

# Universal Stochastic Predictor Testing Infrastructure Implementation

Quality Gates Framework with Decentralized Reporting

Development Team

Document Version: 2.2.2  
Last Updated: 2026-02-22  
Test Framework: v2.2.2

## Abstract

This document describes the **v2.2.2 implementation** of the Universal Stochastic Predictor's testing infrastructure. The system implements a fail-fast quality gate architecture with decentralized reporting:

- **Quality Gates:** Two-stage pipeline (GATE 1: Dependencies, GATE 2: Code Quality)
- **Decentralized Reporting:** Each tool generates JSON, then auto-generates Markdown
- **3-Level Classification:** BLOCKING (won't execute) / ERROR (executes but buggy) / WARNING (style)
- **Relative Path Format:** All file paths relative to project root for portability
- **Audited Stages:** GATE 1 (`dependency_check.py`) and GATE 2 (`code_lint.py`) fully audited
- **Temporal Exit:** After GATE 2, test generation and pytest execution remain unaudited

Status: GATE 1 PASSED (0 blocking, 0 errors, 2 warnings) · GATE 2 PASSED (0 blocking, 0 errors, 47 lint warnings)

## Contents

<b>1 System Overview</b>	<b>2</b>
1.1 Architecture Summary . . . . .	2
1.2 Current Status (v2.2.2 - Audited Stages Only) . . . . .	2
<b>2 GATE 1: Dependency Validation</b>	<b>2</b>
2.1 Location and Purpose . . . . .	2
2.1.1 Execution Flow . . . . .	2
2.1.2 Issue Classification . . . . .	2
2.1.3 Output Format (JSON) . . . . .	3
2.1.4 Latest Results . . . . .	3
<b>3 GATE 2: Code Quality Control</b>	<b>3</b>
3.1 Location and Purpose . . . . .	3
3.1.1 Validation Pipeline . . . . .	3
3.1.2 Issue Classification . . . . .	3
3.1.3 Output Format (JSON) . . . . .	4
3.1.4 Latest Results . . . . .	4

<b>4 Report Generation Framework</b>	<b>4</b>
4.1 Decentralized Architecture . . . . .	4
4.1.1 Pipeline . . . . .	5
4.1.2 Markdown Sections . . . . .	5
<b>5 Output Directory Structure</b>	<b>5</b>
5.1 Artifacts Location . . . . .	5
5.2 Cleanup Behavior . . . . .	5
<b>6 Coming Next: Remaining Stages</b>	<b>5</b>
<b>7 Usage</b>	<b>6</b>
7.1 Running Quality Gates . . . . .	6
<b>8 Path Format Convention</b>	<b>6</b>
<b>9 Conclusion</b>	<b>6</b>

# 1 System Overview

## 1.1 Architecture Summary

The v2.2.2 testing infrastructure comprises:

Component	Location	Purpose
Orchestrator	<code>Test/run_tests.py</code>	Entry point with quality gates
GATE 1	<code>Test/scripts/dependency_check.py</code>	Dependency validation (AUDITED)
GATE 2	<code>Test/scripts/code_lint.py</code>	Code quality check (AUDITED)
Reporting	<code>Test/framework/reports.py</code>	Markdown generation engine
Reports	<code>Test/reports/</code>	Markdown outputs (auto-generated)
Results	<code>Test/results/</code>	JSON intermediates (auto-generated)

## 1.2 Current Status (v2.2.2 - Audited Stages Only)

Stage	Status	Details
GATE 1: Dependencies	AUDITED	0 blocking, 0 errors, 2 warnings
GATE 2: Code Quality	AUDITED	0 blocking, 0 errors, 47 lint warnings
Test Generation	PENDING	Will audit scaffold next
pytest Execution	PENDING	Will audit after test generation

After GATE 2, execution stops (temporal exit) until remaining stages are audited and brought into production.

# 2 GATE 1: Dependency Validation

## 2.1 Location and Purpose

`Test/scripts/dependency_check.py` — Validates all Python imports against declared requirements.

### 2.1.1 Execution Flow

1. Parse `Python/requirements.txt` and `Test/requirements.txt`
2. Scan all `.py` files in `Python/` for `import` statements
3. Flag any imports not in the requirements (MISSING)
4. Flag any requirements not imported (UNUSED)
5. Output: JSON payload and auto-generated Markdown report

### 2.1.2 Issue Classification

Uses 3-level hierarchy with tool prefix [dependency]:

- **BLOCKING:** Missing critical dependency (breaks import)
- **ERROR:** Unused requirement (bloat)
- **WARNING:** Transitive dependency issue

### 2.1.3 Output Format (JSON)

```

1  {
2      "stage": "GATE_1",
3      "tool": "dependency_check.py",
4      "result": {
5          "total_requirements": 22,
6          "total_imports": 45,
7          "summary": {"blocking": 0, "error": 0, "warning": 2},
8          "issues": [
9              {
10                 "level": "warning",
11                 "category": "[dependency]",
12                 "message": "Unused requirement: scipy",
13                 "file": null
14             }
15         ]
16     }
17 }
```

### 2.1.4 Latest Results

Metric	Count
Total Requirements	22
Total Imports	45
Blocking Issues	0
Errors	0
Warnings	2

## 3 GATE 2: Code Quality Control

### 3.1 Location and Purpose

`Test/scripts/code_lint.py` — Unified code quality validator integrating black, isort, flake8, and mypy.

