ChatGPT

Comprehensive Plan for the Phase-6 Excerpt-Aware Runner

Background and Objectives

The **Mustikarasa Project** is creating a reliable digital edition of the historic *Mustika Rasa* cookbook by using an **agent-based editorial workflow** [1]. Multiple AI agents (translators, editors, annotators, etc.) are **orchestrated by a central runner (Orchestrator agent)** to simulate a rigorous human editorial process [2]. Every change made by these agents is logged and justified for full traceability ("zo was het — zo is het — en dit is waarom het veranderde," meaning *"this is how it was, how it is, and why it changed"*) [3].

In earlier phases, an initial "runner" script was developed as a pilot to test this workflow. That **pilot runner** was a proof-of-concept (used in Phase-3/4 trials) and not meant for long-term use. We have since learned important lessons (for example, the need for explicit excerpt metadata on every run and standardized output files) and documented new requirements. Now, in **Phase-6**, the goal is to **rebuild the runner from scratch** with a clean design that implements these lessons and meets all formal specifications. This new **excerpt-aware runner** will be the backbone for executing the editorial workflow on each piece ("excerpt") of content, with **full compliance to governance, traceability, and metadata standards** [4]. It must integrate all currently active capabilities of the system and be extensible to include upcoming ones. Although it will run on a Mac Mini during development, the design should remain platform-agnostic and portable. In summary, the objectives for the new runner are:

- **Fresh Implementation:** Develop a clean runner implementation (disregarding the throwaway pilot code) while leveraging insights gained.
- **Excerpt Awareness:** For each run, require and propagate the *excerpt's* identity and version information throughout, ensuring outputs and logs are tied to the correct source excerpt [5].
- **Standard Outputs & Logging:** Produce output files and logs in the standardized Phase-6 format (no more ad-hoc file names or locations), to facilitate review and version control [6] [7].
- **Integrate All Key Capabilities:** Orchestrate **all active agent capabilities** in the workflow (translation, editing, annotation, etc.), and prepare hooks for designed (but not yet fully integrated) capabilities, so that **every system capability is addressed** either in implementation or in forward-looking design.
- **Governance & Traceability:** Enforce "no final decisions by AI" – the runner must only produce *provisional* outputs that require human review, and log all decisions and uncertainties for transparency [8] [9].
- **Platform and Model Agnosticism:** Ensure the runner's code is not tied to Mac-specific features. The solution will be implemented in Python (the project's chosen language) and should work across environments. It should also be **model-agnostic**, capable of using different large language model backends (e.g. OpenAI GPT or local models via frameworks like CrewAI/Ollama) without code changes [10]. This gives flexibility to run on local hardware (the Mac Mini) or cloud as needed.

With these objectives in mind, we outline below the detailed requirements, workflow design, integration of capabilities, and technical approach for the Phase-6 excerpt-aware runner.

Key Requirements and Specifications

Implementing the new runner involves satisfying a number of **functional requirements** (what the runner must do) and **non-functional requirements** (how it should do it, in terms of reliability, structure, etc.). The key specifications, drawn from the Phase-6 design documents, are as follows:

- **Explicit Excerpt Metadata:** The runner must require **explicit identification of the content excerpt** being processed. On invocation, the CLI should accept parameters for `excerpt_id`, `excerpt_source`, and `excerpt_version` (as well as a `run_id`) [5]. These metadata fields uniquely tie the run to the source text selection and its version. The runner **must not infer** these from filenames or guess them – they have to be provided or looked up in a config. This was a lesson from earlier phases: making excerpt context explicit is crucial to avoid mix-ups. All these fields will be propagated into outputs and logs.

- **Pre-Flight Validation:** Before doing anything irreversible, the runner performs a **pre-flight check** on inputs [11]. It verifies that all required metadata (excerpt id/source/version, run id) are present and coherent. For instance, if both CLI args and a config file specify these fields, the runner will trust the config but log a metadata mismatch warning (setting a flag `cli_config_mismatch:true` in the log header) [12]. If any required field is missing or if the run_id is invalid (e.g. wrong format), the runner must **STOP** immediately *before* invoking any agents [13]. In a stop scenario, it should emit a brief log explaining the error and then abort the run safely. This prevents running with ambiguous context or producing outputs that lack proper identification.

- **Run Identification:** Every run is identified by a stable `run_id`. If the user doesn't supply one, the runner will generate a unique run ID (e.g., incrementally or timestamp-infused) to tag this execution. **Never reuse run IDs or overwrite outputs** of a previous run – each run must be isolated for auditability [14] [15]. The naming convention for run IDs and excerpt IDs should follow project standards (for example, `RUN_SAYUR_CASE01_001` for case-01 of chapter "Sayur") and include a timestamp in logs for uniqueness [16]. This ensures that runs are traceable and that re-running on the same excerpt produces a new record rather than silently replacing history.

- **Logging Requirements:** The runner will create a **detailed log file** for each run, stored in the designated location: `sandbox/crew/run_logs/{excerpt_id}/` [17]. The log filename should include the run_id and timestamp (e.g. `RUN_SAYUR_CASE01_001_20260105T154946.log`) [18]. At the top of the log, the runner must record a **log header** containing at least: the excerpt_id, excerpt_source, excerpt_version, run_id, timestamp, and the mode (which will be "excerpt-aware" for this Phase-6 runner) [7]. This header is documentary (for humans) to clearly identify the context of the run; it doesn't itself enforce any rules, but it's critical for governance review. After the header, the log will chronologically record each step the runner takes (e.g. which agent is running, any important decisions or flags, and whether any exceptions occurred). The log will also note if any special conditions were encountered (for example, a CLI vs config metadata mismatch as mentioned, or a stop condition that prevented a step). By adhering to this logging format, we ensure that **each run's**

configuration and outcome are documented for reproducibility, which aligns with the project's vision of an auditable knowledge infrastructure [19].

- **Output Artifacts and Structure:** The runner's primary job is to produce **structured output files** (in JSON/Markdown) that capture the results of the agents' work on the excerpt. According to the Phase-6 output layout spec, for each run the runner should create a directory: `sandbox/crew/run_outputs/{excerpt_id}/{run_id}/` and place the output files there [20] [21]. At minimum, the runner must output the following files for each run [22]:

- `annotator_primary.json` – the main structured output from the annotation/editing pass (the AI's proposed edits/annotations on the excerpt).
- `challenger_primary.json` – the output from the challenger agent (issues or critiques identified, if any).
- `crew_provisional.json` – the consolidated provisional output, which may combine or reference the above, representing the state of the excerpt after AI processing but *before* any human review.
- `review_notes.md` – (optional) a placeholder Markdown file for any human reviewer notes or decisions. The runner can create an empty template or simply leave this for humans to add when reviewing.

No other files should be required for a basic run. The naming is important, as external tools and governance scripts will expect these exact filenames. The runner should **not alter or delete past outputs** in this directory – each run is separate (so, no retrofitting old runs to the new format) [23]. If we re-run the same excerpt, we'll get a new run_id and a new folder, rather than overwriting the old one.

