bspline-fortran	15.8k	19	23	8	100%	34%	95%	2/6	88.1%	30 min	30B
toml-f	26.0k	94	17	6	100%	87%	88%	10/10	85.5%	110 min	30B
FKB	1.5k	17	20	6	100%	44%	95%	8/8	83.3%	36 min	30B
Mean					99.9%	77.2%	95.2%	116/142	93.0%	81 min	

Table E.2

Repository-level results for the medium-complexity tier.

Repository	LoC	Files	#P	Chunks	Syntax	Import	Semantic	Beh.	Outcome	Time	Model
stdlib	35k	405	16	6	100%	100%	100%	11/11	98.9%	61 min	480B
FOODIE	19k	44	14	5	100%	97%	100%	4/4	96.8%	31 min	480B
arpack-ng	148k	334	16	6	100%	69%	100%	7/7	96.3%	71 min	480B
atomsk	86k	166	12	4	100%	72%	100%	4/4	96.1%	47 min	480B
WPS	32k	123	16	6	100%	73%	100%	4/4	96.1%	53 min	480B
packmol	13k	40	132	42	99%	74%	100%	8/8	96.0%	177 min	480B
F-A-Toolkit	28k	58	90	30	95%	92%	100%	4/4	95.7%	253 min	480B
dftd4	16k	42	15	5	97%	72%	100%	4/4	95.3%	89 min	480B
minpack	10k	13	16	3	100%	74%	98%	9/9	94.9%	75 min	480B
coretran	41k	116	17	6	100%	78%	98%	4/4	94.6%	59 min	480B
Cmathtuts	28k	46	11	4	94%	69%	100%	4/4	94.3%	66 min	480B
xtb	192k	378	14	5	95%	74%	100%	4/4	93.9%	98 min	480B
Incompact3d	45k	55	16	5	91%	60%	98%	0/5	91.6%	120 min	480B
json-fortran	26k	61	22	8	100%	62%	92%	6/6	90.3%	100 min	480B
wavelets	4.8k	10	32	11	95%	88%	100%	9/9	89.5%	58 min	30B
SISSO	7.9k	10	9	3	100%	48%	100%	4/4	87.0%	69 min	30B
crest	94k	178	13	5	98%	75%	85%	3/4	86.8%	91 min	480B
neural-fortran	13k	101	17	6	100%	64%	95%	8/8	86.2%	37 min	480B
quadpack	9.3k	13	25	9	100%	29%	100%	1/12	81.1%	100 min	30B
MPM3D-F90	7.8k	9	44	15	99%	80%	100%	12/12	77.0%	106 min	30B
Mean					98.1%	72.5%	98.3%	110/127	91.9%	88 min	

References

- Bairi, R., Sonwane, A., Kanade, A., C., V.D., Iyer, A., Parthasarathy, S., et al., 2024. CodePlan: Repository-level coding using LLMs and planning, in: Proceedings of the ACM on Software Engineering, pp. 675–698. doi:10.1145/3643757.
- Chakaravarthy, V., Roy Choudhury, A., Kanvar, V., Katan, R., Pandian, S., Raghuvanshi, A., et al., 2026. Efficient translation of long code blocks using large language models, in: Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering. SANER 2026 Research Track.
- Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H.P.d.O., Kaplan, J., et al., 2021. Evaluating large language models trained on code. arXiv abs/2107.03374.
- Cognition Labs, 2024. Devin: The first AI software engineer. <https://www.cognition.ai/blog/introducing-devin>.
- Comella-Dorda, S., Wallnau, K., Seacord, R.C., Robert, J., 2000. A survey of legacy system modernization approaches. Technical Note CMU/SEI-2000-TN-003.
- DeMarco, T., 1979. Structured Analysis and System Specification. Yourdon Press.
- Grosse-Kunstleve, R.W., Terwilliger, T.C., Sauter, N.K., Adams, P.D., 2012. Automatic Fortran to C++ conversion with FABLE. Source Code for Biology and Medicine 7, 5. doi:10.1186/1751-0473-7-5.
- Huang, Y., Qi, M., Yao, Y., Wang, M., Gu, B., Clement, C., et al., 2023. Program translation via code distillation, in: Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, pp. 10903–10914. doi:10.18653/v1/2023.emnlp-main.672. also available as arXiv:2310.11476.
- Ibrahimzade, A.R., Ke, K., Pawagi, M., Abid, M.S., Pan, R., Sinha, S., et al., 2025. AlphaTrans: A neuro-symbolic compositional approach for repository-level code translation and validation, in: Proceedings of the ACM on Software Engineering, pp. 2454–2476. doi:10.1145/3729379.

Table E.3

Repository-level results for the complex tier.

Repository	LoC	Files	#P	Chunks	Syntax	Import	Semantic	Beh.	Outcome	Time	Model
lapack	1.56M	3,587	52	18	100%	90%	100%	16/16	98.3%	105 min	S4.5
pymc2	132k	410	22	7	100%	80%	100%	10/10	96.7%	98 min	S4.5
ccpp-physics	325k	240	45	15	100%	78%	99.9%	7/11	96.6%	135 min	S4.5
Nek5000	281k	327	33	12	100%	90%	98%	13/13	96.5%	232 min	S4.5
petsc	29k	200	25	9	100%	80%	100%	8/8	96.2%	115 min	S4.5
hdf5	94k	158	85	29	100%	77%	100%	10/10	95.7%	107 min	S4.5
elmerfem	1.07M	2,211	35	12	100%	88%	100%	8/8	95.7%	162 min	S4.5
fpm	48k	217	44	15	100%	59%	100%	8/16	94.5%	100 min	S4.5
openfast	505k	330	34	9	100%	66%	100%	11/11	94.2%	108 min	S4.5
cp2k	1.33M	1,325	36	5	100%	90%	92%	10/10	93.1%	65 min	S4.5
Mean					100.0%	79.8%	99.0%	101/113	95.7%	123 min	

