Universal Stochastic Predictor

Testing Infrastructure Implementation

Complete Deployment, Usage Guide, and Expected Outcomes

Development Team

Document Version: 2.0
Last Updated: 2026-02-21
Test Framework Implementation: v2.1.0

Abstract

This is a comprehensive **Implementation Document** describing the complete deployment, operational usage, and expected outcomes of the Universal Stochastic Predictor's automated testing infrastructure. It documents how the system has been architected and deployed, how practitioners use it in daily development workflows, and what measurable quality and performance outcomes to expect.

Scope of this Document:

1. **Deployment Strategy:** How the three-phase system was constructed and integrated
2. **Usage Patterns:** How developers and CI/CD systems invoke the testing framework
3. **Expected Outcomes:** Quantified quality metrics, performance benchmarks, and success criteria

The system features dynamic module discovery without hardcoding, intelligent change detection for incremental testing (13–50× speedup in typical compliance workflows with `code_alignement.py`), comprehensive JSON and Markdown reporting, and extensible architecture for adding new validation rules and tests.

Contents

1 System Overview **5**
1.1 Objectives . . . 5
1.2 Architectural Layers . . . 5
1.3 Pipeline Execution . . . 5

2 Deployment Strategy **5**
2.1 Development Timeline . . . 5
2.2 System Architecture Decisions . . . 6
2.2.1 Why Four-Phase Pipeline? . . . 6
2.2.2 Why Auto-Discovery Instead of Hardcoding? . . . 6
2.2.3 Why Change Detection? . . . 7
2.3 Integration with Version Control . . . 7
2.4 CI/CD Integration Points . . . 7

3 Usage Patterns **8**
3.1 Development Workflow . . . 8
3.1.1 Local Development (Quick Feedback) . . . 8
3.1.2 Pre-Commit Hook . . . 8

3.1.3 Full Audit Before Release . . . 8
3.1.4 Debugging Failed Tests . . . 8
3.2 CI/CD Workflow . . . 9
3.3 Troubleshooting Using Reports . . . 9

4 Expected Outcomes **9**
4.1 Quality Metrics . . . 9
4.2 Performance Expectations . . . 10
4.3 Success Criteria . . . 10
4.4 Failure Analysis . . . 10
4.5 Continuous Improvement . . . 10

5 Logical Architecture & Execution Flow **11**
5.1 Two-Layer Validation Model . . . 11
5.2 Scope Auto-Discovery Pattern . . . 11
5.2.1 Benefits of Auto-Discovery . . . 11
5.3 Change Detection Mechanism . . . 11
5.3.1 Motivation . . . 11
5.3.2 Implementation . . . 12
5.3.3 CLI Flags . . . 12
5.4 Pipeline Execution Flow . . . 12
5.5 Report Generation . . . 13
5.6 Architecture Evolution Note . . . 13
5.6.1 Previous: 3-Layer Model . . . 13
5.6.2 Identified Redundancy . . . 13
5.6.3 Current: 2-Layer Model . . . 13

6 Core Components **14**
6.1 Orchestrator: `Test/scripts/tests_start.sh` . . . 14
6.1.1 Purpose . . . 14
6.1.2 Invocation . . . 14
6.1.3 Implementation Details . . . 14
6.1.4 Exit Codes . . . 14
6.2 Linting Phase: `code_lint.py` . . . 14
6.2.1 Purpose . . . 14
6.2.2 Scope . . . 15
6.2.3 Configuration . . . 15
6.2.4 Output . . . 15
6.2.5 Exit Behavior . . . 15
6.2.6 Change Detection Integration . . . 15
6.3 Compliance Phase: `code_alignement.py` . . . 15
6.3.1 Purpose . . . 15
6.3.2 Scope . . . 16
6.3.3 Verification Method . . . 16
6.3.4 Output . . . 16
6.3.5 Exit Behavior . . . 16
6.4 Execution Phase: `code_structure.py` . . . 16
6.4.1 Purpose . . . 16
6.4.2 Framework . . . 17
6.4.3 Test Organization . . . 17
6.4.4 Output . . . 17
6.4.5 Exit Behavior . . . 17

6.5 Discovery Module: `scope_discovery.py` . . . 17
6.5.1 Purpose . . . 17
6.5.2 Core Functions . . . 18
6.5.3 Change Detection Algorithm . . . 18
6.5.4 Cache Format . . . 18
6.5.5 Performance Characteristics . . . 19

7 Change Detection System **19**
7.1 Motivation . . . 19
7.2 User Interface . . . 19
7.2.1 Incremental (Default) . . . 19
7.2.2 Full Audit . . . 19
7.2.3 Cache Management . . . 20
7.3 Integration with Scripts . . . 20

8 Execution Flow **20**
8.1 Complete Pipeline Diagram . . . 20
8.2 Fail-Fast Semantics . . . 21

9 Reporting and Artifacts **21**
9.1 Directory Structure . . . 21
9.2 Report Formats . . . 21
9.2.1 JSON Format . . . 21
9.2.2 Markdown Format . . . 22
9.3 Log Retrieval . . . 22
9.3.1 Most Recent Results . . . 22
9.3.2 Markdown Reports . . . 23

10 Command Reference **23**
10.1 Complete CLI Options . . . 23
10.2 Practical Workflows . . . 23
10.2.1 Developer Workflow (Iterative) . . . 23
10.2.2 Pre-Commit Workflow . . . 23
10.2.3 CI/CD Integration . . . 24

11 Troubleshooting **24**
11.1 Common Issues . . . 24
11.1.1 Issue: Python virtual environment not found . . . 24
11.1.2 Issue: Cache inconsistent or corrupted . . . 24
11.1.3 Issue: Linting reports false positives . . . 24
11.1.4 Issue: Tests timeout . . . 25
11.1.5 Issue: Memory errors during execution tests . . . 25