- **Embedded Metadata in Outputs:** Each JSON artefact (`annotator_primary.json`, `challenger_primary.json`, `crew_provisional.json`) must **embed the excerpt metadata and run metadata** within it [24]. For example, a JSON might have a top-level structure like:

```
{
  "excerpt": {
    "id": "<excerpt_id>",
    "source": "<excerpt_source>",
    "version": "<locked-version-tag>"
  },
  "run": {
    "id": "<run_id>",
    "timestamp": "<ISO timestamp>"
  },
  "items": [ ... ]  // actual content results here
}
```

The runner will be responsible for injecting these fields so that any JSON file can be understood in isolation – this redundancy is by design for cross-verification [25] [26]. After the run, a governance script or reviewer can cross-check that all files share the same excerpt metadata. If any file is found to have inconsistent IDs or missing metadata, the **run is considered invalid** [27] and must be flagged for human investigation (the system should not trust or publish such a run). The new runner will include a final consistency check to

ensure the excerpt ID/source/version in all outputs match each other and match the input parameters; if not, it will log an error and treat it as a failed run (no "silent" mismatches).

- **No Automatic Canonicalization:** A critical governance requirement is that **the AI runner cannot finalize content as canonical**. All AI outputs remain proposals until a human approves them. Therefore, the runner's outputs are considered **provisional** (even if they pass internal checks). The runner must *never* auto-promote content to the final canonical state, and it should not allow any agent to "approve" changes beyond the provisional stage [28] [29]. In practice, this means the runner stops after producing `crew_provisional.json` and does not merge anything into the official book content. The transition from **CREW_PROVISIONAL** to **READY_FOR_HUMAN_REVIEW** (and eventually to **CANONICAL**) is a **human-only process** outside the runner's automated scope [30] [8]. We will ensure the runner clearly outputs that state (e.g. by storing a marker or by the naming itself) and perhaps generates a summary of what needs human attention, but it hands off at that point. This guarantees compliance with the "human gate" principle: *no AI alone can produce final decisions*.

- **Error Handling and Stop Conditions:** The runner should be robust against errors. If an agent fails or returns an invalid result (e.g., malformatted JSON or other unexpected output), the runner should catch that exception, log an incident, and halt further processing for that run. Notably, the design includes a **Troubleshooting Agent** which, when integrated, can be invoked to generate an `INCIDENT_REPORT` if something goes wrong [31]. We plan to incorporate this: on any caught exception or rule violation during the run, the runner will trigger the Troubleshooting agent to document the issue in a structured way, then stop gracefully. Even if that agent is not fully ready, our runner will be structured to support it (e.g. by preparing error context and calling a stub or logging a TODO). The key is that the runner must *not* plow ahead in an invalid state – better to stop and preserve a trace of what happened for analysis.

- **Performance and Scalability:** While raw performance is not the top priority (we favor traceability and correctness over speed in Phase-6 [32]), the runner should be written in a way that **scales to larger workloads**. This implies cleanly separating the processing of one excerpt from another, such that multiple excerpts could be processed in sequence or parallel in the future. In Phase-6 we target single-excerpt runs (Case-01, Case-02, etc.), but the architecture anticipates eventually running the entire book through in batches. To prepare for this, the runner will be implemented as a reusable module (e.g., a Python function or class that takes an excerpt and runs the pipeline). This way, a future **Batch Governor** agent or script could call the runner for each chapter or recipe in a loop or parallel threads [33]. The runner's design (one agent = one function, clear data inputs/outputs) also helps with optimization: if some steps become bottlenecks, we can swap in faster models or add caching without redesigning the whole pipeline [34]. Being aware of the eventual scale-up ensures we don't introduce any needless bottlenecks or hard-coding that would prevent running, say, dozens of excerpts in a batch. For now, on the Mac Mini, we'll run one excerpt at a time, but we will test that we can run multiple different excerpts back-to-back reliably (to mimic a batch).

With these requirements established, we can now describe **how the runner will work**, step by step, and how it integrates each of the system's capabilities.

Runner Workflow and Architecture

The **Phase-6 Excerpt-Aware Runner** will orchestrate the entire editorial workflow for a given excerpt in three high-level phases: **Pre-flight, Run, and Finalize** [35] . Below we break down the workflow in detail:

1. **Pre-Flight Phase:** When the runner is invoked (via CLI or a higher-level script), it first performs the pre-flight validation. The code will parse CLI arguments or config inputs to obtain the required `excerpt_id`, `excerpt_source`, `excerpt_version`, and `run_id`. It logs the intent to start a run and then **validates these parameters**. If anything is missing or clearly incorrect (like a blank excerpt_id or an invalid run_id format), the runner writes a brief error to the log and aborts (no agents executed) [36] [13] . Assuming validation passes, the runner timestamps the start of the run, generates a run_id if needed, and initializes the output directory for this run. It then opens the log file and writes the **log header** with all the metadata [7] . At this point, we have a log ready to capture the workflow steps.

2. **Execution Phase (Agent Workflow):** This is the core of the runner – it coordinates a **sequence of AI agents** that process the content. The Orchestrator logic (embedded in the runner) will load the excerpt text (from the `excerpt_source` path, which presumably is a markdown of the chosen recipe/section) and then call each agent in turn, passing along the evolving content. The sequence of agents corresponds to the **active capabilities of the system's editorial pipeline**, which are all implemented in Phase-6. In order, the runner will invoke:

3. **Orchestrator Setup:** (Not an agent per se, but the orchestrator code itself) — The orchestrator prepares the initial state for the excerpt. For example, it may apply a template or ensure the excerpt text is parsed into a structured form if needed (with sections like ingredients, steps, etc. recognized). It ensures the workflow adheres to the expected template structure and will log the structure of the content at the start [37] . The orchestrator acts as the **"central brain"** coordinating everything [38] . As it begins, it might log, *"Starting excerpt workflow for [excerpt_id] using run [run_id]..."*. Then it moves to calling the specialized agents one by one:

4. **Translation Quality Agent:** This agent ensures the initial translation from Indonesian to Dutch is accurate and faithful to the source meaning [39] . In practice, if the project already has a base translation (possibly machine-translated or previously translated text), this agent will review it and correct any errors or mistranslations. Since this capability is *Active* (already implemented and integrated) [40] , the runner will definitely invoke it. The agent likely uses an LLM prompt to compare source and translated text and output either an improved translation or some quality score/ annotations. The runner passes in the source text (and possibly an English reference if available, as indicated by "with English references" in the docs [39]) and gets back a reviewed Dutch text. The output of this step would be an updated excerpt text (in Dutch) with high fidelity to the original. The runner logs the completion of this step (and any major changes noted).

5. **Readability Editing Agent:** Next, the runner calls the readability editor to polish the translated text for fluency and clarity [41] . This agent will improve phrasing, fix awkward sentences, and ensure the Dutch reads naturally *while preserving the original meaning* [41] . It is strictly instructed not to introduce unauthorized changes (no altering meaning or adding content) [41] . The runner provides the current text to this agent and receives an edited version. The agent is Active (fully implemented),

so this is a standard part of the pipeline. After this step, the excerpt should be grammatically sound and fluent. The runner logs that readability edits were applied.

