STUNIR Pipeline Audit Report

Date: 2026-01-30
Auditor: STUNIR Team
Purpose: Comprehensive analysis of implementation status across all four pipelines

---

Executive Summary

Metric	Value	Status
Total Categories	24	✅ Defined
SPARK Files	73	✅ Complete (100%)
Python Files	121	⚠️ Partial (~70% depth)
Rust Files	0	❌ Not Started (0%)
Haskell Files	0	❌ Not Started (0%)
Confluence Score	N/A	⚠️ Test suite not implemented

---

1. Category-by-Category Breakdown

#	Category	SPARK	Python	Rust	Haskell	Status
1	asm	2	1	0	0	⚠️ Rust/ Haskell missing
2	asp	2	3	0	0	⚠️ Rust/ Haskell missing
3	assembly	6	6	0	0	⚠️ Rust/ Haskell missing
4	beam	2	4	0	0	⚠️ Rust/ Haskell missing
5	business	2	3	0	0	⚠️ Rust/ Haskell missing
6	bytecode	2	1	0	0	⚠️ Rust/ Haskell missing
7	constraints	2	4	0	0	⚠️ Rust/ Haskell missing
8	embedded	3	1	0	0	⚠️ Rust/ Haskell missing
9	ex-pert_systems	2	4	0	0	⚠️ Rust/ Haskell missing
10	fpga	2	1	0	0	⚠️ Rust/ Haskell missing
11	functional	2	5	0	0	⚠️ Rust/ Haskell missing
12	gpu	2	1	0	0	⚠️ Rust/ Haskell missing

#	Category	SPARK	Python	Rust	Haskell	Status
13	grammar	2	7	0	0	⚠️ Rust/ Haskell missing
14	lexer	2	6	0	0	⚠️ Rust/ Haskell missing
15	lisp	18	26	0	0	⚠️ Rust/ Haskell missing
16	mobile	2	1	0	0	⚠️ Rust/ Haskell missing
17	oop	2	3	0	0	⚠️ Rust/ Haskell missing
18	parser	2	6	0	0	⚠️ Rust/ Haskell missing
19	planning	2	2	0	0	⚠️ Rust/ Haskell missing
20	polyglot	6	4	0	0	⚠️ Rust/ Haskell missing
21	prolog	2	25	0	0	⚠️ Rust/ Haskell missing
22	scientific	2	3	0	0	⚠️ Rust/ Haskell missing
23	systems	2	3	0	0	⚠️ Rust/ Haskell missing
24	wasm	2	1	0	0	⚠️ Rust/ Haskell missing

2. SPARK Pipeline Analysis (Reference Implementation)

Status: ✅ COMPLETE

File Count: 73 files

Structure:

```
targets/spark/
├── emitter_types.ads/adb (shared types)
├── assembly/
│   ├── arm/ (ARM/ARM64 emitters)
│   └── x86/ (x86/x64 emitters)
├── polyglot/
│   ├── c89/ (ANSI C89)
│   ├── c99/ (C99)
│   └── rust/ (Rust)
├── lisp/
│   ├── lisp_base.ads/adb (shared utilities)
│   ├── common_lisp/
│   ├── scheme/
│   ├── clojure/
│   ├── racket/
│   ├── emacs_lisp/
│   ├── guile/
│   ├── hy/
│   └── janet/
├── prolog/ (8 variants)
├── gpu/ (CUDA, ROCm, OpenCL, Metal, Vulkan)
├── wasm/
├── embedded/
└── ... (all 24 categories)
```

Strengths:

- ✅ DO-178C Level A compliant
- ✅ Formal verification with SPARK contracts
- ✅ Complete coverage of all 24 categories
- ✅ Shared type system (`emitter_types.ads/adb`)
- ✅ Consistent architecture across categories

Core Tools:

- ✅ `tools/spark/bin/stunir_spec_to_ir_main`
- ✅ `tools/spark/bin/stunir_ir_to_code_main`

3. Python Pipeline Analysis

Status: ⚠️ PARTIAL (Estimated ~70% depth)

File Count: 121 files

Structure:

```
targets/
├── __init__.py
├── asm/emitter.py
├── asp/
│   ├── clingo/emitter.py
│   ├── dlv/emitter.py
│   └── potassco/emitter.py
├── assembly/
│   ├── arm/emitter.py
│   └── x86/emitter.py
├── lisp/
│   ├── common_lisp/emitter.py
│   ├── scheme/emitter.py
│   ├── clojure/emitter.py
│   └── ... (8 variants)
├── prolog/ (8 variants)
└── ... (all 24 categories represented)
```

Strengths:

- ✅ All 24 categories have at least minimal implementation
- ✅ Good coverage of variant targets (Lisp, Prolog)
- ✅ Readable, well-documented code
- ✅ Native Python idioms

