Neurosymbolic Translation Pipeline of CLM-ml to JAX

Aya Lahlou*, Milind Kudapa, Haoqi Zhang, Linnia Hawkins, Pierre Gentine

Columbia University, Earth and Environmental Engineering Department

Motivation: Technical Debt in ESMs

Architecture Challenges:

- Complex nested hierarchies: Gridcells → Landunits → Columns → PFTs
- Extensive module-level global state with hidden dependencies

ML Integration Barriers:

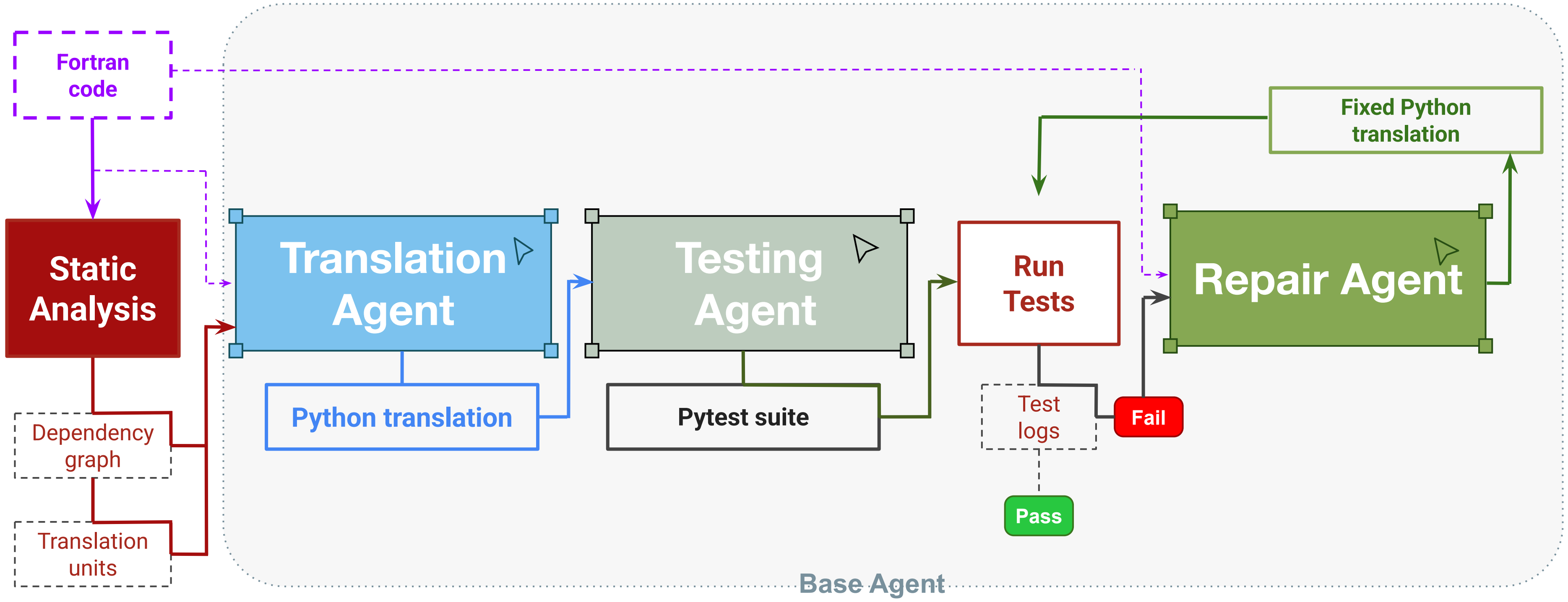
- Language paradigm mismatch (procedural vs. object-oriented/functional)
- Memory management conflicts (static allocation vs. dynamic tensors)

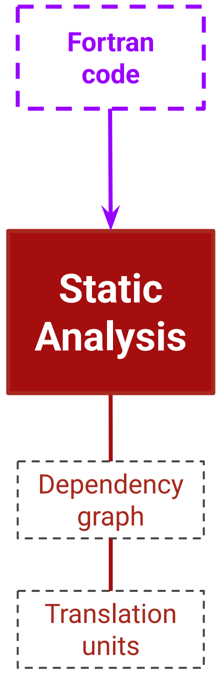
Motivation: Technical Debt in ESMs

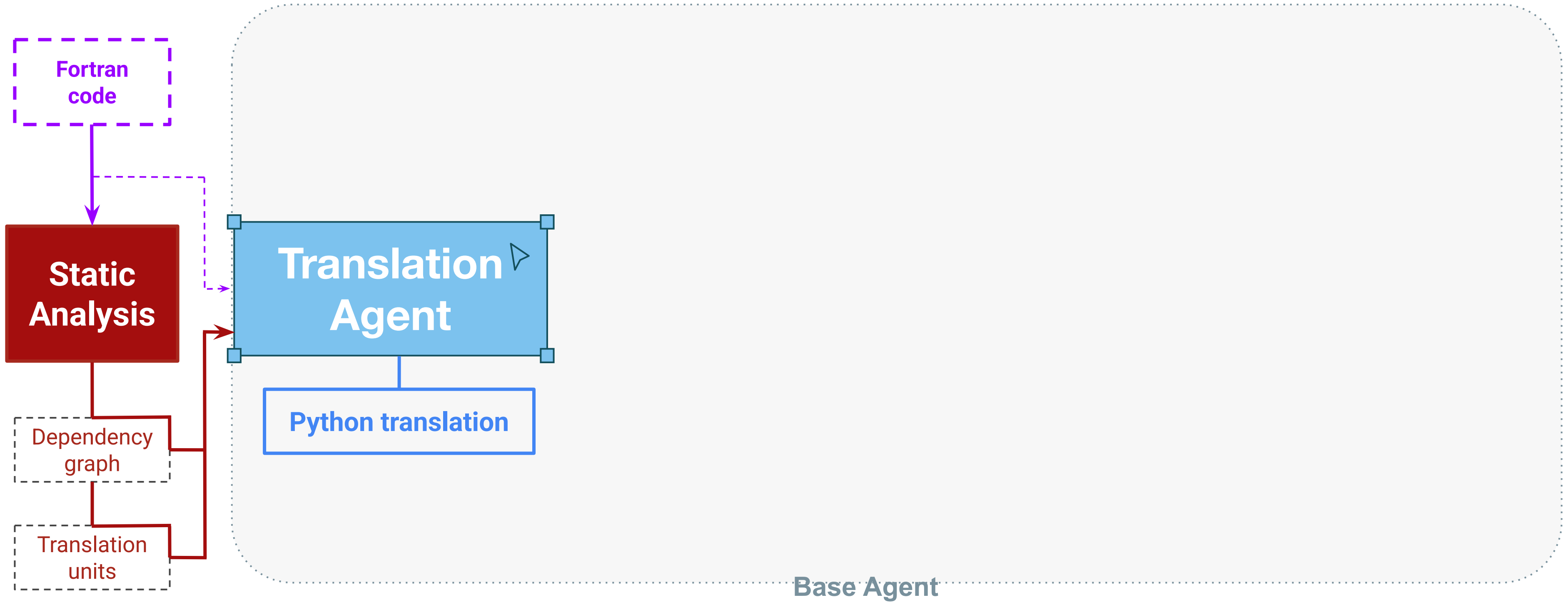
Why Automatic Differentiation Fails on Legacy Fortran:

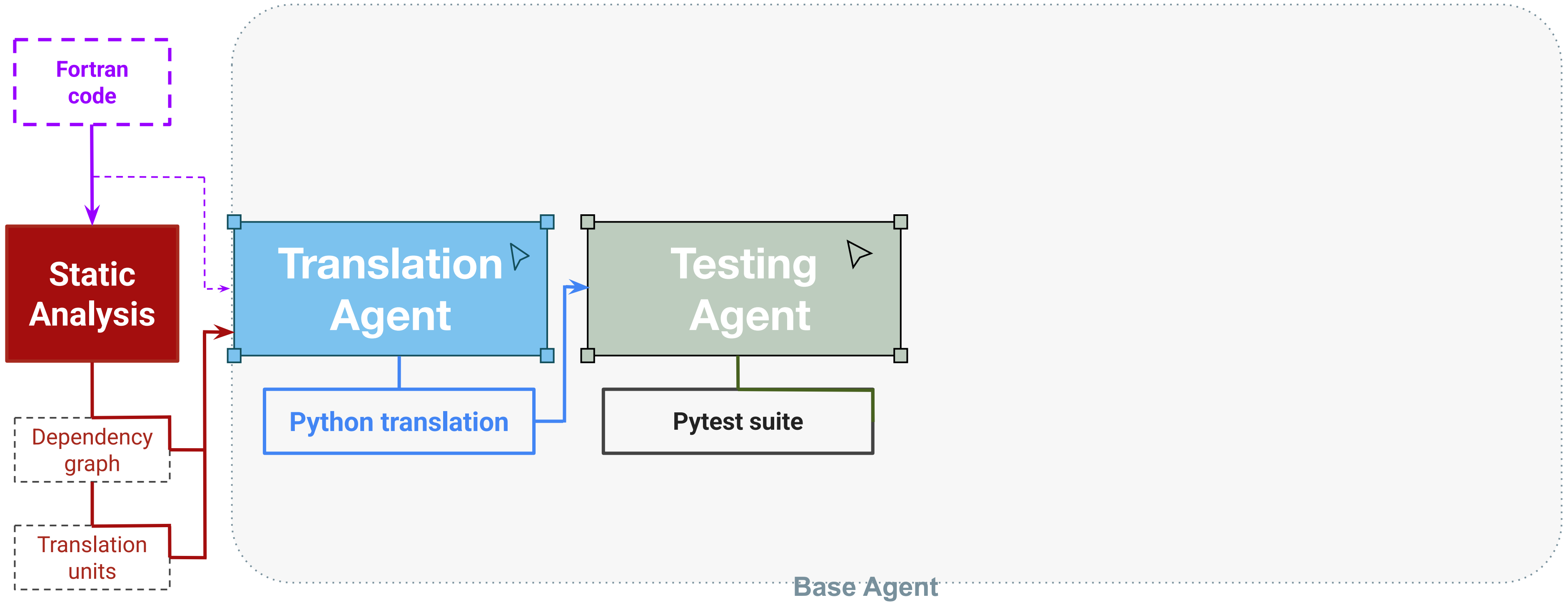
1. Array Mutation Incompatibility
   - Legacy Fortran makes extensive use of mutating operations
   - Requires architectural redesign, not just translation
2. Memory Explosion
   - "Saving all intermediate steps requires prohibitively large RAM"
   - Checkpointing needed, balancing memory vs. recomputation
   - Trade-off complexity compounds with model size
3. Chaotic System Characteristics
   - Gradients "orders of magnitude too large" from ill-conditioned Jacobians
   - Exponential error accumulation in long integrations
   - Requires specialized numerical stabilization
4. Stiff Differential Equations
   - Timescale differences (seconds to years) cause AD errors
   - Standard reverse-mode AD fails for many ESM components