6. **Cultural-Historical Annotation Agent:** The runner then invokes this agent to add any necessary annotations regarding cultural context or historical notes [42]. *Mustika Rasa* being a historical cookbook, there may be terms, ingredients, or techniques that benefit from explanation. This agent scans the text and inserts or prepares annotations (for example, footnotes or bracketed notes) with relevant background info [42]. This capability is Active; thus, the runner includes it. The output might be a set of annotations or an augmented text. The runner will incorporate those annotations into the working excerpt, or store them separately depending on implementation (likely the JSON output will have a section for annotations). The log would note any annotations added (e.g., *"Added 3 cultural notes for terms: [X], [Y]..."*).

7. **Structural Formatting Agent:** Next is the structural or "book structure" agent, which ensures that the content conforms to the proper structural rules [43]. For example, it will check that each recipe has all required sections (ingredients list, instructions, etc.), that section headings are correctly formatted, and that the overall hierarchy (chapters, subchapters) remains intact. It also handles things like table of contents alignment and numbering consistency [43]. This agent is Active, so the runner will execute it. It might rearrange or wrap content in the appropriate markers. The runner logs structural adjustments made (if any).

8. **Recipe-Specific Editing Agent:** The runner will then call the recipe-specific editor agent, which focuses on the culinary content itself [44]. This agent might ensure that ingredient names are standardized, units of measure are consistent, and the recipe steps are clear and logical. It's essentially a domain-specific QA for recipes. Active and implemented, it will run on each recipe excerpt. The changes it makes (if any) are applied to the text. The runner logs such changes (e.g., *"Standardized ingredient names to Dutch culinary terms where needed."*).

9. **Table Handling Agent:** If the excerpt contains tables (nutritional information, ingredient tables, etc.), the table handling agent is invoked to reconstruct or format those properly [45]. Mustika Rasa likely has tables or charts in some sections, and OCR or formatting could have scrambled them. This Active agent ensures tables are captured and presented legibly in the output [45]. The runner will supply any raw table data or images to this agent and get back a structured table (perhaps as Markdown or JSON table). After this, any tables in the excerpt should be correctly formatted. The runner logs that tables were processed (or notes if none were present).

10. **Image Integration Agent:** For sections with images or illustrations, the image integration agent runs to manage those references [46]. This might involve linking scanned images, adding figure captions, or verifying that the image files exist and are referenced correctly. The agent (Active in Phase-6) will ensure that wherever the original book had an image, the digital edition includes it or at least references it properly [46]. The runner might pass a list of expected images to this agent and get back confirmation or insertion points. The log notes any images linked (e.g., *"Linked 2 images (fig_10.png, fig_11.png) to the excerpt."*).

11. **Continuity/Cohesion Agent:** After the content-specific edits, the runner triggers a continuity check across the content [47]. This agent ensures internal consistency – e.g., uniform usage of terms and names across the excerpt, consistent tone and style, no contradictions. If this excerpt is part of a

larger chapter, it might also ensure consistency with related excerpts. The agent is Active and implemented in code [47]. It will flag any inconsistencies or possibly make minor adjustments for cohesion. The runner logs the result (if issues were found or if the excerpt is consistent).

12. **Fidelity Assurance Agent:** As a final content-processing step, the runner calls the fidelity agent to do a compare between the final Dutch text and the original Indonesian (and possibly the intermediate English) [48]. The purpose is to catch any meaning shifts that might have crept in during all the editing. The Fidelity agent produces an "Opmerkingen" (Notes) section listing potential deviations in meaning [48]. This is an Active capability and acts as a safeguard that the modernization hasn't introduced errors or distortions. The runner will take the output of this agent (likely a list of remarks or a short report) and include it in the provisional results (perhaps appended to the annotations or as a separate notes file). The log will state that fidelity checking was done and if any issues were flagged.

13. **Design and Layout Agent:** (If applicable in this phase) The design/layout agent ensures the final text adheres to style guidelines and is properly formatted for publication [49]. It may adjust phrasing to fit a desired tone or check typographic conventions. Since this agent is also Active now, the runner can invoke it at the end of the pipeline to do a pass focusing on presentation (without altering content meaning) [49]. The output might be subtle changes in wording or formatting (like consistent quotes, italics, etc.). The runner logs that the design polish was applied.

After running all these **content processing agents**, the excerpt has been through the full AI editorial pipeline. At this stage, we have what we can call the **"annotator primary" output** – essentially the AI's best effort at a fully edited, annotated piece of content. The runner will compile this result (the edited text plus any inline annotations/notes) and prepare to output it as `annotator_primary.json`.

However, the workflow doesn't end here. There is also the important role of the **Critical Challenger** agent.

- **Critical Challenger Agent:** Once the primary editing run is complete, the runner will initiate the Challenger agent. This agent acts as an **internal critic or QA auditor** on the AI's own work [50]. Its job is to double-check from a different angle and catch issues the primary pass might have missed. For example, the challenger might re-evaluate certain translations or edits and ask, *"Are we sure about X? Could Y be wrong?"* [51]. The Challenger's prompt is designed to identify potential errors or rule violations and flag them (often labeling issues with severities like info/warning/blocker). In Phase-6, the Challenger is **active and integrated** into the workflow [50]. The runner will feed the challenger the current edited excerpt (and perhaps the original for reference) and gather its findings. The output of the challenger will be saved as `challenger_primary.json` – typically a list of issues or questions raised, each with references to the part of text in question. The runner logs that the challenger phase completed and how many issues were raised.

After the Challenger, we have two parallel outputs: the annotated edited content, and the challenger's critique. The runner (playing the orchestrator role) now performs any **internal consistency checks or merges** needed to produce the final provisional output:

- **Internal Resolution (Crew Provisional Assembly):** The term "crew" in `crew_provisional.json` reflects the notion that the *AI crew* (all agents together) produce a joint result. The runner will take the annotator's output and the challenger's notes and **combine them into the provisional output**.

This does **not** mean automatically fixing what the challenger flagged (the AI will not self-correct issues that require judgment), but it means packaging the information together for the human reviewer. For example, if the challenger flagged a translation ambiguity, the crew_provisional output might include an annotation like "⚠ *Challenger note: possible mistranslation of term X*". In practice, the `crew_provisional.json` could mirror the structure of the annotator output but with additional fields or comments indicating challenger-found issues (i.e., *disagreements documented*) [52] . The key point: at **CREW_PROVISIONAL** stage, all machine commentary is included and any disagreements or uncertainties are explicitly noted [53] . The content is *not yet approved* – it remains a proposal with open questions marked. The runner ensures no issues identified by the challenger are ignored: it will either attach them to the relevant part of the text or list them in a summary section of the provisional output. This way, a human reading the provisional output can immediately see where the AI is unsure or found a potential problem.

The assembly of the provisional output is the final act of the Execution Phase. Once `crew_provisional.json` is ready, the runner writes out the **annotator_primary.json**, **challenger_primary.json**, and **crew_provisional.json** files to the output directory (including all required metadata fields as discussed). It then logs that the workflow execution is completed.

