Contest Log Analyzer - Programmer's Guide

Version: 0.88.2-Beta Date: 2025-09-21

--- Revision History ---

[0.88.2-Beta] - 2025-09-21

Fixed

- Corrected the function signature for Custom Multiplier Resolvers in the

Implementation Contracts to use `my_location_type: str`.

- Added the missing `custom_location_resolver` key to the JSON Quick Reference.

[0.87.0-Beta] - 2025-09-18

Added

- Added a "High-Level Data Annotation Workflow" section to clarify the

data processing pipeline for developers.

- Expanded all module "Implementation Contracts" to include "When to Use",

"Input DataFrame State", and "Responsibility" clauses.

Fixed

- Corrected the function signature for Custom Multiplier Resolvers to

match the actual four-argument contract in the source code.

[0.86.7-Beta] - 2025-09-15

Changed

- Updated JSON Quick Reference table to include `time_series_calculator` [2]

and `points_header_label` keys.

- Updated function signatures in the Implementation Contracts to include

Introduction

This document provides a technical guide for developers (both human and AI) looking to extend the functionality of the Contest Log Analyzer. The project is built on a few core principles:

- **Data-Driven:** The behavior of the analysis engine is primarily controlled by data, not code. Contest rules, multiplier definitions, and parsing logic are defined in simple `.json` files. This allows new contests to be added without changing the core Python scripts.
- **Extensible:** The application is designed with a "plugin" architecture. New reports and contest-specific logic modules can be dropped into the appropriate directories, and the main engine will discover and integrate them automatically.
- **Convention over Configuration:** This extensibility relies on convention. The dynamic discovery of modules requires that files and classes be named and placed in specific, predictable locations.

Core Components

Command-Line Interface (`main_cli.py`)

This script is the main entry point for running the analyzer.

- **Argument Parsing:** It uses Python's `argparse` to handle command-line arguments. Key arguments include:
  - `log_files`: A list of one or more log files to process.
  - `--report`: Specifies which reports to run. This can be a single `report_id`, a comma-separated list of IDs, the keyword `all`, or a category keyword (`chart`, `text`, `plot`, `animation`, `html`).
  - `--verbose`: Enables `INFO`-level debug logging.
  - `--include-dupes`: An optional flag to include duplicate QSOs in report calculations.
  - `--mult-name`: An optional argument to specify which multiplier to use for multiplier-specific reports (e.g., 'Countries').
  - `--metric`: An optional argument for difference plots, specifying whether to compare `qsos` or `points`. Defaults to `qsos`.
  - `--debug-data`: An optional flag to save the source data for visual reports to a text file.
  - `--debug-mults`: An optional flag to save intermediate multiplier lists from text reports for debugging.
- **Report Discovery:** The script dynamically discovers all available reports by inspecting the `contest_tools.reports` package. Any valid report class in this package is automatically made available as a command-line option.

Logging System (`Utils/logger_config.py`)

The project uses Python's built-in `logging` framework for console output.

- **`logging.info()`:** Used for verbose, step-by-step diagnostic messages. These are only displayed when the `--verbose` flag is used.
- **`logging.warning()`:** Used for non-critical issues the user should be aware of (e.g., ignoring an `X-QSO:` line). These are always displayed.
- **`logging.error()`:** Used for critical, run-terminating failures (e.g., a file not found or a fatal parsing error).

Regression Testing (`run_regression_test.py`)

The project includes an automated regression test script to ensure that new changes do not break existing functionality.

- **Workflow**: The script follows a three-step process:
  1. **Archive**: It archives the last known-good set of reports by renaming the existing `reports/` directory with a timestamp.
  2. **Execute**: It runs a series of pre-defined test cases from a `regressiontest.bat` file. Each command in this file generates a new set of reports.
  3. **Compare**: It performs a `diff` comparison between the newly generated text reports and the archived baseline reports. Any differences are flagged as a regression.
- **Methodology**: This approach focuses on **data integrity**. Instead of comparing images or videos, which can be brittle, the regression test compares the raw text output and the debug data dumps from visual reports. This provides a robust and reliable way to verify that the underlying data processing and calculations remain correct after code changes.

