ChatGPT

Novelist Project Recovery & Redesign Plan

This plan addresses the context loss issues identified in the Novelist project audit and proposes a multi-phase redesign. It is organized by major subsystems with specific recommendations for phase ordering, context integration, schema updates, and guardrails to ensure continuity from outline through final draft. Implementation phases and a **reseed payload** strategy for recovering lost context are included at the end.

Premise & Project Metadata

- **Centralize Premise Info Early:** Ensure the story's premise, genre, style, and other metadata are captured at the very start of the workflow. This data should become a persistent reference that all subsequent prompts can access. For example, the **premise summary, theme, tone, and target audience** should be stored in a project state object and passed into each generation step (e.g. as part of the prompt or via a shared data model). This guarantees that high-level guidance (like genre conventions or narrative tone) remains consistent throughout the process.
- **Propagate Metadata to All Phases:** Integrate the premise and global settings (e.g. chosen writing style or rule set) into every phase's prompt. If the user selected a style guide (such as "Colleen Hoover" style), include a brief reminder or tag in prompts for characters, arcs, and beats. This could be done by adding a section in the prompt like: `[Style: Colleen Hoover, Genre: Contemporary Romance]` to reinforce consistency. All subsystems should be aware of these project-wide settings to avoid drift in voice or genre fidelity.
- **Metadata Schema Linkage:** Update the data schema to explicitly carry premise and global metadata through the outline structure. For instance, the outline JSON could have a top-level `"premise"` field or embed a reference to a style guide. In the original prototype, there was a separate `style_guide.schema.json` [1], but linking it into the outline artifact will ensure style rules and premise are readily available when generating beats and chapters.
- **Developer Notes:** *Module impact:* The initial wizard or CLI that gathers project info should compile a **Project Metadata** object. This likely affects the input handling portion of the CLI. Ensure that `prompt_builders` includes this metadata when constructing prompts for subsequent stages. The `novelist_cli.models` may be extended to hold global metadata (e.g., a `Project` model containing premise, genre, style) [2]. This might shift some responsibility to earlier in the pipeline, but it guarantees later modules have access to important context. Logging can be enhanced to confirm that each prompt includes expected metadata tags.

Character Generation

- **Reorder Before Arc Planning:** Generate the main characters immediately after establishing the premise (and before story arc planning). In the current flow, characters were generated later, which could lead to arcs being planned without solid character identities, causing misalignment. By moving **Character Generation** earlier, story arcs and plot beats can be designed around well-defined characters with established traits and motivations. This reordering ensures characters are not retrofitted into the plot, but rather drive the plot from the beginning.

- **Integrate Premise and Roles into Prompt:** When prompting the model to create characters, include relevant context such as the premise, setting, and the story's central conflict. For example: *"Based on the premise (X) and genre (Y), generate a list of main characters with names, roles, and key traits that suit the story."* If certain character archetypes or roles are implied by the premise (e.g. a mentor or love interest), mention those so the model includes them. This addresses missing context where earlier iterations might not mention the plot premise during character brainstorming, leading to characters that drift from story needs.
- **Canonical Character IDs & Traits:** Assign each character a unique identifier (or use consistent names/titles) and maintain a **canonical profile** for them. Store these profiles in a structured form (e.g. a JSON list of characters with attributes). Subsequent outline phases should refer to characters via these identifiers or exact names to avoid confusion. For example, if a character "Alice" is described as *shy and clever* in the character sheet, the same name and traits should be referenced in arcs and beats. An architectural change might be to include a character reference in each beat (e.g. `beat.characters = ["Alice", "Bob"]`) to explicitly link which characters are involved in that beat. This way, even if the narrative text of a beat doesn't mention Alice's trait, the system knows she's there and can remind the drafting prompt of her persona.
- **Guardrails – Trait Propagation:** To prevent character identity drift (e.g. a character's personality or background changing between outline and draft), implement prompt guardrails such as **trait checklists**. For instance, when generating a scene or chapter involving a particular character, append a bullet list of that character's key traits or recent continuity notes: "**Checklist for Alice:** remains shy in new social settings, uses clever solutions to problems." This reminds the model to keep her behavior consistent. Another guardrail is to use descriptor tags in the prompt (e.g. `<Alice: shy, clever>` before a scene description) that the model can parse as context. These tags act as hard anchors for the model to adhere to established characterization.
- **Developer Notes:** *Module impact:* The character generation likely lives in the prompt builders or wizard logic. After moving it earlier, ensure the **Story Arc & Act Planning** phase can access the list of characters. That means passing the character list into the arc planning prompt (likely via `prompt_builders.py`) [3]. Data models (`models.py`) should be updated to include a **Character** entity and possibly to attach character references to plot points. This may require adjusting how the outline JSON is constructed (to include character info per act/beat). The code boundary between character generation and arc generation will shift: the arc generator must wait for character generation to complete. In practice, you might refactor the CLI or workflow orchestrator (`__main__.py`) to call character generation first, then feed those results into the arc planner. Confirm that tests (if any exist for models or continuity) are updated to reflect the new order (e.g., test that arcs reference known characters).

