ENGINEERING STANDARDS LIBRARY / SOFTWARE ENGINEERING

Python Engineering, Type Safety, and Automation Standard

Typed Python, deterministic automation, dependency control, testability, packaging, and maintainable scripting at enterprise scale.

PERMANENT DOCUMENT ID

MYTHOS-SWE-0004

PUBLICATION

Candidate Standard
Version 1.0.0-candidate

ASSURANCE PROFILE

Standard / Advanced
High Assurance

PERMANENT PUBLICATION IDENTITY

Python Engineering, Type Safety, and Automation Standard

DOCUMENT ID
MYTHOS-SWE-0004

VERSION
1.0.0-candidate

STATUS
Candidate Standard

PUBLISHER
Mythos Systems

PUBLICATION DATE
2026-07-24

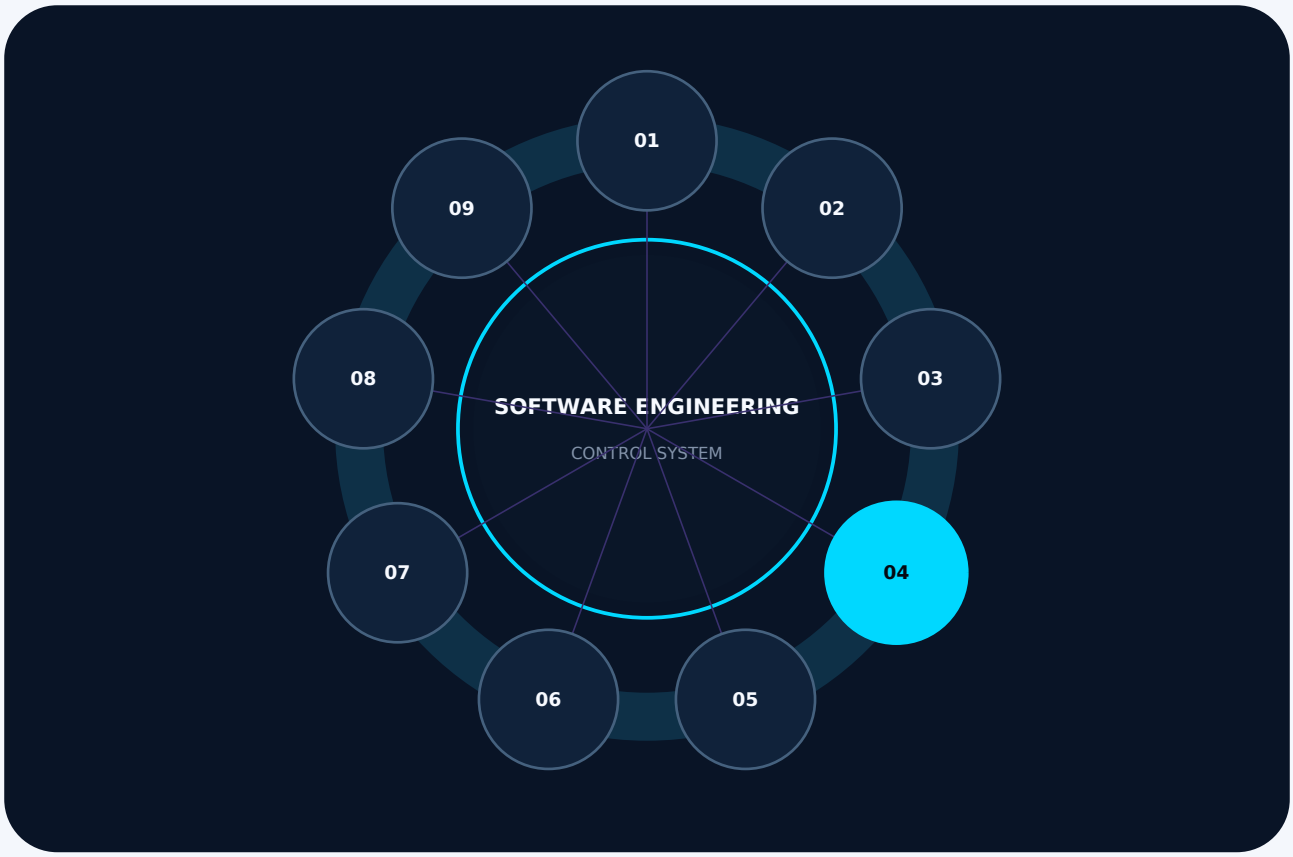
SCHEDULED REVIEW
2027-01-24

16	6	4	1
ATOMIC CRITERIA	ASSURANCE VIEWS	IMPLEMENTATION PROFILES	PERMANENT IDENTITY

Publication doctrine. This publication establishes durable guidance or requirements for its defined scope. Interpretation must preserve the permanent document identity, recorded evidence, controlled exceptions, and formal revision history.

