SPREADSHEETLLM Framework

- **Goal:** Enable LLMs to "understand" and reason over full spreadsheets despite token-length and 2D-layout challenges.
- **Initial (Vanilla) Encoding:**
  - Serialize every cell as a sequence including its address (e.g. "A1"), value, and format.
  - *Limitation:* Large sheets exceed model token limits and performance degrades.

- **SHEETCOMPRESSOR Encoding Pipeline**

To dramatically shrink input size while preserving layout and content, three modular steps are applied in sequence:

1. **Structural-Anchor Compression**
   - Detect "anchors" (rows/columns where content changes) that mark table boundaries.
   - Discard distant, homogeneous rows/columns to yield a compact "skeleton" of key structure.
2. **Inverted-Index Translation**
   - Build a JSON dictionary mapping each unique non-empty cell text to all its addresses.
   - Replace repeated values with index lookups, preserving losslessness but slashing tokens.
3. **Data-Format-Aware Aggregation**
   - Group adjacent numeric cells sharing the same format string (e.g. "$#,##0.00") or data type.
   - Represent each rectangular block by its format, avoiding redundant numeric **1. Structural-Anchor-Based Extraction**

Goal: Identify and retain only structurally important (heterogeneous) rows/columns to produce a skeleton version of the spreadsheet.

Implementation Steps:

- **Input:** Raw spreadsheet with all cells.
- **Step 1 – Detect Structural Anchors:**
  - Use heuristics or statistical features to identify candidate table boundaries (e.g., rows with headers, or columns that differ significantly in format/content from neighbors).
  - These are the yellow-highlighted vertical and horizontal anchor lines in panels (a) and (b).
- **Step 2 – Remove Redundant Rows/Cols:**

  - Delete rows/columns that are homogeneous and far from detected anchors (e.g., repetitive numeric rows far from headers).
  - Retain only surrounding context (within `k` cells) from each structural anchor.
- **Output:** A significantly reduced spreadsheet—e.g., from **576×23 → 24×8** (as in panel (c)).

---

2. Inverted-Index Translation

Goal: Optimize serialization by grouping identical cell values and storing them as key-value pairs where the key is the value and the value is a list of cell addresses.

Implementation Steps:

- **Input:** Skeleton sheet from Step 1.
- **Step 1 – Traverse the Sheet:**
  - For each non-empty cell, check if its value already exists in a dictionary.
  - If so, append the cell's coordinate (e.g., `B4`) to that value's list. If not, create a new entry.
- **Step 2 – Represent as JSON Dictionary:**
  - Structure: `{ "cell_text": [list of coordinates] }`
  - Examples from the encoding block:

```
json
CopyEdit
{
  "Atlantis": ["A2", "A7", "F5"],
  "QuantumMind": ["B2", "B3", "B4", "F4"],
  "20-Aug": ["D2:D18", "D21:D23"]
}
```

- **Output:** A JSON index that eliminates repeated values and avoids encoding empty cells.

---

3. Data-Format-Aware Aggregation

Goal: Group and represent adjacent numeric cells that share identical formats to further compress the spreadsheet.

Implementation Steps:

- **Input:** Inverted-index-translated sheet.
- **Step 1 – Scan for Format Similarities:**
  - Identify contiguous numeric cells (same data type or format string).
  - Examples: currency (`$#,##0.00`), integer, percentage.

- **Step 2 – Group Cells by Format:**
    - Merge them into labeled format clusters, e.g.:

      ```
      json
      CopyEdit
      {
        "IntNum": ["C2:C4", "C6:C12"],
        "Percentage": ["H5:H6", "H8:H9"]
      }
      ```

- **Output:** Further compressed index that encodes ranges by shared formatting blocks.

3 Method: Overview

We represent a sheet `S` as an `m×n` grid of cells and turn it into text via three independent, composable modules. First up:

3.1 Vanilla Spreadsheet Encoding

Goal: Turn every cell into a short Markdown-style line containing its address, value, and (optionally) format.

