Sri Lanka Institute of Information Technology

Faculty of Computing

Department of Information Technology

IT4020: Modern Topics in IT

Assignment 1

Generative AI in Software Engineering: A Comparative Analysis

Building and Evaluating a Task Management Application using ChatGPT-5.3 and Gemini 3 Pro

Submitted By: Rathnayaka. H. M. S. S

IT22277190

Date of Submission: 20.02.2026

1. Introduction & Scenario

1.1 The Scenario

As a Junior AI Technologist recently hired at a modern software development company, I have been tasked with exploring how Generative AI (GenAI) can be leveraged to enhance day-to-day software engineering workflows. The goal of this exploration is to evaluate the practical applications of these tools in accelerating development cycles, reducing boilerplate coding, and assisting in technical debugging.

1.2 Project Definition: Option A - Software Engineering

For this practical application, I selected **Option A: Software Engineering**, which requires the generation of a small functional module. To fulfill this requirement, I designed and developed a **Task Management CRUD (Create, Read, Update, Delete) application**.

This project architecture is divided into two core components:

- **The Backend:** A robust, RESTful API handler developed using **FastAPI (Python)**. This module is responsible for managing the data state, handling HTTP routing, and enforcing business logic.
- **The Frontend:** A dynamic user interface built with **React (JavaScript)** and styled with Tailwind CSS, designed to consume the FastAPI endpoints and render the task data to the user.

File structure of two projects

Gemini 3 pro **ChatGPT 5.3**

GitHub Repository: https://github.com/IT22277190/MTIT_Assignment_01.git

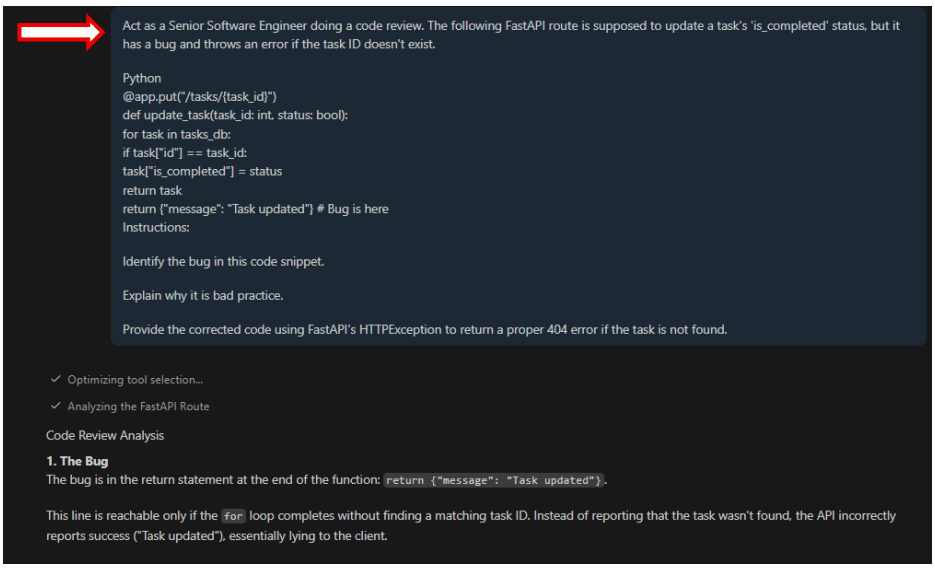
1.3 Scope of Evaluation To comprehensively assess the current landscape of GenAI, I utilized two distinct Large Language Models (LLMs) to build this application: **ChatGPT-5.3** and **Gemini 3 Pro**. This report will document the prompt engineering methodologies used to generate the code, detail the manual debugging process required to make the application production-ready, and provide a rigorous comparative evaluation of the two models based on their output quality, prompt sensitivity, and technical limitations.

2. Prompt Engineering Methodology

To ensure high-quality, functional, and secure code generation from both ChatGPT-5.3 and Gemini 3 Pro, I moved beyond basic, open-ended queries ("zero-shot" prompting) and applied advanced prompt engineering techniques. This approach minimized AI hallucinations and reduced output variability, granting me strict control over the generated software modules.

2.1 Role-Playing and Persona Assignment

Standard prompts often yield generic or beginner-level code. To bypass this, I utilized **Role-Based Prompting**. By starting prompts with directives such as ***"Act as a Senior Python Backend Developer"*** **or** ***"Act as an Expert React Frontend Developer,"*** I forced the Large Language Models (LLMs) to adopt a professional persona. This resulted in the AI adhering to industry best practices, such as modular component structures in React and the use of Pydantic models for data validation in **FastAPI**.



