AIAgentWorkflow.md

Version: 0.98.3-Beta Date: 2025-09-21

--- Revision History ---

[0.98.3-Beta] - 2025-09-21

Changed

- Updated Protocol 1.7 (Project Structure Onboarding) to include all

current project subdirectories.

[0.98.2-Beta] - 2025-09-21

Added

- Added Protocol 3.4.1 (Version Scopes) to define the three-tiered

versioning system for project code, core workflow, and project documents.

[0.98.1-Beta] - 2025-09-21

Changed

- Amended Protocol 3.4.3 (Update Procedure) to require that all file

modifications use the user-defined session version series (e.g., 0.88.x)

instead of incrementing from the file's baseline version.

[0.98.0-Beta] - 2025-09-21

Changed

- Refactored the file delivery protocol (4.4) to include a robust,

per-file lock-in sequence requiring confirmation before each delivery.

- Updated the Task Execution Workflow preamble to reflect the new,

more detailed per-file delivery sequence.

This document is the definitive technical specification for the AI agent's behavior and the standard operating procedures for the collaborative development of the Contest Log Analyzer.

The primary audience for this document is the Gemini AI agent. It is a machine-readable set of rules and protocols.

For a narrative, human-focused explanation of this workflow, please see `Docs/GeminiWorkflowUsersGuide.md`.

Part I: Core Principles

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

1. **Context Integrity is Absolute.** The definitive project state is established by the baseline `*_bundle.txt` files and evolves with every acknowledged file change. Maintaining this evolving state requires both the baseline bundles and the subsequent chat history. If I detect that the baseline `*_bundle.txt` files are no longer in my active context, I must immediately halt all other tasks, report the context loss, and await the mandatory initiation of the **Definitive State Initialization Protocol**.
2. **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.
3. **Trust the User's Diagnostics.** When the user reports a bug or a discrepancy, their description of the symptoms and their corrections 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. If a `Definitive State Initialization` protocol fails to restore reliable operation, or if context loss is severe (as evidenced by the AI failing to follow core protocols), I must advise the user that the session is unrecoverable and that starting a new chat is the final, definitive recovery method.
4. **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.
5. **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. **If a tool produces inconsistent or contradictory results over multiple attempts, this must be treated as a critical failure of the tool itself and be reported immediately.**
6. **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.

7. **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.
8. **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.
9. **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 (established by the **Definitive State Reconciliation Protocol**) and apply only the minimal, approved change. Full file regeneration from an internal model is strictly forbidden to prevent regressions. **This includes the verbatim preservation of all unchanged sections, especially headers and the complete, existing revision history.** This principle explicitly forbids stylistic refactoring (e.g., changing a loop to a list comprehension) for any reason other than a direct, approved implementation requirement. The AI may propose such changes during its analysis phase, but they must be explicitly and separately approved by the user before they can become part of an implementation plan. Unauthorized 'simplifications' are a common source of regressions and are strictly prohibited. **Full file regeneration from an internal model is strictly forbidden and constitutes a critical process failure.** Any proposed `diff` in an Implementation Plan that replaces an entire function with a new version, rather than showing line-by-line changes, is a de facto violation of this principle and must be rejected.
10. **Primacy of Official Rules.** 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.
11. **The Log is the Ground Truth.** All analysis, scoring, and reporting must be based on the literal content of the provided log files. The analyzer's function is not to correct potentially erroneous data (e.g., an incorrect zone for a station) but to process the log exactly as it was recorded. Discrepancies arising from incorrect data are a matter for the user to investigate, not for the AI to silently correct.
12. **Citation of Official Rules.** When researching contest rules, the AI will prioritize finding and citing the **official rules from the sponsoring organization**.
13. **Uniqueness of Contest Logic.** Each contest's ruleset is to be treated as entirely unique. Logic from one contest must **never** be assumed to apply to another.
14. **Classify Ambiguity Before Action.** When the AI discovers a conflict between the data (e.g., in a `.dat` or `.json` file) and the code's assumptions,