1. **Data Structures**

   ```
   # S: 2D array or dict mapping (i,j) → Cell(value, format)
   text_lines: List[str] = []
   ```

2. **Loop over all cells**

   ```
   for i in range(1, m+1):
       for j in range(1, n+1):
           cell = S.get((i,j))
           if not cell or cell.is_empty():
               continue
           addr = f"{col_letter(j)}{i}"      # e.g.  A1, B3, …
           # Omit `cell.format` if you want faster token usage.
           line = f"|{addr},{cell.value},{cell.format}|"
           text_lines.append(line)
   ```

3. **Concatenate into a single string**

   ```
   T_vanilla = "\n".join(text_lines)
   ```

Note: Including rich format metadata (colors, borders, fonts) rapidly exhausts LLM token limits and often hurts accuracy—so default to value + address only, or prune formats to the essentials.

3.2 Structural-Anchor-Based Extraction

Goal: Strip away "boring" homogeneous rows/columns far from any table boundary, keeping only a small "skeleton" that preserves layout cues.

1. **Measure Heterogeneity**
   - For each row `p`, compute `row_score[p]` = number of distinct cell values or format changes in that row.
   - For each column `q`, compute `col_score[q]` similarly.
2. **Select Anchors**

   ```
   anchors_rows  = { p | row_score[p] ≥ θ_row }
   anchors_cols  = { q | col_score[q] ≥ θ_col }
   A = anchors_rows ∪ anchors_cols
   ```

   You can also include first/last row+col by default.

3. **Define Neighborhood Radius**

   ```
   CopyEdit
   k = user_defined_threshold  # e.g. 2 or 3
   ```

4. **Build Retained Row/Col Sets**

   ```
   kept_rows = ⋃_{p ∈ anchors_rows} { i | |i - p| ≤ k }
   kept_cols = ⋃_{q ∈ anchors_cols} { j | |j - q| ≤ k }
   ```

5. **Extract Skeleton Cells**

   ```
   CopyEdit
   S_skeleton = {  (i,j): S[i,j]
                   for i in kept_rows
                   for j in kept_cols
                   if S[i,j] not empty }
   ```

6. **Coordinate Re-mapping**

   ```
   # Remap old row indices to 1…m′, and cols to 1…n′
   new_row_map = { old_i: new_i for new_i, old_i in
   enumerate(sorted(kept_rows), start=1) }
   new_col_map = { old_j: new_j for new_j in enumerate(sorted(kept_cols),
   start=1) }

   S_extracted = {
       (new_row_map[i], new_col_map[j]): cell
       for (i,j), cell in S_skeleton.items()
   }
   ```

   This preserves the logical relationships (e.g. "A1" → "A1" in the compressed grid).

7. **Compression Stats**
   - Typically drops ~ 75% of cells
   - Still retains ~ 97% of boundary rows/columns
8. **Output for Next Step**

```
Se = S_extracted
Te = vanilla_encode(Se)  # reuse 3.1 on the smaller S_extracted
```

3.3 Inverted-Index Translation

Goal: Turn a flat list of `(address, value)` pairs into a **lossless** dictionary `{ value → [address or address_range, …] }`, dropping empties and merging repeats.

🔷 Implementation Steps

1. **Gather Non-Empty Cells**

```
# Input: Se from §3.2
cells = [ (addr, cell.value)
          for addr, cell in Se.items()
          if cell.value not in ("", None) ]
```

2. **Build Value→Address Map**

```
from collections import defaultdict
index = defaultdict(list)
for addr, val in cells:
    index[val].append(addr)    # e.g. "A4", "B2", …
```

3. **Compress Addresses into Ranges**
   - **Sort** each list of addresses.
   - **Group** contiguous runs in the same row or column into compact ranges (e.g. `"C2","C3","C4" → "C2:C4"`; `"D2","E2","F2" → "D2:F2"`).
   - **Function Sketch:**

```
def compress_runs(addrs: List[str]) -> List[str]:
    # parse each addr into (row, col), cluster contiguous,
    # then re-serialize as "A1" or "A1:A3"
    …
for val, addr_list in index.items():
4.     index[val] = compress_runs(sorted(addr_list))
```

5. **Output as JSON**

```
python
CopyEdit
import json
T_inverted = json.dumps(index, indent=2)
```

  - **Result:** `{ "Atlantis": ["A2","A7","F5"], "QuantumMind": ["B2:B4","F4"], … }`

Effect: Boosts sheet-level compression ratio from ~ 4.4× to ~ 14.9× without losing any cell.

3.4 Data-Format-Aware Aggregation

Goal: Further collapse clusters of similarly typed or formatted cells—replacing raw values with their **data type** or **Number Format String (NFS)**.

