CHECKLIST

Python Checklist: Production Readiness Review

A practical Python checklist for validating code quality, security, packaging, observability, and deployment readiness before production use.

Programming - July 03, 2026

Purpose

Python projects often pass local tests but still fail in production because packaging is incomplete, dependencies are unmanaged, secrets leak into code, or operational checks are missing. A production-ready Python checklist turns those risks into verifiable review items so engineers can decide whether a service, script, or library is ready to release.

Use this checklist to confirm whether a Python application is safe to deploy, easy to operate, and recoverable when something goes wrong. After reading, you should be able to judge whether the checklist applies to your project, follow a practical validation workflow, and verify the evidence needed before production use.

How to use this checklist

Work through the phases in order. For each phase, verify the checklist items, collect the evidence, and record whether the acceptance criteria are met. If a phase fails, fix the issue before moving on; do not compensate for a missing control in one phase by over-scoring another.

For larger platform work, a readiness structure similar to an operational review checklist can help keep the review evidence-based instead of opinion-based. The goal is not perfection; it is proving that the Python component can be built, tested, deployed, monitored, and rolled back with acceptable risk.

Phase 1: Scope and runtime baseline

Purpose: confirm what is being reviewed, where it runs, and which runtime assumptions are allowed. This prevents ambiguous reviews where code, infrastructure, and release responsibilities are mixed together.

Checklist items

- ? Confirm the Python version(s) supported by the application and document the minimum and maximum tested versions.
- ? Review the operating system, container image, or server baseline used in production and document any runtime constraints.
- ? Validate whether the application is a web service, batch job, CLI tool, library, or scheduled task, and align controls to that use case.
- ? Assign ownership for code, deployment, secrets, and runtime operations.
- ? Document external dependencies such as databases, queues, object storage, identity providers, and third-party APIs.
- ? Review whether any version-specific behavior, optional C extensions, or platform-specific packages affect supportability.
- ? Confirm that the release scope is explicit: what changed, what did not change, and what is out of scope.

Evidence to capture

- Supported version matrix

- Deployment target description
- Architecture or dependency diagram
- Ownership record or RACI-style note
- Release notes or change summary

Acceptance criteria

The scope is unambiguous, the runtime baseline is documented, and the team can explain what must be verified before production use without relying on tribal knowledge.

Owner

Engineering lead or service owner.

Review cadence

Review on every release that changes runtime, packaging, or production dependency assumptions; otherwise review quarterly.

Phase 2: Source control and change integrity

Purpose: ensure the codebase has traceable, reviewable changes and that production releases are built from approved sources. This phase protects against drift between source code and deployed artifacts.

Checklist items

- ? Confirm all production code changes are tracked in version control.

- ? Review branch protection, required approvals, and merge rules for the mainline branch.
- ? Validate that release tags or commit identifiers can be mapped to a build artifact.
- ? Document how hotfixes, emergency changes, and rollback commits are approved.
- ? Assign responsibility for approving release merges and release notes.
- ? Test that the build pipeline pulls from the expected repository and branch.
- ? Review whether generated files, lock files, and vendored code are intentionally committed and kept in sync.

Evidence to capture

- Pull request or merge approval record
- Tag-to-build mapping
- Build pipeline reference
- Change log or release notes
- Repository policy settings

Acceptance criteria

Every production artifact can be traced to an approved source revision, and unauthorized or unreviewed changes are not part of the release path.

Owner

Engineering lead with code review owners.

Review cadence

Review for every release and after any repository policy change.

Phase 3: Dependencies and packaging

Purpose: verify that dependencies are reproducible, pinned where needed, and compatible with the runtime baseline. Dependency drift is one of the most common reasons Python systems behave differently in production than in test.

