FINAL PROJECT REPORT: DEVELOPING AN AI-ASSISTED GRADING SYSTEM USING LARGE LANGUAGE MODELS

by

Andrei Modiga

A FINAL PROJECT REPORT

Presented to the Faculty of

The School of Computing at the Southern Adventist University

In Partial Fulfilment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Scot Anderson, Ph.D.

Collegedale, Tennessee

August, 2025

FINAL PROJECT REPORT: DEVELOPING AN AI-ASSISTED GRADING SYSTEM USING LARGE LANGUAGE MODELS

Andrei Modiga, M.S.

Southern Adventist University, 2025

Adviser: Scot Anderson, Ph.D.

We present a grading system that accelerates evaluation of open-ended student work across scanned and digital workflows. The system crops answer regions from PDFs, assigns submissions via OCR on identity regions only, and groups answers by visual semantics using a vision LLM. Instructors review and edit groups, apply rubric items once per group, and export grades from an on-screen table. The solution integrates Ghostscript rasterization, PdfPig page orchestration, SkiaSharp region extraction, Tesseract identity OCR, and GPT-4o Vision for grouping[1, 2, 3, 4, 5]. We detail the architecture, token-budgeted batching strategy, and persistence design, then describe testing results for grouping quality, time-on-task, and usability. The approach avoids brittle handwriting OCR while preserving instructor control, fairness, and auditability.

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Motivation

1.1 Problem Statement

Grading open-ended student work (handwritten or typed) is time-consuming, repetitive, and error-prone under deadline pressure. In large classes, feedback latency diminishes learning value. Typical bottlenecks include: (i) organizing mixed-format submissions (bulk scans vs. individual PDFs), (ii) locating answer regions for consistent review, and (iii) repeatedly applying identical rubric deductions to similar mistakes. Handwriting OCR is brittle, often forcing manual review even for simple cases.

1.2 Specific Project Goals/Requirements

This project delivers a practical, instructor-in-the-loop grading system that:

- *Bulk Scans (Filled-form):* PDFs are split by known page counts. Ghostscript rasterizes pages; PdfPig validates page counts; SkiaSharp crops defined regions to PNGs.

- *Identity OCR (filled-form only):* Tesseract extracts name/ID from a designated identity region to auto-assign submissions; unresolved items fall back to a manual pick-list. No OCR is performed on answers; grouping uses GPT-4o Vision directly on the cropped images.
- *Identity Matching:* (filled-form) identity text is matched to the class roster; (free-form) uploads are automatically tied to the uploader's account.
- *Editable AI Assistance:* Vision LLM proposes semantic groups; instructors can merge/split groups, move answers, and apply rubric items per-group.
- *Traceability:* All actions and groupings are persisted with timestamps for audit; grades are summarized in an on-screen table for review/copy.

1.3 Motivation and Benefits

- **Faster feedback:** Instructors grade clusters of similar answers once, reducing turnaround time.
- **Consistency:** Per-group rubric application reduces drift across similar answers and sessions.
- **Lower cognitive load:** The system automates extraction, grouping suggestions, and grade totals; instructors focus on assessment.
- **Reduced brittleness:** Avoiding handwriting OCR on answers eliminates a major failure mode.
- **Privacy-aware:** Only identity crops contain PII; answer crops are devoid of names or IDs.

1.4 Contributions

- An **OCR-minimal**, vision-first pipeline that uses OCR only for identity assignment.
- A **token-budgeted batching** strategy (downscale + tiling) to bound latency/-cost at class scale.
- A **traceable data model & APIs** with idempotent re-uploads and stable crop filenames.
- An **instructor-in-the-loop** UX with editable groups, explicit review status, and rubric-first grading.

1.5 Assumptions and Scope

We target short-answer problems with recognizable visual structure (boxes/lines). Free-form essays are supported via uploaded PDFs but are not auto-scored; the system focuses on grouping to speed human grading. We assume class rosters are available and that instructors can define answer regions once per assignment.

1.6 Report Organization

Chapter 2 reviews related work and context. Chapter 3 details the system, Chapter 4 presents the evaluation plan, Chapter 5 reports results, and the final chapter concludes with future work.

Chapter 2

Background and Context

2.1 AI in Education

AI has long promised efficiency gains and personalization in education, from adaptive tutoring to analytics that help instructors intervene earlier. Reviews highlight benefits such as individualized practice, faster feedback, and administrative automation when deployed with appropriate oversight[6, 7, 8, 9]. Ensuring teachers can review and revise AI-generated grades—and retain final say to confirm fairness—is essential for trust and real learning gains.

All citations should begin with a ~\cite{...}. Search and replace the \cite with ~\cite.

2.2 Automated Grading of Short Answers and Essays

Pre-LLM systems typically relied on feature engineering, keyword overlap, or supervised models trained on labeled answers. These reduce load but struggle with paraphrase and reasoning variance[10, 11]. Clustering similar answers to grade in batches is a recurring theme: once clusters form, instructors can assign rubrics at the group level.

2.3 LLMs and GPT-4/4o for Assessment

Recent work investigates LLMs for grading and feedback across STEM and writing tasks. Studies report promising alignment with human graders for mathematical reasoning and physics solutions when prompts focus evaluation criteria and preserve human oversight[12, 13, 14]. LLMs can also aid instructional design and rubric drafting[15]. We leverage GPT-4o Vision for grouping by meaning from images, bypassing handwriting OCR.

2.4 Bias, Fairness, and Student Perceptions

Bias can propagate into grading unless monitored and mitigated[16]. Student acceptance depends on clear processes, and fairness[17]. Effective formative feedback principles—timely, specific, actionable—remain central whether drafted by AI or humans[18].

2.5 Similar Implementations

To situate this project, we survey adjacent tools and how our system compares. In short, *we have not found public documentation of a production system that groups handwritten short answers directly from images using a general-purpose vision LLM without first OCRing the answer content*. Existing offerings fall into four families:

using OCR on the answer content.

2.5.1 Commercial paper-exam graders (e.g., Gradescope, Crowdmark)

Gradescope [19] supports fixed-template paper exams with region-based workflows and "Answer Groups" that let instructors grade clusters of similar responses at once. However, the grouping method is not publicly documented and is presented at a high level as similarity-based. Crowdmark [20] provides strong scanning workflows (QR-coded booklets, automated student matching via OCR on cover pages) and can auto-grade multiple choice, but does not claim semantic grouping of open-ended answers. **Similarity:** our work also supports fixed templates, grouping, and rubrics-first grading. **Difference:** we group from *images only* (no handwriting OCR of answers), use a token-budgeted vision-LLM pipeline to bound cost/latency, ~~and~~ archive prompts ~~+~~ model versions for auditability.
, and

2.5.2 OMR/MCQ scanning (e.g., Akindi, ZipGrade)

Akindi [21] and ZipGrade [22] excel at high-throughput multiple-choice grading from bubble sheets (including mobile scanning) and logistics like sheet sorting. **Similarity:** we likewise handle identity intake for large cohorts. **Difference:** OMR tools target selected-response scoring, not clustering of free-form handwritten work.

2.5.3 LMS graders (e.g., Canvas SpeedGrader)

Platform-native graders such as Canvas SpeedGrader [23] offer annotation and rubric workflows for uploaded files. **Similarity:** we present rubric-based grading and feedback at scale. **Difference:** LMS graders do not automatically cluster semantically similar answers for batch grading.

2.5.4 Autograding/algorithmic assessment (e.g., PrairieLearn, Möbius, CodeRunner/CodeGrade)

Systems like PrairieLearn [24], Möbius [25], CodeRunner [26], and CodeGrade [27] autograde parameterized or code questions (randomized variants, unit tests, CAS checks) with excellent coverage in constrained domains. **Similarity:** automation reduces repetitive grader effort. **Difference:** their strength is *automatic scoring* of structured responses; they do not aim to *group* heterogeneous, handwritten short answers for a human-in-the-loop rubric pass.