Weaknesses:

- ⚠️ Marked as "reference implementation" (not production-ready)
- ⚠️ Some emitters are minimal (e.g., embedded, GPU)
- ⚠️ Inconsistent depth across categories
- ⚠️ No formal verification

Core Tools:

- ⚠️ `tools/spec_to_ir.py` (marked as fallback)
- ⚠️ `tools/ir_to_code.py` (marked as fallback)

Recommendations:

1. Remove "reference implementation" warnings
2. Enhance minimal emitters (embedded, GPU, WASM)
3. Add comprehensive test coverage
4. Establish as production-ready alternative to SPARK

4. Rust Pipeline Analysis

Status: ❌ NOT STARTED (0%)

File Count: 0 files

Required Implementation:

```
tools/rust/
├── Cargo.toml
├── src/
│   ├── main.rs
│   ├── spec_to_ir.rs
│   ├── ir_to_code.rs
│   └── lib.rs
└── bin/
    ├── stunir_spec_to_ir
    └── stunir_ir_to_code

targets/rust/
├── emitter_types.rs (shared)
├── assembly/
│   ├── arm.rs
│   └── x86.rs
├── polyglot/
│   ├── c89.rs
│   ├── c99.rs
│   └── rust.rs
├── lisp/
│   ├── base.rs
│   ├── common_lisp.rs
│   ├── scheme.rs
│   └── ... (8 variants)
└── ... (all 24 categories)
```

Estimated Effort:

- **Core toolchain** (`spec_to_ir` , `ir_to_code`): 2-4 weeks
- **Representative emitters** (24 categories): 4-8 weeks
- **Complete implementations**: 3-6 months
- **Testing & verification**: 2-4 weeks

Key Considerations:

- Use `serde` for JSON parsing
- Use `sha2` for hashing
- Leverage Rust's type system for safety
- Create shared emitter traits
- Generate idiomatic Rust code

5. Haskell Pipeline Analysis

Status: ❌ NOT STARTED (0%)

File Count: 0 files

Required Implementation:

```
tools/haskell/
├── stunir-tools.cabal
├── src/
│   ├── STUNIR/
│   │   ├── SpecToIR.hs
│   │   ├── IRToCode.hs
│   │   ├── Types.hs
│   │   └── Hash.hs
│   └── Main.hs
└── dist/
    └── build/
        ├── stunir_spec_to_ir
        └── stunir_ir_to_code

targets/haskell/
├── STUNIR/
│   ├── Emitter/
│   │   ├── Types.hs (shared)
│   │   ├── Assembly/
│   │   │   ├── ARM.hs
│   │   │   └── X86.hs
│   │   ├── Polyglot/
│   │   │   ├── C89.hs
│   │   │   ├── C99.hs
│   │   │   └── Rust.hs
│   │   ├── Lisp/
│   │   │   ├── Base.hs
│   │   │   ├── CommonLisp.hs
│   │   │   └── ... (8 variants)
│   │   └── ... (all 24 categories)
```

Estimated Effort:

- **Core toolchain**: 2-4 weeks
- **Representative emitters**: 4-8 weeks
- **Complete implementations**: 3-6 months
- **Testing & verification**: 2-4 weeks

Key Considerations:

- Use `aeson` for JSON parsing
- Use `cryptonite` for hashing
- Leverage Haskell’s type system for correctness
- Use typeclasses for emitter abstraction
- Generate pure, functional code

6. Critical Gaps

6.1 Rust Pipeline

Impact: HIGH
Organizations Affected: Automotive, embedded systems, high-performance computing

Missing:
- ❌ Core toolchain (`spec_to_ir`, `ir_to_code`)
- ❌ All 24 category emitters
- ❌ Build configuration (Cargo.toml)
- ❌ Test infrastructure

Required for Confluence: CRITICAL

6.2 Haskell Pipeline

Impact: HIGH
Organizations Affected: Financial services, formal verification, research

Missing:
- ❌ Core toolchain (`spec_to_ir`, `ir_to_code`)
- ❌ All 24 category emitters
- ❌ Build configuration (.cabal)
- ❌ Test infrastructure

Required for Confluence: CRITICAL

6.3 Python Pipeline Enhancement

Impact: MEDIUM
Organizations Affected: All organizations using Python

Issues:
- ⚠️ "Reference implementation" stigma
- ⚠️ Minimal emitters for embedded, GPU, WASM
- ⚠️ Inconsistent feature depth

Required for Confluence: IMPORTANT

6.4 Confluence Test Suite

Impact: CRITICAL
Organizations Affected: ALL

Missing:
- ❌ Automated test framework
- ❌ Test vectors for all 24 categories
- ❌ Hash comparison tooling
- ❌ CI/CD integration

