QUBER ENGINEERING · QUE-241

# The Camelot Workflow

## From PDF pages to trusted tables

How Quber turns PDF pages into tables you can trust. Camelot reads the PDF and produces candidate cell grids; three PydanticAI agents judge and arrange them; and a pydantic-graph `GraphBuilder` graph runs the stages concurrently while producing byte-identical output. The document assumes Python and async/await, but explains Pydantic, PydanticAI, and pydantic-graph from the ground up.

Engineering Reference · June 2026

Source of truth: `src/quber/core/extractors/camelot` · `src/quber/agents`

Branch `feature/QUE-241-pydantic-graph-async-pipeline`

# Contents



## Audience and scope

This is for an engineer who will read, extend, or debug the Camelot table extractor. It covers the `llm` extraction strategy end to end: the data models, the three LLM agents, the pydantic-graph orchestration, the four invariants that keep concurrent output identical to the old sequential output, and the package layout. You need Python and async/await. You do not need prior Pydantic, PydanticAI, or pydantic-graph knowledge.

# 1 The problem, and the trust boundary

A PDF does not contain tables. It contains positioned glyphs, ruled lines, and images — a drawing of a table, not a data structure. Recovering the table is an act of inference. Financial documents raise the stakes: a digit in the wrong column is a wrong number, not a cosmetic defect, and in a document someone relies on that error carries real liability.

Quber's Camelot workflow attacks this in two layers, with a hard boundary between them.

- **Camelot** is a deterministic, geometry-based extractor. It reads the PDF and produces candidate cell grids. Whatever ends up in a cell came from the PDF's own text layer.
- **LLM agents** judge and arrange what Camelot produced. Is this candidate actually a table? Do these two candidates describe the same physical table? Is the markdown structure faithful to the page image? The agents decide structure; they never invent values.

> **THE TRUST BOUNDARY**
>
> Camelot is the source of truth for cell values. The LLMs decide structure and are forbidden from changing numbers. Every agent prompt in the workflow restates this rule, and the pipeline never asks an LLM to produce a value Camelot did not extract.

## 1.1 Two flavors, always both

Camelot has two extraction strategies, which it calls flavors.

- **lattice** follows visible ruled lines. It is precise when the table is drawn with a grid, and blind when it is not.
- **stream** infers columns from whitespace alignment. It catches unruled tables. It is also permissive enough to emit footnote lists, table-of-contents entries, and narrative paragraphs as “tables”.

The workflow runs both flavors on every document, every time. The obvious optimization is to try lattice first and fall back to stream only when lattice finds nothing. That was rejected for a concrete reason. On pages where tables are aligned by whitespace alone, lattice still returns grids. They are empty shells — a one-row grid of empty pipes — but they count as “lattice found something”, so the fallback gate never opens and the real table is lost. Instead, both flavors always run, and a cheap deterministic check (`is_content_empty`) discards the shells before any LLM sees them.

Running both flavors means the same physical table is often extracted twice, slightly differently — one version with a more complete header, the other with the full row set. That duplication is deliberate. Resolving it is a stage of the pipeline, the unifier of chapter 2, not an accident to remove.

## 1.2 The workflow at a glance

End to end, the pipeline is six stages.

1. **Acquire** — render page images and run both Camelot flavors, concurrently; drop content-empty shells.
2. **Classify** — an LLM gives each candidate a binary `is_table` verdict, dropping stream-flavor noise.
3. **Group** — survivors are grouped per physical page, in Camelot's original order.
4. **Unify** — per page, an LLM merges the flavor outputs of the same physical table into one canonical version, using the page image as ground truth.
5. **Correct** — per table, an LLM checks the markdown structure against the page image and extracts title, subtitle, and footnotes.
6. **Finalize** — restore deterministic output order.

Until QUE-241 these stages ran as a sequential pipeline of loops. The change this document describes rebuilt the orchestration as an asynchronous dataflow graph using pydantic-graph's `GraphBuilder`. The stages, agents, and outputs are unchanged, but the per-item work now runs concurrently and the I/O-bound acquisition overlaps. Chapter 3 builds up the graph machinery from first principles; chapter 8 measures what it bought.

# 2 Building blocks: Pydantic models and PydanticAI agents

The whole workflow is built from two primitives: Pydantic models as typed data contracts, and PydanticAI agents as LLM calls with a validated, typed result. Understand these two and the rest of the system is plumbing.

## 2.1 Pydantic models as data contracts

A Pydantic model is a class whose fields are declared with types and validated at construction. It is the common data type passed between every part of this system: what Camelot's subprocess returns, what flows along graph edges, what the LLM must answer with. The foundational one is `CamelotCandidate` — one candidate table as Camelot saw it.

```
# src/quber/core/extractors/camelot/acquire.py

class CamelotCandidate(BaseModel):
    """Serializable per-Camelot-table output. Crosses the
spawn-process
    boundary; only picklable primitives, no Camelot intern-
als."""

    candidate_id: str = Field(description="Stable id assigned
by the parent; flavor-page-idx")
    flavor: Flavor                          # Liter-
al["lattice", "stream"]
    page: int = Field(ge=1)
    bbox: Optional[Tuple[float, float, float, float]] = None
    accuracy: float = Field(default=0.0, ge=0.0, le=100.0)
    cells: List[List[str]] = Field(
        default_factory=list, description="Camelot's raw cell
grid (rows of strings)"
    )
    markdown: str = Field(description="Camelot's raw markdown
for this table")
```

Two details matter here. First, the model is fully serializable — plain strings, ints, tuples — because it must cross a subprocess boundary (chapter 4). Second, the

`Field(description=...)` strings are not only documentation. When a model is used as an LLM's output type, those descriptions are embedded in the JSON schema the model sees, so the contract and its explanation live in one place.

