Planception: A Smart Task Manager for Multidisciplinary Minds

Terione R. - C. Martin

UAT

Table Of Contents

Planception: A Smart Task Manager for Multidisciplinary Minds

Executive Summary

Planception is a smart task management application designed to help individuals, particularly students, creators, and professionals with diverse workloads, organize and track assignments, projects, and deadlines. Unlike traditional to-do list applications, Planception emphasizes flexibility and scalability, supporting multidisciplinary users who often struggle with structure and prioritization. The project was guided by software engineering principles, including object-oriented design, the Unified Process (UP), and Agile iteration.

The application's first implementation provides a functional web-based interface for creating, updating, and deleting tasks, supported by a layered Model-View-Controller or MVC-style architecture. Core features include task creation, deadline tracking, and status updates, with the groundwork laid for future extensions such as categorization, reminders, and AI-powered planning assistance. The project's development process included requirements gathering, UML modeling, incremental implementation, and unit testing.

This report documents Planception's lifecycle from concept to implementation, including its requirements specification, architecture, testing results, and lessons learned. While the current version represents a foundational product, it demonstrates the viability of a structured, extensible task management system that can evolve into a robust personal productivity tool.

Introduction

Modern students and professionals face increasing demands on their time, often juggling multiple projects, classes, and personal commitments simultaneously. For individuals with attention and focus challenges, such as ADHD, the lack of structure in traditional task management tools can make it difficult to stay organized and productive. Planception was

developed to address this problem by providing a simple yet scalable task management application tailored for multidisciplinary users.

The purpose of Planceptioon is to help users organize tasks, track deadlines, and reduce the cognitive load of planning. The system is designed for clarity and ease of use, while also laying a foundation for future integration with AI assistants to provide personalized task recommendations and intelligent scheduling, down to how long each task would take to complete and where the best place is to complete said task (e.g., writing a short story for three hours at a cafe or coding an application at home PC for 5 hours). By emphasizing clean code, modular design, and iterative development, Plancepotion demonstrates how software engineering principles can be applied to create tools that improve daily productivity.

The report that follows outlines the requirements, design, implementation, and testing of Planception. It also reflects on the challenges encountered during the project and opportunities for future enhancement, highlighting how structured engineering approaches can transform an idea into a functional application.

Software Requirements Specification

Overview

Planception is a task management app designed to help individuals with diverse commitments and neurodivergent thinking patterns (especially ADHD) organize their projects and daily priorities. It is engineered to balance creativity and structure, allowing users to sort tasks by urgency, domain, and cognitive load. The application lays the foundation for a future AI assistant that will optimize schedules, recognize productivity patterns, and provide smart task suggestions.

Scope

Planception will allow users to create, organize, and prioritize tasks across multiple domains (e.g., school, creative work, business). The system will display tasks using filters (by category, due date, urgency) and highlight daily priorities.

What It Will Do:

- Add, update, and delete tasks
- Assign tasks to categories
- Set due dates and priority levels
- Filter/sort by urgency and domain
- Highlight “Today’s Tasks”
- Lay groundwork for AI integration (expandable task model, time analytics-ready)

What It Will Not Do (yet):

- Calendar or full scheduling integration (Google/Outlook)
- AI-based suggestions or time management
- Mobile app support (desktop-first design)
- User authentication/multiple profiles

Functional Requirements

1. FR1: The system shall allow users to create tasks with a title, due date, priority, and category.
2. FR2: The system shall display a list of tasks sorted by due date by default.
3. FR3: The system shall allow filtering tasks by category and/or priority.
4. FR4: The system shall highlight tasks that are due today or overdue.
5. FR5: The system shall allow users to edit or delete existing tasks.

6. FR6: The system shall validate that task names are not empty and that due dates are valid.
7. FR7: The system shall display a dedicated “Today’s View” for urgent tasks.
8. FR8: The system shall persist task data between sessions using a database.
9. FR9: The system shall support responsive layout for smaller windows.
10. FR10 (Stretch): The system shall support future integration of AI task suggestions via modular design.

Non-Functional Requirements

1. Usability: The system should be usable without training, even for neurodivergent users.
2. Performance: The application should respond to user actions in under 1 second.
3. Maintainability: Code should follow clean design patterns (MVC), enabling future AI expansion.
4. Accessibility: The interface should use clear fonts, high contrast, and keyboard navigation.
5. Portability: The system should work on all major browsers and desktop operating systems.

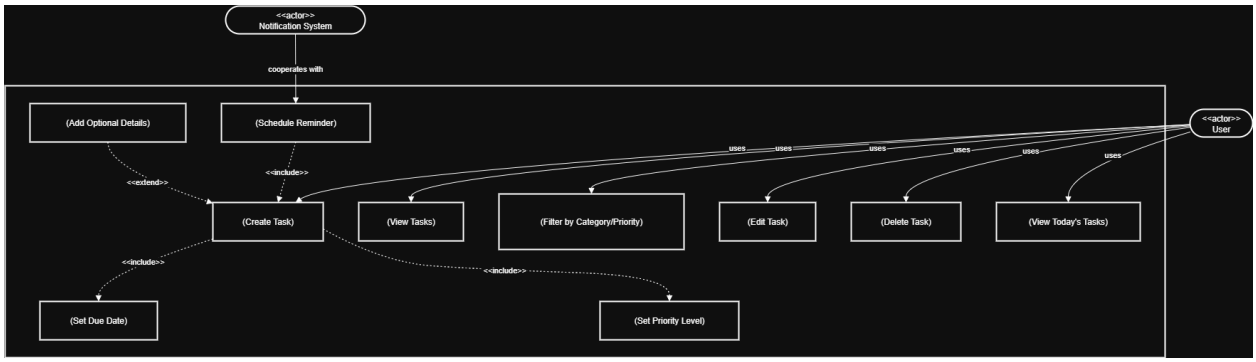
Use Case Diagram

Actor: User

Use Cases:

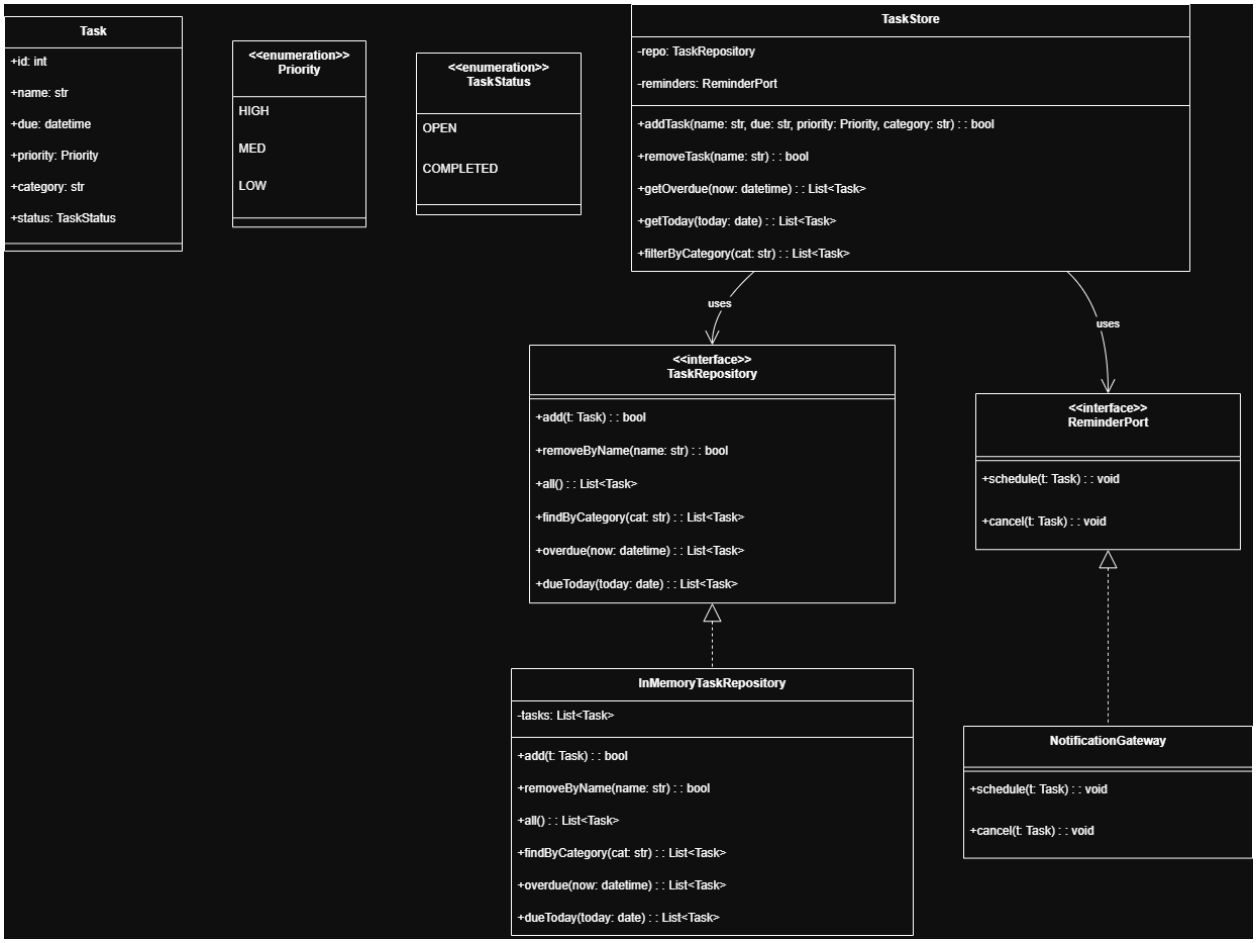
- Create Task
- Edit/Delete Task
- View Tasks

- Filter Tasks
- Sort Tasks
- Highlight “Today’s Tasks”



UML Class Diagram

- Task
    - id, name, due_date, priority, category, status
- TaskManager
    - add_task(), update_task(), delete_task(), filter_tasks(), get_today_tasks()
- UIController (optional)
    - handles interface rendering



Glossary and Assumptions

Glossary:

- Task: A user-defined activity that needs to be completed
- Priority: A flag indicating task importance (e.g., low, medium, high)
- Category: A domain such as "Writing," "School," or "Life"
- Today’s View: A filtered screen that shows tasks relevant to the current day
- Planception: The application name, inspired by “plan” + “Inception”

Assumptions:

- The user is a solo creator with many roles and overlapping projects.
- The user wants to avoid context-switching and overwhelm.

- Tasks do not need to be shared or synced between users (for MVP).
- The project will grow into a larger AI-based assistant.

Tech Stack

- Language: Python
- Framework: Flask (simple web app)
- Database: SQLite (lightweight, no setup overhead)
- Front-end: HTML/CSS + Bootstrap for quick UI
- Testing: pytest

System Architecture & Design

Architectural Overview

Planception follows a layered MVC-style architecture:

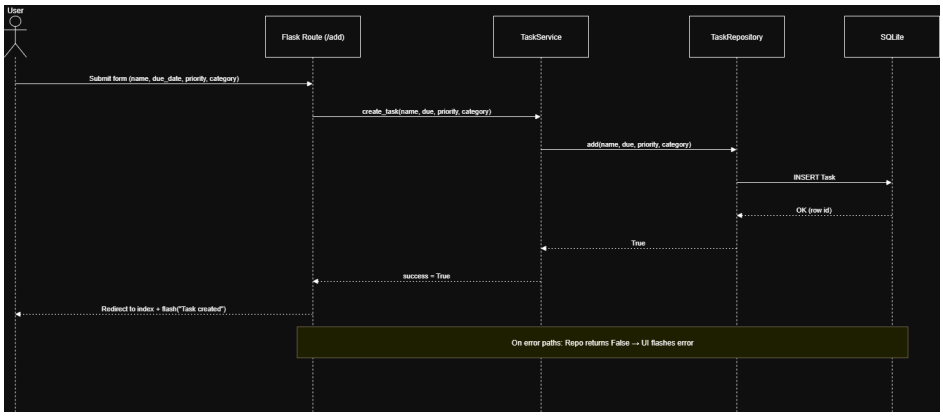
- **Presentation Layer (View)**: HTML templates ([templates/]) and CSS ([static/]) handle user interface and interactions.
- **Application Layer(Controller)**: [app.py] defines the Flask application factory, routes, and request handling.
- **Business Logic Layer (Service)**: [services.py] contains reusable logic (e.g., task creation, updates).
- **Data Access Layer (Model)**: [models.py] defines the Task model using SQLAlchem, and [repo.py] handles persistence.
- **Testing Layer**: Unit tests in [tests/] verify expected behaviors.

This modular approach improves maintainability (changes in one layer do not affect others), testability (unit tests can target isolated components), and scalability (new features like task categorization or AI prioritization can be added without rewriting core layers).

Sequence Diagram

A simple UML sequence diagram will show how a task flows through the system:

1. User submits a task via the web form.
2. Flask route in [app.py] receives input.
3. [services.py] validates and processes the task.
4. [repo.py] inserts the task into the SQLite database via SQLAlchemy.
5. User receives confirmation and sees the updated task list.



This diagram captures the runtime interaction for the ‘Create Task’ use case. The user submits a form to the Flask route, which delegates to [TaskService]. The service validates and orchestrates creation via [TaskRepository], which persists the new task to SQLite. Success or failure is returned to the UI to inform the user via flash messages.

Deployment Diagram

Deployment is simple for now:

- **Local Host (Development Machine)**: Runs Flask server with SQLite database stored in [instance/].
- **Future Deployment Target**: Could easily be deployed on Heroku, Render, or AWS EC2, with PostgreSQL as a production-ready database.