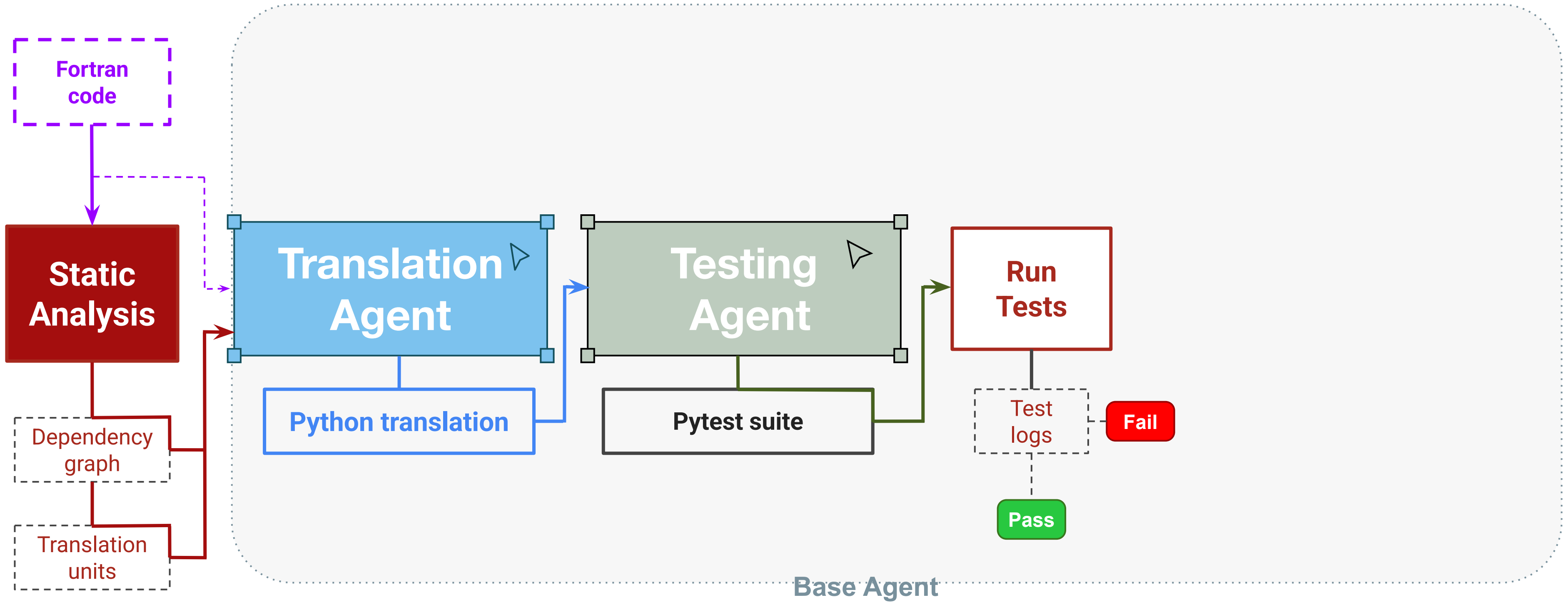
Reference: Gelbrecht et al. GMD 2023 "Differentiable programming for Earth system modeling"

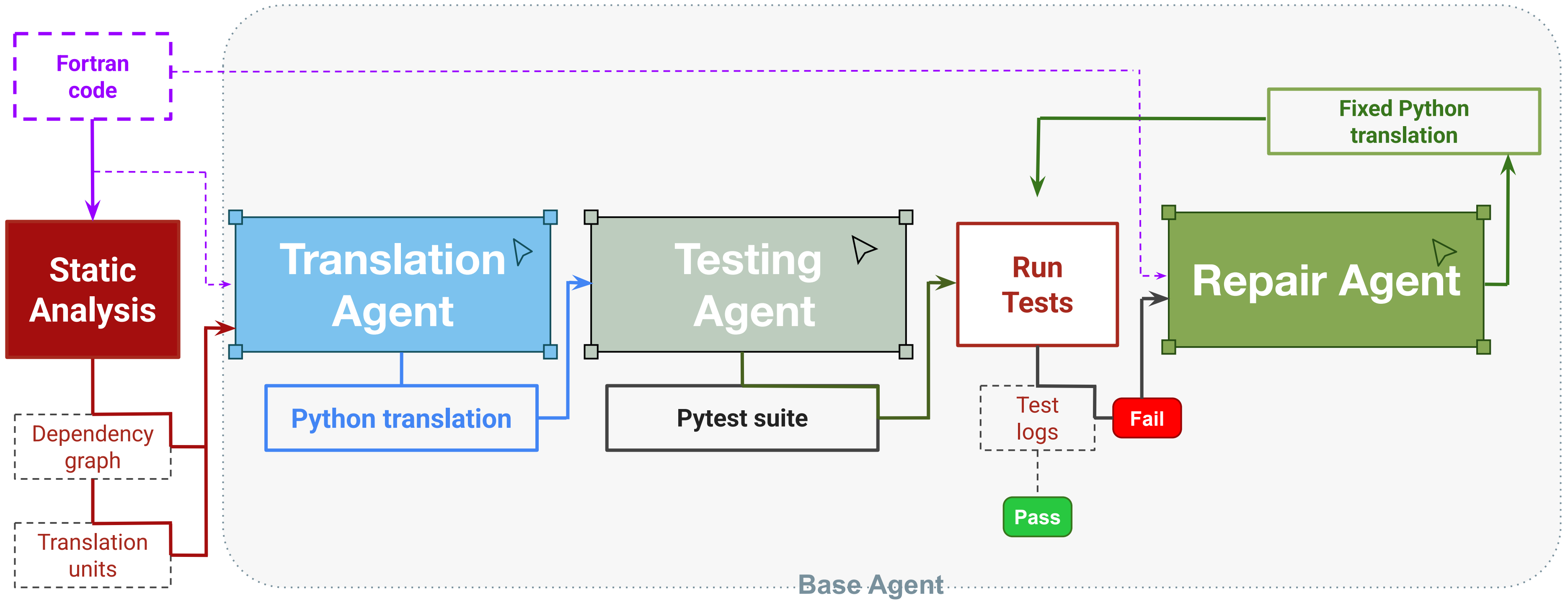

Fortran code
Static Analysis
Dependency graph
Translation units
Translation Agent
Python translation
Testing Agent
Pytest suite
Run Tests
Test logs
Fail
Pass
Repair Agent
Fixed Python translation
Base Agent


Fortran code
Static Analysis
Dependency graph
Translation units


Fortran code
Static Analysis
Dependency graph
Translation units
Translation Agent
Python translation
Base Agent


Fortran code
Static Analysis
Dependency graph
Translation units
Translation Agent
Python translation
Testing Agent
Pytest suite
Base Agent


Fortran code
Static Analysis
Dependency graph
Translation units
Translation Agent
Python translation
Testing Agent
Pytest suite
Run Tests
Test logs
Fail
Pass
Base Agent


Fortran code
Static Analysis
Dependency graph
Translation units
Translation Agent
Python translation
Testing Agent
Pytest suite
Run Tests
Test logs
Fail
Pass
Repair Agent
Fixed Python translation
Base Agent

