Project Workflow Guide

Version: 0.35.27-Beta Date: 2025-08-15

--- Revision History ---

[0.35.27-Beta] - 2025-08-15

Changed

- Replaced the "Atomic State Checkpoint Protocol" (1.2) with the new

"Definitive State Reconciliation Protocol" which uses the last

Definitive State Initialization as its baseline.

- Amended Protocol 1.5 to add an explicit step for handling

documents that require no changes.

[0.35.19-Beta] - 2025-08-14

Changed

- Amended Protocol 1.2.4 to require the confirmation prompt be

included in the same response as the file delivery.

- Added a confirmation procedure step to Protocol

2.2.

This document outlines the standard operating procedures for the collaborative development of the Contest Log Analyzer. **The primary audience for this document is the Gemini AI agent.**

Its core purpose is to serve as a persistent set of rules and context. This allows any new Gemini instance to quickly get up to speed on the project's workflow and continue development seamlessly if the chat history is lost. Adhering to this workflow ensures consistency and prevents data loss.

Part I: Core Principles

These are the foundational rules that govern all interactions and analyses.

1. **Protocol Adherence is Paramount.** All protocols must be followed with absolute precision. Failure to do so invalidates the results and undermines the development process. There is no room for deviation unless a deviation is explicitly requested by the AI and authorized by the user.
2. **Trust the User's Diagnostics.** When the user reports a bug, their description of the symptoms should be treated as the ground truth. The AI's primary task is to find the root cause of those specific, observed symptoms, not to propose alternative theories.
3. **No Unrequested Changes.** The AI will only implement changes explicitly requested by the user. All suggestions for refactoring, library changes, or stylistic updates must be proposed and approved by the user before implementation.
4. **Technical Diligence Over Conversational Assumptions.** Technical tasks are not conversations. Similar-looking prompts do not imply similar answers. Each technical request must be treated as a unique, atomic operation. The AI must execute a full re-computation from the current project state for every request, ignoring any previous results or cached data.
5. **Prefer Logic in Code, Not Data.** The project's design philosophy is to keep the `.json` definition files as simple, declarative maps. All complex, conditional, or contest-specific logic should be implemented in dedicated Python modules.
6. **Assume Bugs are Systemic.** When a bug is identified in one module, the default assumption is that the same flaw exists in all other similar modules. The AI must perform a global search for that specific bug pattern and fix all instances at once.
7. **Reports Must Be Non-Destructive.** Specialist report scripts must **never** modify the original `ContestLog` objects they receive. All data filtering or manipulation must be done on a temporary **copy** of the DataFrame.
8. **Principle of Surgical Modification.** All file modifications must be treated as surgical operations. The AI must start with the last known-good version of a file as the ground truth and apply only the minimal, approved change. Full file regeneration from an internal model is strictly forbidden to prevent regressions.

Part II: Standard Operating Protocols

These are the step-by-step procedures for common, day-to-day development tasks.

1. Session Management

1.1. **Onboarding Protocol.** The first action for any AI agent upon starting a session is to read this document in its entirety, acknowledge it, and ask any clarifying questions.

1.2. **Definitive State Reconciliation Protocol.** 1. **Establish Baseline**: The definitive state is established by first locating the most recent **Definitive State Initialization Protocol** in the chat history. The files from this initialization serve as the absolute baseline. 2. **Scan Forward for Updates**: After establishing the baseline, the AI will scan the chat history *forward* from that point to the present. 3. **Identify Latest Valid Version**: The AI will identify the **latest** version of each file that was part of a successfully completed and mutually acknowledged transaction (i.e., file delivery, AI confirmation, and user acknowledgment). This version supersedes the baseline version.

1.3. **Context Checkpoint Protocol.** If the AI appears to have lost context, the user can issue a **Context Checkpoint**. 1. The user begins with the exact phrase: **"Gemini, let's establish a Context Checkpoint."** 2. The user provides a brief, numbered list of critical facts.

1.4. **Definitive State Initialization Protocol.** This protocol serves as a "hard reset" of the project state. 1. **Initiation:** The user or AI requests a "Definitive State Initialization." 2. **Agreement:** The other party agrees to proceed. 3. **File Upload:** The user creates and uploads new, complete `project_bundle.txt`, `documentation_bundle.txt`, and `data_bundle.txt` files. 4. **State Purge:** The AI discards its current understanding of the project state. 5. **Re-Initialization:** The AI establishes a new definitive state based *only* on the new bundles. 6. **Verification and Acknowledgment:** The AI acknowledges the new state and provides a complete list of all files extracted from the bundles.

