product

idea

Building Integral Solutions for Tomorrow

INTEGRATED MATH SUPPORT LLM SCHEMA

Documentation of the model for the math support AI model used in TxikI/A

Second draft: 17/04/2025

Disclaimer

Changelog

- Second draft (17/04/2024):
    - Changed all multi-valued outputs to JSON objects, removing old "¿¿" separation. This ensures a cleaner, more predictable and easier to work on output.
    - Added specification for question generation and grading.

0. Index

1. Introduction

The flagship project of ACC is TxikI/A, a math suite powered by a Large Language Model ("LLM") offering various assistance tools to the user, such as:

- Concepts, terms and exercises explanation.
- Generation of support code for use in the SageMath suite.
- Generation of multiple-answer quizzes and open-answer questions, to help the user test their knowledge on the topic.

Given the complexity of the software, and the various difficulties that arise when working with a LLM, it is essential to offer a clear, modular structure for handling the user requests. This includes tasks as:

- **Filtering the intent of the user request**: Is it a question or a quiz request? What concept are they asking about?
- **Deciding the needs in real time**: Is code needed? Does the program have to generate a graph to illustrate a function?
- **Offering a predictable output**: When working with content such as quizzes, it is essential that there will not be unexpected runtime errors caused by incorrect LLM outputs. Therefore, a standard format for these has to be enforced.

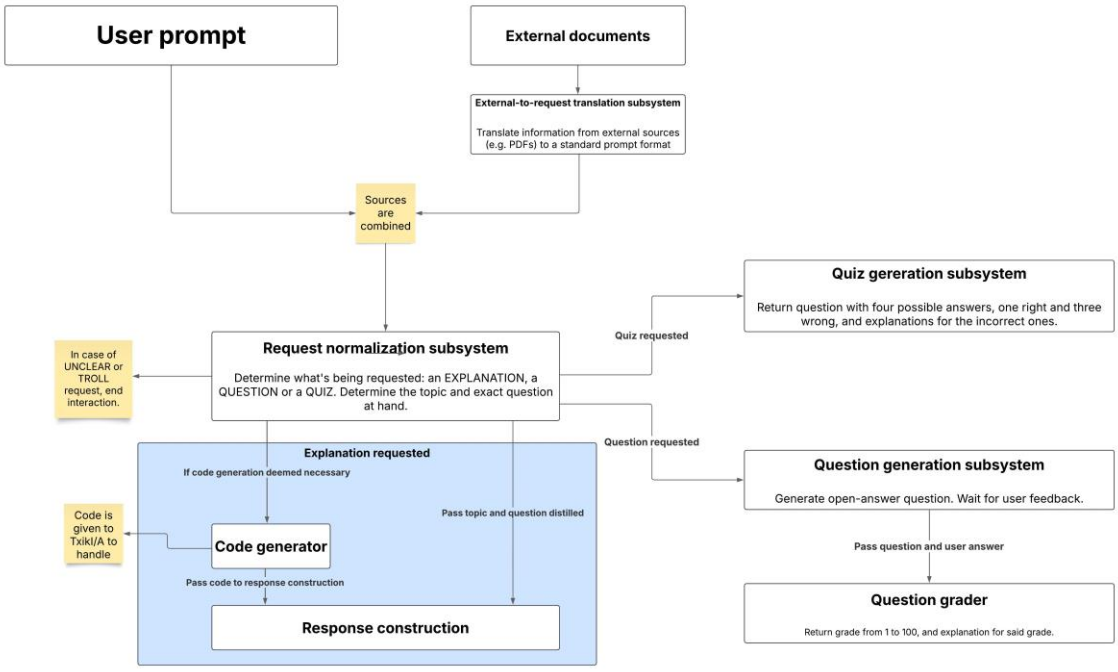
This document offers a coherent, enforceable standard for the whole process. It categorizes the different steps of the content generation in various logical **subsystems** (modules), and offers a standard input and output format for each of them. Modules implementing LLM generation must enforce these standards, and software modules counting on LLM outputs must expect the format described in this document.

2. General workflow overview: subsystems

Due to the inherent non-determinism of LLMs, trying to condense all the steps of the answer generation in one single giant prompt is not only impractical but incredibly error-prone, as the guarantee of getting a correctly formatted and reasonable answer is never absolute. For this reason, *single-order prompts* are preferred: highly detailed prompts that only handle a small step of the process. Motivated by this idea, the LLM system of TxikI/A works on a layered module (*"subsystem"*) schema, with these steps in mind:

- **Interpret the request**: Decide what is being asked (an explanation or a quiz/question), and dismiss troll or unclear requests.
- **Generate the relevant content to serve the request**: For example, in the case of a quiz, generate the question, the four possible answers, and an explanation of what is wrong in each incorrect one.
- **Serve the system with the generated information in a standardized manner**: All the information and distilled context must be served back to the software in a way that guarantees that the event flow will be satisfactory (i.e. in a format that the program understands and can handle).