How to Add a New Report

The Report Interface

All reports must be created as `.py` files in the `contest_tools/reports/` directory. For the program to recognize a report, it must adhere to the contract defined by the `ContestReport` base class.

The `ContestReport` Base Class

This abstract base class, defined in `contest_tools/reports/report_interface.py`, provides the required structure for all report modules. A new report **must** inherit from this class and implement its required attributes and methods. ___CODE_BLOCK___python

Excerpt from contest_tools/reports/report_interface.py

from abc import ABC, abstractmethod from typing import List from ..contest_log import ContestLog

class ContestReport(ABC): # --- Required Class Attributes --- report_id: str = "abstract_report" report_name: str = "Abstract Report" report_type: str = "text" # 'text', 'plot', 'chart', 'animation', or 'html'

```
supports_single: bool = False
supports_pairwise: bool = False
supports_multi: bool = False

def __init__(self, logs: List[ContestLog]):
    # ... constructor logic ...


@abstractmethod
def generate(self, output_path: str, **kwargs) -> str:
    # ... your report logic goes here ...
    pass
```

CODE_BLOCK

Boilerplate Example

Here is a minimal "Hello World" report. ___CODE_BLOCK___python

contest_tools/reports/text_hello_world.py

from .report_interface import ContestReport

class Report(ContestReport): report_id = "hello_world" report_name = "Hello World Report" report_type = "text" supports_single = True

```
def generate(self, output_path: str, **kwargs) -> str:
    log = self.logs[0]
    callsign = log.get_metadata().get('MyCall', 'N/A')
    report_content = f"Hello, {callsign}!"
```

In a real report, you would save this content to a file.

```
    print(report_content)

    return f"Report '{self.report_name}' generated successfully."
```

CODE_BLOCK

How to Add a New Contest

Adding a new contest is primarily a data-definition task that involves creating a `.json` file and, if necessary, contest-specific Python modules.

High-Level Data Annotation Workflow (`contest_log.py`)

After a log is parsed, `contest_log.py` enriches the data in the following strict order. When writing a custom module, you can rely on all previous steps having been completed:

1. **Universal Annotations**: `Run/S&P` and `DXCC/Zone` lookups are applied. The `Run`, `DXCCName`, `DXCCPfx`, `CQZone`, `ITUZone`, `Continent`, `WAEName`, and `WAEPfx` columns are now available.
2. **Mode Normalization**: The `Mode` column is standardized (e.g., `FM` is mapped to `PH`, `RY` to `DG`).
3. **Custom Multiplier Resolver**: If `custom_multiplier_resolver` is defined in the JSON, this module is executed.
4. **Standard Multiplier Rules**: The generic `multiplier_rules` from the JSON are processed.
5. **Scoring**: The contest-specific scoring module is executed to calculate the `QSOPoints` column.

The Core Data Columns

After parsing, all log data is normalized into a standard pandas DataFrame. The available columns are defined in `contest_tools/contest_definitions/_common_cabrillo_fields.json`. When creating exchange parsing rules, the `groups` list **must** map to these column names. **Available Columns**: `ContestName`, `CategoryOverlay`, `CategoryOperator`, `CategoryTransmitter`, `MyCall`, `Frequency`, `Mode`, `Datetime`, `SentRS`, `SentRST`, `SentZone`, `SentNR`, `Call`, `RS`, `RST`, `Zone`, `NR`, `Transmitter`, `RawQSO` (*Note: This is an intermediate column used for diagnostics during parsing and is removed before the final DataFrame is available for reporting*), `Band`, `Date`, `Hour`, `Dupe`, `DXCCName`, `DXCCPfx`, `CQZone`, `ITUZone`, `Continent`, `WAEName`, `WAEPfx`, `Lat`, `Lon`, `Tzone`, `portableid`, `Run`, `QSOPoints`, `Mult1`, `Mult1Name`, `Mult2`, `Mult2Name`.