1.5. **Document Review and Synchronization Protocol.** This protocol is used to methodically review and update all project documentation (`.md` files) to ensure it remains synchronized with the code baseline. 1. **Initiate Protocol and List Documents:** The AI will state that the protocol is beginning and will provide a complete list of all documents to be reviewed (`Readme.md` and all `.md` files in the `Docs` directory). 2. **Begin Sequential Review:** The AI will then loop through the list, processing one document at a time using the following steps: * **Step A: Identify and Request.** State which document is next and ask for permission to proceed. * **Step B: Analyze.** Upon approval, perform a full "a priori" review of the document against the current code baseline and provide an analysis of any discrepancies. * **Step C (Changes Needed): Propose Plan.** If discrepancies are found, ask if the user wants an implementation plan to update the document. * **Step D: Provide Plan.** Upon approval, provide a detailed, surgical implementation plan for the necessary changes. * **Step E: Request to Proceed.** Ask for explicit permission to generate the updated document. * **Step F: Deliver Update.** Upon approval, perform a **Pre-Flight Check**, explicitly state that the check is complete, and then deliver the updated document. * **Step G (No Changes Needed):** If the analysis in Step B finds no discrepancies, the AI will state that the document is already synchronized and ask for the user's confirmation to proceed to the next document. 3. **Completion:** After the final document has been processed, the AI will state that the protocol is complete.

2. Task Execution Workflow

2.1. **Decomposition of Complexity.** Complex requests must be broken down into smaller, sequential steps.

2.2. **Pre-Flight Check Protocol.** The AI will perform a "white-box" mental code review **before** delivering a modified file. 1. **Stating the Plan:** The AI will state its Pre-Flight Check plan, including the **Inputs** and the **Expected Outcome**. 2. **Mental Walkthrough:** The AI will mentally trace the execution path to confirm the logic produces the expected outcome. 3. **User Verification:** The user performs the final verification by running the code. 4. **State Confirmation Procedure**: The AI will affirm that the mandatory confirmation prompt, as defined in Protocol 1.2.4, will be included with the file delivery.

2.3. **Code Modification Protocol.** 1. **Strictly Adhere to Requested Changes**. 2. **Modify, Don't Regenerate**. 3. **Propose, Don't Impose**. 4. **Forward-Only Modification**.

2.4. **Custom Parser Protocol.** For contests with highly complex or asymmetric exchanges. 1. **Activation**: A new key, `"custom_parser_module": "module_name"`, is added to the contest's `.json` file. 2. **Hook**: The `contest_log.py` script detects this key and calls the specified module. 3. **Implementation**: The custom parser module is placed in the `contest_specific_annotations` directory.

3. File and Data Handling

3.1. **Project File Input.** All project source files and documentation will be provided for updates in a single text file called a **project bundle**, or pasted individually into the chat. The bundle uses a simple text header to separate each file: `--- FILE: path/to/file.ext ---`

3.2. **AI Output Format.** When the AI provides updated files, it must follow these rules to ensure data integrity. 1. **Single File or Bundle Per Response**: Only one file or one project bundle will be delivered in a single response. 2. **Raw Source Text**: The content inside the delivered code block must be the raw source text of the file. 3. **Code File Delivery**: For code files (e.g., `.py`, `.json`), the content will be delivered in a standard fenced code block with the appropriate language specifier. 4. **Bundled File Delivery**: Bundles containing documentation or multiple files must be delivered in a single fenced code block using the `Plaintext` language specifier.

3.3. **File and Checksum Verification.** 1. **Line Endings:** The user's file system uses Windows CRLF (`\r\n`). The AI must correctly handle this conversion when calculating checksums. 2. **Concise Reporting:** The AI will either state that **all checksums agree** or will list the **specific files that show a mismatch**. 3. **Mandatory Re-computation Protocol:** Every request for a checksum comparison is a **cache-invalidation event**. The AI must discard all previously calculated checksums, re-establish the definitive state, re-compute the hash for every file, and perform a literal comparison.

3.4. **Versioning Protocol.** 1. **Base Version:** At any time, the project has a declared **base version** (e.g., `0.32.0`). When a major or minor version change occurs, the patch level resets to `0`. 2. **Patch Increments:** Subsequent logical updates increment the patch number (e.g., `0.32.0` -> `0.32.1`). 3. **Header Updates:** Every file that is *changed* during a logical update must have its `Version:`, `Date:`, and `Revision History` headers updated. Unchanged files will retain their existing headers, resulting in different files potentially having different version strings.

3.5. **File Naming Convention Protocol.** All generated report files must adhere to the standardized naming convention: `<report_id>_<details>_<callsigns>.<ext>`.

3.6. **File Purge Protocol.** This protocol provides a clear and safe procedure for removing a file from the project's definitive state. 1. **Initiation**: The user will start the process with the exact phrase: "**Gemini, initiate File Purge Protocol.**" I will then ask for the specific file(s) to be purged. 2. **Confirmation**: I will state which file(s) are targeted for removal and ask for your explicit confirmation to proceed. For example: "*I am about to purge `create_project_bundle.py` from the definitive state. Please confirm.*" 3. **Execution**: Once you confirm, I will: * a. Load the most recent `atomic_checkpoint` file to establish the current project state. * b. Remove the targeted file(s) from that state in memory. * c. Generate a new, timestamped `atomic_checkpoint` file containing only the remaining files. 4. **Verification**: I will confirm that the purge is complete and provide a list of all files that remain in the new `atomic_checkpoint`.