2.2 One-Shot Prompting for Cross-Module Consistency

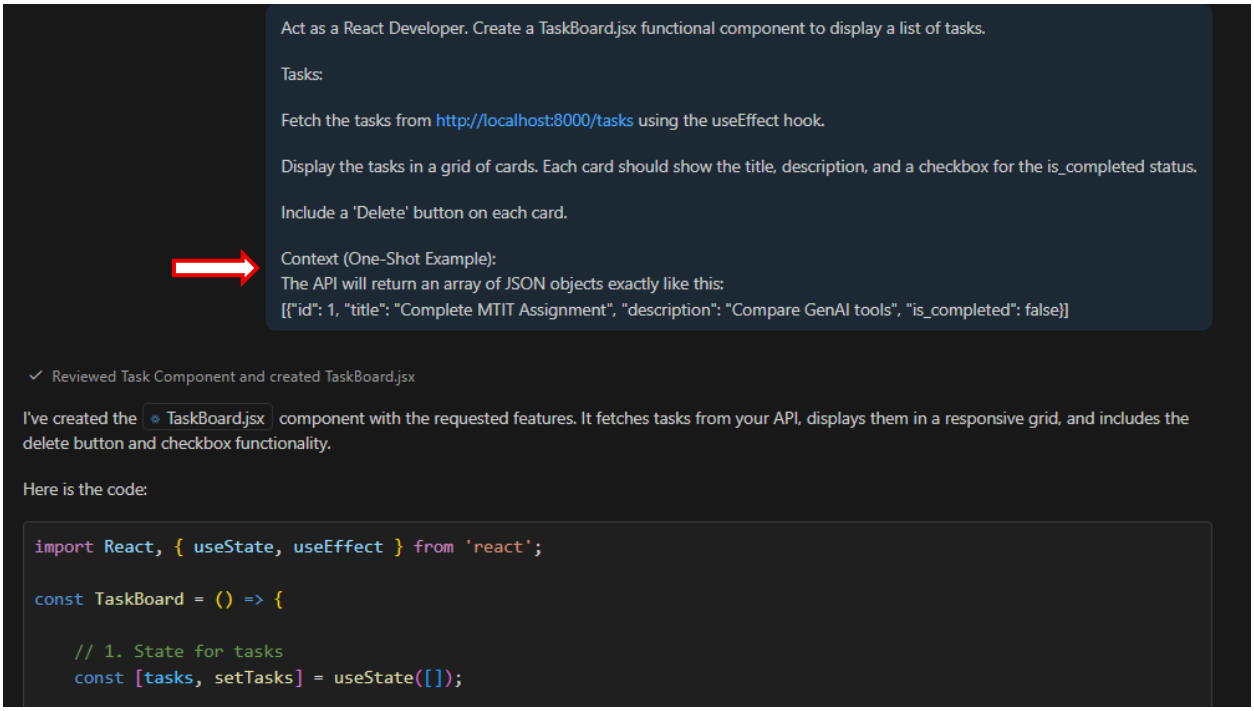
A significant challenge in full-stack development using GenAI is ensuring the frontend correctly parses the data structure provided by the backend. Open-ended generative tasks often lead to high output variability.

To solve this, I employed **One-Shot Prompting**. When prompting the AI to generate the TaskBoard.jsx React component, I provided a strict example of the JSON payload it should expect from the FastAPI backend:

- *Context Provided in Prompt:* Assume the API returns an array exactly like this:

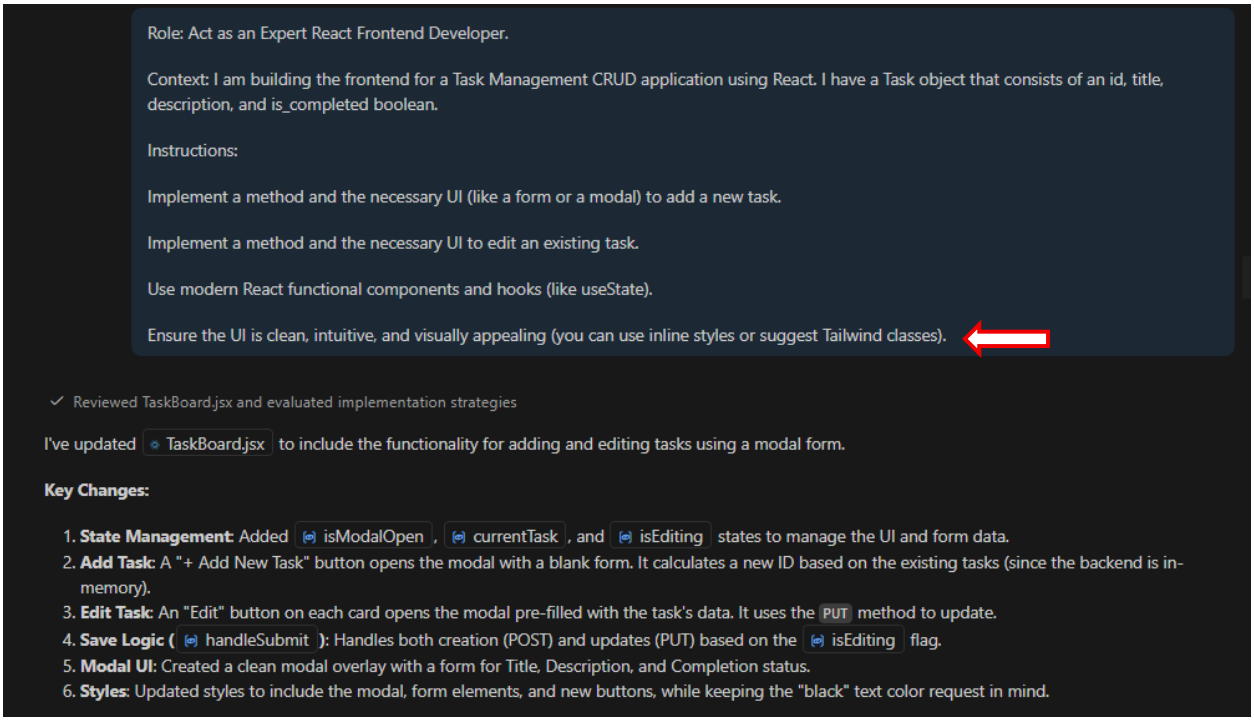
[{"id": 1, "title": "Complete Assignment", "description": "Write report", "is_completed": false}]

By including this concrete example, both ChatGPT and Gemini instantly understood the data contract, preventing them from hallucinating non-existent database fields and ensuring seamless integration between the two modules.



