Contest Log Analyzer - Programmer's Guide

Version: 0.91.2-Beta Date: 2025-10-10

--- Revision History ---

[0.91.2-Beta] - 2025-10-10

Changed

- Documented the new hybrid report model (`is_specialized` flag

and `included_reports` JSON key).

[0.90.17-Beta] - 2025-10-06

Added

- Added a new implementation contract for "Event ID Resolver Modules"

to document the pluggable architecture for handling contests with

multiple events per year.

[0.90.14-Beta] - 2025-10-06

Fixed

- Added the missing `--cty` argument to the CLI documentation.

- Corrected the return signature for Custom Parser Modules in the

Implementation Contracts to account for variable return tuples (e.g.,

for QTC data).

[0.90.10-Beta] - 2025-09-30

Changed

- Updated the implementation contract for Custom Parser Modules to

reflect the new, four-argument function signature.

[0.89.0-Beta] - 2025-09-29

Added

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.
  - `--cty <specifier>`: An optional argument to specify the CTY file: 'before', 'after' (default), or a specific filename (e.g., 'cty-3401.dat').
  - `--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 Contest: A Step-by-Step Guide

This guide walks you through the process of adding a new, simple contest called "My Contest". This contest will have a simple exchange (RST + Serial Number) and one multiplier (US States).

Step 1: Create the JSON Definition File

Navigate to the `contest_tools/contest_definitions/` directory and create a new file named `my_contest.json`. The filename (minus the extension) is the ID used to find the contest's rules.

Step 2: Define Basic Metadata

Open `my_contest.json` and add the basic information. The `contest_name` must exactly match the `CONTEST:` tag in the Cabrillo log files for this contest.

```
{
  "contest_name": "MY-CONTEST",
  "dupe_check_scope": "per_band",
  "score_formula": "points_times_mults",
  "valid_bands": ["80M", "40M", "20M", "15M", "10M"]
}
```_

### Step 3: Define the Exchange
Next, add the `exchange_parsing_rules`. For this example, the exchange is RST + Serial Numbe
```_json
"exchange_parsing_rules": {
  "MY-CONTEST": {
    "regex": "(\\d{3})\\s+(\\d+)\\s+([A-Z0-9/]+)\\s+(\\d{3})\\s+(\\d+)",
    "groups": ["SentRST", "SentSerial", "Call", "RST", "RcvdSerial"]
  }
},
```_

### Step 4: Define the Multipliers
Add the `multiplier_rules`. We want to count US States as multipliers. We'll tell the systen
```_json
"multiplier_rules": [
  {
    "name": "States",
    "source_column": "RcvdLocation",
    "value_column": "Mult1"
  }
]
```_

### Step 5: Create the Scoring Module
Create a new file in `contest_tools/contest_specific_annotations/` named `my_contest_scoring
```_python
import pandas as pd
from typing import Dict, Any

def calculate_points(df: pd.DataFrame, my_call_info: Dict[str, Any]) -> pd.Series:
    points = pd.Series(1, index=df.index)
    if 'Dupe' in df.columns:
        points[df['Dupe'] == True] = 0
```

```
    return points
```_

### Step 6: Run and Verify
You can now run the analyzer on a log file for "MY-CONTEST". The system will use your new JS
---
## How to Add a New Report

### The Report Interface
All reports must be created as `.py` files in the `contest_tools/reports/` directory. For th
### The `ContestReport` Base Class
This abstract base class, defined in `contest_tools/reports/report_interface.py`, provides t
```_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'
    is_specialized: bool = False # False = Generic (opt-out), True = Specialized (opt-in)

    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
```_

### Boilerplate Example
Here is a minimal "Hello World" report.
```_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."
```_

---
## How to Add a New Contest

Adding a new contest is primarily a data-definition task that involves creating a `.json` fi
### High-Level Data Annotation Workflow (`contest_log.py`)
After a log is parsed, `contest_log.py` enriches the data in the following strict order. Whe
1.  **Universal Annotations**: `Run`/`S&P` and `DXCC`/`Zone` lookups are applied. The `Run`
2.  **Mode Normalization**: The `Mode` column is standardized (e.g., `FM` is mapped to `PH`
3.  **Custom Multiplier Resolver**: If `custom_multiplier_resolver` is defined in the JSON,
4.  **Standard Multiplier Rules**: The generic `multiplier_rules` from the JSON are processe
5.  **Scoring**: The contest-specific scoring module is executed to calculate the `QSOPoints
### The Core Data Columns
After parsing, all log data is normalized into a standard pandas DataFrame. The available co
**Available Columns**: `ContestName`, `CategoryOverlay`, `CategoryOperator`, `CategoryTransm
### JSON Quick Reference
Create a new `.json` file in `contest_tools/contest_definitions/`. The following table descr

