WebPorting.md

Of course. Here is the comprehensive roadmap, now including a discussion on porting animations, presented as a single document.

A Phased Roadmap to the Web

This plan allows you to maintain and use the existing command-line tool while incrementally building the new web application.

Phase 1: Local Environment Setup

The goal here is to establish a foundation where the new web project can coexist with and use the existing analysis engine.

1. **Enhance Conda Environment:** Add Django to your existing `cla` conda environment (`pip install django`).
2. **Integrate Django Project:** Inside your `Contest-Log-Analyzer` directory, create a new Django project (`django-admin startproject web_project .`). This places the Django management files alongside your existing `contest_tools` library.
3. **Create First App:** Create a dedicated Django "app" to handle the reports (`python manage.py startapp reports_app`).
4. **Verify:** Run the Django development server (`python manage.py runserver`) and confirm you can see the default welcome page in your browser.

Phase 2: Core Logic Integration & Prototyping

This phase proves that the web front-end can correctly trigger the existing Python back-end.

1. **Build a Django View & Template:** Create a simple webpage with a file upload form.
2. **Connect the Wires:** Write the Python code in your Django view to receive an uploaded log file. This view will then call the familiar `LogManager` and `ReportGenerator` classes from the `contest_tools` package to process the data, just like `main_cli.py` does.
3. **Display a Text Report:** For the first test, simply take the string output from a text-based report (like `summary`) and display it on a results page. This confirms the end-to-end data flow is working.

Phase 3: Transition to Web-Native Outputs

Now, we'll convert the reports from static files to interactive web components.

1. **Convert Static Reports to Plotly:** Go through each visual report (`plot_*` and `chart_*`) and replace the Matplotlib code with its Plotly equivalent. The Python code will now generate a JSON object describing the chart using `fig.to_json()`.
2. **Port the Animation Report:** The animation requires a two-pronged approach to provide both an interactive web view and a downloadable file, while still using a single Plotly codebase:
    - **Interactive Web Version:** We will create a true interactive Plotly animation. The Django view will prepare the data for *all frames* of the animation and pass it as a single JSON object to the template. Plotly.js can then render this as a figure with a play/pause button and a timeline slider, allowing the user to scrub through the contest.
    - **Downloadable Video (MP4):** To create the downloadable video, we'll mimic the current `imageio` process. A background task will loop through each hour of the contest, generate a Plotly figure for that single frame, and use `kaleido` to export it as a static PNG file (`fig.write_image()`). Once all frames are saved to a temporary directory, `imageio` will compile them into an MP4 file, which can then be offered as a download link.
3. **Implement Client-Side Rendering:** Update the Django templates to include the Plotly.js library. Write the small JavaScript snippet to take the JSON from your view and render the interactive charts and animations in the user's browser.
4. **Style the UI:** Use **Tailwind CSS** in your templates to create a professional and responsive layout for your report pages, tables, and navigation.

Phase 4: Production & Deployment

This phase prepares the application to run on a commercial hosting provider.

1. **Select a Production Web Server:** Replace Django's development server with a production-grade server like **Gunicorn**.
2. **Configure a Database:** Switch from the simple `sqlite3` development database to a robust production database like **PostgreSQL**.
3. **Manage Static Files:** Configure a tool like **WhiteNoise** to allow Gunicorn to serve your static files (CSS, JavaScript) efficiently.
4. **Deploy:** Push your code to a commercial hosting provider (like DigitalOcean, Heroku, or PythonAnywhere) and configure it to run your application.

What's Missing? Key Considerations

Your request covered the core technical migration, but a successful commercial service requires a few additional architectural components that we should plan for.

User Accounts & Data Persistence

A commercial service implies users will want to sign up, log in, and save their logs.

- **Authentication:** We'll need to use Django's built-in system for user registration, login, and password management.
- **Database Models:** We must design a database schema to store user profiles and associate uploaded logs with specific users, allowing them to view their past reports.

Asynchronous Task Processing

Some reports on very large logs can take a long time to generate. A web request should never be blocked for minutes waiting for a report to finish.

- **Task Queue:** We need to introduce a task queue system using **Celery** with **Redis** as a broker.
- **Workflow:** When a user requests a report, the Django view will instantly add a "job" to the Celery queue and return a message like "Your report is being processed." A separate "worker" process will pick up the job, run the time-consuming analysis, and save the result. The user can then be notified or see the finished report on their dashboard.

Testing & Security

While `main_cli.py` is great for debugging, a production web application needs a formal testing and security strategy.

- **Automated Testing:** We should create a suite of **unit tests** for our `contest_tools` logic and Django views to ensure they behave as expected and prevent regressions.
- **Security:** We must follow web security best practices, such as using environment variables for secret keys and database passwords, and leveraging Django's built-in protections against common attacks (like CSRF and XSS).