Static Analysis

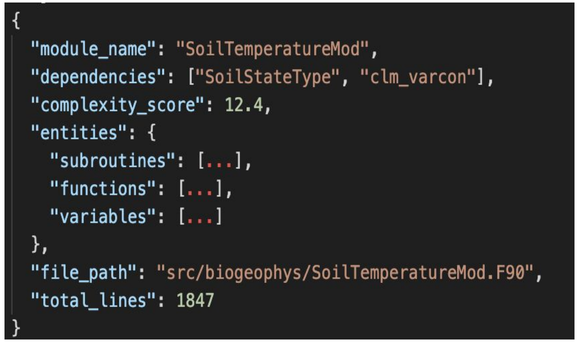
Tool: Fortran-Analyzer (Custom Framework) using **Fparse** and **networkx**

github.com/AyaLahlou/Fortran-Analyzer

- Generic Fortran parsing (F77, F90, F95, F2003, F2008)
- Dependency analysis: Module dependency graphs, circular dependency detection
- Translation unit decomposition: Break large procedures into manageable chunks
- Call graph generation: Visualize relationships between modules/procedures
- Multiple output formats: JSON, YAML, GraphML, interactive HTML

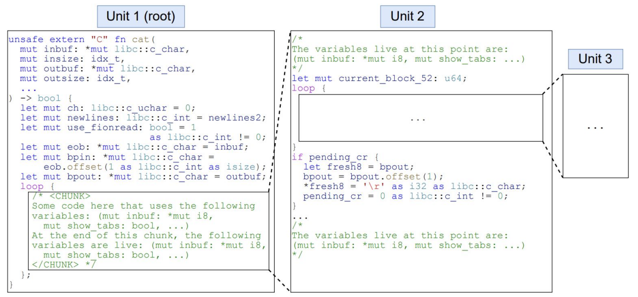
Outputs:

1. Module dependencies



```
{
  "module_name": "SoilTemperatureMod",
  "dependencies": ["SoilStateType", "clm_varcon"],
  "complexity_score": 12.4,
  "entities": {
    "subroutines": [...],
    "functions": [...],
    "variables": [...]
  },
  "file_path": "src/biogeophys/SoilTemperatureMod.F90",
  "total_lines": 1847
}
```

2. decomposed translation units

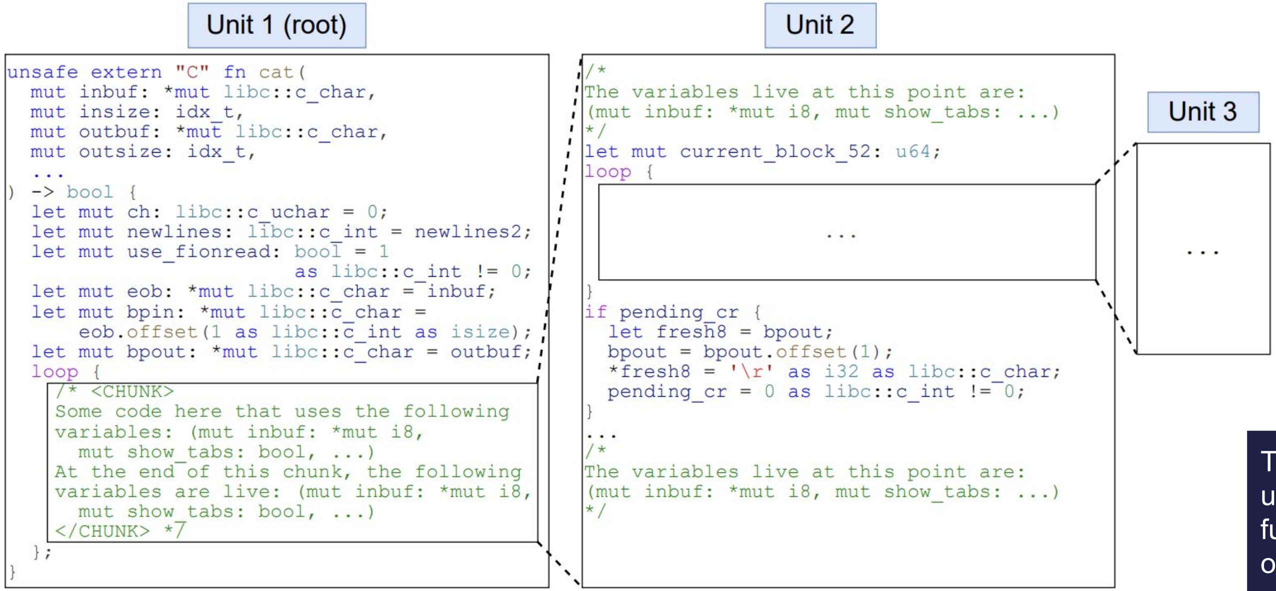


```
unsafe extern "C" fn cat(
  mut inbuf: *mut libc::c_char,
  mut insize: idx_t,
  mut outbuf: *mut libc::c_char,
  mut outsize: idx_t,
  ...
) -> bool {
  let mut ch: libc::c_uchar = 0;
  let mut newlines: libc::c_int = newlines2;
  let mut use_fionread: bool = 1
                          as libc::c_int != 0;
  let mut eob: *mut libc::c_char = inbuf;
  let mut bpin: *mut libc::c_char =
      eob.offset(1 as libc::c_int as isize);
  let mut bpout: *mut libc::c_char = outbuf;
  loop {
    /* <CHUNK>
    Some code here that uses the following
    variables: (mut inbuf: *mut i8,
      mut show_tabs: bool, ...)
    At the end of this chunk, the following
    variables are live: (mut inbuf: *mut i8,
      mut show_tabs: bool, ...)
    </CHUNK> */
  };
}
```

```
/*
The variables live at this point are:
(mut inbuf: *mut i8, mut show_tabs: ...)
*/
let mut current_block_52: u64;
loop {
  ...
}
if pending_cr {
  let fresh8 = bpout;
  bpout = bpout.offset(1);
  *fresh8 = '\r' as i32 as libc::c_char;
  pending_cr = 0 as libc::c_int != 0;
}
...
/*
The variables live at this point are:
(mut inbuf: *mut i8, mut show_tabs: ...)
*/
```

```
...
```

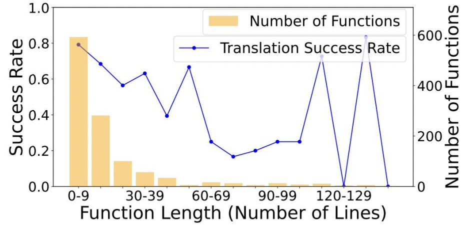
Decomposing Translation Units



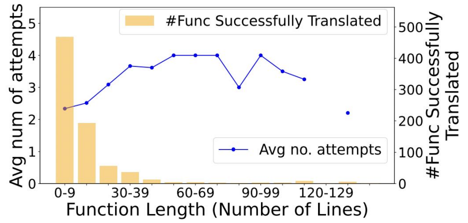
```
unsafe extern "C" fn cat(
  mut inbuf: *mut libc::c_char,
  mut insize: idx_t,
  mut outbuf: *mut libc::c_char,
  mut outsize: idx_t,
  ...
) -> bool {
  let mut ch: libc::c_uchar = 0;
  let mut newlines: libc::c_int = newlines2;
  let mut use_fionread: bool = 1
                              as libc::c_int != 0;
  let mut eob: *mut libc::c_char = inbuf;
  let mut bpin: *mut libc::c_char =
      eob.offset(1 as libc::c_int as isize);
  let mut bpout: *mut libc::c_char = outbuf;
  loop {
    /* <CHUNK>
    Some code here that uses the following
    variables: (mut inbuf: *mut i8,
      mut show_tabs: bool, ...)
    At the end of this chunk, the following
    variables are live: (mut inbuf: *mut i8,
      mut show_tabs: bool, ...)
    </CHUNK> */
  };
}
```

```
/*
The variables live at this point are:
(mut inbuf: *mut i8, mut show_tabs: ...)
*/
let mut current_block_52: u64;
loop {
                    ...
}
if pending_cr {
  let fresh8 = bpout;
  bpout = bpout.offset(1);
  *fresh8 = '\r' as i32 as libc::c_char;
  pending_cr = 0 as libc::c_int != 0;
}
...
/*
The variables live at this point are:
(mut inbuf: *mut i8, mut show_tabs: ...)
*/
```

```
...
```



(a) Translation Success Rate versus Function Length

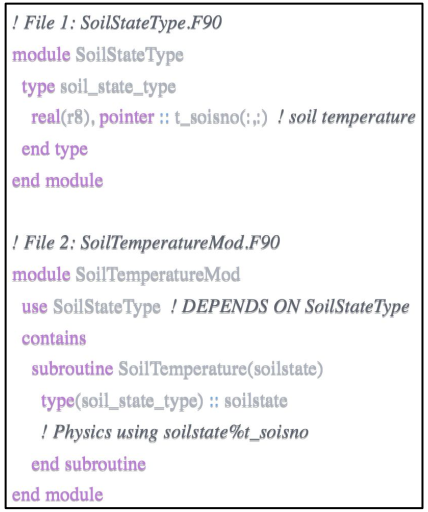


(b) #Attempts Required to Translate a Function, vs Length

Translation Unit 1 is the "root" unit that represents the function as a whole, and all other units are nested within it. The units are annotated with comments showing live-in and live-out information.