## 2.2 Anatomy of a PydanticAI agent

PydanticAI's core abstraction is the `Agent`: a model, a system prompt, and an `output_type`. Given a Pydantic model as the output type, the agent guarantees that `result.output` is a validated instance of that model, retrying the LLM if it returns something that does not parse. The workflow's classifier shows the complete pattern.

```
# src/quber/agents/classifier.py (condensed)

class ClassifierResult(BaseModel):
    is_table: bool = Field(description="True if the candidate
is an actual data table ...")
    reason: str = Field(default="", description="One short sen-
tence justifying the decision ...")

class PydanticAIClassifier:
    def __init__(self, ...) -> None:
        ...
        self.agent = Agent(
            anth_model,                          # AnthropicMod-
el, default Haiku 4.5
            output_type=ClassifierResult,        # validated
structured output
            system_prompt=CLASSIFY_TABLE_PROMPT,
        )

    async def classify(self, markdown: str) -> ClassifierRes-
ult:
        try:
            result = await self.agent.run(markdown)
            return result.output             # a validated Clas-
sifierResult
        except Exception as exc:
            logger.warning("classify: LLM call failed; default-
accepting: {}", exc)
```

```
            return ClassifierResult(is_table=True, reason="de-
fault-accept on failure")
```

That is the entire LLM integration. There is no JSON parsing, no regex over model output, and no hand-rolled retry on a malformed response. The `output_type=ClassifierResult` declaration moves all of that into the framework. The system prompt embeds `ClassifierResult.model_json_schema()`, so the Pydantic model is the single source of truth for the response shape. Change the model, and both the schema sent to the LLM and the validation applied to its answer change together.

## 2.3 Multimodal input: sending the page image

Two of the three agents reason about a rendered page image, not just text. PydanticAI handles this with `BinaryContent`: the user message becomes a list mixing text and image parts. The unifier sends its candidate payload and the page PNG together.

```
# src/quber/agents/unifier.py (condensed)

async def unify(self, page_image: Path, candidates: List[Can-
didateInput]) -> UnifierResult:
    from pydantic_ai import BinaryContent

    user_text = f"Candidates on this page:\n{candid-
ates_payload(candidates)}"
    image = BinaryContent(data=Path(page_image).read_bytes(),
media_type="image/png")

    result = await self.agent.run([user_text, image])   # text
part + image part
    return result.output                                # a valid-
ated UnifierResult
```

The image is the ground-truth reference. The prompt tells the LLM to use it to decide which physical tables exist on the page, while drawing the numeric values only from the candidates' markdown.

## 2.4 The three agents of the workflow

| Agent | Question it answers | Input | Output | On LLM failure |
|---|---|---|---|---|
| `TableClassifier` | Is this candidate an actual data table? | candidate markdown | `ClassifierResult` (is_table, reason) | default-accept — keep the candidate |
| `TableUnifier` | Which candidates on this page describe the same physical table, and what is its canonical form? | page image + all candidates | `UnifierResult` (canonical tables, source ids) | pass-through — each candidate kept as-is |
| `LLMClient.correct_structure` | Is the markdown structure faithful to the image? What are the title, subtitle, footnotes? | page image + table markdown | `LLMTableCorrection` (title, subtitle, markdown, footnotes) | none — table ships uncorrected |

The classifier is deliberately strict. Its prompt says a borderline block should be rejected, because missing a borderline table is cheaper than shipping a fake one downstream. Note the failure column: every agent, when it fails, keeps data rather than dropping it. A classifier outage keeps candidates instead of dropping them; a unifier outage emits candidates unmerged instead of guessing at a merge; a corrector outage ships Camelot's raw markdown. An LLM failure can cost polish, never data.

The three roles are separate classes on purpose. The classifier can run a cheaper model than the corrector — both default to Haiku 4.5, `claude-haiku-4-5-20251001`, but each is independently configurable — and retry behavior never bleeds across roles.

## 2.5 Protocols and mock backends

The pipeline never depends on the PydanticAI-backed classes directly. Each agent role is a `Protocol`, a structural interface, with at least two implementations: the real PydanticAI one and a mock for tests.

```
# src/quber/agents/classifier.py
```

```
@runtime_checkable
class TableClassifier(Protocol):
    async def classify(self, markdown: str) -> ClassifierRes-
ult: ...

class MockClassifier:
    """Returns a canned ClassifierResult. For tests."""

    def __init__(self, result: Optional[ClassifierResult] =
None) -> None:
        self.result = result
or ClassifierResult(is_table=True, reason="mock")

    async def classify(self, markdown: str) -> ClassifierRes-
ult:
        return self.result
```

A factory (`get_classifier()`, `get_unifier()`, `get_llm_client()`) selects the backend from settings. Tests construct the pipeline with `MockClassifier` and `MockUnifier` and exercise the full graph topology with zero network calls. That is how the test suite for this change verifies ordering and grouping behavior.

# 3 From a loop to a graph: pydantic-graph basics

## 3.1 Why a graph at all

The sequential implementation was the obvious one: render pages, run Camelot, then `for`-loop over candidates calling the classifier, loop over pages calling the unifier, loop over tables calling the corrector. It was correct, and slow in a specific, structural way.

- Page rendering and the Camelot subprocesses are independent I/O-bound work, but ran back to back.
- Each LLM call within a stage is independent of its siblings. Fifty classifier calls have no reason to run one at a time.
- The stages have different natural units of work: the classifier's unit is a candidate, the unifier's a page, the corrector's a table. Hand-rolling per-stage fan-out with `asyncio.gather` at three granularities, while keeping ordering guarantees, is a common source of subtle concurrency bugs.

A dataflow graph states the same program declaratively: these are the steps, these are the edges data flows along, this edge fans out one task per item, this node collects results back into a list. The runtime owns task scheduling; the code owns only what each step does. QUE-241 rebuilt the pipeline on pydantic-graph — the graph library underneath PydanticAI, from the same team — using its `GraphBuilder` API.

> **API GENERATIONS**
>
> `GraphBuilder` is the current pydantic-graph orchestration API. It was promoted out of beta in pydantic-graph 1.97.0 and is the v2-ready surface. The older `Graph(nodes=[...])` / `BaseNode` runner is deprecated and slated for removal in v2, and this codebase does not write new orchestration against it. QUE-241's first commit was precisely the dependency bump — pydantic-ai-slim 1.93.0 to 1.106.0 — that brought `GraphBuilder` in.

## 3.2 The vocabulary: steps, edges, maps, joins

Six concepts cover everything the pipeline uses.