12 Implementation Notes **25**
12.1 Design Decisions . . . 25
12.1.1 Why Auto-Discovery? . . . 25
12.1.2 Why Change Detection? . . . 25
12.1.3 Why Separate Scripts? . . . 26
12.2 Extension Points . . . 26
12.2.1 Adding New Linting Tool . . . 26
12.2.2 Adding New Policy . . . 26
12.2.3 Adding New Test Module . . . 26

13 Performance Characteristics **27**
13.1 Benchmarks 27
13.2 Optimization Opportunities 27

14 Future Enhancements **27**
14.1 Planned Features 27
14.1.1 Known Limitations 27

15 Conclusion **28**

1 System Overview

1.1 Objectives

The testing infrastructure serves four primary objectives:

1. **Code Quality Validation:** Enforce PEP 8 standards, type safety, and styling rules via `flake8`, `mypy`, `black`, and `isort`.
2. **Policy Compliance Auditing:** Verify all 36 architecture policies are satisfied (see `Code_Testing_Audit_Policies.tex`).
3. **Functional Correctness:** Execute comprehensive unit tests targeting all API, Core, Kernel, and I/O modules.
4. **Performance Optimization:** Detect only modified files since last run, reducing `code_alignement.py` execution time from 12.5s to <1s in typical workflows (incremental mode).

1.2 Architectural Layers

The testing system is organized into five layers (matching the system architecture):

Layer	Components	Tests
API	`config.py`, `prng.py`, `schemas.py`, etc.	Warmup, validation
Core	`orchestrator.py`, `fusion.py`, etc.	Orchestration, meta-opt
Kernels	`kernel_a/b/c/d.py`, `base.py`	Prediction, singularities
I/O	`snapshots.py`, `telemetry.py`, etc.	Persistence, telemetry
Tests	`code_lint.py`, `scope_discovery.py`, etc.	This infrastructure

1.3 Pipeline Execution

The default test pipeline executes in strict sequence with fail-fast semantics:

0. **Dependency Check:** `Test/scripts/dependency_check.py` validates Golden Master compliance (always runs first)
1. **Linting Phase:** `Test/scripts/code_lint.py` validates style and formatting
2. **Compliance Phase:** `Test/scripts/code_alignement.py` audits 36 policies (with cache optimization)
3. **Execution Phase:** `Test/scripts/code_structure.py` runs 79 unit tests with real JAX execution

If any phase fails, subsequent phases are skipped and the pipeline exits with code 1.

Critical: Dependency version check runs before all other tests to ensure environment matches `requirements.txt`. This prevents false test failures caused by version mismatches.

2 Deployment Strategy

2.1 Development Timeline

The testing infrastructure was developed incrementally to address specific project needs:

1. **Phase 1 (Initial):** `code_alignement.py` implemented to enforce 38 architecture policies from specification

2. **Phase 2:** `code_structure.py` added for comprehensive unit testing with pytest + JAX integration
3. **Phase 3:** `scope_discovery.py` introduced to eliminate hardcoded module lists and enable automatic adaptation
4. **Phase 4:** `code_lint.py` and change detection system added for workflow optimization
5. **Phase 5 (Current):** `dependency_check.py` added for Golden Master validation with platform-aware environment marker parsing

2.2 System Architecture Decisions

2.2.1 Why Four-Phase Pipeline?

The pipeline is structured as four independent validation stages rather than a monolithic script:

Stage	Rationale	Why Separate?
Dependency Check	Validates installed package versions against Golden Master	Fail-fast on env mismatch
Linting	Fast preliminary checks (milliseconds per file)	Fail early on style issues
Compliance	Policy validation across repository. Slow but essential.	Cannot be parallelized
Execution	Resource-intensive functional tests with JAX	Only run if earlier stages pass

Benefit: Fail-fast semantics allow developers to fix issues progressively without running expensive tests.

2.2.2 Why Auto-Discovery Instead of Hardcoding?

Original design used explicit module lists:

```
# Old approach (brittle)
MODULES = ['api', 'core', 'kernels', 'io']  # Must update manually
```

Problems with hardcoding:

- Adding new module requires editing 3+ scripts
- Risk: New modules accidentally omitted from tests
- Maintenance burden: Changes propagate poorly

Current approach uses filesystem discovery:

```
# New approach (automatic)
def discover_modules(root):
    python_dir = root / "Python"
    init_files = sorted(python_dir.glob("*/__init__.py"))
    return [init_file.parent.name for init_file in init_files]
# Finds: ['api', 'core', 'kernels', 'io'] automatically
```

Result: Add new module to `Python/`, tests discover it automatically. Zero configuration.

2.2.3 Why Change Detection?

During development, running 79 tests on every save is unproductive. Analysis:

- Typical commit: 2-5 files modified
- Full test run: 45 seconds
- Incremental run: 3-5 seconds
- **Productivity impact:** Developers don't wait, tests get skipped

Solution: Intelligent caching of file modification times.

```
# Store modification times in cache
{
  "Python/api/config.py": 1708456789.123,
  "Python/core/orchestrator.py": 1708456800.456,
  ...
}

# On next run: only test files with changed mtimes
```

Trade-off: Requires cache reset if external tools modify files, but `--force` and `--reset-cache` flags always available.

2.3 Integration with Version Control

The testing infrastructure assumes git-based development workflow:

```
# Developer workflow:
$ git checkout -b feature/new_kernel_e
$ nano Python/kernels/kernel_e.py
$ ./tests/scripts/tests_start.sh --lint    # Fast
$ ./tests/scripts/tests_start.sh --execute # Real test
$ git add . && git commit -m "Add␣Kernel␣E"

# Pre-push: full audit
$ ./tests/scripts/tests_start.sh --force --all
```

Cache is not version-controlled (lives in `tests/.scope_cache.json`, typically `.gitignored`). Each environment maintains its own cache.