#### 3.1.1 Validation Pipeline

1. **Format Check (black):** Enforce PEP 8 formatting
2. **Import Sort (isort):** Organize imports alphabetically
3. **Lint Check (flake8):** Detect style violations and errors
4. **Type Check (mypy):** Validate type annotations against runtime

#### 3.1.2 Issue Classification

Uses 3-level hierarchy with tool-specific prefixes:

- **BLOCKING:** Type mismatch incompatible with JAX operations
- **ERROR:** Syntax or import errors
- **WARNING:** Style or lint issues (code still executes)

Tool prefixes for filtering:

Prefix	Tool
[format]	black formatting
[isort]	Import organization
[lint]	flake8 violations
[type]	mypy type checking

### 3.1.3 Output Format (JSON)

```

1  {
2      "stage": "GATE_2",
3      "tool": "code_lint.py",
4      "result": {
5          "total_files": 27,
6          "summary": {"blocking": 0, "error": 0, "warning": 47},
7          "breakdown": [
8              {
9                  "tool": "flake8",
10                 "issues": [
11                     {
12                         "level": "warning",
13                         "category": "[lint]",
14                         "message": "E501: Line too long (>88 chars)",
15                         "file": "Test/scripts/code_structure.py",
16                         "line": 42
17                     }
18                 ]
19             }
20         ]
21     }
22 }
```

### 3.1.4 Latest Results

Metric	Count
Total Files Scanned	27
Blocking Issues	0
Errors	0
Warnings	47

## 4 Report Generation Framework

### 4.1 Decentralized Architecture

Each GATE generates its own report in two formats:

1. **JSON:** Structured payload for machine parsing
2. **Markdown:** Human-readable report with sections

Location: `Test/framework/reports.py` — Rendering engine for JSON-to-Markdown conversion.

### 4.1.1 Pipeline

```

1 # Step 1: GATE 1 collects issues
2 gate1_issues = dependency_check.run()
3
4 # Step 2: GATE 1 writes JSON
5 save_json(gate1_issues, 'Test/results/dependency_check_last.json')
6
7 # Step 3: GATE 1 renders Markdown
8 render_report(gate1_issues, 'Test/reports/DEPENDENCY_CHECK_LAST.md')

```

### 4.1.2 Markdown Sections

Auto-generated reports include:

- **Header:** Gate name, timestamp, version
- **Summary:** Blocking/Error/Warning counts
- **Breakdown:** Per-tool results with issue details
- **Classification:** Hierarchical grouping by level

## 5 Output Directory Structure

### 5.1 Artifacts Location

```

1 Test/
2   results/                                # JSON payloads (machine)
3     dependency_check_last.json            # GATE 1 JSON
4     code_lint_last.json                  # GATE 2 JSON
5   reports/                                # Markdown reports (human)
6     DEPENDENCY_CHECK_LAST.md            # GATE 1 report
7     CODE_LINT_LAST.md                  # GATE 2 report
8   run_tests.py                           # Orchestrator (cleans & runs)

```

### 5.2 Cleanup Behavior

**Policy:** Every execution of `run_tests.py` removes all prior JSON and Markdown files.

```

1 # Before GATE 1 starts, cleanup:
2 rm -f Test/results/*.json
3 rm -f Test/reports/*.md

```

This ensures:

- No stale reports from previous runs
- Each execution produces fresh snapshots
- Clear audit trail of "latest" state

## 6 Coming Next: Remaining Stages

The v2.2.2 release includes audited implementation for:

- **GATE 1:** Dependency validation
- **GATE 2:** Code quality control

Planned for future audits:

- **GATE 3:** Test execution
- **GATE 4:** Performance profiling
- **GATE 5:** Security scanning
- **GATE 6:** Documentation validation

Each remaining GATE will follow the same 3-level classification scheme and auto-generated Markdown reporting pattern established in v2.2.2.

## 7 Usage

### 7.1 Running Quality Gates

```
1 # Execute both GATE 1 and GATE 2
2 python Test/run_tests.py
3
4 # Output:
5 # GATE 1: Dependency Validation -> JSON + Markdown
6 # GATE 2: Code Quality Control -> JSON + Markdown
7 # (Exits temporally after GATE 2, remaining gates pending)
8
9 # View reports
10 cat Test/reports/DEPENDENCY_CHECK_LAST.md
11 cat Test/reports/CODE_LINT_LAST.md
```

## 8 Path Format Convention

All file paths in reports are **relative to project root**:

- Python/api/config.py (relative)
- NOT /absolute/path/to/config.py (absolute)

This ensures portability and clarity across development environments.

## 9 Conclusion

The v2.2.2 quality gates framework establishes:

- **GATE 1 (Audited):** Dependency validation—22 packages, 45 imports, 2 warnings
- **GATE 2 (Audited):** Code quality control—27 files, 47 lint warnings, 0 blocking issues
- **Decentralized Reporting:** Each gate auto-generates JSON + Markdown
- **3-Level Classification:** BLOCKING, ERROR, WARNING across all gates
- **Relative Paths:** All reports use project-root-relative paths for portability

Future releases will extend this framework to include GATE 3–6 (test execution, profiling, security, documentation) using the same patterns established in v2.2.2.

Last Updated: February 22, 2026  
Infrastructure Version: 2.2.2  
Framework: pytest 7.3.0 + Quality Gates  
Specification: v2.1.0-Quality Gates (Audited)