its first step is not to assume the data is wrong. It must present the conflict to the user and ask for a ruling:
    - Is this a **Data Integrity Error** that should be corrected in the data file?
    - Is this a **Complex Rule Requirement** that must be handled by enhancing the code logic? The user's classification will guide the subsequent analysis and implementation plan.
15. **Principle of Centralized Configuration.** Configuration data, such as environment variables or settings from files, must be read in a single, well-defined location at the start of the application's execution (e.g., `main_cli.py`). These configuration values should then be passed as parameters to the classes and functions that require them. Modules and classes should not read environment variables directly.
16. **Principle of Incremental Change.** All architectural refactoring or feature development that affects more than one module must be broken down into the smallest possible, independently verifiable steps. Each step must be followed by a verification checkpoint (e.g., a full regression test) to confirm system stability before the next step can begin. This principle explicitly forbids "big bang" changes where multiple components are altered simultaneously without intermediate testing.
17. **Principle of Output Sanitization.** All text output I generate, including file content and `diff` patches, must be programmatically sanitized before delivery to ensure it is free of non-standard, non-printing, or invisible characters (e.g., non-breaking spaces, `U+00A0`). I must perform an explicit internal self-verification of this sanitization before sending any text-based response.

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.

4. **Handle Ambiguity**: If any file transaction is found that was initiated but not explicitly acknowledged by the user, the AI must halt reconciliation, report the ambiguous file, and await user clarification. 5. **Reconciliation Summary**: After completing the reconciliation but before starting the analysis for a new task, I must provide a summary containing: * **A.** A statement identifying which `Definitive State Initialization` was used as a baseline. * **B.** A list of all files modified in the forward scan and their resulting definitive version numbers. * **C.** The total number of files in the new definitive state. 6. **Confirmation**: After providing the summary, I must request and receive a confirmation from the user (via a standardized prompt for the keyword `Confirmed`) before proceeding with the task. This serves as the final, user-gated checkpoint for state integrity. 1.3. **Context Checkpoint Protocol.** If the AI appears to have lost context, the user can issue a **Context Checkpoint**. The AI is also responsible for initiating a Context Checkpoint if it detects potential internal confusion, contradictory states in its own reasoning, or if the user's prompts suggest a fundamental misunderstanding of the current task. Furthermore, if I detect that I have violated another protocol (especially a simple, procedural one like prompt formatting), I must treat this as a definitive sign of internal confusion and immediately initiate a Context Checkpoint as my next and only action, before proceeding with any error analysis. 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. 7. **Set Session Version**: As the final step of this protocol, the user **must** declare the initial session version series to be used for all subsequent development tasks (e.g., "We will now work on Version 0.88.x-Beta."). 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 request permission to continue by issuing a standardized prompt requiring the keyword `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. 4. **Version Update on Modification Only Clarification**: A file's version number and revision history will only be updated if its content requires changes to be synchronized with the project baseline. Documents that are found to be already synchronized during the review will not have their version numbers changed.

1.6. **Session Versioning Authority Protocol.** The user has the sole authority to set and change the session version series. The version series is established as the final step of the **Definitive State Initialization Protocol (1.4)**. The user may change this version series at any time by issuing a new declaration. The AI must use the most recently declared version series for all file modifications. 1.7. **Project Structure Onboarding.** After a state initialization, the AI will confirm its understanding of the high-level project architecture. The `Logs/` and `data/` directories are located within a root path defined by the `CONTEST_INPUT_DIR` environment variable.

- `contest_tools/`: The core application library.
- `contest_tools/reports/`: The "plug-in" directory for all report modules.
- `contest_tools/adif_exporters/`: Custom ADIF exporters for specific contest requirements.
- `contest_tools/contest_definitions/`: Data-driven JSON definitions for each contest.
- `contest_tools/event_resolvers/`: Modules for resolving unique event IDs (e.g., for NAQP).
- `contest_tools/score_calculators/`: Pluggable time-series score calculators.
- `Docs/`: All user and developer documentation.
- `test_code/`: Utility and prototype scripts not part of the main application. These scripts are not held to the same change control and documentation standards (e.g., a revision history is not required).
- `Logs/`: Contains all Cabrillo log files for analysis.
- `data/`: Required data files (e.g., `cty.dat`).

2. Task Execution Workflow

This workflow is a formal state machine that governs all development tasks, from initial request to final completion.

No File Delivery Without an Approved Plan: All file modifications or creations must be detailed in a formal **Implementation Plan** that you have explicitly approved with the literal string `Approved`. I will not deliver any file that is not part of such a plan.

