Phase 0: System Setup & Configuration

Initialize Environment:
Create a Python environment.
Install necessary libraries: pandas, numpy, astropy, astroquery, gwosc, scikit-learn (or xgboost/lightgbm).
Define Configuration Parameters:
Create a configuration file or a set of global variables. This allows you to easily tweak the system's behavior.
DATA_SOURCES: ['gwosc', 'heasarc_grb', 'ztf_transients']
DATE_RANGE: ['YYYY-MM-DD', 'YYYY-MM-DD']
MAX_TIME_WINDOW_DAYS: 1.0 (for initial filtering)
MAX_ANGULAR_DISTANCE_DEG: 15.0 (for initial filtering)
ML_MODEL_PATH: './models/correlation_model.pkl'

Phase 1: Data Ingestion & Standardization
The goal of this phase is to fetch raw data and transform it into a single, standardized format.

Define the Standard Event Schema:
Create a standard structure for your data. The ultimate goal is a pandas DataFrame with these columns:

- **event_id**: Unique ID from the source (e.g., 'GW170817').
- **source**: The origin catalog (e.g., 'GWOSC').
- **event_type**: (e.g., 'GW', 'GRB', 'Optical').
- **utc_time**: The event timestamp in a standard Astropy Time object or Pandas datetime.
- **ra_deg**: Right Ascension in decimal degrees.
- **dec_deg**: Declination in decimal degrees.
- **pos_error_deg**: The radius of the position uncertainty in degrees. (This is crucial!).
- **signal_strength**: A normalized value representing event significance (e.g., SNR for GW, magnitude for optical).

Implement Data Fetcher Functions:
Create a separate function for each data source.

```
def fetch_gwosc_events(date_range):
    # Use the gwosc library to get a list of events.
    # For each event, extract the required fields.
    # Convert GPS time to UTC using astropy.time.Time.
    # For pos_error_deg, estimate the radius of the 90% confidence sky map.
    # Return a standardized DataFrame.

def fetch_heasarc_events(date_range):
    # Use astroquery.heasarc to query for GRB events.
    # Extract fields and standardize them into the DataFrame schema.
    # (Repeat for other sources like ZTF/TNS)
```

Combine and Pre-process Data:

- Run all fetcher functions.
- Concatenate the resulting DataFrames into a single master event list.
- Crucially, sort the master DataFrame by utc_time. This is essential for the next phase's efficiency.
- Clean the data: handle missing values (e.g., fill missing pos_error_deg with a default large value).

Phase 3: Feature Engineering for AI
For each candidate pair, we create a rich "feature vector" for the ML model.

- Initialize `feature_vectors` list.
- Iterate through `candidate_pairs`:
    - For each pair `(event_A, event_B)`:
        - Create a feature dictionary:
            - `'time_delta_sec'`: Time difference in seconds.
            - `'angular_dist_deg'`: Angular distance in degrees.

        - `'pos_error_A', 'pos_error_B'`: The positional errors of each event.
        - `'significance_of_match'`: `angular_dist_deg / (pos_error_A + pos_error_B)`. (A powerful feature — lower values mean stronger spatial correlation).
        - `'signal_strength_A', 'signal_strength_B'`: The normalized signal strengths.
        - One-Hot Encode the event types (e.g., `'is_GW-GRB'`, `'is_GW-Optical'`, etc.).
        - *(Optional Advanced Feature)* Query SIMBAD at the event location to add `'galaxy_nearby'` (0 or 1).
    - Append the feature dictionary to `feature_vectors`.

Manually label your dataset as follows:

- Review the candidate pairs.
- Mark the real astrophysical associations as **`target=1`**.
- Mark the random overlaps as **`target=0`**.

This gives you a labeled training dataset.

Phase 4: ML Scoring & Ranking
This is where the "intelligence" comes in. We use a pre-trained model to score the candidates.

Offline Task (Model Training):

- First, train a model (e.g., XGBoost Classifier) on a labeled dataset of known positive and negative associations (from Phase 3).
- Save the trained model as a `.pkl` file.

