QUBER — ADVANCED TABLE EXTRACTION

The Set-of-Mark Design

Vision-Guided Table Extraction from Document Ingestion through Final Table Generation

A technical reference describing how Quber locates, extracts, corrects, and grounds financial tables — the marking, image-examination, and error-correction steps end to end — with the research lineage and project history that produced it.

Engineering Reference · Generated 14 June 2026
Source of truth: `src/quber/` · Jira project QUE

Contents

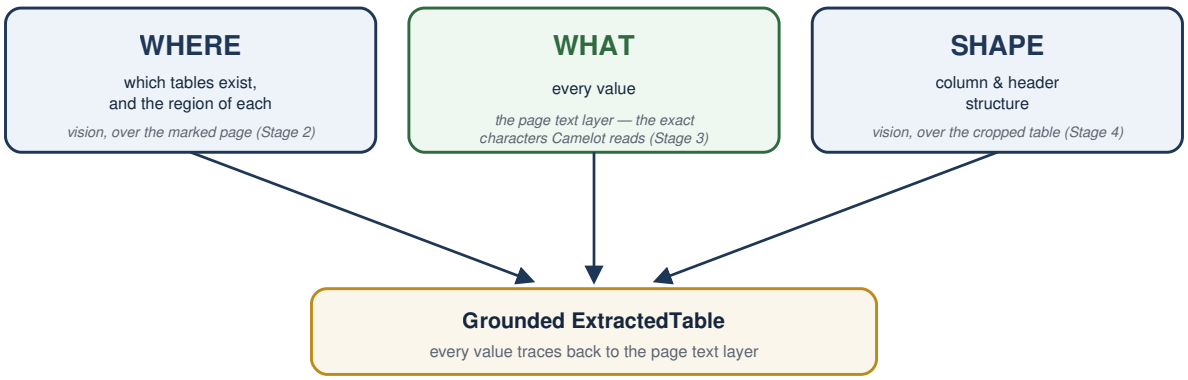
1 Introduction & Motivation

Quber extracts tables from PDF documents — principally dense financial filings such as SEC exhibits and investor fact-books — and renders them as faithful, machine-readable Markdown. The hard part is not parsing a clean table; it is reliably answering two questions on a crowded page: *where is each table*, and *what exactly does it contain*, with a guarantee that no number was invented and no row silently dropped.

The **Set-of-Mark** design is Quber's answer. It splits the job across three components, each trusted only for what it does well:

Component	Responsible for	Why it is trusted for this
The vision model, over a marked image	Table *identity* and *region*	A multi-modal model reads *printed labels* reliably; it cannot be trusted to emit accurate continuous coordinates.
Camelot, over the PDF text layer	The *values* inside a region	Camelot reads the document's real characters, not pixels — it never misreads a digit and never loses an interior row.
The vision model, over a cropped image	Table *structure* (columns, headers)	The picture is the ground truth for how columns split and how headers nest; the LLM re-arranges values to match it but may never read a value off it.

One job each — vision places and shapes each table; the text layer holds every value



*Blue is judged by the vision model; the two blue steps do different jobs — **where** vs **shape** — and neither ever reads a value. Green is the document's own characters — the page text layer that Camelot parses — the single source of every number.*

The phrase that recurs throughout the code captures the whole philosophy:

> *Vision owns identity and region; Camelot owns the values inside that region; the LLM only cleans structure under the no-number guard.*
>
> — `src/quber/core/extractors/set_of_mark/pipeline.py`

This document follows a single page from ingestion to final Markdown, pausing at every step where an IMAGE IS EXAMINED and at every step where an ERROR IS CAUGHT AND CORRECTED. It closes with the research lineage behind the technique and the Jira history (epics, stories, spikes, and the open defects) that produced it.

2 Research Foundations: The Two Cited Papers

The codebase cites exactly two papers, both in the module header of the grid locator (`src/quber/agents/grid_locator.py`, lines 16–27). They form a single line of work on *visual prompting*: instead of asking a model to predict where something is, you *mark* the image and ask the model to read your marks.

PAPER 1

Set-of-Mark Prompting Unleashes Extraordinary Visual Grounding in GPT-4V

Jianwei Yang, Hao Zhang, Feng Li, Xueyan Zou, Chunyuan Li, Jianfeng Gao · *arXiv: 2310.11441* (2023).

Cited at `grid_locator.py:18–21`.

The paper overlays labelled marks (numbers, boxes, masks) on an image so the model answers a spatial question by *reading the marks* rather than predicting raw coordinates. This converts a hard regression problem ("give me the pixel box") into an easy reading problem ("which marks does it cover?"), dramatically improving visual grounding.

PAPER 2

Scaffolding Coordinates to Promote Vision-Language Coordination in Large Multi-Modal Models

Xuanyu Lei, Jianwei Yang, et al. · *arXiv:2402.12058* (2024).

Cited at `grid_locator.py:22–24`.

A direct extension: overlay a *labelled coordinate grid* on the image as positional anchors the model reads. The scaffold gives the model a discrete, legible frame of reference, tightening the link between what it sees and the coordinates it reports.

2.3 How the papers map onto Quber

Quber applies this line of work almost literally. The module docstring states the connection in the authors' own terms:

```
This follows the Set-of-Mark / coordinate-scaffold line of
visual prompting:
```