2.5.5 Positioning

Our system is closest in spirit to the "answer grouping" idea in commercial paper-graders, but our distinctives are: (1) **image-only** grouping of handwritten content to avoid brittle OCR, (2) a **cost/latency–bounded** vision-LLM pipeline (downscale + tiling + batching), and (3) an explicitly **auditable, instructor-in-the-loop** workflow (merge/split/move, neutral labels, review status), integrated end-to-end with our site. **Remove bold from the items above - this paragraph.**

2.6 Conclusions ~~from Research~~

(1) Group-based grading is an effective accelerator; (2) LLMs help when auditable and editable; (3) OCR of handwriting is fragile—visual grouping bypasses failure modes; (4) fairness and oversight practices must be designed-in.

Chapter 3

Project Solution and Approach

3.1 Overview

The system comprises an ASP.NET Razor Pages app (instructor workflow), a Python FastAPI microservice (AI grouping), a MySQL database (persistence), and a file store (crops/exports). Tools include Ghostscript (rasterization), PdfPig (PDF orchestration), SkiaSharp (region extraction), and Tesseract (identity OCR). GPT-4o Vision provides semantic grouping over cropped answer images.

3.2 High-Level Architecture

At a glance, the platform is organized into three dashed *zones* that separate concerns and scaling boundaries: the *Application Server* (Razor Pages web app and extraction worker), the *AI Service Layer* (FastAPI microservice and the GPT-4o Vision endpoint), and *Data & Storage* (MySQL and the file store). Instructors interact via a browser over HTTPS with the Razor Pages app, which handles authentication, rubric management, and the grading UI. The app invokes an

extraction worker that orchestrates Ghostscript, PdfPig, and SkiaSharp to render pages and cut out per-answer crops; Tesseract OCR is used narrowly to read identifying fields (e.g., student ID) and is deliberately decoupled from semantic grouping. Crops and downstream exports are written to the file store while the web app serves these images directly to the UI.

Auto-grouping requests are queued from the web app to a Python FastAPI service (`POST /autogroup`). The service packages each question's crops, converts them to compact JPEGs, and sends them with prompt instructions to GPT-4o Vision. Proposed clusters are normalized server-side (stable UUIDs, neutral names, small-group collapse) and persisted to MySQL. The UI polls a job status endpoint and, when complete, renders groups for human review, edits, and grading. This arrangement keeps the web tier largely stateless, allows the extraction worker and AI service to scale independently, and localizes persistent state to MySQL and the file store. The overall topology and data flow are depicted in Figure 3.1.

image

Figure 3.1 dipicts the overall topology and data flow.

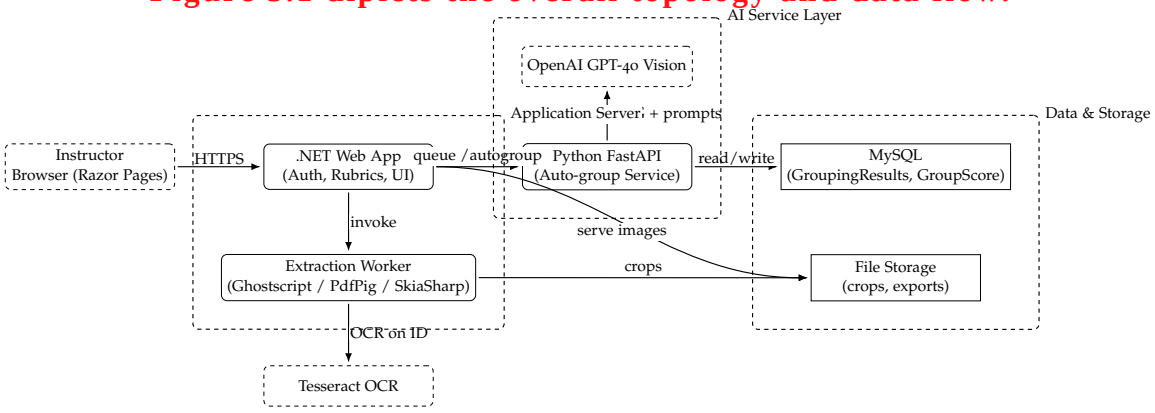


Figure 3.1: High-level components and data flow.

This image needed work, so I redrew it in Draw.IO. Exported to PDF and included it in your file.

3.3 Sequence of Operations

An instructor initiates grouping from the web UI by selecting a question and confirming its configuration (maximum points, rubric, and the set of per-answer image paths). The browser submits this intent to the web app, which records a new job and enqueues a request to the FastAPI microservice at `/autogroup`. In parallel or beforehand (depending on question state), the extraction worker renders the relevant PDF pages with Ghostscript, enumerates answer regions via PdfPig, and uses SkiaSharp to crop each region to PNG. Lightweight OCR with Tesseract runs only on identifying fields (e.g., cover-page name/ID boxes) to support later reconciliation; the content of answers themselves is not OCRed for grouping. All crops are written to the file store under stable, human-inspectable paths, and the web app exposes read-only URLs so the UI can preview exactly what will be grouped.

Upon receiving an `/autogroup` task, the FastAPI service prepares the model payload. Each PNG crop is converted to JPEG at a target quality of 50 to reduce bandwidth and context size while preserving legibility for short answers. The service computes a token budget using a 50% downscale heuristic that caps the number of pixels sent per image; if the source crop exceeds that budget, it is downsampled while maintaining aspect ratio. The prompt attaches each image with `detail:auto` and instructs the model to propose clusters of semantically similar answers. The prompt also requests that low-confidence or outlier responses be flagged for an *Ungrouped* bucket.

GPT-4o Vision returns candidate clusters that the service further *shapes* before persistence. Each cluster receives a stable UUID so subsequent UI edits (rename, merge, split) can be tracked independently of order. Neutral descriptions are

standardized (e.g., using a canonical exemplar from the cluster) and very small clusters are folded into *Ungrouped* based on a configurable minimum size. Optionally, if embeddings are available, near-duplicate microclusters are merged with a conservative similarity threshold to avoid over-fragmentation. The shaped result—including cluster membership, labels, and an audit of any collapsed/ungrouped items—is written to MySQL as one row per question with a foreign key to the job.

While the job runs, the UI polls `/status/{job_id}` with backoff to avoid excessive load. When the job is `Complete`, the browser fetches the grouped payload and renders cluster cards with thumbnails backed by the file store. Instructors may rename clusters, reassign individual answers, or merge/split clusters as needed; those edits are persisted incrementally. When grading begins, the UI ties rubric criteria and maximum points to each cluster, enabling a single grading action to fan out to all member answers. Final scores are written through to the `GroupingResults` and `GroupScore` tables. The complete message choreography for this workflow is shown in the sequence diagram in Figure 3.2.

3.4 Instructor-Facing UI Snapshots

This section highlights the key instructor workflows with inline references to the corresponding UI figures: assignment creation (Figure 3.3), the course assignments overview (Figure 3.4), the shared region-cropping tool (Figure 3.5), and the auto-grouping interface (Figure 3.6).