The diagram would show:

- One node for Web Browser (user)
- One node for Flask App (server)
- One node for Database (SQLite, local)

Current: 

Future: 

The MVP runs on a single machine. A web browser interacts with a local Flask app, and data is persisted to an on-disk SQLite database under the [instance/] directory. This layout keeps deployment simple while preserving a clean separation between application logic and data storage. A production-ready topology would deploy the Flask app to a cloud host and use a managed PostgreSQL database.

Database Schematic

Since we're using SQLite + SQLAlchemy, the core schematic is minimal but extensible:

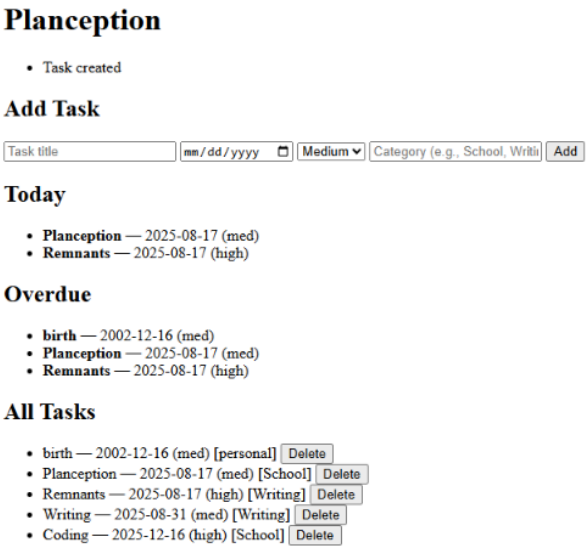
Table: Task

- [id] (Primary Key, Integer)
- [title] (String, required)
- [description] (String, optional)
- [due_date] (DateTime, optional)
- [status] (String, default: "pending")

- [created_at] (DateTime, auto)

Future additions might include: [priority], [tags], or [user_id] (if multi-user support is added).

Mockup & Screenshots



V1:

V2:

Implementation

Project Structure

Planception uses a layered, MVC-style layout. The goal is to keep UI, business logic, and data access separate so features can evolve without ripple effects.

planception/

├─ app.py # Presentation layer (Flask routes, app factory)

├─ services.py # Business logic orchestration (TaskService)

├─ repo.py # Data access (TaskRepository abstraction)

├─ models.py # Domain model (SQLAlchemy Task + enums)

```
├─ requirements.txt        # Dependencies
├─ instance/               # Runtime SQLite DB (created automatically)
├─ templates/
│  ├─ base.html            # Layout
│  └─ index.html           # Main UI (Add / Today / Overdue / All)
├─ static/
│  └─ styles.css           # Minimal styling
├─ tests/
│  └─ test_app.py          # Unit/integration tests (pytest)
└─ docs/
   └─ ... SRS, UML, testing report, user manual, screenshots
```

Lay mapping

- Presentation (UI): [app.py], [templates/], [static/]
- Application/Service: [services.py]
- Data/Repository: [repo.py], [models.py], [instance/]
- Quality/Docs: [tests/], [docs/]

Development Iterations

V0.5 - Code Inspiration/Example:

Goals: A small, basic version of the main program to help learn how it works and where I should start, along with understanding what a bad variation is and what a good refactored variation is.

Delivered:

- I got basic code for how a task management application should look and work.

- Application tests presenting the example inputs/outputs in the correct order, as if they were organized tasks.
- Presented a messy variation that was difficult to read and understand, allowing me to understand what not ot do.
- Provided a refactored variation that was easier to read and understand.

Outcome: I had the perfect foundation of what to do and what not to do, along with how the inputs and outputs are taken and presented, so I can work beyond that very foundation ot something better, like the UI we present in these first versions, with an actual webpage instead of being presented in the terminal.

V1 - MVP Scaffold:

Goals: Stand up a functional task manager with persistence and tests.

Delivered:

- Create/Delete tasks with due date, priority, and category
- Today/Overdue views derived from due dates
- SQLite persistence in [instance/] via SQLAlchemy
- Layered structure (UI → Service → Repository → DB)
- 5+ pytest tests: valid add, invalid priority, duplicate rejection, today/overdue display, delete
- Basic flash messages for user feedback

Outcome: A stable, demoable foundation aligned with the SRS core requirements.

V1.1 - Docs & Diagrams:

Goals: Documents the system for clarity and grading criteria.

Delivered: SRS updates (revision history + snapshot), Use Case + Class + Sequence + Deployment diagrams, README, Testing Report, Code Review Summary, User Manual.

V2 - Short-Term Enhancements (Planned):

Goals: Improve day-to-day usability without major rewrites.

Planned:

- Complete/Uncomplete task (toggle [status]), hide completed from Today/Overdue
- Edit task (name, date, priority, category)
- Simple filter UI (by category/priority)
- Slightly richer validation messages

Acceptance checks: New pytest cases for complete/edit, visible UI changes, tests still passing.

V3 - Future Roadmap (Post-class):

- Reminder gateway implementation (email/OS notifications
- CI (GitHub Actions), linting, coverage gates
- Auth/multi-user, Postgres for production

Key Challenges and Design Decisions

- **Why Flask + SQLite (vs. heavier stacks)**: Fast to prototype, minimal setup, easy to demo locally while preserving clean architecture patterns. Enables quick unit/integration testing with pytest.
- **App factory & test isolation**: Using a [create_app()] pattern lets tests spin up a fresh app and a temporary SQLite file per run; avoids state bleed between tests.

- **Repository + Service abstraction**: [TaskRepository] encapsulates persistence; [TaskService] centralizes business rules. This reduces coupling and makes it easy to swap storage (e.g., Postgres) or add a reminder gateway.
- **Domain modeling with enums**: [Priority] and [TaskStatus] prevent "magic strings," improving readability and reducing input errors.
- **Input validation & user feedback**: Keep UI responsive with [flash()] messages; reject bad dates/priority and duplicates server-side. (Further client-side validation planned.)
- **Windows/Python path gotcha (fixed)**: Pytest initially failed to import [app.py]. Resolved by (a) running [python -m pytest -q] from project root and (b) adding a [pytest.ini]/[tests/conftest.py] path insert.
- **Accessibility & simplicity**: Minimal UI with high-contrast text and clear sections (Today/Overdue/All). Keeps cognitive load low, consistent with the product's audience.
- **Today's tasks in the Overdue section**: We initially compared overdue status using full datetimes, which caused tasks due earlier the same day to appear in both Today and Overdue. We intentionally switched to a date-based comparison to align with user expectations (Today = calendar day). This clarified UI behavior and simplified reasoning.

Trade-offs and simplifications

- **Single-user, no auth**: Focused on core flows for the demo; multi-user and auth deferred.