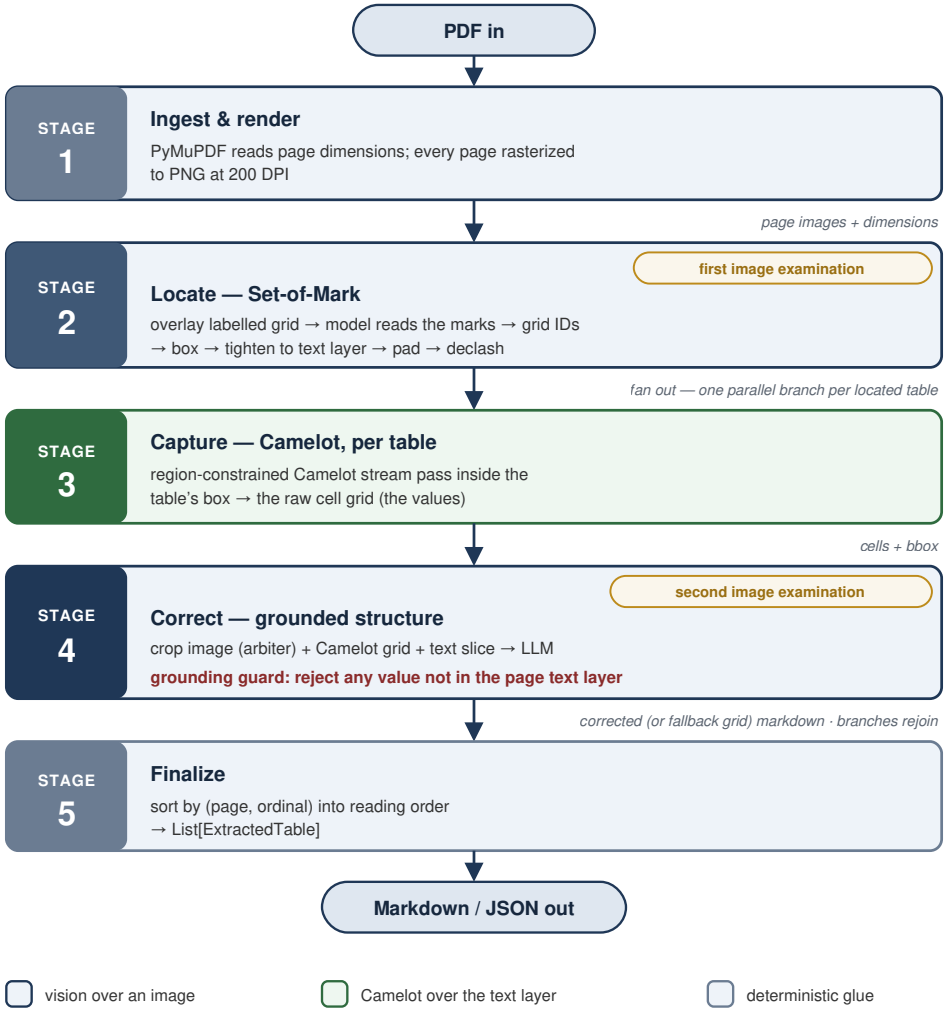
```
  - Yang, Zhang, Li, Zou, Li, Gao, "Set-of-Mark Prompting Un-
leashes
    Extraordinary Visual Grounding in GPT-4V," arXiv:2310.11441
(2023):
    overlay labelled marks on the image so the model answers by
reading the
    marks rather than predicting continuous coordinates.
  - Lei, Yang, et al., "Scaffolding Coordinates to Promote Vis-
ion-Language
    Coordination in Large Multi-Modal Models," arXiv:2402.12058
(2024):
    overlay a labelled coordinate grid as positional anchors
the model reads.

Overlaying a labelled row/column grid and reading back grid IDs
in place of
continuous coordinates is the direct application of that work
here.
```

The problem the papers solve is exactly the failure Quber measured first-hand. A vision model asked for a continuous bounding box “emits a discretized, often evenly-spaced grid that walks off the actual tables on repetitive / stacked layouts” — precisely the stacked, near-identical tables common in financial supplements. The marked grid sidesteps that failure mode by never asking for a coordinate at all.

3 The Pipeline at a Glance

Set-of-Mark extraction is implemented as a `pydantic-graph` with three steps. Page location fans out into one parallel branch per detected table; the branches rejoin and sort into reading order. The orchestrator lives in `core/extractors/set_of_mark/orchestrator.py`; the graph in `set_of_mark/pipeline.py`.



There are **two distinct image-examination steps** and **several layered error-correction steps**. Stage 2 examines the *whole page* (with a grid) to find tables; Stage 4 examines a *cropped table* to fix its structure. The grounding guard (Stage 4) and the completeness layer (§8) are the error-correction mechanisms. Each is detailed below.

PIPELINE · INGEST & RENDER

4 Document Ingestion & Page Rendering

The entry point is `SetOfMarkExtractor.extract_tables(source)` (`set_of_mark/orchestrator.py`). Two artifacts are produced from the PDF up front and carried for the rest of the run on `SetOfMarkDeps`:

- **Page dimensions** — PyMuPDF (`fitz.open()`) supplies each page's width and height in PDF points. These are needed to convert normalized boxes into the point frames Camelot and the text-layer reader expect.
- **Page raster images** — every page is rendered to a PNG at 200 DPI (the configurable default) via `render_pages()`, named `page-0001.png`, … in a temporary directory. 200 DPI is tuned so the default 36×12 grid resolves to roughly 22 pt per row and 51 pt per column on US-Letter.

Pixels and text are kept separate from the very first step. The rendered PNG is used only for what vision does — locating tables and judging structure. The text layer, read later through PyMuPDF and by Camelot, is the only source of character-accurate values. The two never cross.