Story Arc & Act Planning

- **Leverage Characters and Premise in Arc Planning:** With characters generated, feed them into the arc planning prompt. The arcs (overall plot outline, typically broken into acts) should be **character-driven**. For example, when prompting for a three-act structure, include a summary of each main character's goal or arc so that Act I, II, III incorporate character development. This addresses previous omissions where arcs might have been too high-level or disconnected from character progression. The prompt could say: *"Using the premise and characters above, outline the story in three acts, ensuring each act reflects the characters' internal and external journeys."* Missing context like character motivations or backstory must be present so the model can weave them into each act's summary.

- **Include Subplot Hooks:** If subplots are part of the project (see next section), the act planning should include placeholders or mentions for those subplot threads. For instance, instruct the model: *“Also introduce the subplot arcs (if any) within the act structure.”* This ensures that from the top-level outline, subplots are acknowledged and tied into the main narrative flow, preventing them from being dropped later. Each act summary could explicitly note how the subplot progresses in that act (e.g., “Act II: … Meanwhile [Subplot: Bob’s secret investigation intensifies].”).
- **Schema Enhancement – Link Arcs to Characters:** Consider enhancing the outline schema to capture relationships between acts and characters. For example, each act could list which characters are primary in that act or what each character’s state/change is by that act. This creates a **canonical linkage**: the arc plan isn’t just a freeform text, but a structured data pointing to characters (e.g., `act1.majorCharacters = ["Alice","Bob"]`). Architecturally, this might involve extending the outline JSON format to embed references or IDs from the character list. The original `outline.schema.json` likely defined acts and chapters but not an explicit link to character objects [4] – adding this can enforce continuity (the system can verify that every main character appears in the outline).
- **Guardrails – Prompt Templates for Acts:** Use a prompt template that includes a **checklist of elements** to cover in each act. For example, a template might list: “Each Act summary **must** include: the key conflict, how it escalates or resolves, the state of each main character, and any subplot developments.” This checklist (possibly commented or in a system prompt) can reduce the chance of missing pieces like a subplot. We can also tag the output requirement, e.g., “**[Ensure Act plans remain consistent with character arcs and include subplot threads]**” as a reminder in the prompt. In effect, we’re injecting a self-review directive for the model.
- **Developer Notes:** *Module impact:* The act planning is likely handled by a generator function (perhaps within prompt_builders or a dedicated outline generator). After reordering, it should accept the characters data structure as input. Changes needed: update the prompt construction in `prompt_builders.py` [3] to append character info (names, roles, arcs) when calling the model for story arc generation. The output format might need adjustments; if we decide to structure the output (like JSON with act keys or a list of acts), update `outline.schema.json` accordingly and the parsing logic. Ensure that any summarization or truncation (possibly done by a `summarizer` in the pipeline) does not strip out character or subplot mentions – if summarizer is used to condense outlines for tokens, instruct it to keep all proper nouns and plot points. We might modify the `draft_driver/summarizer.py` to preserve certain fields or markers [5] to protect vital context.

Subplot Threading

- **Explicit Subplot Development Phase:** Introduce a dedicated phase (or explicit handling) for **Subplot generation and integration**. If not already separate, treat subplots as first-class citizens: after or alongside act planning, prompt the model to outline subplots (e.g., secondary storylines, character arcs for side characters, etc.) in a structured way. This could be a list of subplot descriptions or a timeline of each subplot across acts. Doing this explicitly prevents subplots from being implied and then lost; instead, they are documented and can be referenced in subsequent steps.
- **Integrate Subplots with Main Outline:** Once subplots are outlined, merge them into the main outline structure. For example, mark certain beats or chapters as “subplot beats” or tag them with which subplot they advance. The outline JSON could have a field in each beat like `"subplot": "BobInvestigates"` or similar identifier. During beat generation (scene-level detailing), include these subplot tags so the model remembers to include that narrative thread. By weaving subplot

references in early (acts) and often (beats), we avoid the issue where subplots identified early on disappear by the final draft.