DOMAIN CONTROL SYSTEM

MYTHOS-SWE-0004 inside the software engineering standard constellation



Connected assurance	Architecture, implementation, operations, evidence, and lifecycle are evaluated as one controlled system.
Specialization rule	Companion standards may add precision, but they cannot silently weaken this publication's requirements.

EXECUTIVE POSITION

Python Engineering, Type Safety, and Automation Standard

The deployment of Python in production infrastructure has failed.

Evidence-bearing delivery path

Assurance maturity spectrum

CONTENTS

Publication navigation

MYTHOS-SWE-0004 inside the software engineering standard constellation . . 3

Python Engineering, Type Safety, and Automation Standard 4

Assurance model and evaluation surface 7

Criteria and evidence map 8

Requirements, evaluation methodology, and implementation guidance 9

Executive Mandate 9

Scope and Applicability 9

In Scope 9

Out of Scope 10

Audience 10

Definitions and Taxonomy 10

Python Dimensions 10

The Python Doctrine 10

Evaluation Methodology 10

Scoring Model 10

Weighting 11

Evaluation Categories 11

Type Safety and Strict Mode (Weight: 25%) 11

Static Type Enforcement 11

`Any` Quarantine 11

Data Modeling and Validation (Pydantic) (Weight: 20%) 12

Schema Enforcement 12

Serialization Determinism 12

Dependency and Packaging (Weight: 20%) 12

Dependency Locking 12

Environment Isolation 13

Async and Concurrency (Weight: 15%) 13

Structured Concurrency 13

CPU-Bound Offloading 13

Linting and Code Quality (Weight: 10%) 14

Linting and Formatting . . . 14
Performance and Native Extensions (Weight: 10%) . . . 14
Performance Profiling . . . 14

The Mythos Python Suite (M-PY-S) . . . 14
Suite Composition . . . 14
M-PY-S-Types . . . 14
M-PY-S-Models . . . 14
Execution Protocol . . . 14

Implementation evidence and review gates . . . 15
Current authority register . . . 16

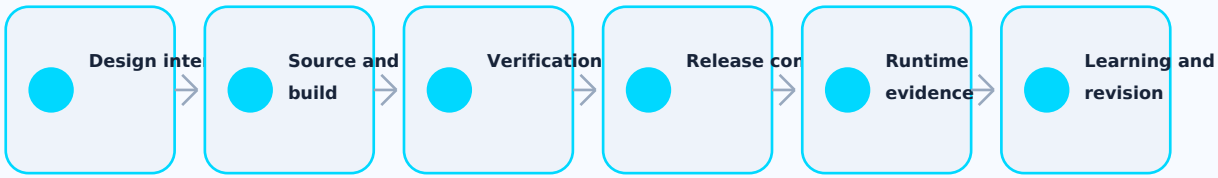
CONTROL TOPOLOGY

Assurance model and evaluation surface

The standard is assessed as an evidence-bearing system. Capability scores do not replace mandatory gates, implementation proof, or operational validation.

Capability family	Evaluation nodes
Type Safety and Strict Mode (Weight: 25%)	2
Data Modeling and Validation (Pydantic) (Weight: 20%)	2
Dependency and Packaging (Weight: 20%)	2
Async and Concurrency (Weight: 15%)	2
Linting and Code Quality (Weight: 10%)	1
Performance and Native Extensions (Weight: 10%)	1
Suite Composition	2
Additional Evaluation Dimension	4

Lifecycle control loop



EVALUATION INDEX

Criteria and evidence map

#	Capability family	Criterion
1	Type Safety and Strict Mode (Weight: 25%)	Static Type Enforcement
2	Type Safety and Strict Mode (Weight: 25%)	`Any` Quarantine
3	Data Modeling and Validation (Pydantic) (Weight: 20%)	Schema Enforcement
4	Data Modeling and Validation (Pydantic) (Weight: 20%)	Serialization Determinism
5	Dependency and Packaging (Weight: 20%)	Dependency Locking
6	Dependency and Packaging (Weight: 20%)	Environment Isolation
7	Async and Concurrency (Weight: 15%)	Structured Concurrency
8	Async and Concurrency (Weight: 15%)	CPU-Bound Offloading
9	Linting and Code Quality (Weight: 10%)	Linting and Formatting
10	Performance and Native Extensions (Weight: 10%)	Performance Profiling
11	Suite Composition	M-PY-S-Types
12	Suite Composition	M-PY-S-Models
13	Additional Evaluation Dimension	Python Dimensions
14	Additional Evaluation Dimension	The Python Doctrine
15	Additional Evaluation Dimension	Scoring Model
16	Additional Evaluation Dimension	Suite Composition

NORMATIVE PUBLICATION

Requirements, evaluation methodology, and implementation guidance

The following body preserves the substantive source standard while normalizing publication status, identity, navigation, and visual presentation.