Required for Confluence: BLOCKER

---

7. Recommended Prioritization

Phase 1: Foundation (Weeks 1-4)

Goal: Enable basic confluence testing

1. ✅ **Document confluence requirements** (DONE)
2. **Create confluence test framework**
   - Test vectors (minimal, simple, complex)
   - Hash comparison script
   - CI/CD integration
3. **Rust core toolchain**
   - `spec_to_ir` implementation
   - `ir_to_code` implementation
   - Basic testing
4. **Haskell core toolchain**
   - `spec_to_ir` implementation
   - `ir_to_code` implementation
   - Basic testing

Phase 2: Representative Emitters (Weeks 5-12)

Goal: One working example per category per pipeline

For **each of 24 categories**, implement:
- Rust emitter (representative target)
- Haskell emitter (representative target)
- Enhanced Python emitter (if minimal)

Priority order:
1. **Assembly** (ARM) - Most critical for embedded
2. **Polyglot** (C99) - Most common target
3. **Embedded** (ARM Cortex-M) - Safety-critical
4. **GPU** (CUDA) - High-performance computing
5. **WASM** - Web/portable targets
6. **Lisp** (Common Lisp) - Language family example
7. **Prolog** (SWI-Prolog) - Logic programming
8-24. Remaining categories

Phase 3: Verification (Weeks 13-16)

Goal: Achieve 100% confluence score

1. Run full test suite on all pipelines
2. Measure confluence score
3. Fix discrepancies

4. Document results
5. Generate precompiled binaries

Phase 4: Complete Implementations (Months 4-6+)

Goal: Support all variants within each category

- Expand from representative to all targets
- Example: All 8 Lisp dialects, not just Common Lisp
- Lower priority (can be phased)

---

8. Resource Estimates

Development Time

Task	Rust	Haskell	Python	Total
Core toolchain	160h	160h	40h	360h
Representative emitters (24)	320h	320h	80h	720h
Testing frame-work	80h	-	-	80h
Documentation	40h	40h	40h	120h
Total (Phase 1-3)	600h	520h	160h	**1280h**

Estimated Calendar Time: 4-6 months with 2-3 developers

Maintenance Burden

Once confluence is achieved:
- New features must be implemented in **all 4 pipelines**
- Bug fixes must be applied to **all 4 pipelines**
- Confluence tests must pass **100%** before merge

Ongoing Cost: ~30-40% increase in development time per feature

Benefit: Universal deployability, cross-validation, organizational acceptance

---

9. Risk Assessment

High Risks

1. **Resource Availability**
   - Risk: Not enough developers with Rust/Haskell expertise
   - Mitigation: Phased approach, focus on core toolchain first

2. **Confluence Divergence**
   - Risk: Implementations produce different outputs
   - Mitigation: Automated testing, reference implementation (SPARK)

3. **Maintenance Burden**
   - Risk: 4x code to maintain
   - Mitigation: Shared test vectors, automated confluence checks

Medium Risks

1. **Performance Variations**
   - Risk: Different pipelines have different speeds
   - Impact: Low (correctness > performance)

2. **Language-Specific Bugs**
   - Risk: Each language has unique edge cases
   - Mitigation: Extensive testing, cross-validation

---

10. Success Metrics

Short-Term (3 months)

- ✅ Confluence specification documented
- [ ] Rust core toolchain implemented
- [ ] Haskell core toolchain implemented
- [ ] 12/24 representative emitters per pipeline (50%)
- [ ] Confluence test framework operational
- [ ] Initial confluence score measured

Medium-Term (6 months)

- [ ] 24/24 representative emitters per pipeline (100%)
- [ ] Confluence score ≥ 95%
- [ ] Build system supports all 4 pipelines
- [ ] Documentation complete

Long-Term (12 months)

- [ ] Confluence score = 100%
- [ ] All target variants implemented
- [ ] Precompiled binaries for all platforms
- [ ] Production deployments in all 4 contexts

---

11. Conclusion

Current State

- **SPARK**: Production-ready, complete
- **Python**: Functional but needs enhancement
- **Rust**: Not started
- **Haskell**: Not started

Path to Confluence

1. ✅ Document requirements (DONE)
2. Build Rust & Haskell core toolchains (4-8 weeks)
3. Implement representative emitters (8-12 weeks)
4. Create test framework and measure confluence (2-4 weeks)
5. Iterate to 100% confluence (ongoing)

Bottom Line

Confluence is achievable in 4-6 months with focused effort on:
1. Core toolchains (highest priority)
2. Representative emitters (one per category)
3. Automated testing (catch divergence early)

Recommendation: Proceed with Phase 1 immediately.

Audit Control:
- **Version**: 1.0
- **Date**: 2026-01-30
- **Next Audit**: After Phase 1 completion (estimated 2026-03-30)
- **Auditor**: STUNIR Team