- **Context Inputs for Subplot Prompts:** When generating subplots, include context about how they relate to the main plot and characters. For instance: *“Generate 2 subplots for this story: one focusing on Alice’s internal struggle, and one comedic side-plot involving Bob’s coworker. Ensure they complement the main plot’s theme.”* This prompt would use premise and character info to produce relevant subplots. Then, when integrating, explicitly prompt the model to **thread** these subplots through the acts: *“Integrate the subplot ‘Alice’s internal struggle’ such that it progresses in parallel with the main plot across the three acts.”* By being explicit, we supply the model with the missing link that each subplot has a progression alongside the main arc.
- **Schema & Data Model Changes:** Augment the data schema to record subplots in the outline. For instance, add a top-level `"subplots"` list in the outline JSON, each with its own identifier and summary. Additionally, allow each beat or chapter entry to reference which subplot(s) it touches on. This formal linkage acts as a checklist that can be validated: e.g., if we have 2 subplots, each should appear in some form in each act or at least have a beginning, middle, end in the outline. The goal is a **bidirectional traceability**: from subplot entry to the beats that implement it, and vice versa. The prototypes had tests for artifact validation [6] – these can be extended to verify that every subplot identified is actually present in the beats and draft.
- **Guardrails – Subplot Continuity Checks:** Employ automated checks or prompt-based verifications to ensure subplots don’t vanish. For example, after outline generation, run a simple script (or even an LLM-based check) to scan the outline and list where each subplot is mentioned. If a subplot is missing in later acts, flag it. On the prompt side, when drafting chapters, use the subplot tags to remind the model: *“In this chapter, make sure to include progress on subplot X.”* Another guardrail: use the summarizer or an assistant prompt to recap subplot status between chapters (“Recap: Alice’s struggle subplot last left off when…”) and feed that to the next chapter’s prompt so the model remembers to continue it.
- **Developer Notes:** *Module impact:* Depending on current implementation, you might have to create a new generator for subplots or adjust existing outline generation to produce subplots. This could be an extension of the outline JSON or a separate artifact that is later merged. Code-wise, this might involve new functions in `prompt_builders.py` or a new section in the wizard flow. The merging of subplot data into beats will affect the beat generation logic (likely in the beat generator or chapter structuring code). Consider updating the `outline.schema.json` to include subplots formally (if it isn’t already). Tests (`validate_artifacts.py` and others) [6] should be updated or added to ensure that for any outline with subplots, those subplots are present in the final manuscript. This may also entail updating the `build_manuscript` logic [7] to handle multiple threads more explicitly, perhaps by generating each subplot’s content and splicing or by guiding the chapter assembly to cover each thread in turn.

Beat Generation

- **Context-Rich Beat Prompts:** When generating beats (detailed events or scenes within chapters/acts), use **all relevant context as input**. Each beat prompt should include: the act it belongs to, the summary of that act, the specific goal of the beat, the involved characters (with traits), and any subplot or foreshadowing that needs to occur. In earlier iterations, beats were sometimes generated from a minimal prompt (maybe just the act summary), leading to omissions of context. Enriching the prompt ensures the model knows the full situation. For example: *“Act II, Beat 3: Alice confronts her fear at the school dance. Context: Act II summary …; Characters: Alice (shy, clever), Bob (supportive friend);*

*Subplot: Alice's secret tutoring of Bob revealed at the dance. Generate the beat details…***" – This prompt explicitly anchors all context, leaving less for the model to infer (or forget).