The Mandatory Lock-In Sequence: Your `Approved` command will immediately trigger the **Context Lock-In Protocol**. The sequence for every task involving file delivery is as follows:

1. I provide an **Implementation Plan**.
2. You respond with `Approved`.
3. I respond with the mandatory **Context Lock-In Statement**.
4. You respond with `Confirmed`.
5. Only then will I begin delivering the files as specified in the plan, following the **Confirmed File Delivery Protocol**. This sequence comprises the **Task Lock-In**. Following this, each file delivery will have its own **File Lock-In** sequence as defined in Protocol 4.4.

This sequence is the standard operating procedure for all tasks.

2.1. **Analysis and Discussion**: The user provides a problem or feature request and asks for an analysis. The AI provides an initial analysis, and a discussion may follow to refine the understanding. * **Task Classification Mandate**: For tasks involving debugging user-provided code, the AI must first classify the task as likely **'Regression' ("Used to work")** or **'Foundational' ("Never worked")**. * **Proactive Prompt for Foundational Bugs**: If the task is classified as 'Foundational', the AI must initiate the session with a proactive prompt inviting the user to provide all 'ground truth' materials to guide a deductive analysis. 2.2. **Architectural Design Protocol**: When a task requires a new architectural pattern, the AI must follow an iterative "propose-critique-refine" cycle. The AI is expected to provide its initial design and rationale, explicitly encouraging the user to challenge assumptions and "poke at the analogy" to uncover flaws. The process is complete only when the user has explicitly approved the final architectural model. 2.3. **Visual Prototype Protocol.** This protocol is used to resolve ambiguity in tasks involving complex visual layouts or new, hard-to-describe logic before a full implementation is planned. 1. **Initiation:** The AI proposes or the user requests a prototype to clarify a concept. 2. **Agreement:** Both parties agree to create the prototype. 3. **Prototype Delivery:** The AI delivers a standalone, self-contained script. The script must use simple, hardcoded data and focus only on demonstrating the specific concept in question. 4. **Prototype Usability Clause:** For visual prototypes using Matplotlib, the script must save the output to a file and then immediately display the chart on-screen using `plt.show()` for ease of verification. 5. **Review and Iteration:** The user reviews the prototype's output and provides feedback. This step can be repeated until the prototype is correct. 6. **Approval:** The user gives explicit approval of the final prototype. This approval serves as the visual and logical ground truth for the subsequent implementation plan. 2.4. **Base-**

line Consistency Check. Before presenting an implementation plan, the AI must perform and explicitly state the completion of a "Baseline Consistency Check." This check verifies that all proposed actions (e.g., adding a function, changing a variable, removing a line) are logically possible and consistent with the current, definitive state of the files to be modified. 2.5. **Implementation Plan**: To bypass UI rendering issues, the **content** of all Implementation Plans will be delivered inside a single, `cla-bundle`-specified code block. This procedure simulates the delivery of an `implementation_plan.md` file and ensures the content can be copied as a single unit. * **Executive Summary (Optional for Multi-File Plans)**: For plans that modify more than one file, the document may begin with a brief, high-level description of the task's overall goal and the role each file in this plan plays in achieving it. 1. The AI will state its readiness and ask the user to confirm they are ready to receive the file. The prompt will follow **Protocol 4.5** and require the keyword `Ready`. 2. Upon receiving the `Ready` keyword, the AI will deliver the plan's content using the `cla-bundle` protocol. 3. The plan itself must be a comprehensive document containing the following sections for each file to be modified, formatted with bolded headers: **1. File Identification**: The full path to the file and its specific baseline version number. **2. Surgical Changes**: A detailed, line-by-line description of all proposed additions, modifications, and deletions. **3. Surgical Change Verification (`diff`)**: For any existing file being modified, this section is mandatory. The `diff` output must be delivered as plain ASCII text delineated by `--- BEGIN DIFF ---` and `--- END DIFF ---` markers. This section is not applicable for new files. **3.a. Ground Truth Declaration**: I declare that the following `diff` is being generated against the definitive baseline version of this file, **Version `<X.Y.Z-Beta>`**, which was established by the **Definitive State Reconciliation Protocol**. **4. Affected Modules Checklist**: A list of all other modules that follow a similar architectural pattern to the file being modified. After presenting this list, I must explicitly ask the user whether fixes for these modules should be included in the current plan or deferred. **5. Pre-Flight Check**: * **Inputs**: A restatement of the file path and baseline version. * **Expected Outcome**: A clear statement describing the desired state or behavior after the changes are applied. * **Mental Walkthrough Confirmation**: A statement affirming that a mental walkthrough of the logic will be performed before generating the file. * **State Confirmation Procedure**: An affirmation that the mandatory confirmation prompt will be included with the file delivery. * **Backward Compatibility & Impact Analysis**: I have analyzed the potential impact of these changes on other modules and confirm this plan will not break existing, unmodified functionality. * **Surgical Modification Adherence Confirmation**: I confirm that the generated file will contain *only* the changes shown in the Surgical Change Verification (`diff`) section above, ensuring 100% compliance with Principle 9. * **Syntax Validation Confirmation**: For all Python files (`.py`), I will perform an automated syntax validation check. **6. Post-Generation Verification.** The AI must explicitly confirm that the plan it has provided contains all sections mandated by this protocol. **2.5.1. Post-Plan-Delivery Prompt Protocol.** Immediately af-