Checklist items

- ? Confirm dependency management uses a repeatable method such as a lock file, constraints file, or pinned build manifest.
- ? Validate that installation is reproducible from a clean environment.
- ? Review whether build-time and runtime dependencies are separated where appropriate.
- ? Document any packages that require native compilation, external libraries, or OS-level headers.
- ? Confirm that private package indexes, mirrors, or artifact repositories are reachable in the deployment environment.
- ? Test that dependency installation succeeds without relying on a developer workstation state.
- ? Review whether optional dependencies are enabled only when explicitly required.
- ? Validate that package metadata, entry points, and import paths resolve correctly.

Evidence to capture

- Lock file or pinned dependency manifest
- Reproducible build log
- Dependency tree output
- Packaging configuration
- Artifact repository access record

Acceptance criteria

A clean build produces the same dependency set every time, and the application starts without hidden dependency assumptions.

Owner

Build and release owner.

Review cadence

Review whenever dependencies change, a new Python version is introduced, or the packaging method changes.

Phase 4: Code quality and test coverage

Purpose: verify that the code behaves as intended under normal, edge, and failure conditions. This phase should prove that the implementation is testable, not just that it imports successfully.

Checklist items

- ? Confirm unit tests cover the critical logic paths and error handling branches.
- ? Review integration tests for database, queue, file system, and API interactions that matter in production.
- ? Validate that test data is realistic enough to exercise meaningful behavior without exposing sensitive information.

- ? Test that the application fails safely when upstream services are unavailable or return invalid data.
- ? Document any known gaps in coverage and assign an owner for remediation.
- ? Review whether static analysis, type checking, or linting gates are required for merges.
- ? Confirm that tests run in automation and are not dependent on manual setup.
- ? Validate that smoke tests cover the main deployment path and startup sequence.

Evidence to capture

- Test run report
- Coverage summary or test matrix
- Static analysis results
- CI job logs
- Known gaps register

Acceptance criteria

Critical paths are covered, failure handling is exercised, and the automated test suite runs reliably in CI.

Owner

Application team or service owner.

Review cadence

Review on every change set and after any test framework or CI change. If your project is model-driven or data-heavy, a structure similar to a Learning Checklist for AI and Machine Learning Projects can also help separate functional validation from release readiness.

Phase 5: Security and secrets handling

Purpose: confirm the application does not expose secrets, execute unsafe input, or rely on insecure defaults. Python services often fail this phase because environment variables, config files, and dependency transitivity are treated too casually.

Checklist items

- ? Confirm secrets are not stored in source files, notebooks, fixtures, or build artifacts.
- ? Review how secrets are injected at runtime and whether rotation is supported.
- ? Validate that input validation exists for external data, file names, paths, and query parameters.
- ? Test that command execution, file writes, and deserialization paths are restricted to trusted inputs.
- ? Document dependency vulnerability scanning and how findings are triaged.
- ? Confirm that logging redacts tokens, passwords, personal data, and other sensitive fields.
- ? Review whether environment-specific security settings are required and documented.
- ? Validate that least-privilege permissions are used for the application identity and file system access.

Evidence to capture

- Secret handling design note
- Runtime configuration example
- Vulnerability scan output

- Redaction or logging policy
- Permission review record

Acceptance criteria

Secrets are handled outside source control, sensitive data is protected in logs and transport, and obvious injection or deserialization risks are addressed.

Owner

Security owner or application security reviewer.

Review cadence

Review on every release, after dependency updates, and after any change to auth, secrets, or data handling.

Phase 6: Configuration and environment management

Purpose: ensure the application can move between environments without manual edits or hidden defaults. Production readiness depends on predictable configuration, not just correct code.

Checklist items

- ? Confirm configuration values are externalized and environment-specific.
- ? Review defaults to ensure they are safe if a required variable is missing.
- ? Validate that configuration is documented and versioned where appropriate.
- ? Test startup with production-like configuration values in a non-production environment.

- ? Assign ownership for environment variables, config files, and secret references.
- ? Document any feature flags, toggles, or runtime switches that affect behavior.
- ? Verify that configuration validation fails fast on invalid or incomplete settings.
- ? Review whether timezone, locale, encoding, and file path assumptions are explicit.