- **Step** — an async function that receives a `StepContext` and returns a value. Registered with `g.step(fn)`.

- **Edge** — `g.add_edge(a, b)`: the value returned by `a` becomes the input of `b`. The special nodes `g.start_node` and `g.end_node` mark where the run's input enters and its output leaves.
- **Map edge** — `g.add_mapping_edge(a, b, ...)`: step `a` returns a list, and the runtime spawns one concurrent task of step `b` per element.
- **Join** — the meeting point where fanned-out tasks are collected back into a single value. Built with `g.join(reducer, ...)`; the reducer folds each arriving branch result into an accumulator. For example, `reduce_list_append` when each branch contributes one item, or `reduce_list_extend` when each branch contributes a list to flatten.
- **State** — one mutable object shared by every step in a run, for cross-stage facts. Here: rendered page-image paths and counters for logs.
- **Deps** — one immutable object carried into every step: clients, configuration, semaphores. This is the dependency-injection channel.

A step's signature names the last three at once. `StepContext[PipelineState, PipelineDeps, PageGroup]` reads as: this step can mutate run state of type `PipelineState`, gets dependencies of type `PipelineDeps`, and its per-invocation input (`ctx.inputs`) is a `PageGroup`.

> **STATE VS. DEPS VS. INPUTS**
>
> Inputs flow along edges and differ per task. Deps are injected, identical for every step, and frozen. State is shared and mutable, so use it sparingly. This pipeline writes state from a single step only — acquire writes the page-image paths once — which keeps the mutation safe.

## 3.3 A minimal worked example

Before the real thing, here is the whole machine in miniature: a graph that fans out over a list of numbers, squares each concurrently, and sums the results.

```
# Illustrative sketch, following the production wiring pattern
from pydantic_graph import GraphBuilder, StepContext, re-
duce_list_append
from pydantic_graph.id_types import JoinID, NodeID

async def explode(ctx: StepContext[None, None, list[int]]) ->
list[int]:
    return ctx.inputs                # returning a list feeds
```

```
the map edge

async def square(ctx: StepContext[None, None, int]) -> int:
    return ctx.inputs ** 2          # runs once per element,
concurrently

async def total(ctx: StepContext[None, None, list[int]]) ->
int:
    return sum(ctx.inputs)          # receives the join's
collected list

g = GraphBuilder(name="square-sum", input_type=list[int], out-
put_type=int)
explode_step, square_step, total_step = g.step(explode),
g.step(square), g.step(total)
j_squares = g.join(reduce_list_append, initial_factory=list,
                   node_id="j_squares", pre-
ferred_parent_fork="closest")

g.add_edge(g.start_node, explode_step)
g.add_mapping_edge(explode_step, square_step,          # fan
out: one task per int
                   down-
stream_join_id=JoinID(NodeID("j_squares")))
g.add_edge(square_step, j_squares)                     # fan
in: collect results
g.add_edge(j_squares, total_step)
g.add_edge(total_step, g.end_node)

graph = g.build()
result = await graph.run(inputs=[1, 2, 3, 4])          # -> 30
```

Three things scale up in the real pipeline. The map edge turns one returned list into N concurrent tasks. The join declares how branch results recombine. And the builder is plain declarative wiring: no step knows it runs concurrently.

## 3.4 The real graph

The production pipeline is the same shape applied three times in a row — three map/join pairs at three granularities, with plain edges between them.

```
# src/quber/core/extractors/camelot/llm/pipeline.py
@cache
def build_pipeline_graph() -> Graph[PipelineState,
PipelineDeps, Path, List[ExtractedTable]]:
    g = GraphBuilder(
        name="camelot.llm_pipeline",
        state_type=PipelineState,
        deps_type=PipelineDeps,
        input_type=Path,
        output_type=List[ExtractedTable],
    )

    acquire_step  = g.step(acquire_candidates)
    classify_step = g.step(classify_candidate)
    group_step    = g.step(group_pages)
    unify_step    = g.step(unify_page)
    log_unified_step = g.step(log_unified)
    correct_step  = g.step(correct_table)
    finalize_step = g.step(finalize)

    j_classified = g.join(reduce_list_append, ini-
tial_factory=list,
                          node_id="j_classified", pre-
ferred_parent_fork="closest")
    j_rows       = g.join(reduce_list_extend, ini-
tial_factory=list,
                          node_id="j_rows", pre-
ferred_parent_fork="closest")
    j_corrected  = g.join(reduce_list_append, ini-
tial_factory=list,
                          node_id="j_corrected", pre-
ferred_parent_fork="closest")

    g.add_edge(g.start_node, acquire_step)
    g.add_mapping_edge(acquire_step, classify_step,
                       down-
stream_join_id=JoinID(NodeID("j_classified")))
    g.add_edge(classify_step, j_classified)
    g.add_edge(j_classified, group_step)
    g.add_mapping_edge(group_step, unify_step,
                       down-
stream_join_id=JoinID(NodeID("j_rows")))
```

```
    g.add_edge(unify_step, j_rows)
    g.add_edge(j_rows, log_unified_step)
    g.add_mapping_edge(log_unified_step, correct_step,
                       downstream_join_id=JoinID(NodeID("j_cor-
rected")))
    g.add_edge(correct_step, j_corrected)
    g.add_edge(j_corrected, finalize_step)
    g.add_edge(finalize_step, g.end_node)

    return g.build()
```

Figure 1 shows the resulting topology. The green double-bordered steps are the mapped, fanned-out ones; the trapezoids are joins; the edge labels carry the payload types from Appendix A.