🔷 Implementation Steps

1. **Extract NFSs**
   - Use a library (e.g. **OpenPyXL** or **ClosedXML**) to read the built-in Number Format String for each cell.
   - If NFS is missing, **infer** via rule-based matching on the cell's text:

     ```
     python
     CopyEdit
     RULES = {
         r"^\d{4}-\d{2}-\d{2}$": "Date",
         r"^\d+$":              "Integer",
         r"^\d+\.\d+$":         "Float",
         r"^\d+%$":             "Percentage",
         # … plus Scientific, Time, Currency, Email, etc.
     }
     def infer_type(text):
         for pattern, dtype in RULES.items():
             if re.match(pattern, text):
                 return dtype
         return "Others"
     ```

2. **Label Each Cell**

   ```
   python
   CopyEdit
   label_map = {}
   for addr, cell in Se.items():
       nfs = get_nfs(cell)            # library call, or None
       label = nfs or infer_type(cell.value)
       label_map.setdefault(label, []).append(addr)
   ```

3. **Cluster Adjacent Addresses**
   - As in §3.3, **sort** and **merge** `label_map[label]` into ranges, but now grouping only those addresses that form **rectangular blocks** of identical label.
   - This yields for example:

     ```
     json
     ```

```
CopyEdit
{
  "IntNum":     ["C2:C4","C6:C12"],
  "Percentage": ["H5:H6","H8:H9"],
  …
}
```

4. **Produce Final Aggregated JSON**

```
python
CopyEdit
T_aggregated = json.dumps(label_map, indent=2)
```

   - **Effect:** Lift compression ratio from ~ 14.9× up to ~ 24.8×—now only format/type clusters remain

3.5 Chain of Spreadsheet (CoS)

We wrap our compressor+LLM pipeline into a two-stage procedure:

Stage 1: Table Identification & Boundary Detection

1. **Inputs:**
   - `T_compressed` (the output of §3.4)
   - `user_query` (natural-language question)
2. **Prompt Template:**

```
css
CopyEdit
You are a spreadsheet-understanding assistant.
Given the following compressed sheet representation:
  {T_compressed}
And the query:
  "{user_query}"
Identify the single table that contains the information needed to
answer.
Output exactly:
  {
    "table_id": <TABLE_INDEX>,
    "bounds": { "top": r1, "left": c1, "bottom": r2, "right": c2 }
  }
```

3. **LLM Call & Parsing:**

```
python
CopyEdit
detection_resp = llm(prompt_detection)
result = json.loads(detection_resp)
r1,c1,r2,c2 = result["bounds"].values()
```

4. **Extract Sub-Table:**
   Map `(r1…r2, c1…c2)` back to your compressed or skeleton grid to get `Table_segment`.

Stage 2: Response Generation

1. **Prompt Template:**

```
css
CopyEdit
You are given a table:
  {serialize(Table_segment)}
And the question:
  "{user_query}"
Answer concisely, citing cell addresses or formulas where appropriate.
```

2. **LLM Call:**

```
python
CopyEdit
answer = llm(prompt_answer)
```

3. **Output:**
   Return `answer` to the user.

---

4 Experiments: Evaluation Setup

4.1 Spreadsheet Table Detection

- **Dataset:**
  - 188 real-world spreadsheets (311 tables) from Dong et al. 2019b, human-validated boundaries.
  - Split by token-count into Small/Medium/Large/Huge.
- **Metric:**
  - **Error-of-Boundary-0 (EoB-0):** exact match of top/left/bottom/right.
  - Report **F1** over all tables.
- **Baselines & Models:**
  - **Baseline:** TableSense-CNN (Dong et al. 2019b).
  - **LLMs:** GPT-4, GPT-3.5; open-source: Llama2, Llama3, Phi3, Mistral-v2.
  - Few-shot prompting or fine-tuning on detection prompts.

4.2 Spreadsheet QA

- **Dataset:**
  - 64 multi-table spreadsheets sampled from the corpus.

- 4–6 handcrafted questions each (search, compare, basic arithmetic), total **307** (Q, A, S) tuples.
- Answers as cell addresses or formulas.