Evidence to capture

- Example configuration file or template
- Environment matrix
- Startup validation logs
- Feature flag inventory
- Configuration owner record

Acceptance criteria

The application starts consistently with documented configuration and fails early when required settings are missing or invalid.

Owner

Platform or service owner.

Review cadence

Review on every environment change, release, or infrastructure migration.

Phase 7: Observability and operational visibility

Purpose: verify that operators can detect, diagnose, and respond to issues after deployment. Without logs, metrics, and health checks, a Python service may be running but still be effectively invisible.

Checklist items

- ? Confirm structured logging is enabled and includes correlation identifiers where appropriate.
- ? Review whether log levels are meaningful and do not overproduce noisy output.
- ? Validate that health checks distinguish between liveness, readiness, and dependency availability when needed.
- ? Test that key operational metrics are emitted for request rate, error rate, latency, queue depth, or job progress as applicable.
- ? Document alert thresholds and the owner for each actionable alert.
- ? Confirm that logs and metrics are retained long enough for incident investigation and trend analysis.
- ? Review whether tracing or request correlation is required for distributed workflows.
- ? Validate that operational dashboards answer the questions an on-call engineer would ask during an incident.

Evidence to capture

- Sample log output
- Dashboard screenshots or export
- Health check behavior summary
- Alert routing record
- Metric names and definitions

Acceptance criteria

Operators can detect failure, identify the affected component, and see enough context to start recovery without guessing.

Owner

Operations owner or on-call owner.

Review cadence

Review on every release that changes instrumentation and during incident postmortems.

Phase 8: Deployment and rollback safety

Purpose: ensure releases can be deployed predictably and reversed safely. A Python application that cannot be rolled back quickly creates unnecessary production risk.

Checklist items

- ? Confirm the deployment method is documented and repeatable.
- ? Review whether the deployment is blue-green, rolling, canary, or single-step, and document the failure mode.
- ? Validate that the artifact built in CI is the same artifact deployed to production.
- ? Test rollback using the most recent previous known-good release.
- ? Document database migration behavior, especially whether migrations are reversible.
- ? Assign an approver for production promotion and rollback authorization.
- ? Confirm the deployment includes post-deploy validation checks.
- ? Review whether rollback depends on external state that cannot be restored quickly.

Evidence to capture

- Deployment runbook
- Artifact checksum or digest reference
- Rollback test record
- Migration plan
- Post-deploy validation checklist

Acceptance criteria

A release can be promoted and reverted with documented steps, and the team knows which changes are not safely reversible.

Owner

Release manager or platform owner.

Review cadence

Review every release and after any change to deployment tooling or migration strategy.

Phase 9: Data protection and compliance controls

Purpose: verify that the application handles data according to its sensitivity and regulatory requirements. This matters even for internal tools because Python scripts often process exports, logs, and ad hoc datasets.

Checklist items

- ? Confirm the data types processed by the application are documented.
- ? Review whether personal data, credentials, financial data, or regulated data are stored, transformed, or transmitted.
- ? Validate that retention, deletion, and archival rules are defined.
- ? Document encryption requirements for data in transit and at rest.
- ? Test that access to sensitive datasets is restricted to approved identities and roles.
- ? Confirm that audit logging exists where required by policy or regulation.
- ? Review whether data masking or minimization is applied in non-production environments.
- ? Validate that export paths, reports, and backups do not leak sensitive content.

Evidence to capture

- Data classification note
- Retention policy mapping
- Access control record
- Encryption settings
- Audit log sample

Acceptance criteria

Data handling aligns with documented classification and policy requirements, and sensitive data is not exposed through logs, exports, or backups.

Owner

Security, privacy, or compliance owner.

Review cadence

Review on any data model change, new data source, or compliance review cycle.