Reference: Nitin et al. 2025 C2SAFERRUST (arXiv:2501.14257)

Dependency-Ordered Translation

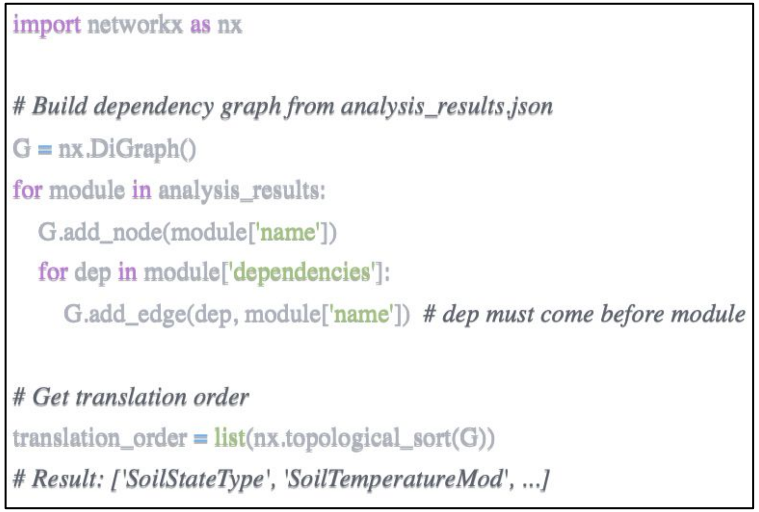


```
! File 1: SoilStateType.F90
module SoilStateType
  type soil_state_type
    real(r8), pointer :: t_soisno(:,:)  ! soil temperature
  end type
end module

! File 2: SoilTemperatureMod.F90
module SoilTemperatureMod
  use SoilStateType  ! DEPENDS ON SoilStateType
  contains
    subroutine SoilTemperature(soilstate)
      type(soil_state_type) :: soilstate
      ! Physics using soilstate%t_soisno
    end subroutine
end module
```

If translated in the wrong order:

- SoilTemperatureMod translated first, the code references undefined types
- LLM must guess structure of soil_state_type -> **Hallucination risk**

Topological Sorting



```
import networkx as nx

# Build dependency graph from analysis_results.json
G = nx.DiGraph()
for module in analysis_results:
    G.add_node(module['name'])
    for dep in module['dependencies']:
        G.add_edge(dep, module['name'])  # dep must come before module

# Get translation order
translation_order = list(nx.topological_sort(G))
# Result: ['SoilStateType', 'SoilTemperatureMod', ...]
```

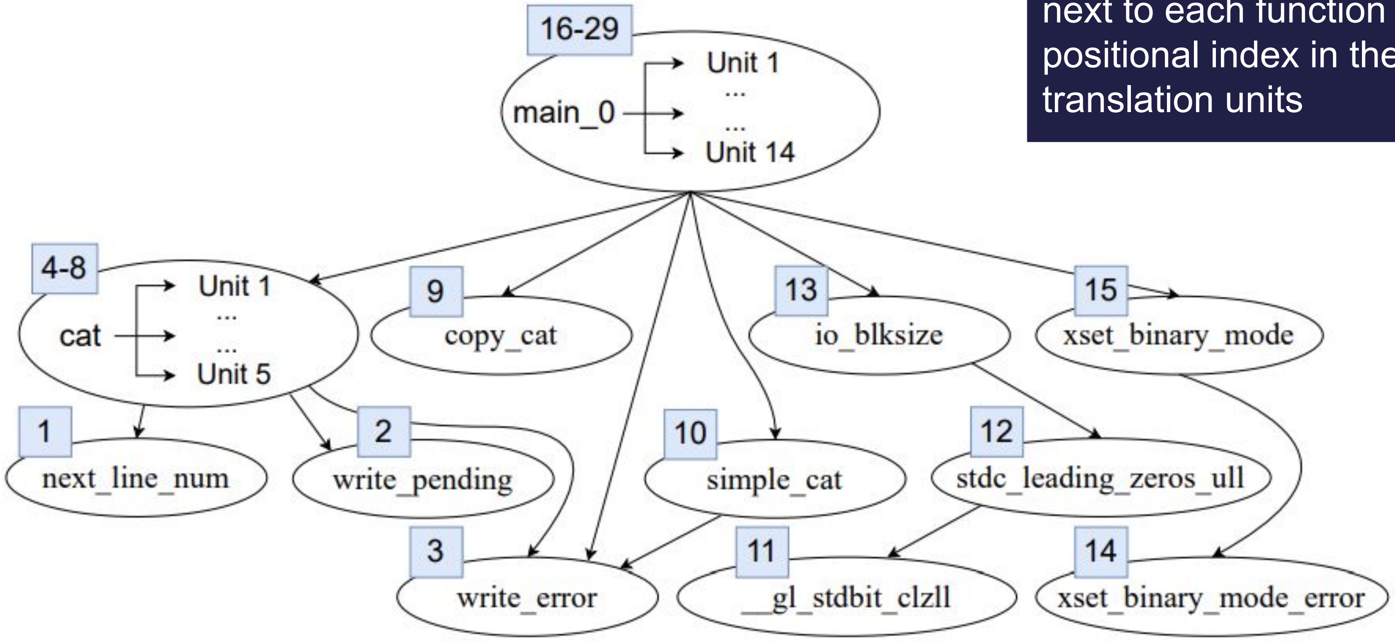
- Dependencies always translated first → No undefined references
- Context accumulates → Later translation units see previously translated code

Reduces LLM errors by 30.8%

Reference: NetworkX topological sort, RustRepoTrans benchmark (arXiv 2411.13990)

Dependency-Ordered Translation

Each node denotes a function, and the edges denote calls. The functions are sorted in a reverse topological order. The numbers next to each function denote their positional index in the ordering of translation units



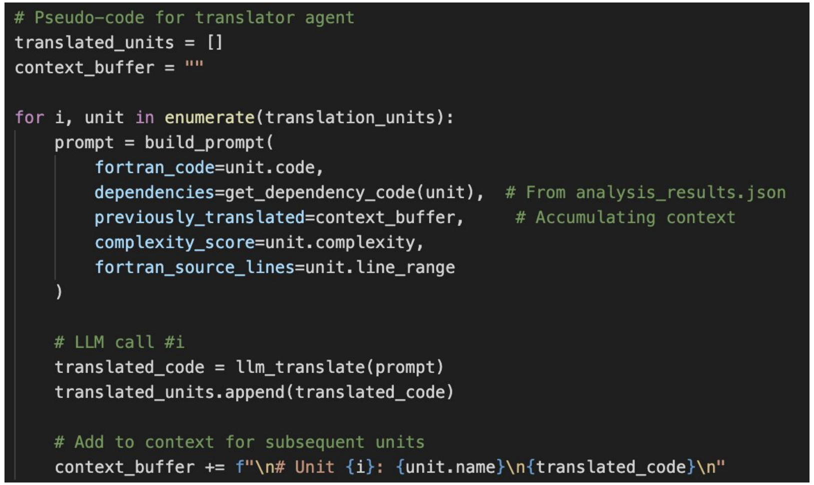
Reference: Nitin et al. 2025 C2SAFERRUST (arXiv:2501.14257)

Translation Agent

Iterative Context Accumulation

N+1 LLM Calls

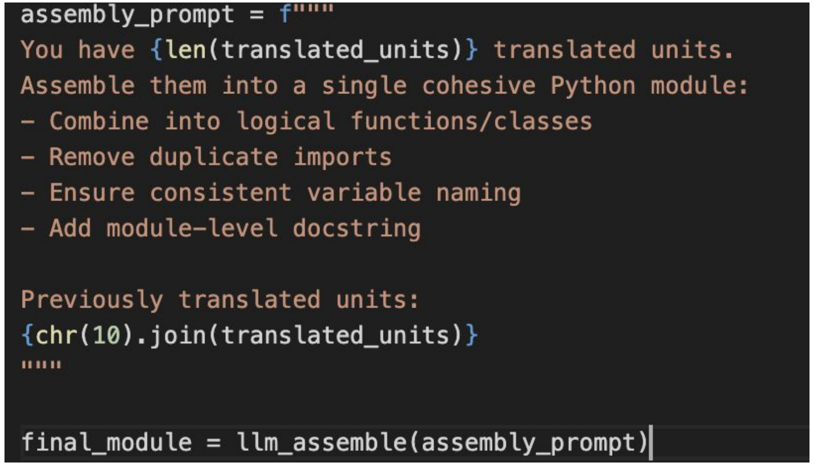
Step 1: Translate Each Unit Independently (N calls)



```python
# Pseudo-code for translator agent
translated_units = []
context_buffer = ""

for i, unit in enumerate(translation_units):
    prompt = build_prompt(
        fortran_code=unit.code,
        dependencies=get_dependency_code(unit),  # From analysis_results.json
        previously_translated=context_buffer,     # Accumulating context
        complexity_score=unit.complexity,
        fortran_source_lines=unit.line_range
    )

    # LLM call #i
    translated_code = llm_translate(prompt)
    translated_units.append(translated_code)

    # Add to context for subsequent units
    context_buffer += f"\n# Unit {i}: {unit.name}\n{translated_code}\n"
```

Step 2: Final module assembly

(simplified prompt for illustration)



```python
assembly_prompt = f"""
You have {len(translated_units)} translated units.
Assemble them into a single cohesive Python module:
- Combine into logical functions/classes
- Remove duplicate imports
- Ensure consistent variable naming
- Add module-level docstring

Previously translated units:
{chr(10).join(translated_units)}
"""

final_module = llm_assemble(assembly_prompt)
```

