Organizational Requirements for STUNIR Pipelines

Document Status: INFORMATIONAL
Date: 2026-01-30
Purpose: Explain why all four pipelines must be equally complete

---

Why Four Different Pipelines?

Different organizations have fundamentally different requirements for code review, auditability, and assurance. STUNIR must support all organizational contexts to be universally deployable.

---

1. Python Pipeline

Target Organizations

- Research institutions
- Data science teams
- Organizations prioritizing rapid development
- Teams requiring maximum readability
- Environments with strong Python expertise

Why Python?

- **Readability**: "Executable pseudocode" - easy for humans to audit
- **Ubiquity**: Most widely taught programming language
- **Tooling**: Extensive ecosystem for testing and analysis
- **Accessibility**: Lower barrier to entry for reviewers

Assurance Argument

> "We can trust this implementation because **any competent programmer can read and verify the logic**. The code is simple, well-documented, and testable."

Example Organizations

- Academic institutions
- NASA JPL (Python used for Mars rovers)
- Financial services (rapid prototyping)
- Machine learning teams

---

2. Haskell Pipeline

Target Organizations

- Formal methods teams
- High-assurance systems
- Organizations requiring mathematical proof
- Financial services (critical infrastructure)
- Research labs (advanced type systems)

Why Haskell?

- **Type Safety**: Advanced type system prevents entire classes of bugs
- **Formal Verification**: Strong theoretical foundation
- **Mathematical Rigor**: Code is close to mathematical specification
- **Immutability**: No hidden state, easier to reason about
- **Maturity = Assurance**: If it compiles, it's likely correct

Assurance Argument

> "We can trust this implementation because **the type system proves correctness**. Entire classes of errors are impossible at compile-time. The mature, well-tested codebase reduces risk."

Example Organizations

- Financial trading systems (Jane Street, Standard Chartered)
- Aerospace (formal verification required)
- Cryptocurrency infrastructure
- High-frequency trading

---

3. Rust Pipeline

Target Organizations

- Pure Rust development shops
- Safety-critical systems
- Organizations requiring memory safety
- High-performance computing
- Systems programming teams

Why Rust?

- **Memory Safety**: Guaranteed at compile time without garbage collection
- **Performance**: Zero-cost abstractions, competitive with C/C++
- **Modern Tooling**: Cargo, built-in testing, documentation
- **Safety + Speed**: No trade-off between the two
- **Growing Ecosystem**: Industry momentum (Linux kernel, Android, AWS)

Assurance Argument

> "We can trust this implementation because **the compiler enforces safety**. Memory errors, data races, and undefined behavior are impossible. Performance is predictable and measurable."

Example Organizations

- Automotive (safety-critical embedded)
- Operating system development
- Blockchain infrastructure
- Cloud providers (Cloudflare, AWS)
- Mozilla, Microsoft, Google (internal tooling)

---

4. SPARK Pipeline (Ada SPARK)

Target Organizations

- DO-178C Level A certification required
- Medical devices (FDA approval)
- Military systems
- Nuclear systems
- Aviation (Boeing, Airbus)
- Space systems (ESA)

Why SPARK?

- **DO-178C Compliance**: Directly supports certification
- **Formal Verification**: Integrated static analysis and proof
- **Runtime Error Freedom**: Proven absence of overflow, divide-by-zero, etc.
- **Deterministic Behavior**: No undefined behavior by design
- **Legacy Trust**: Decades of use in critical systems

Assurance Argument

> "We can trust this implementation because **it meets DO-178C Level A requirements**. The code has been formally verified, all runtime errors proven impossible, and the toolchain is certified for safety-critical use."

Example Organizations

- Boeing (avionics)
- Airbus (flight control systems)
- Thales (defense systems)
- ESA (space missions)
- Medical device manufacturers
- Nuclear power plants

Real-World Scenarios

Scenario 1: Aviation Certification

Organization: Commercial aircraft manufacturer
Requirement: DO-178C Level A for flight control software

Pipeline Choice: **SPARK**
Rationale: Certification authority requires SPARK for Level A. No alternatives acceptable.

Problem if SPARK-only:
- Development team is Python experts, not Ada experts
- Code review is slow and expensive
- Finding Ada developers is difficult

Solution with Confluence:
- **Develop** using Python pipeline (fast, familiar)
- **Test** using Python pipeline (comprehensive test suite)
- **Verify** confluence between Python and SPARK
- **Certify** using SPARK pipeline (meets DO-178C)
- **Audit** using Python (reviewers can read the logic)

Scenario 2: Financial Trading System

Organization: Hedge fund developing high-frequency trading algorithms
Requirement: Mathematical correctness, type safety, performance
Pipeline Choice: **Haskell**
Rationale: Organization's entire stack is Haskell. Type system provides assurance.

