

Contest Log Analytics - Programmer's Guide

Version: 0.127.0-Beta Date: 2025-12-23

--- Revision History ---

[0.127.0-Beta] - 2025-12-23

Added

- Added "The Manifest & WORM Strategy" section to Phase 3.
- Documented `contest_tools/manifest_manager.py`.

[0.126.0-Beta] - 2025-12-18

Added

- Added "Phase 3: Web Architecture" section detailing the Django stateless design.
- Updated "The Data Abstraction Layer (DAL)" to include `ComparativeEngine`.
- Updated "Shared Utilities & Styles" to include `pivot_utils` and `log_fetcher`.

[0.94.1-Beta] - 2025-12-06

Changed

- Updated "The Data Abstraction Layer (DAL)" section to reflect the formal Aggregator Class architecture.

[0.94.0-Beta] - 2025-12-06

Added

- Added "The Data Abstraction Layer (DAL)" section.
- Added "Shared Utilities & Styles" section.

[0.91.1-Beta] - 2025-10-11

Added

- Added the "Python File Header Standard" section to document the mandatory header format for all .py files.

[0.91.0-Beta] - 2025-10-10

Introduction

This document provides a technical guide for developers (both human and AI) looking to extend the functionality of the Contest Log Analytics. 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.
-

Python File Header Standard

All Python (`.py`) files in the project must begin with the following standard header block. This ensures consistency and proper version tracking.

```
# {filename}.py
#
# Purpose: {A concise, one-sentence description of the module's primary responsibility.}
#
# Author: {Author Name}
# Date: {YYYY-MM-DD}
# Version: {x.y.z-Beta}
#
# Copyright (c) {Year} Mark Bailey, KD4D
# Contact: kd4d@kd4d.org
#
# License: Mozilla Public License, v. 2.0
#           (https://www.mozilla.org/MPL/2.0/)
#
# This Source Code Form is subject to the terms of the Mozilla Public
# License, v. 2.0.
# If a copy of the MPL was not distributed with this
# file, You can obtain one at http://mozilla.org/MPL/2.0/.
#
# --- Revision History ---
# [{version}] - {YYYY-MM-DD}
# - {Description of changes}
```

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').
 - `--wrtc <year>`: An optional argument to score IARU-HF logs using the rules for a specific WRTC year.
 - `--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.
-

Phase 3: Web Architecture

The project now includes a stateless, containerized web dashboard (`web_app`) built on Django.

Core Principles (ADR-007)

- **Stateless Operation:** The Django app does not use a database for domain logic. It relies on the file system (session directories) for data persistence.
- **Shared Presentation Layer:** The `web_app/config/settings.py` is configured to map the `TEMPLATES` setting to `contest_tools/templates`. This ensures that the CLI (which uses Jinja2) and the Web App (which uses Django Templates) share the exact same HTML report templates.

Key Components

- `web_app/analyzer/views.py`: Contains the view logic.
 - `analyze_logs`: Orchestrates the upload, parsing (via `LogManager`), and reporting (via `ReportGenerator`) pipeline. It serializes the dashboard context to `dashboard_context.json`.
 - `dashboard_view`: Loads the serialized context to render the "Strategy Board".
- `get_progress`: Provides a JSON endpoint for the client-side progress bar.

- `contest_tools/utils/log_fetcher.py`: A utility class used by the web view to scrape public contest log archives (e.g., CQ WW) and download logs on-demand.
- `contest_tools/manifest_manager.py`: Manages the `session_manifest.json`, providing a decoupled way for the UI to discover generated artifacts without hardcoding paths.

The Manifest & WORM Strategy

To ensure the dashboard remains responsive even with large datasets, the application employs a **”Write Once, Read Many” (WORM)** strategy for expensive aggregations.

1. **Generation Phase:** During the initial analysis (`analyze_logs`), specific “Artifact Reports” (e.g., `json_multiplier_breakdown.py`) are executed. These reports do not produce human-readable text but instead serialize processed aggregation trees into JSON files.
 2. **Hydration Phase:** When the user loads a dashboard view (e.g., `multiplier_dashboard`), the view uses the `ManifestManager` to locate the JSON artifact.
 3. **Rendering:** The view hydrates its context directly from this pre-computed JSON, avoiding the need to re-parse Cabrillo logs or re-run aggregators on every page load.
-

The Data Abstraction Layer (DAL)

The `contest_tools.data_aggregators` package is the sole authority for data summarization.

- **Design Principle:** “Pure Python Primitives”. To ensure complete decoupling between the data processing logic and the presentation layer (Reports, Web UI), all Aggregators must return standard Python types (Dictionaries, Lists, Ints, Strings). They **must not** return Pandas DataFrames or NumPy arrays.
- **Primary Aggregators:**
 - **CategoricalAggregator**: Handles set operations (Unique/Common QSOs), point breakdowns, and generic categorical grouping.
 - **ComparativeEngine**: Implements pure Set Theory logic to calculate Universe, Common, Differential, and Missed counts for any set of items.
 - **MatrixAggregator**: Generates 2D grids (Band x Time) for heatmaps and activity status tracking.
 - **MultiplierStatsAggregator**: Handles all multiplier summarization logic, including unique counts and “Missed Multiplier” analysis.

- **TimeSeriesAggregator:** Generates the standard TimeSeries Data Schema (v1.4.0), including cumulative rates, scores, and scalar metrics.
 - **WaeStatsAggregator:** Specialized logic for WAE contests, handling QTCs and weighted multiplier calculations.
-

Shared Utilities & Styles

- **contest_tools.styles.MPLStyleManager:** A centralized source for Matplotlib styles and color consistency. It provides methods like `get_point_color_map()` and `get_qso_mode_colors()` to ensure visual uniformity across different reports.
 - **contest_tools.utils.CtyManager:** Manages the lifecycle of the `cty.dat` country file, including downloading, version management, and local caching.
 - **contest_tools.utils.json_encoders.NpEncoder:** A custom JSON encoder class used to serialize NumPy data types (like `int64` or `float64`) and Pandas Timestamps into standard JSON formats.
 - **contest_tools.utils.pivot_utils:** Contains shared DataFrame pivoting logic to prevent circular imports between reports.
 - **contest_tools.utils.log_fetcher:** Provides the `fetch_log_index` and `download_logs` functions for interacting with public log archives.
-

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"]  
}
```