
CampusPulse – Task 1 Project Report

Executive Summary

This report details the complete journey of solving Task 1 for CampusPulse, a project designed to uncover the drivers of student relationships and well-being using real-world data. The focus is not only on the final solution, but also on the experimental process, the reasoning behind each step, and the strategies used to overcome various challenges.

---

1. Introduction

The goal of Task 1 was to analyze a rich student dataset to identify key factors influencing romantic relationship status, academic performance, and social behaviors. The dataset included demographic, academic, behavioral, and anonymized features, with some missing values and unclear variable meanings.

---

2. Data Cleaning and Preprocessing

Null Value Detection and Data Preparation

- **Initial Check:** Used isnull().sum() to identify missing values in all columns.
- **Type Conversion:** Converted relevant categorical columns (e.g., parental status, higher education aspiration) to numerical codes for better analysis and modeling.

Correlation Matrix and Smart Imputation

- **Correlation Matrix:** Created a correlation matrix to understand relationships between features and guide the imputation process.
- **Correlation-Based Imputation:**
    - Imputed Fedu (father's education) from Medu (mother's education) due to strong correlation.
    - Imputed higher (aspiration for higher education) using academic grades (G1, G2, G3).
    - Imputed traveltime using address (urban/rural).
    - Imputed freetime using goout (time spent out with friends).
- **Median Imputation:** For features like famsize and absences with low correlation to other variables, used median imputation.

- **Error Handling:** Initially encountered errors with model training due to remaining NaNs. Solved by ensuring imputation was completed before splitting data and model fitting.

---

3. Exploratory Data Analysis & Feature Identification

- **Feature 1 (F1):** Identified as a social/lifestyle variable based on strong correlation with alcohol consumption, number of failures, and higher relationship rates.
- **Feature 2 (F2):** Identified as an academic performance metric due to strong correlation with grades.
- **Feature 3 (F3):** Identified as a social activity metric, given its high correlation with drinks and time spent out with friends.
- **EDA Experiments:** Explored various visualizations (boxplots, bar charts, scatterplots) to confirm feature identities and answer key questions about student life and relationships.

---

4. Model Building, SHAP Analysis & Error Resolution

Model Construction

- **Algorithm:** Used Random Forest Classifier for its robustness and interpretability.
- **Feature Selection:** Chose features based on EDA and correlation analysis.

SHAP Model Interpretation

- **Global Importance:** Used SHAP summary plots to highlight the most important predictors (e.g., goout, freetime, family relation).
- **Dependence Plots:** Explored how features like social activity and free time interact to influence relationship status.

Error Handling

- **SHAP Plot Errors:** Encountered TypeError with unsupported parameters (dot_size). Fixed by removing these and customizing plots with matplotlib.
- **NaN Value Errors:** Ensured all missing values were imputed before model fitting.
- **SHAP Array Indexing:** Correctly sliced SHAP arrays (e.g., shap_values[:, :, 1]) for binary classification plots.

---

5. Results & Key Insights

- **Top Predictors:** Social activity (goout), free time, and family dynamics are the strongest predictors of being in a relationship.
- **Family Structure:** Students with lower family relationship quality or parents living apart are more likely to be in relationships.
- **Academic and Social Balance:** Academic performance and lifestyle choices both play important roles.

---

6. Reflections & Lessons Learned

- **Iterative Experimentation:** Multiple imputation strategies and feature engineering attempts were necessary to achieve clean, usable data.
- **Error Handling:** Addressed errors methodically, learning from each and improving the pipeline.
- **Transparency:** Documented both successful and failed approaches for reproducibility and future reference.

---

7. Technical Appendix

- All code and visualizations are available in the accompanying Jupyter notebook (task1.ipynb).
- For any questions or collaboration, contact the project maintainers.

This report highlights not just the final solution, but the full experimental journey, as required for Task 1 of CampusPulse.

Task 2 Project Report: WeatherMind Conversational Agent

Introduction

WeatherMind is a multi-tool, memory-enabled conversational agent built with LangGraph. The project demonstrates how to create a chatbot that can:

- Answer general queries
- Evaluate mathematical expressions
- Provide real-time weather information (Agro Monitoring API)
- Recommend city-specific fashion trends (Tavily API)
- Maintain context and memory across user turns

This report details the journey, approach, experiments, results, and error handling throughout the development process.

Level 1: Core Agent and Calculator Tool (task2file1.ipynb)

Approach

- Defined the State class for message passing.
- Initialized the graph and chatbot node using Google Gemini LLM.
- Implemented a BODMAS calculator tool as a Python function.
- Registered the calculator as a ToolNode and added it to the graph.
- Demonstrated correct tool invocation for math queries.

