

# Backend: Python

- [Developer Best Practices Guide](#)

# Developer Best Practices Guide

## CONTRIBUTING.md

## Contributing Guide

This repository follows strict coding, review, testing, and release standards. Every contributor (AI agent) must follow the rules below.

---

### 1) Coding Standards

#### 1.1 General rules

- Use the **Pythonic** way of writing code
- Follow **DRY** (Don't Repeat Yourself)
- Keep logic simple and easy to understand
- Do not overcomplicate the implementation
- Prioritize readability and maintainability
- Write production-ready code

#### 1.2 Software engineering principles

- Follow **SOLID** principles wherever applicable
- Write clean, modular, testable code
- Prefer composition and clear abstractions over deeply coupled logic
- Separate concerns properly (API, service, repository, utility, model, constants, etc.)

## 1.3 Object-oriented programming

- Follow **OOP** standards where appropriate
- Keep classes focused on a single responsibility
- Avoid god classes / oversized service classes
- Use inheritance only when it is truly needed

## 1.4 Design patterns

- Use the correct design pattern where applicable
- Prefer clarity over pattern overuse
- Use **Factory Pattern** for common integrations across multiple cloud/external applications

## 1.5 Asynchronous programming

- Use **asynchronous programming** (`async` / `await`) where applicable
  - Do not block async workflows with unnecessary synchronous operations
  - Keep async usage consistent across the call chain
- 

## 2) Project Structure Rules

- Each Python class must be placed in a **separate** `.py` file
- Follow a clean and modular folder structure
- Use `snake_case` for:
  - file names
  - method names
  - variable names
  - function names
- Keep file responsibilities limited and clear
- Avoid putting unrelated classes or utilities into the same file

## Recommended separation

- `api/` route or controller layer
- `service/` business logic
- `repository/` DB interaction
- `models/` schemas/entities/domain models
- `constants/` all static messages and constants

- `tests/` pytest-based tests
- 

## 3) Constants and Messages

- Every user-facing or system message must come from a **constants file**
- Do **not** hardcode messages directly in business logic
- Keep error messages, labels, and static responses centralized
- Reuse constants to maintain consistency across the codebase

### Not allowed

```
raise ValueError("Invalid request")
```

### Preferred

```
raise ValueError(ErrorMessages.INVALID_REQUEST)
```

---

## 4) Docstrings and Documentation in Code

- Every Python file must contain proper docstrings where needed
- Add docstrings for:
  - modules
  - classes
  - public methods
  - non-trivial helper functions
- Keep docstrings clear and meaningful
- Explain intent, parameters, return types, and important side effects

# Minimum expectation

- What the class/function does
  - Input parameters
  - Return value
  - Exceptions raised (if relevant)
- 

## 5) Database Rules

### 5.1 Schema change restrictions

- Do **not** create a new DB table without discussion/approval
- Do **not** create a new DB column without discussion/approval

### 5.2 Documentation requirements for DB objects

- Every DB table must include a description of what it stores / why it exists
- Every DB column must include a description of what it stores / how it is used

### 5.3 Timestamps

- Any timestamp saved in the database must be stored in **UTC**

### 5.4 How DB changes must happen

Any DB update involving the following must happen through **Alembic**:

- schema change
- CRUD-related DB update scripts
- seed data insertion
- migration for reference/master data updates when applicable

### 5.5 Alembic note

- Use Alembic for all database migration work
  - Do not bypass migrations with manual DB changes in application code
- 

## 6) Testing Requirements

### 6.1 Framework

- Use **pytest** for all tests
- Add proper unit tests and integration tests wherever applicable

### 6.2 Coverage

- Code coverage must be **above 90%**
- PRs below the coverage threshold should not be considered ready for merge

### 6.3 Test quality expectations

- Test actual behavior, not implementation details only
  - Cover positive, negative, and edge cases
  - Mock external dependencies where appropriate
  - Keep tests readable and maintainable
- 

## 7) Linting and Code Quality Gates

### 7.1 Pylint

- Run pylint before marking the PR ready for merge
- Required pylint score: **above 9.30**