- **Evaluation:**
  - **Exact match** on cell address or formula string.
  - Compare performance when using:
    1. Vanilla encoding + direct QA prompt
    2. Full CoS pipeline

4.2.2 Experiment Setup

1. **Baselines Chosen**
   - **TAPEX** and **Binder** (optimized for single-table QA).
2. **Multi-Table Adaptation**

```
python
CopyEdit
# Step A: Table Detection
table_bounds = detect_table_bounds(Se, question)  # from your
fine-tuned detection model

# Step B: Extract & Compress
table_segment = extract_subtable(Se, table_bounds)
compressed = apply_compression(table_segment,
               use_structural_anchor=True,
               use_inverted_index=True,
               use_format_aggregation=True)

# Step C: Format for Baseline
# e.g. for TAPEX: serialize as Markdown table; for Binder: JSON input
schema
baseline_input = format_for_baseline(compressed)
```

3. **Token-Limit Handling**

```
python
CopyEdit
max_tokens = baseline.token_limit()  # e.g. 4K tokens

if count_tokens(baseline_input) > max_tokens:
    # Strategy 1: Truncate less-important rows/cols from ends
    baseline_input = truncate_context(baseline_input, max_tokens)
    if count_tokens(baseline_input) > max_tokens:
        # Strategy 2: Split into overlapping chunks
        chunks = split_table(baseline_input, chunk_size=max_tokens,
                             overlap=header_rows)
        # feed each chunk separately and merge answers or pick the
highest-confidence
```

4. **Answer Evaluation**

  - Compare predicted cell address or formula string against ground-truth `(Q,A,S)` tuples.
  - **Accuracy** = % exact matches.
5. **Model Configuration**
  - Use **GPT-4** (instruct-tuned) for your CoS pipeline and for any fine-tuning.
  - Refer to Appendix G for hyperparameters (learning rate, batch size, prompt templates).

4.2.3 Experiment Procedure

1. **Detection → QA Loop**

```
python
CopyEdit
for (Q, A, S) in spreadsheet_QA_dataset:
    # 1. Table detection (Stage 1 of CoS)
    bounds = detect_table_bounds(S, Q)

    # 2. Extract, compress, and possibly split
    seg = extract_subtable(S, bounds)
    seg_c = apply_compression(seg, modules=[1,2,3])
    seg_chunks = split_if_needed(seg_c, max_tokens)

    # 3. For each chunk: run QA (Stage 2 of CoS)
    answers = [ run_baseline(chunk, Q) for chunk in seg_chunks ]
    pred = merge_or_select(answers)

    # 4. Record correctness
    log_result(Q, A, pred)
```

2. **Further Compression & Splitting**
  - If `modules=[1,2,3]` still exceeds GPT-4's token limit, deploy the **table-splitting algorithm** (see Appendix M.2):
    – Identify natural "sub-tables" via header detection.
    – Concatenate them into minimal overlapping segments.
3. **Logging & Metrics**
  - Track per-question: detection EoB-0, QA exact-match.
  - Aggregate overall and by spreadsheet size (Small/Medium/Large/Huge).

Table 1: Compression Ratios by Module

Modules	Total Tokens	Compression Ratio
None	1,548,577	1.00
1 (Structural Anchor)	350,946	4.41

Modules	Total Tokens	Compression Ratio
2 (Inverted Index)	580,912	2.67
3 (Format Aggregation)	213,890	7.24
1 + 2	103,880	14.91
1 + 3	96,365	16.07
2 + 3	211,445	7.32
1 + 2 + 3	62,469	**24.79**

Tip: Start with all three modules (`1+2+3`) for maximal compression; fall back to subsets only if you observe accuracy drops.

3.X Lightweight Heuristics for Structural-Anchor Proposal

Goal

Quickly propose rough table boundaries by purely heuristic rules—so that your LLM can refine them later.

Step 1: Identify High-Discrepancy Rows & Columns

1. **Compute Discrepancy Features** for each row `i` and column `j`:
   - **Value Changes:** count how many adjacent cells differ in text or numeric value.
   - **Merged-Cell Presence:** count merged cells in that row/column.
   - **Border Usage:** count cells with non-default borders.
   - **Fill Color:** count cells whose background color ≠ default.
   - **Font Emphasis:** count bold/italic cells.
2. **Score & Threshold**:

   ```
   python
   CopyEdit
   row_score[i] = w1*Δvalue + w2*merges + w3*borders + w4*colors +
   w5*fonts
   anchors_rows = {i for i in all_rows if row_score[i] ≥ ROW_THRESH}
   # similarly for columns → anchors_cols
   ```

   - Choose weights `w1…w5` and `ROW_THRESH` via small grid search on a dev set.