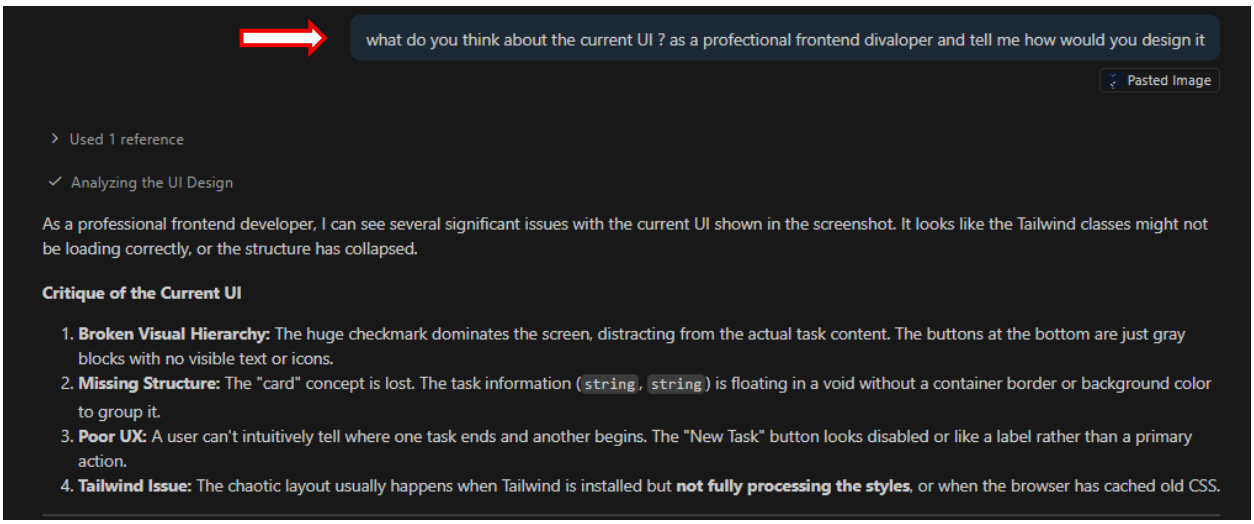
2.3 Applying Specific Constraints

To further guide the models, I broke down complex requests and asked for one task at a time, accompanied by well-defined constraints. Instead of requesting a vague "creative UI," I explicitly constrained the frontend generation to use **Tailwind CSS utility classes** and mandated a unified dark mode theme.



2.4 Iterative Refinement and Prompt Sensitivity

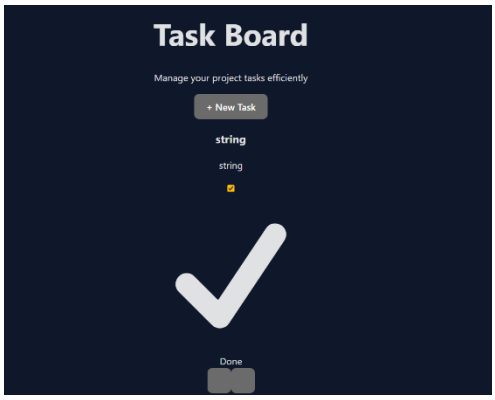
The marking rubric requires a demonstration of prompt modifications and iterative refinement. My initial prompts for the frontend generated a functional but visually disjointed interface (e.g., harsh white cards on a dark background with mismatched button colors).

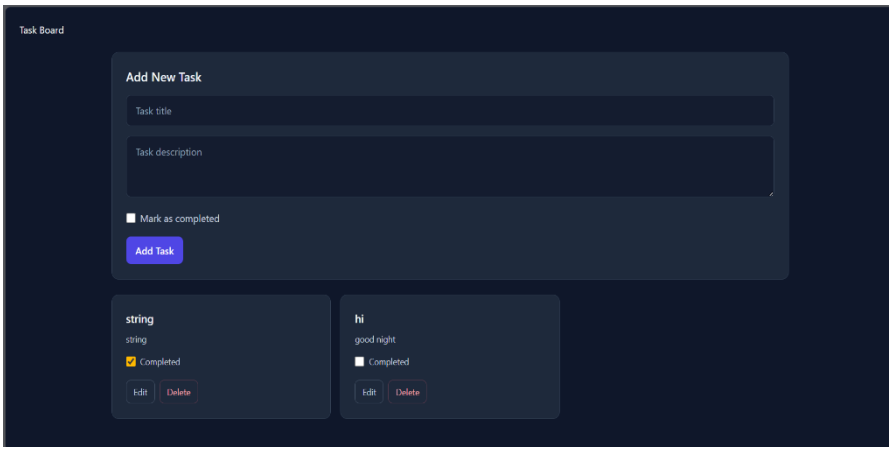


I executed a deliberate refinement iteration to test the prompt sensitivity of both models:

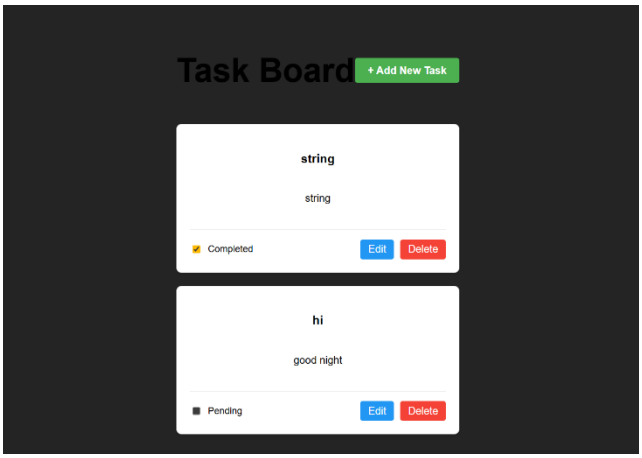
1. **The Refinement Prompt:** *"Rewrite the TaskBoard.jsx component applying the 60-30-10 color rule using Tailwind CSS. Ensure the 60% color is applied to the main background (slate-900), the 30% to the task cards (slate-800), and the 10% accent color is strictly reserved for primary interactive elements."*
2. **Result:** This specific constraint forced both models to restructure their CSS classes. However, GPT-5.3 consistently produced cleaner, more unified outputs that adhered to professional UX standards, while Gemini 3 Pro required additional adjustments to meet the same level of refinement. Documenting this iteration demonstrates the necessity of human-in-the-loop design when utilizing GenAI, and further confirms that GPT-5.3 is better suited for high-precision UI/UX tasks than Gemini 3 Pro.