Common mistakes to watch for

A checklist is only useful when it catches predictable failure patterns. The most common mistakes in Python production reviews are easy to miss because the code still ?works? in a developer environment.

- Assuming pip install on a workstation proves packaging is reproducible in CI or production.
- Shipping unpinned dependencies and discovering breakage after a transitive update.
- Treating tests as complete when only happy-path unit tests exist.
- Logging secrets, tokens, or payloads with sensitive fields intact.
- Leaving configuration implicit and relying on defaults that differ by environment.
- Deploying code without a tested rollback path.
- Skipping health checks because manual validation ?usually works.?
- Ignoring platform differences between Linux, macOS, Windows, containers, and serverless runtimes.

Pass/fail criteria

Use clear rules so the review result is defensible and repeatable.

Pass

- All checklist items in the required phases are verified or an approved exception is documented.
- Evidence is available for the build, test, security, deployment, and rollback controls.
- No critical issues remain open.
- The owner for each open follow-up item is named and the due date is recorded.

Conditional pass

- Non-critical gaps exist, but each has a documented workaround, owner, and deadline.
- The team agrees the residual risk is acceptable for the release scope.
- The release can be limited to non-sensitive or low-risk traffic if applicable.

Fail

- Any critical secret-handling, dependency, rollback, or access-control issue remains unresolved.
- The build cannot be reproduced from approved sources.
- The release cannot be monitored or rolled back with confidence.
- Required evidence is missing and cannot be reconstructed from automation or logs.

Readiness scoring

A simple score helps compare releases and identify recurring weak spots. Score each phase from 0 to 2.

- 0 = Not verified or failed
- 1 = Partially verified, with evidence gaps or open follow-ups
- 2 = Fully verified and documented

Scoring interpretation

- 16?18: Ready for production release, assuming no critical exceptions
- 12?15: Conditionally ready; release only after closing the highest-risk gaps
- 8?11: Not ready; the checklist has too many unresolved risks
- 0?7: Stop and remediate before any production promotion

Track the score over time to spot patterns. Repeated low scores in dependency management, observability, or rollback usually indicate process problems rather than isolated defects.

Practical review record

Keep the review output short and explicit. A good record shows what was checked, what failed, and who owns the correction.

A Python production review should end with a clear decision, not a vague sense that things ?look good.? If the checklist passes, you have evidence that the code, runtime, and operations model are ready. If it fails, the missing control should be fixed and re-verified before the release proceeds.

Use this guidance together with .NET checklist to connect the workflow with related operational context already available on the site.

Use this guidance together with ASP checklist and algorithms best practices to connect the workflow with related operational context already available on the site.

Related guides in this cluster

- Python JSONDecodeError Troubleshooting for Invalid API Responses

Related guides in this cluster

- Python Security Checklist for Safe File and Data Handling

Related guides in this cluster

- Python Multiprocessing: Avoid Deadlocks in Concurrent Code

Related guides in this cluster

- Python AST Parsing for Secure Code Analysis and Auditing

Related guides in this cluster

- How to Parse JSON in Python with Type Hints

Related guides in this cluster

- Python Memory Profiling with tracemalloc and objgraph
- Python Socket Programming Tutorial for Network Communication

Related guides in this cluster

- Python Regex Validation for Secure Log Parsing and Data Extraction
- Python Logging Best Practices for Secure Production Systems

Related guides in this cluster

- Python Requests Tutorial: Secure API Calls with TLS Verification

Related guides in this cluster

- Python Tutorial: Parse and Validate JSON with Pydantic
- Python Thread-Safe Logging with QueueHandler and QueueListener

Related guides in this cluster

- Python Asyncio Timeout Handling for Reliable Network Tasks

Related guides in this cluster

- Python Reporting Template FAQ

Related guides in this cluster

- Python Tutorial: Parse JSON Logs with Pandas and Regex

Related guides in this cluster

- Python Logging Best Practices for Secure, Scalable Applications

Related guides in this cluster

- Python Type Hints for Secure API Input Validation