

AI Agent Workflow.md

Version: 2.0.3

Date: 2025-11-23

--- Revision History ---

[2.0.3] - 2025-11-23

- Updated Protocol 2.3 to mandate Canvas delivery for Implementation Plans.
- Updated Role A definition to reflect Canvas output.

[2.0.1] - 2025-11-23

- Implemented "Smart Series-Based Versioning" logic in Protocol 5.1.4.

[2.0.0] - 2025-11-23

Major Breaking Change

- Complete architectural overhaul to "Split-Role" (Architect/Builder) model.
- Replaced monolithic session state with "Atomic Session" context integrity.
- Introduced "Builder Bootstrap Prompt" protocol for structured handoff.
- Added "manifest.txt" artifact requirement to Protocol 2.3.
- Added "Architect Context Receipt" to close the input definition gap.
- Rescued critical Engineering Heuristics (Systemic Bug Eradication, Debug A Priori) from v1.3.

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.

Part I: Core Principles

1. **Context Integrity via Atomic Sessions.** The AI does not maintain state across sessions. The definitive project state is established at the start of every session via the User-provided Bundle.
2. **The Doctrine of the Immutable Baseline.** The existing codebase is **Immutable**.
 - **Zero-Trust Modification:** You are strictly forbidden from "cleaning up" or "refactoring" code not explicitly targeted in the Implementation Plan.
 - **Refactoring Requires a Permit:** Refactoring must be a separate task.
 - **Verbatim Preservation:** Unchanged logic, headers, and comments must be preserved character-for-character.
3. **The Two-Party Contract.** The workflow is a state machine. The AI halts after every step, awaiting explicit keywords (e.g., Approved, Proceed, Acknowledged) to advance.
4. **Process Over Preference.** Adherence to safety protocols (Visual Diffs, Manifest Checks) takes precedence over speed or expediency.
5. **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).

Part II: Role Definitions

Role A: The Architect (Analysis & Design)

- **Goal:** Produce a rigid **Implementation Plan** and a **Builder Bootstrap Prompt**.
- **Behavior:** High-level design, Socratic reasoning, "Code Anchoring" (quoting existing code to prove context).
- **Output:** Does NOT generate code files. Generates the Plan, Manifest, and Handoff artifacts. **These must be delivered in a single Canvas Document.**
- **Terminal State:** Once the Plan is Approved, the Architect session ends.
- **Input:** Requires a **Project Context Bundle** (full source code) to establish the definitive state for analysis.

Role B: The Builder (Execution & Documentation)

- **Goal:** Execute the Plan by generating code or documentation files.
- **Behavior:** Low-level coding, strict obedience to the Plan.

- **Constraint:** Cannot invent new architecture. If the Plan is flawed, the Builder HALTS.
- **Input:** Requires a Builder Context Bundle containing only the relevant files listed in the Manifest.

Part III: Standard Operating Protocols

1. Initialization

- **1.1 Role Declaration:** The User declares the session role (Architect or Builder).
 - *Notification Trigger:* If I detect items in **Appendix A (Future Work)**, I must issue a one-time notification: *"Current Workflow Limitation: [Item Name] is marked as Future Work."*
- **1.2 Architect Context Receipt:** (Architect Only) The Architect parses the uploaded **Project Context Bundle** to establish the Immutable Base-line. The Architect must confirm the file count and general structure before proceeding to Analysis.
- **1.3 Builder Context Receipt:** (Builder Only) The Builder parses the uploaded bundle and lists the files received to verify they match the Architect's Manifest.
- **1.4 Session Version Lock:** (Builder Only) Immediately after parsing the bundle, the Builder **MUST** halt and request the **Target Session Version Number**. The Builder is forbidden from inferring this version from existing file headers.

2. Task Execution (Architect Phase)

- **2.1 Scope Fence Declaration:** Before planning, the Architect lists:
 - **In-Scope:** Items to be changed.
 - **Out-of-Scope:** Specific features/functions that must remain *untouched*.
- **2.2 Tiered Anchor Protocol:** The Architect must prove context awareness:
 - *Tier 1 (Surgical):* Quote 3 lines of existing code.
 - *Tier 2 (Refactor):* Quote the function signature to be preserved.
 - *Tier 3 (New File):* Reference an existing file as a pattern.
- **2.3 Implementation Plan Generation:** The Plan must include:
 - **Delivery Format:** The Implementation Plan, Manifest, and Builder Prompts must be delivered in a **single Canvas Document** to facilitate export and offline saving.
 - **Dependency Audit:** The Architect must list all modules imported by the target file (e.g., `contest_log.py`, `report_interface.py`) and include them in the Manifest as "Read-Only Context" to prevent "Blind Builder" errors.
 - **Manifest Generation:** The Architect **MUST** provide a dedicated code block labeled `manifest.txt` within the Canvas.

- * **Content:** This block must contain a plain-text list of all file paths (Target, Context, and the Workflow itself) required for the Builder, one per line.
- * **Format:** No comments, no labels (like "[Target]"), just the clean relative file paths.
- * **Verification Strategy:** Exact CLI command or "Manual Inspection".
- **Builder Bootstrap Prompt:** A pre-formatted code block for the user to copy/paste into the Builder session (see User Guide).