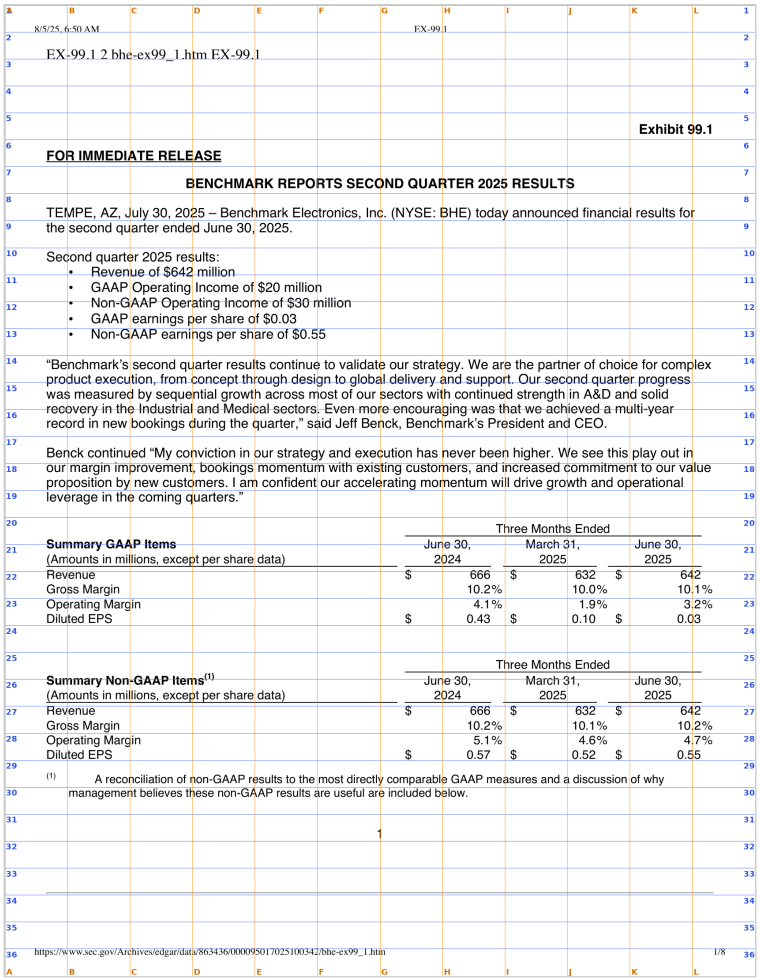
5 Set-of-Mark Localization

Localization is the step the design is named for. The owner is `PydanticAIGridLocator` (`src/quber/agents/grid_locator.py`), invoked once per page inside the `locate_tables` graph step (page-concurrent, bounded by a semaphore).

5.1 Overlaying the labelled grid

`overlay_grid()` draws a numbered/lettered grid onto the rendered page: by default 36 rows (numbered 1…36, printed on both side margins) and 12 columns (lettered A…L, printed on the top and bottom margins), with thin gridlines — blue for rows, orange for columns. Labels are repeated on opposite margins so a label is always near whatever the model is looking at.

```
# grid_locator.py — overlay_grid(): rows labelled on both side
margins
for r in range(rows + 1):
    y = r * rstep
    draw.line([(0, y), (img.width, y)], fill=(80, 120, 255),
width=1)
    if r < rows:
        draw.text((2, y + 2), str(r + 1), fill=(40, 80, 230),
font=font)
        draw.text((img.width - 26, y + 2), str(r + 1),
fill=(40, 80, 230), font=font)
# columns labelled on top and bottom margins (A..L)
for ci in range(cols + 1):
    x = ci * cstep
    draw.line([(x, 0), (x, img.height)], fill=(255, 140, 0),
width=1)
    if ci < cols:
        draw.text((x + 3, 2), labels[ci], fill=(210, 110, 0),
font=font)
        draw.text((x + 3, img.height - 20), labels[ci],
fill=(210, 110, 0), font=font)
```



The actual marks the model sees. *A real Quber render: page 1 of a Benchmark Electronics filing under* `overlay_grid()` *— rows 1–36 numbered on both side margins, columns A–L lettered top and bottom. The model answers "which rows/columns does each table cover?" by reading these printed labels, not by predicting coordinates.*

5.2 What the model sees and answers

The gridded PNG is sent to the model with a system prompt (`build_prompt()`) that defines what a table is, insists on bounding the whole table anatomy, and — critically — tells the model to read the printed labels rather than estimate coordinates:

```
WHAT COUNTS AS A TABLE: a block where values are arranged in a
GRID — the same
two or more columns line up vertically across multiple rows.

Bound the WHOLE table, including all of its parts:
 - TITLE / CAPTION ... and any units line directly above the
columns.
 - COLUMN HEADERS: every header row, including multi-level /
```

```
spanning headers.
 - ROW LABELS (the stub column): the left-hand column of row
names.
 - BODY: every data row, including subtotal and total rows.
 - FOOTNOTES tied to the table by a marker ("(1)", "(2)") just
below it.

For each table, in reading order, report:
 - ordinal (1 = first)
 - title  - row_start / row_end (printed row numbers)
 - col_start / col_end (printed column LETTERS, INCLUDING the
row labels)

Read the printed labels off the grid — do not estimate coordin-
ates.
```

The model returns a structured `GridFlagResult` — a list of tables, each an *ordinal*, a *title*, and a discrete *grid-cell range*. No floats. Because the answer is a set of marks, the model is doing the thing it is reliably good at (reading printed text), and the burden of turning marks into geometry falls on Quber's own deterministic code.

> **WHY THIS IS ROBUST**
>
> The feared failure — a non-deterministic table drop on a stacked layout — did not reproduce under this scheme. Grid discretization plus a low-temperature model gives a **table-count stability of 1.00** page-to-page in the validating spike, against a raw continuous-coordinate detector that drifted, produced false positives, and flipped table counts (macro IoU 0.42 vs 0.70 for the anatomy-prompted grid).

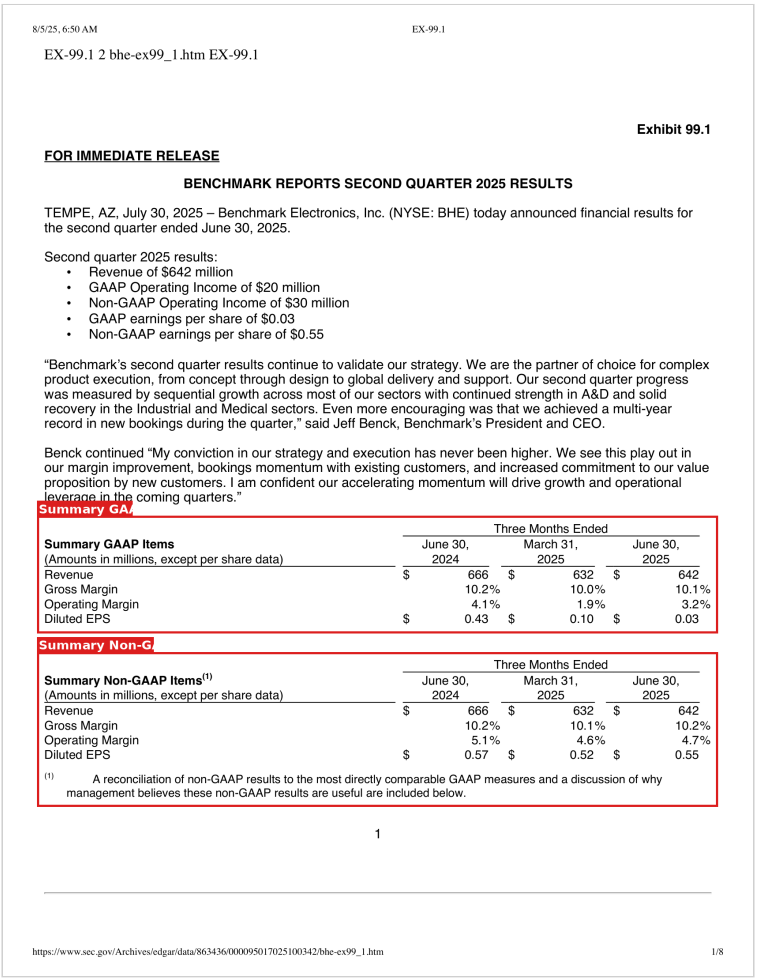
5.3 From grid IDs to a precise box: tighten, pad, declash