1. **Finalize Phase:** In the finalize step, the runner performs some housekeeping and verification:
2. It closes out any open file handles and writes a summary in the log (e.g., "Run complete. Outputs at … All systems nominal." or if there were issues, "Run complete with X issues flagged – human review recommended").
3. It double-checks that the output files have been placed in the correct folder and named properly according to spec [54] . If any file is missing or misnamed, the runner should log a warning (this scenario is unlikely if coded correctly, but it's good to have a sanity check).
4. It may perform a **metadata consistency audit**: ensure that each JSON file contains the correct excerpt and run metadata and that they all match each other [24] [27] . Any discrepancy here would also be logged (and ideally, marked so that the run is not trusted until corrected).
5. Optionally, the runner can mark the outputs as **READY_FOR_HUMAN_REVIEW**. In practice this could be just an informational log entry or setting a status somewhere. According to the Phase-6 human review workflow, before humans get the output, one should verify excerpt binding and JSON sanity one last time [55] – our runner does this as described, so once done, it effectively means the outputs are ready for the human review stage.
6. The runner does *not* attempt any further action beyond this. In particular, it will **not auto-promote anything to canonical** or push the content into a publication pipeline [29] . It stops at producing the provisional files.

At this point, the deep-learning portion of the workflow is complete and the ball is in the human court. The outputs (annotated text, challenger issues, etc.) will then await a human editor to review. The **human review** process itself (opening `crew_provisional.json`, examining the changes and notes, possibly editing `review_notes.md` with decisions) is outside the runner – but our runner has set everything up for it. It ensured that all AI outputs are clearly labeled as *CANDIDATE* or *PROVISIONAL* (not canon) and that any required human attention is signaled (e.g., tags like `[AMBIGUOUS]` or `[ESCALATE-HUMAN]` might appear in the content per agent design guidelines [56] [57]). The human reviewers will later bundle up these signals and decide on approvals or changes in a controlled session [58] , eventually creating a canonical version through a separate process in Phase-7.

In summary, the runner's architecture is a **linear orchestrator** that enforces input integrity (pre-flight), runs a *pipeline of AI agents* covering all editing capabilities, and outputs the results with all necessary context for governance. It is a self-contained routine for one excerpt, designed such that it could be invoked repeatedly for multiple excerpts (with different IDs each time). The workflow **never calls for human intervention mid-run** (agents never pause to ask a human – they instead mark uncertainties), aligning with the project's principle that human decisions happen *after* the AI run in scheduled reviews [59] [60]. This ensures that the automated workflow is reproducible and not gated by unpredictable human inputs in the middle, which is important for scaling and consistency.

Next, we detail **how each system capability is addressed** in this plan – covering the active agents we just walked through, and how we are handling the not-yet-active (designed or conceptual) capabilities in our design.

Active Pipeline Capabilities (Phase-6 Implemented)

The Mustikarasa system's core content processing capabilities are all **active and integrated** in Phase-6 [61] [62]. The runner will invoke each of these in turn as described above. For clarity, here is a summary of each active capability (agent) and how the runner addresses it:

- **Orchestrator (Workflow Management)** – **Active.** This isn't a separate AI model but the coded logic of the runner itself. The orchestrator function is fully implemented and acts as the *central coordinator* of the workflow [37]. It enforces the template structure of the content, triggers each agent in sequence, and logs every step, essentially serving as the "brain" of the operation [38]. Our new runner is built around this orchestrator role: it takes on responsibility for calling agents and applying governance rules (like stops and logging). By redesigning the runner from scratch, we ensure the Orchestrator logic is clean and aligns with Phase-6 expectations (e.g., handling excerpt metadata carefully, calling new governance checks, etc., which the old pilot orchestrator might not have done).

- **Translation Quality Agent** – **Active.** This agent ensures accurate translation from Indonesian to Dutch, preserving fidelity [39]. The runner will call this first (after initial setup), supplying the source text. The agent exists in working code (with a prompt) and will either produce a corrected Dutch text or validate the existing translation [40]. The runner incorporates its output into the workflow (replacing the initial text with the improved one). Because translation accuracy is foundational, any issues it encounters (e.g., untranslatable terms) might be flagged for later human review as `[AMBIGUOUS]` or similar – the runner doesn't resolve such ambiguities, but logs that they were noted, and continues.

- **Readability Editing Agent** – **Active.** This agent polishes the text for fluency and clarity in modern Dutch [41]. It is already implemented and is invoked after translation is confirmed. The runner passes the latest text to it. We ensure that the agent's rule of *"no unauthorized changes"* is followed [41] – basically, the prompt for this agent is constrained to only improve language, not meaning. The runner's role is simply to take the edited output and continue; however, if this agent were to misinterpret and change meaning (unlikely if prompt is correct), the fidelity check or challenger later would hopefully catch it. Regardless, the runner doesn't have to enforce that rule itself; it's baked into the agent's design, but we remain aware of it.

- **Cultural-Historical Annotation Agent** – **Active.** Adds cultural context annotations [42]. The runner calls this after the text is fluent. The agent's output (annotations) are integrated into the content or stored as a separate list. The runner ensures these annotations are kept with the excerpt's data (for example, in the `annotator_primary.json` we might have an array of annotation objects). Because the agent is active and tested, we expect it to produce helpful notes which the runner will simply log and include. This addresses the capability of enriching the text with background information automatically.

- **Structural Formatting (Book Structure) Agent** – **Active.** Ensures structural consistency (sections, formatting, numbering) [43]. The runner invokes this agent to check structural elements. If the excerpt had any structural issues (like a missing section header or numbering out of order), the agent will adjust or flag them. The runner then has to make sure that any structural fixes (like inserting a missing header) are reflected in the output. This might involve updating the markdown or JSON structure of the excerpt. By including this step, we address the capability that the digital edition's format will mirror the original book's structure accurately (important for a cookbook to have correct sections, etc.).

- **Recipe-Specific Editing Agent** – **Active.** Handles domain-specific consistency in recipes (ingredients, instructions) [44]. The runner uses this agent to parse and standardize recipe content. For instance, if an ingredient is listed slightly differently in two places, this agent might fix that. Or it could ensure all recipes follow a similar style (like imperative sentences for steps). The runner doesn't need special handling for this agent beyond passing the text; it simply incorporates any edits returned. Logging will note that "recipe content normalized" or similar. This covers the specialized editing that is unique to recipe texts, which is a key capability for Mustikarasa given the content type.

- **Table Handling Agent** – **Active.** Reconstructs and formats tables from the original document [45]. The runner calls this whenever the excerpt contains tabular data. Implementation-wise, the runner might need to detect tables (maybe via placeholders or markdown tables present) and feed that portion to the table agent. The agent will output a clean representation of the table. The runner replaces the placeholder or raw table with the formatted version in the content. Logging notes the table was handled. This ensures that nutritional tables, etc., in the book are properly captured (a capability that was error-prone with OCR and thus important to automate).

- **Image Integration Agent** – **Active.** Manages images and figures, linking them to text [46]. The runner will provide this agent with references to any images related to the excerpt. The agent might verify image files and produce captions or ensure the text references (like "see figure 3") are linked to actual image assets. The runner then includes those references or placeholders in the output. This capability means the digital edition won't lose track of illustrations from the original. Our runner addressing it by invoking the image agent ensures that part of the workflow is not neglected (even if an excerpt has no images, the agent can quickly noop; if it has, we cover it).