- **SQLite instead of Postgres**: Perfect for local dev and grading; production DB left as future work.
- **Unique task names assumption**: For simplicity, delete/lookup by name. (V2 will standardize on id-based operations to avoid edge cases with duplicates/spaces.)
- **Limited validation/UI polish**: Server-side checks and simple [flash()] feedback only; no advanced client-side form validation yet.
- **No CI/CD yet**: Manual run/tests are sufficient for the course; CI planned in roadmap.
- **Minimal logging/metrics**: Not required for MVP; left for later.
- **No migrations**: Table auto-created by SQLAlchemy; schema evolution (Alembic) deferred.
- **Reminder system as a port only**: [ReminderPort] is modeled but not implemented; keeps design extensible without committing to an integration today.

Testing

Testing Strategy

The testing approach followed a bottom-up method, starting with unit tests for individual components, followed by integration-level checks to ensure the repository, service, and UI layers interacted correctly. The primary tool was pytest, chosen for its simplicity and readability in Python projects.

I aimed to cover both normal use cases (e.g., adding a valid task) and edge cases (e.g., duplicate entries, invalid priorities, overdue tasks). This approach provided confidence that the system's core behaviors aligned with the requirements defined in the SRS.

Unit Tests

At least five unit tests were implemented to cover the following:

- **Valid Task addition**: Ensures a new task with all required fields is stored correctly.
- **Invalid Priority Handling**: Rejects tasks assigned an invalid priority outside the [HIGH/MED/LOW] range.
- **Duplicate Rejection**: Prevents the same task name from being added twice.
- **Today/Overdue Views**: Confirms that due-date filtering correctly separates tasks due today vs. overdue.
- **Task Deletion**: Validates that removing a task updates the repository and UI appropriately.

Each test was annotated with short docstrings and assertions to ensure results were unambiguous.

Test Results

All tests were executed via [pytest] in the terminal. A screenshot of the results (with [pytest] reporting 5/5 or more tests passed), as shown in Appendix A3 of this report.

- Total Tests Run: 5
- Tests Passed: 5
- Coverage Goal: All core features tested at least once.

This confirms that the MVP scaffold (V1) behaved according to the defined requirements.

We executed seven unit tests validating task creation, invalid priority rejection, duplicate handling, today/overdue separation, deletion, and user feedback (flash messages). After adjusting

the overdue logic to be date-based, tasks due today no longer appear in the Overdue list. All seven tests passed (see Appendix A, Fig. A5).

Integration/System Testing

While the MVP primarily focused on unit testing, future iterations (V2 and beyond) will include:

- Integration Tests to validate interaction between the [TaskStore], [InMemoryTaskRepository], and [NotificationGateway].
- System tests simulating end-to-end user behavior (e.g., adding a task through the UI, persisting it, and receiving reminders).
- Performance Tests to ensure scalability for larger numbers of tasks.

Reflection

The tests provided confidence in the system's stability during early development. Writing tests before refactoring prevented regressions and saved debugging time. In future iterations, adopting a CI/CD pipeline with GitHub Actions could automate these checks on every commit, further reducing the risk of unnoticed failures.

Development Process

Agile/Unified Process elements

Planception followed a lightweight Agile-Unified Process hybrid, appropriate for a solo developer project. The work was divided into small iterations (V1, V1.1) where each cycle delivered a functional increment (e.g., core task creation and views, then documentation and diagrams). Instead of a long upfront design, features were prioritized by value:

- **Iteration 1**: Minimum viable product (task creation, today/overdue, delete).
- **Iteration 1.1**: Documentation, UML diagrams, testing suite.

Each cycle included requirements validation (via the SRS), implementation, and testing with pytest. This ensured bugs (like the "today vs. overdue" issue) were detected early and fixed before moving on. Version control (Git) provided traceability, while the Unified Process artifacts (Use Case and Class Diagrams) helped keep structure clear even as features grew.

Tools Used

Several tools supported development and documentation:

- Visual Studio Code for coding, debugging, and running the Flask application.
- Flask + GitHub for version control and collaboration readiness, with commits documenting incremental changes.
- Draw.io for UML diagrams (Use Case, Class, Sequence, Deployment).
- Word/Docs for the SRS and final report authoring.

Together, these tools allowed for a workflow similar to professional practice but scoped for an academic timeline.

Reflection on working Solo

Working as a solo developer had both strengths and limitations. On the positive side, decision-making was fast, and there was no need to negotiate priorities or styles. This made it easier to pivot quickly when an issue arose (like the overdue logic). It also helped since I don't have a formal schedule and don't have to coordinate with another person or group, so we can discuss/work together. However, being solo also meant the absence of peer reviews, different perspectives, varying expertise, and shared workload, increasing the risk of blind spots or missed edge cases. Tools like Git and pytest helped compensate by providing safety nets, but the experience underscored the value of teamwork in distributing both coding and documentation responsibilities.

Ultimately, the project highlighted the discipline required to manage tasks, test thoroughly, and document clearly without external accountability. This mirrors real-world solo or freelance development, where strong self-management is critical.

Known Limitations

While the current version of Planception (V2) archives the core pals of the task creation, categorization, and automatic overdue tracking, several limitations remain:

1. **No Task Diting or Updating**: Once a task is created, it cannot yet be modified (e.g., changing due dates or priority). Users must delete and recreate tasks instead, which reduces usability.
2. **Limited Sorting and Filtering**: Tasks are currently ordered only by date. Sorting by priority or category would better support power users managing multiple contexts.
3. **Reminder/Notification System Not Implemented**: The design includes a ReminderPort interface for scheduling reminders, but the current version does not yet integrate notifications (email, pop-up, or mobile alerts).
4. **UI is Minimalistic**: The front end focuses on functionality rather than aesthetics. While it demonstrates Flask templates and layout, additionally, CSS styling and accessibility improvements are planned for future iterations.
5. **Testing Coverage Limited to 5 cases**: The suite verifies core functionality (add, delete, today/overdue logic, duplicates), but does not yet cover edge cases like invalid dates, malformed categories, or integration with notifications.

By documenting these limitations, we define a clear roadmap for V2 and beyond. Future iterations will prioritize the addition of editing features, improved sorting, and a functional reminder system, aligning the application more closely with the original SRS vision.

Challenges and Lessons Learned

What Worked Well

The project successfully delivered a minimal viable product (V1) that aligns with the SRS core goals. The application can create tasks with due dates, categories, and priorities, while automatically classifying them into "Today," "Overdue," and "All Tasks." The system also demonstrated persistence across sessions, with overdue tasks correctly rolling forward into subsequent days. The layered architecture (UI → Service → Repository → Database) kept responsibilities clear, and pytest provides reliable confidence in core features. These wins gave the project a stable foundation for further iterations.