2.4 CI/CD Integration Points

The system is designed for seamless CI/CD integration:

- **Exit codes:** 0 (success), 1 (failure) follow Unix conventions
- **JSON reports:** Structured output for metrics dashboards
- **Markdown reports:** Human-readable for review systems (GitHub Actions, GitLab CI)
- **Deterministic:** Fixed PRNG seeds ensure test reproducibility

Typical CI pipeline:

```
# .github/workflows/test.yml
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-python@v4
```

```
        with:
          python-version: '3.13'
      - run: pip install -r requirements.txt
      - run: ./Test/scripts/tests_start.sh --force --all
      - uses: actions/upload-artifact@v3
        with:
          name: test-reports
          path: Test/reports/
```

3 Usage Patterns

3.1 Development Workflow

Developers use different invocations depending on context:

3.1.1 Local Development (Quick Feedback)

```
# Edit a file
$ nano Python/api/config.py

# Run only affected tests (incremental, ~2s)
$ ./tests/scripts/tests_start.sh --lint

# Fix issues, test again
$ ./tests/scripts/tests_start.sh

# Run full execution tests when ready
$ ./tests/scripts/tests_start.sh --execute
```

3.1.2 Pre-Commit Hook

```
# Create .git/hooks/pre-commit
#!/bin/bash
set -e
./Test/scripts/tests_start.sh --lint
echo "␣Pre-commit␣checks␣passed"
```

3.1.3 Full Audit Before Release

```
# Reset cache and run everything
$ ./tests/scripts/tests_start.sh --force --all
$ echo $?  # Exit code 0 = safe to release
```

3.1.4 Debugging Failed Tests

```
# Run just execution tests with verbose output
$ ./tests/scripts/tests_start.sh --execute -v

# View detailed report
$ cat tests/reports/code_structure_last.md

# Parse JSON for programmatic inspection
```

```
$ python -c "import␣json;␣\
␣␣␣␣r␣=␣json.load(open('tests/results/code_structure_last.json'));␣\
␣␣␣␣print(r['failed_checks'])"
```

3.2 CI/CD Workflow

Automated systems run in **full-audit mode** always:

```
# CI job always: --force (full audit)
./tests/scripts/tests_start.sh --force --all

# Parse results for downstream actions
if [ $? -ne 0 ]; then
    echo "Tests␣failed!"
    exit 1
fi

# Post results to dashboard
curl -X POST https://dashboard.example.com/test-results \
    -H "Content-Type:␣application/json" \
    -d @tests/results/code_structure_last.json
```

3.3 Troubleshooting Using Reports

Practitioners use JSON and Markdown reports for debugging:

```
# Find which test failed
grep -A 5 '"passed":␣false' Test/results/code_structure_last.json

# Short summary
head -20 Test/reports/code_lint_last.md

# All compliance failures
grep "FAIL" Test/reports/code_alignement_last.md
```

4 Expected Outcomes

4.1 Quality Metrics

The testing system maintains these minimum quality thresholds:

Metric	Target	Current Status
Policy Compliance	36/36 (100%)	Passing
Linting (flake8)	0 violations	Passing
Code Formatting (black)	0 issues	Passing
Import Ordering (isort)	100% compliant	Passing
Type Checking (mypy)	*Informational*	Recommended to fix
Unit Tests	79/79 passing	Passing
JAX Determinism	Fixed PRNG seeds	Verified
64-bit Precision	All operations	Enabled

4.2 Performance Expectations

Expected test execution times on standard development machine (MacBook Pro M1):

Scenario	Linting	Compliance	Execution
Full audit	8.2s	12.5s	24.3s
Incremental (2 files)	2.1s	12.5s	24.3s
Incremental (0 files)	0.8s	12.5s	24.3s
Cumulative Total	45.0s	39.0s	37.8s

Note: Linting scales with number of changed files; compliance and execution are repository-wide (not file-specific).

4.3 Success Criteria

A successful test run satisfies all criteria:

1. **Exit Code 0:** All phases completed without failure
2. **All Policies Pass:** All 36 architecture policies verified
3. **Linting Green:** No PEP 8, formatting, or import violations
4. **All Tests Pass:** 79/79 unit tests execute successfully
5. **Reports Generated:** Both JSON and Markdown reports created
6. **Deterministic:** Same results for same code (reproducible)

4.4 Failure Analysis

When tests fail, use this decision tree:

1. **Linting Failed:**
   - **flake8 error:** Refactor code to remove violations
   - **black/isort error:** Run auto-fix: `black Python/ && isort Python/`
2. **Compliance Failed:**
   - Read policy number from report
   - Consult `code_audit_policies.tex` for requirement
   - Modify code/config to satisfy policy
3. **Execution Failed:**
   - Review pytest output for failing test class/method
   - Check test expectations in `code_structure.py`
   - Verify JAX operations and numerical precision
   - Run test locally with verbose output: `pytest -v -s`

4.5 Continuous Improvement

The system is designed for iterative refinement:

- **Add new policy:** Update `policy_checks()` in `code_alignement.py`
- **Add new test:** Create test class in `code_structure.py`; use `scope_discovery` for module enumeration
- **Add new linter:** Extend `code_lint.py` with new function returning `LinterResult`
- **Optimize performance:** Parallelization hooks exist for future enhancement

5 Logical Architecture & Execution Flow

5.1 Two-Layer Validation Model

The testing system implements a refined **2-layer validation architecture** with centralized orchestration:

Layer	Script	Purpose	Discovery
1. Compliance	`code_alignement.py`	Validate 36 policies	Auto-discover modules
2. Execution	`code_structure.py`	Real tests with pytest+JAX	Auto-discover + parametrize

Note: The deprecated `tests_coverage.py` was consolidated into `code_structure.py` via the `TestIOModuleImportable` test class to avoid redundancy.

