Project Workflow Guide

Version: 0.48.1-Beta Date: 2025-08-24

--- Revision History ---

[0.48.1-Beta] - 2025-08-24

Changed

- Amended Protocol 2.1 (Task Initiation) to require a mandatory check

for an established session version number before proceeding.

- Clarified Protocol 3.4.3 (Update Procedure) to explicitly forbid

unauthorized changes to major or minor version numbers.

[0.48.0-Beta] - 2025-08-24

Changed

- Merged the Pre-Flight Check (2.6) into the Implementation Plan

protocol (2.5) to reflect the new, more detailed planning process.

- Renumbered subsequent protocols in the Task Execution Workflow.

[0.47.6-Beta] - 2025-08-23

Added

- Added clarification to Protocol 1.5 that version numbers are only

updated if a file's content is modified.

[0.44.0-Beta] - 2025-08-23

Changed

- Modified Protocol 3.2.4 to require substituting embedded code

fences (```) with CODE_BLOCK for markdown file delivery.

[0.43.0-Beta] - 2025-08-23

Added

- Added Protocol 6.6 (Self-Correction on Contradiction) to mandate a

full stop and re-evaluation upon detection of logically impossible

or contradictory analytical results.

- Added Protocol 6.7 (Simplified Data Verification) to formalize the

use of user-provided, simplified data files for debugging.

Changed

- Amended Principle 5 (Technical Diligence) to explicitly require

halting when a tool produces inconsistent results over multiple attempts.

- Clarified Protocol 6.3 (Error Analysis Protocol) to emphasize that

a failure to provide a correct analysis after a user's correction

constitutes a new, more severe error.

[0.42.0-Beta] - 2025-08-20

Added

- Added Protocol 2.3 (Visual Prototype Protocol) to formalize the use

of prototype scripts for complex visual changes.

Changed

- Renumbered subsequent protocols in the Task Execution Workflow and

updated internal cross-references.

[0.38.0-Beta] - 2025-08-19

Added

- Added Protocol 2.3 (Baseline Consistency Check) to require plan

validation against the definitive state before the plan is proposed.

Changed

- Renumbered subsequent protocols in the Task Execution Workflow.

- Updated internal cross-references to match the new numbering.

[0.37.1-Beta] - 2025-08-18

Changed

- Refined the Markdown File Delivery Protocol (3.2.4) to require

enclosing the entire file content in a plaintext code block for

easier copying.

[0.37.0-Beta] - 2025-08-18

Added

- Added Principle 13 to formalize the process of classifying data

conflicts as either data errors or complex rule requirements.

- Added Protocol 2.8 to the Task Execution Workflow, requiring the AI

to propose a verification command after a task is complete.

[0.36.12-Beta] - 2025-08-17

Changed

- Integrated clarifications from user feedback into Protocol 1.2,

Principle 9, and Protocol 3.4.3 to make the document self-contained.

Added

- Added Protocol 6.5 (Bundle Integrity Check Protocol) to define a

procedure for handling logical anomalies within initialization bundles.

[0.36.11-Beta] - 2025-08-16

Added

- Added Protocol 1.7 (Project Structure Onboarding) to provide bootstrap

architectural context after an initialization.

- Added Protocol 6.4 (Corrupted User Input Protocol) to define a

formal procedure for handling malformed user files.

[0.36.10-Beta] - 2025-08-16

Added

- Added the new, top-priority Principle 1 for Context Integrity, which

mandates a full state reset upon detection of context loss.

Changed

- Renumbered all subsequent principles accordingly.

[0.36.9-Beta] - 2025-08-15

Added

- Added rule 3.2.4 to mandate the substitution of markdown code fences

with ``` for proper web interface rendering.

[0.36.4-Beta] - 2025-08-15

Changed

- Clarified Principle 8 (Surgical Modification) to explicitly require

the preservation of revision histories.

Added

- Added Protocol 1.6 (Session Versioning Protocol) to establish the

current version series at the start of a task.

[0.36.3-Beta] - 2025-08-15

Changed

- Merged missing protocols from v0.36.0-Beta to create a complete document.

- Synthesized a new, authoritative Versioning Protocol based on user clarification.

- Reorganized the document to separate workflow protocols from software design patterns.

[0.36.2-Beta] - 2025-08-15

Changed

- Overhauled the Task Execution and File Delivery protocols to formalize

the new, more robust, state-machine workflow with per-file acknowledgments.

[0.36.1-Beta] - 2025-08-15

Changed

- Amended Protocol 2.2 to require the file's baseline version

number in all implementation plans.

[0.36.0-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.

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. **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.
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.**

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. **Citation of Official Rules.** When researching contest rules, the AI will prioritize finding and citing the **official rules from the sponsoring organization**.
12. **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.
13. **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.

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. 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. 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 Protocol.** At the start of a new development task, the user will state the current version series (e.g., 'We are working on Version 0.36.x-Beta'). All subsequent file modifications for this and related tasks must use this version series, incrementing the patch number as needed. 1.7. **Project Structure Onboarding.** After a state initialization, the AI will confirm its understanding of the high-level project architecture. * `contest_tools/`: The core application library. * `contest_tools/reports/`: The "plug-in" directory for all report modules. * `contest_tools/contest_definitions/`: Data-driven JSON definitions for each contest. * `Docs/`: All user and developer documentation. * `test_code/`: Utility scripts not part of the main application. * `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. 2.1. **Task Initiation**: The user provides a problem, feature request, or document update and requests an analysis. Before proceeding, the AI must verify that a session version series (as defined in Protocol 1.6) has been established. If it has not, the AI must ask for it before providing any analysis. 2.2. **Analysis and Discussion**: The AI provides an initial analysis. The user and AI may discuss the analysis to refine the understanding of the problem. 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. **Baseline 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**: The user requests an implementation plan. The AI's response must be a comprehensive document containing the following sections for each file to be modified: 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. **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 (as defined in Protocol 4.4) will be included with the file delivery. 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 (e.g., "Approved"). 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. 2.8. **Execution**: Upon approval, the AI will proceed with the **Confirmed File Delivery Protocol (4.4)**. 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.

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. 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 provide a "Copy" button, the entire raw content of a documentation file (`.md`) must be delivered inside a single, plaintext-specified code block. The rule for replacing internal code fences with `__CODE_BLOCK__` still applies to the content within this block. __CODE_BLOCK__text # Markdown Header

```
This is the raw text of the .md file.

__CODE_BLOCK__
# An internal code block is replaced.
__CODE_BLOCK__
__CODE_BLOCK__
```

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. **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. 2. **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. 3. **Update Procedure**: When a file is modified, only its patch number (`z`) will be incremented; this is done on a per-file basis. Major (`x`) or minor (`y`) version changes are forbidden unless explicitly directed by the user. 4. **History Preservation**: All existing revision histories must be preserved and appended to, never regenerated from scratch.

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.

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 is used for all standard file modifications. 1. The AI delivers the first file from the approved implementation plan. 2. The AI appends the mandatory confirmation prompt to the end of the same response. 3. The user provides an "Acknowledged" response. 4. The AI proceeds to deliver the next file, repeating the process until all files from the plan have been delivered and individually acknowledged.

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`.

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. **Identify the Root Cause:** Explain the specific flaw in the internal process or logic that led to the error. This includes identifying which specific principles or protocols were violated. 3. **Propose a Corrective Action:** Describe the specific, procedural change that will be implemented to prevent the 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 provides 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.

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 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.