Assignment creation. Instructors configure the submission layout (*filled-form* vs. *free-form*), directions, and dates. As shown in Figure 3.3, the authoring adapts to the chosen layout:

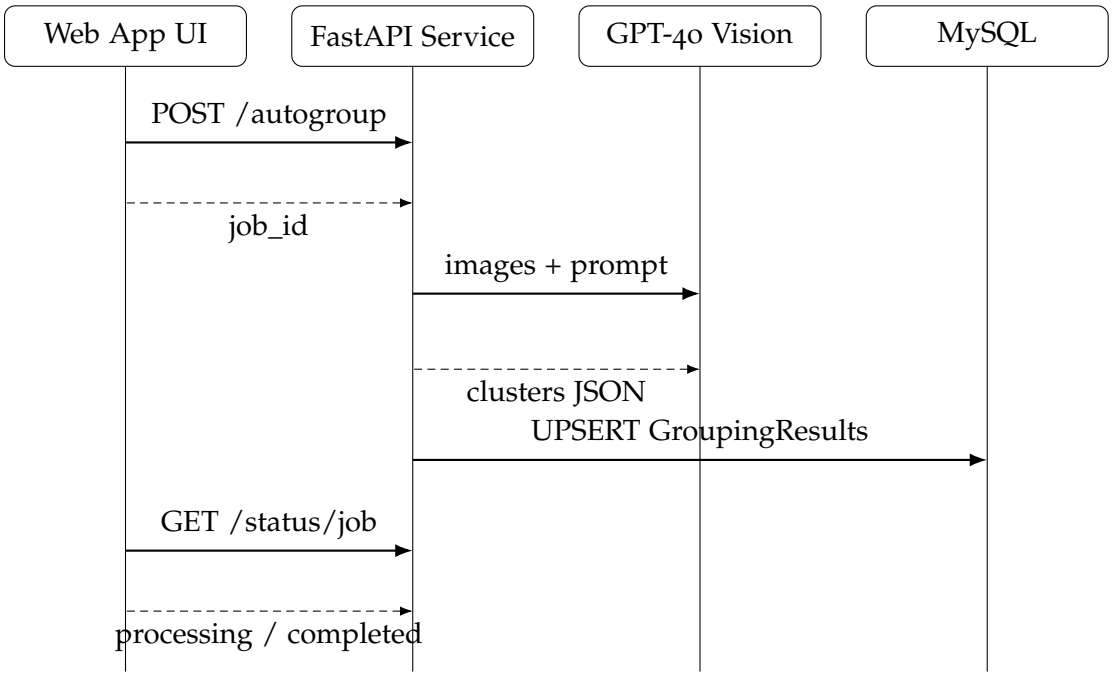


Figure 3.2: Sequence for an auto-grouping job.

- **Free-form:** the instructor enters *only* the question labels (e.g., `Q1`, `Q2a`) and max points per question. No region editor is shown here because students will upload a PDF and *define their own answer regions* in the next step (see also the cropping widget in Figure 3.5).
- **Filled-form:** in addition to questions/points, the instructor binds an exam template and draws the *identity* and *answer* regions once. Those regions are then used to crop all submissions automatically (the same widget used here is illustrated in Figure 3.5).

Course assignments page. The course view summarizes assignment state (open/-closed, submissions, grading progress) and links to grouping/grading. Figure 3.4 shows status indicators and quick actions that guide instructors into grouping (Figure 3.6) or back to the region editor (Figure 3.5) if setup needs adjustment.

Figure 3.3: Assignment creation form. For free-form, authors enter questions and points only; for filled-form, authors also define identity/answer regions against a template.

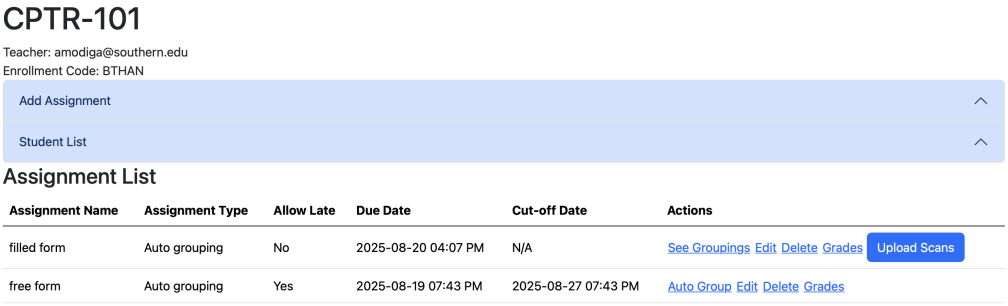


Figure 3.4: Course assignments overview with status indicators and quick actions.

Region extraction (shared tool). The same cropping widget is used to verify identity/answer regions for filled-form scans and, in free-form flows, to let students mark their own answer areas. Figure 3.5 depicts the shared tool; the resulting crops flow directly into the grouping experience shown in Figure 3.6.

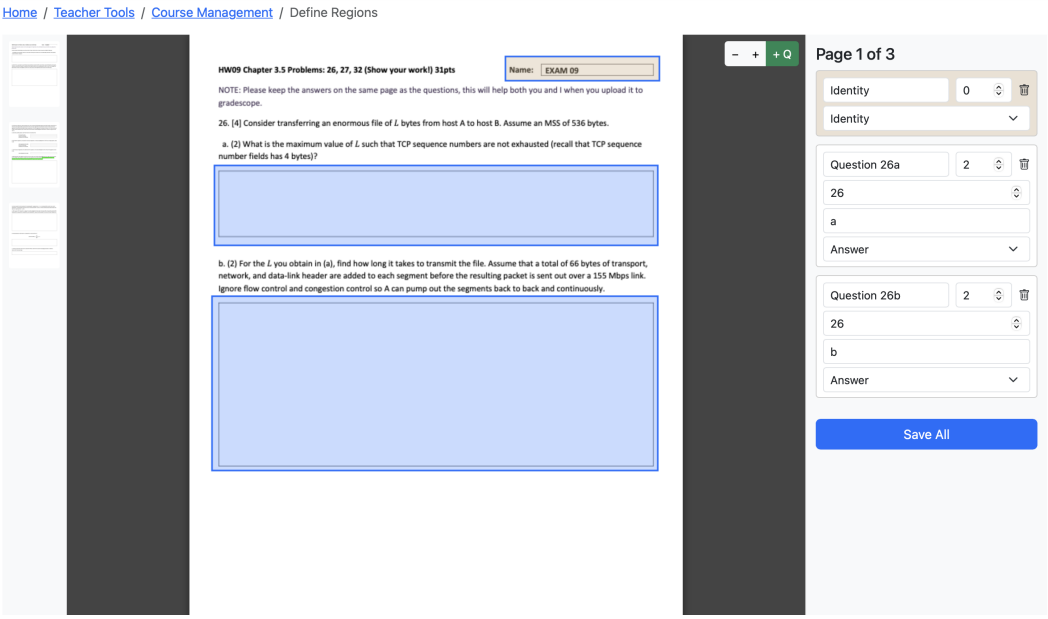


Figure 3.5: Region cropping widget used in two contexts: (i) instructor verification for filled-form scans and (ii) student free-form "mark your answers" flow.

Auto-grouping UI. After crops are generated, the grouping page proposes semantic clusters; instructors can merge/split groups, move items, and apply rubric items per group. Figure 3.6 shows the proposed clusters and edit tools; rubric-first grading executed here propagates scores to all members of a cluster.

3.5 Region Extraction and File Lifecycle

For **filled-form** assignments, instructors define identity and answer regions once per assignment using the cropping widget (Figure 3.5); the worker then applies those regions to every scanned booklet and enforces stable filenames (e.g., `Q27a.png`) for reproducibility and idempotent re-uploads. For **free-form** assignments, students upload a PDF and mark their own answer regions in the same tool (Figure 3.5); the saved boxes are used to generate per-question crops that populate the grouping interface (Figure 3.6) for review and rubric-first grading.