Step 2: Enumerate Candidate Boundaries

Form every rectangle `(r_top, r_bot, c_left, c_right)` by pairing:

```
python
```

```
CopyEdit
candidates = []
for r1 in anchors_rows:
  for r2 in anchors_rows:
    if r2 <= r1: continue
    for c1 in anchors_cols:
      for c2 in anchors_cols:
        if c2 <= c1: continue
        candidates.append((r1, r2, c1, c2))
```

Step 3: Filter Unreasonable Candidates

For each `(r1,r2,c1,c2)`:

1. **Size Check**: require `(r2−r1+1) ≥ MIN_ROWS` and `(c2−c1+1) ≥ MIN_COLS`.
2. **Density Check**:

   ```
   python
   CopyEdit
   area = (r2-r1+1)*(c2-c1+1)
   non_empty = count_nonempty_cells(r1, r2, c1, c2)
   sparsity = non_empty / area
   ```

   – Reject if `sparsity < SPARSITY_MIN` or `> SPARSITY_MAX`.

3. **Header-Row/Col Evidence**:
   - Check top row `r1` has high text-ratio (≥ HEADER_TEXT_RATIO).
   - Check left col `c1` has ≥ HEADER_LABEL_RATIO of non-numeric values. – Reject if neither edge looks header-like.

Step 4: Resolve Overlapping Candidates

1. **Detect Overlaps:**
   Two boundaries overlap if their intersection area is non-empty.
2. **Pairwise Pruning:**
   For each overlapping pair `A` and `B`:
   - If `|A.r1 − B.r1| ≤ DELTA_R` (close top edges), compare:
     - **Header Score** = text_ratio(top row) + format_ratio(year/date patterns).
   - **Keep** the one with higher header score; **discard** the other.
3. **Repeat** until no overlaps remain.

Step 5: Expand to Structural Anchors

From the final set of boundaries, collect:

```
python
CopyEdit
raw_anchor_rows = {r1, r2 for each boundary}
raw_anchor_cols = {c1, c2 for each boundary}
```

Then **include neighbors within `k` units**:

```
python
CopyEdit
k = 4  # preserves >97% true boundary lines
anchors_rows = ⋃_{p ∈ raw_anchor_rows} {i | |i-p| ≤ k}
anchors_cols = ⋃_{q ∈ raw_anchor_cols} {j | |j-q| ≤ k}
```

These expanded sets become your structural anchors `A = {rp, cq}` for §3.2.

Implementation Tips

- **Parameter Tuning:**
  – Calibrate `ROW_THRESH`, `MIN_ROWS`, `SPARSITY_MIN`, etc. on a small validation set.
- **Efficiency:**
  – Precompute per-row/col scores once.
  – Discard obviously tiny candidates early to avoid combinatorial explosion.
- **Robustness:**
  – If boundaries are sparse, falling back to a higher `k` ensures you don't lose headers/notes.

F Spreadsheet Table Detection Test Dataset Partition

1. Encode Each Sheet in Markdown-Like Style (per §3.1)

- **Text Lines:**
  For every non-empty cell `(i,j)` produce

  ```
  less
  CopyEdit
  |{col_letter(j)}{i},{cell.value}|
  ```

- **Format Lines:**
  In parallel, for each cell output

  ```
  less
  CopyEdit
  |{col_letter(j)}{i},{comma-joined list of format attributes}|
  ```

- **Concatenate:**
  Join all text lines into one big string, and likewise for format lines (or interleave them, depending on your tokenizer setup).

---

2. Count Tokens

- **Choose Tokenizer:**
  Use the same tokenizer your detection model (e.g. GPT-4) expects—for instance, OpenAI's `tiktoken`.
- **Compute:**

  ```
  python
  CopyEdit
  import tiktoken
  enc = tiktoken.get_encoding("cl100k_base")
  tokens = enc.encode(full_markdown_string)
  token_count = len(tokens)
  ```

---

3. Assign Size Buckets

Bucket	Token Range
Small	`< 4 000`
Medium	`4 000 ≤ tokens < 8 000`
Large	`8 000 ≤ tokens < 32 000`
Huge	`tokens ≥ 32 000`

```
python
CopyEdit
if token_count < 4_000:
    bucket = "Small"
elif token_count < 8_000:
    bucket = "Medium"
elif token_count < 32_000:
    bucket = "Large"
else:
    bucket = "Huge"
```

- **Store** this `bucket` label alongside each sheet in your test metadata.