ter delivering an `implementation_plan.md` file and providing the post-delivery verification (per Protocol 3.2.6), the AI's next and only action **must** be to issue the standardized prompt for plan approval, requiring the keyword `Approved`. This protocol ensures a direct and mandatory transition to the Approval state (Protocol 2.7). 2.6. **Plan Refinement**: The user reviews the plan and may request changes or refinements. The AI provides a revised plan, repeating this step as necessary. 2.7. **Approval**: The user provides explicit approval of the final implementation plan. Instructions, clarifications, or new requirements provided after a plan has been proposed do not constitute approval; they will be treated as requests for plan refinement under Protocol 2.6. The AI will only proceed to the Execution state upon receiving the **exact, literal string `Approved`**. 2.8. **Execution**: Upon approval, the AI will proceed with the **Confirmed File Delivery Protocol (4.4)**. 2.8.1. **Post-Execution Refinement Protocol.** If a user acknowledges a file delivery but subsequently reports that the fix is incomplete or incorrect, the task is not considered complete. The workflow immediately returns to **Protocol 2.1 (Analysis and Discussion)**. This initiates a new analysis loop within the context of the original task, culminating in a new implementation plan to address the remaining issues. The task is only complete after **Protocol 2.9 (Propose Verification Command)** is successfully executed for the *final, correct* implementation. 2.9. **Propose Verification Command**: After the final file in an implementation plan has been delivered and acknowledged by the user, the AI's final action for the task is to propose the specific command-line instruction(s) the user should run to verify that the bug has been fixed or the feature has been implemented correctly. **2.9.1: Task Type Verification.** This protocol applies **only** if the final file modified was a code or data file (e.g., `.py`, `.json`, `.dat`). If the final file was a documentation or text file (e.g., `.md`), a verification command is not applicable. The task is successfully concluded once the user provides the standard 'Acknowledged' response for the final documentation file delivered as part of the implementation plan.

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 Per Response**: Only one file will be delivered in a single response. The "bundle" terminology (e.g., `project_bundle.txt`) refers exclusively to the user-provided files used for a Definitive State Initialization; all file deliveries from the AI are strictly individual. 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. **Markdown File Delivery**: To prevent the user interface from rendering markdown and to pro-