- fSE 2025; arXiv:2410.24117.
- Kahn, A.B., 1962. Topological sorting of large networks. *Communications of the ACM* 5, 558–562.
- LFortran Contributors, 2023. LFortran: Modern interactive Fortran compiler. <https://lfortran.org>. Accessed 15 July 2026.
- Liu, N.F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F., et al., 2024. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics* 12, 157–173.
- Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., et al., 2021. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation, in: *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*.
- Macedo, M., Tian, Y., Nie, P., Cogo, F.R., Adams, B., 2025. InterTrans: Leveraging transitive intermediate translations to enhance LLM-based code translation, in: *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering*, pp. 1153–1164. doi:10.1109/ICSE55347.2025.00236. iCSE 2025; arXiv:2411.01063.
- May, V., Misra, D., Luo, Y., Sridhar, A., Gehring, J., Ribeiro, S.S., et al., 2026. FreshBrew: A benchmark for evaluating AI agents on Java code migration, in: *Proceedings of the 48th IEEE/ACM International Conference on Software Engineering*. ICSE 2026; arXiv:2510.04852.
- von Mayrhauser, A., Vans, A.M., 1995. Program comprehension during software maintenance and evolution. *Computer* 28, 44–55.
- Pan, R., Ibrahimzada, A.R., Krishna, R., Sankar, D., Wassi, L.P., Merler, M., et al., 2024. Lost in translation: A study of bugs introduced by large language models while translating code, in: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pp. 995–1007. doi:10.1145/3597503.3639226.
- Panickssery, A., Bowman, S.R., Feng, S., 2024. LLM evaluators recognize and favor their own generations. arXiv preprint arXiv:2404.13076.
- Papineni, K., Roukos, S., Ward, T., Zhu, W.J., 2002. BLEU: A method for automatic evaluation of machine translation, in: *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pp. 311–318.
- Peterson, P., 2009. F2PY: a tool for connecting Fortran and Python programs. *International Journal of Computational Science and Engineering* 4, 296–305.
- Pohle, C., 2026. AgenticTyper: Automated typing of legacy software projects using agentic AI. arXiv preprint arXiv:2602.21251. ICSE 2026 Student Research Competition.
- Ren, S., Guo, D., Lu, S., Zhou, L., Liu, S., Tang, D., et al., 2020. CodeBLEU: A method for automatic evaluation of code synthesis. arXiv abs/2009.10297.
- Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X.E., et al., 2023. Code Llama: Open foundation models for code. arXiv preprint arXiv:2308.12950.
- Rozière, B., Lachaux, M.A., Chatussot, L., Lample, G., 2020. Unsuper-vised translation of programming languages, in: *Advances in Neural Information Processing Systems*.
- Saha, S.K., Rabbi, F., Pham, T.M.T., Wang, S., Yang, J., 2026. Specification-driven code translation powered by large language models: How far are we? *Journal of Systems and Software* 241, 112964. doi:10.1016/j.jss.2026.112964.
- Surana, S., Srinivasan, A., Bain, M., 2026. Engineering systems for data analysis using interactive structured inductive programming, in: *Advanced Information Systems Engineering*, Springer Nature Switzerland, Cham. pp. 249–266.
- Szafraniec, M., Roziere, B., Leather, H., Charton, F., Labatut, P., Synnaeve, G., 2023. Code translation with compiler representations. arXiv preprint arXiv:2207.03578.
- Tarjan, R., 1972. Depth-first search and linear graph algorithms. *SIAM Journal on Computing* 1, 146–160.
- Wang, X., Li, B., Song, Y., Xu, F.F., Tang, X., Zhuge, M., et al., 2025. Openhands: An open platform for ai software developers as generalist agents. arXiv:2407.16741.
- Yang, J., Jimenez, C.E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K., et al., 2024a. SWE-Agent: Agent-computer interfaces enable automated software engineering, in: *Advances in Neural Information Processing Systems*. NeurIPS 2024; arXiv:2405.15793.
- Yang, Z., Liu, F., Yu, Z., Keung, J.W., Li, J., Liu, S., et al., 2024b. Exploring and unleashing the power of large language models in automated code translation. *Proceedings of the ACM on Software Engineering* 1. doi:10.1145/3660778.
- Yin, X., Ni, C., Nguyen, T.N., Wang, S., Yang, X., 2024. Rectifier: Code translation with corrector via LLMs. arXiv abs/2407.07472.
- Yourdon, E., 1989. *Modern Structured Analysis*. Prentice Hall.
- Zhang, W., Zhang, C., Luo, Z., Ma, J., Yuan, W., Gu, C., et al., 2025. SemanticForge: Repository-level code generation through semantic knowledge graphs and constraint satisfaction. arXiv preprint arXiv:2511.07584.
- Zhang, Y., Ruan, H., Fan, Z., Roychoudhury, A., 2024. AutoCodeRover: Autonomous program improvement, in: *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 1592–1604. doi:10.1145/3650212.3680384.
- Zheng, Q., Xia, X., Zou, X., Dong, Y., Wang, S., Xue, Y., et al., 2023. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x, in: *Proc. 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pp. 5673–5684. doi:10.1145/3580305.3599790.