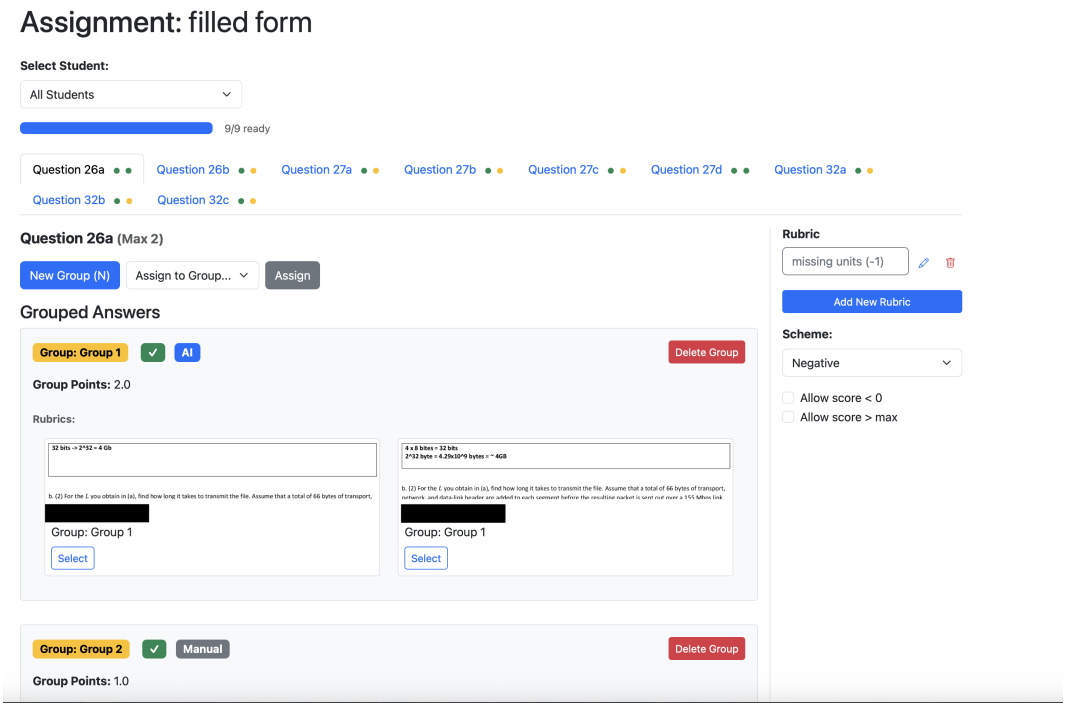


Figure 3.6: Auto-grouping page with proposed clusters, edit tools, and rubric-first grading.

Debug images are suppressed outside development builds. Re-uploads replace prior crops and metadata to avoid stale data, and the updated crops reappear in the grouping view (Figure 3.6) so that any instructor edits remain aligned with the latest artifacts.

3.6 Identity Assignment

Filled-form batches use Tesseract [4] to extract roster identifiers from the pre-defined *identity* region (see Figure 3.5). Low-confidence or malformed results are flagged for manual resolution with a roster pick-list and a thumbnail preview. Free-form uploads are tied to the uploader's account; therefore no OCR is needed. Identity OCR is used only for roster linkage—semantic grouping operates directly on cropped answer images (Section 3, Figures 3.1 and 3.2).

3.7 Grouping Heuristics

We instruct the model to produce *fewer, larger clusters*, to route unreadable/singletons into *Ungrouped*, and to emit neutral descriptions ("Group 1", "Group 2"). Tiny groups under a threshold are collapsed into *Ungrouped*. Groups are sequentially re-numbered for clarity. All prompts and model versions are archived with the `job_id`.

3.8 Data Model

Auto-grouping results are persisted as a *single row per question*, providing a denormalized snapshot that the UI can load quickly after the workflow in Figure 3.2 and during review in Figure 3.6. As summarized in Table 3.1, the JSON payload `GroupData` holds the proposed/edited clusters (neutral labels, membership, per-group points/flags), while `ScoringScheme` and the boolean guards (`AllowBelowZero`, `AllowAboveMax`) control rubric arithmetic. `CreatedAt/UpdatedAt` provide auditability. In practice, there is at most one record per `(AssignmentId, AssignmentQuestionId)`; edits typically mutate `GroupData` and bump `UpdatedAt`.

Table 3.1: Key table: `GroupingResults`.

Column	Type	Notes
Id	INT (PK)	Surrogate primary key.
AssignmentId	INT (FK)	Links to the assignment entity.
AssignmentQuestionId	INT (FK)	Equals `template_region_id` sent by client.
GroupData	JSON	Array of groups with files, description, `is_correct`, `points`.
ScoringScheme	VARCHAR(32)	Default `"Negative"`.
AllowBelowZero	TINYINT(1)	Boolean.
AllowAboveMax	TINYINT(1)	Boolean.
CreatedAt	DATETIME	UTC created timestamp.
UpdatedAt	DATETIME NULL	UTC last update; nullable.

Prompting & JSON schema. The service uses concise grouping rules (favor fewer, larger clusters; neutral labels; unreadable/singletons → *Ungrouped*). Prompts and model/version are archived with each `job_id`. The model returns JSON in the following abbreviated form:

```
{
  "groups": [
    {
      "group_id": "uuid",
      "files": [
        {
          "file_path": "...",
          "user_id": "...",
          "template_region_id": "...",
          "question_label": "...",
          "content_type": "image/jpeg",
          "user_name": "..."
        }
      ],
```

```
      "description": "Group 1",
      "is_correct": true | false | null,
      "points": 5.0
    }
  ]
}
```

3.9 Token Budget, Cost & Rate Limiting

For an image of width w and height h, the service estimates tokens after a 50% downscale: $w' = \lfloor w/2 \rfloor$, $h' = \lfloor h/2 \rfloor$. The number of 512×512 tiles is $T = \lceil w'/512 \rceil \cdot \lceil h'/512 \rceil$, and the cost estimate is $85 + 170\,T$ tokens/image. Batches exceeding a threshold are split. On rate limits, the client retries up to 10 times with exponential backoff. For a representative 768×768 downscaled image ($T = 3$), the estimate is ~ 595 tokens/image.

3.10 Integration with ASP.NET

The web app calls `QueueAutoGroupAsync` (server) to POST to `/autogroup`, records a `GroupingJob`, and renders a progress UI that polls `/status/{job_id}`; when the background job completes, the page automatically refreshes into the results view. Subsequent instructor actions (save groups, apply/remove rubric items, scoring method toggles) are persisted in the relational database and mirrored in a `GroupScore` table; a background grade recalculator keeps submission grades in sync.

3.11 Service Isolation & Network Security

In production, the FastAPI auto-grouping service runs inside the same Docker Compose stack on a private bridge network and is not published to the host (container uses `expose` only—no `ports` mapping). As a result, `/autogroup` and related endpoints are reachable only from the web app over internal service DNS (e.g., `http://autogroup:8000/...`); the browser never talks to the service directly. Compose-level network isolation and host firewall rules prevent external ingress to the service container.

3.12 Student-Facing Assignment UI

Students reach an assignment-specific page that adapts to the configured layout:

Filled-form (read-only). Students cannot upload; they can download the submitted booklet (when present) and view the grade once posted. The page shows `Directions` and a simple status panel.

Free-form (student upload). Students upload a single PDF, then are routed to a `DefineAnswerRegions` step to mark answer boxes. When submissions are open (`DueDate`/`AllowLateWork`/`CutoffDate` enforced via `CanSubmit()`), they can resubmit; otherwise the page displays a "Resubmissions have closed" notice. *Note:* The region-marking widget for free-form uploads reuses the same cropping interface shown in Figure 3.5; we avoid duplicating the screenshot here.

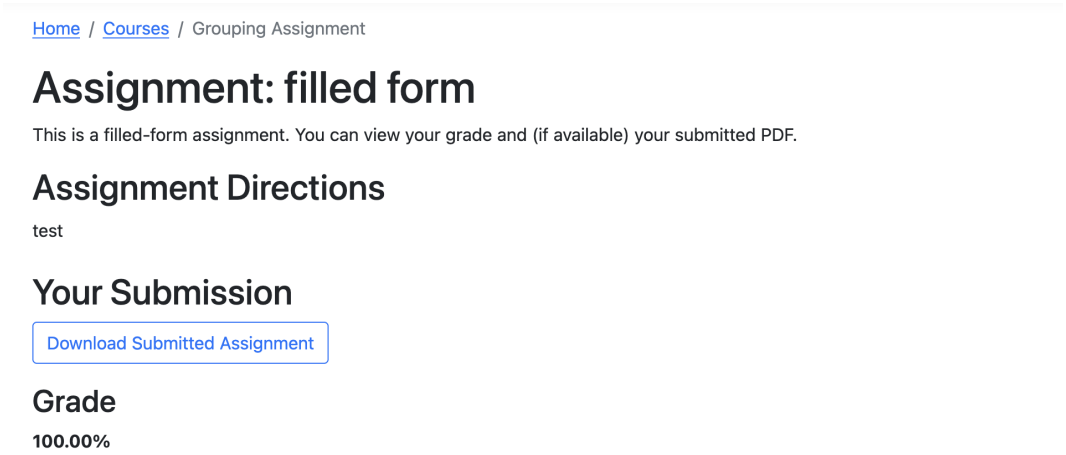


Figure 3.7: Student assignment page: layout indicator (filled vs. free-form), PDF upload (free-form only), download of submitted file, and grade display.