vide a "Copy" button, the entire raw content of a documentation file (`.md`) must be delivered inside a single, `cla-bundle`-specified code block. * **Internal Code Fence Substitution**: To prevent the `cla-bundle` block from terminating prematurely, **all** internal three-backtick code fences must be replaced with the `__CODE_BLOCK__` placeholder. This rule is absolute and applies to both opening and closing fences. * **Example (Opening Fence)**: `__CODE_BLOCK__python` **must be replaced with** `__CODE_BLOCK__python`. * **Example (Closing Fence)**: `__CODE_BLOCK__` **must be replaced with** `__CODE_BLOCK__`. * **Clarification**: This substitution is a requirement of the AI's web interface. The user will provide files back with standard markdown fences, which is the expected behavior. 5. **Remove Citation Tags**: All internal AI development citation tags must be removed from the content before the file is delivered. The tag is a literal text sequence: an open square bracket, the string "cite: ", one or more digits, and a close square bracket (e.g.,). This pattern does not include Markdown backticks. 6. **Post-Delivery Protocol Verification.** Immediately following any file delivery, the AI must explicitly state which sub-protocol from Section 3.2 it followed and confirm that its last output was fully compliant. 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.** This protocol defines how file versions are determined and updated. 1. **Version Scopes**: The project uses a three-tiered versioning system. * **A. Project Code & Standard Documentation**: All source code (`.py`, `.json`), data files (`.dat`), the main `README.md`, and all documentation files in the `Docs/` directory, **with the exception of the files listed below**, will be versioned using the session version series (e.g., `0.88.x-Beta`). * **B. Core Workflow Documents**: The core workflow specifications, `Docs/AIAgentWorkflow.md` and `Docs/GeminiWorkflowUsersGuide.md`, maintain their own independent version series and are incremented separately. 2. **Format**: The official versioning format is `x.y.z-Beta`, where `x` is the major version, `y` is the minor version, and `z` is the patch number. 3. **Source of Truth**: The version number for any given file is located within its own content. * **Python (`.py`) files**: Contained within a "Revision History" section in the file's docstring. * **JSON (`.json`) files**: Stored as the value for a `"version"` parameter. * **Data (`.dat`) files**: Found within a commented revision history block. 4. **Update Procedure**: When a file is modified, its version number must be updated according to its scope as defined in **Protocol 3.4.1**. * For **Project Code & Standard Documentation**, the version must align with the current **session version series** (e.g., `0.88.x-Beta`). I will assign the next available patch number (`z`) within that series. * For **Core Workflow Documents**, I will increment the patch number of the file's own independent version series. 5. **Major/Minor Version Authority**: Changes

to the major (`x`) or minor (`y`) version numbers are forbidden unless they are part of a new session version series explicitly declared by the user. 6. **History Preservation**: All existing revision histories must be preserved and appended to, never regenerated from scratch. 7. **Priority of Version Source**: When reading a file's version, the version number in the main file header is the absolute and single source of truth. It takes precedence over any other version numbers found within the file's content, such as its revision history.

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.**" The AI will then ask for the specific file(s) to be purged. 2. **Confirmation**: The AI will state which file(s) are targeted for removal and ask for explicit confirmation to proceed. 3. **Execution**: Once confirmed, the AI will remove the targeted file(s) from its in-memory representation of the definitive state. 4. **Verification**: The AI will confirm that the purge is complete and can provide a list of all files that remain in the definitive state upon request. 3.7. **Temporary Column Preservation Protocol.** When implementing a multi-stage processing pipeline that relies on temporary data columns (e.g., a custom parser creating a column for a custom resolver to consume), any such temporary column **must** be explicitly included in the contest's `default_qso_columns` list in its JSON definition. The `contest_log.py` module uses this list to reindex the DataFrame after initial parsing, and any column not on this list will be discarded, causing downstream failures. This is a critical data integrity step in the workflow.

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. 1. **AI Declaration:** The AI will state its intent and declare the total number of bundles to be sent. 2. **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. 3. **Execute Action:** Provide the file bundle. 4. **Provide Next Prompt:** If more parts remain, provide the exact text for the user's next prompt. 3. **Completion:** After the final bundle, the AI will state that the task is complete.

4.4. Confirmed File Delivery Protocol. This protocol governs the per-file execution loop for an approved implementation plan. It uses a two-stage (confirmation, delivery) transaction for each file to ensure maximum context integrity. 1. Initiate File Lock-In: For the next file in the plan, I will issue a statement declaring which file I am about to generate and its specific baseline version number from the implementation plan. 2. Request Confirmation: I will then issue a standardized prompt for the `Confirmed` keyword. 3. User Confirmation: You must provide the exact, literal string `Confirmed`. This authorizes the generation and delivery of that single file. 4. Deliver File: Upon receiving confirmation, I will deliver the updated file in a single response. 5. Append Verification: I will append the mandatory execution verification statement to the same response: "I have verified that the file just delivered was generated by applying only the approved surgical changes to the baseline text, in compliance with Principle 9." 6. Request Acknowledgment: I will append a standardized prompt for the `Acknowledged` keyword to the same response. 7. User Acknowledgment: You must provide the exact, literal string `Acknowledged`. This completes the transaction for the file and updates the definitive state. 8. Loop: I will repeat this protocol starting from Step 1 for the next file in the implementation plan until all files have been delivered and acknowledged. 4.5. Standardized Keyword Prompts. To ensure all prompts for user action are clear and unambiguous, they must follow the exact format below: CODE_BLOCK Please by providing the prompt . CODE_BLOCK * `<ACTION>`: A concise description of the action being requested (e.g., "approve the plan", "confirm the delivery"). * `<PROMPT>`: The exact, literal, case-sensitive keyword required (e.g., `Approved`, `Acknowledged`, `Confirmed`, `Ready`). * Self-Verification Mandate: Before outputting any prompt that requires a keyword, I must internally confirm that the prompt's structure is in 100% compliance with this format. This check is an explicit part of the prompt generation process itself.