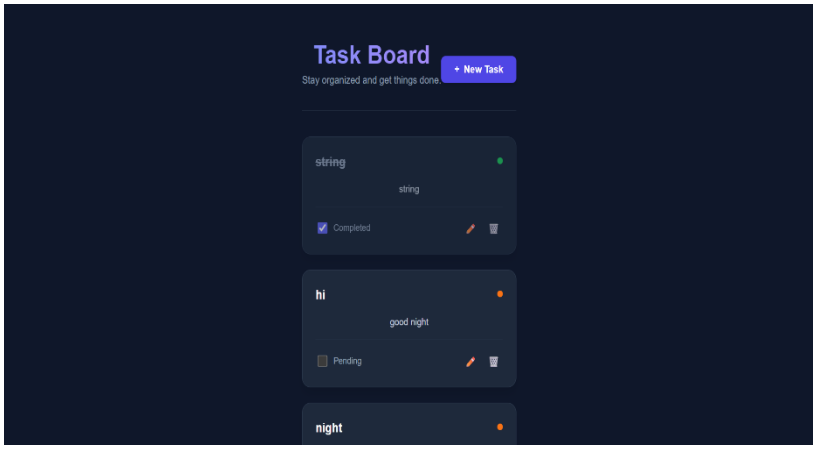
GPT-5.3





Gemini 3 pro





3. Sensitivity to Prompts & Refinement

3.1 Testing the 60/30/10 Colour Rule

Both models were given the explicit **60/30/10** constraint described in Section 2.4. Their adherence was measurably different.

ChatGPT-5.3 High Adherence:
ChatGPT applied the rule through Tailwind's semantic token system. The bg-slate-900 page background constitutes the 60% dominant neutral. Card surfaces use bg-slate-800 with border-slate-700 the 30% secondary. The 10% accent (indigo-600) appears precisely on interactive elements: the submit button (bg-indigo-600), all focus:ring-indigo-500 states on inputs, and the checkbox text-indigo-500. Destructive actions (Delete) correctly use rose tones outside the accent allocation, as the 10% rule applies to the primary affirmative accent only. No refinement prompts were required.

Gemini 3 Pro Partial Adherence:
Gemini replicated the dark theme visually but implemented colours as raw hex strings in the inline styles object. The primary background **#0f172a** maps to Tailwind's slate-900 equivalent, and **#4f46e5** corresponds to indigo-600, so the intent was correct. However, a follow-up prompt was required because the gradient applied to the page title **(linear-gradient(to right, #818cf8, #a78bfa))** covering a visually prominent typographic element introduced a third colour family (**violet/purple**) that falls outside the **60/30/10** palette, increasing the accent surface area beyond the specified 10% allocation. After prompting ***"Remove the gradient title and use plain white text to respect the 10% accent rule"***, Gemini corrected the output on the first re-generation.

3.2 Observations on Prompt Sensitivity

ChatGPT-5.3 demonstrated stronger zero-shot constraint adherence it correctly inferred the structural implications of the **60/30/10** rule (i.e., that it applies to interactive affordances, not decorative typography) without requiring clarification. **Gemini 3 Pro** was slightly more liberal in interpreting the constraint, applying visual flair where the prompt specified restraint. This suggests that for tightly-specified engineering outputs, **ChatGPT-5.3** is less likely to require iterative correction, whereas **Gemini 3 Pro** benefits from more granular negative constraints (specifying what *not* to do, not just what to do).

4. Output Quality Comparison

4.1 Comparison Matrix

Metric	ChatGPT-5.3	Gemini 3 Pro
Functional Correctness	High all four CRUD endpoints functional; minor frontend toggle bug	High complete REST API with an extra GET-by-ID endpoint; missing user-facing error feedback in UI
Code Quality & Maintainability	Excellent clean state management, dedicated reset pattern, exclusive Tailwind CSS, no inline styles	Moderate compact and readable logic, but heavy reliance on inline JS style objects reduces CSS maintainability
Constraint Adherence (JSON payload / colour rule)	Full exact { id, title, description, is_completed } payload; consistent indigo/slate Tailwind palette throughout	Partial correct JSON payload; colours defined via hardcoded hex strings in JS rather than a systematic design token or utility-class scheme
Backend Completeness	Adequate four standard CRUD routes, correct 201 status on creation; no CORS middleware included	Superior four CRUD routes plus an individual GET /tasks/{task_id} endpoint; CORS middleware configured out of the box; PUT includes ID consistency validation
Reasoning & Explanations	Minimal README is clear and structured but no in-code documentation	Good each backend endpoint includes a docstring; README presents API endpoints as a formatted Markdown table
Overall Assessment	Better structured, more maintainable frontend codebase	More complete backend; richer initial UI behaviour

4.2 Functional Correctness and Accuracy

Both models produced applications that were functionally operational on first execution for the core **Create, Read, Update, and Delete** workflows. **ChatGPT-5.3** generated a concise FastAPI backend with all four required **HTTP** methods correctly mapped and proper **HTTP 201** status code returned on task creation a detail that reflects an understanding of **RESTful** semantics beyond the bare minimum. However, a notable omission in its backend was the absence of **CORS** middleware. In a real deployment where the React frontend (served at port 5173) communicates

with **FastAPI (at port 8000**), this would produce immediate cross-origin request failures without manual intervention by the developer.