The coarse grid range is then refined by a deterministic geometry chain. This is the first place the system *corrects* for the imprecision of a coarse mark:

Step	Function	What it does
1. Grid → box	`grid_region_norm()`	Maps the 1-based row range and column letters to a normalized 0–1 box. The grid is ours, so the mapping is exact.
2. Tighten	`tighten_region()`	Snaps the coarse cell span to the page's actual *text layer*, pulling the box in to the real word

Step	Function	What it does
		boundaries. If a region has *no* text (an image-only chart), the coarse region is kept — a region is never dropped.
3. Pad	`pad_boxes()` (`BOX_PAD_PTS = 6.0`)	Grows each box by a small fixed margin so a downstream crop cannot clip an edge.
4. Declash	`declash_stacked()`	Pulls the bottom of a stacked table off the top of the next, so no box contains a neighbour's content — the essential guard for back-to-back tables.

The spike settled on **declash + 6 pt pad** as the finishing combination. Declash is essential to stop a stacked-table box bleeding into a neighbour's title. A larger 12 pt pad crowded prose. Pure text-snapping alone was fragile. The output is a `LocatedTable` carrying the ordinal, title, normalized region, grid extents, and a `tightened` flag.



8/5/25, 6:50 AM EX-99.1

EX-99.1 2 bhe-ex99_1.htm EX-99.1

Exhibit 99.1

FOR IMMEDIATE RELEASE

BENCHMARK REPORTS SECOND QUARTER 2025 RESULTS

TEMPE, AZ, July 30, 2025 – Benchmark Electronics, Inc. (NYSE: BHE) today announced financial results for the second quarter ended June 30, 2025.

Second quarter 2025 results:

- Revenue of $642 million
- GAAP Operating Income of $20 million
- Non-GAAP Operating Income of $30 million
- GAAP earnings per share of $0.03
- Non-GAAP earnings per share of $0.55

"Benchmark's second quarter results continue to validate our strategy. We are the partner of choice for complex product execution, from concept through design to global delivery and support. Our second quarter progress was measured by sequential growth across most of our sectors with continued strength in A&D and solid recovery in the Industrial and Medical sectors. Even more encouraging was that we achieved a multi-year record in new bookings during the quarter," said Jeff Benck, Benchmark's President and CEO.

Benck continued "My conviction in our strategy and execution has never been higher. We see this play out in our margin improvement, bookings momentum with existing customers, and increased commitment to our value proposition by new customers. I am confident our accelerating momentum will drive growth and operational leverage in the coming quarters."

Summary GA

	Three Months Ended		
Summary GAAP Items (Amounts in millions, except per share data)	June 30, 2024	March 31, 2025	June 30, 2025
Revenue	$ 666	$ 632	$ 642
Gross Margin	10.2%	10.0%	10.1%
Operating Margin	4.1%	1.9%	3.2%
Diluted EPS	$ 0.43	$ 0.10	$ 0.03

Summary Non-G

	Three Months Ended		
Summary Non-GAAP Items[1] (Amounts in millions, except per share data)	June 30, 2024	March 31, 2025	June 30, 2025
Revenue	$ 666	$ 632	$ 642
Gross Margin	10.2%	10.1%	10.2%
Operating Margin	5.1%	4.6%	4.7%
Diluted EPS	$ 0.57	$ 0.52	$ 0.55

(1) A reconciliation of non-GAAP results to the most directly comparable GAAP measures and a discussion of why management believes these non-GAAP results are useful are included below.

1

https://www.sec.gov/Archives/edgar/data/863436/000095017025100342/bhe-ex99_1.htm 1/8

The boundary boxes that come out of Stage 2. *The same page after grid IDs were converted, tightened to the text layer, padded, and de-clashed. Each red box bounds a whole table anatomy — title, multi-row header, body, and footnote — and the two stacked tables are cleanly separated. These regions are what constrain Camelot in Stage 3.*

PIPELINE · CAPTURE

6 Region-Constrained Camelot Extraction

Each located table fans out into its own parallel branch (`capture_table`). The branch first converts the normalized region into Camelot's `table_areas` string (`norm_bbox_to_table_area()`) and runs a **single, region-constrained Camelot stream pass** inside just that box (`camelot_targeted()`):

```
# recovery.py — one region-bounded Camelot pass, not a whole-page scan
def camelot_targeted(source_str, page, area, ordinal):
    tables = camelot.read_pdf(source_str, pages=str(page),
                              flavor="stream",
table_areas=[area])
    if not tables:
        return None
    ...  # returns a CamelotCandidate: flavor, page, bbox, accuracy, cells, markdown
```

Constraining Camelot to the vision-identified box is what inverts the older pipeline: vision has already decided *how many* tables there are and *where* each one is, so Camelot only has to read cells within a known boundary. The result is a `CamelotCandidate` whose `cells` field — a grid of strings read from the PDF's text layer — is the **authoritative source of every value** downstream.

> **IF THE REGION YIELDS NOTHING**
>
> When `camelot_targeted` returns `None` (a region with no extractable grid — e.g. an image-only chart), the branch does not discard the table. It emits an `ExtractedTable` with the visual identity intact (title and `som_region`) and an empty body. A vision-found table is never silently dropped — the standing financial-liability rule applied in code.

PIPELINE · CORRECT · SECOND IMAGE EXAMINATION

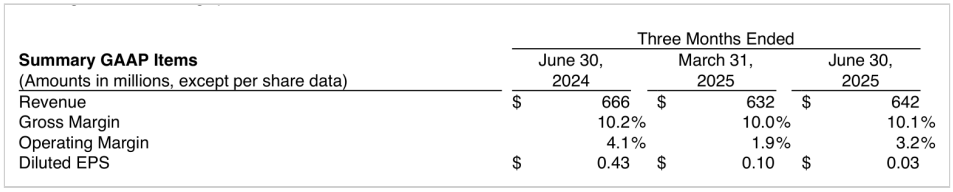
7 Grounded Structure Correction

Camelot reads values correctly but often gets the *structure* wrong: it splits a currency symbol into its own column, fragments a multi-row header, or leaves a phantom empty column from stream over-segmentation. The second image-examination step fixes structure — and only structure. The owner is `correct_structure()` (`core/extractors/camelot/correspondence/correction.py`).