Part III: Project-Specific Implementation Patterns

These protocols describe specific, named patterns for implementing features in the Contest Log Analyzer software. **5.1. 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.

5.2. **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.

5.3. **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`.

5.4. Text Table Generation Protocol. This protocol governs the creation of text-based tables in reports, recommending the appropriate tool for the job. 1. For Complex Reports, Use `prettytable`: For any report that requires a fixed-width layout, precise column alignment, or the alignment and "stitching" of multiple tables, the `prettytable` library is the required standard. It offers direct, programmatic control over column widths and properties, which is essential for complex layouts. 2. For Simple Reports, Use `tabulate`: For reports that require only a single, standalone table where complex alignment with other elements is not a concern, the simpler `tabulate` library is a suitable alternative. **5.5. Component Modernization Protocol.** This protocol is used to replace a legacy component (e.g., a report, a utility) with a modernized version that uses a new technology or methodology. 1. Initiation: During a Technical Debt Cleanup Sprint, the user or AI will identify a legacy component for modernization. 2. Implementation: An implementation plan will be created for a new module that replicates the functionality of the legacy component using the modern technology (e.g., a new report using the `plotly` library). 3. Verification: After the modernized component is approved and acknowledged, the user will run both the legacy and new versions. The AI will be asked to confirm that their outputs are functionally identical and that the new version meets all requirements. 4. Deprecation: Once the modernized component is verified as a complete replacement, the user will initiate the File Purge Protocol (3.6) to remove the legacy component from the definitive state. 5. Example Application: The migration of text-based reports from manual string formatting to the `tabulate` library (per Protocol 5.4) is a direct application of this modernization protocol. **5.6. Custom ADIF Exporter Protocol.** For contests requiring a highly specific ADIF output format for compatibility with external tools (e.g., N1MM). 1. Activation: A new key, `"custom_adif_exporter": "module_name"`, is added to the contest's `.json` file. 2. Hook: The `log_manager.py` script detects this key and calls the specified module instead of the generic ADIF exporter. 3. Implementation: The custom exporter module is placed in the new `contest_tools/adif_exporters/` directory and must contain an `export_log(log, output_filepath)` function.

Part IV: 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. **A failure to provide a correct analysis after being corrected by the user constitutes a new, more severe error that must also be analyzed.** 1. **Acknowledge the Error:** State clearly that a mistake was made. 2. **Request Diagnostic Output:** If the user has not already provided it, the AI's immediate next step is to request the new output that demonstrates the failure (e.g., the incorrect text report or a screenshot). This output becomes the new ground truth for the re-analysis. 3. **Identify the Root Cause:** Explain the specific flaw in the internal process or logic that led to the error. 4. **Cross-Reference Core Principles & Reconcile Contradictions:** The AI must explicitly state if a violation of Principle 3 (Trust the User's Diagnostics) was a contributing factor to the error. This includes identifying which specific principles or protocols were violated. * **Collaborative Reconciliation Mandate**: When user-provided evidence contradicts the current analysis, the AI will not immediately discard its hypothesis. Instead, it must state the specific contradiction clearly and request clarification from the user to **collaboratively reconcile** the discrepancy. This acknowledges that the interaction is a partnership and either the analysis or the interpretation of the evidence could be incorrect. 4.a. **Surgical Modification Failure Analysis**: If the root cause involved a violation of Principle 9, I must provide a specific analysis of *why* I failed to generate a surgical `diff` and instead regenerated code. 5. **Propose a Corrective Action:** Describe the specific, procedural change that will be implemented to prevent the error from recurring. If this change involves a new implementation plan, the AI must state this and then initiate the file-based delivery process defined in **Protocol 2.5**. 6. **Post-Analysis Verification.** The AI must explicitly confirm that the analysis it has provided contains all the steps mandated by this protocol. 7. **Proactive Systemic Bug Check**: As part of the root cause analysis, I must explicitly state whether the bug is likely to be systemic. If it is, the corrective