---

4. Example Encoding Snippet

Text Input

```
less
```

```
CopyEdit
|B2,Table 4: Diesel-driven passenger cars, 2015|
|C2,|  |D2,|  |E2,|  |F2,|  |G2,|  |H2,|
|B3,|  |C3,|  |D3,|  |E3,|  |F3,|  |G3,|  |H3,|
|B4,|  |C4,|  |D4,|  |E4,|  |F4,|  |G4,|  |H4,|
|B5,|  |C5,Diesel engine|D5,|  |E5,|  |F5,Share of all passenger cars
(%)|G5,|  |H5,|
…
```

Format Input

```
mathematica
CopyEdit
|B2,Font Bold|C2,|D2,|E2,|F2,|G2,|H2,|
|B3,|C3,|D3,|E3,|F3,|G3,|H3,|
|B4,Bottom Border|C4,Bottom Border|D4,Bottom Border|E4,Bottom
Border|F4,Bottom Border|G4,Bottom Border|H4,Bottom Border|
|B5,Top Border,Right Border,Fill Color,Font Bold|
 C5,Top Border,Bottom Border,Left Border,Fill Color,Font Bold|
 D5,Top Border,Bottom Border,Fill Color,Font Bold|
 E5,Top Border,Bottom Border,Right Border,Fill Color,Font Bold|
 F5,Top Border,Bottom Border,Left Border,Fill Color,Font Bold|
 G5,Top Border,Bottom Border,Fill Color,Font Bold|
 H5,Top Border,Bottom Border,Fill Color,Font Bold|
```

1. Example Spreadsheet QA Data Item

Each QA example is a tuple `(question, ground_truth, prompt)`:

```
python
CopyEdit
qa_item = {
    "question": "What were the highest temperatures in Washington DC in
1998?",
    "ground_truth": ["X23", "X24"],
    "prompt": build_prompt(
        instruction=YOUR_INSTRUCTION_STRING,
        sheet_encoding=T_aggregated  # the fully compressed JSON/Markdown
    )
}
```

- **`question`**: the user's natural-language query.
- **`ground_truth`**: list of cell addresses or formula(s) expected.
- **`prompt`**: your “Instruction + Encoded Spreadsheet” string fed into the LLM.

---

2. Cost Calculation

Track token usage and API cost like this:

```
python
CopyEdit
# Constants (per-1K-token prices)
PRICE_GPT35 = 0.0005    # USD per 1K tokens
PRICE_GPT4   = 0.03

# Measured average compressed tokens per sheet
avg_tokens = 62_000 / 198  # ~313 tokens/sheet

# Cost per task
cost_gpt35 = avg_tokens * PRICE_GPT35 / 1000
cost_gpt4   = avg_tokens * PRICE_GPT4   / 1000

print(f"GPT-3.5 turbo cost: ${cost_gpt35:.6f}")  # ~$0.000157
print(f"GPT-4            cost: ${cost_gpt4:.6f}")  # ~$0.00939
```

- **Baseline (vanilla)** uses ~1,548,000 tokens → ~$0.00391 (GPT-3.5) / $0.235 (GPT-4).
- **Compressed** uses ~62,000 tokens → saves ~96% in cost.

---

3. Qualitative Comparison with TableSense-CNN

Example from **Figure 10**:

```
python
CopyEdit
baseline_regions  = [("A1","G44"), ("K5","M14"), ("K16","M38"),
("Q20","W29")]
sllm_regions      = [("A1","G44"), ("K5","R14"),  # extended K5→R14
                     ("K16","M38"), ("Q20","W29")]

# Compute what SLLM added:
added = set(sllm_regions) - set(baseline_regions)
# added == {("K5","R14")}
```

- **Insight:** SLLM correctly includes column **R5:R14**, which TableSense-CNN missed because it's semantically linked (percentages of earlier columns) but spatially distant.