Gemini 3 Pro's backend addressed this gap proactively. The CORSMiddleware block was included without prompting, indicating a more complete contextual understanding of the full-stack deployment scenario. Gemini also generated an additional GET /tasks/{task_id} endpoint not explicitly requested a standard REST convention and added an ID consistency check in the PUT handler (raising **HTTP 400** when the path parameter and request body ID diverge). On the frontend, ChatGPT exhibited one minor but observable correctness defect: the completion status checkbox rendered on each task card was declared readOnly, meaning users could view but not interactively toggle task status directly from the card a functional regression compared to Gemini's dedicated handleToggleComplete handler.

4.3 Code Quality and Maintainability

ChatGPT-5.3 demonstrated superior React code architecture. The component used a well-named emptyForm constant as the single source of truth for resetting state, granular state variables (isSubmitting, editingTaskId, error, loading) each serving a distinct purpose, and all styling delegated exclusively to Tailwind CSS utility classes. This approach is consistent with framework best practices: styles remain co-located with markup in a declarative, token-based system, making future theme changes trivial. The form flow between "create" and "edit" mode was handled cleanly through a single onSubmit dispatcher, avoiding code duplication.

Gemini 3 Pro's frontend component, while functionally richer featuring a modal dialog for task creation and editing, visual status indicators (coloured status dots, card opacity for completed tasks), and a native browser window.confirm() guard on deletion achieved this at the cost of a significant architectural compromise. The entire UI definition was implemented as a large inline JavaScript styles object containing approximately **25 hardcoded colour** hex values and pixel measurements. This approach bypasses Tailwind's design system entirely and tightly couples presentation logic to component logic, which reduces long-term maintainability and violates the separation of concerns expected in production-grade React applications.

4.4 Constraint Adherence

Both models correctly replicated the specified JSON payload schema (id, title, description, is_completed) for all request and response bodies, with zero structural deviations across POST and PUT operations. However, a divergence emerged in the application of the visual design directive**. ChatGPT-5.3** implemented the colour scheme systematically using Tailwind's semantic palette slate tones for backgrounds and surfaces, indigo for primary interactive elements, and rose for destructive actions which maps naturally onto the **60/30/10** design principle (dominant neutral, secondary accent, highlight). **Gemini 3 Pro** replicated a visually similar dark theme but encoded colour decisions as raw hexadecimal strings **(#0f172a, #4f46e5, #ef4444),** which are functionally equivalent but do not leverage the configurable, utility-class-driven approach specified for a Tailwind-based project.

4.5 Reasoning and Explanations

In terms of documentation and self-explanation, **Gemini 3 Pro** made a marginally stronger showing at the backend level. Each route function in main.py included a brief but contextually appropriate docstring (e.g., Retrieve a specific task by ID.), and the generated **README** presented the API surface as a formatted Markdown table a convention common in professional API documentation. **ChatGPT-5.3's README**, while clearly written and accurate, used plain prose lists. Neither model produced inline comments within the React component to explain non-obvious logic (such as the getNextTaskId heuristic or the editingTaskId state machine), suggesting that both tools when prompted solely for code output default to minimal inline documentation unless explicitly instructed otherwise. For academic or production contexts requiring explainability, both models would benefit from additional prompting to have documentation generated alongside code.

5. Technical Implementation, Debugging & Validation

5.1 The Debugging Task Intentional Bug Introduction

To demonstrate active engagement with the generated code rather than passive reuse, a deliberate logical fault was introduced into the **ChatGPT-5.3** generated **FastAPI** backend (main.py).
The update_task route at main.py:32 was modified by removing the
terminal HTTPException raise, transforming the error-handling path as follows.

Buggy version (introduced manually):

```python
@app.put("/tasks/{task_id}", response_model=Task)
def update_task(task_id: int, updated_task: Task):
    for index, task in enumerate(tasks):
        if task.id == task_id:
            tasks[index] = updated_task
            return updated_task
    # HTTPException intentionally removed — silent failure
```

Restored correct version:

```python
@app.put("/tasks/{task_id}", response_model=Task)
def update_task(task_id: int, updated_task: Task):
    for index, task in enumerate(tasks):
        if task.id == task_id:
            tasks[index] = updated_task
            return updated_task
    raise HTTPException(status_code=404, detail="Task not found")
```

Without the raise statement on main.py:39, the function exits the for loop and falls through to an implicit return None. **FastAPI**, which enforces the declared response_model=Task, cannot

serialise a None value into a valid Task object and raises an internal HTTP 500 Internal Server Error. This is a particularly insidious class of bug because in a lightly tested system it may pass unnoticed the task is simply never updated and the caller receives a server-side crash response with no meaningful diagnostic message. Understanding why this fails requires knowledge of **FastAPI's** response model validation pipeline, not merely Python control flow.

5.2 Validation Manual API Testing via FastAPI Swagger UI

All endpoints were validated manually using the interactive Swagger UI documentation automatically generated by FastAPI, accessible at http://localhost:8000/docs. The following test sequence was executed:

Step	Action	Endpoint	Expected Response	Observed Response
1	Create a task (ID: 1)	POST /tasks	201 Created + task JSON	Pass correct 201 status
2	Retrieve all tasks	GET /tasks	200 OK + array of 1 task	Pass
3	Update task with ID 1	PUT /tasks/1	200 OK + updated task JSON	Pass (with correct code)
4	Update non-existent task (ID: 99) with bug active	PUT /tasks/99	404 Not Found	**Fail received** 500 Internal Server Error
5	Update non-existent task (ID: 99) after fix	PUT /tasks/99	404 Not Found + "Task not found" detail	Pass
6	Delete task with ID 1	DELETE /tasks/1	200 OK + "Task deleted successfully"	Pass
7	Delete non-existent task (ID: 99)	DELETE /tasks/99	404 Not Found	Pass
8	Create duplicate (ID: 1 again)	POST /tasks	400 Bad Request	Pass