- **Sequential Dependency & Memory:** Ensure that beat generation is sequentially informed. A beat is not standalone; it should consider the prior beat and lead into the next. To enforce this, include a brief summary of "what happened in the previous beat" in the prompt for the next beat. This can prevent logical jumps and maintain cause-and-effect. If the system currently generates all beats in one prompt, consider switching to iterative generation (one beat at a time with context from previous beats). Alternatively, if generating all in one go, instruct the model to maintain coherence and explicitly list each beat in order. The key is to avoid context failure where a beat doesn't acknowledge what came right before.
- **Integrate Checklist for Beats:** Use a prompt checklist to verify each beat covers required elements. For example, for each beat, have a bullet list in the prompt: "- Which character's arc is advanced here? - What subplot is touched? - What is the conflict/outcome of this beat?" The model should answer these implicitly by writing the beat. By posing these questions in the prompt (even if as comments or just mentally), it primes the model to include those details. This technique was not present in earlier designs and adding it will guard against a beat that, say, has action but no character development when it should.
- **Schema – Beat Linking:** Modify the outline data structure for beats to carry more context fields. For instance, extend each beat entry with keys like `"characters": []`, `"subplot": ""`, `"conflict": ""`, `"resolution": ""` as applicable. These act as a form of **embedded documentation** of each beat's purpose. The drafting engine can then consume these structured fields to ensure it writes them into the narrative. This is an architectural change: it means the outline is richer and the final draft generator needs to map these fields into prose. But it significantly reduces ambiguity. (If the outline JSON already partially does this, reinforce it and ensure the fields are always populated. If not, add them and adjust prompt generation to fill them.)
- **Guardrails – Automated Beat Validation:** After generating beats, run a check to compare the beats against the arc and character list. For example, verify that every main plot point from the act summary appears in at least one beat, and that every main character has some involvement per act. If anything is missing, either loop back and regenerate that beat with added emphasis or log a warning for the writer. The existence of a `validate_artifacts.py` test [6] indicates the system can be extended to validate such conditions. Even a simple script that flags "Beat missing subplot mention where expected" can alert the developer or trigger a corrective prompt.
- **Developer Notes:** *Module impact:* The beat generation likely happens in either an outline refining function or the draft phase. We may need to break it out: e.g., have an explicit `generate_beats(outline)` function that we can enhance. This function should now accept a fully populated outline (with characters, subplots, etc.) and iterate through acts and chapters to produce beats. Code changes include updating `prompt_builders.py` (or wherever beat prompts are built) to include the richer context. Also, if beats are generated as part of the outline JSON, ensure the schema and validators accept the new fields (characters, subplot, etc.). This could involve updating `outline.schema.json` and adjusting any code that reads/writes outline files. The **module boundaries** might shift: previously, maybe the outline and beats were done in one go; going forward, it might be outline (high-level) then a separate pass for beats. Plan accordingly for the orchestration (in `novelist_cli._main__` or wizard logic).

Chapter Structuring

- **Group Beats into Chapters with Continuity:** After beats are generated, the system should organize them into chapters in a way that preserves narrative flow. Chapters are higher-level groupings (possibly each act could be 1+ chapters). The key redesign here is to ensure that when transitioning from one chapter to the next, context is not lost. To do this, **carry over a summary of the previous chapter's ending** into the prompt for the next chapter's opening. For example, when concluding Chapter 1 and starting Chapter 2, the Chapter 2 generation prompt could include: *"End of Chapter 1 recap: Alice agreed to go to the dance with Bob, but is anxious. Now write Chapter 2…***". By summarizing the prior chapter, we maintain continuity of plot and emotional state.
- **Ensure Chapter-Level Arcs:** Each chapter should roughly correspond to a mini-arc or a set of beats that have a small narrative climax. In the outline, mark where chapter breaks occur explicitly and why. E.g., after a particularly big turning point beat, insert a chapter break. This information (the rationale for a chapter break) can be included in the prompt for the chapter generation to help the model pace the narrative (e.g., "Chapter 2 should cover Alice's low point after the dance fiasco, ending on a hopeful note as she finds a clue about…."). This was likely missing in the prior implementation, where chapters might have been formed simply by chunking beats without explicit guidance, risking awkward pacing. Make the chapter structuring an intentional step with its own logic and context.
- **Schema and Data Flow:** Represent chapters explicitly in the data model. The outline might already group beats under chapters or acts; if not, introduce a structure: e.g., `"chapters": [ { "title": "Chapter 1", "beats": [ ... ] }, ...]`. If chapters are just sequential, decide how to split beats into chapters (maybe every X beats or at act boundaries). A better approach: design a heuristic or let the model propose chapter breaks (e.g. a prompt: *"Propose a chapter structure given these beats"*). Once chapters are delineated, store any chapter-specific context (like point-of-view or setting changes) in the structure. That way, the drafting engine knows, for instance, Chapter 3 is from Bob's POV and can maintain that style internally. This structural clarity will fight context confusion where the model might otherwise forget which POV or tone was last used.
- **Guardrails – Chapter Transition Checks:** Implement checks between chapters to catch continuity errors (e.g., a character disappearing or time skipping without explanation). For example, automatically compare the last beat of Chapter N with the first beat of Chapter N+1: if there's a large context gap (say, Chapter N ends at night and Chapter N+1 starts next morning with no mention of the time change), flag it. As a prompt guardrail, you could also instruct the model at chapter generation time: "Make sure to reference unresolved issues from the previous chapter." Essentially, treat the chapter break as a soft boundary, not a hard reset. The model should treat the start-of-chapter prompt as a continuation, not something completely new.
- **Developer Notes:** *Module impact:* Chapter structuring might be partially handled in the current `build_manuscript.py` or similar [7], which likely compiles beats into a full manuscript. We might need to pull some logic out of there or enhance it to handle explicit chapters. Possibly introduce a new function to segment beats into chapters according to either a fixed scheme or model suggestions. The `chapters.schema.json` in the prototype [4] indicates an intended structure for chapters; review it to align our changes. We may need to update how the final draft is assembled: instead of simply concatenating beats, group them by chapter with chapter headings. Code changes could involve the formatter or manuscript builder to insert chapter breaks and titles properly. Test-wise, if there are tests for continuous text, we might add new ones to ensure chapter breaks don't

