✅ Part 1: Preprocessing & Encoding Framework Design

This section will guide the developer on building a modular preprocessor to transform raw spreadsheets into compact, token-efficient encodings that preserve structural integrity.

📌 Subsections:

1. **Spreadsheet Parsing**:
    - Input: `.xlsx` file using `openpyxl` or `ClosedXML`.
    - Extract: cell values, addresses, format types, merged cells, borders, font style, fill color.
    - Output: structured matrix representation (`M`) with metadata.
2. **Data-Type & Format Inference**:
    - Implement NFS parsing.
    - Add rules to infer data types: `IntNum`, `FloatNum`, `Percentage`, `DateData`, `EmailData`, etc.
    - Store in a `FormatDict {(r,c): val_type}`.
3. **Encoding Module**:
    - Implement a **Markdown-like encoding** that serializes `[(value, cell_address)]` or `[(value_type, range)]`.
    - Use inverted-index style where repeated values/types share cell ranges.
4. **Token Simulation**:
    - Estimate token count using `tiktoken` or similar.
    - Include checks to ensure final encoding doesn't exceed limits (e.g., 4K tokens).

✅ Part 2: Core Compression Modules (SHEETCOMPRESSOR)

This part focuses on implementing the **three core compression modules** with full logic and reusable abstractions.

📌 Subsections:

1. **Module 1: Structural-Anchor-Based Extraction**
    - Detect boundary rows/columns based on:
        - Cell content variation (text/numbers/colors/borders).
        - Heuristics for title/header/note regions.
    - Retain rows/cols within **`k=4`** of these anchors (ablation-tested optimum).
    - Output: reduced matrix `Se`.
2. **Module 2: Inverted-Index Translation**
    - Scan matrix for **repeating values** or **empty cells**.
    - Group repeated values under one key with their address ranges.
    - Format into JSON-like dict: `{ value_type: [range1, range2, ...] }`.
3. **Module 3: Data-Format-Aware Aggregation**
    - Use Algorithm 1 (DFS over similar `val_type`) to merge adjacent cell blocks.

- Output: list of rectangular regions representing compressed semantic clusters.

4. **Coordinate Remapping**:
   - Re-index cells to maintain continuity post-compression.
   - Map original addresses to new ones for model compatibility.

✅ Part 3: Downstream Task Logic – Table Detection & QA

This section will describe how to implement logic for two downstream tasks: **table detection** and **QA**, including chunking logic for large regions.

📌 Subsections:

1. **Table Detection (via GPT Prompting)**:
   - Use compressed encoding as input.
   - Use **SPREADSHEETLLM prompt** format.
   - Predict table ranges like `["range": "A1:F10"]`.
2. **Table Split QA Logic**:
   - Implement Algorithm 2:
     - Check if region < 4096 tokens → process directly.
     - Else, split into `header + body` chunks.
     - Feed into LLM with **CoS Stage 1 and Stage 2 prompts**.
   - Use the `answer_question(question, table)` method.
3. **Module Ablation Support**:
   - Allow toggling modules (`--no-aggregation`, `--no-translation`) for QA ablation experiments.
   - Log region detection + answer accuracy.
4. **Cost & Token Logging**:
   - Log token count before/after each module.
   - Track estimated LLM cost per spreadsheet (per token price table from OpenAI).