- **Continuity/Cohesion Checking Agent** – **Active.** Checks for internal consistency across the content [47]. The runner uses this agent near the end of the pipeline to validate that, after all edits, the excerpt still reads consistently and aligns with the rest of the edition. For example, uniform terminology, consistent narrative voice, no conflicting information. If the project has style guidelines (e.g., "use either metric or traditional units consistently"), this agent would catch violations. The runner will log any consistency issues flagged (such issues might also be passed to the challenger's

domain, but continuity agent is more systematic). By including this, we cover the capability of maintaining a coherent final product even though multiple agents may have touched the text.

- **Fidelity Assurance Agent – Active.** Flags meaning shifts by comparing final text to original [48]. The runner runs this as a safeguard at the end. The agent produces notes ("Opmerkingen") listing any spots where the meaning might have changed from the source. The runner captures these notes (likely as part of `crew_provisional.json` or as separate output file, but per spec it should be in the JSON). This addresses the critical capability of *ensuring authenticity*: even after extensive editing, we must confirm we haven't strayed from what the original says. The fidelity agent's output will be crucial for human reviewers to double-check those points. By integrating it, the runner helps fulfill the project's authenticity requirement [63] [64].

- **Design and Layout Agent – Active.** Applies stylistic consistency and prepares content for final publishing layout [49]. The runner will call this last (since it shouldn't override content decisions, only presentation). It might enforce things like a consistent tone, or ensure that formatting (italics, bold, etc.) meets the style guide, and that the output is suitable for print or web. For instance, it could check that line breaks are correct, or that a recipe title is properly marked up as a heading. Including this agent means the runner's output will not only be correct in substance but also look polished and ready for publication, aligning with the "coffee-table quality" aspiration for the public edition [65]. We will incorporate any suggestions it makes and log the final stylistic adjustments.

- **Critical Challenger Agent – Active.** Performs a secondary critical review of the AI's output [50]. Although listed last here, this runs in parallel to/after the annotator's main work. We described it in the workflow: the runner calls the challenger to critique the provisional output. This addresses the capability of "organized doubt" – having the AI challenge itself to ensure nothing obvious was missed [51]. The runner writing out `challenger_primary.json` is directly fulfilling the need to capture those critical questions. This agent effectively introduces a form of QA and is integral for catching subtle issues (thus improving reliability of the system's output before any human even sees it). By integrating the challenger, we adhere to the practice that every AI-generated result should be checked by an independent agent, which greatly increases trust in the output.

In summary, the runner covers **all active capabilities** identified in the system architecture [66] [67]. Each of the above agents is invoked in the proper sequence. The result is that by the end of a run, the excerpt has undergone translation verification, language polishing, contextual annotation, structural enforcement, domain-specific editing, table/image inclusion, consistency checks, fidelity verification, and a critical audit. **Nothing is left unaddressed in the content processing pipeline.** This ensures the output is as thorough and high-quality as possible, and any remaining issues are explicitly marked for human attention.

All of these steps and outputs are handled within the single runner script/workflow. The **pilot runner** we had before might not have included all these (for example, perhaps Glossary or Research were omitted, and maybe excerpt metadata handling was incomplete), but this new runner incorporates the full Phase-6 scope of active agents.

Next, we consider the **designed but not yet fully integrated capabilities** (governance and support agents) and how our runner accounts for them.

Designed Capabilities (Integration Pending in Phase-6)

Beyond the content-focused agents, Mustikarasa has a set of **governance and support capabilities** that are designed (prompts and plans exist) but were not fully part of the automated workflow in earlier phases. As we rebuild the runner, we want to **prepare for these capabilities** so that adding them will be straightforward when they become active. We will not leave them out of consideration, since the user explicitly wants to ensure all system capabilities are addressed. Here's how we handle each Phase-6 *designed* capability in our plan:

- **Methodology Archivist (Process Logger)** – *Designed (prompt ready, integration pending).* This agent is meant to produce a structured log of the methodology: essentially an audit trail of decisions and context during a run [68]. In the current system, some of this logging is done ad-hoc by the Orchestrator (runner) itself. The new runner will continue to log key events, but we are designing it such that the **Methodology Archivist** can be integrated with minimal changes. Concretely, once active, this agent would be called at the end of the run (or at key checkpoints) to output a formal **METHODOLOGY_LOG** artifact describing what happened and why [69]. For now, our runner might simulate this by writing comprehensive logs (perhaps in Markdown or JSON form) – essentially doing the archivist's job in a primitive way. We will keep the log format aligned with what the archivist is expected to produce. For example, if the archivist's prompt would include rationale for each agent's action, we ensure our logs have those rationale from the agents (we might parse agent outputs for tagged reasons). When the dedicated agent comes online, we can simply insert a call to it and have it generate the final log or augment ours. In short, the **capability of detailed methodology logging is addressed** in Phase-6 by the orchestrator's logging (as a stopgap) [70], and our runner is structured to seamlessly hand off this responsibility to the Methodology Archivist agent when it becomes active.

- **Technical Advisor (Model Strategy Advisor)** – *Designed.* This agent will provide advice on model usage, e.g. if a larger model is needed for a certain task or if some parameters should change [71]. It's not yet active in the live pipeline, but the design is there for it to output a **MODEL_ADVICE** note when invoked [72]. To address this capability, our runner will be built to **allow monitoring of model performance** during runs. For instance, the runner can track if an agent's output quality seems low or if it had to retry calls (which could hint at model issues). We can log those observations. In a minimal implementation, we might not automatically call the Technical Advisor each run (since it's not fully integrated), but we could include a **hook**: e.g., if a certain debug flag is set or if an agent signals a problem that might be model-related, we can invoke a (stub) technical advisor routine. That routine could, for now, log something like "(Technical Advisor placeholder)". When the actual agent integrates, it could be called at that hook to analyze model usage in the run and output recommendations (like "use GPT-4 for translation for better fidelity" or "model X seems to be struggling with OCR text") [73]. Our design ensures the runner is *model-agnostic* at baseline (able to switch models via config), so when Technical Advisor becomes active, it complements this by suggesting *which* model to use. In sum, we **acknowledge the Technical Advisor capability** and prepare for it: logging model info, providing a slot in the workflow (perhaps after an agent finishes, we could call Technical Advisor to evaluate that step). Until it's active, we rely on static config and human tuning, but the runner will not need a redesign to plug it in.

- **Troubleshooting Agent (Incident Responder)** – *Designed.* This is highly relevant to the runner's robustness. The Troubleshooting agent's role is to step in when an error occurs, log a structured

INCIDENT_REPORT, and prevent uncontrolled continuation [31]. The orchestrator (runner) is already *prepared to call it on errors*, as noted in the design [74]. We will implement exactly that: wrap agent calls and critical sections in try/except. If an exception or known error condition occurs (for example, an agent returns malformed JSON or a governance rule is violated), our runner will catch it. At that point, if the Troubleshooting agent is available, we call it with the context (error details, excerpt ID, etc.) to generate an incident report artifact. If it's not yet fully integrated, we still **log the incident** in the log with enough detail for a human to debug. The runner will then halt further processing, as per design, to avoid compounding the error [75]. By doing this, we ensure that the **capability of formal error handling** is addressed. No error goes unlogged, and the system will not continue in a faulty state. When the Troubleshooting agent becomes active in production, our error handler can invoke it to produce a JSON report (maybe saved alongside other outputs) with details like stack trace, which agent failed, etc., and suggestions for next steps. This approach means improved safety: even in Phase-6 pilot runs, we simulate the presence of a troubleshooter by careful exception logging, fulfilling the intention of that capability.

