Contest Log Analyzer - Programmer's Guide

Version: 0.35.23-Beta Date: 2025-08-15

--- Revision History ---

[0.35.23-Beta] - 2025-08-15

Changed

- Updated the "Available Reports" list and the `--report` argument

description to be consistent with the current codebase.

[0.32.15-Beta] - 2025-08-12

Added

- Added documentation for the new "Custom Parser Module" plug-in pattern.

Changed

- Replaced nested markdown code fences with ``` placeholder.

[1.0.1-Beta] - 2025-08-11

Changed

- Updated CLI arguments, contest-specific module descriptions, and the

report interface to be fully consistent with the current codebase.

[1.0.0-Beta] - 2025-08-10

Added

- Initial release of the Programmer's Guide.

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`).
  - `--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').
- **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).

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 follow these conventions:

1. The file must contain a class named **`Report`**.
2. This class must inherit from the **`ContestReport`** base class.
3. The class must define the following required attributes:

Attribute	Type	Description
`report_id`	`str`	A unique, machine-friendly identifier (e.g., `score_report`). Used in the `--report` argument.
`report_name`	`str`	A human-friendly name for the report (e.g., "Score Summary").
`report_type`	`str`	The category of the report. Currently `text`, `plot`, `chart`, or `animation`.
`supports_single`	`bool`	`True` if the report can be run on a single log.
`supports_multi`	`bool`	`True` if the report can be run on multiple logs (non-comparative).
`supports_pairwise`	`bool`	`True` if the report compares exactly two logs.

4. The class must implement a `generate(self, output_path: str, **kwargs) -> str` method. This method contains the core logic of the report and must accept `**kwargs` to handle optional arguments.

Dynamic Discovery

As long as a report file is in the `contest_tools/reports` directory and its class is named `Report`, the `__init__.py` in that directory will find and register it automatically.

Helper Functions and Factoring (`_report_utils.py`)

The `contest_tools/reports/_report_utils.py` module contains common helper functions. The philosophy for factoring is as follows:

- **Keep it Self-Contained:** If a piece of logic is highly specific to a single report and unlikely to be reused, it should remain inside that report's `generate` method.
- **Factor it Out:** If a function or component (like a chart style or data preparation step) is likely to be useful for other future reports, it should be factored out into a new helper function in `_report_utils.py`.

Boilerplate Example

Here is a minimal "Hello World" report.

```
# 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 can range from simple (creating a new `.json` file) to complex (extending the core parsing logic).

JSON Quick Reference

The primary way to add a contest is by creating a new `.json` file in the `contest_tools/contest_definitions/` directory. The following table describes the key attributes.

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" } ]`
`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`

`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.p` file with a `calculate_points` function.	N/A (Convention-based)

Basic Guide: Creating a New Contest Definition

1. Create a new `.json` file in `contest_tools/contest_definitions/`.
2. Define the `contest_name` to match the Cabrillo logs.
3. Define the `exchange_parsing_rules`. If the exchange can have multiple valid formats, you can provide a list of rule objects. The parser will try each one in order.
4. Define the `multiplier_rules`. For simple multipliers, you can use `"source_column"` to copy data from an existing column (like `CQZone` or `DXCCName`) into a multiplier column (`Mult1`, `Mult2`).

Boilerplate Example

```
{
  "_filename": "contest_tools/contest_definitions/my_contest.json",
  "_version": "1.0.0-Beta",
  "_date": "2025-08-10",
  "contest_name": "MY-CONTEST-CW",
  "dupe_check_scope": "per_band",
  "exchange_parsing_rules": {
    "MY-CONTEST-CW": {
      "regex": "(\\d{3})\\s+(\\w+)",
      "groups": [ "RST", "RcvdExchangeField" ]
    }
  },
  "multiplier_rules": [
    {
      "name": "MyMults",
      "source_column": "RcvdExchangeField",
      "value_column": "Mult1",
      "totaling_method": "once_per_log"
    }
  ]
}
```

Advanced Guide: Extending Core Logic

If a contest requires logic that cannot be defined in JSON, you can extend the Python code. Create a new Python file in `contest_tools/contest_specific_annotations/` for any of the following modules.

- **Custom Parser Module:** Create a file (e.g., `my_contest_parser.py`) containing a `parse_log` function. In the `.json` file, set the `custom_parser_module` key to the module name (e.g., `"my_contest_parser"`). The `contest_log.py` script will call this module *instead of* the generic `cabrillo_parser.py` for that contest.
- **Custom Multiplier Resolver:** Create a file (e.g., `my_contest_resolver.py`) containing a `resolve_multipliers` function. In the `.json` file, set the `custom_multiplier_resolver` key to the module name (e.g., `"my_contest_resolver"`).
- **Event ID Resolver:** Create a file (e.g., `my_contest_event_resolver.py`) with a `resolve_event_id` function. Set the `contest_specific_event_id_resolver` key in the JSON.

- **Scoring Module:** Create a file named `my_contest_cw_scoring.py` containing a `calculate_points` function. The system will find this by convention.
- **Multiplier Calculation Module:** Create a file (e.g., `my_contest_mult_calc.py`) with a function that returns a pandas Series. In the JSON `multiplier_rules`, set `"source": "calculation_module"` and add the `module_name` and `function_name` keys. The `contest_log.py` script will see these rules, use the `importlib` library to dynamically load your module, execute your function, and integrate the results.