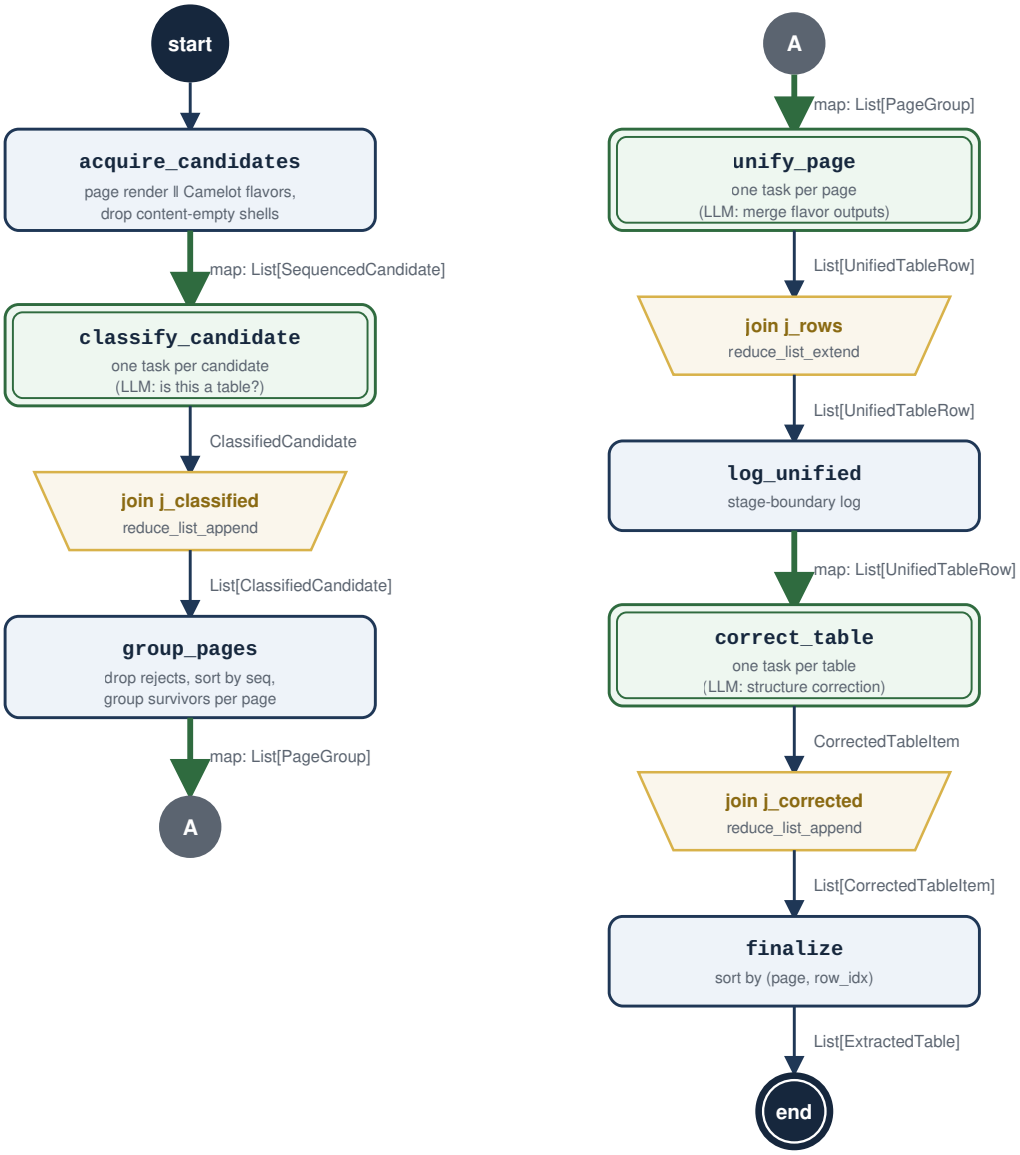


*Figure 1 — The extraction pipeline as a pydantic-graph `GraphBuilder` topology. Read down the left column, then down the right; the Ⓐ connector joins them. Bold green edges are map fan-outs (one concurrent task per item), and each pairs with its own join.*

# 4 The pipeline, stage by stage

This chapter walks the graph top to bottom. Every step lives in `src/quber/core/extractors/camelot/llm/pipeline.py`; the acquisition functions it calls live in `camelot/acquire.py`.

## Stage 1 — acquire_candidates

The entry step does two expensive, independent things at the same time: rasterize every page to PNG (the LLMs will need the images later) and run both Camelot flavors. Each Camelot flavor itself runs in its own spawn-context subprocess, so that OpenCV initialization stays inside the worker and never crosses a fork boundary. Inside this one step there are already three units of parallelism: the renderer thread, the lattice process, and the stream process.

```
# src/quber/core/extractors/camelot/llm/pipeline.py
async def acquire_candidates(
    ctx: StepContext[PipelineState, PipelineDeps, Path],
) -> List[SequencedCandidate]:
    deps = ctx.deps
    page_images, candidates = await asyncio.gather(
        asyncio.to_thread(render_pages, deps.source, deps.dpi,
deps.tmp_dir),
        asyncio.to_thread(run_camelot_flavors_parallel,
deps.source, deps.flavor_timeout_s),
    )
    ctx.state.page_images = tuple(page_images)          #
single writer of run state

    populated = [c for c in candidates if not is_con-
tent_empty(c.markdown)]
    return [SequencedCandidate(seq=seq, candidate=c)
for seq, c in enumerate(populated)]
```

The two operations overlap, so the page render comes off the critical path. Figure 2 shows the schematic; the timing gains are measured in chapter 8.

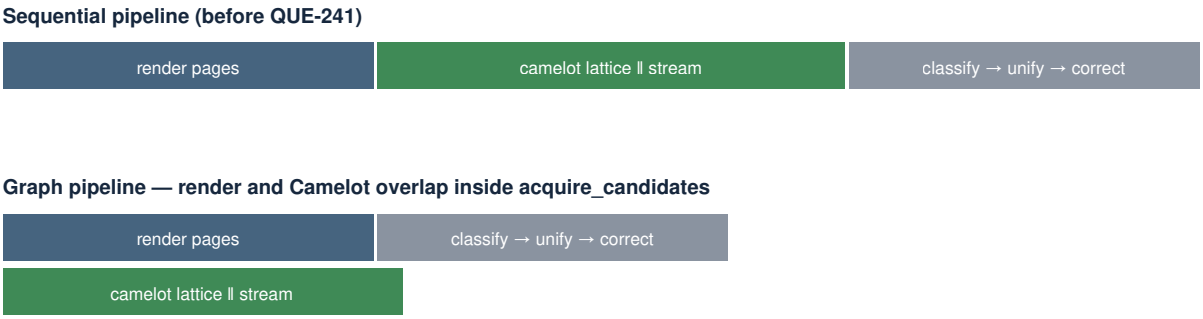


*Figure 2 — Schematic of the acquire overlap: the page render comes off the critical path. Bar widths are illustrative, not measured.*

Before returning, the step filters out content-empty shells (the lattice false positives from chapter 1) and wraps each survivor as a `SequencedCandidate`, pairing it with its index `seq` in Camelot's output order. That integer is essential to correctness; chapter 5 explains why.