Example for SoilTemperatureMod:

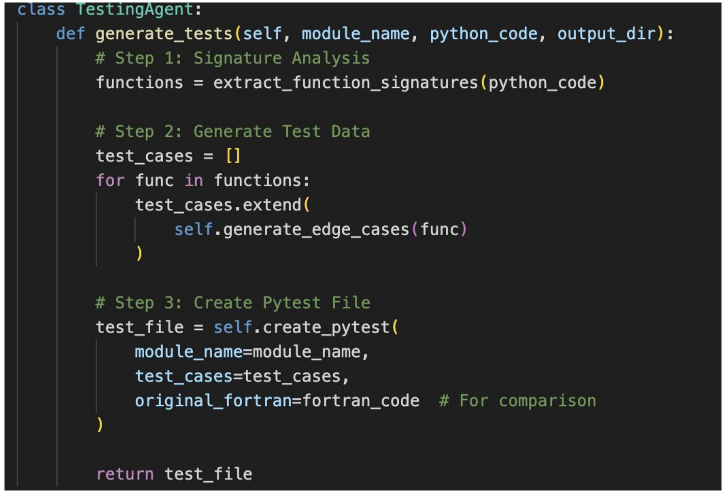
- 8 translation units → 8 unit translation calls
- 1 assembly call
- **Total: 9 API calls × $0.062 = ~$0.56 per module**

Testing Agent

- When legacy unit tests are unavailable
- Builds test suite for future development

Targets:

- Typical operating conditions
- Edge cases (e.g. frozen, dry, saturated)
- Boundary conditions (e.g. 0°C phase transition)
- Physical constraints (e.g. positive conductivity)



```python
class TestingAgent:
    def generate_tests(self, module_name, python_code, output_dir):
        # Step 1: Signature Analysis
        functions = extract_function_signatures(python_code)

        # Step 2: Generate Test Data
        test_cases = []
        for func in functions:
            test_cases.extend(
                self.generate_edge_cases(func)
            )

        # Step 3: Create Pytest File
        test_file = self.create_pytest(
            module_name=module_name,
            test_cases=test_cases,
            original_fortran=fortran_code  # For comparison
        )

        return test_file
```

Repair Agent

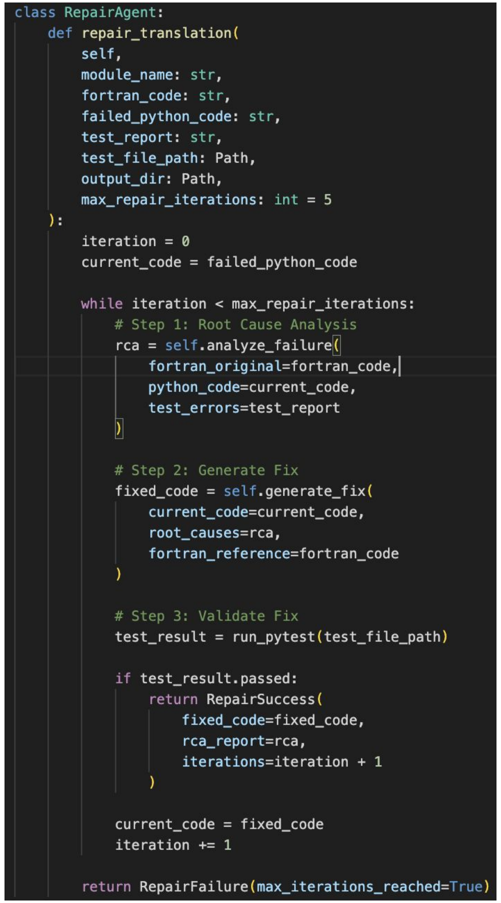
Why?

- 45-73% first-pass translation success rate
  - ➢ 25-55% need repair

Iterative Repair process increases accuracy

- Initial translation: 45-73% success
- After 1 repair iteration: 65-85% success
- After 2-3 iterations: **80-92% success**

(reference: TRANSAGENT arXiv 2409.19894)

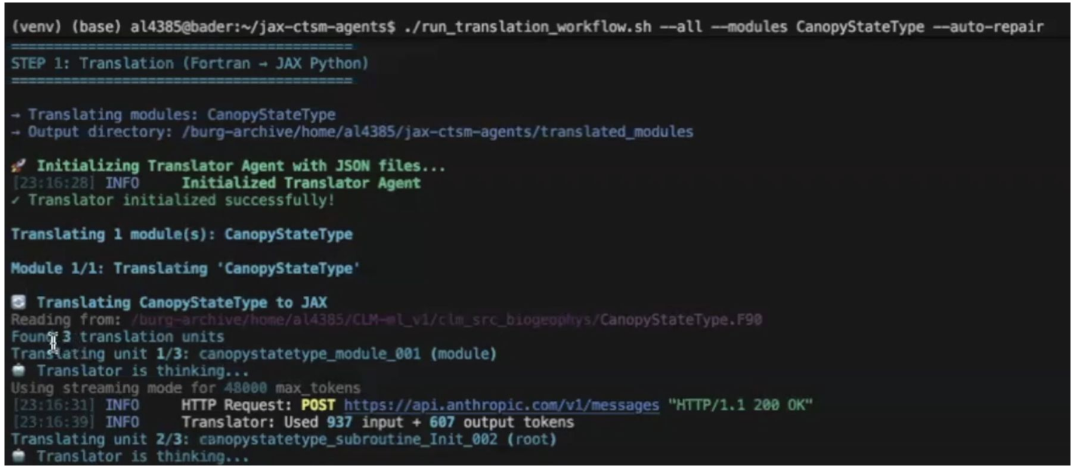


```python
class RepairAgent:
    def repair_translation(
        self,
        module_name: str,
        fortran_code: str,
        failed_python_code: str,
        test_report: str,
        test_file_path: Path,
        output_dir: Path,
        max_repair_iterations: int = 5
    ):
        iteration = 0
        current_code = failed_python_code

        while iteration < max_repair_iterations:
            # Step 1: Root Cause Analysis
            rca = self.analyze_failure(
                fortran_original=fortran_code,
                python_code=current_code,
                test_errors=test_report
            )

            # Step 2: Generate Fix
            fixed_code = self.generate_fix(
                current_code=current_code,
                root_causes=rca,
                fortran_reference=fortran_code
            )

            # Step 3: Validate Fix
            test_result = run_pytest(test_file_path)

            if test_result.passed:
                return RepairSuccess(
                    fixed_code=fixed_code,
                    rca_report=rca,
                    iterations=iteration + 1
                )

            current_code = fixed_code
            iteration += 1

        return RepairFailure(max_iterations_reached=True)
```

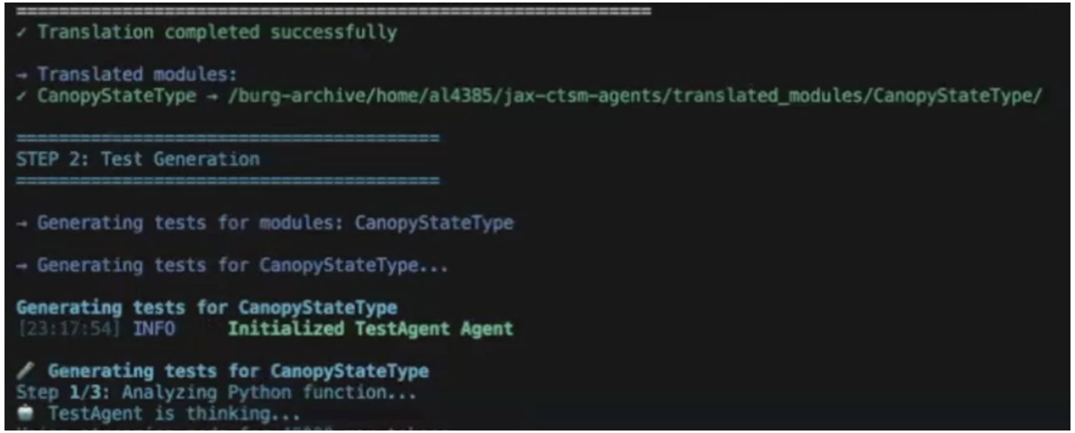
Running the End-to-End pipeline

Example: CanopyStateType module

Translation Agent



```
(venv) (base) al4385@bader:~/jax-ctsm-agents$ ./run_translation_workflow.sh --all --modules CanopyStateType --auto-repair
============================================
STEP 1: Translation (Fortran → JAX Python)
============================================

→ Translating modules: CanopyStateType
→ Output directory: /burg-archive/home/al4385/jax-ctsm-agents/translated_modules

🚀 Initializing Translator Agent with JSON files...
[23:16:28] INFO      Initialized Translator Agent
✓ Translator initialized successfully!

Translating 1 module(s): CanopyStateType

Module 1/1: Translating 'CanopyStateType'

🔄 Translating CanopyStateType to JAX
Reading from: /burg-archive/home/al4385/CLM-ml_v1/clm_src_biogeophys/CanopyStateType.F90
Found 3 translation units
Translating unit 1/3: canopystatetype_module_001 (module)
🤖 Translator is thinking...
Using streaming mode for 48000 max_tokens
[23:16:31] INFO      HTTP Request: POST https://api.anthropic.com/v1/messages "HTTP/1.1 200 OK"
[23:16:39] INFO      Translator: Used 937 input + 607 output tokens
Translating unit 2/3: canopystatetype_subroutine_Init_002 (root)
🤖 Translator is thinking...
```