7.1 The cropped image as structural arbiter

Three artifacts, all scoped to this one table, are assembled:

1. **A cropped image of just this table** — the *arbiter* of column and header structure. The crop reaches `CAPTION_PAD_PTS = 28` points *above* the data box to capture the title/caption that sits just over the grid. Only the *image* is widened; the value text slice stays tight to the data box, so a taller crop cannot leak a neighbour's numbers.



Summary GAAP Items (Amounts in millions, except per share data)	Three Months Ended June 30, 2024	March 31, 2025	June 30, 2025
Revenue	$ 666	$ 632	$ 642
Gross Margin	10.2%	10.0%	10.1%
Operating Margin	4.1%	1.9%	3.2%
Diluted EPS	$ 0.43	$ 0.10	$ 0.03

The arbiter image for one table *("Summary GAAP Items") — the exact crop handed to the model, reaching 28 pt above the data box to capture the caption. The LLM reads the shape here (one spanning header "Three Months Ended" over three period columns); it is forbidden from reading any number off it.*

1. **The Camelot Markdown** — correct values, possibly wrong column structure.
2. **The table's text-layer slice** (`region_text_in_bbox`) — the authoritative source for any value, including rows Camelot may have truncated.

Everything is scoped to the table's bbox — crop *and* region text — so an adjacent table can never bleed in. Region-scoping prevents a specific failure seen on Visa page 1, where an adjacent table's "Dec/Sep" values overwrote this table's (see §11).

7.2 The VET_STRUCTURE contract

The system prompt (`VET_STRUCTURE_PROMPT` in `agents/llm_client.py`) names the image the arbiter of structure and the text the source of values, enumerates the structural errors to fix, and lays down a non-negotiable constraint on values:

```
You vet and fix the STRUCTURE of one table ... using a cropped
image of that
table as the arbiter of structure and the provided text as the
source of values.

Expected structural errors to fix:
 - SYMBOL SPLIT INTO ITS OWN COLUMN: a currency symbol or sign
Camelot placed
   in a separate column belongs WITH the number beside it. Re-
join them.
 - MULTI-LEVEL / SPANNING COLUMN HEADER: flatten into ONE head-
er row in which
   every column's label is its full top-to-bottom path.
 - INTERLEAVED EMPTY / SPACER COLUMNS: drop them, bring real
columns together.
 - HEADER FRAGMENTED ACROSS ROWS: merge the fragments.

Hard constraints — non-negotiable:
 - Every value in `markdown` must come from the Camelot mark-
down or the text
   layer. NEVER read a number off the image, and NEVER invent,
compute, or
   estimate one.
 - Preserve every Camelot value; only move it to the correct
column.
```

This is the heart of the "structure ← image, values ← text" split. The model looks at the picture to decide *shape*; it is forbidden from looking at the picture to read a *number*. If the image shows rows Camelot missed, the model recovers them from the text layer, not from the pixels.

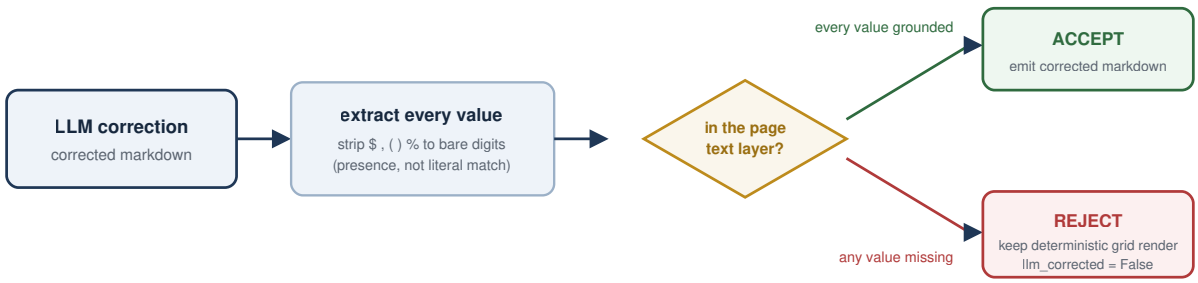
7.3 The grounding guard & presence grounding

A prompt instruction is not a guarantee. The code enforces the no-invented-number rule deterministically *after* the model responds. This **grounding guard** is the single most important error-correction step in the system:

```
# correction.py — every number in the corrected table must
already exist
# in Camelot's grid OR the page text layer. Anything else is
```

```
rejected.
corrected = correction.markdown
allowed = numeric_keys(grid_md) | numeric_keys(" ".join(w[4]
for w in words))
ungrounded = numeric_keys(corrected) - allowed
if ungrounded:
    logger.warning(
        "page {}: structure correction REJECTED (ungrounded
values {}); "
        "keeping grid markdown", page, sorted(ungrounded))
    return None   # fall back to the deterministic grid render
```

If the corrected table contains any figure that is not in the page text layer — the exact characters Camelot reads — the entire correction is thrown away and the deterministic grid render is emitted instead (`llm_corrected = False`). A hallucinated number can never reach the output. (Camelot's extracted grid is a subset of that same text layer; the guard unions both, but the text layer is the underlying source.)



The grounding guard as control flow: a single ungrounded figure rejects the entire correction.

PRESENCE GROUNDING, NOT LITERAL MATCH

The comparison is by *presence* of the figure, not its exact rendering. `numeric_keys()` strips currency, percent, parentheses, commas and spacing, so `$8,273.04`, `8273.04` and `8,273.04%` all key to the same `8273.04`. This is deliberate: it lets the model legitimately move a currency symbol out of its own column or fold a percent back into a cell without tripping the guard. A value is rejected only when the *figure itself* is absent from the sources — never merely because a symbol moved.

An earlier design used an *exact-literal* drift guard (compare corrected cells against Camelot cell-for-cell). A validation sweep proved that guard *wrong*: on Visa Q1FY25 p4 the model correctly recovered a truncated row (`240,083`) that was present in the page text layer but absent from Camelot's grid — an exact-vs-Camelot check would have rejected a correct fix. Presence grounding against the page text layer — the exact characters Camelot reads — is the replacement.