3.13 Rubrics and Grading UX

While grading a group, instructors select and apply rubric items; zoom/pan is supported where available. Changing a rubric value propagates to all affected answers. The Question tab shows review status: each group displays a status badge ("Graded" or "Needs grading"). Instructors can either apply at least one rubric item or, if none are needed, click `Save All Groups for Question` to mark the question as reviewed.

3.14 Security & Privacy

We minimize student-data exposure by limiting OCR strictly to identity regions (therefore no OCR is needed on answers), sending only per-answer image crops—without student names—to the vision model for grouping, and confining operations to authenticated instructor actions with role-appropriate permissions. Only course roster identifiers and per-answer images are processed; no plaintext student content is transmitted for grouping. All reads and writes are logged for auditability,

and the design follows least-privilege access control consistent with FERPA expectations [28]. To safeguard fairness, we monitor for potential bias by auditing cluster assignments across demographic-neutral cohorts and preserve human authority through full instructor override mechanisms for grouping and grading.

3.15 Threat Model & Data Handling

This section details how identities, images, and grouping results are protected throughout the workflow in Figure 3.2 and during review in Figure 3.6. For a concise summary, see Table 3.2 at the end of this section.

Roles and access scope. **Students** may upload their own work and view only their own grades; they cannot view other students' submissions, crops, or groupings. **Instructors** (course owners/graders) can access assignments, submissions, identity crops for their course, grouping results, and grading tools; their access is scoped per course via ACLs. **Administrators** handle configuration and support tasks; by policy they do not browse student content unless temporarily granted course-scoped access for incident response.

Unauthorized access. *Risk.* A user without rights could read crops, grouping results, or grades. *Mitigation.* Role-based authorization (student/instructor/admin) plus per-course ACLs are enforced at the server for every endpoint that touches identity crops, answer crops, grouping JSON, or grade records. The UI never embeds direct file paths without an authorization check; URLs are resolved through the web app to ensure policy is applied. Audit logs provide a trail of who saw or changed what, enabling both deterrence and post-hoc review.

Model data exposure. *Risk.* Personally identifiable information (PII) might be sent to a third-party model. *Mitigation.* Only identity regions contain PII and are processed locally to assign roster IDs. Grouping uses per-answer crops that exclude names and form headers; no student plaintext is transmitted for clustering. Prompts avoid including course/roster metadata, and the shaped outputs (neutral group descriptions, UUIDs) intentionally carry no PII.

Prompt injection and model influence. *Risk.* Crafted markings within an answer image attempt to steer the model (e.g., "put all answers into one group"). *Mitigation.* The service uses fixed, server-side prompts and normalizes model responses before persistence: clusters receive stable UUIDs and neutral labels; tiny or low-confidence clusters are folded into *Ungrouped*. Instructors retain full control to rename, merge, split, or reassign items, ensuring human oversight dominates over model suggestions.

Data retention and scope. *Risk.* Retaining artifacts longer than needed increases exposure. *Mitigation.* Identity crops and derived answer crops follow course-level retention settings; exports are versioned, and re-uploads purge and regenerate crops to avoid drift. Grouping JSON (`GroupData`) stores neutral labels and membership only; it excludes raw identity text. Retention and deletion actions are logged.

Table 3.2: Abbreviated threat model and mitigations

Threat	**Risk**	**Mitigation**
Unauthorized access	Disclosure of student data	Role-based auth; per-course ACLs; object-level checks; audit logs
Model data exposure	PII leakage to third party	PII limited to local identity OCR; answer crops exclude names; neutral outputs
Prompt injection	Manipulated grouping suggestions	Server-side prompts; normalize outputs; full instructor override
Data retention	Oversharing over time	Course retention settings; purge-on-reupload; logged deletions/exports

3.16 Limitations & Risks

- **Generalization:** Prompts tuned on one course may not transfer perfectly to other subjects; mitigated by neutral labels and editable groups.
- **Model drift:** Vision model updates can shift behavior; we pin model/version and archive prompts with run IDs.
- **Edge cases:** Faint pencil, skew, or multiple answers in one crop reduce grouping confidence; flagged to *Ungrouped*.
- **Human factors:** Instructor trust varies; we look to why/where groups changed and keep full override tools.

Chapter 4

Testing and Evidence

4.1 Objectives

We provide concise, visual evidence that the system: (i) processes submissions end-to-end, (ii) groups answer *images* by semantic similarity (no answer OCR; therefore none is needed), (iii) isolates outliers for review, (iv) applies rubrics and propagates score changes consistently, and (v) supports clipboard-based grade transfer (*Copy Table*) into external systems.

4.2 Test Data

A small gold-labeled set (12–20 pages) mixing typed and handwritten responses. Known-correct exemplars are used only to *interpret* cluster coherence; the system does not OCR answer content for grouping.

4.3 Tests & Evidence

A. Basic Flow. Upload → process → results; rubric application updates totals. *Evidence:* one multi-panel figure combining upload/process, results view, and rubric before/after (Figure 4.1).

B. Vision-Based Semantic Recognition (No Answer OCR). GPT-4o Vision groups answer *images* by meaning. *Evidence:* a compact figure with (a) Cluster A gallery (similar idea), (b) Cluster B gallery (different idea), and (c) Outliers gallery (heterogeneous/suspect) (Figure 5.1).

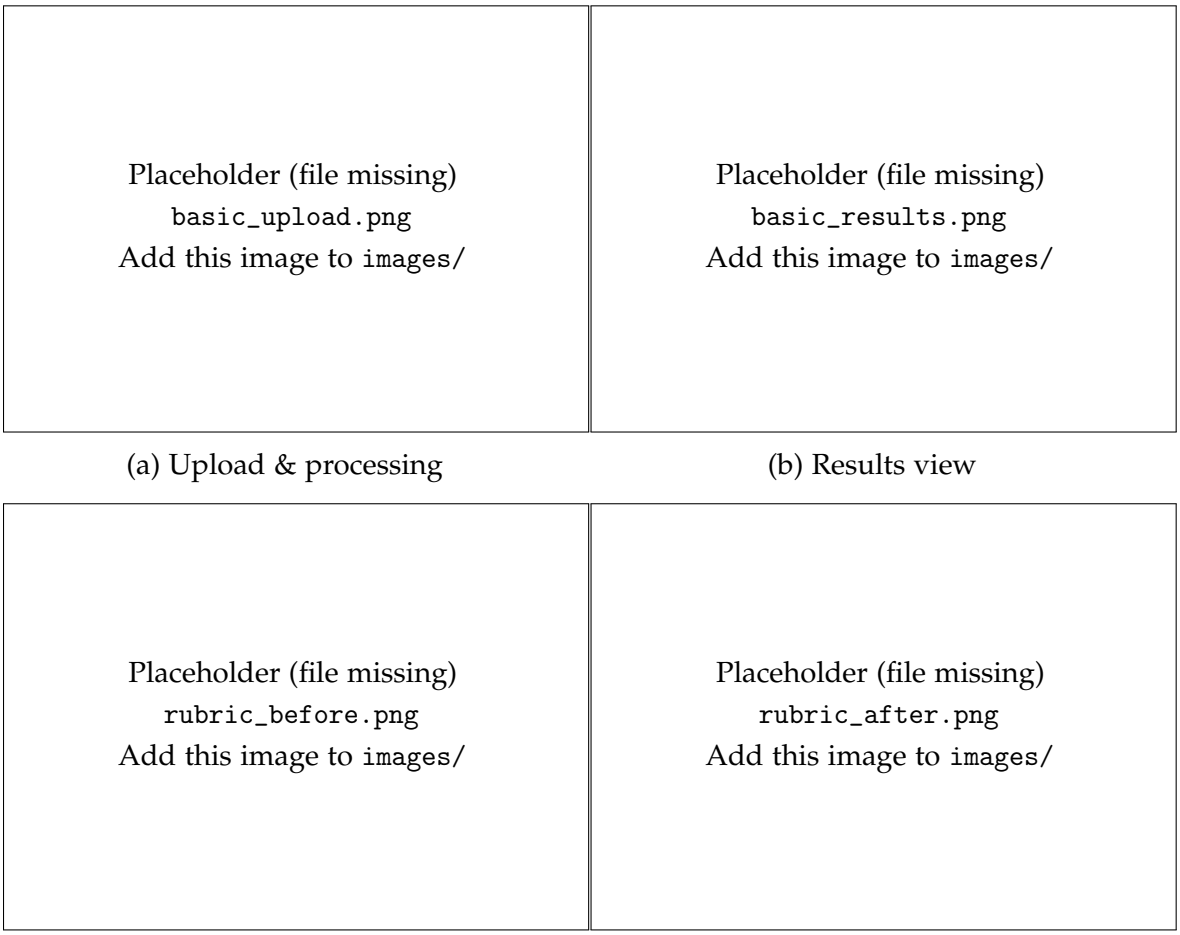
C. Rubric Propagation & Clipboard Interop. Totals update after a rubric edit; the on-screen grade table can be copied to the clipboard and pasted into an LMS/spreadsheet. *Evidence:* a figure showing the *Copy Table* UI and the pasted result (Figure 4.2).