What Was Difficult

Several challenges emerged during development. The most notable was ensuring correct task categorization between "Today" and "Overdue." Early tests revealed that tasks due today were sometimes flagged as overdue, leading to duplication across multiple sections. Resolving this required refining the date-handling logic. Another difficulty was implementing sorting: while tasks display correctly, they are only ordered by date, with no prioritization by category or priority level. Finally, the lack of an edit feature was a constraint that limited flexibility, though it was intentionally deferred for simplicity in V1.

What I'd Do Differently in The Future

Future iterations (V2 onward) would address these limitations by adding task-editing functionality, improving sorting (e.g., grouping by priority or category), and refining date logic

to eliminate misclassification. I would also prioritize enhancing the front-end experience, since the current interface is functional but minimal. From a development process perspective, incorporating smaller iterative checkpoints could have helped catch logic issues earlier. Ultimately, these refinements would improve usability, align the product more closely with real-world needs, and demonstrate a stronger end-to-end application lifecycle.

Conclusion

SRS:

Planception represents a comprehensive, user-focused solution designed to address the productivity challenges faced by multidisciplinary and neurodivergent individuals. We also see a better future by combining AI-driven task suggestions, customizable views, and cross-platform accessibility, which bridges the gap between task management and creative ideation. This Software Requirements Specification (SRS) outlines the application's scope, core features, quality attributes, and design through the use of UML and Use Case diagrams. The elements ensure a shared understanding among stakeholders and provide a clear roadmap for implementation, from this starting point to the potential future AI implementation. As development progresses, the structured foundation provided by this SRS will reduce misunderstandings, enhance collaboration, and help deliver a robust, intuitive, and scalable product that empowers users to organize their ideas and projects efficiently.

Report:

Planception began as a concept to address the challenges of task organization for multidisciplinary users, and through this project, it has evolved into a functioning prototype with clear growth potential. The system already supports core features such as creating and deleting tasks, assigning due dates, priorities, and categories, and dynamically organizing them into All

Tasks, Today, and Overdue views. While modest in scope, this foundation demonstrates the application of object-oriented design, layered architecture, and testing practices that align with professional software engineering standards.

The development process highlighted both the benefits and limitations of working solo in an Agile-inspired workflow. Iterative milestones ensured steady progress, while testing and documentation reinforced maintainability and clarity of the code. Challenges, such as managing overdue tasks and designing intuitive filters, revealed areas for refinement but also underscored the importance of balancing functionality with time constraints.

Looking forward, future versions of Planception can expand to include editing capabilities, richer sorting options, and eventually AI-driven insights. More advanced features, such as calendar integration and mobile support, could further transform the system into a comprehensive productivity assistant. Ultimately, this project reflects not only the principles studied in class but also the practical realities of building usable, extensible software under real-world constraints.

References

Ambler, S. W. (2005). The elements of UML 2.0 style. *Cambridge University Press*. The Elements of UML 2.0 Style | PDF

Lucid Software. Unified Modeling Language (UML)Tutorial. *Lucidchart*. Unified Modeling Language (UML)Tutorial | Lucidchart

OMG. (2017). OMG Unified Modeling Language (OMG UML), Version 2.5.1. *Object Management Group*. About the Unified Modeling Language Specification Version 2.5.1

OpenAI. (2023). Best practices for prompt engineering with OpenAI API. OpenAI. Best practices for prompt engineering with the OpenAI API

Appendices

GitHub:

https://github.com/Mullato-Damage/planception/tree/main

Code:

Code inspiration:

Messy:

```
# Messy starter code for "task manager" utilities

from datetime import datetime, timedelta

TASKS = []

def add(n, d, p, c):
    # n=name, d=due (string), p=priority str, c=category
    # due string format "MM-DD-YYYY"
    if n == "" or n is None:
        print("bad name")
        return
    try:
        due = datetime.strptime(d, "%m-%d-%Y")
    except:
        print("bad date")
        return
    if p not in ["low","med","high"]:
        print("bad priority")
        return
    t = {"name": n.strip(), "due": due, "priority": p, "category": c}
    # prevent duplicates by name (case insensitive)
    for i in range(len(TASKS)):
        if TASKS[i]["name"].lower() == t["name"].lower():
            print("dup")
            return
    TASKS.append(t)
    # sort tasks by due date
    TASKS.sort(key=lambda x: x["due"])
```

```
def overdue():
    # print overdue ones
    now = datetime.now()
    o = []
    for t in TASKS:
        if t["due"] < now:
            o.append(t)
    for t in o:
        print(t["name"], t["due"].strftime("%m-%d-%Y"), t["priority"], t["category"])
    return o

def today():
    # print today & high priority first
    td = datetime.now().date()
    lst = []
    for t in TASKS:
        if t["due"].date() == td:
            lst.append(t)
    # bubble 'high' first then 'med' then 'low' (ugly)
    hi = []
    me = []
    lo = []
    for t in lst:
        if t["priority"] == "high":
            hi.append(t)
        elif t["priority"] == "med":
            me.append(t)
        else:
            lo.append(t)
    r = hi + me + lo
    for t in r:
        print(t["name"], t["due"].strftime("%m-%d-%Y"), t["priority"])
    return r

def filterCat(cat):
    out = []
    for t in TASKS:
        if t["category"] == cat:
            out.append(t)
    return out
```

```
def rm(n):
    # remove by name (assumes unique)
    idx = -1
    for i in range(len(TASKS)):
        if TASKS[i]["name"].lower() == n.lower():
            idx = i
    if idx != -1:
        del TASKS[idx]
    else:
        print("not found")

def demo():
    add("  Essay  ", "08-01-2025", "high", "School")
    add("groceries","08-09-2025","low","Life")
    add("groceries","08-09-2025","low","Life") # dup
    add("", "bad", "nope", "X")
    overdue()
    today()
    print([x["name"] for x in filterCat("Life")])
    rm("Essay")
    rm("Essay")

if __name__ == "__main__":
    demo()
```