7.4 Worked examples: what Camelot gets wrong, and what the correction fixes

The two figures below are real output captured from running the live extractor on a Benchmark Electronics filing. Each shows three things side by side: the **source** table as it appears in the PDF, the **raw Camelot grid** the region-constrained pass produces, and the **corrected Markdown** after the image-arbiter pass — with the grounding guard confirming every figure already existed in the page text layer (what Camelot reads). Notice what Camelot gets wrong, and what the correction fixes:

Phantom empty columns from stream over-segmentation

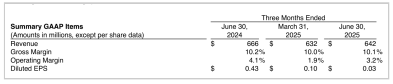
Currency "$" split into its own column

Multi-row header ("Three Months Ended" / "June 30," / "2024") fragmented

Title broken mid-word ("C↵ash Conversion Cycle")

Figure — Structure correction: Summary GAAP Items (p1 · raw grid 7×5 · llm_corrected=True)

1 · SOURCE (THE PDF TABLE)



Summary GAAP Items (Amounts in millions, except per share data)	Three Months Ended June 30, 2024	March 31, 2025	June 30, 2025
Revenue	$ 666	$ 632	$ 642
Gross Margin	10.2%	10.0%	10.1%
Operating Margin	4.1%	1.9%	3.2%
Diluted EPS	$ 0.43	$ 0.10	$ 0.03

2 · RAW CAMELOT — IMPERFECT

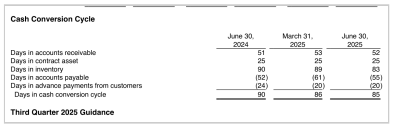
```
| | | | Three Months Ended | |
| --- | --- | --- | --- | --- |
| Summary GAAP Items | | June 30, |
March 31, | June 30, |
| (Amounts in millions, except per share
data) | | 2024 | 2025 | 2025 |
| Revenue | $ | 666
 $ | 632
 $ | 642 |
| Gross Margin | | 10.2% |
10.0% |
10.1% |
| Operating Margin | | 4.1% |
1.9% |
3.2% |
| Diluted EPS | $ | 0.43
 $ | 0.10
 $ | 0.03 |
```

3 · CORRECTED MARKDOWN

	Three Months Ended June 30, 2024	Three Months Ended March 31, 2025	Three Months Ended June 30, 2025
Revenue	$ 666	$ 632	$ 642
Gross Margin	10.2%	10.0%	10.1%
Operating Margin	4.1%	1.9%	3.2%
Diluted EPS	$ 0.43	$ 0.10	$ 0.03

Figure — Structure correction: Cash Conversion Cycle (p2 · raw grid 11×7 · llm_corrected=True)

1 · SOURCE (THE PDF TABLE)



Cash Conversion Cycle

	June 30, 2024	March 31, 2025	June 30, 2025
Days in accounts receivable	51	53	52
Days in contract asset	25	25	25
Days in inventory	90	89	83
Days in accounts payable	(52)	(61)	(55)
Days in advance payments from customers	(24)	(20)	(20)
Days in cash conversion cycle	90	86	85

Third Quarter 2025 Guidance

2 · RAW CAMELOT — IMPERFECT

```
| C
ash Conversion Cycle | | | | |
| |
| --- | --- | --- | --- | --- |
--- | --- |
| | June 30, | | March 31, | |
June
30, | |
| | 2024 | | 2025 | | 2025 | |
| Days in accounts receivable |
| 51 |
| 53 | | 52 |
| Days in contract asset | | 25
| | 25
| | 25 |
| Days in inventory | | 90 | |
89 | |
83 |
| Days in accounts payable | |
(52) |
| (61) | | (55) |
| Days in advance payments from
customers | | (24) | | (20) | |
(20)
|
| | | | | | |
| | | 90 | | 86 | | |
| T
hird Quarter 2025 Guidance | |
| | |
| |
```

3 · CORRECTED MARKDOWN

	June 30, 2024	March 31, 2025	June 30, 2025
Days in accounts receivable	51	53	52
Days in contract asset	25	25	25
Days in inventory	90	89	83
Days in accounts payable	(52)	(61)	(55)
Days in advance payments from customers	(24)	(20)	(20)
Days in cash conversion cycle	90	86	85

In both cases `llm_corrected=True`: the correction differed from the deterministic grid render, and every numeric value survived the grounding guard. The structure was rearranged to match the picture; not a single number was read off the image, invented, or changed.

8 The Error-Correction & Completeness Layer

Beyond the grounding guard, Quber carries a deterministic completeness-and-repair toolkit in `src/quber/agents/completeness.py`. It was built during the Camelot rebuild to answer one question: *was the whole table captured, or did part of it get cut off at an edge?* — and to rebuild the missing edge rows from the text layer when so. These are the text-driven error-correction steps that complement the two image-examination steps.

> **WHERE THIS LIVES IN THE FLOW**
>
> The active Set-of-Mark capture path bounds the whole table anatomy up front (Stage 2's anatomy prompt + tighten/pad), which largely removes edge-truncation at the source. The completeness machinery below is the deterministic backstop developed alongside the correspondence rebuild; it embodies the same "compare real text-layer tokens against real output tokens" principle the grounding guard uses, and is the canonical reference for how Quber detects and repairs truncation.

8.1 The completeness auditor (text-layer truncation check)

The key insight (`CompletenessAuditor` docstring): because Camelot reads the text layer, not pixels, it never drops a row from the *middle* of a table. The only way an assembled table can be incomplete is truncation at an *edge*. So the audit is deterministic — no model, no network, no image — and reads the same text layer Camelot does:

- For the table's region it finds every value-like figure in the text layer and compares it against the figures present in the assembled Markdown.
- "Value-like" is deliberately narrow (`VALUE_TOKEN_RE`): a number must carry a thousands separator, a decimal, a percent, parentheses, or five-or-more digits. Bare 1–4 digit integers — years like 2024, days like 31, footnote markers — are excluded so a caption or footnote inside the rough region does not read as a dropped row.
- A missing figure counts as a truncation only if it sits within `EDGE_BAND_PTS = 40` points of the captured span's top or bottom edge. Figures deep inside the span are not drops (Camelot keeps interior rows); figures far outside belong to a neighbour. This edge band is a geometry guard that does not depend on a clean box separating two stacked, near-identical tables — exactly the Visa p1 case where box-based regions bleed.