Executive Mandate

The deployment of Python in production infrastructure has failed.

Python is the lingua franca of AI and automation, yet it is treated as a scripting language. Organizations deploy Python agents that accept `Dict[str, Any]` as inputs, rely on runtime duck-typing to catch errors, and manage dependencies via unpinned `requirements.txt` files. This results in runtime `AttributeError` and `KeyError` exceptions that crash automation pipelines, hallucinate infrastructure configurations, and exfiltrate data through unvalidated schemas.

This standard exists because:

1. A Runtime TypeError is an Architectural Failure: If a type error can reach production, the development pipeline is broken. Python's dynamic nature makes it exceptionally powerful, but also exceptionally dangerous without rigorous static analysis (`mypy`, `pyright`).
2. `Dict[str, Any]` is a Liability: Passing untyped dictionaries through agent pipelines is the root cause of schema drift and silent data corruption. All domain boundaries MUST use strictly typed models (Pydantic).
3. `Any` is a Betrayal: Using `typing.Any` disables the type checker. It is the `unsafe` of Python. It MUST be quarantined and explicitly justified.
4. Automation Code is Production Code: A Python script that pushes a firewall rule has the same blast radius as a compiled C program. It MUST be held to the same rigor: strict typing, locking, testing, and deterministic dependency management.

This document establishes the Mythos Python Engineering Standard (MPES), a comprehensive, evidence-based methodology for evaluating Python codebases against type safety, dependency determinism, and automation rigor.

Scope and Applicability

In Scope

This standard covers the evaluation of:

1. Static type checking (`mypy --strict`, `pyright`)
1. Strict data modeling and validation (Pydantic v2)
1. Dependency management and locking (uv, Poetry, PEP 621)
1. Packaging and distribution (wheel, sdist, PyO3/maturin)

1. Async concurrency and structured concurrency (asyncio, anyio)
1. Code quality and linting (Ruff)
1. Performance profiling and native extensions (Cython, Rust)

Out of Scope

1. General domain-driven design (covered in MYTHOS-SWE-0001)
1. Rust engineering internals (covered in MYTHOS-SWE-0002)
1. General supply chain security (covered in MYTHOS-SEC-0002)

Audience

1. Python engineers and ML/AI developers
1. Principal architects selecting languages for automation/agent planes
1. Security teams evaluating Python supply chain risks
1. Platform engineering teams building Python tooling
1. Standards bodies seeking a reference Python methodology

Definitions and Taxonomy

Python Dimensions

Dimension	Definition
Strict Typing	The enforcement of type hints across the entire codebase using `mypy --strict` or `pyright --strict`, disallowing untyped definitions and `Any`.
Pydantic v2	A data validation library using Rust (pydantic-core) for parsing and validation, ensuring runtime schemas match static types.
Deterministic Dependencies	The practice of locking all dependencies (including transitive) to exact versions and hashes (`uv.lock`, `poetry.lock`) to ensure reproducible builds.
Type-State Modeling	A pattern where the state of an object is encoded in its generic type parameters (e.g., `Agent[State.Uninitialized]`), making invalid states unrepresentable.

The Python Doctrine

> A runtime TypeError is an architectural failure. A dictionary is not a domain model. Any is a betrayal. Automation code is production code. If it runs in production, it must be typed.

Evaluation Methodology

Scoring Model

Each capability is scored from 0 to 5.

