---

1. Overall Workflow & Major Steps

1. **Ingest Excel & Batch-Splitter**
    - Read the input `.xlsx` that contains “Name” + “GitHub URL.”
    - Split into consecutive groups (batches) of 4 rows each.
2. **Per-Batch Processing**
    - For each batch of 4 students:
        1. Dispatch an “Analyzer Agent” for each of the 4 GitHub URLs in parallel (so up to 4 analyzers running concurrently).
        2. Each Analyzer:
            - Clone or fetch the student’s repo.
            - Walk through relevant files (e.g., look for `README`, source code folders, tests, etc.).
            - Apply the grading rubric (total 12 points) by checking correctness, style, documentation, edge cases, etc.
            - Produce a **concise analysis**:
                - “Grade: X/12”
                - Bullet-point list of mistakes (e.g., “Missing error handling in `parse_input()`,” “Inefficient loop in `sort.py`,” etc.)
                - A brief “Congrats”: “Wellstructured modules, clear README, test cases all passing,” etc.
        3. Once all 4 Analyzers finish (or time out), an “Aggregator Agent” collects the four outputs, compiles them into a single DataFrame (or similar), and

writes/appends to an output `.xlsx`.

4. Optionally, log any failed/malformed repo links for manual review.

3. **Output**
    - After each batch finishes, you generate/update the “results.xlsx” containing:
      | Student Name | Grade (out of 12) | Concise Analysis |
    - Once all batches are done, you have a complete grading sheet.
4. **(Optional) Final Report or Notification**
    - If you want to notify someone or push the final `.xlsx` to some shared location (Slack, Google Drive, etc.), you could add a “Publisher Agent.”

---

2. Agent Roles & Estimated Counts

Below is a suggested minimum set of agent **roles**. You can scale the “Analyzer” role up or down depending on how many repos you want to process truly in parallel. If CrewAI lets you spawn more simultaneous workers, you could even run multiple batches concurrently—but let’s assume you process batches strictly in sequence (per your description).

1. **Controller Agent (1)**
    - Responsible for:
        - Reading the master `.xlsx` (contains N students).
        - Splitting into batches of 4.
        - Orchestrating the hand-off: “Here’s batch #1 ➔ send to Analyzer Agents #1–4.”
        - Watching for completion/failures and then signaling the Aggregator.
        - Looping batch by batch until you hit the end of the file.
2. **Analyzer Agents (up to 4 per batch)**

- Role: “Given (name, github_url, rubric), produce (grade, concise analysis).”
- They run in parallel for each batch.
- Once an analyzer finishes or times out, it returns its result back to the Controller.
- You could conceivably have a small **pool** of, say, 4 Analyzer Agents that the Controller reuses batch after batch. In other words, 4 “Analyzer slots” is usually enough if you want exactly four at a time. If you expect spikes or want more parallelism (e.g. you have hundreds of students and want 8 at once), scale accordingly. But the minimum to support “batches of 4” is 4 Analyzer Agents.

3. **Aggregator Agent (1)**
   - Once the Controller detects all 4 Analyzer Agents have returned (or a timeout/failure), it calls the Aggregator.
   - Aggregator responsibility:
     - Assemble the four (name, grade, feedback) tuples.
     - Append them to (or rewrite) a central `results.xlsx`.
     - Also record any anomalies (e.g., repo-clone errors) for manual review (a separate “Error Log” sheet or a CSV).
   - If you want to generate the output at the very end, the Controller could wait until all batches are done and then call a single Aggregator. But it’s often simpler (and safer) to append after each batch finishes to avoid losing work if something goes wrong mid-run.
4. **(Optional) Publisher / Notifier Agent (1)**
   - If, for instance, you need to push the `.xlsx` somewhere (Slack, email, cloud storage), you might add one more agent to do so.
   - This can be triggered by: “All batches done ➔ Publisher pushes to S3 or emails stakeholders.”