Steps:

Load the Pre-trained Model

```
 from joblib import load
model = load(ML_MODEL_PATH)
```

1. **Predict Probabilities**
    - Convert `feature_vectors` into a 2D array (DataFrame or NumPy array).
2. **Append Scores**
    - Add `confidence_score = probs[i]` to each candidate pair's record.
    - Now each candidate pair has features + predicted association probability.

Phase 5: Output & Presentation
The final step is to present the results to the user in a clear and actionable way.

Steps:

1. **Sort the Results**
    - Order candidate pairs by `confidence_score` (highest first).
2. **Generate Final Output**
    - Build a DataFrame / JSON with:
        - `event_A_id`, `event_A_type`
        - `event_B_id`, `event_B_type`
        - `confidence_score`
        - `time_delta_sec`, `angular_dist_deg`

3. **Display Results**
    - Show the sorted list in a clean table (web dashboard or notebook).
4. **Visualization**
    - For top candidates, plot RA/Dec with uncertainty circles on an interactive sky map (Plotly, D3.js, or Aladin Lite).
    - This helps users visually confirm correlations.

After Data

Phase 1: The Dynamic Configuration Engine
"Configuration Engine." - This will be a class or module that provides search parameters tailored to the specific type of event.

Input:

- An `event_type` string (e.g., 'GW', 'GRB', 'Optical').

Output:

- A configuration object with the following adaptive parameters:
    - `time_window_days`: The temporal search radius.
    - `max_search_radius_deg`: The maximum angular search radius.
    - `use_adaptive_radius`: A boolean. If `True`, the search radius will be a multiple of the event's `pos_error_deg`, capped by `max_search_radius_deg`.

Detailed Implementation: The engine will contain a dictionary mapping event types to their specific configurations.

```
EVENT_CONFIG = {
    'GW': {
        'time_window_days': 2.0,  # Kilonovae can brighten over a couple of days
        'max_search_radius_deg': 90.0, # Some GW events are poorly localized
        'use_adaptive_radius': True
    },
    'GRB': {
        'time_window_days': 0.5,  # Afterglows are typically prompt
        'max_search_radius_deg': 2.0,
        'use_adaptive_radius': False # GRBs are usually well-localized
    },
    'DEFAULT': {
        'time_window_days': 1.0,
        'max_search_radius_deg': 10.0,
        'use_adaptive_radius': True
    }
```

Phase 2: Standardized Data Ingestion
We will implement robust fetcher functions for each data source (GWOSC, HEASARC, ZTF, etc.). The key is to transform the disparate outputs from these APIs into a single, standardized pandas DataFrame.

Input:

- A date range.
- A list of data sources.

Output:

- A single, time-sorted pandas DataFrame with the standardized schema you defined, including `event_id`, `source`, `event_type`, `utc_time`, `ra_deg`, `dec_deg`, `pos_error_deg`, and `signal_strength`.

Phase 3: Adaptive Candidate Search

Here we implement the time-windowed spatial search, but instead of using fixed parameters, we'll query our Phase 1 Configuration Engine for every event.

Input:

- The time-sorted DataFrame from Phase 2.
- The Configuration Engine from Phase 1.

Output:

- A list of `candidate_pairs`, where each pair is a tuple of two event records.

Detailed Algorithm:

1. Initialize an empty list, `candidate_pairs`.
2. Loop through each `event_A` in the master DataFrame.
3. Fetch the adaptive configuration: `config = ConfigurationEngine.get(event_A.event_type)`.
4. Determine the search radius:
    - If `config.use_adaptive_radius` is `True`, set `search_radius = min(3 * event_A.pos_error_deg, config.max_search_radius_deg)`. (Using a 3-sigma multiple is a common practice).
    - Else, `search_radius = config.max_search_radius_deg`.
5. Start an inner loop for subsequent events (`event_B`).
6. Calculate `time_delta = event_B.utc_time - event_A.utc_time`.

7. If `time_delta > config.time_window_days`, break the inner loop.
8. If the time is within the window, calculate `angular_dist`.
9. If `angular_dist < search_radius`, add (`event_A`, `event_B`) to `candidate_pairs`.

Phase 4: Multi-Factor Scoring Engine
We will calculate a final score by combining several independent, statistically-motivated sub-scores.

