OS Write Up

# iae225 - TA Assignment Write Up

## Assignment Summary

### Summary

Welcome to your first major assignment in Open Source & Professional Software Development. Your team's mission is to create a professional-grade **template repository** for a modern Python project. This is not a throwaway exercise; the repository you build will serve as the foundation for all subsequent project work in this course.

This assignment focuses on establishing the **structure, processes, and tooling** that enable sustainable, high-quality software development. You will build a small, multi-component system, but the primary deliverables are the repository structure, the CI/CD pipeline, and a clear separation of concerns.

The task is to create the foundational components for an AI-powered email assistant: a **Mail Client Interface** and a **Gmail Implementation** for that interface, demonstrating the principles of dependency injection discussed in class.

Since this is your first big assignment, we're providing a bit more help than usual to help you get started (e.g. file locations, tool guides, and other general tips), but this will not be the case for future assignments.

### Learning Goals

* **Master the Toolchain:** Gain proficiency with uv for environment and dependency management, ruff for linting and formatting, mypy for static type checking, and pytest for testing (see below for some starter guides on the tools you will be using).
* **Implement Component-Based Architecture:** Structure your code into distinct, reusable components with clearly defined interfaces and implementations.
* **Understand and Apply Dependency Injection:** Decouple components by defining abstract interfaces and injecting concrete implementations at runtime.
* **Embrace Test-Driven Development:** Write unit tests for your interfaces before the implementation exists, using mocks to validate the contract.
* **Build a CI/CD Pipeline:** Configure CircleCI to automatically enforce code quality, run tests, and report on code coverage, creating a safety net for future development.
* **Practice Professional Documentation:** Learn to write clear README.md and component.md files that make your project understandable and usable by others.

## Best Practices and General Guidelines

In this course, we treat software development as "programming integrated over time." The greatest challenge we face is not writing code that works now, but managing the **complexity** that makes code difficult to change later. This assignment is your first exercise in **strategic**, *not tactical*, programming.

Our entire framework is designed to arm you with the mindset, architectural patterns, and collaborative processes required to combat this complexity and build sustainable, evolvable systems.

To this end, below are some core principles you should be familiar with from the lectures, but are also provided here for clarity.

You should feel comfortable with this before getting to the Requirements section below. If you are still confused, please do not hesitate to reach out to Professor Kamen, Professor Nikolai, or the TAs :)

### What is a “Component”?

The single most important thing you can learn about software design is to break large problems into smaller, more manageable pieces. For this class, we will call these pieces **components**. **A well-designed component is a self-contained unit of functionality that you can reason about in isolation**. The ultimate test is what Kamen calls the "forklift test": a good component is something you can easily lift out of this project and use in another one with minimal effort.

To achieve this, every component must have a clear separation between its **interface** (the 'what') and its **implementation** (the 'how'). The interface is a contract. It should be abstract, simple, and expose nothing about the underlying details. A consumer of your mail\_client\_api shouldn't know or care if the backend is Gmail or Hotmail. The implementation's job is to fulfill that contract. To keep the interface pure, we will use **dependency injection**. This means your implementation package will "inject" its functionality into the API package at runtime, rather than having the API depend on the implementation. Getting this right is the difference between a maintainable system and spaghetti code.

### Regarding Physical Structure

A project's directory layout is the physical manifestation of its architecture. A clean structure makes component boundaries obvious and reduces cognitive load. We enforce a src layout where each component is a distinct, self-contained package with its own pyproject.toml and README.md. This structure ensures that a component can be "forklifted" out of the project and reused elsewhere with minimal effort. Unit tests, which validate a component's internal logic, must live alongside the component's source code. In contrast, integration and end-to-end tests, which verify how components work together, belong in a top-level tests directory, treating the components as black boxes just as a final application would.

### On Entropy

The second pillar is the establishment of **rigorous, automated processes as guardrails against entropy**. Recognizing that human discipline is fallible under pressure, we use a toolchain that automates best practices. The mandatory use of uv for dependency management, ruff for linting, mypy for static analysis, and a comprehensive CI/CD pipeline with CircleCI is not about learning specific tools for their own sake. It's about instilling a culture where quality is not an afterthought but a prerequisite for every change. The emphasis on high test coverage and the integration of these checks directly into the development workflow ensures that structural integrity is continuously verified, transforming quality from a subjective debate into an objective, measurable, and enforceable standard.

### On Teamwork and Collaboration

Finally, our philosophy is grounded in the reality that software development is a **socio-technical system**. We extend the concept of structured design from code to human interaction. The emphasis on small, self-contained pull requests, detailed descriptions, and structured peer review is designed to manage the "cognitive load" of collaboration. By making changes digestible and their intent clear, the process becomes more efficient and less error-prone. The structure of this class, with its emphasis on teamwork and peer feedback, is a direct simulation of a professional environment. Success here is defined not just by what you build, but by how you build it with others.

## Deliverables

Your final submission will be a single GitHub repository configured as a **Project Template**. This repository must be self-contained and ready for a new developer to clone and begin working immediately.

### Repository & Tooling Setup

* **Git Configuration:**
  + A clean, well-documented pull request that merges all work from a feature branch into the root commit of your repository.
  + A .gitignore file properly configured for a Python project, excluding virtual environments, IDE files, and sensitive credentials (token.json, .env).
* **Standardized Templates:**  
  In a .github/ directory:
  + A pull\_request\_template.md that guides contributors.
  + Issue templates for both bug\_report and feature\_request.
* **Repository Structure:**
  + Create a root src directory. All components will live inside this directory.
  + Create a root tests directory. Integration and End-to-End tests will live here.
  + Create a .github/ directory for issue and pull request templates.
* **Root pyproject.toml:**
  + Initialized as a uv workspace.
  + **Workspace Members:** The [tool.uv.workspace] section must declare all component packages (e.g., src/mail\_client\_api).
  + **Tool Configuration:** All tool configurations (ruff, mypy, pytest, coverage.py) must be in this file.
    - **Ruff:** select = ["ALL"]. *All disabled rules must be justified with a comment.*
    - **MyPy:** strict = true.
    - **Coverage:** Pick some % threshold for code coverage and justify it.

