Checklist for Documentation

While you've noted documentation will be done later, core code annotations and inline explanations are still crucial during the development process to ensure maintainability and ease of understanding for future contributors. Below is a checklist to ensure foundational documentation is integrated into Orgo without disrupting current priorities.

---

1. Inline Code Comments

[ ] Function-Level Comments:

Add a concise docstring for every function to explain:

Purpose: What the function does.

Inputs: Expected parameters and their data types.

Outputs: Return values and their types.

Example Function Docstring

def parse\_email(email\_payload):

"""

Parses an email payload to extract subject, sender, and body.

Args:

email\_payload (dict): The raw email data containing headers and content.

Returns:

dict: A dictionary with 'subject', 'sender', and 'body' keys.

"""

return {

"subject": email\_payload["subject"],

"sender": email\_payload["from"],

"body": email\_payload["body"],

}

[ ] Inline Comments:

Add comments for complex logic or non-obvious code sections.

Use single-line comments to explain why (not just what) the code does.

Example Inline Comments

# Retry email fetch with exponential backoff to handle temporary server issues

for attempt in range(3):

try:

fetch\_email()

break

except ConnectionError:

time.sleep(2 \*\* attempt)

---

2. Metadata in Configuration Files

[ ] File Metadata:

Add a metadata section to all YAML configuration files with:

Version

Last updated date

Environment applicability (e.g., development, production)

Example Configuration Metadata

# version: "1.0"

# last\_updated: "2024-11-25"

# environment: "production"

smtp:

server: "<SMTP\_SERVER>"

port: 587

---

3. Modular Logging Annotations

[ ] Log Entries:

Add descriptive messages to log entries, detailing the action and context.

Include unique identifiers (e.g., task IDs, workflow names) in log messages for traceability.

Example Log Entry

[2024-11-25 14:00:00] Workflow: Maintenance | Task ID: 123 | Status: Assigned to john.doe@organization.com

---

4. Code Structure Documentation

[ ] File-Level Descriptions:

Add a brief header comment at the start of each file explaining:

The file’s purpose.

Key functions or classes it contains.

Example File Header

"""

email\_parser.py

Handles parsing and validation of email data for the Orgo system.

Functions:

- parse\_email: Extracts email details.

- validate\_email: Ensures email payloads meet system requirements.

"""

[ ] Folder-Level ReadMe Files:

Add a README.md to each directory with:

A brief overview of the folder's purpose.

A list of files with a one-line description for each.

Example Folder-Level README

/email/

This directory contains modules for handling email parsing, sending, and receiving.

Files:

- email\_parser.py: Extracts subject, sender, and body from email payloads.

- email\_sender.py: Sends emails using SMTP.

- email\_receiver.py: Fetches emails from an IMAP server.

---

5. Placeholder Standardization

[ ] Placeholder Comments:

Ensure all configuration placeholders (e.g., <SMTP\_SERVER>, <USERNAME>) are accompanied by comments explaining what they represent and example values.

Example Placeholder with Explanation

smtp:

server: "<SMTP\_SERVER>" # Address of the SMTP server (e.g., smtp.example.com)

port: 587 # Port for TLS encryption

---

6. Testing Documentation

[ ] Test Descriptions:

Add docstrings or comments to test cases explaining:

The scenario being tested.

Expected behavior or results.

[ ] Error Scenarios:

Document specific edge cases tested (e.g., invalid email payloads, missing task fields).

Example Test Case Description

def test\_parse\_email\_missing\_subject():

"""

Tests the parse\_email function with a payload missing the 'subject' field.

Expects a KeyError to be raised.

"""

with pytest.raises(KeyError):

parse\_email({"from": "user@example.com", "body": "Test body"})

---

7. Workflow Diagrams

[ ] Basic Diagrams (Optional at this stage):

Create visual representations of complex workflows (e.g., email-to-task routing, escalation processes) to clarify dependencies and data flow.

Tools:

Use lightweight tools like Lucidchart, draw.io, or even simple flowcharts.

---

8. Build Automation

[ ] Autogenerate API Docs:

Use tools like Swagger or FastAPI's built-in documentation generator for RESTful APIs.

[ ] Test Coverage Reports:

Generate test coverage reports (e.g., pytest-cov) to document which parts of the code are untested.

---

9. Security Annotations

[ ] Sensitive Data Handling:

Add comments in modules or scripts that interact with sensitive data (e.g., credentials, logs) explaining:

How data is encrypted or anonymized.

Why certain security measures are necessary.

Example Security Annotation

# Encrypt all sensitive fields before storing them in the database

encrypted\_password = encrypt(password)

---

10. Scalability Notes

[ ] Future Expansion Points:

Add comments or TODOs for areas that might need scaling or enhancement.

[ ] Version Compatibility:

Document version requirements for external dependencies (e.g., Python 3.10+).

Example Scalability Note

# TODO: Replace this static task queue with a distributed solution like RabbitMQ for better scaling.

---

Optimized Workflow

1. Write Inline Comments:

Start with inline comments for each module to clarify the logic.

2. Add Docstrings:

Add function-level docstrings for all newly written or edited code.

3. Iterate:

Review and refine comments periodically as the codebase evolves.

---

This checklist focuses on core code annotations and foundational documentation to ensure the system remains maintainable while formal documentation is postponed. Let me know if you'd like to focus on any specific component for inline annotations!