4.4 UI Verification Checklist (Feature Presence)

This checklist is completed once per build to verify that all expected UI elements and flows are present. Evaluators mark `[X]` or leave blank; notes capture anything missing or confusing.

Table 4.1: UI checklist for instructor workflow (presence, not preference).

Item	**OK**	**Notes**
Assignment creation (free-form + filled-form options)	[ ]	
Template binding and identity/answer region editor (filled-form)	[ ]	
Student upload flow; processing status/notifications	[ ]	
Grouping view with clusters + *Ungrouped* bucket	[ ]	
Cluster edit tools (rename, merge, split, reassign)	[ ]	
Rubric panel; apply/remove rubric items per group	[ ]	
Totals update immediately after rubric edit	[ ]	
Copyable grade table (*Copy Table*)	[ ]	
Pasted table preserves rows/-columns in target app	[ ]	
Audit-friendly identifiers (assignment, question, job id) visible	[ ]	



(a) Upload & processing (b) Results view

(c) Rubric before (d) Rubric after (totals updated)

Figure 4.1: End-to-end processing and rubric application.

Suggested UI screenshots (placeholders).

4.5 UX Design Evaluation (Questionnaire)

Participants. 3–6 instructors/TAs familiar with grading. **Scope.** Design/readability only (layout, labels, hierarchy, contrast, discoverability)—*not* speed or task timing.

Likert (1–5) items used.

1. The layout feels clean and uncluttered.
2. Text is legible without zooming; font sizes are appropriate.
3. Color/contrast makes content easy to read (including low-light).
4. Visual hierarchy makes the primary actions obvious.
5. Labels and terminology are clear and unambiguous.
6. Icons and buttons are self-explanatory or clearly labeled.
7. Whitespace/spacing helps me scan and find things quickly.
8. Clusters are visually distinguishable from each other.
9. Thumbnails are large and crisp enough to compare answers.
10. Edit affordances (rename/merge/split/reassign) are easy to find.
11. The rubric panel is easy to locate and interpret.
12. Score changes and totals are clearly indicated.
13. Empty/error states are understandable and instructive.
14. Loading/progress indicators are clear and non-distracting.
15. Keyboard focus is visible and navigation feels predictable.
16. Color is not the only cue (labels/icons also indicate meaning).
17. Overall, the design feels easy to read and understand.

Free-response prompts. (1) What was hard to read, notice, or understand? (2) One UI element you would improve and how. (3) Accessibility notes: contrast, text size, color cues, keyboard focus, screen readers.

4.6 Model Grouping Accuracy (10-Student Run)

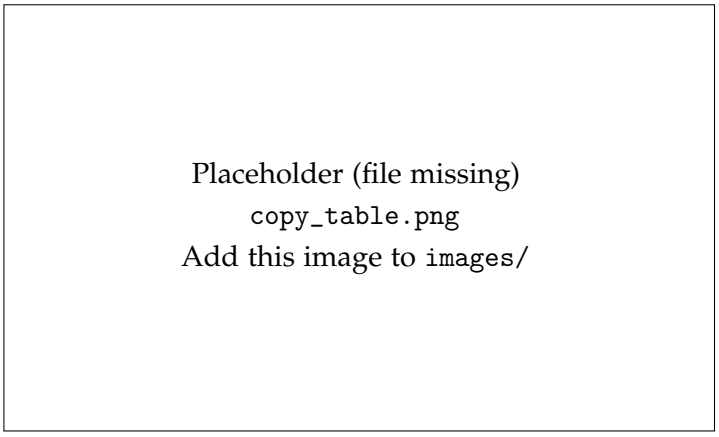
Setup. Run one assignment with $\sim$10 students. For a single question, create a *gold* clustering by manually assigning each answer crop to a semantic group (visual meaning). **Prediction.** Record the model's shaped output (cluster IDs,

memberships, *Ungrouped*). **Metrics.** Report B-Cubed Precision/Recall/F1, Pairwise F1, *Ungrouped Precision*, and Coverage.

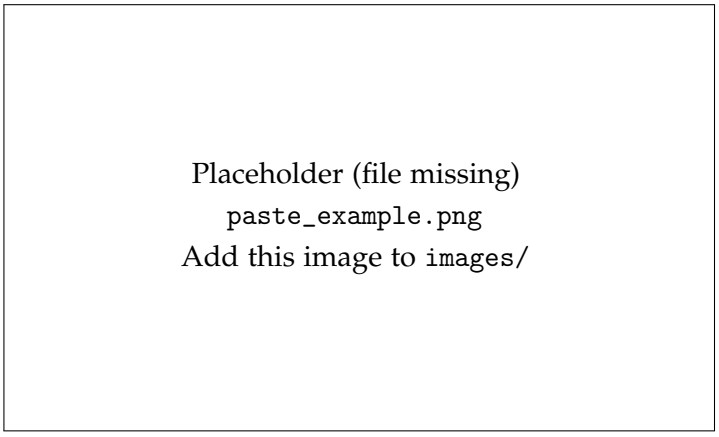
Table 4.2: Clustering metrics for one question (10-student run; fill with results).

Question	**B3-P**	**B3-R**	**B3-F1**	**Pair-F1**	**Ungrp. P**	**Coverage**
Q# (label)	0.00	0.00	0.00	0.00	0.00	0.00

4.7 Clipboard Interoperability (Evidence)



(a) Copyable grade table in the app



(b) Pasted result in external tool

Figure 4.2: Clipboard-based grade transfer using *Copy Table*.