## Stage 2 — classify_candidate (map: one task per candidate)

The first map edge fans out: one concurrent task per surviving candidate. The step itself is four lines.

```
# src/quber/core/extractors/camelot/llm/pipeline.py
async def classify_candidate(
    ctx: StepContext[
        PipelineState, PipelineDeps, SequencedCandidate,
    ],
) -> ClassifiedCandidate:
    item = ctx.inputs
    async with ctx.deps.classify_sem:                # at most
max_concurrent LLM calls
        decision = await ctx.deps.classifi-
er.classify(item.candidate.markdown)
    return ClassifiedCandidate(seq=item.seq, candid-
ate=item.candidate, decision=decision)
```

The interesting line is the `async with`. The graph will happily start fifty classify tasks at once. The semaphore in deps ensures that at most `max_concurrent` (default 5) of them are inside an actual LLM call at any moment. The rest wait at the gate, already scheduled.

Figure 3 shows the pattern, drawn for the unify stage; classify and correct are identical in shape.

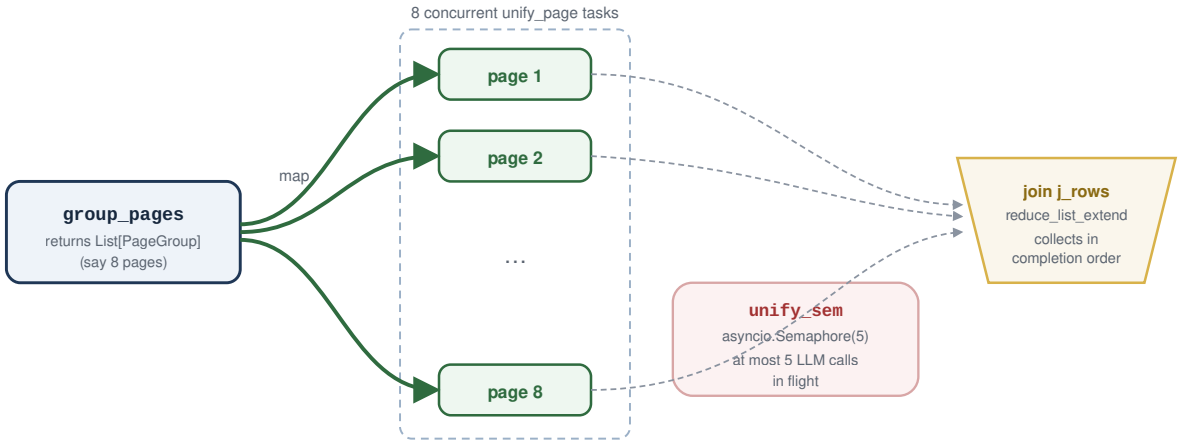


*Figure 3 — A map fan-out gated by a per-stage semaphore. The graph creates one task per item; the semaphore bounds how many are inside an LLM call; the join collects results as they complete.*

## Stage 3 — group_pages

The `j_classified` join hands this step the full list of `ClassifiedCandidate` results, in completion order, which is no useful order at all. The step drops classifier rejects, restores Camelot order by sorting on `seq`, and groups survivors by physical page.

```
# src/quber/core/extractors/camelot/llm/pipeline.py
async def group_pages(
    ctx: StepContext[
        PipelineState, PipelineDeps, List[ClassifiedCandidate],
    ],
) -> List[PageGroup]:
    # restore Camelot order by sorting on seq
    accepted = sorted((item for item in ctx.inputs if item.de-
cision.is_table),
                      key=lambda item: item.seq)
    ctx.state.accepted_count = len(accepted)

    by_page: dict[int, List[ClassifiedCandidate]] = {}
    for item in accepted:
        by_page.setdefault(item.candidate.page,
[]).append(item)
```

```
    return [PageGroup(page=page, items=tuple(items)) for page,
items in sorted(by_page.items())]
```

## Stage 4 — unify_page (map: one task per page)

The second fan-out is one task per `PageGroup`. Pages with a single surviving candidate bypass the LLM entirely, since there is nothing to merge. Pages with several get the full unifier treatment: page image plus all candidates in, canonical table set out. Each output row records which candidate ids contributed to it (`source_ids`) and its position in the unifier's output (`row_idx`).

```
# src/quber/core/extractors/camelot/llm/pipeline.py (condensed)
async def unify_page(
    ctx: StepContext[PipelineState, PipelineDeps, PageGroup],
) -> List[UnifiedTableRow]:
    group = ctx.inputs
    if len(group.items) == 1:                          # nothing
to combine: bypass the LLM
        item = group.items[0]
        return [UnifiedTableRow(page=group.page, row_idx=0,
candidate=item.candidate,
                                decision=item.decision, mark-
down=item.candidate.markdown,

source_ids=(item.candidate.candidate_id,))]

    page_image = ctx.state.page_images[group.page - 1]
    unifier_input = [CandidateIn-
put(candidate_id=i.candidate.candidate_id,
                                    mark-
down=i.candidate.markdown, bbox=i.candidate.bbox)
                     for i in group.items]
    async with ctx.deps.unify_sem:
        result = await ctx.deps.unifier.unify(page_image,
unifier_input)

    return [UnifiedTableRow(page=group.page,
row_idx=row_idx, ...,
                            markdown=table.markdown,
```

```
source_ids=tuple(table.source_candidate_ids))
            for row_idx, table in enumerate(result.tables)]
```