Summary of Agents

- **Controller Agent:** 1 (orchestrates the whole pipeline)
- **Analyzer Agents:** 4 (fixed pool, re-used each batch)
- **Aggregator Agent:** 1 (compiles results into `.xlsx`)
- **Publisher Agent (optional):** 1

Thus, in the simplest setup, you need **6** distinct agent "roles," but you can reuse or multiplex some of them:

- Realistically, you only need one instance of Controller + one Aggregator + one Publisher (if needed).
- For analyzers, you need 4 parallel "slots." Those can be 4 separate long-running agents (each waiting for tasks), or a dynamic pool where the Controller says "spawn another analyzer" up to a max of 4 at a time.

---

3. Task Allocation & Data Flow

Below is a conceptual "step-by-step" of how each agent talks to one another:

1. **Controller starts**
    - Reads `input_students.xlsx` → gets a list of e.g. 20 rows.
    - Splits into batches of 4: batch 1 = rows 1–4, batch 2 = rows 5–8, etc.
2. **Controller dispatches Batch 1**
    - For each student in Batch 1:
        - Grab (Name_i, URL_i).

Send a "grade_request" to Analyzer_i (i = 1..4) with payload:

```
{
 "name": "Alice Mwangi",
 "github_url": "https://github.com/...",
```

```
  "rubric": { /* embed or reference your 12-point rubric here */ }
}
```

3. **Analyzer Agents run in parallel**
    - Each analyzer sees “grade_request” →
        - Clone/fetch the student’s repo (via `git clone` or GitHub API).
        - Read/analyze code (e.g., look for folder structure, run a linter, inspect tests, etc.).
        - Apply rubric: increment points for each category (e.g. 3/3 for “Documentation,” 4/4 for “Correctness,” etc.).

Produce a JSON response, e.g.:

```
{
  "name": "Alice Mwangi",
  "grade": 10,
  "feedback": [
    "Good separation of concerns across modules. (+3 docs)",
    "Missing test cases for edge conditions in `utils.py`. (-1 correctness)",
    "Inefficient nested loops in data parsing; consider using list comprehensions. (-1 efficiency)",
    "README is well-written. Keep it up!"
  ]
}
```

    - Once finished, analyzer returns that payload to Controller.
4. **Controller waits**
    - As soon as all 4 analyzers respond—or if a timeout (e.g., 5 minutes) occurs for any—Controller collects whatever responses did come back.

If someone timed out or failed to clone, Controller can create a placeholder:

```
{
  "name": "Bob Otieno",
  "grade": 0,
  "feedback": ["Repo could not be cloned (invalid URL or private)."]
```

}

-

5. **Controller calls Aggregator**
    - Payload: array of up to 4 “analysis_result” objects.
    - Aggregator’s job:
        - Open (or create) `results.xlsx` (with columns: Name | Grade | Feedback).
        - For each object in the array, append a row:
            - Name → cell A
            - Grade → cell B
            - Feedback → concatenate feedback array into one cell (e.g., join by newline or semicolons).
        - Save `results.xlsx`.
        - Return “success” or list of any failures (e.g., sheet is locked, disk error).
6. **Controller logs & moves to next batch**
    - If Aggregator succeeded, Controller sends the next 4 students to the same 4 Analyzer slots.
    - Repeat until all batches are done.
7. **Controller triggers Publisher (optional)**
    - When batches 1…N are complete, “All done” → Publisher Agent:
        - Upload `results.xlsx` to a shared folder, or send an email with an attachment, etc.

---

4. How Many Physical Agents vs. Logical Agents?

- **Logical Agent Roles** (6 roles as listed above).
- **Physical Agent Instances**
    - Controller: 1 process/agent.
    - Aggregator: 1 process/agent.
    - Publisher: 1 process/agent (only if you need that final step).
    - Analyzer: up to 4 concurrent processes/agents.
        - You can implement these as 4 identical “grading” agents listening to a shared queue. The Controller pushes tasks onto the queue, and whichever of the 4 is free picks it up. Once it finishes, it returns results to Controller.

If CrewAI charges per “active agent,” you could spin up exactly 6 agents total—4 running identical grading code and 1 each for Controller/Aggregator. If you do not need Publisher, then 5 is enough. If you want a hot spare analyzer, spin up 5 analyzers and let 4 work at a time—your choice.