drop context (for instance, a test that the first sentence of chapter 2 contains a reference to the last events of chapter 1, if applicable).

Drafting Engine

- **Incorporate Full Outline in Draft Prompts:** The drafting engine (which converts outline and beats into prose) must be fed as much structured context as feasible. If currently the final draft is generated by prompting the model with something like "Write the full story based on this outline…", this is prone to omissions. Instead, **generate the draft in smaller sections (chapter by chapter or even scene by scene)**, providing the relevant outline snippet each time. For example, when drafting Chapter 1, include in the prompt: the premise, character list, Act/Chapter goals, and the specific beats for Chapter 1. By doing this iteratively, the model focuses on a manageable chunk and is less likely to forget details. It also allows injecting mid-course corrections between chapters if something was missed. The code might loop over chapters and call the model for each, rather than one-shot the whole manuscript.
- **Fidelity Enforcement – No Beat Left Behind:** Implement measures to ensure every beat and subplot from the outline appears in the final draft. One approach is after drafting, do a cross-check: for each beat description, search the draft text to see if its key event is present. If any are missing, those sections can be regenerated or a prompt can specifically address them ("Add a paragraph about X"). Better yet, during generation, explicitly instruct the model: *"Make sure to include [Beat synopsis] in the narrative."* This could even be formatted as inline comments or an outline in the prompt that the model expands. For example: *"Scene outline: (1) Alice arrives at dance (nervous). (2) Alice discovers Bob's secret. (3) They argue and part ways. Now write this as a continuous narrative."* This way the model is guided stepwise. The presence of an earlier **artifact validation test** [6] can be extended at this stage to verify that the draft covers all outline points (the test could parse the draft and tick off each beat or subplot mention).
- **Summarizer Usage & Caution:** If the project uses a summarizer to compress context for the model (as suggested by the presence of `summarizer.py` in the draft driver [5]), re-evaluate how it's used. Summarization can inadvertently drop specifics (like a minor character's name or a subtle subplot detail). Adjust the summarizer to preserve critical details: for instance, always include character names, essential plot points, and unique terms in the summary to avoid losing them. You could mark these items in the outline (with special tokens) and have the summarizer explicitly keep those tokens verbatim. If context length is a concern (for very large outlines), consider a hybrid approach: feed the model the raw outline for the chapter it's writing (which is smaller) and only use summaries for global context beyond the immediate chapter. The redesign should minimize reliance on heavy summarization by breaking the task into smaller chunks (as noted above) so the model doesn't need a summary of the entire novel at once.
- **Quality & Consistency Checks in Draft:** Add guardrails in the drafting prompts like a final **consistency checklist**. For example, at the end of a chapter draft prompt, you might add: "Ensure the tone matches the style guide (e.g., suspenseful, witty), and the character dialogues reflect their traits. Verify that any open questions from previous chapters are acknowledged." This reminds the model to self-edit for consistency with the plan. Additionally, consider using a second pass where the model acts as a reviewer: after the full draft is done, prompt the model to list any inconsistencies or missing pieces ("Review this draft for any missing beats or character inconsistencies"). This can catch subtle context losses that automated tests might miss.
- **Developer Notes:** *Module impact:* The drafting engine is implemented in the `draft_driver` module (likely `draft_driver.py` and `build_manuscript.py`) [7]. Major changes here include