| 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`. | `
| `exchange_parsing_rules` | An object containing regex patterns to parse the exchange porti
| `multiplier_rules` | A list of objects defining the contest's multipliers. | `[ { "name":
| `mutually_exclusive_mults` | *Optional.* Defines groups of multiplier columns that are mut
| `score_formula` | Scoring method. Can be `total_points`, `qsos_times_mults`, or `points_ti
| `multiplier_report_scope`| Determines if mult reports run `per_band` or `per_mode`. | `"pe
| `excluded_reports`| A list of generic `report_id` strings to disable for this contest (opt
| `included_reports`| *Optional.* A list of specialized `report_id` strings to explicitly en
| `operating_time_rules`| Defines on-time limits for the `score_report`. | `{ "single_op_max

```
| `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 fil
| `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": "Satur
| `custom_parser_module` | *Optional.* Specifies a module to run for complex, asymmetric par
| `custom_multiplier_resolver` | *Optional.* Specifies a module to run for complex multiplie
| `custom_adif_exporter` | *Optional.* Specifies a module to generate a contest-specific ADI
| `custom_location_resolver` | *Optional.* Specifies a module to determine the logger's loca
| `time_series_calculator` | *Optional.* Specifies a module to calculate the time-series sco
| `points_header_label` | *Optional.* A custom label for the "Points" column in score report
| `contest_specific_event_id_resolver` | *Optional.* Specifies a module to create a unique e
| `scoring_module` | *Implied.* The system looks for a `[contest_name]_scoring.py` file with
| `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 lin
| `qso_common_field_names` | A list of names for the groups in the common regex. | `["Freque
| `default_qso_columns` | The complete, ordered list of columns for the final DataFrame. | `
| `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. Th
**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`,
3.  **Custom Multiplier Resolver**: If `custom_multiplier_resolver` is defined in the JSON,
4.  **Standard Multiplier Rules**: The system processes the `multiplier_rules` from the JSON
5.  **Scoring**: The system looks for a scoring module by convention (e.g., `cq_ww_cw_scorin
### A Note on `__init__.py` Files
The need to update an `__init__.py` file depends on whether a package uses dynamic or explic
* **Dynamic Importing (No Update Needed):** Directories like `contest_tools/contest_specific
* **Explicit Importing (Update Required):** When a new parameter is added to a `.json` file,
### Advanced Guide: Extending Core Logic (Implementation Contracts)
For contests requiring logic beyond simple JSON definitions, create a Python module in `cont
* **Custom Parser Modules**:
    * **Purpose**:
        * **When to Use**: When a contest's exchange is too complex or asymmetric to be defi
    * **Input DataFrame State**: This is the first processing step; it receives the raw file
    * **Responsibility**: To parse the raw Cabrillo file and return a DataFrame of QSOs and
    * **Required Function Signature**: `parse_log(...) -> Union[Tuple[pd.DataFrame, Dict], T
    * **Note on Temporary Columns**: Any temporary columns created by the parser that are ne
* **Custom Multiplier Resolvers**:
    * **Purpose**:
        * **When to Use**: When multiplier identification requires complex logic, such as lo
    * **Input DataFrame State**: The DataFrame will have `Run`, `DXCCName`, `DXCCPfx`, `CQZo
    * **Responsibility**: To populate the appropriate `Mult_` columns (e.g., `Mult_STPROV`)
* **Required Function Signature**: `resolve_multipliers(df: pd.DataFrame, my_location_type:
* **Scoring Modules**:
    * **Purpose**:
```

```
        * **When to Use**: For any contest that requires more than a simple "points per QSO"
        * **Input DataFrame State**: The DataFrame will have all universal annotations and a
    * **Responsibility**: To calculate the point value for every QSO and return the results
    * **Required Function Signature**: `calculate_points(df: pd.DataFrame, my_call_info: Dic
    * **Note on Naming**: The scoring module is loaded by convention. The filename must be `
    * **Custom ADIF Exporter Modules**:
    * **Purpose**:
        * **When to Use**: When a contest's ADIF output requires specific, non-standard tags
    * **Input DataFrame State**: The exporter receives the final, fully processed `ContestLo
    * **Responsibility**: To generate and save a complete, spec-compliant `.adi` file.
    * **Event ID Resolver Modules**:
    * **Purpose**:
        * **When to Use**: For any contest that runs multiple times per year and requires a
    * **Input**: The function receives a pandas `Timestamp` object from the first QSO in the
    * **Responsibility**: To return a short, directory-safe string that uniquely identifies
    * **Location**: `contest_tools/event_resolvers/`
    * **Required Function Signature**: `resolve_event_id(qso_datetime: pd.Timestamp) -> str`
    * **Location**: `contest_tools/adif_exporters/`
    * **Implementation Details and Conventions**:
        * **External Tool Compatibility**: Custom exporters must be aware of the specific ta
        * **Conditional Tag Omission**: A critical function of a custom exporter is to condi
        * **Redundant `APP_CLA_` Tags**: To ensure our own ADIF files are self-descriptive f
        * **Timestamp Uniqueness**: The Cabrillo format provides only minute-level precision
* **Utility for Complex Multipliers (`_core_utils.py`)**:
    * For contests with complex multiplier aliases (like NAQP or ARRL DX), developers should
---
## Advanced Report Design: Shared Logic
A key architectural principle for creating maintainable and consistent reports is the **sepa
### The Shared Aggregator Pattern
The preferred method is to create a dedicated, non-report helper module within the `contest_
#### Example: `_qso_comparison_aggregator.py`
To generate both `html_qso_comparison` and `text_qso_comparison` reports, we can create a sh
1.  **Create the Aggregator**: A new file, `_qso_comparison_aggregator.py`, would contain a
2.  **Update the Report Modules**:
    * `html_qso_comparison.py` would import and call this function. Its only remaining job w
    * `text_qso_comparison.py` would also import and call the *same* function. Its job would
This pattern ensures that both reports are always based on the exact same data, eliminating
---
## Appendix: Key Source Code References
This appendix lists the most important files for developers to consult to understand the app
* **`contest_tools/contest_definitions/_common_cabrillo_fields.json`**: The definitive sourc
* **`contest_tools/reports/report_interface.py`**: Defines the `ContestReport` abstract base
* **`contest_tools/contest_log.py`**: The central orchestrator for applying contest-specific
* **`contest_tools/core_annotations/_core_utils.py`**: Contains shared utilities, most notab
```