Results

- The agent correctly answered mathematical expressions (e.g., 3 + 6 * (5 + 4) / 3 - 7).
- For general queries, the agent responded conversationally via the LLM.

Level 2: Tool Integration, Routing, and Real APIs (task2file3.ipynb, task2file2.ipynb)

Approach

- Implemented get_agro_weather using the Agro Monitoring API for real-time weather by coordinates.

- Implemented get_fashion_trends_tavily using the Tavily API for city-specific fashion trends.
- Registered both as ToolNodes in the graph.
- Enhanced the routing logic:
    - If the LLM called a tool, the agent routed to the correct node.
    - If not, fallback keyword-based routing ensured weather/fashion/calculator nodes were used for relevant queries.
- Thoroughly tested each tool by direct invocation and through the agent.

Results

- The weather tool returned live weather for valid coordinates (e.g., Tokyo).
- The fashion tool surfaced trends from Tavily for cities like Tokyo and Paris.
- The agent could now answer, in a single conversation, both general and tool-specific queries.

Level 3: Conversational Memory & Multi-Turn Dialogue (task2file4.ipynb)

Approach

- Integrated LangGraph's InMemorySaver checkpointer to persist conversation state.
- Used a unique thread_id in the config to maintain state across turns.
- Modified the invocation pattern:
    - For each user message, only the new message is passed;
    - LangGraph automatically restores the conversation history.
- Demonstrated the agent's ability to:
    - Answer follow-up questions (e.g., "And what about tomorrow's weather at the same place?"),
    - Maintain context (e.g., remembering the last location or topic),
    - Route queries to the correct tool or fallback to the LLM as needed.

Results

- The agent maintained memory between turns, as shown by context-aware responses.

- If the user asked about “the same place” in a follow-up, the agent could reference the previous location.
- The agent gracefully handled tool errors (e.g., missing API data) and provided fallback responses.

Error Handling and the Journey

1. Simultaneous Tool/Node Creation Errors

What happened:
When trying to add multiple tools or nodes after compiling the graph, errors like
ValueError: Node 'fashion_tools' already present.
and
WARNING:langgraph.graph.state:Adding a node to a graph that has already been compiled. This will not be reflected in the compiled graph.
persisted.

How I fixed it:

- I learned that in LangGraph, all nodes and edges must be added before compiling the graph.
- To debug, I created separate files and separate chatbots for each tool, ensuring each worked independently.
- Finally, I unified all tool nodes in a single graph, added all edges and routing before compiling, and then compiled the graph once.

2. Checkpointer/Memory Integration Errors

What happened:
When using the checkpointer, I encountered
ValueError: Checkpointer requires one or more of the following 'configurable' keys: ['thread_id', 'checkpoint_ns', 'checkpoint_id'].

How I fixed it:

- I ensured every graph invocation included a config with a unique thread_id:

python

config = {"configurable": {"thread_id": "user-123"}}

- This allowed LangGraph to persist and restore conversational memory between turns.

3. LLM Not Using Tools (Fallback Answers)

What happened:
The LLM would sometimes answer directly (e.g., “I cannot provide real-time weather information...”) instead of invoking the weather or fashion tool.

How I fixed it:

- Ensured tools had clear and explicit docstrings.
- Improved the system prompt to encourage tool use.
- Added keyword-based fallback routing in the routing function, so relevant queries are always routed to the correct tool node even if the LLM does not produce a tool call.

4. Weather API Not Returning Data

What happened:
Using the /weather/forecast endpoint often returned no data for arbitrary coordinates.

How I fixed it:

- Switched to using the /weather endpoint for current weather, which reliably returns data for lat/lon pairs.

5. Memory Not Maintained Between Turns

What happened:
When appending messages manually to the state, memory was not persistent and context was lost between turns.

How I fixed it:

- Stopped manually appending to the messages list. Instead, for each turn, only the new user message is passed and LangGraph memory (with checkpointer and thread_id) handles the rest.

Key Learnings

- **LangGraph requires all nodes/edges to be defined before compiling.**
- **Memory integration is seamless if you use a checkpointer and thread_id, and pass only the new user message per turn.**
- **Tool invocation depends on clear docstrings, proper registration, and sometimes prompt engineering.**
- **Debugging in isolation (separate files/tools) helps resolve complex integration issues.**

Conclusion

WeatherMind demonstrates a modern, multi-tool conversational agent with memory and reasoning, built using LangGraph.
The project covers:

- Core agent and tool setup,
- Real-world API integration,
- Intelligent routing,
- Persistent conversational memory,
- Robust error handling and debugging.

Each level is documented in a separate notebook, showing the evolution from a basic chatbot to a full-featured, context-aware agent.

For code and further details, see the corresponding notebook files for each level.