JSON Quick Reference

Create a new `.json` file in `contest_tools/contest_definitions/`. The following table describes the available keys.

| Key | Description | Example Value | | --- | --- | --- | | `contest_name` | The official name from the Cabrillo `CONTEST:` tag. | `"CQ-WW-CW"` | |

`dupe_check_scope` | Determines if dupes are checked `per_band` or across `all_bands`. | `"per_band"` | | `exchange_parsing_rules` | An object containing regex patterns to parse the exchange portion of a QSO line. | `{ "NAQP-CW": [ { "regex": "...", "groups": [...] } ] }` | | `multiplier_rules` | A list of objects defining the contest's multipliers. | `[ { "name": "Zones", "source_column": "CQZone", "value_column": "Mult1" } ]` | | `mutually_exclusive_mults` | *Optional.* Defines groups of multiplier columns that are mutually exclusive for a single QSO, used for diagnostic reporting. | `[["Mult1", "Mult2"]]` | | `score_formula` | Scoring method. Can be `total_points`, `qsos_times_mults`, or `points_times_mults`. | `"points_times_mults"` | | `multiplier_report_scope`| Determines if mult reports run `per_band` or `per_mode`. | `"per_band"` | | `excluded_reports`| A list of `report_id` strings to disable for this contest. | `[ "point_rate_plots" ]` | | `operating_time_rules`| Defines on-time limits for the `score_report`. | `{ "single_op_max_hours": 30 }` | | `mults_from_zero_point_qsos`| `True` if multipliers count from 0-point QSOs. | `true` | | `enable_adif_export` | `True` if the log should be exported to an N1MM-compatible ADIF file. | `true` | | `valid_bands` | A list of bands valid for the contest. | `[ "160M", "80M", "40M" ]` | | `contest_period` | Defines the official start/end of the contest. | `{ "start_day": "Saturday" }` | | `custom_parser_module` | *Optional.* Specifies a module to run for complex, asymmetric parsing. | `"arrl_10_parser"` | | `custom_multiplier_resolver` | *Optional.* Specifies a module to run for complex multiplier logic (e.g., NAQP). | `"naqp_multiplier_resolver"` | | `custom_adif_exporter` | *Optional.* Specifies a module to generate a contest-specific ADIF file. | `"iaru_hf_adif"` | | `custom_location_resolver` | *Optional.* Specifies a module to determine the logger's location type for asymmetric contests (e.g., W/VE vs DX). | `"arrl_dx_location_resolver"` | | `time_series_calculator` | *Optional.* Specifies a module to calculate the time-series score. Defaults to `standard_calculator`. | `"wae_calculator"` | | `points_header_label` | *Optional.* A custom label for the "Points" column in score reports for contests with complex scoring (e.g., WAE). | `"QSO+QTC Pts"` | | `contest_specific_event_id_resolver` | *Optional.* Specifies a module to create a unique event ID for contests that run multiple times a year. | `"naqp_event_id_resolver"` | | `scoring_module` | *Implied.* The system looks for a `[contest_name]_scoring.py` file with a `calculate_points` function. | N/A (Convention-based) | | `cabrillo_version` | The Cabrillo version for the log header. | `"3.0"` | | `qso_common_fields_regex`| *Deprecated.* Regex to parse the non-exchange part of a QSO line. This is now handled internally by the parser. | `"QSO:\\s+(\\d+)..."` | | `qso_common_field_names` | A list of names for the groups in the common regex. | `["FrequencyRaw", "Mode"]` | | `default_qso_columns` | The complete, ordered list of columns for the final DataFrame. | `["MyCall", "Frequency", "Mode"]` | | `scoring_rules` | *Legacy.* Defines contest-specific point values. | `{"points_per_qso": 2}` |

The Annotation and Scoring Workflow (`contest_log.py`)

After initial parsing, `contest_log.py` orchestrates a sequence of data enrichment steps. This is the plug-in system for contest-specific logic. The sequence is defined in the `apply_contest_specific_annotations` method. A developer needing to add complex logic should reference this file to understand the workflow. **Sequence of Operations:**

1. **Universal Annotations**: Run/S&P and DXCC/Zone lookups are applied to all logs.
2. **Mode Normalization**: The `Mode` column is standardized (e.g., `FM` is mapped to `PH`, `RY` to `DG`).
3. **Custom Multiplier Resolver**: If `custom_multiplier_resolver` is defined in the JSON, the specified module is dynamically imported and its `resolve_multipliers` function is executed.
4. **Standard Multiplier Rules**: The system processes the `multiplier_rules` from the JSON. If a rule has `"source": "calculation_module"`, it dynamically imports and runs the specified function. This is how WPX prefixes are calculated.
5. **Scoring**: The system looks for a scoring module by convention (e.g., `cq_ww_cw_scoring.py`) and executes its `calculate_points` function.