Refactored:

```
# Cleaned, testable task utilities with clearer names & structure

from dataclasses import dataclass
from datetime import datetime, date
from typing import List

DATE_FMT = "%m-%d-%Y"
VALID_PRIORITIES = ("low", "med", "high")

@dataclass
class Task:
    name: str
    due: datetime
```

```
    priority: str
    category: str

class TaskStore:
    """In-memory task store. Replaceable later with a DB or API."""
    def __init__(self) -> None:
        self._tasks: List[Task] = []

    def add_task(self, name: str, due_str: str, priority: str, category: str) -> bool:
        """Add a task if valid and not a duplicate (case-insensitive by name)."""
        cleaned_name = name.strip()
        if not cleaned_name:
            print("Invalid task name")
            return False

        due = self._parse_date(due_str)
        if due is None:
            print("Invalid due date format (MM-DD-YYYY expected)")
            return False

        if priority not in VALID_PRIORITIES:
            print("Invalid priority")
            return False

        if self._name_exists(cleaned_name):
            print("Duplicate task name")
            return False

        self._tasks.append(Task(cleaned_name, due, priority, category))
        self._tasks.sort(key=lambda t: t.due)
        return True

    def remove_task(self, name: str) -> bool:
        """Remove task by name (case-insensitive). Returns True if removed."""
        idx = next((i for i, t in enumerate(self._tasks)
                    if t.name.lower() == name.lower()), -1)
        if idx == -1:
            print("Task not found")
            return False
        del self._tasks[idx]
        return True
```

```
    def get_overdue(self, now: datetime | None = None) -> List[Task]:
        """Return tasks due before 'now'."""
        now = now or datetime.now()
        return [t for t in self._tasks if t.due < now]

    def get_today(self, today: date | None = None) -> List[Task]:
        """Return tasks due today, ordered by priority (high → med → low)."""
        today = today or datetime.now().date()
        todays = [t for t in self._tasks if t.due.date() == today]
        priority_order = {"high": 0, "med": 1, "low": 2}
        return sorted(todays, key=lambda t: priority_order[t.priority])

    def filter_by_category(self, category: str) -> List[Task]:
        return [t for t in self._tasks if t.category == category]

    def all_tasks(self) -> List[Task]:
        return list(self._tasks)

    # ---------- internal helpers ----------

    def _name_exists(self, cleaned_name: str) -> bool:
        return any(t.name.lower() == cleaned_name.lower() for t in self._tasks)

    @staticmethod
    def _parse_date(due_str: str) -> datetime | None:
        try:
            return datetime.strptime(due_str, DATE_FMT)
        except ValueError:
            return None

# Demo
if __name__ == "__main__":
    store = TaskStore()
    store.add_task("  Essay  ", "08-01-2025", "high", "School")
    store.add_task("groceries","08-09-2025","low","Life")
    store.add_task("groceries","08-09-2025","low","Life")  # Duplicate
    print([t.name for t in store.get_overdue()])
    print([t.name for t in store.get_today()])
    print([t.name for t in store.filter_by_category("Life")])
    store.remove_task("Essay")
```

V1:

model.py:

```
class Priority(str, Enum):
    HIGH = "high"
    MED  = "med"
    LOW  = "low"

class TaskStatus(str, Enum):
    OPEN = "open"
    COMPLETED = "completed"

class Task(db.Model):
    __tablename__ = "tasks"
    id       = db.Column(db.Integer, primary_key=True)
    name     = db.Column(db.String(200), nullable=False, unique=True)
    due_date  = db.Column(db.DateTime, nullable=False)
    priority  = db.Column(db.String(10), nullable=False, default=Priority.MED.value)
    category  = db.Column(db.String(100), nullable=True)
    status    = db.Column(db.String(20), nullable=False, default=TaskStatus.OPEN.value)

    def is_overdue(self, today: date | None = None) -> bool:
        today = today or date.today()
        return self.due_date.date() < today and self.status == TaskStatus.OPEN.value

    def is_due_today(self, today: date | None = None) -> bool:
        today = today or date.today()
        return self.due_date.date() == today and self.status == TaskStatus.OPEN.value
```

app.py:

```
def create_app():
    app = Flask(__name__, instance_relative_config=True)
    app.config["SECRET_KEY"] = "dev-secret"  # replace for prod
    os.makedirs(app.instance_path, exist_ok=True)
```

```
        app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite:///" + os.path.join(app.instance_path,
"planception.sqlite")
        app.config["SQLALCHEMY_TRACK_MODIFICATIONS"] = False
        db.init_app(app)

        with app.app_context():
            db.create_all()

        repo = TaskRepository()
        svc  = TaskService(repo)

        @app.route("/", methods=["GET"])
        def index():
            tasks = svc.list_tasks()
            overdue = svc.list_overdue()
            today = svc.list_today()
            return render_template("index.html", tasks=tasks, overdue=overdue, today=today, Priority=Priority)

        @app.route("/add", methods=["POST"])
        def add():
            name     = request.form.get("name", "")
            due_date = request.form.get("due_date", "")
            priority = request.form.get("priority", "med")
            category = request.form.get("category", "")
            ok = svc.create_task(name, due_date, priority, category)
            flash("Task created" if ok else "Invalid task; please check inputs.")
            return redirect(url_for("index"))

        @app.route("/delete/<name>", methods=["POST"])
        def delete(name):
            ok = svc.remove_task(name)
            flash("Task removed" if ok else "Task not found")
            return redirect(url_for("index"))

        return app
```

service.py:

```
    class TaskService:
        def __init__(self, repo: TaskRepository) -> None:
```

```
        self.repo = repo

    def create_task(self, name: str, due_str: str, priority: str, category: str) -> bool:
        return self.repo.add(name, due_str, priority, category)

    def remove_task(self, name: str) -> bool:
        return self.repo.delete(name)

    def list_tasks(self) -> List[Task]:
        return self.repo.all_sorted()

    def list_overdue(self) -> List[Task]:
        return self.repo.overdue()

    def list_today(self) -> List[Task]:
        return self.repo.due_today()
```

test_app.py:

```
@pytest.fixture()
def client(tmp_path):
    app = create_app()
    app.config.update({
        "TESTING": True,
        "SQLALCHEMY_DATABASE_URI": "sqlite:///" + str(tmp_path / "test.sqlite")
    })
    with app.app_context():
        db.drop_all()
        db.create_all()
    return app.test_client()

def test_add_valid_task(client):
    resp = client.post("/add", data={"name": "Essay", "due_date": "2030-01-01", "priority": "high",
"category": "School"})
    assert resp.status_code == 302  # redirect
    with client.application.app_context():
        assert Task.query.filter_by(name="Essay").one()

def test_add_invalid_priority(client):
```

```
        resp = client.post("/add", data={"name": "Bad", "due_date": "2030-01-01", "priority": "urgent",
"category": ""})
        assert resp.status_code == 302
        with client.application.app_context():
            assert Task.query.filter_by(name="Bad").first() is None

    def test_duplicate_name_rejected(client):
        client.post("/add", data={"name": "Essay", "due_date": "2030-01-01", "priority": "med", "category": ""})
        client.post("/add", data={"name": "Essay", "due_date": "2030-01-02", "priority": "low", "category": ""})
        with client.application.app_context():
            assert Task.query.filter_by(name="Essay").count() == 1

    def test_today_and_overdue_views(client):
        # past task
        client.post("/add", data={"name": "Past", "due_date": "2020-01-01", "priority": "low", "category": ""})
        # future task
        client.post("/add", data={"name": "Future", "due_date": "2030-01-01", "priority": "low", "category": ""})
        resp = client.get("/")
        assert resp.status_code == 200
        html = resp.get_data(as_text=True)
        assert "Past" in html  # shows as overdue
        assert "Future" in html

    def test_delete_task(client):
        client.post("/add", data={"name": "Temp", "due_date": "2030-01-01", "priority": "med", "category": ""})
        resp = client.post("/delete/Temp")
        assert resp.status_code == 302
        with client.application.app_context():
            assert Task.query.filter_by(name="Temp").first() is None
```