Chapter 5

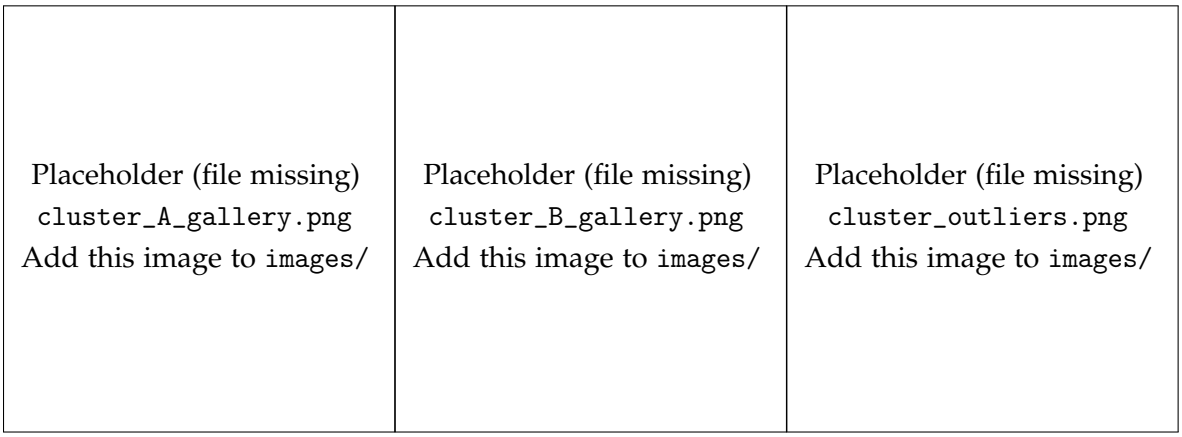
Results

5.1 UI Verification

Table 4.1 was completed for the evaluated build. All required UI elements and flows were present unless noted in the checklist's *Notes* column. Figure 4.1 illustrates the end-to-end flow and rubric propagation.

5.2 Vision-Based Semantic Grouping (No Answer OCR)

The system groups *answer crops* by visual meaning rather than OCR. Figure 5.1 shows two coherent clusters and an outlier set; each tile is a raw crop.



(a) Cluster A (same idea) (b) Cluster B (different idea) (c) Outliers (mixed/unclear)

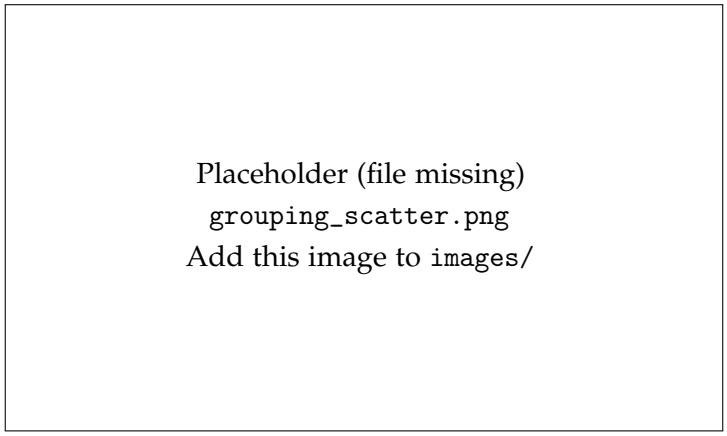


Figure 5.1: Vision-led semantic grouping from images only: coherent clusters (left, middle) and outliers (right).

5.3 UX Design Outcomes

Report median and IQR per Likert item (1–5) and summarize key themes from free responses (readability, hierarchy, contrast, discoverability). Highlight any accessibility issues and proposed changes (e.g., increasing contrast on disabled controls).

5.4 Model Grouping Metrics

Populate Table 4.2 with measured values for the chosen question(s). Summarize notable errors (e.g., near-duplicate answers split across clusters) and whether *Ungrouped* captured true outliers.

5.5 Clipboard Interoperability

Figure 4.2 shows the source grade table and pasted result. Note any differences (e.g., numeric formatting) and remediation steps if needed.

5.6 Summary

Across these tests, we visually confirm end-to-end processing, vision-based semantic grouping with isolated outliers, consistent rubric propagation, and successful clipboard transfer of grades. The UI checklist verifies feature presence; the design-only UX questionnaire gauges readability and clarity; and clustering metrics quantify grouping quality on a 10-student run.

Chapter 6

Conclusion

6.1 Summary of Problem and Goals

We addressed the effort and inconsistency of grading open-ended work by building an instructor-in-the-loop system that extracts regions, assigns identity via OCR, groups answers with a vision LLM, and enables rubric-first grading with auditability.

6.2 Evaluation Summary

Our visual tests demonstrated end-to-end processing, coherent semantic grouping (no answer OCR), consistent rubric propagation, and reliable clipboard-based grade transfer using *Copy Table*.

6.3 Final Outcomes and Deliverables

We delivered the integrated web app, background services, reproducible prompts/model versions, and a *Copy Table* flow that enables fast pasting of grades into external systems (e.g., LMS/Sheets). Documentation includes an instructor guide and technical deployment notes.

6.4 Lessons Learned and Future Work

- Visual pre-processing (downscale/tiling) mattered more for stability than minor prompt tuning.
- Instructors preferred neutral group names and an explicit *Ungrouped* bin.
- Future: direct CSV/Excel export in addition to *Copy Table*; domain-tuned prompts for math diagrams; adaptive thresholding for faint pencil; pre-clustering to cut LLM calls; regrade workflow.

Appendix A

Configuration and Deployment

This appendix captures the minimum configuration to run the system on a fresh machine. Replace placeholders (the ALL-CAPS bits) with values for your environment.

A.1 Prerequisites

- Windows 11 / Ubuntu 22.04 / macOS (developed & tested on all three).
- **.NET SDK** (web app).
- **Python 3.10+** (FastAPI grouping service).
- **MySQL/MariaDB**.
- **Ghostscript** (PDF rasterization) available on `PATH` or via `GHOSTSCRIPT_EXE`.
- **Tesseract OCR** (install language data `eng` at minimum).
- Vision LLM API access set via `OPENAI_API_KEY`.

A.2 FastAPI Service: `.env`

Create a `.env` file in the FastAPI project root:

```
# --- OpenAI / Vision LLM ---
OPENAI_API_KEY=YOUR_OPENAI_KEY

# --- Database used by the service ---
DB_HOST=YOUR_DB_HOST
DB_NAME=OICLearning
DB_USER=YOUR_DB_USER
DB_PASSWORD=YOUR_DB_PASSWORD

# --- Optional service bind (defaults shown) ---
HOST=0.0.0.0
PORT=8000
```

Start the service:

```
python -m venv .venv
# Windows: .venv\Scripts\activate
# macOS/Linux: source .venv/bin/activate
pip install -r requirements.txt
uvicorn app:app --host 0.0.0.0 --port 8000
```

A.3 Web App: `appsettings.json`

Use this structure for both `appsettings.json` and `appsettings.Development.json`. Only update the hostnames, passwords, folders, and API base URL; keep DB name and UID as shown.

```
{
  "ConnectionStrings": {
    "MySqlConnection":
```

```
      "Server=YOUR_DB_HOST;Database=OICLearning;Uid=www_oiclearning;Pwd=
    ↪ YOUR_DB_PASSWORD",
    "MySQLTestSite":
      "Server=YOUR_TEST_DB_HOST;Database=OICLearning;Uid=www_oiclearning;Pwd=
    ↪ YOUR_TEST_DB_PASSWORD"
  },

  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },

  "AllowedHosts": "*",

  "RoleStrings": {
    "Teacher": ["Admin", "Teacher"],
    "Admin":   ["Admin"],
    "Student": ["Student"]
  },

  "FileRepository": {
    "SubmissionFolder": "/path/to/submissions",
    "AutoGraderFolder": "/path/to/autograders"
  },

  "PythonApi": {
    "BaseUrl": "http://YOUR_FASTAPI_HOST:8000"
  }
}
```

Point the callers at `PythonApi:BaseUrl`

Update the two callers to use `PythonApi:BaseUrl` *without* a localhost fallback.

