CSC510 Fall 2025: Software Engineering

Proj1d1 Solutions

Group number: 25

Team Members:

1. Shreyas Raviprasad (sravipr@ncsu.edu)
2. Smruthi Bangalore Thandava Murthy(sbangal6@ncsu.edu)
3. Swasti Sadanand(ssadana@ncsu.edu)
4. Vineeta Vishwas Bhujle(vbhujle@ncsu.edu)

Github Repository Link:

https://github.com/shreyas457/SE_G25/tree/main

What are the pain points in using LLMs?

In requirements engineering, the first prompt and its initial response from an LLM are rarely sufficient. They almost always require additional context and iterative back-and-forth with the user to clarify what's actually needed. Another challenge lies in the limits of context window sizes, which constrain how much information can be provided at once and may reduce the quality of prompts and responses. While techniques like Retrieval-Augmented Generation (RAG) and frameworks such as DSPy can help overcome these limitations, setting them up often requires additional effort. Complicating matters further, the landscape of LLMs is highly fragmented: models differ in architecture, parameter size, and training data, making it unclear which one is best suited for a given task without experimentation.

Any surprises? Eg different conclusions from LLMs?

What surprised us during our previous projects was how differently the LLMs approached the same problem. ChatGPT, for example, consistently leaned toward a business-oriented perspective, while Claude's answers were more academically precise. Claude even went a step further by generating a fully formatted document that listed the use cases and stakeholders it addressed. Even more striking, GPT chose to downplay certain stakeholders it deemed less critical and instead prioritized pitching to investors as the more effective route to securing seed funding.

What worked best:
Providing examples to ChatGPT and asking it to generate similar use cases worked exceptionally well. This gave the LLM context, enabling it to produce richer, more structured, and relevant scenarios for the online food delivery domain. We also asked each LLM to suggest which use cases it considered most important for the MVP, then selected the ones we judged to be most suitable, incorporating a human-in-the-loop process. Additionally, having one LLM critique the outputs of another model proved valuable, as it helped us understand the reasoning and thought process behind different suggestions, uncovering ideas or considerations we might have otherwise missed. Iteratively providing context and examples was much more effective than single-shot prompting, which often produced superficial or irrelevant outputs. Some generated use cases were variations of the same scenario, requiring manual filtering to remove duplicates. This combination helped generate additional useful ideas, refine ambiguous outputs, and ensure the scenarios were actionable and aligned with real-world system requirements.

What worked worst:
When using retrieval-augmented generation (RAG) without providing sufficient background information about the system, the LLM often produced very generic or unrealistic outputs. For example, it suggested routine SDE-level tasks like making API changes, which did not meaningfully reflect the operational needs of a food delivery app. Critical aspects, such as system availability during peak hours, operational checks, and edge cases in the ordering or delivery workflow, were often missed. This highlighted

that LLMs struggle to account for domain-specific constraints or business-critical requirements without explicit context and guidance.

What pre-post processing was useful for structuring the import prompts, then summarizing the output?

When we used GPT to generate use cases and UML diagrams after identifying stakeholders, the most effective improvements came from how we structured the inputs and refined the outputs. Pre-processing involved making the prompts clean, consistent, and predictable. We removed boilerplate or repeated text, standardized terminology so the model would not confuse “restaurant,” “vendor,” or “partner,” and split long descriptions into small, role-specific chunks to keep focus. Adding simple tags like Customer – discounts or Delivery Agent – availability gave GPT clear anchors. Providing a scaffold with headings such as Preconditions, Main Flow, Alternatives acted like a template so the model filled in slots instead of writing freely. In retrieval-augmented steps, overlapping chunks and normalized embeddings helped preserve context, and numbering retrieved snippets made it easy to cite and trace sources.

Post-processing was just as valuable. We performed consistency checks to ensure terminology and depth matched across stakeholders. Long responses were compressed into shorter summaries, and duplicates were merged into a single version. A second pass to condense large paragraphs into brief abstracts made the content easier to compare. Formatting into PlantUML diagrams was delayed until the text was finalized, which avoided brittle or shallow outputs. For retrieval-grounded answers, we kept snippet IDs with the output and revised anything that lacked a clear source, making the results more accurate and auditable.

Did you find any best/worst prompting strategies?

Yes, we found prompting strategies. The strategies that worked best shared three traits: clarity, staging, and use of examples. Showing GPT a small, concrete example of the exact structure we wanted such as a miniature use case with headings produced consistent outputs. Explicitly naming the roles in the prompt kept the model aligned with stakeholders. Breaking the task into stages also helped. First generating plain summaries or notes, then asking for diagrams. This “summarize first, structure later” approach gave more coherent logic and fewer errors. In retrieval mode, instructing GPT to answer only from the numbered context and cite after each fact improved reliability. The least effective strategies were overloaded one-shot prompts that asked for cleaning, summarization, and diagramming in a single step, which often lost detail or broke formatting. Vague instructions like “make this better” produced unpredictable results. Forcing strict diagram syntax too early created outputs that looked correct but had shallow or inconsistent content. In practice, the strongest results came from normalizing inputs, guiding GPT step by step with examples, verifying the content first, and only then generating structured diagrams.