The subsystems should have logical connections, and pass information (or request content generation) to one another as described by the following workflow schema:



These are, on a high level, the roles of the subsystems:

- **External-to-request translation subsystem**: External media (PDFs, scanned images, etc.) can come from various sources and in various (structured and non-structured) formats. Therefore, if any document is uploaded by the user as part of their request (for example, handwritten notes or a copy of a past paper), it must be adequately processed for it to be added to the final prompt.
- **Request normalization subsystem**: Once the user prompt is received, it is necessary to infer the intent of it, alongside with the required system resources to serve it. This subsystem "normalizes" all prompts to the same format, in two blocks:
    - **Resource needs**: Do we need an explanation, a quiz or an open answer? Do we have to generate code?
    - **Formalized question**: The module interprets the question of the user, a generates a formal/corrected version of it, which allows for a better explanation later, given that the user might ask the question in a confusing way.
- **Code generation subsystem**: If it is determined that code is useful or needed in the answer, this subsystem generates it and sends it back both to TxikI/A and to the response construction.
- **Quiz generation subsystem**: This system generates a question with four possible answers, one marked as correct, and an explanation for why the other three are incorrect.
- **Question generation** and **question grader**: The question generator generates an open question, which the user has to answer. Once the user has sent their answer, the question grader will return both a grade (from 1 to 100) and three pieces of feedback: the correct aspects of the answer, the incorrect ones, and a proposed solution.

For each of the systems, this document provides the following:

- **Functional requirements for the output of each subsystem**: Each subsystem must follow a strict protocol in the format of its answer, so that it can be safely processed by the backend code of TxikI/A.
- **Prompt suggestions and notes**: LLMs are prone to hallucinations and undesired behavior, so prompt-engineering requires extreme care to avoid adverse effects (such as in-line format attempts).

To split different sections of the outputs, they will be returned as JSON objects when necessary.

3. Request normalization subsystem

The request normalization subsystem does two tasks: infer the user intent, and formalize their question, so that better content can be generated from it.

- **Expected input (regex)**: ^USERPROMPT (.+?)
  (?:CONTENT_FROM_ATTACHED_DOCUMENT (.+)|NO_DOCUMENT_ATTACHED)$
    - Word USERPROMPT followed by a space.
    - Full user prompt, as written by them, and a space.
    - If external content is attached:
        - Word CONTENT_FROM_ATTACHED_DOCUMENT followed by a space.
        - Information from document, as extracted by the external-to-request translator.
    - Otherwise:
        - Word NO_DOCUMENT_ATTACHED.

- **Expected output (JSON)**:
    - **question_type**: One word. Will be either EXPLANATION, QUIZ, QUESTION, TROLL or UNCLEAR.
        - **[only if question_type=EXPLANATION]** code_needed: Will be either CODE or NOCODE. CODE if the LLM determines SageMath code is helpful in the answer.
    - **formalized_request**: The formalized version of the request. An ambiguous request is translated into a clear, direct one, with adequate terminology, so that a better explanation and code can be designed.
        - This SHOULD include an assessment of the user's familiarity with the topic, so that the explanation and questions are adapted.
    - **attached_data**: If a document has been attached, a full explanation of the content MUST be provided in this formalization. That is: a list of all the contents explained, in order and with the nuances of the document, **AND** the raw data of the exercises (if available), numbered and ordered.

4. Code generation subsystem

The code generator will receive as input the formalized request, and will ONLY output the text corresponding to the script. That is, the output MUST be able to be piped into a script file that's executable in SageMath, and contain NO OTHER DIALOGUE.

Some points should be emphasized on the system prompt:

- Only the code has to be outputted, and NO OTHER DIALOG or FORMATTING.
- Code MUST be well commented and readable, including a header comment carefully explaining what the script does.
- The subsystem must be context-aware: it has been determined that code should be generated based on a request for an explanation. Thus, the type of output should be coherent with that (e.g. being careful with which concepts are used according to the user expertise). This naturally extends to the functionality of the code: it doesn't make sense to implement complex behaviours for a simple explanation.
- The script should log every important step: if, for example, the user wants to visualize a Taylor expansion getting closer to the original function as terms increase, this behaviour should be shown in a "step-by-step" fashion in the script's execution, with explanations so that the user understands what is going on.

5. Quiz generation subsystem

The quiz generation subsystem is responsible for creating a question with four possible answers each, one correct. For the other three answers, it will give a short explanation on why the answer is wrong.

Note that to prevent unexpected behaviour in the output, this system has to be called once per question, instead of generating all questions at once. On each call, a history of already asked questions can be passed to the subsystem, to avoid repeated questions on the output.

- **Expected input (regex)**: ^(PAST_CONTEXT (.+?)) ALREADY_ASKED_QUESTIONS (.+?) CURRENT_QUIZ_PROMPT (.+?)$
    - The word PAST_CONTEXT followed by a space and all the chat context, as it has been generated in the response construction, and a space.
    - **Optional**: If more questions have been generated in the quiz already, the word ALREADY_ASKED_QUESTIONS followed by a space, the list of questions in text, and another space at the end.
    - The word CURRENT_QUIZ_PROMPT followed a space and the current (formalized) quiz prompt.