### Documentation

Documentation is a primary design tool. The repository must include:

* **Root README.md:** A comprehensive guide detailing the template's purpose, setup (uv sync), and usage of all integrated tools.
* **component.md:** A design document in the root directory explaining the component architecture and the dependency injection strategy.
* **Component README.md files:** Each component package must have its own README.md describing its specific API, purpose, and usage.

### Component Creation

The repository must contain the following components, demonstrating a clear separation of interface and implementation.

#### Mail Interface Component (src/mail\_client\_api)

* A self-contained Python package with its own pyproject.toml and README.md.
* Defines abstract Message and MailClient Protocol classes.
* NOTE: **This package must have no dependencies on any concrete mail provider library.**

#### Gmail Implementation Component (src/mail\_client\_gmail\_impl):

* A self-contained Python package with its own pyproject.toml and README.md.
* Depends on mail\_client\_api as a workspace dependency.
* Provides a concrete GmailClient class that implements the MailClient protocol.
* **Performs dependency injection.**

### Testing

1. **Unit Tests:**
   * Located within each component's tests/ directory (e.g., src/mail\_client\_gmail\_impl/tests/).
   * Tests for the GmailClient must use mocks to isolate it from the live Google API.
2. **Integration Test:**
   * Located in the root tests/integration/ directory.
   * Must validate the dependency injection mechanism by using mail\_client\_api.get\_client() and receiving a GmailClient instance.
   * Must include at least one test that makes a real, authenticated call to the Gmail API (marked with @pytest.mark.integration). (You might want to ignore this test in CircleCI)

* **End-to-End (E2E) Test:**
  + Located in the root tests/e2e/ directory.
  + Provide one test that simulates the full application flow: performing a calculation, logging the action, and triggering a notification.

### CI/CD

The repository must have a working .circleci/config.yml pipeline that automates quality control.

* **Automated Jobs:** The pipeline must execute jobs for linting (ruff), static analysis (mypy), and unit testing (pytest).
* **Quality Gates:** The build **must fail** if any check fails or if code coverage is below a certain % threshold which *you must justify*.
* **Reporting:**
  + Test results must be parsed and displayed in the CircleCI **"Tests"** tab.
  + The coverage report must be uploaded as a browsable **artifact**.
  + **Links to successful/failed CircleCI builds and the coverage report must be included in your pull request description.**

## Tooling Guide (Python)

#### uv - The Project and Package Manager

* **What It Is:** An extremely fast, all-in-one tool for managing Python virtual environments and project dependencies.
* **Why We Use It:** uv replaces the need for separate tools like pip and venv. Its **workspace** feature is essential for our component-based architecture, allowing us to manage multiple local packages from a single uv.lock file. This guarantees that every team member has an identical, reproducible development environment.

#### ruff - The Linter and Formatter

* **What It Is:** An integrated tool for enforcing code style and detecting common programming errors.
* **Why We Use It:** ruff automates quality control. By configuring it in pyproject.toml with select = ["ALL"], we adopt a high standard by default. Disabling a rule becomes a conscious design decision that **must be justified with a comment**. This prevents style debates in code reviews and keeps the codebase consistent and predictable.

#### mypy - The Static Type Checker

* **What It Is:** A tool that enforces Python's type hints.
* **Why We Use It:** Type safety is not optional. mypy helps us build robust interfaces and catches an entire class of bugs before the code is ever run. We use strict = true to ensure all parts of our component contracts are explicit and well-defined.

#### pytest & coverage.py - The Testing Framework

* **What It Is:** pytest is the framework for writing and running our tests. coverage.py measures which lines of code our tests execute.
* **Why We Use It:** Tests are the only way to verify behavior and refactor with confidence. We will structure our tests into three distinct types:
  1. **Unit Tests:** Test a single class or function in isolation (using mocks).
  2. **Integration Tests:** Test how two or more components work together.
  3. **End-to-End Tests:** Test the complete application workflow from a user's perspective.

#### CircleCI - The Continuous Integration Pipeline

* **What It Is:** An automated service that runs our quality checks for every code change.
* **Why We Use It:** The CI pipeline is the ultimate, impartial gatekeeper. It protects the main branch by ensuring no code is merged unless it is formatted correctly, passes all type checks, and meets the required test coverage. It turns our quality standards from guidelines into enforceable rules.

#### mkdocs - The Documentation Generator

* **What It Is:** A fast, simple static site generator for building project documentation from Markdown files.
* **Why We Use It:** "Good code is not self-documenting" is a core tenet of this class. mkdocs elevates your documentation from simple Markdown files into a professional, searchable, and navigable website. It treats documentation as a first-class product that can be versioned and deployed alongside your code.

## Submission

1. Follow the instructions provided in class to create a pull request (PR) from a feature branch (e.g., hw1-submission) to a root branch in your repository.
2. Submit the link to this PR on Brightspace.

As a final reminder: **Working code isn't enough**.

Your goal is to create systems that are not only functional today but are also understandable, adaptable, and a source of productivity—not frustration—for the developers who will inherit them tomorrow.

This is a practical, industry-forged approach that values clarity, modularity, and disciplined collaboration as the essential ingredients for building software that lasts.