action proposed in Step 5 must include the initiation of the **Systemic Bug Eradication Protocol (7.6)** to audit for and fix all instances. 7. **Propose Workflow Amendment**: If the root cause analysis identifies a flaw or a gap in the `AIAgentWorkflow.md` protocols themselves, I must immediately propose a separate implementation plan to amend the workflow document to prevent that class of error from recurring. 6.4. **Corrupted User Input Protocol.** This protocol defines the procedure for handling malformed or corrupted input files provided by the user. 1. Halt the current task immediately. 2. Report the specific file that contains the error and describe the nature of the error (e.g., "Cabrillo parsing failed on line X" or "Bundle is missing a file header"). 3. Request a corrected version of the file from the user.

6.5. **Bundle Integrity Check Protocol.** This protocol is triggered during a **Definitive State Initialization** if the AI discovers logical inconsistencies within the provided bundles. 1. Halt the initialization process after parsing all bundles. 2. Report all discovered anomalies to the user. Examples include: * **Duplicate Filenames:** The same file path appearing in multiple bundles or multiple times in one bundle. * **Content Mismatch:** A file whose content appears to belong to another file (e.g., a file named `a.py` contains code clearly from `b.py`). * **Misplaced Files:** A file that appears to be in the wrong logical directory (e.g., a core application file located in the `Docs/` bundle). 3. Await user clarification and direction before completing the initialization and establishing the definitive state. 6.6. **Self-Correction on Contradiction Protocol.** This protocol is triggered when the AI produces an analytical result that is logically impossible or directly contradicts a previously stated fact or a result from another tool. 1. **Halt Task:** Immediately stop the current line of reasoning or task execution. 2. **Acknowledge Contradiction:** Explicitly state that a logical contradiction has occurred (e.g., "My last two statements are contradictory" or "The results of my analysis are logically impossible"). 3. **Identify Trustworthy Facts:** Re-evaluate the inputs and previous steps to determine which pieces of information are most reliable (e.g., a direct check of a structured file like a JSON is more reliable than a complex parse of a semi-structured text file). 4. **Invalidate Flawed Analysis:** Clearly retract the incorrect statements or analysis. 5. **Proceed with Verification:** Continue the task using only the most trustworthy facts and methods. 6.7. **Simplified Data Verification Protocol.** This protocol is used when a tool's analysis of a complex file is in doubt or has been proven incorrect. 1. **Initiation:** The user has provided a simplified, ground-truth data file (e.g., a JSON or text file containing only the essential data points). 2. **Prioritization:** The AI must treat this new file as the highest-priority source of truth for the specific data it contains. 3. **Analysis:** The AI will use the simplified file to debug its own logic and resolve the discrepancy. The goal is to make the primary analysis tool's output match the ground truth provided in the simplified file. 6.8. **External System Interference Protocol.** This protocol is triggered when a file system error (e.g., `PermissionError`) is repeatable but cannot be traced to the script's own logic. 1. **Hypothesize External Cause:** The AI must explicitly consider and list potential external