## Stage 5 — correct_table (map: one task per table)

After a logging step marks the stage boundary (accepted-to-unified counts), the third fan-out runs structure correction on each unified table. The LLM receives the unifier's markdown and the page image as reference, and returns title, subtitle, footnotes, and possibly corrected markdown. The guards around the call encode the trust boundary.

```
# src/quber/core/extractors/camelot/llm/pipeline.py (condensed)
async def correct_table(
    ctx: StepContext[
        PipelineState, PipelineDeps, UnifiedTableRow,
    ],
) -> CorrectedTableItem:
    row, deps = ctx.inputs, ctx.deps
    corrected_md, llm_corrected = row.markdown, False
    title = subtitle = ""
    footnotes: List[str] = []

    if deps.run_llm_correction and page_image is not None and
not is_content_empty(row.markdown):
        async with deps.correct_sem:
            try:
                correction = await deps.llm.cor-
rect_structure(page_image, row.markdown)
            except Exception as exc:
                logger.error("LLM correction failed page={}
candidate={}: {}", ...)
                correction = None
        if correction is not None:
            title, subtitle = correction.title, correc-
tion.subtitle
            footnotes = list(correction.footnotes)
            if correction.markdown and correc-
tion.markdown.strip():
                corrected_md = correction.markdown
                llm_corrected = corrected_md != row.markdown

    table = ExtractedTable(title=title, subtitle=subtitle,
```

```
markdown=corrected_md,
                            footnotes=footnotes, page=row.page,
bbox=row.candidate.bbox,
                            flavor=row.candidate.flavor,
source=str(deps.source),
                            cam-
elot_accuracy=row.candidate.accuracy,
                            classifier_decision=row.decision,
llm_corrected=llm_corrected)
    return CorrectedTableItem(page=row.page,
row_idx=row.row_idx, table=table)
```

A Camelot-empty input gets no correction — strict source-of-truth, no image-reading rescue. If the LLM call fails, the table ships with Camelot's markdown untouched. The step finishes by assembling the public `ExtractedTable` with full provenance: page, bbox, flavor, Camelot accuracy, the classifier's decision, and whether the LLM actually changed the markdown (`llm_corrected`).

## Stage 6 — finalize

The last join collects corrected tables in completion order; the final step sorts them by `(page, row_idx)` and unwraps the payload.

```
# src/quber/core/extractors/camelot/llm/pipeline.py
async def finalize(
    ctx: StepContext[
        PipelineState, PipelineDeps, List[CorrectedTableItem],
    ],
) -> List[ExtractedTable]:
    ordered = sorted(ctx.inputs, key=lambda item: (item.page,
item.row_idx))
    return [item.table for item in ordered]
```

The result handed back from `graph.run(...)` is a `List[ExtractedTable]` in exactly the order the sequential pipeline produced: pages ascending, unifier row order within each page.

# 5 The invariants that keep outputs identical

Making this pipeline concurrent was not the hard part — pydantic-graph does that with a few `add_mapping_edge` calls. The hard part was concurrency without changing a single byte of output. Four invariants carry that guarantee.

## Invariant 1 — LLM-input ordering is essential

Graph joins collect results in completion order: whichever task finishes first lands first. For pure data that is harmless. But one consumer in this pipeline is order-sensitive: the unifier LLM.

> **OBSERVED FAILURE**
>
> On Visa Q2FY25 page 3, feeding the unifier the same candidates in shuffled order produced digit-altered merges. The LLM combined the same two flavor outputs differently depending on which it read first, and numbers changed. An LLM call is not a pure function of the set of its inputs; it is a function of their sequence.

Hence `SequencedCandidate`: every candidate carries its index in Camelot's original output order, assigned once in the acquire step. After the classify join scrambles completion order, `group_pages` sorts on `seq` before grouping, so every unifier call sees its page's candidates exactly as the sequential pipeline presented them.

```
# src/quber/core/extractors/camelot/llm/pipeline.py
class SequencedCandidate(BaseModel):
    """A Camelot candidate plus its position in Camelot's out-
put order.

    `seq` exists because graph joins collect in completion or-
der: without
    it, the unifier would see each page's candidates in a non-
deterministic
    order, and the unifier LLM's output is sensitive to candid-
ate order
    (observed on Visa Q2FY25 page 3: shuffled input produced
digit-altered
    merges). Sorting on `seq` restores the sequential
pipeline's exact
    LLM-input ordering."""
```

```
    model_config = ConfigDict(frozen=True)

    seq: int
    candidate: CamelotCandidate
```

## Invariant 2 — bounded LLM concurrency

The map edges put no ceiling on task count: a 100-candidate document creates 100 classify tasks. The pre-graph pipeline had a contract of at most `max_concurrent` LLM calls in flight per stage, and the graph version keeps it with three `asyncio.Semaphore` objects carried in `PipelineDeps` — one per LLM stage (`classify_sem`, `unify_sem`, `correct_sem`). Fan-out is unbounded and cheap; the expensive resource is bounded where it is spent, inside the step, around just the LLM call.

## Invariant 3 — one join per map edge, closest fork

Every join in the graph is declared with `preferred_parent_fork="closest"`, and every map edge names its own dedicated join via `downstream_join_id`. This exact shape is deliberate.

> **TOPOLOGY RULE**
>
> A join fed by other joins across a broadcast fork fires before all branches have finished, silently dropping results (observed against pydantic-graph 1.106.0). Keep the one-join-per-map shape: each fan-out pairs with its own closest-fork join, and joins never feed joins across a fork. The module docstring of `pipeline.py` carries the same warning for future editors.

## Invariant 4 — deterministic output order

Two sort keys travel with the data precisely so the end can be put back in order: `seq` (Camelot order, used before LLM input) and `(page, row_idx)` (output order, applied in `finalize`). Nothing downstream of the graph needs to know the run was concurrent.

## Failure containment, restated

Concurrency adds one more reason the per-agent fallbacks of chapter 2 matter: with up to fifteen LLM calls in flight across stages, a transient API error must not poison a whole run. Each failure is contained to its one candidate, page, or table, and each fails in the conservative direction — keep, pass through, or ship uncorrected. On the acquisition side

the same rule holds: a Camelot flavor that times out or crashes contributes zero candidates while the other flavor's output still flows. Only if both fail does the run produce nothing, and it says so at ERROR level rather than pretending the document was empty.