### 7.2 Command

Run:

```
run_pylint.bat
```

## 7.3 Alembic exception

- Alembic-related pylint issues may be ignored where already agreed

## 7.4 Readiness rule

A PR is not ready to merge unless:

- pylint score is acceptable
  - tests pass
  - coverage is above threshold
- 

# 8) Versioning and Release File Updates

For **every PR**, update the following files if the repository/version policy requires it:

- `application.yml`
- `pyproject.toml`
- `CHANGELOG.md`

## Versioning rules

- Version must remain **coherent/consistent** across all versioned files
- Do not update one file and forget the others
- Keep release notes aligned with the actual change set

## Author metadata

- Add yourself as author in `pyproject.toml` if you are contributing to the repository
-

# 9) Required PR Checklist

Use this checklist before marking the PR ready:

- ☐ Branch name follows convention
  - ☐ Latest changes pulled from `main`
  - ☐ Latest remotes fetched
  - ☐ One PR contains only one logical change
  - ☐ Code follows Pythonic style
  - ☐ DRY principle applied
  - ☐ Logic kept simple and readable
  - ☐ OOP standards followed
  - ☐ Correct design patterns used
  - ☐ Factory pattern used for common external/cloud integrations
  - ☐ Async/await used where applicable
  - ☐ Each class is in a separate `.py` file
  - ☐ `snake_case` naming followed
  - ☐ All messages come from constants file
  - ☐ Proper docstrings added
  - ☐ Proper pytest cases added
  - ☐ Code coverage 90%
  - ☐ Pylint score 9.30
  - ☐ `run_pylint.bat` executed
  - ☐ DB schema/table/column changes discussed if applicable
  - ☐ Table/column descriptions added if applicable
  - ☐ All DB timestamps stored in UTC
  - ☐ Any DB changes done via Alembic
  - ☐ `application.yml` updated
  - ☐ `pyproject.toml` updated
  - ☐ `CHANGELOG.md` updated
  - ☐ Version is coherent across files
  - ☐ Contributor added as author in `pyproject.toml` if applicable
  - ☐ Reviewers added
  - ☐ GitHub Copilot added as reviewer (for Python repos)
- 

# 10) AI Agent Instructions

If an AI agent is used to generate or modify code in this repository, it must follow all repository standards defined in this file.

# AI agent must do the following

- Read this file before generating code
- Follow Pythonic style
- Keep code simple and maintainable
- Apply DRY
- Use OOP and proper design patterns
- Use async code where appropriate
- Put each class in a separate file
- Use `snake_case`
- Use constants for messages
- Add docstrings
- Add pytest test cases
- Keep coverage above 90%
- Respect pylint requirements
- Update versioned files where required
- Update changelog where required
- Respect DB and Alembic rules strictly

# AI agent must never do the following

- Hardcode messages in logic
  - Introduce DB changes without instruction/discussion
  - Combine multiple unrelated features in one PR
  - Skip tests/docstrings/versioning updates
  - Ignore repository conventions defined here
- 

## 11) Recommended Commit Hygiene

While not mandatory unless otherwise specified, contributors are encouraged to:

- Write clean and meaningful commit messages
- Avoid noisy WIP commits in final PR history
- Squash/fixup commits if required by team process

Example commit messages:

fix: handle refresh token expiration in oauth service

feat: add async cloud provider factory implementation

refactor: simplify user onboarding validation flow

## 12) Quick Reference Summary

### Must follow

- Pythonic code
- DRY
- SOLID
- OOP
- Correct design patterns
- Async where applicable
- One class per file
- `snake_case`
- Constants file for all messages
- Docstrings in each file
- Proper pytest tests
- Coverage 90%
- Pylint 9.30
- UTC timestamps in DB
- Alembic for DB changes
- Update `application.yml`, `pyproject.toml`, `CHANGELOG.md`
- Add reviewers + GitHub Copilot reviewer
- One PR per logical change

### Must avoid

- Hardcoded messages
- Unapproved DB schema changes
- Multiple features in one PR
- Skipping lint/test/version updates
- Overcomplicated logic

---

# 13) Final Rule

If there is any conflict between convenience and these standards, follow these standards.

Quality, readability, consistency, and reviewability are mandatory for every contribution.