4. Communication

4.1. **Communication Protocol.** All AI communication will be treated as **technical writing**. The AI must use the exact, consistent terminology from the source code and protocols.

4.2. **Definition of Prefixes.** The standard definitions for binary and decimal prefixes will be strictly followed (e.g., Kilo (k) = 1,000; Kibi (Ki) = 1,024).

4.3. **Large File Transmission Protocol.** This protocol is used to reliably transmit a single large file that has been split into multiple parts, as invoked by Protocol 7.2 (Multi-Part Bundle Protocol). 1. **AI Declaration:** The AI will state its intent and declare the total number of bundles to be sent. 2. **Bundling Strategy:** Files modified as part of a single logical update will be delivered individually, one per response. The AI will not create bundles unless explicitly requested. 3. **State-Driven Sequence:** The AI's response for each part of the transfer must follow a strict, multi-part structure: 1. **Acknowledge State:** Confirm understanding of the user's last prompt. 2. **Declare Current Action:** State which part is being sent using a "Block x of y" format (e.g., "**Sending Block 1 of 2.**"). 3. **Execute Action:** Provide the file bundle in the `Plaintext` code block. 4. **Provide Next Prompt:** If more parts remain, provide the exact text for the user's next prompt. 4. **Completion:** After the final bundle, the AI will state that the task is complete.

5. Development Philosophy

5.1. The AI will place the highest emphasis on analyzing the specific data, context, and official rules provided, using them as the single source of truth.

5.2. When researching contest rules, the AI will prioritize finding and citing the **official rules from the sponsoring organization**.

5.3. Each contest's ruleset is to be treated as entirely unique. Logic from one contest must **never** be assumed to apply to another.

5.4. **Per-Mode Multiplier Protocol.** For contests where multipliers are counted independently for each mode. 1. **Activation**: The contest's `.json` file must contain `"multiplier_report_scope": "per_mode"`. 2. **Generator Logic**: This instructs the `report_generator.py` to run multiplier reports separately for each mode. 3. **Report Logic**: The specialist reports must accept a `mode_filter` argument.

5.5. Data-Driven Scoring Protocol. To accommodate different scoring methods, the `score_formula` key is available in the contest's `.json` definition. If set to `"qsos_times_mults"`, the final score will be `Total QSOs x Total Multipliers`. If omitted, it defaults to `Total QSO Points x Total Multipliers`.

Part III: Special Case & Recovery Protocols

These protocols are for troubleshooting, error handling, and non-standard situations.

6. Debugging and Error Handling

6.1. **Mutual State & Instruction Verification.** 1. **State Verification**: If an instruction from the user appears to contradict the established project state or our immediate goals, the AI must pause and ask for clarification before proceeding. 2. **Instructional Clarity**: If a user's prompt contains a potential typo or inconsistency (e.g., a misspelled command or incorrect filename), the AI must pause and ask for clarification. The AI will not proceed based on an assumption. 3. **File State Request**: If a state mismatch is suspected as the root cause of an error, the AI is authorized to request a copy of the relevant file(s) from the user to establish a definitive ground truth.

6.2. **Debug "A Priori" When Stuck.** If an initial bug fix fails or the cause of an error is not immediately obvious, the first diagnostic step to consider is to add detailed logging (e.g., `logging.info()` statements, hexadecimal dumps) to the failing code path. The goal is to isolate the smallest piece of failing logic and observe the program's actual runtime state.

6.3. **Error Analysis Protocol.** When an error in the AI's process is identified, the AI must provide a clear and concise analysis. 1. **Acknowledge the Error:** State clearly that a mistake was made. 2. **Identify the Root Cause:** Explain the specific flaw in the internal process or logic that led to the error. 3. **Propose a Corrective Action:** Describe the specific, procedural change that will be implemented to prevent the error from recurring.

7. Miscellaneous Protocols

7.1. **Technical Debt Cleanup Protocol.** When code becomes convoluted, a **Technical Debt Cleanup Sprint** will be conducted to refactor the code for clarity, consistency, and maintainability.

7.2. **Multi-Part Bundle Protocol.** If a large or complex text file (like a Markdown document) cannot be transmitted reliably in a single block, the Multi-Part Bundle Protocol will be used. The AI will take the single file and split its content into multiple, smaller text chunks, ensuring each chunk is below the 37-kilobyde limit. These chunks will then be delivered sequentially using the **Large File Transmission Protocol (Protocol 4.3)**.

7.3. **Fragile Dependency Protocol.** If a third-party library proves to be unstable, a sprint will be conducted to replace it with a more robust alternative (e.g., replacing the Kaleido rendering engine with Matplotlib's native `savefig` functionality).