The auditor returns a `CompletenessVerdict`: `complete`, a human-readable gap description, a `reason`, and a list of `MissingFigure` records (each with its value, position, and which edge it sits past) — the input to the repair step.

8.2 Rounding off truncated rows

When the audit reports missing edge figures, `round_off_grid()` rebuilds the truncated rows from the text layer and merges them back into the cell grid. It is deterministic and value-preserving by construction:

- Numeric column centres are inferred from the value tokens inside the captured span (`infer_columns`); each missing figure is slotted into the grid column whose centre is nearest its x-position.
- The restored row's label is read from the non-figure words to the left of the first numeric column (`row_label_at`).
- If a row with that label already exists and has an empty target cell, the figure fills it in place — no duplicate row, column count preserved. A label can repeat across sections ("Dec 31, 2024" under both "3 Months" and "12 Months"); if the matching row's target cells are already full, a new row is inserted rather than overwriting.
- A non-empty Camelot cell is never overwritten; inserted figures are the literal text-layer tokens, never invented or computed.

8.3 Repairing a drifted detector box

For the legacy detector path, `tabular_bands()` + `repair_box()` relocate a bounding box that landed in whitespace onto the tabular text actually there. `tabular_bands` finds vertical bands of contiguous rows that each carry two-or-more value-like figures (real data rows, excluding prose and lone labels); `repair_box` snaps the box to the band with the most vertical overlap, fixing both a too-narrow box (clipped row labels) and a vertically offset box. If no tabular band overlaps or sits near the box, it returns `None` — a phantom box over whitespace or a decorative banner is dropped rather than trusted.

8.4 Report, don't drop

The unifying rule across all of these steps: **when a gap cannot be repaired, it is reported, never ignored.** A table the vision step found but Camelot could not extract is emitted with its identity and an empty body. A truncation that no leftover text can supply is flagged. This is the direct codification of the standing instruction that, for financial documents, silent omission is a liability with no defence — failures must be surfaced, never silently judged acceptable.

PIPELINE · FINALIZE

9 Finalization & Output

The per-table branches rejoin at a list-append join (with `preferred_parent_fork='closest'` to avoid a premature firing), and `finalize` sorts the captured tables by (page, ordinal) — restoring document reading order, which the downstream LLM inference and rendering depend on. Each emitted `ExtractedTable` carries both its content and a full provenance trail:

Field group	Fields	Source
Content	`title`, `subtitle`, `markdown`, `footnotes`	Structure correction (or fallback grid render)
Camelot provenance	`page`, `bbox`, `flavor`, `camelot_accuracy`	Region-constrained Camelot pass
Set-of-Mark provenance	`som_region` (normalized 0–1 box)	The grid locator
Audit	`llm_corrected` (did correction differ from the grid render?)	Grounding guard outcome

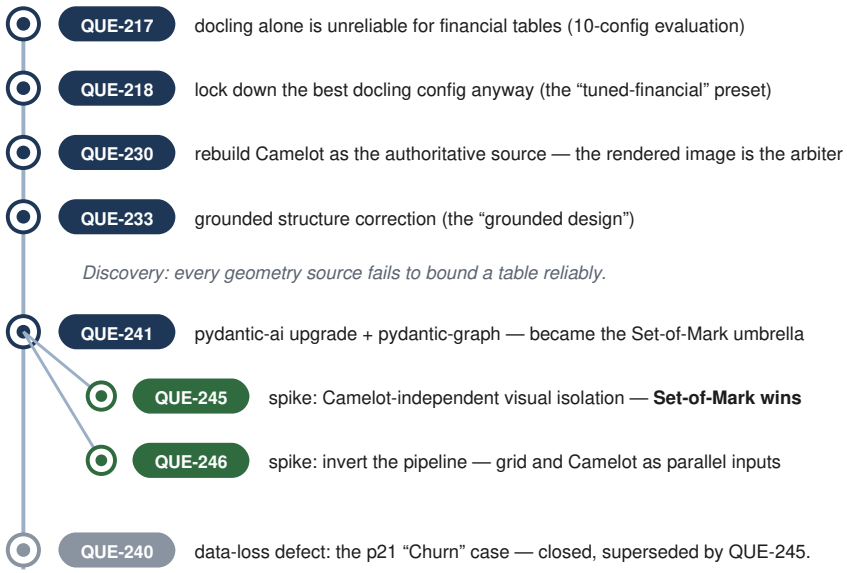
Every numeric value in the output is traceable to the page text layer (the characters Camelot reads), and each table records exactly how it was produced.

10 Design Principles, Distilled

1. **Mark, don't regress.** Never ask the model for a continuous coordinate; overlay labelled marks and have it read them. (The two cited papers.)
2. **Vision for identity & structure; text for values.** Pixels locate and judge shape; the text layer is the only source of numbers.
3. **Two image examinations, two jobs.** The page-level grid finds tables; the table-level crop fixes structure. Neither ever reads a value off the image.
4. **Ground every number.** A figure that is not in the page text layer cannot appear in the output — enforced in code, by presence, after the model responds.
5. **Presence, not literal match.** Allow legitimate reformatting; reject only invented figures.
6. **Region-scope everything.** Crop and value slice are bounded to one table so neighbours cannot contaminate it.
7. **Report, don't drop.** Unextractable or truncated tables are surfaced with their identity, never silently omitted.
8. **Determinism where possible.** Grid→box mapping, completeness audit, and row round-off are deterministic; the model is used only where judgement is genuinely required.