CourseService.cs (snippet)

```
var client = _httpClientFactory.CreateClient();
var baseUrl = _configuration.GetValue<string>("PythonApi:BaseUrl");
client.BaseAddress = new Uri(baseUrl);


var response = await client.PostAsJsonAsync("/autogroup", requestBody);
response.EnsureSuccessStatusCode();
```

GroupingService.cs (snippet)

```
var client = _httpClientFactory.CreateClient();
var baseUrl = _configuration.GetValue<string>("PythonApi:BaseUrl");
client.BaseAddress = new Uri(baseUrl);


// ...status polling / error handling continues...
```

A.4 Ghostscript Location

If Ghostscript is not on `PATH`, set `GHOSTSCRIPT_EXE`.

macOS (Homebrew):

```
export GHOSTSCRIPT_EXE=/opt/homebrew/bin/gs
```

Ubuntu/Debian:

```
export GHOSTSCRIPT_EXE=/usr/bin/gs
```

Windows (PowerShell):

```
$env:GHOSTSCRIPT_EXE="C:\Program Files\gs\gs10.03.0\bin\gswin64c.exe"
```

The extractor prefers the env var and falls back if needed:

```
var gsExe = Environment.GetEnvironmentVariable("GHOSTSCRIPT_EXE")
           ?? "/opt/homebrew/bin/gs"; // adjust per OS
```

A.5 Tesseract on macOS (dev builds)

If you hit dylib resolution issues with Homebrew installs, use this post-build step:

```
<!-- OICLearning.csproj (snippet) -->
<Target Name="link_deps" AfterTargets="AfterBuild">
  <Exec Command="ln -sf /opt/homebrew/lib/libleptonica.dylib
                 $(OutDir)x64/libleptonica-1.82.0.dylib" />
  <Exec Command="ln -sf /opt/homebrew/lib/libtesseract.dylib
                 $(OutDir)x64/libtesseract50.dylib" />
</Target>
```

A.6 Build and Run (Web App)

1. Restore NuGet packages and build.
2. Apply EF Core migrations:

   ```
   dotnet tool restore
   dotnet ef database update
   ```

3. Launch the app:

   ```
   dotnet run
   ```

4. Ensure `FileRepository.SubmissionFolder` exists and is writable.

A.7 Operational Notes

- Re-uploads are idempotent; crops use stable names (e.g., `Q27a.png`).
- Only identity regions are OCR'd; answer crops go to the vision model.
- Groupings are editable; an *Ungrouped* bucket catches outliers.
- Grades appear in an on-screen table; CSV/Excel export is available.

Bibliography

[1] "Ghostscript," https://ghostscript.com/, Artifex Software, Inc., 2025, postScript/PDF interpreter.

[2] "Uglytoad pdfpig," https://github.com/UglyToad/PdfPig, UglyToad, 2025, .NET PDF reading library.

[3] "Skiasharp," https://github.com/mono/SkiaSharp, .NET Foundation, 2025, .NET 2D graphics library (Skia bindings).

[4] "Tesseract ocr," https://github.com/tesseract-ocr/tesseract, Tesseract OCR Developers, 2025, open-source OCR engine.

[5] "Openai gpt-4o and gpt-4o vision," https://platform.openai.com/docs/overview, OpenAI, 2025, multimodal LLM used for grouping/vision.

[6] V. Alto, *Modern Generative AI with ChatGPT and OpenAI Models: Leverage the capabilities of OpenAI's LLM for productivity and innovation with GPT-3 and GPT-4*. Packt Publishing Ltd, 2023.

[7] L. Chen, G. Chen, and X. Lin, "Artificial intelligence in education: A review," *IEEE Access*, vol. 8, pp. 75 264–75 278, 2020.

[8] R. Luckin, W. Holmes, M. Griffiths, and L. B. Forcier, "Intelligence unleashed: An argument for AI in education," Pearson Education, Tech. Rep., 2016.

[Online]. Available: https://www.pearson.com/content/dam/one-dot-com/one-dot-com/global/Files/about-pearson/innovation/Intelligence-Unleashed-Publication.pdf

[9] O. Zawacki-Richter, V. I. Marín, M. Bond, and F. Gouverneur, "Systematic review of research on artificial intelligence applications in higher education – where are the educators?" *International Journal of Educational Technology in Higher Education*, vol. 16, no. 1, p. 39, 2019.

[10] R. Weegar and P. Idestam-Almquist, "Reducing workload in short answer grading using machine learning," *International Journal of Artificial Intelligence in Education*, vol. 32, pp. 611–643, 2022. [Online]. Available: https://link.springer.com/article/10.1007/s40593-022-00322-1

[11] R. Könnecke and T. Zesch, "Automated scoring of content and style in short essays," *Frontiers in Education*, vol. 5, p. 90, 2020.

[12] T. Liu, J. Chatain, L. Kobel-Keller, G. Kortemeyer, T. Willwacher, and M. Sachan, "Ai-assisted automated short answer grading of handwritten university level mathematics exams," *arXiv preprint arXiv:2308.11728*, 2023. [Online]. Available: https://arxiv.org/abs/2308.11728

[13] G. Kortemeyer, "Toward AI grading of student problem solutions in introductory physics: A feasibility study," *Physical Review Physics Education Research*, vol. 19, no. 2, p. 020163, 2023. [Online]. Available: https://journals.aps.org/prper/abstract/10.1103/PhysRevPhysEducRes.19.020163

[14] G. Kortemeyer, J. Noh, and D. Onishchuk, "Grading assistance for a handwritten thermodynamics exam using artificial intelligence: An

exploratory study," *arXiv preprint arXiv:2306.17859*, 2023. [Online]. Available: https://arxiv.org/abs/2306.17859

[15] B. D. Lund, D. Wang, K. Chao, and M. S. Gerber, "Chatgpt's ability to provide timely, novel, and impactful ideas for research," *arXiv preprint arXiv:2305.06566*, 2023. [Online]. Available: https://arxiv.org/abs/2305.06566

[16] N. Mehrabi, F. Morstatter, N. Saxena, K. Lerman, and A. Galstyan, "A survey on bias and fairness in machine learning," *ACM Computing Surveys*, vol. 54, no. 6, pp. 1–35, 2021.

[17] C. C. Tossell, N. L. Tenhundfeld, A. Momen, K. Cooley, and E. J. de Visser, "Student perceptions of chatgpt use in a college essay assignment: Implications for learning, grading, and trust in artificial intelligence," *IEEE Transactions on Education*, 2023. [Online]. Available: https://ieeexplore.ieee.org/document/10120620

[18] D. J. Nicol and D. Macfarlane-Dick, "Formative assessment and self-regulated learning: A model and seven principles of good feedback practice," *Studies in Higher Education*, vol. 31, no. 2, pp. 199–218, 2006.

[19] "Gradescope," https://www.gradescope.com/, Turnitin, LLC, 2025, online grading platform; fixed-template paper exams and Answer Groups.

[20] "Crowdmark," https://crowdmark.com/, Crowdmark Inc., 2025, collaborative grading and assessment for paper and online exams.

[21] "Akindi," https://www.akindi.com/, Akindi Inc., 2025, oMR bubble-sheet creation and scanning.

[22] “Zipgrade,” https://www.zipgrade.com/, ZipGrade LLC, 2025, mobile multiple-choice scanning and analytics.

[23] “Canvas speedgrader,” https://www.instructure.com/canvas, Instructure, Inc., 2025, lMS grading workflow with rubrics and annotations.

[24] “Prairielearn,” https://www.prairielearn.org/, PrairieLearn, 2025, auto-graded, parameterized questions for STEM.

[25] “Möbius,” https://www.digitaled.com/mobius/, DigitalEd, 2025, algorithmic assessment and math engine-based autograding.

[26] “Coderunner,” https://coderunner.org.nz/, University of Canterbury, 2025, programming question autograder (often used via Moodle plugin).

[27] “Codegrade,” https://www.codegrade.com/, CodeGrade B.V., 2025, code autograding and rubric workflows.

[28] “The family educational rights and privacy act (ferpa),” Pub. L. No. 93-380 (1974), codified as 20 U.S.C. § 1232g and 34 CFR Part 99, 1974. [Online]. Available: https://www.govinfo.gov/content/pkg/USCODE-2023-title20/pdf/USCODE-2023-title20-chap31-subchapIII-part4-sec1232g.pdf