Testing Flow Analysis

Universal Stochastic Predictor (v2.1.0-RC1)

Stochastic Predictor Team

20 de febrero de 2026

Resumen

Este documento describe la arquitectura de testing reorganizada en dos capas de validación: **compliance** y **execution**. La pipeline utiliza auto-discovery dinámico para adaptarse a nuevos módulos sin hardcoding.

Índice

1. Executive Overview

El sistema de testing ha sido reorganizado en una arquitectura modular de **2 capas de validación** orquestadas por un **entrypoint central** (`tests_start.sh`). Cada capa valida un aspecto diferente del código:

Capa	Script	Propósito	Scope Discovery
1. Compliance	`code_alignement.py`	Valida 36 políticas	Auto-descubre módulos
2. Execution	`code_structure.py`	Tests reales con pytest+JAX	Auto-descubre+parametriza

DEPRECATED: `tests_coverage.py` integrado en `code_structure.py` (TestIOModuleImportable)

2. Arquitectura Lógica

2.1. Pipeline de 2 Capas

La arquitectura sigue un flujo secuencial con estrategia *fail-fast*:

Entrypoint `tests_start.sh` - Orchestrator principal (Bash)

Layer 1 `code_alignement.py` - Policy Compliance Checker

- Valida 36 CODE_AUDIT_POLICIES
- Scope: Todo el repositorio
- Auto-descubre módulos Python
- Output: JSON + Markdown reports
- **FAIL** → **EXIT** (detiene pipeline)

Layer 2 `code_structure.py` - Code Execution Tests

- Ejecuta tests reales con pytest + JAX
- Auto-genera tests parametrizados para io/
- Coverage: 126 funciones en api, core, kernels, io
- Output: JSON + Markdown reports

2.2. Reportes Generados

Cada script genera dos tipos de reportes:

Script	JSON Report	Markdown Report
code_alignement.py	tests/results/code_alignement_last.json	tests/reports/code_alignement_last.md
code_structure.py	tests/results/code_structure_last.json	tests/reports/code_structure_last.md

Nota: Los archivos usan sufijo `_last` (sin timestamp) para facilitar acceso al último resultado.

3. Scope Discovery

3.1. Módulo Central: scope_discovery.py

Todas las funciones de auto-descubrimiento están centralizadas en `tests/scripts/scope_discovery.py`:

```python
def discover_modules(root) -> List[str]:
    """Auto-discover all submodules in Python/ directory."""
    # Returns: ['api', 'core', 'io', 'kernels']

def extract_public_api(module_name) -> Set[str]:
    """Extract __all__ exports from module __init__.py"""

def discover_all_public_api() -> Dict[str, Set[str]]:
    """Map all modules to their public symbols"""
```

3.2. Ventajas del Auto-Discovery

- **Cero hardcoding**: No listas manuales de módulos/funciones
- **Auto-adaptación**: Detecta nuevos módulos automáticamente
- **Mantenimiento reducido**: Sin actualización manual de scopes
- **Consistencia**: Misma lógica de discovery en todos los scripts

4. Scripts Detallados

4.1. Stage 1: code_alignement.py (Policy Compliance)

Propósito: Validar 36 políticas de auditoría de código.
Scope: Repositorio completo (no limitado a Python/)
Políticas Validadas:

- Configuration sourcing (zero-heuristics)
- Configuration immutability (locked subsections)
- Validation schema enforcement
- Atomic configuration mutation protocol
- 64-bit precision enablement
- Stop-gradient for diagnostics
- Kernel purity and statelessness
- Five-layer architecture enforcement
- Dependency pinning (exact versions)
- *...y 27 políticas adicionales*

Output:

- Console: PASS/FAIL por política
- JSON: `tests/results/code_alignement_last.json`
- Markdown: `tests/reports/code_alignement_last.md`

4.2. Stage 2: code_structure.py (Execution Tests)

Propósito: Ejecutar tests reales con JAX y pytest.
Scope: Python/* (todos los submódulos)
Estrategia de Testing:

- Tests explícitos para api, core, kernels (20+ test classes)
- Tests auto-parametrizados para io/ (via TestIOModuleImportable)
- Coverage: 126 funciones (api:53, core:20, kernels:23, io:30)
- Ejecución real con JAX x64 habilitado