5.2 Scope Auto-Discovery Pattern

Central to the architecture is the **Dynamic Scope Discovery** system implemented in `scope_discovery.py`. This module provides functions used by all test scripts:

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

# Extract public API from module's __init__.py
def extract_public_api(module_name) -> Set[str]:
    return {'PredictorConfig', 'initialize_jax_prng', ...}

# Get all modules' public APIs
def discover_all_public_api() -> Dict[str, Set[str]]:
    return {'api': {...}, 'core': {...}, ...}
```

5.2.1 Benefits of Auto-Discovery

- **Zero Hardcoding:** No manual lists of modules or functions
- **Automatic Adaptation:** Detects new modules when added to `Python/`
- **Reduced Maintenance:** Scope changes propagate across all scripts automatically
- **Consistency:** All test phases use identical discovery logic

5.3 Change Detection Mechanism

5.3.1 Motivation

In production workflows, code changes are localized (typically 1-5 files per commit). Full test audits on every commit waste computational resources. The system includes intelligent **change tracking** to run only affected tests:

Scenario	Files	Time
Incremental (2 files changed)	2/28	~3s
Incremental (no changes)	0/28	~1s
Full audit (`--force`)	28/28	~45s

5.3.2 Implementation

The cache file `Test/.scope_cache.json` stores modification timestamps of all Python files:

```python
# From scope_discovery.py
def discover_changed_files(root=None, force_all=False) -> List[str]:
    # Load cache from last run
    previous = load_file_timestamps()

    # Get current state
    current = get_all_python_files()

    # Compare: include file if new or timestamp changed
    if force_all:
        changed = list(current.keys())
    else:
        changed = [f for f in current
                   if f not in previous or current[f] != previous[f]]

    # Update cache for next run
    save_file_timestamps(current)
    return changed
```

5.3.3 CLI Flags

Flag	Behavior	Cache
(default)	Process only modified files (code_alignement)	Persistent
`--force-all`	Process all files, update cache	Updated with all files
`--no-cache`	Disable cache (same as –force-all)	Updated but not used
`--reset-cache`	Clear cache without running tests	Cleared

Table 1: Cache control flags for code_alignement.py and code_structure.py

Note: `code_alignement.py` uses cache to skip unchanged files (10-50× speedup). `code_structure.py` reports changed files but runs all tests due to pytest fixture dependencies.

5.4 Pipeline Execution Flow

The complete execution follows this sequence:

1. **Environment Validation**
   - Verify `.venv/bin/python` exists
   - Verify `Test/results/` and `Test/reports/` directories
   - Initialize cache system from `Test/.scope_cache.json`
2. **Stage 0: Dependency Version Check** (`dependency_check.py`)
   - Parse `requirements.txt` with PEP 508 environment marker support
   - Detect current platform (`sys.platform`, `platform.machine`)
   - Compare installed versions against Golden Master specification
   - Generate JSON and Markdown reports
   - **If FAIL** → Stop pipeline immediately (environment mismatch)
3. **Stage 1: Code Linting** (`code_lint.py`)

- Run flake8, black, isort, mypy on incremental (changed) files
- Generate JSON and Markdown reports
- **If FAIL** → Stop pipeline (fail-fast)

4. **Stage 2: Policy Compliance** (`code_alignement.py`)
   - Validate 38 CODE_AUDIT_POLICIES across entire repository
   - Generate detailed compliance reports
   - **If FAIL** → Stop pipeline (fail-fast)
5. **Stage 3: Execution Tests** (`code_structure.py`)
   - Run pytest suite with unit tests
   - Real JAX execution with 64-bit precision
   - Generate test results and coverage reports
6. **Summary**
   - Total/Passed/Failed counts per stage
   - List latest artifacts in `Test/results/`
   - Return aggregated exit code (0 = all passed, 1 = any failed)

5.5 Report Generation

Each test script produces two output formats for maximum compatibility:

Script	JSON Report	Markdown Report
`code_lint.py`	`code_lint_last.json`	`code_lint_last.md`
`code_alignement.py`	`code_alignement_last.json`	`code_alignement_last.md`
`code_structure.py`	`code_structure_last.json`	`code_structure_last.md`

All reports use `_last` suffix (no timestamp) to enable scripted access in CI/CD pipelines. JSON format includes structured failure data for dashboard integration; Markdown format is human-readable for manual review.

5.6 Architecture Evolution Note

5.6.1 Previous: 3-Layer Model

The system previously used three validation stages:

1. `code_alignement.py` (policies)
2. `tests_coverage.py` (coverage)
3. `code_structure.py` (execution)

5.6.2 Identified Redundancy

Since `code_structure.py` auto-generates tests for all discovered functions via `TestIOModuleImportable`, a separate coverage stage was logically redundant: all public functions are already tested.

5.6.3 Current: 2-Layer Model

The coverage stage was consolidated into `code_structure.py`, resulting in:

- Simpler pipeline (fewer stages)
- No redundant validation
- Faster overall test execution

6 Core Components

6.1 Orchestrator: `Test/scripts/tests_start.sh`

6.1.1 Purpose

Central orchestration script that:

- Detects virtual environment and Python executable
- Parses CLI options (`--force-all` for full audit, `--lint`, `--execute`)
- Manages test sequencing and fail-fast logic
- Generates unified summary reports

6.1.2 Invocation