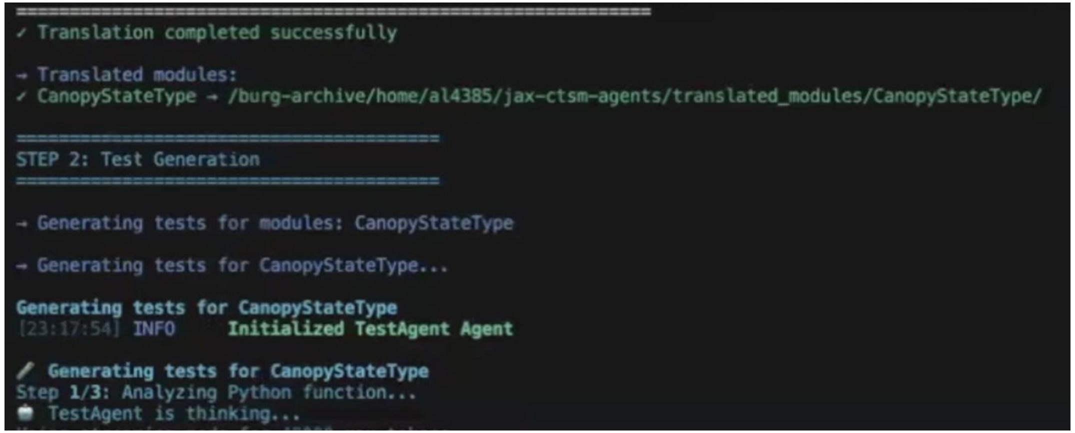


```
================================================================
✓ Translation completed successfully

→ Translated modules:
✓ CanopyStateType → /burg-archive/home/al4385/jax-ctsm-agents/translated_modules/CanopyStateType/

============================================
STEP 2: Test Generation
============================================

→ Generating tests for modules: CanopyStateType

→ Generating tests for CanopyStateType...

Generating tests for CanopyStateType
[23:17:54] INFO      Initialized TestAgent Agent

✏ Generating tests for CanopyStateType
Step 1/3: Analyzing Python function...
🤖 TestAgent is thinking...
```

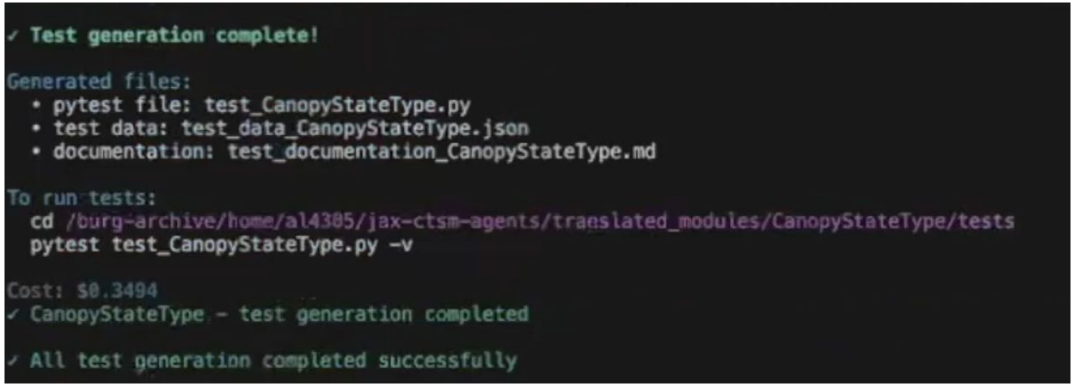
Running the End-to-End pipeline

Example: CanopyStateType module

Testing Agent



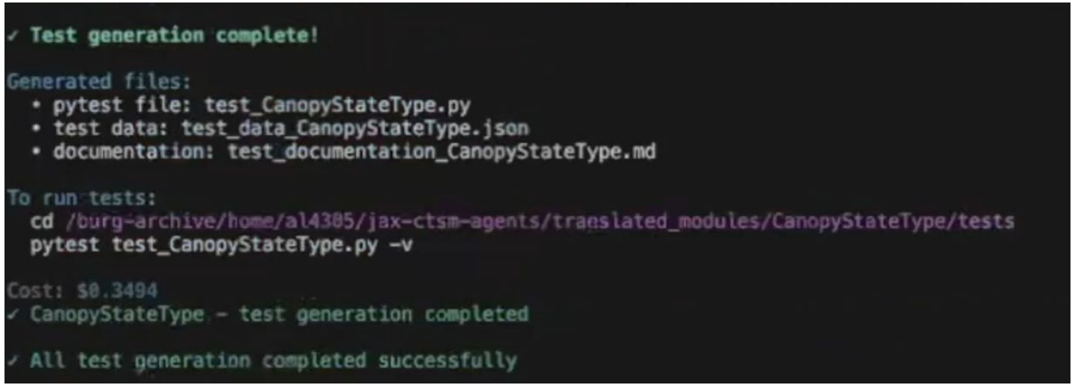
```
============================================================
✓ Translation completed successfully

→ Translated modules:
✓ CanopyStateType → /burg-archive/home/al4385/jax-ctsm-agents/translated_modules/CanopyStateType/

========================================
STEP 2: Test Generation
========================================

→ Generating tests for modules: CanopyStateType

→ Generating tests for CanopyStateType...

Generating tests for CanopyStateType
[23:17:54] INFO     Initialized TestAgent Agent

 Generating tests for CanopyStateType
Step 1/3: Analyzing Python function...
 TestAgent is thinking...
```



```
✓ Test generation complete!

Generated files:
  • pytest file: test_CanopyStateType.py
  • test data: test_data_CanopyStateType.json
  • documentation: test_documentation_CanopyStateType.md

To run tests:
  cd /burg-archive/home/al4385/jax-ctsm-agents/translated_modules/CanopyStateType/tests
  pytest test_CanopyStateType.py -v

Cost: $0.3494
✓ CanopyStateType - test generation completed

✓ All test generation completed successfully
```

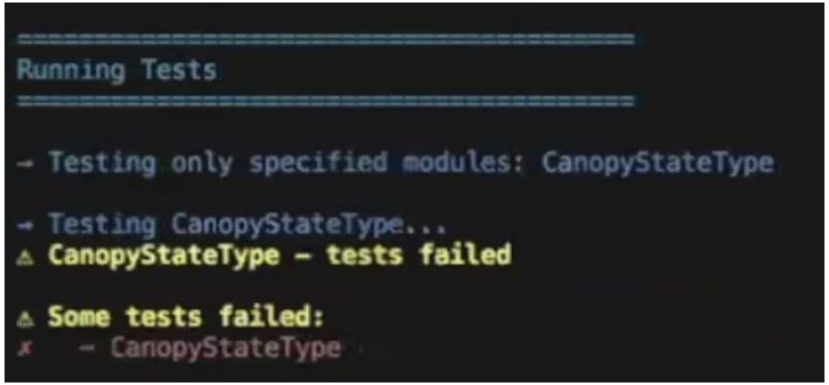
Running the End-to-End pipeline

Example: CanopyStateType module



```
✓ Test generation complete!

Generated files:
  • pytest file: test_CanopyStateType.py
  • test data: test_data_CanopyStateType.json
  • documentation: test_documentation_CanopyStateType.md

To run tests:
  cd /burg-archive/home/al4385/jax-ctsm-agents/translated_modules/CanopyStateType/tests
  pytest test_CanopyStateType.py -v

Cost: $0.3494
✓ CanopyStateType - test generation completed

✓ All test generation completed successfully
```

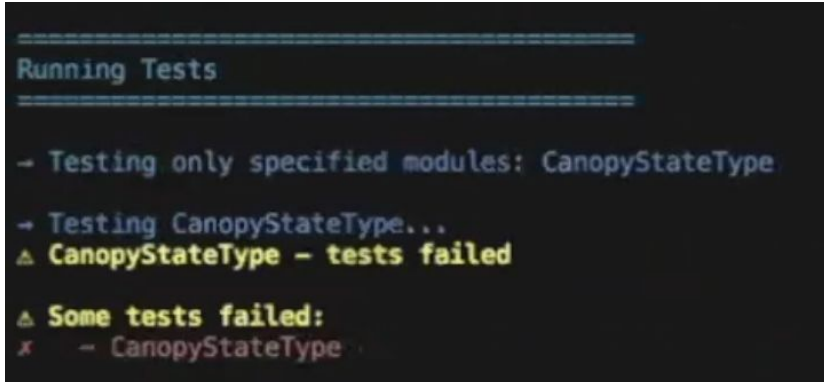
Run Tests



```
==================================================
Running Tests
==================================================

→ Testing only specified modules: CanopyStateType

→ Testing CanopyStateType...
⚠ CanopyStateType - tests failed

⚠ Some tests failed:
✗   - CanopyStateType
```