11 Project History: The Jira Lineage

There is no single "Set-of-Mark" epic. The technique was discovered, validated, and adopted through a chain of tickets, almost all under **Epic QUE-91 (Extraction & Validation)**. The sequence that produced it:



11.1 The hierarchy

Key	Type	Summary	Status
QUE-91	Epic	Extraction & Validation (parent of nearly all of the below)	**To Do**
QUE-217	Task	Evaluate docling extraction modes (10-config eval)	Done
QUE-218	Story	Adopt `tuned-financial` docling config as canonical	Done
QUE-216	Story	Surface docling stage errors as `quality_flags`	**To Do**
QUE-219	Story	Enable GPU for the RapidOCR/onnxruntime stage	Done
QUE-230	Task	Rebuild Camelot with LLM region correspondence (image=arbiter)	Done

Key	Type	Summary	Status
QUE-233	Task	Restructure presentation without altering Camelot values (grounded design)	Done
QUE-237	Task	Rename TableMerger → TableUnifier (no behaviour change)	Done
QUE-241	Story	Migrate pydantic-ai + stage pydantic-graph (became SoM umbrella)	Done
└ QUE-245	Sub-task	**The Set-of-Mark spike** — reliable visual isolation	Done
└ QUE-246	Sub-task	Spike: parallel visual-grid + Camelot as first-class inputs	Done
QUE-240	Sub-task	Fix recovery misfire that drops a table (p21 Churn)	Closed
QUE-247	Task	Column ID for shared-header stacked tables (deferred from QUE-246)	**To Do**
QUE-251	Bug	`extract_page_context` silently returns no context	Done
QUE-244	Task	Upgrade docling to 2.102.1 and validate (Epic QUE-1)	Done
QUE-250	Task	Clean up pyright warnings in the test suite (Epic QUE-89)	Done

11.2 The decisive tickets, in detail

QUE-217 — the evaluation that justified the program

A page-by-page evaluation of 10 docling configurations against the TMUS Q2 FY25 investor fact-book. It established the failures that justify the entire program: a *silent* legal-disclaimer drop on page 29 in all 10 runs; VLM pipelines disqualified (Granite-Docling transcribed "AT&T" as "ATAT" and hallucinated quarters; SmolDocling generation-collapsed on ~44% of the file); chart extraction unsafe (a mislabelled stacked bar risks a ~120× overstatement); picture-description hallucinating authoritative-looking captions. It selected `tuned-financial` as the best candidate and explicitly seeded QUE-216 and QUE-218.

QUE-230 & QUE-233 — image as arbiter, and the grounded design

QUE-230 rebuilt Camelot as the authoritative source around "the rasterized page as the arbiter," recognizing lattice (high precision) and stream (high recall) as complementary rather than redundant. QUE-233 then proved the *grounded design* in a 31-table sweep across four real financial PDFs: the image arbitrates structure, Camelot ∪ text layer together hold every value, the grounding guard replaces the wrong exact-literal guard, and correction is region-scoped to prevent the Visa p1 "Dec/Sep clobbering." Result: **31 tables, 0 hallucinations, 0 real losses.**

QUE-245 — the spike that chose Set-of-Mark

The pivotal experiment. It compared three Camelot-independent isolation methods: raw continuous-coordinate vision (macro IoU 0.42, drift and count-flips), the existing grid "data-grid" prompt (IoU 0.66, count-stability 1.00), and a new grid "anatomy" prompt that bounds the full table anatomy — title, units, spanning headers, stub column, body, totals, footnotes (IoU 0.70, count-stability 1.00, worst title clip 13 pt). The feared non-deterministic table drop did not reproduce. The accepted recommendation: **promote vision + grid (Set-of-Mark), anatomy prompt, low-temperature model, 36-row grid, finished with declash + 6 pt pad, from a last-resort escalation to a primary every-page pass.**

QUE-246 — inverting the pipeline

The follow-on spike that made the grid a primary, parallel input alongside Camelot instead of a last-resort cleanup — the shape the current `set_of_mark/pipeline.py` implements. Because the grid is count-stable and accurate while the raw detector is the drift-prone component, leading with the grid makes boundary and count correct from the start and renders most of the old drift-repair and escalation scaffolding redundant. It deferred shared-header column labelling to QUE-247.

12 Known Gaps & Open Work

OPEN · LABELLING GAP

QUE-247 — shared-header stacked tables. When several sub-tables stack under one shared column-header band, each sub-table's crop sits *below* the header, so structure correction sees no header and falls back to generic "Column 1–6" labels. The data is correct and aligned; only column *identity* (which period each number belongs to) is missing — unfixable by prompt alone because the header is physically absent from the crop. The proposed fix carries the shared header as column metadata onto sibling sub-tables.

OPEN · DEFENSE-IN-DEPTH

QUE-216 — surface docling stage errors as `quality_flags`. Converts docling WARN-
ING/ERROR log records into typed quality flags attached to the output, with an `is_clean()` gate (any flag → human review). Intended to ship before any production rollout of automated docling extraction; operationalizes the "surface failures explicitly" rule.

For completeness: QUE-251 (now fixed) was a silent degradation on the analyze path — `extract_page_context` always returned no context because it iterated docling items incorrectly and read a non-existent `_page` attribute. The bug confirmed that the Set-of-Mark extraction engine does not use that method, so table data integrity was unaffected; the impact was context-free LLM titles/descriptions only.

13 References

Cited research

1. J. Yang, H. Zhang, F. Li, X. Zou, C. Li, J. Gao. *Set-of-Mark Prompting Unleashes Extraordinary Visual Grounding in GPT-4V.* arXiv:2310.11441 (2023).
2. X. Lei, J. Yang, et al. *Scaffolding Coordinates to Promote Vision-Language Coordination in Large Multi-Modal Models.* arXiv:2402.12058 (2024).

Primary source files

- `src/quber/agents/grid_locator.py` — Set-of-Mark grid locator (marks, geometry, paper citations)
- `src/quber/core/extractors/set_of_mark/orchestrator.py` — ingestion, render, run the graph
- `src/quber/core/extractors/set_of_mark/pipeline.py` — the three-step pydantic-graph
- `src/quber/core/extractors/camelot/correspondence/correction.py` — grounded structure correction + grounding guard
- `src/quber/agents/llm_client.py` — `VET_STRUCTURE_PROMPT` and the correction schema
- `src/quber/agents/completeness.py` — completeness audit, round-off recovery, box repair
- `src/quber/core/extractors/camelot/acquire.py` / `.../correspondence/recovery.py` / `.../geometry.py` — render, region-constrained Camelot, frame conversions
- `docs/EXTRACTION_PIPELINE.md` — canonical `tuned-financial` configuration

Jira

Project QUE. Epics: QUE-91 (Extraction & Validation), QUE-1 (Document Processing), QUE-89 (Development Infrastructure). Set-of-Mark lineage: QUE-217 → 218 → 230/233 → 241 (245, 246). Defect QUE-240 (p21 Churn), superseded by 245. Open: QUE-216, QUE-247.

— End of document —