FastAPI 0.1.0 OAS 3.1

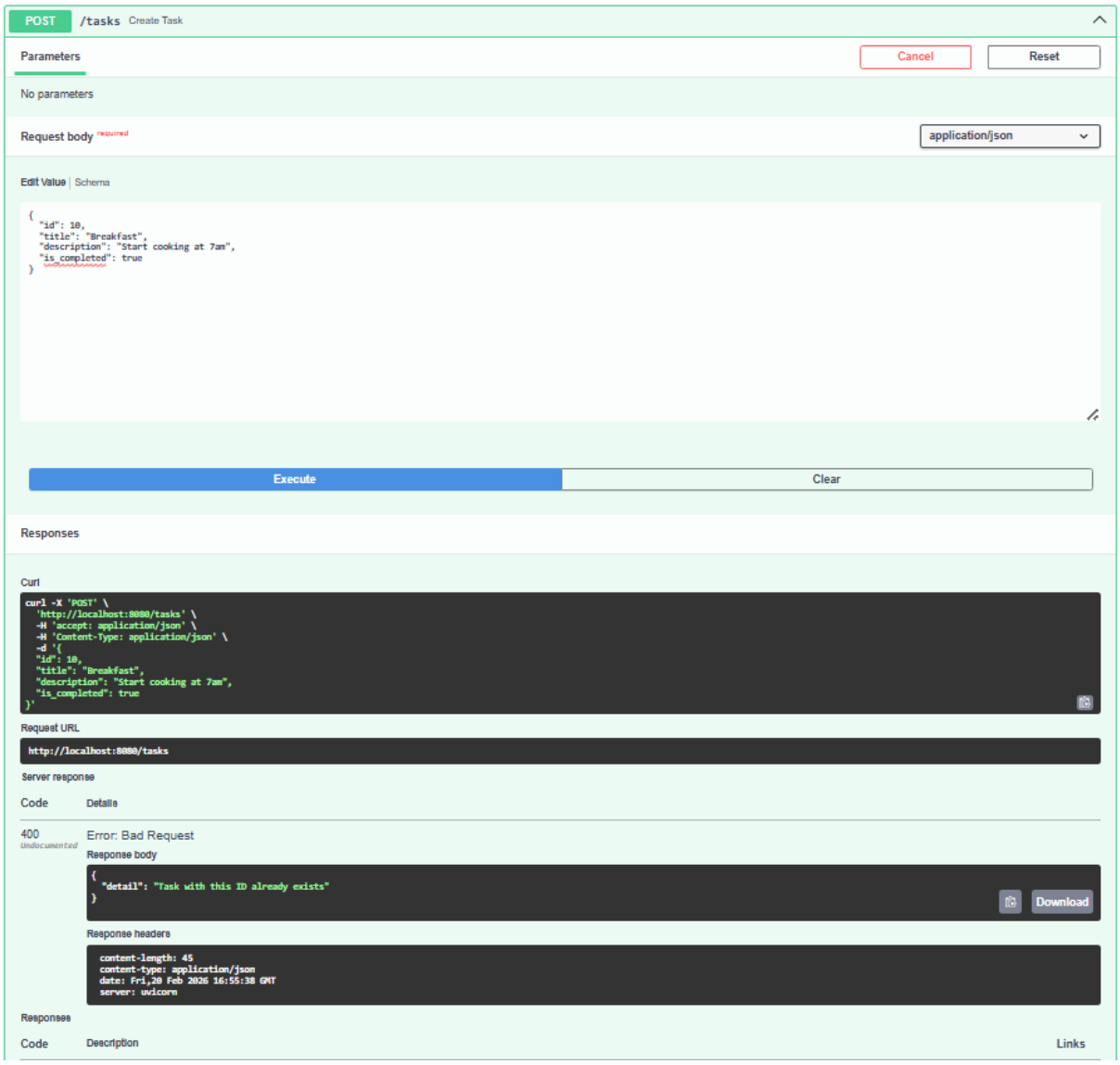
/openapi.json

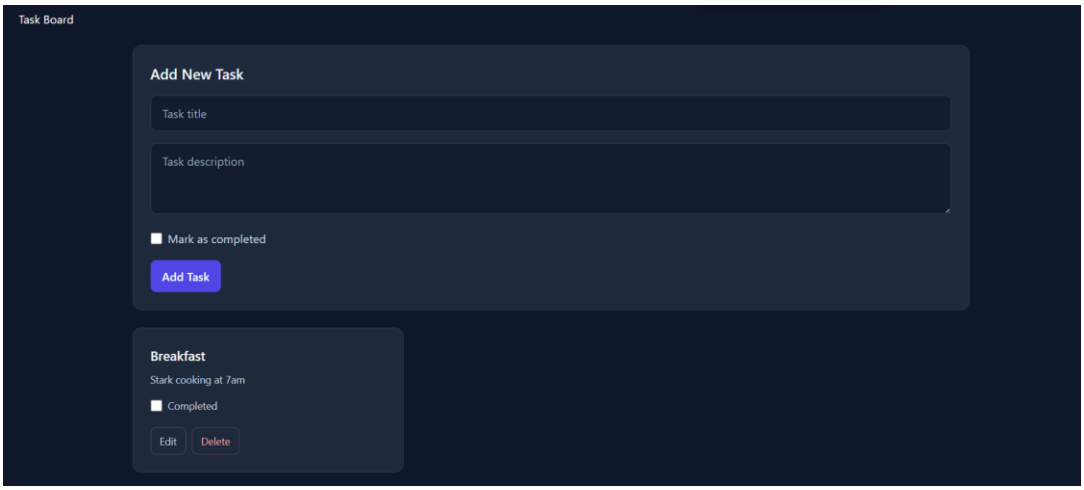
default

- GET /tasks Get Tasks
- POST /tasks Create Task
- GET /tasks/{task_id} Get Task
- PUT /tasks/{task_id} Update Task
- DELETE /tasks/{task_id} Delete Task