shifting from one-shot generation to iterative (if not already). This could mean refactoring `build_manuscript.py` to loop through the outline's chapters and call the model for each, rather than building an entire prompt for the whole story. Introduce functions to inject outline context into each chapter's prompt. Also implement the verification logic after draft generation – possibly within `draft_driver.py` or a new module (e.g., `draft_validator.py`). If using GPT for review, that could be an optional step in the pipeline for developers. Be mindful of token limits: ensure that by chunking and providing targeted context, each prompt stays within limits. The code boundary between outline data and final text might become more integrated – for instance, the outline JSON might be kept in memory throughout drafting, rather than just converting to text early. Make sure any such changes don't break the existing interface (the CLI command should still output a manuscript). Update tests or add new ones focusing on the final manuscript's completeness (every outline item accounted for).

Implementation Phases

Implement the above recommendations in phases to manage complexity and ensure stability at each step. Below is a proposed roadmap:

- **Phase 1: Continuity Recovery** – Focus on linking and consistency of existing content without major prompt overhauls.
- *Reorder critical steps:* Adjust the execution order so Character Generation occurs before Arc Planning. This is a low-hanging fruit to immediately ground arcs in character context.
- *Data linking:* Introduce unique IDs and references for characters and subplots across the outline. Update schemas and internal data structures (e.g., include character references in beats).
- *Minimal context injection:* Start passing the premise and character list into later prompts (even if just appended as raw text). Ensure the summarizer keeps these references [5].
- *Validation:* Expand artifact validation tests [6] to catch glaring continuity issues (e.g., a beat referencing a character not in character list, or an outline subplot not in final draft). Address any failing cases by patching the data flow.

- **Phase 2: Prompt Scaffolding & Guardrails** – Redesign prompt templates and add guardrails to reduce context loss during generation.

- *Prompt templates:* Rewrite prompts for each subsystem (characters, arcs, subplots, beats, chapters) to include structured context (using lists, tags, or QA format). Introduce the checklists within prompts (e.g., things to include in each output).
- *Guardrail mechanisms:* Implement checklist injection, prompt tags (like `<CharacterTrait>` tags), and instructive system messages that remind the model of required elements. For instance, ensure every beat prompt has a list of involved characters and expected plot points, reducing omission.
- *Wizard/UI updates:* If a wizard interface exists, update it to reflect new steps (e.g., a step for subplots). Make sure the user input process still feels coherent with the new order (the user might now see characters being generated earlier).
- *Module refactors:* Clean up module boundaries if needed – e.g., have distinct functions or classes for handling each major phase. This makes maintenance easier and each phase's prompt logic clearer. At this stage, code may be reorganized (perhaps new files for subplot or chapter logic) to align with the new conceptual model.

- *Test prompt outputs:* Manually or with small automated checks, verify that the new prompts indeed include all intended context. For example, log a sample prompt for beat generation and check it contains character traits and subplot tags. Iterate on prompt wording based on results (this might require a few experiments with the model to fine-tune effectiveness of guardrails).

- **Phase 3: Draft Fidelity Enforcement** – Ensure the final manuscript fully realizes the outline and is free of drift.