A Note on `__init__.py` Files

The need to update an `__init__.py` file depends on whether a package uses dynamic or explicit importing.

- **Dynamic Importing (No Update Needed):** Directories like `contest_tools/contest_specific_annotations` and `contest_tools/reports` are designed as "plug-in" folders. The application uses dynamic importing (`importlib.import_module`) to load these modules by name from the JSON definitions or by discovery. Therefore, the `__init__.py` files in these directories are intentionally left empty and **do not need to be updated** when a new module is added.
- **Explicit Importing (Update Required):** When a new parameter is added to a `.json` file, the `ContestDefinition` class in `contest_tools/contest_definitions/__init__.py` **must be updated**. A new `@property` must be added to the class to expose the new data from the JSON file to the rest of the application. This is a critical maintenance step for extending the data model. Similarly, packages like `contest_tools/core_annotations` use their `__init__.py` to explicitly expose functions and classes, and would need to be updated if a new core utility were added.

Advanced Guide: Extending Core Logic (Implementation Contracts)

For contests requiring logic beyond simple JSON definitions, create a Python module in `contest_tools/contest_specific_annotations/`. Each module type has a specific contract (required function and signature) it must fulfill.

- **Custom Parser Modules**:
  - **Purpose**:
    - **When to Use**: When a contest's exchange is too complex or asymmetric to be defined by a single regular expression in the JSON file (e.g., ARRL DX).
  - **Input DataFrame State**: This is the first processing step; it receives the raw file path, not a DataFrame.
  - **Responsibility**: To parse the raw Cabrillo file and return a DataFrame of QSOs and a dictionary of metadata. The custom parser is now part of a **mandatory two-stage process**: it must first call the shared `parse_qso_common_fields` helper from the main `cabrillo_parser.py` module to handle the fixed fields (frequency, mode, date, etc.). The custom parser's only remaining job is to parse the `ExchangeRest` string that the helper returns.
  - **Required Function Signature**: `parse_log(filepath: str, contest_definition: ContestDefinition, root_input_dir: str) -> Tuple[pd.DataFrame, Dict[str, Any]]`
  - **Note on Temporary Columns**: Any temporary columns created by the parser that are needed by a downstream module (like a custom resolver) **must** be included in the `default_qso_columns` list in the contest's JSON definition.
- **Custom Multiplier Resolvers**:
  - **Purpose**:
    - **When to Use**: When multiplier identification requires complex logic, such as looking up aliases from external `.dat` files (e.g., NAQP) or parsing prefixes from callsigns (e.g., CQ WPX).
  - **Input DataFrame State**: The DataFrame will have `Run`, `DXCCName`, `DXCCPfx`, `CQZone`, `ITUZone`, `Continent`, `WAEName`, and `WAEPfx` columns populated.
  - **Responsibility**: To populate the appropriate `Mult_` columns (e.g., `Mult_STPROV`) for *every* relevant QSO. It should **not** perform "first-worked" filtering; that is handled by the reporting engine.
- **Required Function Signature**: `resolve_multipliers(df: pd.DataFrame, my_location_type: str, root_input_dir: str, contest_def: ContestDefinition) -> pd.DataFrame`
- **Scoring Modules**:
  - **Purpose**:
    - **When to Use**: For any contest that requires more than a simple "points per QSO" rule defined in the legacy `scoring_rules` JSON key. This is the standard method for all new contests.
    - **Input DataFrame State**: The DataFrame will have all universal annotations and all `Mult_` columns populated.
  - **Responsibility**: To calculate the point value for every QSO and return the results as a pandas Series that will become the `QSOPoints`