Score	Meaning
0	Fails basic task; no typing, unvalidated dicts, unpinned deps

Score	Meaning
1	Basic type hints but not enforced; partial validation
2	Functional but lacks strict mode or Pydantic boundaries
3	Solid production performance; `mypy --strict`, Pydantic v2, locked deps
4	Strong performance; type-state modeling, Rust extensions, Ruff
5	Best-in-class; sets the standard for deterministic, typed Python

Weighting

Category	Weight	Rationale
Type Safety & Strict Mode	25%	Runtime type errors are silent killers
Data Modeling & Validation (Pydantic)	20%	Unvalidated dicts cause schema drift
Dependency & Packaging	20%	Unpinned deps cause supply chain attacks and drift
Async & Concurrency	15%	Blocking I/O kills performance; unbounded async kills availability
Linting & Code Quality	10%	Inconsistent style hides bugs
Performance & Native Extensions	10%	CPU-bound Python blocks the event loop

Total: 100%

A codebase MUST score at least 3.0 in Type Safety and Data Modeling to be considered for production use.

Evaluation Categories

Type Safety and Strict Mode (Weight: 25%)

Static Type Enforcement

Is the codebase strictly typed and enforced in CI?

Score	Criteria
0	No type hints; dynamic typing everywhere
1	Type hints exist but not enforced; `Any` used frequently
2	`mypy` or `pyright` run locally but not in CI
3	`mypy --strict` or `pyright --strict` mandatory in CI
4	Zero `Any` usage without explicit `# type: ignore` with justification
5	Perfect enforcement: strict mode, zero `Any`, type-state modeling, typed generics, no `cast` hacks

`Any` Quarantine

Is `typing.Any` isolated and justified?

Score	Criteria
0	`Any` used as default for complex types
1	`Any` used in external API boundaries only
2	`Any` replaced with `object` or generic `T` where possible
3	`Any` requires code review justification
4	`Any` restricted to deserialization boundaries; immediately validated into typed model
5	Perfect quarantine: zero `Any` in domain logic, formally verified type boundaries

Data Modeling and Validation (Pydantic) (Weight: 20%)

Schema Enforcement

Are all domain boundaries and API contracts validated using strict models?

Score	Criteria
0	`Dict[str, Any]` or raw JSON passed between functions
1	`dataclasses` used but no runtime validation
2	Pydantic models used for API boundaries
3	Pydantic v2 with `strict=True` mode globally
4	Custom Pydantic types for domain primitives (e.g., `IPv4Address`, `SVID`)
5	Perfect enforcement: Pydantic v2 strict, type-state modeling via generics, zero unvalidated dicts in codebase

Serialization Determinism

Are serialization/deserialization deterministic and versioned?

Score	Criteria
0	`json.loads` into dict; no schema
1	Pydantic serialization but no versioning
2	Versioned schemas via model validators
3	Backward/forward compatibility tested
4	Deterministic serialization (sorted keys, explicit excludes)
5	Perfect determinism: formally verified schema evolution, zero breaking changes, cryptographic hashing of serialized state

Dependency and Packaging (Weight: 20%)

Dependency Locking

Are dependencies strictly locked and hashed?

Score	Criteria
0	`requirements.txt` without versions
1	Pinned versions in `requirements.txt`
2	`poetry.lock` or `uv.lock` committed

Score	Criteria
3	Lock file includes transitive dependencies and hashes
4	Automated dependency scanning (SBOM generation) in CI
5	Perfect locking: `uv` with strict hashes, air-gapped registry support, zero floating versions, automated CVE blocking

Environment Isolation

Are Python environments deterministic and isolated?

Score	Criteria
0	Global Python install; `pip install` locally
1	`virtualenv` used manually
2	`direnv` or `venv` automated
3	`uv` for fast, deterministic environment creation
4	Docker images with multi-stage builds; no build tools in runtime
5	Perfect isolation: `uv`, reproducible builds, Nix-like determinism, zero "works on my machine"

Async and Concurrency (Weight: 15%)

Structured Concurrency

Is async code structured and bounded?

Score	Criteria
0	Synchronous blocking I/O in services
1	`asyncio` used but unstructured `asyncio.create_task`
2	`anyio` or `asyncio.TaskGroup` (Python 3.11+) used for structured concurrency
3	Bounded semaphores for concurrent operations
4	Cancellation explicitly handled; resources cleaned up on `CancelledError`
5	Perfect discipline: structured concurrency, bounded, cancellation-safe, zero leaked tasks

CPU-Bound Offloading

Are CPU-bound tasks offloaded to avoid blocking the event loop?

Score	Criteria
0	CPU-intensive parsing/computation in async functions
1	`asyncio.to_thread` used for blocking calls
2	`ProcessPoolExecutor` for CPU-bound work
3	Rust extensions (PyO3) for performance-critical paths
4	Subprocess isolation for untrusted code execution
5	Perfect offloading: Rust/compiled extensions for hot paths, zero event loop blocking

Linting and Code Quality (Weight: 10%)

Linting and Formatting

Is the codebase consistently formatted and linted?