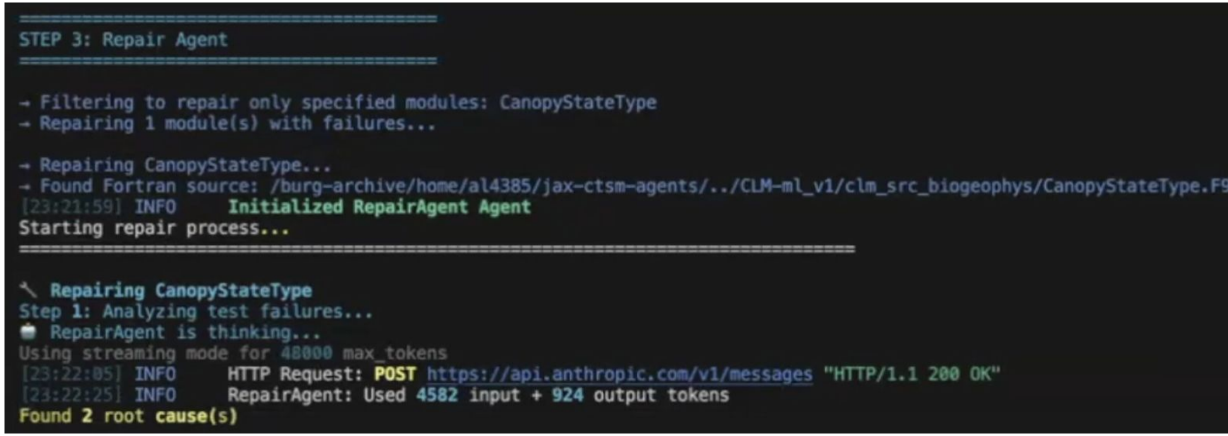
Running the End-to-End pipeline

Example: CanopyStateType module



```
==========================================
Running Tests
==========================================

→ Testing only specified modules: CanopyStateType

→ Testing CanopyStateType...
⚠ CanopyStateType – tests failed

⚠ Some tests failed:
✗   – CanopyStateType
```

Repair Agent

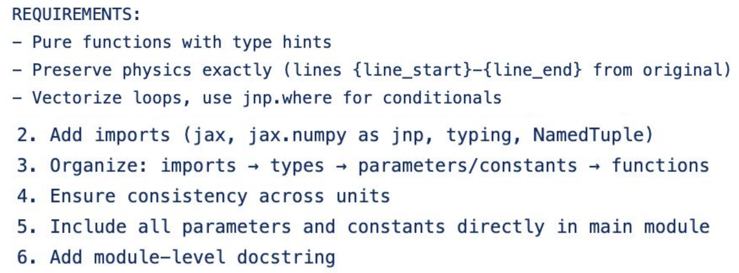


```
==========================================
STEP 3: Repair Agent
==========================================

→ Filtering to repair only specified modules: CanopyStateType
→ Repairing 1 module(s) with failures...

→ Repairing CanopyStateType...
→ Found Fortran source: /burg-archive/home/al4385/jax-ctsm-agents/../CLM-ml_v1/clm_src_biogeophys/CanopyStateType.F9
[23:21:59] INFO     Initialized RepairAgent Agent
Starting repair process...
================================================================================

🔧 Repairing CanopyStateType
Step 1: Analyzing test failures...
🤖 RepairAgent is thinking...
Using streaming mode for 48000 max_tokens
[23:22:05] INFO     HTTP Request: POST https://api.anthropic.com/v1/messages "HTTP/1.1 200 OK"
[23:22:25] INFO     RepairAgent: Used 4582 input + 924 output tokens
Found 2 root cause(s)
```

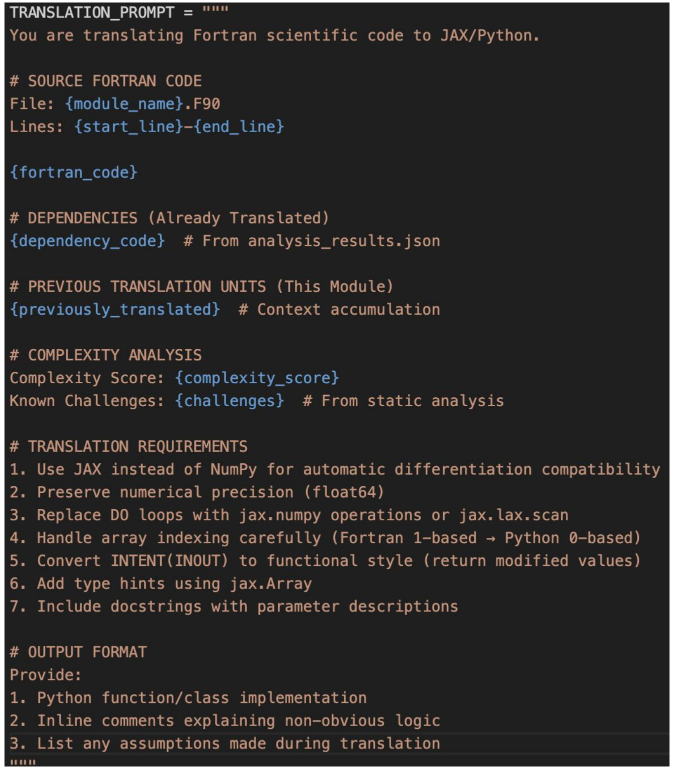
Prompt Engineering

Context-Augmented Prompt Structure:

Effective Prompts = 3× Better Results



```
REQUIREMENTS:
- Pure functions with type hints
- Preserve physics exactly (lines {line_start}-{line_end} from original)
- Vectorize loops, use jnp.where for conditionals
 2. Add imports (jax, jax.numpy as jnp, typing, NamedTuple)
 3. Organize: imports → types → parameters/constants → functions
 4. Ensure consistency across units
 5. Include all parameters and constants directly in main module
 6. Add module-level docstring
```

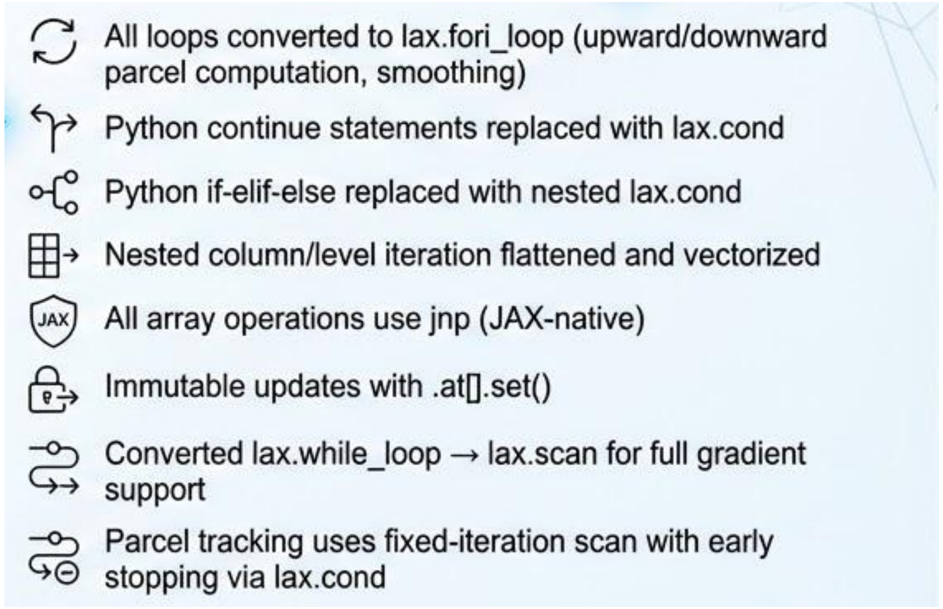


```
TRANSLATION_PROMPT = """
You are translating Fortran scientific code to JAX/Python.

# SOURCE FORTRAN CODE
File: {module_name}.F90
Lines: {start_line}-{end_line}

{fortran_code}

# DEPENDENCIES (Already Translated)
{dependency_code}  # From analysis_results.json

# PREVIOUS TRANSLATION UNITS (This Module)
{previously_translated}  # Context accumulation

# COMPLEXITY ANALYSIS
Complexity Score: {complexity_score}
Known Challenges: {challenges}  # From static analysis

# TRANSLATION REQUIREMENTS
1. Use JAX instead of NumPy for automatic differentiation compatibility
2. Preserve numerical precision (float64)
3. Replace DO loops with jax.numpy operations or jax.lax.scan
4. Handle array indexing carefully (Fortran 1-based → Python 0-based)
5. Convert INTENT(INOUT) to functional style (return modified values)
6. Add type hints using jax.Array
7. Include docstrings with parameter descriptions

# OUTPUT FORMAT
Provide:
1. Python function/class implementation
2. Inline comments explaining non-obvious logic
3. List any assumptions made during translation
"""
```

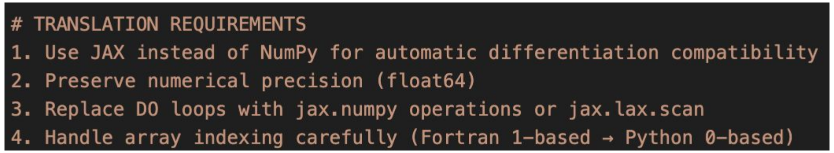
Reference: K3Trans triple knowledge augmentation (arXiv), leap-stc/jax-ctsm-agents prompts/translation_prompts.py

Can we achieve differentiability?