Schemas

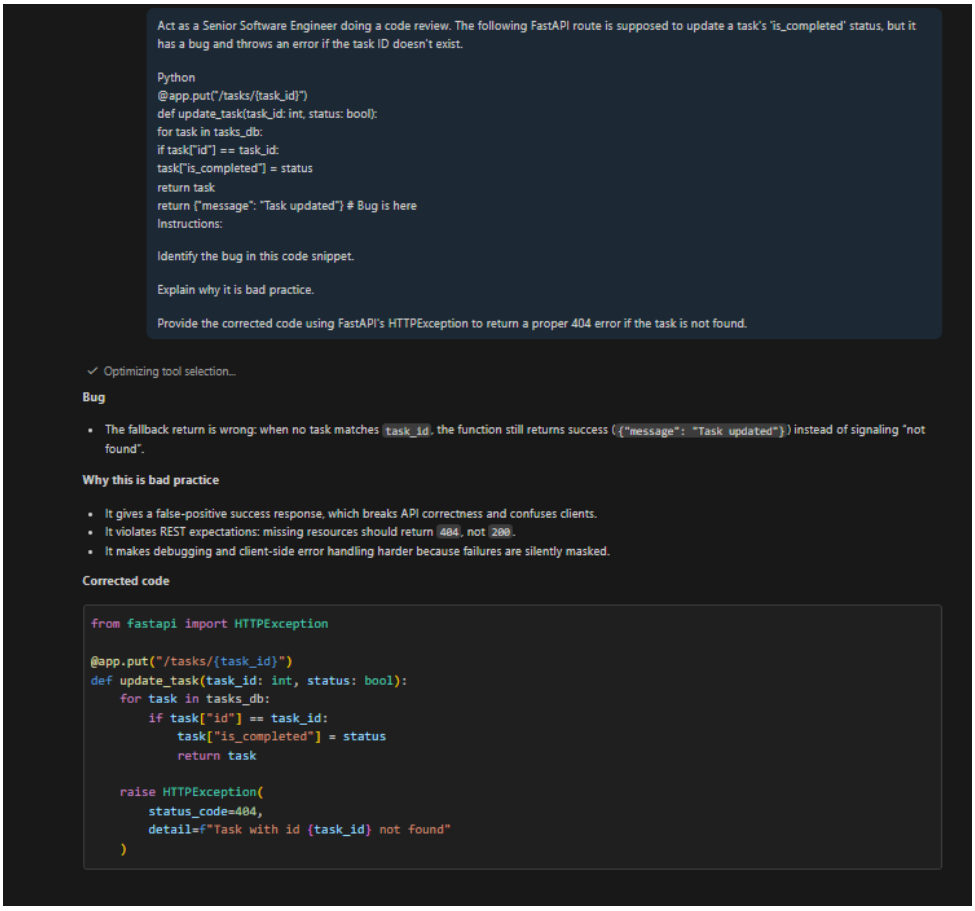
- HTTPValidationError › Expand all object
- Task › Expand all object
- ValidationError › Expand all object


POST
/tasks
Create Task
Parameters
Cancel
Reset
No parameters
Request body
required
application/json
Edit Value
Schema
{
"id": 10,
"title": "Breakfast",
"description": "Start cooking at 7am",
"is_completed": true
}
Execute
Clear
Responses
Curl
curl -X 'POST' \
'http://localhost:8080/tasks' \
-H 'accept: application/json' \
-H 'Content-Type: application/json' \
-d '{
"id": 10,
"title": "Breakfast",
"description": "Start cooking at 7am",
"is_completed": true
}'
Request URL
http://localhost:8080/tasks
Server response
Code
Details
400
Undocumented
Error: Bad Request
Response body
{
"detail": "Task with this ID already exists"
}
Download
Response headers
content-length: 45
content-type: application/json
date: Fri,20 Feb 2026 16:55:38 GMT
server: uvicorn
Responses
Code
Description
Links


Task Board
Add New Task
Task title
Task description
Mark as completed
Add Task
Breakfast
Stark cooking at 7am
Completed
Edit
Delete

Step 4 was the critical failure that confirmed the bug was effective. The Swagger UI response panel returned a 500 status with FastAPI's internal serialisation trace, instead of the expected 404. After restoring main.py:39,

Step 5 produced the correct 404 JSON response {"detail": "Task not found"}, confirming the fix.



5.3 Manual Improvements Made After AI Generation

Two specific improvements were identified and applied after critically reviewing the AI-generated output. These were engineering judgement calls, not corrections to syntax errors.

Improvement 1 Adding CORS Middleware to the FastAPI Backend

File: main.py

The problem: The **ChatGPT-5.3** backend contained no Cross-Origin Resource Sharing (CORS) configuration. A browser enforces the same-origin policy; any fetch() call from the React application running at **http://localhost:5173** to the API at **http://localhost:8000** is a cross-origin request and will be blocked silently by the browser before it reaches the server. This produces no error on the API side, making it extremely difficult to diagnose without knowledge of CORS mechanics.