- **Glossary / Terminology Manager** – *Designed.* This agent manages the project's glossary: proposing new terms or ensuring consistent term usage across the content [76]. Its prompt and output format (e.g., a **GLOSSARY_PROPOSALS** list) are defined, but full automation is pending some governance rules for glossary updates [77]. To address this capability, we incorporate logic in the runner to **trigger glossary checks at appropriate times**. For instance, after the main content agents run (or perhaps at the very end), the runner can call the Glossary agent to scan the final text for any terms that might need glossary entries or that violate glossary standards. However, since integration is not complete, this might currently run in a "pilot" mode – perhaps logging suggestions rather than actually changing the official glossary. We know that currently glossary changes were handled semi-manually in pilot mode [78]. Our runner can mimic that: it could generate a draft glossary update file or log the terms that should be reviewed by glossary editors. By doing so, we **ensure terminology consistency is not overlooked**. For example, if the text introduced a new culinary term, the runner (via the Glossary agent or a placeholder) could output: "Term X not in glossary; suggested definition: ...". We would include that in the run outputs (maybe as part of review_notes or a separate glossary report). The human team can then take those proposals and incorporate them after proper approval. In future, once the Glossary Manager is fully active with governance, the runner would formally call it and possibly auto-update a glossary data store (with human sign-off). Our design keeps this in mind, so adding that functionality will be smooth. In short, we address the glossary capability by *optionally invoking a glossary consistency check* in the runner and by extending the workflow documentation to note when glossary agent should be consulted [79] (for example, if an agent flags a term as `[GLOSSARY?]`, the orchestrator knows to call the glossary agent).

- **Research / Historical Knowledge Agent** – *Designed.* This agent fetches and provides contextual knowledge, producing a **RESEARCH_REPORT** to enrich annotations and decisions [80]. It has been tested in some pilot scenarios but is not yet part of every run [81]. The idea is that if an excerpt or an ambiguity calls for outside knowledge (e.g., the historical background of a recipe, or confirming a fact), the Research agent can retrieve information from reference sources and present it. To integrate this capability, our runner will include **conditions or triggers for research**. For example, if during the run an agent flags something as `[REQUIRES-RESEARCH]` (perhaps the annotator or challenger might do so for some uncertain claim), the runner can then invoke the Research agent with a query. In the Phase-6 pilot, research wasn't routine, but design documents suggest tying research and glossary together for richer outputs [81]. We plan for that by making the runner

capable of calling the Research agent and including its report in the outputs. Perhaps initially this is off by default (to save time or because the agent might not be reliable yet), but the capability is addressed by *designing the runner to allow a research step*. For instance, we could have a config flag `enable_research: true/false`. If true, after the cultural annotation step (or when an ambiguity is found), the runner calls the Research agent with context (like "Topic: what is this ingredient's history?") and then attach whatever it returns as an addendum in `annotator_primary.json` or a separate `research_notes.md`. During early Phase-6, we might run it in a read-only fashion (not affecting main content) just to see what it provides [81]. This is consistent with how a pilot was done to see research and glossary interplay. Summarily, our runner will **support the research capability** by providing a slot for it, even if the results are advisory. By Phase-6 completion, if research agent is stable, we can integrate it fully so that each run is enriched with contextual evidence, which will greatly help human reviewers and adds to the system's credibility.

- **Repository Archivist / Documentation Admin** – *Designed.* This role is meant to handle end-of-run tasks like committing changes to a repository, updating indices, and managing documentation of changes [82]. It's more about wrapping up the run's outputs into the project's knowledge base. While less detail is given about its implementation, it's expected to become important as the project moves from isolated pilot runs to continuous operations (where each run's outcome needs to be saved and indexed) [83]. Our runner will address this by **ensuring outputs are easily commit-ready and by possibly automating some repository steps**. For example, after generating outputs, the runner could have an option to auto-commit the new files to the version control system (e.g., git) along with a message. Or it might call a script to update a central index of processed excerpts. Since in Phase-6 this agent is not active, we won't implement a full auto-commit (which could be risky without human overview). But we will create a **post-run hook** where such actions could occur. Perhaps we generate a summary file or log entry that a documentation update is needed. We also maintain an organized output directory (already specified) so that a separate archival script can easily find new results. Essentially, we **design for smooth hand-off to the Repository Archivist**: everything the runner produces is in known locations with consistent naming, which makes it straightforward for a future automated archivist to do its job. When the time comes, integrating that agent could mean just adding a call like `repository_archivist.finalize_run(run_id, excerpt_id)` at the end of our runner. This would handle the commit and housekeeping. Until then, those tasks might be done manually by the development team, but the runner's outputs are such that manual archival is trivial (just commit the `run_outputs` directory, etc.). Thus, this capability is addressed conceptually and the path to implementing it is prepared.

By accommodating all these designed-but-pending features, we **future-proof the runner**. We ensure that as each of these governance or support agents reaches maturity, the runner does not require a fundamental rewrite to include them. Instead, it will be a matter of plugging in the new calls or toggling them on. This approach keeps our Phase-6 runner aligned with the system's growth, covering not only what we have now but what is around the corner.

Future Conceptual Capabilities and Considerations

The Mustikarasa architecture also mentions some **conceptual future capabilities** that are not yet fleshed out in design. While these are beyond Phase-6, it's worth mentioning how our runner design **keeps the door open for them**, ensuring we don't inadvertently preclude their later addition:

- **OCR / Source Integrity Agent (Phase-0 Automation)** – *Concept.* In early phases, much work was done to clean up OCR errors in the source scans manually. A future concept is an AI agent to automate scan quality control and detect OCR errors by cross-referencing scanned images [84]. This would operate *before* the main editorial workflow (essentially on the raw source text). In our runner's context, this capability would be utilized prior to even Phase-1 translation. While our Phase-6 runner doesn't directly include OCR processing (it assumes we already have a "locked" excerpt text to work on), we remain aware of this future need. If in the future the pipeline starts with an OCR agent, that might be a separate process that generates the excerpt text and certifies it. Our runner can then start after that. So to address this concept now, we simply ensure that our input (`excerpt_source` and `excerpt_version`) points to a text that has gone through whatever source integrity checks exist. In Phase-6, that might mean the input texts are those "locked-2026-01-05" versions that have been manually corrected [85]. The runner doesn't need to replicate OCR logic, but by requiring an `excerpt_version` tag, it implicitly supports source integrity (the version tag indicates the source text's integrity status). When the OCR agent is eventually built, it will likely produce these locked versions, and our runner already consumes them via the metadata. So, we consider this capability acknowledged – just not executed by the runner. It's a separate upstream step.