L.1 Vanilla Prompt Template for Table Detection

```
text
CopyEdit
INSTRUCTION:
You are given a serialized spreadsheet as a single line of cell tuples in
row-major order.
Each tuple is formatted as `<Address>,<Value>`, and tuples are separated by
`|`.
Empty cells appear as `<Address>,`.
Identify every table in this sheet (header + data rows only; exclude titles
or comments).
Return a JSON array of range strings, e.g.:
```

```
  ["A2:D5", "K5:M14"]

Do **not** output any other text or explanation.

INPUT:
[Encoded Spreadsheet]
```

L.2 Compressed-Input Prompt Template for Table Detection

```
text
CopyEdit
INSTRUCTION:
You are given a compressed spreadsheet encoding as a JSON object mapping each
cell text or format label to one or more addresses/ranges, e.g.:

  {
    "Year": ["A1"],
    "IntNum": ["B2:B10"],
    "#,##0": ["C2:C10"],
    …
  }

Detect all tables (header + data) in the sheet.
Return a JSON array of range strings, e.g.:

  ["A1:F9", "K5:R14"]

Do **not** include titles or comments, and do **not** add any other text.

INPUT:
[Compressed JSON]
```

L.3 Prompt Templates for Spreadsheet QA (CoS)

Stage 1 – Table Identification

```
text
CopyEdit
INSTRUCTION:
You are given:
  • A compressed spreadsheet (JSON as above).
  • A question about this sheet.

Determine which single table contains the answer.
Return exactly one JSON object:

  {"range": "A1:F9"}

Do **not** add any other text or explanation.

INPUT:
```

```
Spreadsheet: [Compressed JSON]
Question: "<Your Question>"
```

Stage 2 – Answer Generation

```
text
CopyEdit
INSTRUCTION:
You are given:
  • A single table (Markdown-style or JSON).
  • A question whose answer lies within this table.

Locate the answer and return exactly one JSON object:

  {"answer": "<CellAddress or Formula>"}

Examples: `{"answer":"B3"}`, `{"answer":"SUM(A2:A10)"}`.
Do **not** add any other text or explanation.

INPUT:
Table: [Table Representation]
Question: "<Your Question>"
```

Tips for Use in Your Guide:

- Embed these templates verbatim in your code or README.
- Fill in `[Encoded Spreadsheet]`, `[Compressed JSON]`, and `<Your Question>` programmatically.
- Always parse the LLM's output as strict JSON to avoid downstream errors.

M.1 Identical Cell Aggregation (Algorithm 1)

Input:

- `nfs[m][n]` — 2D matrix of Number Format Strings or inferred type labels for each cell.

Output:

- `areas` — list of tuples `((r1,c1),(r2,c2), val_type)` denoting each rectangular block of identical type.

```
pseudo
CopyEdit
1. Let m, n = dimensions of nfs
2. visited ← 2D array [m][n], all False
3. areas ← []
4. FormatDict ← mapping from NFS string → canonical val_type

5. define dfs(r, c, val_type):
6.   if visited[r][c] OR val_type != FormatDict[nfs[r][c]]:
```

```
    return bounds (r,c,r,c)  # single-cell block
  visited[r][c] ← True
  bounds ← (r, c, r, c)
  for each neighbor (tr, tc) in {up, down, left, right}:
    if not visited[tr][tc] AND val_type == FormatDict[nfs[tr][tc]]:
      sub_bounds = dfs(tr, tc, val_type)
      expand bounds to include sub_bounds
  return bounds

for r in 0 … m−1:
  for c in 0 … n−1:
    if not visited[r][c]:
      val_type = FormatDict[nfs[r][c]]
      (r1,c1,r2,c2) = dfs(r, c, val_type)
      areas.append(((r1,c1),(r2,c2), val_type))

return areas
```

M.2 Table-Split QA Algorithm (Algorithm 2)

Input:

- `question` — string
- `region` — 2D table (list of rows), already compressed or extracted

Output:

- `answers` — list of answer strings (cell address or formula)

```
pseudo
CopyEdit
headers ← []
answers ← []

if calculateTokens(region) ≤ 4096:
  return [ answer_question(question, region) ]

else:
  headers = predict_header(region)
  body = region[ len(headers) : ]  # rows below header

  for i in 0 … len(body)−1:
    # build a small sub-table: header + 3 rows at a time
    sub_table = headers + body[i : i+3]
    answer = answer_question(question, sub_table)
    answers.append(answer)

return answers
```

- **`calculateTokens`**: counts tokens in your chosen encoding.
- **`predict_header`**: identifies header rows (e.g. via heuristic or model).
- **`answer_question`**: runs your Stage 2 QA LLM prompt on the given sub-table.