# 6 Wiring it up: deps, state, and the public surface

The graph itself is stateless and immutable, so it is built exactly once per process (`@cache`) and reused across documents. Everything run-scoped — clients, paths, semaphores — travels through `PipelineDeps`; the only mutable run state is the page-image list and a counter.

```
# src/quber/core/extractors/camelot/llm/pipeline.py
@dataclass
class PipelineState:
    """Mutable run-scoped state. `page_images` is written once
by the
    acquire step (single writer); the counters feed stage-
boundary logs."""

    page_images: Tuple[Path, ...] = ()
    accepted_count: int = 0

@dataclass(frozen=True)
class PipelineDeps:
    """Run-scoped clients + config carried into every graph
step."""

    llm: LLMClient
    classifier: TableClassifier
    unifier: TableUnifier
    source: Path
    tmp_dir: Path
    dpi: int
    run_llm_correction: bool
    flavor_timeout_s: float
    classify_sem: asyncio.Semaphore = field(repr=False)
    unify_sem: asyncio.Semaphore = field(repr=False)
    correct_sem: asyncio.Semaphore = field(repr=False)
```

The public face is `CamelotLLMTableExtractor`. Its `extract_tables` assembles deps, constructing the three semaphores from one `max_concurrent` knob, opens a tem-

porary directory whose lifetime brackets the whole run (page images live there and vanish afterwards), and runs the graph.

```
# src/quber/core/extractors/camelot/llm/orchestrator.py
class CamelotLLMTableExtractor:
    async def extract_tables(self, source: Path) -> List[Ex-
tractedTable]:
        source = Path(source)
        if not source.exists():
            raise FileNotFoundError(source)

        graph = build_pipeline_graph()
        with tempfile.TemporaryDirectory(prefix="quber-cam-
elot-") as tmpdir:
            deps = PipelineDeps(
                llm=self.llm, classifier=self.classifier, uni-
fier=self.unifier,
                source=source, tmp_dir=Path(tmpdir),
dpi=self.dpi,
                run_llm_correction=self.run_llm_correction,
                flavor_timeout_s=self.flavor_timeout_s,
                classi-
fy_sem=asyncio.Semaphore(self.max_concurrent),
                uni-
fy_sem=asyncio.Semaphore(self.max_concurrent),
                cor-
rect_sem=asyncio.Semaphore(self.max_concurrent),
            )
            return await graph.run(state=PipelineState(),
deps=deps, inputs=source)

    def extract_tables_sync(self, source: Path) -> List[Extrac-
tedTable]:
        """Sync entry point for callers without an event loop
(CLI, scripts)."""
        return asyncio.run(self.extract_tables(source))
```

## 6.1 The TableExtractor seam

Both Camelot extractors implement one `Protocol`, and QUE-241 aligned it with the async reality: async owns the bare name, sync gets the explicit suffix.

```python
# src/quber/core/extractors/base.py
@runtime_checkable
class TableExtractor(Protocol):
    async def extract_tables(self, source: Path) -> List[Ex-
tractedTable]: ...

    def extract_tables_sync(self, source: Path) -> List[Extrac-
tedTable]: ...
```

The CLI is typed against the Protocol, not the concrete class (`extractor: TableExtractor = CamelotLLMTableExtractor(...)` in `cli.py`), so a real call site uses the interface rather than leaving it defined and never called. CLI commands, having no event loop, call `extract_tables_sync`; async callers, including any future service embedding, await `extract_tables` directly.

# 7 The wider camelot namespace

QUE-241 also reorganized the code so the directory layout matches the design. Everything Camelot lives under `quber.core.extractors.camelot`, namespaced by product and subdivided by function.

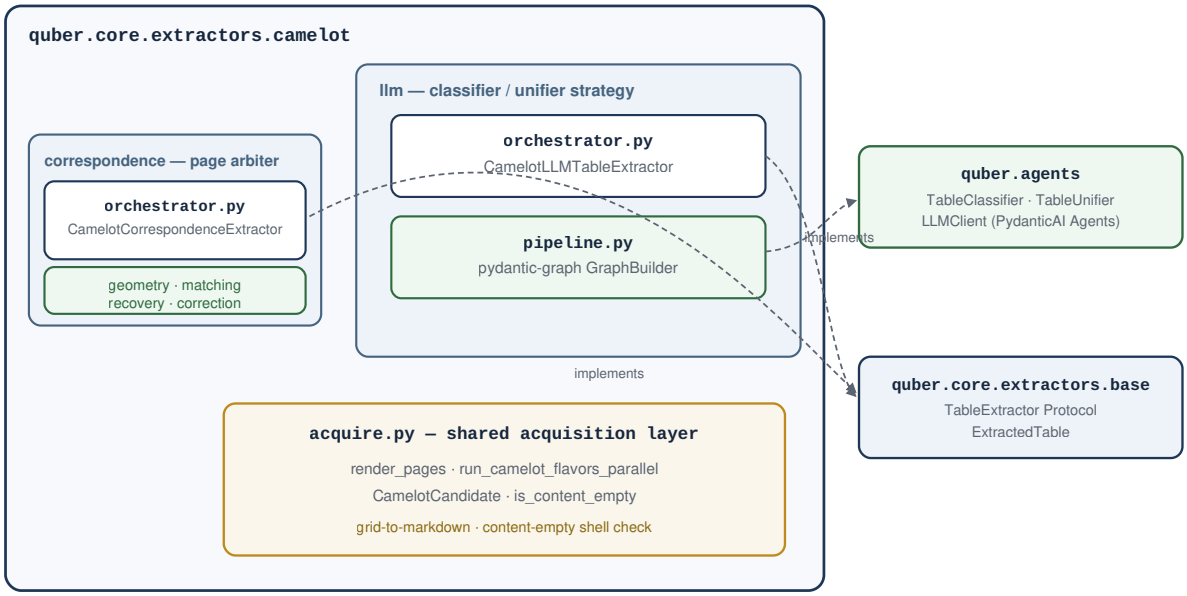


*Figure 4 — The camelot namespace. Two extraction strategies (`llm` and `correspondence`) share one acquisition layer and both implement the `TableExtractor` Protocol.*