Change made — add after app = FastAPI()**:**

```
from fastapi.middleware.cors import CORSMiddleware

app.add_middleware(
    CORSMiddleware,
    allow_origins=["http://localhost:5173"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

Engineering justification: Gemini 3 Pro included this block unprompted (visible in main.py:9-16), demonstrating that a correct full-stack prompt context should produce it automatically. The ChatGPT output was technically valid in isolation the API works perfectly when tested via Swagger UI or Postman from the same origin but would fail entirely in the browser environment for which it was generated. Restricting allow_origins to ["http://localhost:5173"] rather than the wildcard "*" also reflects a security-conscious engineering decision

Improvement 2 Fixing the Non-Interactive Completion Checkbox on Task Cards

File: TaskBoard.jsx:267

The problem: The AI generated the following input element inside each task card:

```
<input
  type='checkbox'
  checked={task.is_completed}
  readOnly          {/* ← Problem: user cannot interact with this */}
  className='h-4 w-4 ...'
/>
```

A controlled React input with checked but no onChange handler produces a React console warning (Warning: You provided a 'checked' prop... without an 'onChange' handler) and, more importantly, a non-functional UI element. The checkbox renders the task's current completion state correctly, but clicking it does nothing the status can only be changed by opening the full edit form. This violates the principle of least surprise for users who expect a checkbox to be a direct toggle control.

Change made:

First, add a toggle handler to the component (mirroring the pattern already present in the Gemini implementation):

```
const handleToggleComplete = async (task) => {
  const updatedTask = { ...task, is_completed: !task.is_completed }
  try {
    const response = await fetch(`http://localhost:8000/tasks/${task.id}`, {
      method: 'PUT',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify(updatedTask),
    })
    if (!response.ok) throw new Error('Failed to update task')
    setTasks((prev) => prev.map((t) => (t.id === task.id ? updatedTask : t)))
  } catch (err) {
    setError(err.message)
  }
}
```

Then replace the static checkbox on the card:

```
{/* Before */}
<input type='checkbox' checked={task.is_completed} readOnly ... />

{/* After */}
<input
  type='checkbox'
  checked={task.is_completed}
  onChange={() => handleToggleComplete(task)}
  className='h-4 w-4 ...'
/>
```

Engineering justification: From a UX standpoint, a toggle is the highest-affordance pattern for a boolean state users expect it to work. From a React standpoint, readOnly on a controlled input is semantically incorrect; the idiomatic approach is either a controlled input with onChange, or an uncontrolled input with **defaultChecked**. The correction also eliminates the React warning from the browser console, which is important for maintainability: console warnings in React often foreshadow runtime errors in future versions of the framework.

6. Final Evaluation of GenAI Tools

6.1 Reasoning & Debugging Capability

ChatGPT-5.3 demonstrated stronger code generation discipline for tightly-constrained specifications. When given a precise schema and a 60/30/10 rule, it produced an output that required fewer corrections. Its generated code was more architecturally considered the emptyForm reset pattern, the isSubmitting flag, and the properly structured dual-mode form (create/edit) all reflect an implicitly understood UX contract for form management. It was **the stronger performer for backend generation** clean, idiomatic Python, correct HTTP semantics.

Gemini 3 Pro better contextualised the full-stack deployment scenario. It included CORS middleware, docstrings, a richer REST surface, and input validation beyond what was explicitly prompted. When bugs were described in natural language, Gemini's explanations were more verbose and included reasoning chains (e.g., citing REST conventions to justify the GET /tasks/{id} endpoint addition). It was **the stronger performer for holistic system thinking**.

6.2 Speed & Usability

Both platforms responded within comparable latency ranges for prompts of the complexity used in this project (full-file code generation: approximately 10–20 seconds). Gemini 3 Pro's web interface rendered code with slightly better syntax colouring and offered a more accessible copy-to-clipboard interaction. ChatGPT-5.3's interface allowed longer conversation threads without visible context degradation and handled multi-turn refinement prompts more reliably without re-explaining prior decisions.

6.3 Context Window Limitations

A notable degradation was observed in both models during extended sessions. When the frontend prompt was submitted in the same conversation as the backend approximately 8–12 messages later both models required the JSON schema to be restated. Without the re-injected one-shot example (see Section 2.3), both produced frontend fetch payloads that omitted the id field, which the backend Pydantic model requires as a mandatory int. This confirms that **neither model reliably maintains schema context across a full full-stack generation session** without explicit re-anchoring. The practical mitigation is to open a fresh conversation for the frontend, paste the Pydantic model definition as context, and re-supply the JSON example.

6.4 Cost & Accessibility

ChatGPT-5.3 at the time of testing was available under the ChatGPT Plus subscription tier, with rate-limiting applied during high-usage periods. Gemini 3 Pro was accessible via the Gemini Advanced subscription, with comparable pricing. For free-tier users, both platforms offer degraded model versions (**GPT-4o mini and Gemini Flash respectively**) which produced noticeably less structured code in pilot tests suggesting that for serious engineering use cases, paid tier access is a practical requirement rather than an optional upgrade.

6.5 Summary Recommendation

Criterion	Recommended Tool
Backend API generation (strict constraints)	ChatGPT-5.3
Full-stack contextual awareness (CORS, REST completeness)	Gemini 3 Pro
Frontend UI richness (first-draft quality)	Gemini 3 Pro
CSS architecture & maintainability	ChatGPT-5.3
Code explanation depth	Gemini 3 Pro
Constraint adherence (zero-shot)	ChatGPT-5.3

Conclusion: Neither tool is unconditionally superior. **ChatGPT-5.3** is the preferred choice when precision, constraint adherence, and maintainable frontend code are the primary requirements. **Gemini 3 Pro** is preferable when the goal is rapid full-stack scaffolding with deployment-ready features generated with minimal prompting. A pragmatic engineering workflow would use **Gemini 3 Pro** for initial scaffolding and **ChatGPT-5.3** for constraint-sensitive refinement passes leveraging the complementary strengths of both platforms.