- **Expected output (JSON**):
    - **quiz_question**: The quiz question.
    - **correct_answer**: The correct answer to the question.
    - **incorrect_answer1**: The first incorrect answer to the question. A ¿¿ separates the answer from the explanation on why it is wrong.
    - **incorrect_answer2**: The second incorrect answer to the question. A ¿¿ separates the answer from the explanation on why it is wrong.
    - **incorrect_answer3**: The third incorrect answer to the question. A ¿¿ separates the answer from the explanation on why it is wrong.

Some notes for the system prompt:

- There must be NO inline formatting on the questions. Special attention must be put on preventing mistakes like adding double asterisks (**) as a means of adding bold type text. Such features might be useful on other chatbot contexts but completely break the program flow here.
- On the same line, no additional punctuation (such as unnecessary colons) should be added in the answer for presentation purposes.
- The LLM has to decide whether the past context in the chat is relevant. For example, if the context shows that the user has been talking about genetic algorithms, and then suddenly asks for a test on basic trigonometry, the subsystem is not forced to somehow relate them, unless explicitly asked by the user.
- The LLM must NOT enumerate the answers, just write them.
- Questions must NOT be about numerical values. Those might work well on extremely simple operations, such as asking for the result of a product, but do not escalate to more complex operations, where the LLM is not actually using a calculator. Instead, the quiz must lean toward conceptual questions, even if the user requests otherwise.

6. Response construction subsystem

The response construction subsystem is responsible for crafting the final answer when explaining a concept. Note that a **REASONER** model must be used in this subsystem, if available.

- **Expected input (regex)**: ^(.?)\sPAST_CONTEXT\s*(.?)\sGENERATED_CODE\s*(.*?)$
  - Formalized request followed by a space.
  - PAST_CONTEXT and all past context from the chat, followed by a space.
  - GENERATED_CODE and the generated code, if any.
    - If no code has been generated, it must be said here to the LLM (e.g. "No code has been generated").
- **Expected output (JSON):**
  - **full_answer**: Full answer to be displayed to the user.
    - **NOTE:** The LLM can embed LaTeX in the answer by wrapping it inside <latex-js> HTML tags. The class has to be either container or graph.
    - **NOTE**: If code has been provided, the LLM must give a BRIEF explanation of what it does, and why it has been included there.
  - **answer_context**: A paragraph resuming what the user asked and what the LLM explained, without entering in detail.
    - This will be used for "context" in future inputs, so that the LLM does not have to re-read the whole chat every time.

Some notes for the system prompt:

- Ensure NO ATTEMPTS AT FORMATTING the answer, such as adding double asterisks (**).
- LLM must be aware of its capabilities (e.g. using latex, or offering to create a quiz/question), obligations (explaining the code, adjusting the answer to the knowledge level of the user, etc.) and limitations (it should not hallucinate features such as in-line code execution).
- The explanation must be informational yet friendly, adjusting to the knowledge level of the user. Conversation should feel natural, not overly dry.

7. Question generation and grading subsystems

The question generator receives a topic and generates an open-ended question (that is, a question that requires an elaborate answer). The user types their answer out, sends it, and the question grader evaluates it, offering both a grade and an explanation.

There are the input and output formats for the question generator:

- **Expected input for the generator (regex): ^TOPIC\s+(.+?)\s+QUESTIONS_ALREADY_ASKED\s+(.+)$**
    - Word **TOPIC** followed by the formalized prompt from which the question arises, and a space.
    - Word **QUESTIONS_ALREADY_ASKED** followed by the history of previous questions. The goal is to prevent asking the same question many times.

- **Expected output**: A string with the question.

Some notes for the system prompt:

- If there are already asked questions, the new question should try to differ in some meaningful way from them, while still being interesting and not getting out of bounds in terms of difficulty.
- Questions should focus on a conceptual side, rather than a numerical one. On the case of numerical algorithms, for example, asking the reasoning behind certain steps or instructions on how to apply them is better than just asking for the expected result in a certain execution.

These are the input and output formats for the question grader:

- **Expected input for the question grader (regex): ^CONTEXT\s+(.+?)\s+QUESTION\s+(.+?)\s+USER_ANSWER\s+(.+)$**
    - Word **CONTEXT** followed by the formalized prompt, and a space.
    - Word **QUESTION** followed by the question generated before, and a space.
    - Word **USER_ANSWER** followed by the full user answer.

- **Expected output (JSON)**:
    - **feedback**: A text with the feedback that the user will receive.
    - **grade**: A numeric integer value from 1 to 100.

Some notes for the system prompt:

- The grader should try to balance being strict and being forgiving. That means blatantly incorrect questions should never pass (>50/100), but very-well answered questions shouldn't be overly penalized by small mistakes.
- The grader should penalize correct answers that don't offer an adequate justification. The goal is to show understanding, not just a quick ability to answer.
- [**Future idea; do not implement yet**] The grader could execute SageMath code internally to check for numerical computations.