- **Batch Governor (Batch Orchestrator)** – *Concept.* As the project scales up to process the entire book or large batches of recipes, a Batch Governor agent is envisioned to manage running multiple excerpts in sequence or in parallel [86]. Instead of launching one excerpt at a time manually, the Batch Governor would oversee the queue of excerpt runs, handle any dependencies, and manage aggregate results. Our Phase-6 runner is intentionally focused on a single excerpt ("excerpt-aware" runner). However, we have structured it such that it can be invoked programmatically multiple times. The **Batch Governor, when it exists, could call our runner for each excerpt** – for instance, spinning up parallel processes each running `run_excerpt_workflow.py` on different excerpts, or orchestrating them one after the other and collecting statuses. We have designed the output layout in per-excerpt folders, which the Batch Governor can easily collate or monitor. We also considered error isolation: one failing excerpt run will not crash the whole system – the runner stops itself and returns an error code, which a Batch Governor can catch and then decide (maybe skip that excerpt or log a batch-level incident) [87]. The conceptual Batch Governor capability highlights needs like ensuring one recipe's failure doesn't halt the entire book processing [88]. Our runner's practice of halting on errors but containing them, and writing incident logs, supports that – a Batch orchestrator can just move to the next item if one fails, since the failure was neatly handled and logged. We also output all runs in a structured way, making it feasible to combine outputs later (e.g., compile a whole chapter). In short, our single-excerpt runner is a *modular component that fits into a future batch processing framework*. When Mustikarasa enters Phase-7 or beyond, implementing the Batch Governor will not require changes to the runner, only an additional layer on top. Thus, we address this future capability by **designing the runner as a reusable, isolated unit** that can be scaled out.

- **Meeting/Redaction Agent (Collaborative Review Simulator)** – *Concept.* This forward-looking idea suggests an AI agent to facilitate meetings or discussions among multiple human experts and the AI system [89]. It might summarize review meetings or orchestrate multi-party conversations to make final editorial decisions. This is more relevant in a scenario where multiple stakeholders review the AI outputs together (like an editorial board meeting). While this is beyond our runner's current scope (which deals with automation up to the human hand-off), we ensure nothing in our design prevents such a workflow. For example, if later the project implements a "Meeting agent" that takes several `crew_provisional` outputs and human comments to produce a consensus or minutes, our runner's output format should be rich enough to feed that process. By including things like review_notes and comprehensive logs, we make sure that any downstream collaborative tool has the data needed. Additionally, the *redaction* aspect (which in Dutch context refers to editorial decision-making) might tie into how the final canonical decisions are recorded. Our runner already ensures traceability so that a future Meeting agent can point to **"why something changed"** for each excerpt [90]. Essentially, while we don't implement the Meeting agent, we keep the system **traceable and modular**, enabling such higher-level coordination in the future. We can imagine that after a human review session (possibly aided by such an AI), the results (canonical decisions) feed back into the repository. The runner's role in that future would likely remain the same (producing provisional outputs), so no change is needed except to ensure we capture all the info a Meeting agent would need (which we do: all changes, all uncertainties, all metadata).

In summary, the **conceptual capabilities** like OCR automation, batch orchestration, and AI-assisted human collaboration are acknowledged in our plan. We design the runner as a component that can be extended or invoked by these future systems, rather than a monolithic pipeline that would conflict with them. This means our Phase-6 runner is aligned with the *"architectural runway"* described in the system overview [91] [92] – the foundations we lay now won't need rework when new capabilities land; instead, they can build on what we have.

Technical Implementation and Platform Considerations

Having described *what* the runner will do, we also outline *how* we will implement it, keeping in mind platform and technical environment:

- **Language and Framework:** The runner will be implemented in **Python**, consistent with the rest of Mustikarasa's tooling (the architecture notes that the system is implemented in Python with custom scripts managing the agents) [93]. We will likely create a Python module or script (e.g. `run_excerpt_workflow.py`) as indicated by the pilot logs [94]. This script will use the existing Mustikarasa agent framework – possibly there are modules like `mustikarasa_agents.py` that provide classes or functions for each agent, as hinted in the architecture doc [10]. We will leverage those where available rather than coding each prompt from scratch. For example, if there's a function `run_translation_agent(excerpt)` already, the runner will call that. If not, we will implement each agent call following the prompt specifications documented (ensuring outputs are machine-checkable JSON as per prompt guidelines [95] [96]).

- **Platform (Mac Mini vs Agnostic):** The development and initial execution environment is a Mac Mini (likely with an Apple Silicon chip). We must ensure the runner runs smoothly on macOS. However, we keep it agnostic by avoiding any Mac-specific dependencies. Python is cross-platform, so as long as we don't use things like Apple-only libraries, we're fine. For instance, file paths and OS operations will

use Python's standard libraries (which handle Windows/Linux/Mac differences). We also consider that the Mac Mini might be used to run local models. There is mention of **CrewAI/Ollama** integration for local LLMs [97]. On a Mac (especially Apple Silicon), tools like Ollama allow running models like Mistral or Llama 2 efficiently. Our runner should be able to utilize these if configured. That means our code might interface with a local inference server or library when an agent runs. We'll design it such that the **model backend is configurable** – e.g., an agent call could abstract "call LLM with prompt X", and depending on config, it either hits OpenAI's API or a local endpoint. This model-agnostic approach is explicitly part of the architecture [98]. It ensures that whether on Mac Mini (with local models) or on another machine (perhaps using cloud APIs), the runner works the same.

- **Dependencies:** We'll manage necessary libraries for JSON handling, HTTP requests (if needed for APIs), etc. One area to watch is the LLM integration – if using OpenAI API, we'll use the OpenAI Python SDK; if using local (like Ollama), we might use an HTTP call to Ollama's local server or its Python bindings. We will design an interface for agent inference that can be swapped easily.

- **Parallelism and Performance:** Currently, the runner will operate in a mostly **sequential** manner (agent by agent). Given that all these agents are AI model calls, the runtime might be significant for each excerpt (could be minutes per recipe). The architecture notes that adding agents like Research can slow things, and suggests optimizing or parallelizing where possible [99]. In our implementation, we might consider minor parallelism (for example, some tasks could potentially run concurrently if they don't depend on each other – maybe image processing and table processing could theoretically overlap). However, most of our agents have logical order dependencies (you want translation done before readability, etc.), so we will keep it sequential to maintain determinism and traceability. We prefer clarity over micro-optimizations at this stage. Where we can improve speed is by allowing the use of faster models or configurations as needed. Since the Technical Advisor capability is not active yet, we rely on static configs: e.g., use a smaller model for certain simpler tasks to save time. We will make these model choices configurable via the YAML config (`sandbox/crew/configs/*.yaml` as seen in logs) so that on a Mac Mini (which might have limited VRAM), one can choose lighter models for now. For example, maybe use a local 7B model for some agents, and call OpenAI's larger model only for fidelity check where precision matters more. The **Phase-6 strategy prioritizes traceability and safety over raw speed** [32], so it's acceptable if the runner is somewhat slow; we just need to ensure it's stable and correct. In the future, if performance becomes a bottleneck, we can incorporate multi-processing or move certain agents to asynchronous calls. The architecture already envisions possibly running heavy analyses offline or using asynchronous patterns for slow steps [34] – we'll keep that in mind (for example, maybe the Research agent call could be done asynchronously if it's very slow, allowing a human to retrieve it later, etc.). But for now, simplicity is key.