Phase 1: The Dynamic Configuration Engine

Expert Solution

Instead of a simple configuration file with static values, we'll design a "Configuration Engine." This will be a class or module that provides search parameters tailored to the specific type of event we are investigating. This directly solves the "one-size-fits-all" problem.

Input:

- An `event_type` string (e.g., 'GW', 'GRB', 'Optical').

Output:

- A configuration object with the following adaptive parameters:
  - `time_window_days`: The temporal search radius.
  - `max_search_radius_deg`: The maximum angular search radius.
  - `use_adaptive_radius`: A boolean. If `True`, the search radius will be a multiple of the event's `pos_error_deg`, capped by `max_search_radius_deg`.

Detailed Implementation: The engine will contain a dictionary mapping event types to their specific configurations.

Python
```
# Example Configuration Dictionary
EVENT_CONFIG = {
    'GW': {
        'time_window_days': 2.0,  # Kilonovae can brighten over a couple of days
        'max_search_radius_deg': 90.0, # Some GW events are poorly localized
        'use_adaptive_radius': True
    },
    'GRB': {
```

```
        'time_window_days': 0.5,  # Afterglows are typically prompt
        'max_search_radius_deg': 2.0,
        'use_adaptive_radius': False # GRBs are usually well-localized
    },
    'DEFAULT': {
        'time_window_days': 1.0,
        'max_search_radius_deg': 10.0,
        'use_adaptive_radius': True
    }
}
```

Critic's Evaluation

- **Impactfulness & Effectiveness:** This approach is a **massive improvement** over a static configuration. It injects crucial domain knowledge directly into the search algorithm, making the candidate selection process far more efficient and physically motivated. It will dramatically reduce the number of random, nonsensical pairs the system has to evaluate later.
- **Weakness:** The effectiveness of this entire system now hinges on the quality of these "magic numbers" in the configuration. Where do they come from? They are based on current astrophysical models, but what if a new type of event defies these models? The system is flexible, but it's still biased by our current understanding and requires an expert to tune it. It is not a self-learning system.

Phase 2: Standardized Data Ingestion

Expert Solution

This phase remains largely the same as the initial proposal, as its logic is sound. We will implement robust fetcher functions for each data source (GWOSC, HEASARC, ZTF, etc.). The key is to transform the disparate outputs from these APIs into a single, standardized pandas DataFrame.

Input:

- A date range.
- A list of data sources.

Output:

- A single, time-sorted pandas DataFrame with the standardized schema you defined, including

`event_id`, `source`, `event_type`, `utc_time`, `ra_deg`, `dec_deg`, `pos_error_deg`, and `signal_strength`.

Critic's Evaluation

- **Impactfulness & Effectiveness:** This is a necessary utility phase. Its impact is binary: if it works, the system functions; if it fails, nothing else matters. The logic is standard practice and correct.
- **Weakness:** The system is vulnerable to external API changes, rate limits, or downtime. A production-grade system would need robust error handling, retry logic, and logging for each data fetcher. Furthermore, normalizing `signal_strength` across different event types (e.g., comparing a GW's SNR to an optical transient's magnitude) is non-trivial and requires careful, physically-justified scaling functions. This proposal hasn't specified *how* that normalization will be done.

Phase 3: Adaptive Candidate Search

Expert Solution 🔭

Here we implement the time-windowed spatial search, but instead of using fixed parameters, we'll query our Phase 1 Configuration Engine for every event.

Input:

- The time-sorted DataFrame from Phase 2.
- The Configuration Engine from Phase 1.

Output:

- A list of `candidate_pairs`, where each pair is a tuple of two event records.

Detailed Algorithm:

1. Initialize an empty list, `candidate_pairs`.
2. Loop through each `event_A` in the master DataFrame.
3. Fetch the adaptive configuration: `config = ConfigurationEngine.get(event_A.event_type)`.
4. Determine the search radius:
    - If `config.use_adaptive_radius` is `True`, set `search_radius = min(3 * event_A.pos_error_deg, config.max_search_radius_deg)`. (Using a 3-sigma multiple is a common practice).