```
./tests/scripts/tests_start.sh [OPTION]

# Examples
./tests/scripts/tests_start.sh              # Default: all tests, changed files only
./tests/scripts/tests_start.sh --force      # Full audit: reset cache, all files
./tests/scripts/tests_start.sh --lint       # Only: code linting
./tests/scripts/tests_start.sh --compliance # Only: policy audit
./tests/scripts/tests_start.sh --execute    # Only: functional tests
./tests/scripts/tests_start.sh --reset-cache # Clean cache, exit
```

6.1.3 Implementation Details

- **Python Detection:** Prefers `.venv/bin/python`, falls back to system Python
- **Change Tracking:** Calls `reset_cache()` if `--force` flag present
- **Fail-Fast:** Each test may pass extra arguments (e.g., `--force-all`)
- **Artifact Management:** Creates `tests/results/` and `tests/reports/` directories

6.1.4 Exit Codes

Code	Meaning
0	All tests passed
1	Test failure (first failure stops pipeline)
2	Invalid CLI argument
127	Python virtual environment not found

6.2 Linting Phase: `code_lint.py`

6.2.1 Purpose

Python code quality validation using four industry-standard tools:

1. **flake8:** PEP 8 style violations, complexity checks
2. **black:** Code formatting audit (not auto-fix in CI)
3. **isort:** Import statement ordering
4. **mypy:** Static type annotation checking (informational)

6.2.2 Scope

Scans all `*.py` files in `Python/` directory recursively, automatically discovering modules `api`, `core`, `kernels`, `io`.

6.2.3 Configuration

Tool	Configuration
flake8	`--max-line-length=120`, ignore E203/W503/E501
black	`--line-length=120`, check mode
isort	`--profile=black`, PEP 8 compliant
mypy	`--ignore-missing-imports`, silent import mode

6.2.4 Output

Generates two reports in standardized formats:

- **JSON:** `tests/results/code_lint_last.json`
  - Timestamp, total checks, pass/fail counts
  - Per-tool results with file counts and violation details
- **Markdown:** `tests/reports/code_lint_last.md`
  - Human-readable table format
  - Auto-fix recommendations for formatting issues

6.2.5 Exit Behavior

- **Mandatory tools** (flake8, black, isort): Return 0 only if all pass
- **Informational** (mypy): Report findings but do not fail pipeline
- Fail-fast: Stop orchestrator if any mandatory tool fails

6.2.6 Change Detection Integration

By default, processes only files detected as modified since last cache. Use `--force-all` to scan all 28 files:

```
from tests.scripts.scope_discovery import discover_changed_files
changed = discover_changed_files(force_all=False)  # Incremental
all_files = discover_changed_files(force_all=True)  # Full audit
```

6.3 Compliance Phase: `code_alignement.py`

6.3.1 Purpose

Repository-wide audit of 36 architecture policies derived from the specification corpus. Policies cover:

- Configuration sourcing and immutability (Policies 1–5)
- Numerical stability and precision (Policies 11–12)
- Kernel purity and JAX constraints (Policies 13, 36–38)
- I/O safety: snapshots, telemetry, credentials (Policies 18–21, 30–31)
- Advanced settings: entropy, Sinkhorn, DGM (Policies 23–26)

6.3.2 Scope

Repository-wide. Policies apply to:

- Configuration files (`config.toml`)
- Python initialization files (`__init__.py`)
- Core module implementations
- Documentation and specification corpus

6.3.3 Verification Method

Each policy is implemented as a closure returning `(bool, str)` tuple:

```
def policy_checks() -> List[Tuple[int, str, Callable[[], Tuple[bool, str
    ]]]]:
    return [
        (1, "Configuration␣Sourcing", lambda: check_config_sourcing()),
        (2, "Configuration␣Immutability", lambda: check_immutability()),
        # ... 34 more policies
    ]
```

6.3.4 Output

- **JSON:** `Test/results/code_alignement_last.json`
  - Timestamp, policy count, pass/fail breakdown
  - Per-policy result with details
  - Cache status: incremental vs full scan
- **Markdown:** `Test/reports/code_alignement_last.md`
  - Policy table with pass/fail status
  - Detailed failure messages for debugging
  - Files checked count (incremental mode)

6.3.5 Exit Behavior

- Returns 0 only if all 36 policies pass
- Fails entire pipeline if any policy violated
- Non-fatal: Continues checking all policies (doesn't stop at first failure)

6.4 Execution Phase: `code_structure.py`

6.4.1 Purpose

Functional unit testing via pytest framework. Executes 79 parametrized tests covering:

- **API Module (26 tests):** Config, PRNG, types, validation, schemas, state buffer, warmup
- **Core Module (7 tests):** Orchestrator, fusion, meta-optimizer
- **Kernels Module (18 tests):** Each kernel (A/B/C/D), base functions
- **I/O Module (7 tests):** Module importability checks
- **Integration (10 tests):** Warmup sequences, profile, retry logic

6.4.2 Framework

- **Framework:** pytest 7.3.0 with hypothesis (property-based testing)
- **JAX Integration:** Real JAX operations with 64-bit precision enabled
- **Determinism:** All tests use fixed PRNG seeds for reproducibility
- **Isolation:** Each test receives fresh config and random key

6.4.3 Test Organization

Tests are grouped into classes for organization:

Test Class	Count
TestBasicSetup	2
TestAPIConfig	4
TestAPIPRNG	6
TestAPITypes	4
TestAPIValidation	12
TestAPISchemas	2
TestAPIStateBuffer	5
TestCoreOrchestrator	6
TestCoreFusion	1
TestCoreMetaOptimizer	1
TestKernelsBase	4
TestKernelA	4
TestKernelB	3
TestKernelC	2
TestKernelD	3
TestAPIWarmup	7
TestIOModuleImportable	7
Total	**79**

6.4.4 Output

- **Console:** pytest-formatted output with pass/fail summary and cache status
- **JSON:** `Test/results/code_structure_last.json` (structured results)
- **Markdown:** `Test/reports/code_structure_last.md` (detailed summary with coverage)

6.4.5 Exit Behavior

- Returns 0 only if all 79 tests pass
- Returns 1 if any test fails (fail-fast not applied; all tests run)
- Captures full traceback for failed assertions

6.5 Discovery Module: `scope_discovery.py`

6.5.1 Purpose

Provides dynamic module discovery and change detection without hardcoding paths or module lists. Enables:

1. **Auto-Discovery:** Enumerate modules from `Python/` directory structure
2. **Change Tracking:** Detect modified files since last test run

3. **Incremental Scanning:** Process only changed files for speed
4. **Public API Extraction:** Parse module `__init__.py` for public symbols

6.5.2 Core Functions

Function	Purpose
`discover_modules()`	List all modules in `Python/`: ['api', 'core', 'kernels', 'io']
`discover_module_files()`	List .py files in a specific module
`extract_public_api()`	Parse module's `__all__` or extract public symbols
`discover_all_public_api()`	Return dict of all modules' public APIs
`get_all_python_files()`	Get all .py files with modification timestamps
`discover_changed_files(force_all)`	Return files modified since last cache, or all if `force_all=True`
`discover_module_files_changed()`	Changed files in specific module
`load_file_timestamps()`	Load cached file timestamps from `.scope_cache.json`
`save_file_timestamps()`	Update cache with current file state
`reset_cache()`	Delete cache (forces full scan on next run)
`get_cache_info()`	Return cache statistics

6.5.3 Change Detection Algorithm

Algorithm for detecting file changes:

1. **Get Current State:** Scan all .py files, store path + modification timestamp
2. **Load Previous State:** Read timestamps from `tests/.scope_cache.json`
3. **Compare:** For each file:
   - If not in previous cache: File is NEW → Include
   - If mtime differs: File is MODIFIED → Include
   - If mtime unchanged: File not changed → Omit
4. **Save New State:** Update cache for next run
5. **Return:** List of changed files

6.5.4 Cache Format

Cache file: `Test/.scope_cache.json`

```
{
  "Python/api/__init__.py": 1708456789.123,
  "Python/api/config.py": 1708456795.456,
  "Python/api/prng.py": 1708456801.789,
  "Python/core/orchestrator.py": 1708456812.345,
  ...
}
```

Key	Relative path from project root (28 Python files total)
Value	Unix timestamp (float with microsecond precision from st_mtime)

Coverage: Cache automatically ignored by `*.json` rule in .gitignore. Detection threshold: 0.001 seconds (prevents false negatives).

6.5.5 Performance Characteristics

Scenario	Files	Time	Cache?
Incremental (2 files changed)	2/28	≈3s	Yes
No changes since last run	0/28	≈1s	Yes
Full audit (`--force`)	28/28	≈45s	Reset

7 Change Detection System

7.1 Motivation

During development, most changes are localized to 1-3 files. Running all tests on all files is inefficient:

- Full test suite: ≈45s (lint + compliance + execution)
- Typical workflow: 3s with incremental detection
- **Speedup:** 15× for common case

7.2 User Interface

7.2.1 Incremental (Default)

```
$ ./tests/scripts/tests_start.sh
Mode: Incremental (changed files only)
Changed files: 2 (api/config.py, core/fusion.py)
RUNNING: Code Linting Checks
 PASSED

# Next run (no changes)
$ ./tests/scripts/tests_start.sh
Changed files: 0
RUNNING: Code Linting Checks
 PASSED (trivial, no files to check)
```

7.2.2 Full Audit

```
$ ./tests/scripts/tests_start.sh --force
Mode: FULL AUDIT (all files, cache reset)
Changed files: 28 (all)
RUNNING: Code Linting Checks
...
RUNNING: Policy Compliance Check
...
RUNNING: Code Structure Execution Tests
 ALL TESTS PASSED
```

7.2.3 Cache Management

```
# View cache status
python -c "from␣tests.scripts.scope_discovery␣import␣get_cache_info;␣\
␣␣␣␣␣␣␣␣␣␣import␣json;␣print(json.dumps(get_cache_info(),␣indent=2))"
# Output: {"cached_files": 28, "cache_size_bytes": 1369, "exists": true}

# Reset cache
./tests/scripts/tests_start.sh --reset-cache
# Output:   Cache reset

# Or programmatically
python -c "from␣tests.scripts.scope_discovery␣import␣reset_cache;␣
    reset_cache()"
```

7.3 Integration with Scripts

Each test script can optionally receive `--force-all` to bypass change detection:

```
# code_lint.py
if __name__ == "__main__":
    force_all = "--force-all" in sys.argv
    changed = discover_changed_files(force_all=force_all)
    # Process changed files...
```

8 Execution Flow

8.1 Complete Pipeline Diagram

The following describes the complete flow from invocation to results:

1. **User invokes:** `./tests/scripts/tests_start.sh [OPTIONS]`
2. **Orchestrator prepares:**
   - Verify .venv exists
   - Parse CLI options
   - Reset cache if `--force`
   - Create `tests/results/` and `tests/reports/` directories
3. **Test Phase 1 - Linting:**
   - `code_lint.py` discovers changed files (or all if `--force-all`)
   - Runs: flake8, black, isort, mypy
   - Generates: JSON result, Markdown report, console output
   - Fails if mandatory tool fails
4. **If Phase 1 passes, Test Phase 2 - Compliance:**
   - `code_alignement.py` executes 36 policy checks
   - Policies are repository-wide (not file-specific)
   - Generates: JSON result, Markdown report, console output
   - Fails if any policy violated
5. **If Phase 2 passes, Test Phase 3 - Execution:**

- `code_structure.py` runs 79 unit tests via pytest
- Uses real JAX with 64-bit precision
- Generates: JSON result, Markdown report, pytest output
- Fails if any test assertion fails

6. **After all phases:**
   - Print unified summary
   - List generated artifacts
   - Return exit code (0 = success, 1 = failure)

8.2 Fail-Fast Semantics

Pipeline stops on first failure:

```
$ ./tests/scripts/tests_start.sh
RUNNING: Code Linting Checks
 FAILED (exit code: 1)
 Stopping execution: linting checks failed
EXIT CODE: 1

# Phases 2 and 3 do NOT run
```

9 Reporting and Artifacts

9.1 Directory Structure

```
tests/
 scripts/
    tests_start.sh           # Orchestrator
    dependency_check.py      # Dependency validation (Stage 0)
    code_lint.py             # Linting phase (Stage 1)
    code_alignement.py       # Compliance phase (Stage 2)
    code_structure.py        # Execution phase (Stage 3)
    scope_discovery.py       # Discovery + change tracking
    __init__.py
 results/
    dependency_check_last.json # Latest dependency check results
    code_lint_last.json      # Latest linting results
    code_alignement_last.json # Latest compliance results
    code_structure_last.json  # Latest execution results
    .scope_cache.json         # Change detection cache
 reports/
     dependency_check_last.md  # Human-readable dependency report
     code_lint_last.md         # Human-readable linting report
     code_alignement_last.md   # Human-readable compliance report
     code_structure_last.md    # Human-readable execution report
```

9.2 Report Formats

9.2.1 JSON Format

All JSON reports follow a consistent structure:

```
{
  "timestamp": "2026-02-21 14:32:15 UTC",
  "total_checks": 36,
  "passed_checks": 36,
  "failed_checks": 0,
  "results": [
    {
      "tool": "flake8",
      "passed": true,
      "details": "Checked 28 files"
    },
    ...
  ]
}
```

Enables programmatic CI/CD integration: parse JSON to extract pass/fail status.

9.2.2 Markdown Format

Human-readable format with:

- Timestamp and scope information
- Summary statistics
- Detailed results table
- Recommendations for failures

Example excerpt:

```
# Code Quality Linting Report
**Generated:** 2026-02-21 14:32:15 UTC
**Result:** 4/4 checks passed

## Linting Results
| Tool | Status | Details | Files |
|------|--------|---------|-------|
| flake8 |  PASS | Code style OK | 28 |
| black |  PASS | Formatting OK | 28 |
| isort |  PASS | Import ordering OK | 28 |
| mypy |  PASS | Type checking OK | 28 |
```

9.3 Log Retrieval

9.3.1 Most Recent Results

```
# View latest linting results (pretty-printed)
python -m json.tool tests/results/code_lint_last.json

# View latest compliance results
python -m json.tool tests/results/code_alignement_last.json

# View latest execution results
python -m json.tool tests/results/code_structure_last.json
```

9.3.2 Markdown Reports

```
# Display in terminal
cat tests/reports/code_lint_last.md

# Open in your editor
code tests/reports/code_lint_last.md
```

10 Command Reference

10.1 Complete CLI Options

Option	Behavior
(no option)	Run all phases: lint → compliance → execute (incremental)
`--all`	Explicit form of default behavior
`--force`	Reset cache, full audit (all files), all phases
`--lint`	Linting phase only (incremental)
`--compliance`	Compliance phase only
`--execute`	Execution phase only
`--reset-cache`	Clear change detection cache and exit (no tests run)
`--help`	Display usage information

10.2 Practical Workflows

10.2.1 Developer Workflow (Iterative)

```
# Modify a few files
nano Python/api/config.py
nano Python/core/orchestrator.py

# Quick test (only changed files)
./tests/scripts/tests_start.sh --lint

# If linting passes, check compliance
./tests/scripts/tests_start.sh --compliance

# If compliance passes, run full test
./tests/scripts/tests_start.sh --execute
```

10.2.2 Pre-Commit Workflow

```
# Before committing, run full audit
./tests/scripts/tests_start.sh --force

# If all pass, safe to commit
git add .
git commit -m "Feature␣X␣implementation"
```

10.2.3 CI/CD Integration

```
#!/bin/bash
set -e  # Exit on error

# Always run full audit in CI
./tests/scripts/tests_start.sh --force --all

# Parse JSON results for metrics
python scripts/parse_test_results.py tests/results/code_lint_last.json
python scripts/parse_test_results.py tests/results/code_alignement_last.
    json
python scripts/parse_test_results.py tests/results/code_structure_last.json

echo "All␣tests␣passed!␣Safe␣to␣merge."
```

11 Troubleshooting

11.1 Common Issues

11.1.1 Issue: Python virtual environment not found

Error Message:

```
ERROR: Python virtual environment not found at .venv/bin/python
```

Solution:

```
python3 -m venv .venv
source .venv/bin/activate
pip install -r requirements.txt
./tests/scripts/tests_start.sh --reset-cache
```

11.1.2 Issue: Cache inconsistent or corrupted

Symptom: Tests report 0 files changed but you modified files.

Solution:

```
./tests/scripts/tests_start.sh --reset-cache
./tests/scripts/tests_start.sh
```

11.1.3 Issue: Linting reports false positives

Symptom: `flake8` complaints that seem unrelated.

Solution:

```
# Auto-fix formatting issues
python -m black Python/ --line-length=120
python -m isort Python/ --profile=black

# Re-run linting
./tests/scripts/tests_start.sh --lint
```

11.1.4 Issue: Tests timeout

Symptom: `pytest` or `mypy` hangs indefinitely.
Solution:

```
# For pytest (default timeout: 60s per test)
./tests/scripts/tests_start.sh --execute

# For mypy (default timeout: 180s)
# These are longer on first run due to cache building
python -m mypy Python/ --ignore-missing-imports
```

11.1.5 Issue: Memory errors during execution tests

Symptom: JAX kernel tests crash with OOM (Out of Memory).
Solution:

```
# Limit JAX memory allocation
export JAX_PLATFORM_NAME=cpu
export JAX_CPU_FALLBACK=true
./tests/scripts/tests_start.sh --execute
```

12 Implementation Notes

12.1 Design Decisions

12.1.1 Why Auto-Discovery?

Hardcoding module lists creates maintenance burden:

- Adding new module requires updating multiple test scripts
- Risk of stale lists causing missed tests
- Violates DRY principle

`scope_discovery.py` solves by:

- Scanning `Python/` for `__init__.py` files
- Dynamically building module list at runtime
- Works immediately for new modules (zero config)

12.1.2 Why Change Detection?

During development, running all 79 tests on every keystroke is excessive:

- Most changes are localized (1-3 files)
- Full test suite: 45s
- Typical workflow: 3s incremental

Implementation trade-offs:

- **Pros:** 15× speedup, instant feedback
- **Cons:** Cache can desync if external tools modify files
- **Mitigation:** `--force` and `--reset-cache` always available

12.1.3 Why Separate Scripts?

Three distinct phases (lint, compliance, execution) instead of monolithic script:

- **Modularity:** Each script has single responsibility
- **Reusability:** Can invoke scripts independently
- **Parallelization:** Future: run phases in parallel (with proper ordering)
- **Debugging:** Easier to isolate failures

12.2 Extension Points

12.2.1 Adding New Linting Tool

```
# In code_lint.py, add new function
def run_custom_linter() -> LinterResult:
    try:
        result = subprocess.run(
            ["custom-linter", PYTHON_DIR, ...],
            capture_output=True, timeout=60
        )
        return LinterResult(
            linter="custom",
            passed=(result.returncode == 0),
            violations=...,
            details=...
        )
    except Exception as e:
        return LinterResult(linter="custom", passed=False, ...)

# In main(), add to results list
results.append(run_custom_linter())
```

12.2.2 Adding New Policy

```
# In code_alignement.py, add to policy_checks()
(39, "My New Policy", lambda: check_my_new_policy()),

# Implement check function
def check_my_new_policy() -> Tuple[bool, str]:
    if condition_met():
        return True, "Policy satisfied"
    else:
        return False, "Policy violation: ..."
```

12.2.3 Adding New Test Module

New modules in `Python/` are automatically discovered:

```
mkdir -p Python/mymodule
touch Python/mymodule/__init__.py
# Add tests to code_structure.py (search for "TestMyModule")

./Test/scripts/tests_start.sh --execute
# mymodule tests now included automatically!
```

13 Performance Characteristics

13.1 Benchmarks

Measurements on a typical development machine (MacBook Pro M1, 16GB RAM):

Phase	Full (–force-all)	Incr (2 files)	Incr (0 files)	Unit
Linting	8.2s	2.1s	0.8s	sec
Compliance	12.5s	$<$1s*	$<$0.5s*	sec
Execution	24.3s	24.3s†	24.3s†	sec
Total	45.0s	26.4s	25.6s	sec

* = `code_alignement.py` now uses cache effectively: only checks changed files (13-50× speedup).
† = `code_structure.py` runs all 79 tests regardless (pytest fixture dependencies prevent selective execution).

13.2 Optimization Opportunities

1. **Parallel Phases (Future):** Run lint + compliance in parallel (if dependencies allow)
2. **Test Sharding:** Distribute 79 tests across multiple processes
3. **Cached JAX Models:** Pre-compile JAX kernels (requires XLA cache management)
4. **Type Cache:** mypy maintains cache, but rebuild on major changes

14 Future Enhancements

14.1 Planned Features

1. **Parallel Execution:** Run phases concurrently with proper dependency management
2. **Coverage Reporting:** pytest-cov integration for code coverage metrics
3. **Performance Tracking:** Historical benchmark comparisons
4. **Pre-Commit Hooks:** Git integration to auto-run tests before commits
5. **Web Dashboard:** Real-time test results with history

14.1.1 Known Limitations

1. **Change Detection:** Cannot detect changes made by external tools without filesystem mtime update (workaround: `--force-all`)
2. **code_structure.py Cache:** Reports changed files but executes all tests (pytest fixture dependencies prevent selective execution)
3. **Parallel Execution:** Not yet supported (would require refactoring orchestrator)
4. **Remote Testing:** All tests assume local environment (no SSH/network testing)
5. **Windows Support:** Shell scripts bash-specific (would need PowerShell port)

15 Conclusion

The Universal Stochastic Predictor's testing infrastructure provides:

- **Three-phase validation:** Code quality → Policy compliance → Functional correctness
- **Zero-hardcoding:** Dynamic module discovery adapts to new modules automatically
- **Intelligent caching:** 15× faster typical workflow vs. full audit
- **Comprehensive reporting:** Machine-readable JSON + human-readable Markdown
- **Extensible design:** Easy to add tools, policies, and tests

This architecture supports both development-time rapid iteration and pre-release full audits, balancing productivity with rigor.

Last Updated: February 21, 2026
Test Framework Version: 2.1.0
Specification Version: v2.1.0-Diamond-Spec