---

5. Rubric Management

Since “grade out of 12” is central, you’ll want to standardize how each Analyzer applies it. A few recommendations:

1. **Embed the Rubric as JSON or a Plain-Text Prompt Template**
    - Example rubric could have categories like:
        1. **Correctness (0–4)**
        2. **Code Organization & Readability (0–3)**
        3. **Documentation & Comments (0–2)**
        4. **Testing / Error Handling (0–2)**
        5. **Efficiency & Best Practices (0–1)**

- Summed = 12. Each Analyzer should be handed that rubric in a machine-readable form so it can explicitly say “I gave X points here because…”.

2. **Force Structured Output**

When you write the prompt for the Analyzer Agent, require it to output something like:

```
{
 "correctness": 3,
 "organization": 2,
 "documentation": 2,
 "tests": 1,
 "efficiency": 1,
 "grade": 3+2+2+1+1,
 "feedback": [
   "...",
   "..."
 ]
}
```

-
- The Controller or Aggregator can double-check that “grade” = sum of subcategories, avoiding human error.

3. **Version & Update**

- If you ever tweak the rubric (e.g., move some weight from “docs” to “tests”), you only need to update the JSON or text file that every Analyzer reads. This ensures consistency.

---

6. Error Handling & Edge Cases

1. **Private or Invalid Repos**

- If the GitHub link is private or malformed, the Analyzer should catch the clone-error and return a special flag `"repo_access": false`, plus feedback like “Couldn’t clone repository. Please verify link or make repo public.”

- The Aggregator can then flag that student as “0” or “N/A” and maybe add them to a separate “Check Manually” list.

2. **Timeouts**

Decide on a reasonable upper bound for code analysis (e.g., 2 minutes per repo). If the Analyzer doesn’t finish in that window, it sends back:

{ "name": "...", "grade": 0, "feedback": ["Timed out during analysis."] }

-
- The Controller can optionally retry once, or just move on.

3. **Large Repos**

    - Some students might have very big repos with multiple languages. Clarify in your rubric/agent instructions exactly which folder or language to focus on (e.g., “grade only `.py` files under `/src`”).

4. **Network / API Rate Limits**

    - If you clone many public GitHub repos in parallel, be mindful of GitHub’s rate limits. You might want to use a cached mirror or alternate approach (e.g., use the GitHub API to fetch individual files rather than `git clone`) if you see “rate limit exceeded” errors.

5. **Result-Sheet Locking**

    - If multiple Aggregator calls try to write to `results.xlsx` at the same time (e.g., if you did two batches in parallel), you could corrupt the file. To avoid this, you have two options:

        - **Sequential Batches**: Only one Aggregator ever writes at a time. The Controller enforces “start Batch 2 only after Batch 1 Aggregator finishes.”

        - **Single Aggregator Instance**: Every batch’s results get pushed into a shared in-memory store or database, and a single Aggregator writes once at the very end.

---

7. Scaling Beyond Batches of 4

- If you eventually decide to push more than 4 students to “analyze” in parallel (say, batches of 8 or 16), just spin up more Analyzer slots. For example, if you want 8 parallel analyzers, change your Controller to split into batches of 8, and instantiate 8 Analyzer Agents (or a dynamic pool of size 8).

- The same Controller/Aggregator pattern holds; just adapt the “batch size” variable.

---

8. Miscellaneous Tips & Best Practices

1. **Logging & Monitoring**

    - Have each agent write a minimal log (student name, repo URL, start/end timestamps, error codes). This makes it easier to debug why a particular student’s code got “0” or “timed-out.”

    - Optionally centralize logs (e.g., push to a simple text log or a lightweight database) so you can query “Which repos failed to clone?” or “Which Analyzer timed out most often?”

2. **Local vs. Remote Execution**

    - Decide whether the Analyzer Agents run inside Docker containers (to keep dependencies consistent) or on a bare VM/host. Containers help isolate “rogue” student code that tries to install weird packages or runs indefinitely.

3. **Rubric Transparency**

    - If students will see their feedback, consider sharing the rubric breakdown (e.g., “You got 3/4 on organization because files are in a single folder; consider splitting into modules.”). This teaches them where to improve.

4. **Version Control for the Rubric & Prompts**

    - Keep your rubric JSON and any prompt templates under version control (e.g., a Git repo). That way, if you tweak criteria mid-semester, you know exactly which runs used which rubric version.