- *Chunked drafting:* Implement the chapter-by-chapter (or scene-by-scene) drafting approach. This likely involves modifying `build_manuscript` to loop over outline sections, as discussed. Verify that this does not break the flow – transitions should be smooth with the recap method.
- *Coverage verification:* Develop a post-draft verification routine. This could be a script that compares outline vs draft, or even leveraging the AI to answer questions like "Was the subplot X resolved in the story?". Start with straightforward checks: every beat description phrase should find a match in the draft text. Integrate this into tests or as a warning output.
- *Fine-tune summarizer or remove if possible:* If splitting drafting into chapters, you may no longer need to summarize the entire outline – just feed the relevant part. Simplify or bypass the summarization step for final drafting. If it remains for any global context, tune it as per Phase 2 so it doesn't drop critical info.
- *Quality enhancements:* Incorporate the review step where the model double-checks consistency. For implementation, this could be a mode in the CLI (e.g., `--review-draft`) that runs the draft through a consistency checker (which might be an LLM call or regex checks for key terms). This stage is about **polish** – e.g., ensuring character voices are consistent (which might involve enforcing style rules from the style guide schema [1] throughout the chapters).
- *User testing:* Have a few complete runs of the pipeline on example projects (possibly using the provided presets or styles) to see if all pieces come together. Use these to further tweak prompts or logic. The outcome of Phase 3 should be a reliably coherent draft generation where if something is wrong, the system is aware (through tests or warnings) and can prompt the developer/user to fix or retry specific parts.

Each phase builds on the previous, so it's recommended to merge Phase 1 changes before moving to Phase 2, and so on. This incremental approach will make it easier to isolate issues introduced by each set of changes.

Reseed Payload for Context Recovery

To handle situations where the conversation context is lost (e.g. lengthy sessions or switching to a new OpenAI model instance mid-project), it's crucial to have a **"reseed" payload** that can restore the state. The minimal reseed payload should include the core artifacts of the project in a concise form:

- **Premise & Settings:** A short paragraph summarizing the premise, genre, and style guidelines. Essentially, remind the model what story it's writing and in what manner.
- **Character Roster:** A bullet list of characters with one-line descriptions each (name and key traits/roles). This serves as a quick reference for the model to recall who's who.
- **Outline Summary:** A high-level outline of the story structure. This can be act-wise or chapter-wise, with one sentence per major beat. Focus on the major plot points and any subplots. For example:

“Act I: Alice introduced, accepts challenge to save village (Subplot: strained friendship with Bob). Act II: Trials and failures…,” etc. This should fit in a couple of paragraphs if possible.

- **Progress Pointer:** If resuming in the middle (say we lost context during drafting chapter 5), include a note of where we are in the narrative and any important recent events. E.g., “Current point: Chapter 5, Alice has just discovered Bob’s betrayal and is about to confront him.” This ensures continuity from the last known event.
- **Instructions/Mode Reminder:** Finally, include a line reminding the assistant of its role (if needed): e.g., “You are assisting in drafting a novel. Continue from the provided outline and prior content.” This helps reestablish the format and expectations.

In practice, this reseed payload can be prepared as a small JSON or markdown file (for instance, `reseed_summary.md`) that the system writes whenever a major phase is completed. If an OpenAI session is interrupted or a new session is started (like using a fresh GPT-4 instance), the developer can copy-paste this payload to quickly bring the AI back up to speed. Essentially, it condenses the project state into a digestible brief for the model. By saving key prompt blocks or the outline JSON at checkpoints, you can regenerate this summary on the fly. The goal is that even without the prior chat history, the model can read this and continue with minimal inconsistencies.

By following this recovery and redesign plan, the Novelist project should resolve the drift in character identity, omitted beats, lost subplots, and other context failures. The story generation pipeline will become more robust, with each phase informing the next through explicit, structured context. Moreover, developers and writers using the system will gain confidence that the final draft remains faithful to the outline they crafted, and any hiccups in AI session continuity can be mitigated with the reseed strategy. Each recommendation above is grounded in the lessons from the context loss audit and the design insights from earlier prototypes, ensuring that we build on past knowledge to create a more resilient storytelling engine.

Sources: The recommendations reference the structure and files from the original Novelist prototypes, which include defined schemas and utility modules for maintaining story artifacts [1] [6]. The presence of modules like `prompt_builders.py` and `summarizer.py` in the codebase guided our strategy to enhance prompt content and manage context length [3] [5]. Additionally, the artifact validation tests in the prototype suggest a framework for enforcing outline-to-draft fidelity which this plan expands upon [6]. All these guided the phased approach to redesigning the Novelist project for improved consistency and reliability.

---

[1] [2] [3] [4] [5] [6] [7] prototype_novelist_urls.txt
file://file-PhooWeucjMGoqweGLS4sZ2