- **Version Control and Traceability:** The entire project (including the code and outputs) is under version control (the mention of markdown logs and CODEX_SESSION_LOG suggests a Git-based repo with chronological logs) [100]. We will integrate the runner into that practice. That means after the runner finishes, developers (or the automated archivist later) will commit the new outputs and logs. Our logging format and file outputs already follow the conventions needed for that, as discussed. We might also increment some index or log of runs. Possibly, the CODEX_SESSION_LOG.md is a running list of all runs performed [100]. We can have the runner append an entry to that file each time it runs, summarizing what was done (this could be done by calling a helper or just instructing users to do it manually for now). Given the focus on traceability, we won't forget to include information like

the exact prompt versions or model versions used in the run if possible. For example, we might log: "Using model GPT-4 for Translation agent, prompt version v5.2" etc., so that the run can be reproduced exactly in the future. This information may come from the config file or from agent definitions. By recording it in the log (and possibly in run_metadata.json if we include such a file), we align with the idea that anyone can later see what models and settings produced a given output [100].

- **Testing and Verification:** Before deploying this runner widely, we will test it on at least Case-01 (the Sayur excerpt) and Case-02 (Lobak excerpt) which have been used in pilots. We expect to verify that all Phase-6 design requirements are met (e.g., does the log file actually show `mode: excerpt-aware` and all fields? Do the JSON outputs include the excerpt id and run id?). We can use the **Phase-6 Case-01 readiness plan** (referenced in docs) as a checklist [101] [102]. Any discrepancies (like if an output file is misnamed or a metadata field missing) will be fixed. We also confirm that the runner's outputs can be consumed by the next steps (for instance, a human tester will take `crew_provisional.json` and run through the human review process defined, making sure nothing is hindering them).

- **User Interface (UI) Integration:** While not directly part of the runner, it's worth noting that there are "Human UI" documents in the project. Possibly in the future, a UI might be built to trigger runs and display results. Our runner should be designed to be usable in such contexts (e.g., callable via an API or CLI that a UI could invoke). That means keeping input/output clear and not requiring interactive prompts. It runs to completion autonomously and writes outputs to known locations, which a UI or a web service could then present to users (like showing diff of "how it was vs how it is and why" — which our logs and outputs support). Thus, we indirectly support UI capabilities by following the contracts.

By covering these technical aspects, we ensure that the **Phase-6 runner is not only conceptually sound but also practical to implement and run** on the given hardware and software stack. The Mac Mini will serve as our test bench, but the runner's portability means we could run it on a Linux server for more power or on CI/CD pipelines for automated testing. The Pythonic, modular approach aligns with the team's workflow and the enterprise architecture guidelines.

Conclusion and Next Steps

In conclusion, the rebuilt Phase-6 excerpt-aware runner will be a **comprehensive orchestration tool** that aligns with Mustikarasa's vision of a traceable, agent-driven editorial process. We have outlined how it will function from start to finish, enforcing metadata integrity, invoking each of the system's active capabilities in sequence, and producing standardized, review-ready outputs. We also accounted for all other capabilities in the system's design: those already implemented are integrated, those in design phase are anticipated via hooks and logging, and even future concepts have been considered in the architecture so that nothing in our runner will hinder the project's evolution.

All capabilities of the system have been addressed: The content transformation agents ensure the book's text is handled expertly (translation accuracy, editing quality, annotations, etc.), and the governance and support agents (whether active or planned) ensure the process around those transformations is safe and well-documented (with logging, error handling, terminology management, and so on). The runner is effectively the **convergence point** of these capabilities – it coordinates the "crew" of AI agents and records

their collective work, living up to the project's ethos of *"no black boxes"*, where every action by the AI is transparent and reversible.

With the deep research and Phase-6 documentation as our guide, the next steps would be to proceed with implementation. This involves: 1. Writing the new `run_excerpt_workflow.py` script (or equivalent module) following the plan, and replacing any legacy pilot code. 2. Ensuring all agent prompts and logic are up-to-date with Phase-6 specs (e.g., adopting the excerpt-binding spec so that excerpt_id appears everywhere it should [4]). 3. Testing the runner on known excerpts (Case-01 "Sayur" which is likely already used in pilots) and verifying that the outputs match expectations (for instance, logs go to `sandbox/crew/run_logs/sayur_...`, JSON files contain correct metadata, etc.). 4. Iterating on any issues (if, say, the challenger output needs a tweak in how we merge it, or if performance on Mac Mini requires adjusting model choices). 5. Documenting the runner usage for the team – how to run it, what it produces – likely updating the project README or technical docs (like *TECH_ARCHITECTURE_V2_EXECUTION_READY.md* presumably covers this new design).

By implementing this plan, we will achieve an **"execution-ready" technical architecture (v2)** for Mustikarasa, wherein the runner is robust enough to carry out real excerpt runs under governance constraints [91]. This forms a solid foundation for Phase-7 and beyond, where the focus will shift to handling the full book and formalizing human approvals (canonical decisions). Thanks to the groundwork in the Phase-6 runner, the system will be ready to scale up in a controlled and traceable manner [103] [104]. The end result will be an AI-assisted editorial pipeline that can reliably produce a high-quality digital edition of *Mustika Rasa*, with every change tracked from raw scan to polished text, and with humans firmly in charge of the final product.

---

[1] [2] [3] [10] [19] [31] [32] [33] [34] [37] [38] [39] [40] [41] [42] [43] [44] [45] [46] [47] [48] [49] [50] [51] [61] [62] [63] [64] [65] [66] [67] [68] [69] [70] [71] [72] [73] [74] [75] [76] [77] [78] [80] [81] [82] [83] [84] [86] [87] [88] [89] [90] [91] [92] [93] [97] [98] [99] [100] [103] [104] Mustikarasa System Architecture Overview (TOGAF-Based Approach).pdf
file://file_00000000025071f4a6714b3d3c5b6a2e

[4] [79] [94] GOV_LABEL_AUDIT.md
file://file_00000000c85471f4b8f874499d8fafdb

[5] [6] [7] [11] [12] [13] [15] [21] [22] [24] [25] [26] [27] [35] [36] [54] [85] [101] [102] P6_EXCERPT_AWARE_RUNNER_IMPLEMENTATION_GUIDE.md
file://file_00000000131c720a83ac8603b545f2a0

[8] [28] [30] [52] [53] [55] [56] [57] [58] [59] [60] P6_HUMAN_REVIEW_WORKFLOW.md
file://file_00000000c6e871f4ad0dc1f3bc485c5d

[9] [29] P7_CANONICAL_TRAIL_SPEC.md
file://file_000000000d0871f4ac192369c2ad6118

[14] [16] [17] [18] [20] [23] P6_RUNNER_OUTPUT_LAYOUT.md
file://file_000000000fa071f49117c3ee2c9dec29

[95] [96] AGENT_PROMPT_PATTERN_P5.md
file://file_00000000d1fc71f4815ac4ff2eb93af8