Problem if Haskell-only:
- Regulators may not have Haskell experts for audit
- New team members struggle with Haskell learning curve
- Integration with Python-based data science tools is complex

Solution with Confluence:
- **Develop** using Haskell pipeline (native to team)
- **Test** using Haskell pipeline (QuickCheck, property-based testing)
- **Verify** confluence with Python pipeline
- **Audit** using Python (regulators can review)
- **Integrate** with data science using Python

Scenario 3: Embedded Automotive System

Organization: Automotive supplier for safety-critical ECU
Requirement: ISO 26262 ASIL-D, memory safety, no runtime errors
Pipeline Choice: **Rust**
Rationale: Organization standardized on Rust for memory safety + performance.

Problem if Rust-only:
- Safety auditors may not have Rust expertise
- Legacy teams prefer C/C++ and are skeptical of Rust
- Formal verification tools for Rust are less mature than SPARK

Solution with Confluence:
- **Develop** using Rust pipeline (memory safety guaranteed)
- **Test** using Rust pipeline (robust testing framework)
- **Verify** confluence with SPARK pipeline

- **Certify** using SPARK (formal verification for critical paths)
- **Audit** using Python (safety auditors can review logic)

Scenario 4: Open Source Project

Organization: Community-driven safety-critical project
Requirement: Broad accessibility, trust through transparency
Pipeline Choice: **Python** (primary) + **SPARK** (verification)
Rationale: Maximize contributor pool, verify critical paths formally.

Problem if Python-only:
- Lack of formal verification
- Performance concerns for production use
- Memory safety not guaranteed

Solution with Confluence:
- **Develop** using Python pipeline (broad contributor base)
- **Verify** using SPARK pipeline (formal proofs for critical code)
- **Deploy** using Rust pipeline (performance + safety)
- **Audit** using any pipeline (pick your language)

Confluence Enables Trust

Without Confluence

- Each organization must trust a **single implementation**
- If your organization’s language isn’t supported, you can’t use STUNIR
- No way to cross-validate implementations
- Single point of failure (bugs in one implementation)

With Confluence

- Organizations can **choose their trusted language**
- All implementations produce **identical outputs** (verified by testing)
- Cross-validation **increases confidence** (4 independent implementations)
- If one implementation has a bug, **confluence testing catches it**

The Assurance Stack

Each organization builds trust differently:

Organization Type	Primary Pipeline	Secondary Verification	Assurance Mechanism
Avionics	SPARK	Python (review)	DO-178C certification
Finance	Haskell	Python (audit)	Type system + testing
Automotive	Rust	SPARK (proofs)	Memory safety + ISO 26262
Research	Python	SPARK (verify)	Peer review + formal verification

---

Regulatory Acceptance

Different regulatory bodies accept different languages:

- **FAA/EASA** (Aviation): Require SPARK for DO-178C Level A
- **FDA** (Medical): Accept SPARK, C, Ada; prefer formal verification
- **SEC/FINRA** (Finance): Language-agnostic but require auditability (Python often preferred)
- **ISO 26262** (Automotive): Language-agnostic but require safety analysis (Rust gaining acceptance)
- **IEC 61508** (Industrial): Accept multiple languages with appropriate tooling

STUNIR's Strategy: Support all regulatory contexts by providing pipelines in all required languages.

Developer Experience

Different teams have different expertise:

- **Python developers**: Can read/write Python, struggle with Haskell/SPARK
- **Haskell developers**: Can read/write Haskell, struggle with imperative code
- **Rust developers**: Can read/write Rust, prefer memory safety guarantees
- **Ada/SPARK developers**: Can read/write SPARK, prefer formal verification

STUNIR's Strategy: Let teams use their native language, verify confluence automatically.

Summary

Confluence is not a luxury—it's a necessity.

STUNIR's mission is to be the **universal deterministic build system**. To be universal, it must support:
1. **Languages organizations trust** (Python, Haskell, Rust, SPARK)
2. **Regulatory requirements** (DO-178C, ISO 26262, FDA)

3. **Developer expertise** (let teams use their best language)
4. **Formal verification** (where needed)
5. **Auditability** (in human-readable languages)

By achieving confluence, STUNIR becomes **truly universal**—every organization can use it, trust it, and verify it in their own way.

Document Control:
- **Version**: 1.0
- **Status**: INFORMATIONAL
- **Owner**: STUNIR Team
- **Next Review**: Annually