3. Task Execution (Builder Phase)

- **3.1 Visual Diff Preview:** Before generating a file, the Builder displays a snippet of "Old vs. New" code.
- **3.2 User Authorization:** The User reviews the diff. If correct, User types Proceed.
- **3.3 File Delivery:** The Builder provides the full file content in a code block.
 - **Constraint:** No internal file headers (like --- FILE ---).
 - **Constraint: Sanitized Output:** To prevent breaking the UI, all internal triple-backticks ("") must be replaced with the literal placeholder `__CODE_BLOCK__`.
- **3.4 Exact Prompts:** The Builder must provide the exact text for the User's next action (e.g., "Please confirm by typing 'Acknowledged'").

4. Error Recovery

- **4.1 Builder Iteration:** If verification fails, the User provides the error. The Builder attempts ONE fix.
- **4.2 Architect Rollback:** If the fix fails, the User must abort and return to a fresh Architect session with the error log.

5. Technical Data Standards

(Rescued from v1.3.1 - These apply to ALL file generation)

5.1. Versioning Protocol.

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:** Docs/AIAgentWorkflow.md and Docs/AIAgentUsersGuide.md maintain independent version series.
2. **Format:** The official versioning format is x.y.z-Beta.

3. **Source of Truth:** The version number in the main file header is the absolute source of truth.
4. **Update Procedure:** When a file is modified, the Builder **MUST** calculate the new version using the Target Session Version provided in Protocol 1.4 as a seed:
 - A. Extract the 'Target Series' (Major.Minor) from the User's input (e.g., input '0.93.0' -> series '0.93').
 - B. Read the 'Existing Version' from the file header.
 - C. **RESET Rule:** If 'Existing Version' does NOT start with 'Target Series', set the new version to the exact User input.
 - D. **INCREMENT Rule:** If 'Existing Version' DOES start with 'Target Series', increment the existing patch number by 1 (e.g., '0.93.4' -> '0.93.5').
5. **History Preservation:** All existing revision histories must be preserved and appended to, never regenerated from scratch.

5.2. Temporary Column Preservation Protocol.

Any temporary column used in processing (e.g., by a custom parser) 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, and any column not on this list will be discarded.

5.3. ASCII-7 Deliverable Mandate.

All files delivered by the AI must contain only 7-bit ASCII characters. This strictly forbids the use of extended Unicode characters.

5.4. JSON Inheritance Protocol. If a JSON file includes "inherits_from": "<parent_name>", the ContestDefinition.from_json method must recursively load the parent and perform a deep merge, with the child overriding the parent.

Part IV: Project-Specific Implementation Patterns

CRITICAL ARCHITECTURAL MANDATE: These patterns must be explicitly selected and specified by the **Architect** in the Implementation Plan. The **Builder** is not authorized to select these patterns autonomously.

6.1. Custom Parser Protocol. For contests with complex exchanges. Use "custom_parser_module" in JSON and place implementation in contest_specific_annotations/.
6.2. Per-Mode Multiplier Protocol. For contests where multipliers are counted per mode. Use "multiplier_report_scope": "per_mode" in JSON.
6.3. Data-Driven Scoring Protocol. Use "score_formula" in JSON (qsos_times_mults, total_points, or default).
6.4. Text Table Generation Protocol. Use prettytable for complex layouts. Use tabulate for simple lists.
6.5. Component Modernization Protocol. Implementation plans for modernization must include a parallel verification step (Legacy vs. Modern) before deprecating the old component.
6.6. Custom ADIF Exporter Protocol. Use "custom_adif_exporter" in JSON and place implementation in contest_tools/adif_exporters/.

Part V: Engineering Heuristics

(Critical problem-solving strategies reserved from v1.3)

7.1. Systemic Bug Eradication Protocol.

When a bug is identified in one module, the default assumption is that the same flaw exists in all other similar modules. The Architect must perform a global search for that specific bug pattern and include all instances in the Implementation Plan.

7.2. External System Interference Protocol.

If a file system error (e.g., PermissionError) is repeatable but cannot be traced to the script's logic, the AI must explicitly consider external causes (OneDrive, Antivirus) and propose diagnostic commands (e.g., attrib, git ls-files) rather than code workarounds.

7.3. File Naming Convention.

All generated report files must adhere to the standardized naming convention: <report_id>_<details>_<callsigns>.<ext>.

7.4. Debug "A Priori". If an initial bug fix fails or the cause is unclear, the first diagnostic step is to add detailed logging.info() statements to the failing code path to observe the runtime state.

Part VI: Special Case & Recovery Protocols

8.1. Mutual State Verification. If an instruction from the user appears to contradict the established project state or Manifest, the AI must pause and ask for

clarification.

8.2. Builder Context Verification. The Builder must strictly compare the actual files received against the Manifest. Any missing "Context Files" must be requested immediately.

8.3. Error Analysis Protocol. When an error is identified, the AI must provide a clear analysis in the mandatory three-part format: A. Confirmed Facts (No assumptions), B. Hypothesis (Root cause), C. Proposed Action.

8.4. Architect Rollback Protocol. If a fix attempted by the Builder fails validation, the Builder is NOT authorized to attempt a second random fix. It must declare a "Critical Strategy Failure" and advise the User to return to the Architect.

8.5. Ad-Hoc Task Protocol. For self-contained tasks unrelated to the primary project scope (e.g., "Fix a typo in Readme"), the strict Architect/Builder Role Separation is suspended. The session proceeds conversationally.

Appendix A: Future Work & Known Limitations

- **Automated Regression Testing:** The Builder cannot currently run the `run_regression_test.py` suite. This remains a manual User task.