    - Else, `search_radius = config.max_search_radius_deg`.
5. Start an inner loop for subsequent events (`event_B`).
6. Calculate `time_delta = event_B.utc_time - event_A.utc_time`.
7. If `time_delta > config.time_window_days`, **break** the inner loop.
8. If the time is within the window, calculate `angular_dist`.
9. If `angular_dist < search_radius`, add (`event_A`, `event_B`) to `candidate_pairs`.

Critic's Evaluation

- **Impactfulness & Effectiveness:** This is the core algorithmic enhancement. The impact is a huge reduction in computational waste and a massive improvement in the signal-to-noise ratio of our candidate list. We are no longer looking for a needle in a haystack; we are looking for a needle in a small, well-defined patch of hay.
- **Weakness:** The algorithm is fundamentally pairwise. It excels at finding `A+B` correlations. It is not designed to find higher-order correlations, such as a triplet `A+B+C` (e.g., a GW event followed by a GRB and a high-energy neutrino from the same location). Finding such clusters would require a more complex graph-based or clustering approach after this initial pairwise filtering.

Phase 4: Multi-Factor Scoring Engine

Expert Solution

This engine replaces the flawed AI model. We will calculate a final score by combining several independent, statistically-motivated sub-scores.

Input:

- A single `candidate_pair` from Phase 3.

Output:

- The same pair, enriched with the following new data fields:
    - `score_spatiotemporal`: A score from 0 to 1 representing the spatio-temporal alignment.
    - `score_significance`: A score from 0 to 1 representing the combined importance of the events.
    - `score_contextual`: A score from 0 to 1 based on astrophysical context (e.g., presence of a galaxy).

- `confidence_score`: The final weighted combination of the sub-scores.

Detailed Implementation:

1. **Spatio-Temporal Score:**
   - Calculate `S_match = angular_dist_deg / (event_A.pos_error_deg + event_B.pos_error_deg)`.
   - Use an exponential decay function to map this to a score. `score_spatial = exp(-S_match)`. This heavily rewards values of `S_match` close to zero.
   - Do the same for time: `score_temporal = exp(-time_delta_sec / T)`, where `T` is a characteristic timescale (e.g., 3600 seconds).
   - `score_spatiotemporal = score_spatial * score_temporal`.
2. **Significance Score:**
   - Normalize the `signal_strength` for each event to a 0-1 range based on its type (e.g., map SNR/magnitude to a percentile rank within its own event class). Let these be `norm_sig_A` and `norm_sig_B`.
   - `score_significance = norm_sig_A * norm_sig_B`.
3. **Contextual Score:**
   - For the pair's sky position, query an astronomical catalog like SIMBAD or VizieR for known galaxies within the error circle.
   - `score_contextual = 1.0` if a plausible host galaxy is found, `0.1` otherwise (a low non-zero value acknowledges the possibility of discovering events in intergalactic space).
4. **Final Confidence Score:**
   - Define weights: `W_st = 0.5`, `W_sig = 0.3`, `W_ctx = 0.2`.
   - `confidence_score = (W_st * score_spatiotemporal) + (W_sig * score_significance) + (W_ctx * score_contextual)`.

Multi-Factor Scoring Engine. This is where you win or lose.

- Implement the sub-scores: `score_spatiotemporal`, `score_significance`, and `score_contextual`.
- **Use the Percentile Ranking system** for `score_significance`. It's a powerful and correct way to normalize.
- **Use hard-coded, expert weights.** Don't try to learn them. Start with `W_st = 0.5, W_sig = 0.3, W_ctx = 0.2` and justify why you chose them (spatio-temporal alignment is the most important evidence, etc.). This is transparent and defensible.

Instead of AI/ML which is black-box we decided to go with the white-box physics based engine. This engine calculates a final `confidence_score` by combining three independent sub-scores, each representing a different aspect of the evidence.

Sub-score 1: The Spatio-Temporal Score (`score_spatiotemporal`)

Goal: To produce a single number (from 0 to 1) that answers: "How perfect was the alignment in space and time, considering the uncertainties involved?"

This score is itself a combination of two parts: a spatial score and a temporal score.