column.
    - **Required Function Signature**: `calculate_points(df: pd.DataFrame, my_call_info: Dict[str, Any]) -> pd.Series`
- **Custom ADIF Exporter Modules**:
    - **Purpose**:
        * **When to Use**: When a contest's ADIF output requires specific, non-standard tags (e.g., `<APP_N1MM_HQ>`) or the conditional omission of standard tags for compatibility with external programs like N1MM Logger+.
    - **Input DataFrame State**: The exporter receives the final, fully processed `ContestLog` object.
    - **Responsibility**: To generate and save a complete, spec-compliant `.adi` file.
    - **Required Function Signature**: `export_log(log: ContestLog, output_filepath: str)`
    - **Location**: `contest_tools/adif_exporters/`
    - **Implementation Details and Conventions**:
        * **External Tool Compatibility**: Custom exporters must be aware of the specific tags required by external programs. For example, N1MM Logger+ uses `<APP_N1MM_HQ>` for IARU HQ/Official multipliers.
        * **Conditional Tag Omission**: A critical function of a custom exporter is to conditionally *omit* standard ADIF tags when required by a contest's rules to ensure correct scoring by external tools. For the IARU contest, the standard `<ITUZ>` tag must be omitted for any QSO that provides an HQ or Official multiplier.
        * **Redundant `APP_CLA_` Tags**: To ensure our own ADIF files are self-descriptive for future use (e.g., log ingestion by CLA), it is a project convention to include redundant, parallel `APP_CLA_` tags for all contest-specific data. For example, an IARU HQ QSO should contain both `<APP_N1MM_HQ:4>DARC` for N1MM and a corresponding `<APP_CLA_MULT_HQ:4>DARC` for our own tools.
        * **Timestamp Uniqueness**: The Cabrillo format provides only minute-level precision (`HHMM`), but external tools like N1MM require unique timestamps (`HHMMSS`) to avoid double-counting multipliers. The generic `export_to_adif` method in `contest_log.py` now handles this by adding a per-second offset to QSOs occurring in the same minute. Any custom ADIF exporter must replicate this behavior to ensure compatibility.
- **Utility for Complex Multipliers (`_core_utils.py`)**:
    - For contests with complex multiplier aliases (like NAQP or ARRL DX), developers should use the `AliasLookup` class found in `contest_tools/core_annotations/_core_utils.py`. This utility is designed to be used within a custom multiplier resolver to parse `.dat` alias files.

Advanced Report Design: Shared Logic

A key architectural principle for creating maintainable and consistent reports is the **separation of data aggregation from presentation**. When multiple reports need to display the same underlying data in different formats (e.g., HTML and plain text), the data aggregation logic should not be duplicated.

The Shared Aggregator Pattern

The preferred method is to create a dedicated, non-report helper module within the `contest_tools/reports/` directory. This module's sole responsibility is to perform the complex data calculations and return a clean, structured data object (like a dictionary or pandas DataFrame).

Example: `_qso_comparison_aggregator.py` To generate both `html_qso_comparison` and `text_qso_comparison` reports, we can create a shared helper:

1. **Create the Aggregator**: A new file, `_qso_comparison_aggregator.py`, would contain a function like `aggregate_qso_comparison_data(logs)`. This function would perform all the necessary calculations (Unique QSOs, Common QSOs, Run/S&P breakdowns, etc.) and return a final dictionary.
2. **Update the Report Modules**:
   - `html_qso_comparison.py` would import and call this function. Its only remaining job would be to take the returned data and render it into the final HTML string.
   - `text_qso_comparison.py` would also import and call the *same* function. Its job would be to take the data and render it into a fixed-width text table using a tool like pandas' `to_string()` method. This pattern ensures that both reports are always based on the exact same data, eliminating the risk of inconsistencies and reducing code duplication.

Appendix: Key Source Code References

This appendix lists the most important files for developers to consult to understand the application's framework. The sections above provide context and instructions on how these files are used.

- **`contest_tools/contest_definitions/_common_cabrillo_fields.json`**: The definitive source for all available DataFrame column names. Essential for writing exchange parsing rules.
- **`contest_tools/reports/report_interface.py`**: Defines the `ContestReport` abstract base class that all new reports must inherit from.

- **`contest_tools/contest_log.py`**: The central orchestrator for applying contest-specific logic, including custom parsers, multiplier resolvers, and scoring modules.
- **`contest_tools/core_annotations/_core_utils.py`**: Contains shared utilities, most notably the `AliasLookup` class for handling complex multiplier aliases.