5. **Testing Your Pipeline**

    - Before pointing it at real student repos, run a “dummy batch” with 4 known sample repos (e.g., 2 that pass all rubric checks, 1 with missing files, 1 invalid

URL) and verify:

- Each Analyzer returns exactly the JSON you expect.
- Aggregator appends correctly.
- Controller cleanly moves on to the next batch.

6. **Concurrency Limits & Resource Usage**
    - If each “Analyzer” is cloning a repo, doing static analysis, maybe even executing a small test suite, they will spike CPU/memory. Make sure your hosting environment can handle 4 (or more) simultaneous clones + analysis.
7. **Security / Isolation**
    - Since you’re running untrusted student code, best practice is to run each Analyzer in a sandbox (Docker, Firejail, or similar) with no external network access except to clone. This prevents malicious code from “phone home” attempts or from modifying your agent’s filesystem.
8. **Feedback Format**
    - Keep each feedback as a bullet list of short sentences. The Aggregator can join them with line breaks. In Excel, too many line breaks can look messy; consider joining with semicolons or using a single cell that wraps text.
9. **Regrading / Re-runs**
    - If a student pushes a fix and wants a regrade, just re-enqueue that single “(name, URL)” pair by sending it through the same Controller → Analyzer → Aggregator flow, and overwrite their previous row in `results.xlsx`.

---

9. Concrete Agent Count & Allocation Summary

Agent Role	# of Instances/Slots	Responsibilities
Controller	1	Reads input .xlsx, splits into batches, dispatches to analyzers, tracks progress.

Analyzer (“Grader”)	4	Each takes one student URL + rubric, clones repo, applies rubric, returns (grade+feedback).
Aggregator	1	Gathers batch results, appends to `results.xlsx`, handles errors or missing data.
Publisher / Notifier (opt.)	1	After all batches done, pushes `results.xlsx` somewhere (email, cloud, Slack).

- Total “live” agents: **7** if you include the optional Publisher, otherwise **6**.
- If you know you’ll never notify externally, you can omit the Publisher → **6**.
- If CrewAI charges per agent, you can spin down the Publisher or merge Controller+Aggregator into one “meta-agent,” but that adds complexity. Usually keeping them separate clarifies responsibilities.

---

10. Final Advice

1. **Keep Agents Focused**
    - One reason to separate “Controller” from “Aggregator” is that Controller’s job (scheduling, queuing, timeouts) differs from Aggregator’s (file I/O, Excel formatting). If you combine them, you risk mixing concerns and harder debugging.
2. **Plan for Partial Failures**
    - A batch might partially succeed (e.g., 3 analyzers return good data, 1 fails). Make sure Aggregator can still append the three valid results and separately record the one failure—don’t let one failed clone wipe out the entire batch.
3. **Monitoring & Alerts**
    - Build a lightweight dashboard (even a shared Google Sheet or a text log) where you can see “Batch 3 – Analyzer #2 timed out.” That way you can manually re-trigger just that student’s analysis if needed.

4. **Version Your Prompts**
    - If your rubric changes halfway through the term, tag each run with a "rubric_version" field. Then you can audit which students got graded under version 1 vs. version 2.
5. **Security**
    - Always run analyzers in a sandbox (e.g., Docker). If a student's code has malicious intent—like trying to exfiltrate data, use infinite loops, or spawn processes—you want to isolate it.
6. **Testing**
    - Before you point it at a large roster, validate the entire pipeline end-to-end with a handful of known repos. That will flush out any YAML/Excel-format mismatches, rate-limit issues, or timeout flows.
7. **Scale Up or Down**
    - If 4 analyzers become a bottleneck (e.g., you have hundreds of repos and it's taking days), consider increasing to 8 or 16 analyzer slots. You can still batch in groups of 4, but have multiple batches in flight. Just make sure your Aggregator isn't writing to the same `.xlsx` simultaneously; either queue all results in memory or switch to a simple database (SQLite, Postgres) and export to `.xlsx` as a final step.
8. **Documentation**
    - Finally, write a short README (or mental note) describing this pipeline so that, in future terms or if a team member joins, they can see:
        - "Controller waits for 4 analyzers, then calls Aggregator, which writes to results.xlsx,"
        - How to update the rubric,
        - How to add more analyzer slots,
        - Where logs live and what a typical error looks like.