Differentiable Translation Example: Mixing length module in CLUBB



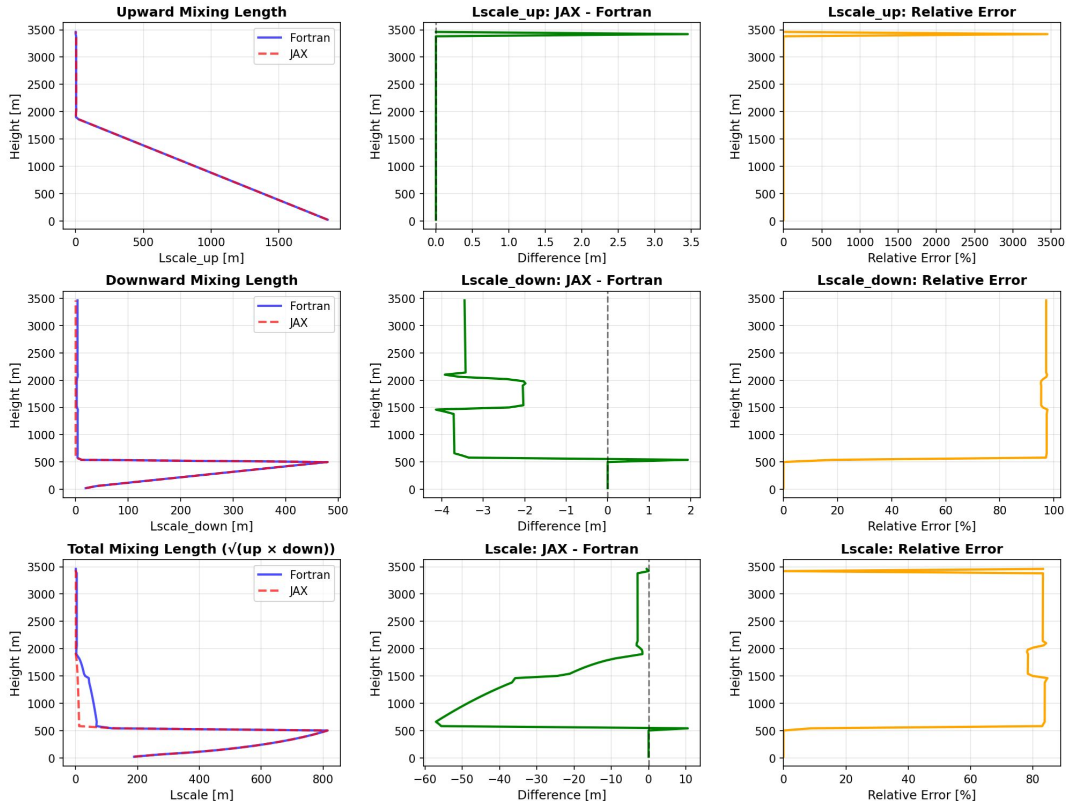
- All loops converted to lax.fori_loop (upward/downward parcel computation, smoothing)
- Python continue statements replaced with lax.cond
- Python if-elif-else replaced with nested lax.cond
- Nested column/level iteration flattened and vectorized
- All array operations use jnp (JAX-native)
- Immutable updates with .at[].set()
- Converted lax.while_loop → lax.scan for full gradient support
- Parcel tracking uses fixed-iteration scan with early stopping via lax.cond



```
# TRANSLATION REQUIREMENTS
1. Use JAX instead of NumPy for automatic differentiation compatibility
2. Preserve numerical precision (float64)
3. Replace DO loops with jax.numpy operations or jax.lax.scan
4. Handle array indexing carefully (Fortran 1-based → Python 0-based)
```

Can we achieve differentiability?

Differentiable Translation Example: Mixing length module in CLUBB

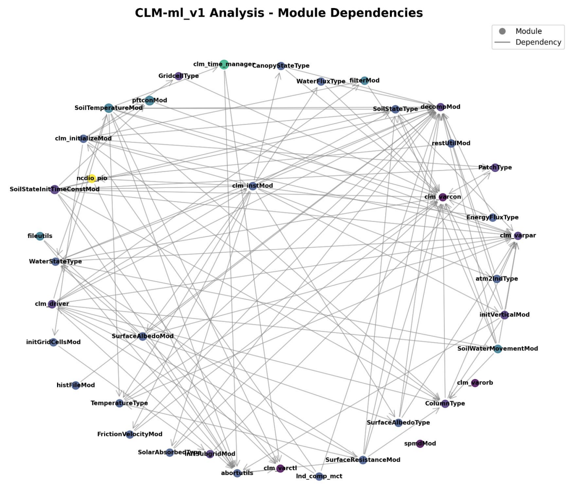


CLM multilayer canopy model

Computational Cost:
- 5-10× vertical resolution increase vs. big-leaf
- Prime candidate for GPU acceleration with JAX

Practical Advantages:
- Modularity
- Standalone capability: Can run with offline driver for tower sites
- Validation data available

https://github.com/AyaLahlou/clm-ml-jax/

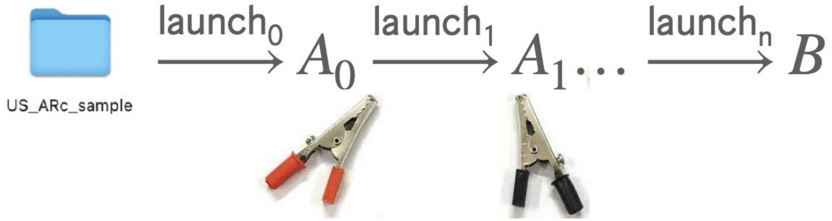


Work in progress… Results coming soon

References: Bonan, G.B. et al. 2021

Lessons Learned from our Automatic Translation Project

- Pre-modularising is key
- Write **very specific prompts** for LLM agents
- Test-driven translation: tests guide validation at each step
    - Use ‘crocodile clips’ higher-order function to get real input-output data from Fortran


US_ARc_sample
$\xrightarrow{\text{launch}_0} A_0 \xrightarrow{\text{launch}_1} A_1 \ldots \xrightarrow{\text{launch}_n} B$

- Single test suite that can target Fortran or Python
- Property based testing

Next steps

- Offline tests & Validation of JAX CLM-ml_v2
- Coupled runs with CLM 5 & long term simulations
- Differentiability tests
- Performance Benchmark: Fortran, JAX CPU, JAX GPU
- ... A full CLM6 refactoring?

In Summary

Urgent need for ESM modernization

Legacy ESMs face major modernization barriers from 100k+ Fortran code lines lacking **differentiability** and **modularisation**, with critical gaps in unit testing and fundamental **architecture incompatibilities preventing ML integration and systematic parameter optimization.**

Fast Cost effective AI agents

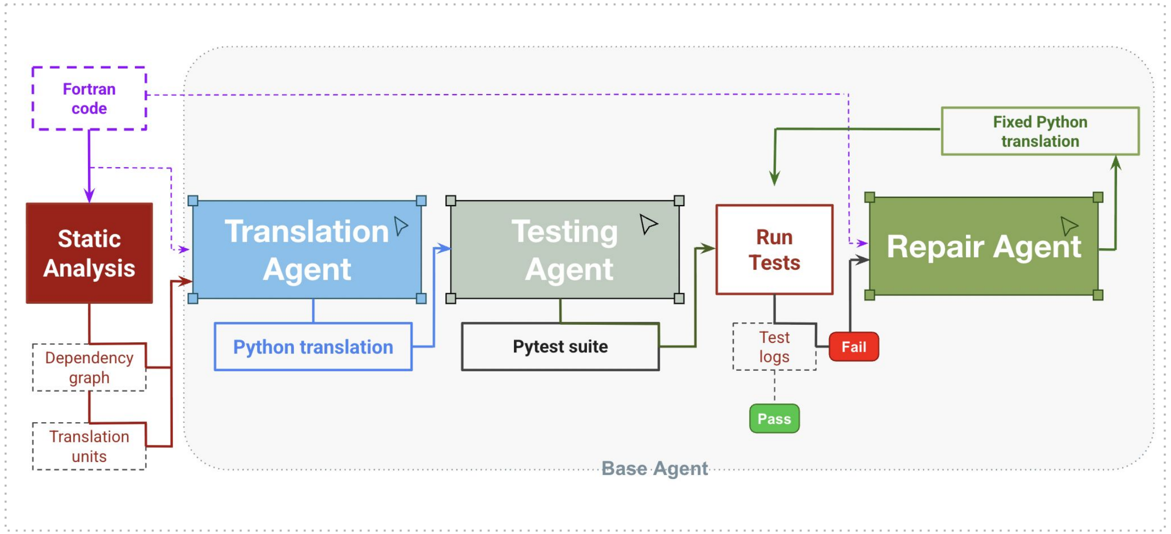
AI-assisted translation achieving **75-85% time savings** and ROI through productivity gains, breaking even within days while **reducing project costs by 80-90%** compared to developer time.

Context-Aware & Accurate Refactoring

Dependency context and topological sorting improves LLM code translation quality by 19-40%, with context-aware approaches (including relevant usage examples) achieving 92% compilation coverage.

25-50x GPU speedup

Refactored JAX CLM-ml should achieve **25-50x GPU speedup over CPU**, making previously computationally prohibitive 10 canopy layer vertical resolution calculations practical for long climate simulations.



Neurosymbolic Translation Pipeline from Fortran to JAX

Thank you for your time!

Any questions/suggestions?