Módulos Cubiertos:

Módulo	Funciones	Test Strategy
api	53	Explicit test classes
core	20	Explicit test classes
kernels	23	Explicit test classes
io	30	Auto-parametrized (TestIOModuleImportable)
Total	**126**	**Mixed**

5. Ejecución de Tests

5.1. Comando Principal

```
# Ejecutar toda la pipeline (compliance + execution)
./tests/scripts/tests_start.sh --all

# Solo compliance
./tests/scripts/tests_start.sh --compliance

# Solo execution tests
./tests/scripts/tests_start.sh --execute
```

5.2. Flujo de Ejecución

1. **Validación del entorno**
   - Verifica existencia de `.venv/bin/python`
   - Valida directorios `tests/results` y `tests/reports`
2. **Stage 1: Compliance Check**
   - Ejecuta `code_alignement.py`
   - Valida 36 políticas
   - Si FAIL → detiene pipeline (fail-fast)
3. **Stage 2: Execution Tests**
   - Ejecuta `code_structure.py` vía pytest
   - Tests reales con JAX
   - Genera reportes JSON + Markdown
4. **Summary**
   - Total/Passed/Failed por stage
   - Lista últimos artefactos en `tests/results/`

6. Troubleshooting

6.1. Errores Comunes

Error: Python virtual environment not found

```
# Solución: Crear y activar .venv
python3 -m venv .venv
source .venv/bin/activate
pip install -r requirements.txt
```

Error: Compliance check failed

- Revisar `tests/reports/code_alignement_last.md`
- Identificar qué política falló
- Corregir código según especificación en `doc/latex/tests/code_audit_policies.tex`

Error: JAX not installed

```
# Solución: Instalar dependencias
source .venv/bin/activate
pip install -r requirements.txt
```

6.2. Verificación Manual

```
# Verificar descubrimiento de módulos
cd tests/scripts
python3 -c "from scope_discovery import discover_modules; print(
    discover_modules())"
# Expected: ['api', 'core', 'io', 'kernels']

# Verificar API pública de un módulo
python3 -c "from scope_discovery import extract_public_api; print(
    extract_public_api('api'))"

# Ejecutar compliance manualmente
.venv/bin/python tests/scripts/code_alignement.py

# Ejecutar structure tests manualmente
.venv/bin/python tests/scripts/code_structure.py
```

7. Changelog: 3-Layer → 2-Layer

7.1. Arquitectura Anterior (3 Capas)

1. **code_alignement.py** - Policy compliance
2. **tests_coverage.py** - Coverage validator
3. **code_structure.py** - Execution tests

7.2. Problema Identificado

Si `code_structure.py` auto-descubre funciones y las testea, entonces `tests_coverage.py` siempre pasará (redundancia lógica).

7.3. Solución: Consolidación

- **Eliminado**: `tests_coverage.py` (marcado como DEPRECATED)
- **Consolidado**: Lógica de coverage en `code_structure.py` (TestIOModuleImportable)
- **Resultado**: Pipeline más simple y sin redundancia

7.4. Commits Relevantes

```
# feat: create scope_discovery.py with auto-discovery functions
# refactor: use dynamic paths in code_alignement.py (38+ replacements)
# refactor: consolidate to 2-layer pipeline (remove coverage stage)
# feat: add TestIOModuleImportable for Python/io/ coverage
# docs: create LaTeX documentation at doc/latex/tests/
```

8. Referencias

8.1. Documentación Relacionada

- `doc/latex/tests/code_audit_policies.tex` - 36 políticas de código
- `doc/latex/tests/testing_audit_policies.tex` - 45 políticas de testing
- `tests/scripts/scope_discovery.py` - Módulo de auto-discovery
- `tests/scripts/code_alignement.py` - Implementación de políticas
- `tests/scripts/code_structure.py` - Suite de tests de ejecución

8.2. Especificación Matemática

- `doc/latex/specification/Stochastic_Predictor_Test_Cases.tex`
- `doc/latex/specification/Stochastic_Predictor_Tests_Python.tex`

Document Approved For: Implementation
Last Updated: 2026-02-20
Maintainer: Stochastic Predictor Team