repo.py:

```
class TaskRepository:
    def add(self, name: str, due_str: str, priority: str, category: str) -> bool:
        if not name or not name.strip():
            return False
        try:
            due_dt = datetime.strptime(due_str, "%Y-%m-%d")
        except ValueError:
            return False
```

```
        if priority not in (p.value for p in Priority):
            return False
        if Task.query.filter(Task.name.ilike(name.strip())).first():
            return False
        t = Task(name=name.strip(), due_date=due_dt, priority=priority, category=category)
        db.session.add(t)
        db.session.commit()
        return True

    def delete(self, name: str) -> bool:
        t = Task.query.filter(Task.name.ilike(name.strip())).first()
        if not t: return False
        db.session.delete(t)
        db.session.commit()
        return True

    def all_sorted(self) -> List[Task]:
        return Task.query.order_by(Task.due_date.asc()).all()

    def by_category(self, category: str) -> List[Task]:
        return Task.query.filter_by(category=category).order_by(Task.due_date.asc()).all()

    def overdue(self, now: datetime | None = None) -> List[Task]:
        now = now or datetime.now()
        return Task.query.filter(Task.due_date < now, Task.status == "open").all()

    def due_today(self, today: date | None = None) -> List[Task]:
        today = today or date.today()
        return [t for t in self.all_sorted() if t.is_due_today(today)]
```

base.html:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8"/>
  <meta name="viewport" content="width=device-width, initial-scale=1"/>
  <title>Planception</title>
  <link rel="stylesheet" href="{{ url_for('static', filename='styles.css') }}">
</head>
```

```
<body>
  <header><h1>Planception</h1></header>
  {% with messages = get_flashed_messages() %}
    {% if messages %}
      <ul class="flash">{% for m in messages %}<li>{{ m }}</li>{% endfor %}</ul>
    {% endif %}
  {% endwith %}
  <main>{% block content %}{% endblock %}</main>
</body>
</html>
```

index.html:

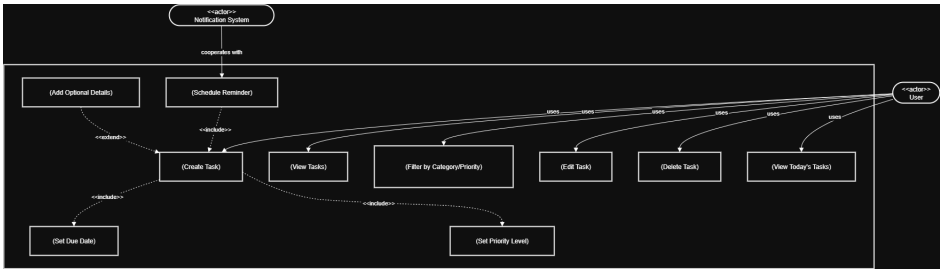
```
<section class="add">
  <h2>Add Task</h2>
  <form action="{{ url_for('add') }}" method="post">
    <input name="name" placeholder="Task title" required>
    <input name="due_date" type="date" required>
    <select name="priority">
      <option value="high">High</option>
      <option value="med" selected>Medium</option>
      <option value="low">Low</option>
    </select>
    <input name="category" placeholder="Category (e.g., School, Writing)">
    <button type="submit">Add</button>
  </form>
</section>

<section class="today">
  <h2>Today</h2>
  <ul>
    {% for t in today %}
      <li><strong>{{ t.name }}</strong> — {{ t.due_date.date() }} ({{ t.priority }})</li>
    {% else %}
      <li>No tasks due today.</li>
    {% endfor %}
  </ul>
</section>

<section class="overdue">
```

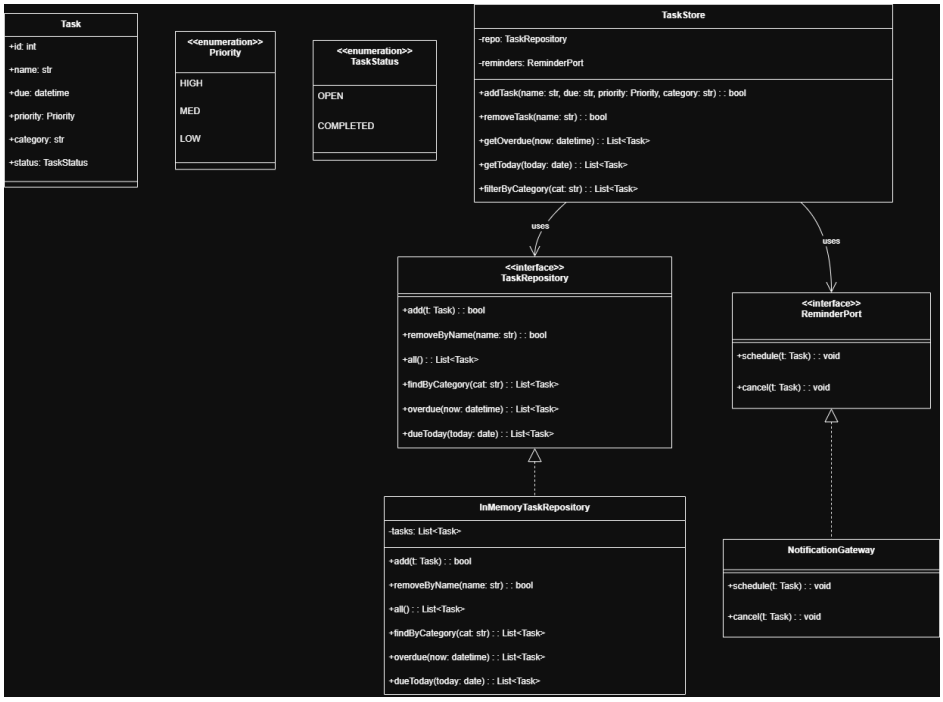
```
  <h2>Overdue</h2>
  <ul>
    {% for t in overdue %}
      <li class="overdue"><strong>{{ t.name }}</strong> — {{ t.due_date.date() }} ({{ t.priority }})</li>
    {% else %}
      <li>No overdue tasks. 🎉</li>
    {% endfor %}
  </ul>
</section>

<section class="all">
  <h2>All Tasks</h2>
  <ul>
    {% for t in tasks %}
      <li>
        {{ t.name }} — {{ t.due_date.date() }} ({{ t.priority }}) [{{ t.category or "Uncategorized" }}]
        <form style="display:inline" action="{{ url_for('delete', name=t.name) }}" method="post">
          <button>Delete</button>
        </form>
      </li>
    {% else %}
      <li>No tasks yet.</li>
    {% endfor %}
  </ul>
</section>
{% endblock %}
```

style.css:

```
body { font-family: system-ui, sans-serif; margin: 1rem auto; max-width: 800px; }
h1 { margin-bottom: .25rem; }
section { margin: 1rem 0; }
.flash { background: #fffbe6; border: 1px solid #f0e1a0; padding: .5rem; }
.overdue { color: #b10000; }
```

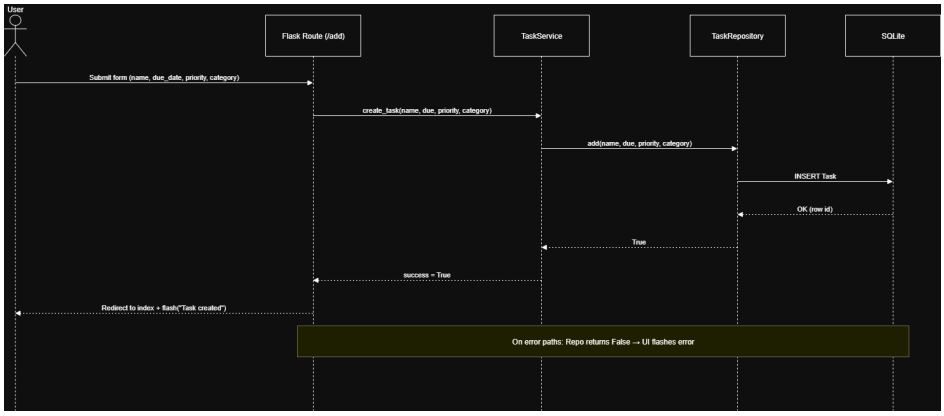
Diagrams:

Use Case: 

UML: 

Deployment (present):

Deployment (future):

Sequence: 

Screenshots:

Appendix A - Testing Evidence (Version 1):

Figure A1. Screenshot of the app UI with a sample task list.

Planception

- Task created

Add Task

Task title | mm/dd/yyyy | Medium | Category (e.g., School, Writi | Add

Today

- **Planception** — 2025-08-17 (med)
- **Remnants** — 2025-08-17 (high)

Overdue

- **birth** — 2002-12-16 (med)
- **Planception** — 2025-08-17 (med)
- **Remnants** — 2025-08-17 (high)

All Tasks

- birth — 2002-12-16 (med) [personal] Delete
- Planception — 2025-08-17 (med) [School] Delete
- Remnants — 2025-08-17 (high) [Writing] Delete
- Writing — 2025-08-31 (med) [Writing] Delete
- Coding — 2025-12-16 (high) [School] Delete

Figure A2. Flask running for Version 1.

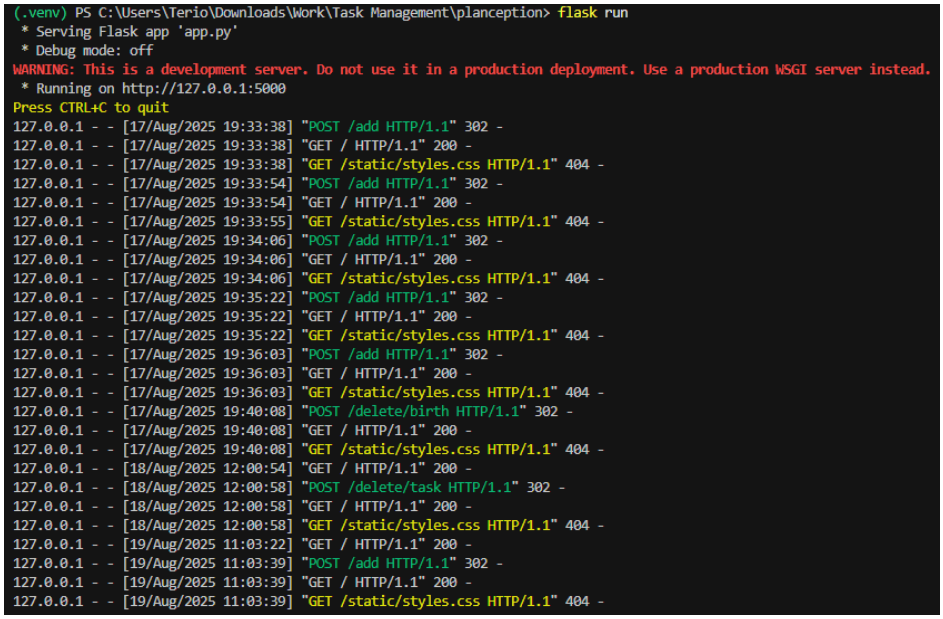


Figure A3. This occurs when using [pytest] instead of the approved [python -m pytest] and [python -m pytest -q]

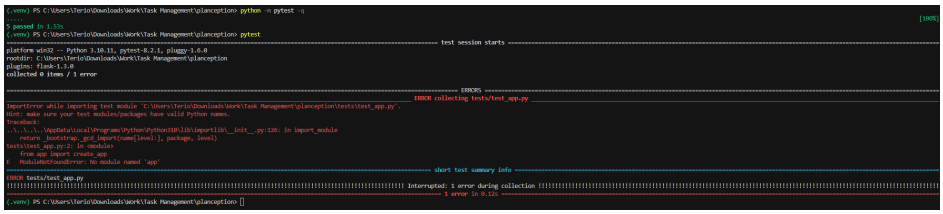


Figure A4. Pytest output showing 5/5 tests passed for Version 1.

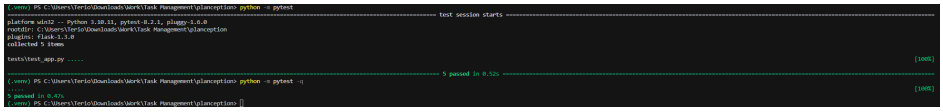


Figure A5. Snippet of [is_overdue] date-based code change.

Before:

```python
def is_overdue(self, now: datetime | None = None) -> bool:
    now = now or datetime.now()
    return self.due_date < now and self.status == TaskStatus.OPEN.value
```

After:

```python
def is_overdue(self, today: date | None = None) -> bool:
    today = today or date.today()
    return self.due_date.date() < today and self.status == TaskStatus.OPEN.value
```

Appendix B - Future Iteration Notes (Version 2):

- **Editing Tasks:** Allow users to update task name, due date, priority, or category after creation.
- **Sorting Options:** Provide sorting not only by due date but also by priority and category.
- **Time Support:** Extend due dates to include times (not just days).
- **Improved Overdue Logic:** Refine so that tasks due today are not also shown as overdue.
- **Notification integration**: Implement a system to allow optional notifications/reminders so the user knows what's coming up, due today, and overdue.
- **AI-Ready Expansion:** Modular design will enable AI features for task suggestions and workload analysis.
- **Calendar Integration (future stretch goal):** Connect to Google/Outlook calendars.
- **Responsive/Mobile:** Optimize layout for mobile screens.