1. Spatial Component:

- First, we calculate the **Significance of Match**, or `S_match`, using the formula from your plan: `S_match = angular_dist_deg / (pos_error_A + pos_error_B)`. This is the most critical calculation. It tells us how many "error bars" apart the two events are. A low `S_match` (e.g., < 1) is excellent.
- Next, we convert `S_match` into a score using an exponential decay function: `score_spatial = exp(-S_match)`. This function heavily rewards strong matches:
  - If `S_match` is 0 (a perfect overlap), `score_spatial` is 1.0.
  - If `S_match` is 1 (the events are one "error bar" apart), `score_spatial` is ~0.37.
  - If `S_match` is large, the score rapidly approaches 0.

2. Temporal Component:

- We use a similar exponential decay for time: `score_temporal = exp(-time_delta_sec / T)`, where `T` is a characteristic timescale. For a hackathon, you can set `T = 3600` (one hour in seconds).
- This means time differences much smaller than an hour receive a high score, while those of several hours or days receive a score close to zero.

Final `score_spatiotemporal`:

- We combine these by multiplying them: `score_spatiotemporal = score_spatial * score_temporal`. By multiplying, we enforce that **both** the spatial and temporal alignments must be good for the score to be high. A perfect spatial match that happened three days late is useless.

Sub-score 2: The Significance Score (`score_significance`)

Goal: To answer the question: "Were these two events loud, bright, and clear, or were they faint whispers that could be noise?" A correlation between two powerful events is more believable.

This is where the **Percentile Ranking system** comes in.

1. **The Problem:** You can't directly compare the `signal_strength` of a gravitational wave (measured in SNR) to that of an optical flash (measured in magnitude).
2. **The Solution (Offline Step):** Before you run your main analysis, you process your entire dataset for each event type.
    - For all the GW events, you calculate the percentile rank of their `signal_strength` (SNR). The loudest event gets a rank of ~1.0, the median event gets 0.5, and the faintest gets ~0.0.
    - You do the same for all the GRB events, and all the optical events, etc..
3. **The Calculation:** When scoring a candidate pair, you simply look up their pre-calculated percentile ranks. Let's call them `norm_sig_A` and `norm_sig_B`.
4. **Final `score_significance`:** You multiply them: `score_significance = norm_sig_A * norm_sig_B`. This gives a single, normalized score representing the joint significance of the pair.

Sub-score 3: The Contextual Score (`score_contextual`)

Goal: To answer the question: "Did this event happen in a plausible location?" Since most powerful cosmic events happen in galaxies, finding a candidate within a known galaxy boosts our confidence.

1. **The Query:** For the candidate pair's sky position, use a library like `astroquery` to query an astronomical database (like SIMBAD) for known objects at that location.
2. **The Logic:** Check if any of the returned objects are classified as a "Galaxy."
3. **Final `score_contextual`:** Assign a score based on the result:
    - `score_contextual = 1.0` if a galaxy is found in the error circle.
    - `score_contextual = 0.1` if no galaxy is found. We use 0.1 instead of 0 to acknowledge the small possibility of discovering a new phenomenon (like an event in intergalactic space), so we penalize it but don't completely discard it.

Combining Them: Justifying the Expert Weights

Finally, you combine these three independent scores into a single `confidence_score` using a weighted average:

```
confidence_score = (0.5 * score_spatiotemporal) + (0.3 *
score_significance) + (0.2 * score_contextual)
```

For a hackathon, you must justify these weights. Here is your defense:

- **Spatio-Temporal (`W_st = 0.5`):** This receives the **highest weight** because a tight coincidence in both space and time is the **primary, defining evidence** of a multi-messenger event. If the alignment is poor, the correlation is meaningless, regardless of other factors.
- **Significance (`W_sig = 0.3`):** This is the **secondary evidence**. It answers "how good was the data?" A strong alignment between two high-quality, powerful signals is far more convincing than one between two faint signals close to the noise floor. This score modulates our belief in the primary evidence.
- **Contextual (`W_ctx = 0.2`):** This is **supporting evidence**. Finding the event in a galaxy makes it astrophysically plausible and consistent with our current understanding. It's a valuable reality check and tie-breaker but is weighted the lowest because the raw event data (the first two scores) is more fundamental. We don't want to overly penalize a potential groundbreaking discovery that defies our current models by happening outside a galaxy.