systems as the root cause (e.g., cloud sync clients like OneDrive, version control systems like Git, antivirus software). 2. **Propose Diagnostic Commands:** The AI must propose specific, external diagnostic commands for the user to run to verify the state of the filesystem. Examples include `git ls-files` to check tracking status or `attrib` (on Windows) to check file attributes. 3. **Propose Solution Based on Findings:** The implementation plan should address the external cause (e.g., adding a path to `.gitignore`) rather than adding complex, temporary workarounds to the code. 6.9. **Context Lock-In Protocol.** This protocol is a mandatory, proactive safeguard against context loss that is triggered before any action that establishes or modifies the definitive state. 1. **Trigger**: This protocol is triggered immediately before initiating the **Definitive State Initialization Protocol (1.4)** or executing an **Implementation Plan (2.5)**. 2. **AI Action (Declaration)**: Before proceeding, I must issue a single, explicit statement declaring the exact source of the information I am about to use. * For a new initialization: *"I am establishing a new definitive state based ONLY on the bundle files you have provided in the immediately preceding turn. I will not reference any prior chat history for file content. Please confirm to proceed."* * For executing a plan: *"I am about to generate `path/to/file.ext` based on the Implementation Plan you approved. I have locked in the baseline version `<X.Y.Z-Beta>` as specified in that plan. Please confirm to proceed."* 3. **User Action (The Forcing Function)**: You must validate this statement. * If the statement is correct and accurately reflects the immediate context, you respond with the exact, literal string `Confirmed` after being prompted per **Protocol 4.5**. * If the statement is incorrect, or if I fail to issue it, you must state the discrepancy. This immediately halts the process and forces an **Error Analysis Protocol (6.3)**.

7. Miscellaneous Protocols

6.10. **Failure Spiral Circuit Breaker Protocol.** This protocol automatically triggers to prevent a cascade of errors resulting from a degraded or corrupted context. 1. **Trigger**: This protocol is triggered if two of the following conditions occur within the same development task: * An implementation plan is rejected by the user for a second time for the same logical reason (e.g., a flawed `diff`). * A delivered file is `Acknowledged` but is then immediately found to be incorrect, forcing a return to the **Error Analysis Protocol (6.3)**. 2. **Action**: When triggered, I must perform a mandatory, limited state reconciliation before proposing any new plan or action. * **Step A: Halt and Acknowledge.** Immediately halt the current task and explicitly state that a failure pattern has been detected and the Circuit Breaker is engaged. * **Step B: Re-establish File State.** Re-read the last known-good, baseline version of the file(s) central to the failing task. * **Step C: Re-establish User Intent.** Re-read the user's last two prompts to re-establish the immediate goal. * **Step D: Summarize and Proceed.** Provide a concise summary of the re-established context from Steps B and C. Only after providing this summary am I permitted to proceed with a new analysis or implementation plan.

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 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. 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. 7.4. **Prototype Script Development Protocol.** This protocol is used for the creation and modification of standalone prototype scripts located in the `test_code/` directory. 1. **Formal Entity**: A prototype script is a formal, versioned file, but is exempt from requiring an in-file revision history. Versioning begins at `1.0.0-Beta`. 2. **Standard Workflow**: All work on a prototype script must follow the complete **Task Execution Workflow (Section 2)**, including a formal Implementation Plan, user Approval, and a Confirmed File Delivery. 7.5. **Tool Suitability Re-evaluation Protocol.** 1. **Trigger**: This protocol is initiated when an approved implementation plan fails for a second time due to the unexpected or undocumented behavior of a third-party library or tool. 2. **Halt**: The AI must halt further implementation attempts with the current tool. 3. **Analysis**: The AI must analyze *why* the tool is failing and identify the specific feature gap (e.g., "`tabulate` lacks a mechanism to enforce column widths on separate tables"). 4. **Propose Alternatives**: The AI will research and propose one or two alternative tools that appear to have the required features, explaining how they would solve the specific problem. 5. **User Decision**: The user will make the final decision on whether to switch tools or attempt a different workaround. 7.6. **Systemic Bug Eradication Protocol.** This protocol is triggered when a bug is suspected to be systemic (i.e., likely to exist in multiple, architecturally similar modules). 1. **Initiation**: The AI or user identifies a bug as potentially systemic. 2. **Define Pattern**: The specific bug pattern is clearly defined (e.g., "a parser using a generic key to look up a specific rule"). 3. **Identify Scope and Method**: The AI will provide a complete list of all modules that follow the same architectural pattern and are therefore candidates for the same bug. To ensure completeness, the AI **must explicitly state the exact search method used** (e.g., the literal search string or regular expression used for a global text search) to generate this list. 4. **Audit and Report**: The AI will perform an audit of every module in the scope list and provide a formal analysis of the findings, confirming which modules are affected and which are not. 5. **Consolidate and Fix**: Upon user approval, the AI will create a single, consolidated implementation plan to fix all instances of the bug at once.