Score	Criteria
0	No linters or formatters
1	`flake8` and `black` used inconsistently
2	`ruff` used for linting and formatting in CI
3	`ruff` with strict rule set (e.g., `ALL` with specific ignores)
4	Custom `ruff` rules enforcing project standards
5	Perfect quality: zero lint warnings, automated formatting, custom rules enforcing type safety and domain invariants

Performance and Native Extensions (Weight: 10%)

Performance Profiling

Are performance bottlenecks identified and resolved with compiled code?

Score	Criteria
0	No profiling; blind optimization
1	`cProfile` used occasionally
2	Continuous profiling in staging (e.g., `py-spy`, `austin`)
3	Hot paths identified and optimized with pure Python
4	Cython or C-extensions for critical paths
5	Perfect performance: PyO3/Rust extensions for all hot paths, formally benchmarked, zero GIL contention

The Mythos Python Suite (M-PY-S)

Traditional Python benchmarks measure test coverage. The M-PY-S evaluates type strictness, schema validation, and dependency determinism.

Suite Composition

M-PY-S-Types

1. Strict Mode: 100+ tests running `mypy --strict` with zero errors.
1. Any Detection: 50+ tests scanning for `Any` usage outside explicit quarantine zones.

M-PY-S-Models

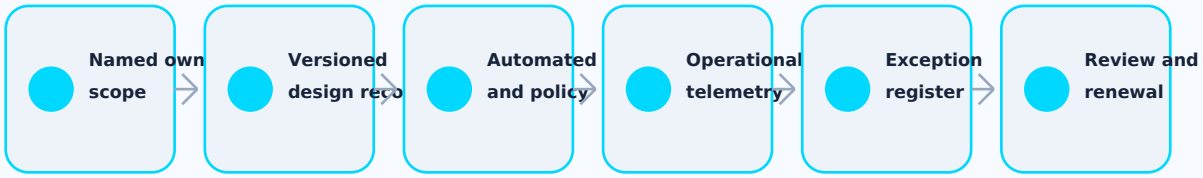
1. Dict Leak: 100+ tests scanning for `Dict[str, Any]` in function signatures.
1. Pydantic Strict: 50+ tests verifying `model_config = ConfigDict(strict=True)`.

Execution Protocol

ASSURANCE AND ADOPTION

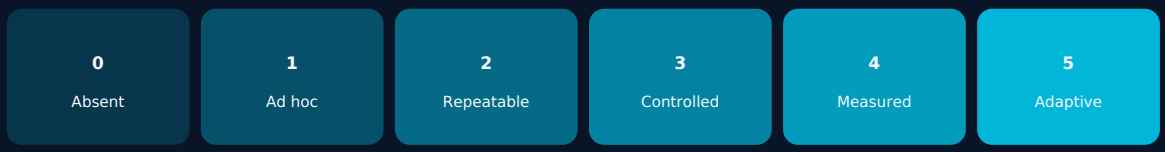
Implementation evidence and review gates

Required assurance chain



A passing score is not sufficient without current evidence. High-assurance adoption requires traceable design decisions, reproducible builds, independently reviewable tests, controlled secrets, runtime observability, failure exercises, and a dated exception register.

Assurance maturity spectrum



Current authority register

Authority	Version / snapshot	Applicability	Review by
Rust Project: The Rust Programming Language and Rust Reference	Rolling official documentation; book assumes Rust 1.90.0 or later and Rust 2024 Edition	High	2026-10-22
Go Project: The Go Programming Language Specification and Effective Go	Rolling official documentation snapshot	High	2026-10-22
Python Software Foundation: Python Language Reference, Packaging, Typing, and PEP Index	Rolling official documentation snapshot	High	2026-10-22
Tauri Programme: Tauri 2 Documentation	Tauri 2 rolling documentation snapshot	High	2026-10-22
Svelte Project: Svelte Documentation	Svelte 5 rolling documentation snapshot	High	2026-10-22
TLA+ Foundation: TLA+ Language and Tools Documentation	Rolling documentation snapshot	High	2027-01-24
OpenAPI Initiative: OpenAPI Specification	3.2.0	High	2027-01-24
OpenTelemetry Project: OpenTelemetry Documentation and Specification	Rolling specification snapshot; profiles public alpha and declarative configuration stable in 2026	High	2026-10-22

Research limitation. Official documentation establishes published interfaces and declared behavior. It does not prove the performance, security, interoperability, cost, or operational suitability of a specific implementation. Those claims require controlled testing and retained evidence.