## 7.1 The shared acquisition layer

`acquire.py` is the part both strategies need and neither owns: page rendering, both Camelot flavors in spawn-context subprocesses with a per-flavor timeout, the `CamelotCandidate` model, grid-to-markdown rendering, and the content-empty shell check. It knows nothing about LLMs or graphs.

```
# src/quber/core/extractors/camelot/acquire.py (condensed)
def run_camelot_flavors_parallel(source: Path, timeout_s: float
= 60.0) -> List[CamelotCandidate]:
    """Run lattice + stream concurrently in spawn-context sub-
processes.

    A flavor that times out or raises is logged ERROR and con-
tributes
    zero candidates; the other flavor's output is still re-
```

```
turned."""

    ctx = mp.get_context("spawn")
    candidates: List[CamelotCandidate] = []
    with ProcessPoolExecutor(max_workers=2, mp_context=ctx) as
pool:
        futures = {flavor: pool.submit(camelot_worker,
str(source), flavor)
                   for flavor in ("lattice", "stream")}
        for flavor, future in futures.items():
            try:
                candid-
ates.extend(future.result(timeout=timeout_s))
            except FutureTimeoutError:
                logger.error("camelot {flavor} timed out after
{timeout}s on {source}", ...)
            except Exception as exc:
                logger.error("camelot {flavor} raised on
{source}: {exc}", ...)
    return candidates
```

## 7.2 The sibling strategy: correspondence

This document's subject is the `llm` strategy, but the namespace diagram shows a sibling worth one paragraph. `CamelotCorrespondenceExtractor` reverses which side judges what exists. Instead of asking an LLM to judge Camelot's candidates, a vision detector agent reads the rendered page and declares what tables exist and where. Camelot chunks are then assigned to detected tables by deterministic bbox overlap, and a completeness audit checks the assembled grids against the page's text layer. Its defining policy is report, don't drop: a table the detector saw but Camelot never produced is emitted explicitly with status `detected_not_extracted`, never silently omitted. Each of its outputs carries an `ExtractionRecord` audit trail on the `ExtractedTable`. The two strategies share `acquire.py` and the output model, and differ in who decides which tables exist.

# 8 What the rewrite bought

QUE-241 landed in two deliberately separated phases.

1. **Phase 1 — dependency bump only:** pydantic-ai-slim 1.93.0 to 1.106.0, bringing in the stable `GraphBuilder` API with the sequential pipeline untouched.
2. **Phase 2 — the orchestration rewrite** described in this document, plus the package reorganization of chapter 7.

Measured on a real production document (`TMUS_Q225_992.pdf`, run with `--no-llm` so the deterministic work isolates the orchestration change), with outputs verified identical:

| | Sequential pipeline | Graph pipeline |
|---|---|---|
| **Wall-clock time** | ~53.3 s | ~34.7 s |
| **Outputs** | byte-identical | |

That win comes from one place: overlapping page rendering with the Camelot subprocesses inside `acquire_candidates`. With `--no-llm` the fan-out stages have little to parallelize. On full runs the per-stage LLM fan-out additionally overlaps classifier, unifier, and corrector latency within each stage, bounded by the semaphores so the API-side behavior matches the sequential pipeline's concurrency contract.

The other gains are structural. The pipeline's concurrency is now declarative and inspectable — six steps, three map/join pairs, one diagram — instead of hand-threaded `gather` calls. The ordering guarantees that used to be implicit in loop order are now explicit, named fields (`seq`, `row_idx`) with documented reasons. And the orchestration sits on the API surface pydantic-graph will carry into v2, not the one it is deprecating.

# A Appendix A — Payload model reference

Every value flowing along a graph edge is a frozen Pydantic model (immutable by `model_config = ConfigDict(frozen=True)`, safe to share across concurrent tasks). In pipeline order:

| Model | Produced by | Consumed by | Carries |
|---|---|---|---|
| `SequencedCandidate` | `acquire_candidates` | `classify_candidate` | candidate + seq (Camelot output order) |
| `ClassifiedCandidate` | `classify_candidate` | `group_pages` (via join) | candidate + seq + classifier decision |
| `PageGroup` | `group_pages` | `unify_page` | page number + accepted candidates in seq order |
| `UnifiedTableRow` | `unify_page` | `correct_table` (via join + log) | canonical markdown + (page, row_idx) + source ids + primary candidate |
| `CorrectedTableItem` | `correct_table` | `finalize` (via join) | finished ExtractedTable + (page, row_idx) ordering key |
| `ExtractedTable` | `finalize` | caller of `graph.run` | title, subtitle, markdown, footnotes + full provenance |

# B Appendix B — pydantic-graph vocabulary

| Term | In this pipeline |
|---|---|
| `GraphBuilder(...)` | Declares the graph's four type parameters: state `PipelineState`, deps `PipelineDeps`, input `Path`, output `List[ExtractedTable]`. |
| `g.step(fn)` | Registers an async function as a node. Seven steps: acquire, classify, group, unify, log, correct, finalize. |
| `StepContext[S, D, I]` | What every step receives: `ctx.state` (mutable, shared), `ctx.deps` (frozen, injected), `ctx.inputs` (this task's payload of type `I`). |
| `g.add_edge(a, b)` | Sequential dataflow: `a`'s return value becomes `b`'s input. |
| `g.add_mapping_edge(a, b, downstream_join_id=...)` | Fan-out: `a` returns a list; one concurrent `b` task per element. Names the join its results must reach. |
| `g.join(reducer, initial_factory, ...)` | Fan-in: folds branch results into one value. `reduce_list_append` when each branch yields one item; `reduce_list_extend` when each yields a list. |
| `preferred_parent_fork="closest"` | Pairs each join with its nearest upstream fork — the topology rule of chapter 5, invariant 3. |
| `g.start_node` / `g.end_node` | Where `graph.run(inputs=...)` feeds the source `Path` in, and where the final `List[ExtractedTable]` leaves. |
| `g.build()` | Validates the wiring and returns the immutable, reusable `Graph`, built once per process under `@cache`. |