Detecting Coordinated Harassment: A Step-by-Step Methodology

Introduction: The Challenge of "Pile-On" Harassment

This project addresses the complex problem of detecting **group-level cyberbullying**, a type of online harm where a target is swamped by a high volume of similar, toxic messages in a short time—an event often called a "pile-on" or "raid." For platforms that need to moderate in real time, this presents a significant challenge.

Common moderation methods often fall short when faced with this kind of coordinated activity:

- **Cost and Latency:** Analyzing every single message with powerful but slow large language models (LLMs) is too expensive and introduces too much latency for high-volume streams.
- **Lack of Group-Level Insight:** Basic tools evaluate messages in isolation. They don't directly produce a *set-level* risk score that captures the overall **density** and **concentration** of harm, which are the defining traits of a coordinated raid.

The goal of this project is to overcome these limitations by engineering a **fast discriminative monitoring loop**. This system uses a Reinforcement Learning (RL) policy to make intelligent, real-time alert decisions without relying on costly per-message analysis.

To build this real-time decision system, we can't work with raw text. The first and most critical task is to engineer a data pipeline that converts a chaotic stream of messages into a concise, machine-readable summary of group behavior.

1. The Data Pipeline: From Raw Messages to Actionable Signals

Before any alert decisions can be made, the raw message stream must be processed and transformed into a set of meaningful features. This pipeline is designed to be efficient and, most importantly, to capture group-level dynamics.

1.1. Simulating a Realistic Message Stream

To train and test the system, a realistic message stream containing harassment events was simulated using two key components:

1. **Base Dataset:** The `civil_comments` dataset provides a large volume of real-world text. Its `toxicity` score is used as a proxy for the ground truth label of whether a message is harmful.
2. **Harassment Simulation:** A function called `inject_raid(...)` simulates a coordinated attack by injecting a burst of "dense and semantically consistent" toxic

messages into the stream. This allows for controlled testing of the system's ability to detect these specific events.

1.2. Step 1: Message Encoding with MiniLM

The first step in the pipeline is to convert each raw text message into a numerical representation, or "embedding vector," that captures its semantic meaning. To do this efficiently, we use the `sentence-transformers/all-MiniLM-L6-v2` model, a lightweight and fast encoder perfect for processing high-volume streams.

1.3. Step 2: Group-Level Scoring with a Set Transformer

While embeddings give us a representation for each message, we still need a way to see the forest for the trees. How can we aggregate the signals from dozens of recent messages into a single, coherent picture of group activity? This is where the Set Transformer comes in.

This model is the core component for understanding group-level dynamics. Instead of analyzing messages one by one, it operates on a rolling buffer—a "context set"—of the embedding vectors from recent messages. From this set, it produces two critical output signals:

- `mean_hat`: An estimate of the **intensity** or average toxicity of the messages currently in the buffer.
- `conc_hat`: An estimate of the **concentration** or "burstiness" of the messages, indicating how similar and clustered they are.

1.4. Step 3: Constructing Features for the RL Agent

Finally, the signals from the Set Transformer are combined to construct a simple, low-dimensional state that the Reinforcement Learning agent can use to make a decision. The key here is to provide a compact yet informative summary of the current situation. The agent's observation is composed of four specific features:

- `rolling_mean` (The rolling average of the Set Transformer's intensity score)
- `rolling_conc` (The rolling average of the Set Transformer's concentration score)
- `ctx_density` (A measure of how many messages are in the current context window)
- `ctx_toxic_frac` (The fraction of messages in the context window flagged as toxic by the proxy score)

With this concise state vector, we have everything we need to pass control to our decision-making layer.

2. The Decision Layer: A Reinforcement Learning Agent

The features generated by the data pipeline are now fed into a Reinforcement Learning (RL) agent. This agent's job is to look at the current state of the message stream and decide whether an alert is warranted.

2.1. The Learning Environment

The agent learns within a custom-defined `AlertEnv`, a standard Gymnasium environment. This training ground has three core components:

- **Observation:** The 4-dimensional state constructed in the previous step: `[rolling_mean, rolling_conc, ctx_density, ctx_toxic_frac]`.
- **Action:** The simple, binary choice the agent can make at each step: `0 = no alert` or `1 = alert`.
- **Episode:** A single training run is defined as a full pass over the entire simulated stream of messages.

2.2. The Reward System: Defining a "Good" Decision

The agent learns to make good decisions by receiving rewards and penalties for its actions. The reward structure is designed to strongly encourage alerting on toxic messages (+1.0), penalize false alarms (-0.5), and apply a small penalty for missed toxic events (-0.2), creating a clear trade-off for the agent to learn.

Agent Action	Condition (based on proxy toxicity)	Reward
Alert (1)	`toxicity > 0.7` (True Positive)	**+1.0**
Alert (1)	`toxicity <= 0.7` (False Positive)	**-0.5**
No Alert (0)	`toxicity > 0.7` (False Negative)	**-0.2**
No Alert (0)	`toxicity <= 0.7` (True Negative)	**0.0**

2.3. The Learning Algorithm

The project uses the **Proximal Policy Optimization (PPO)** algorithm, a popular and robust RL algorithm from the Stable-Baselines3 library. The training progress table shows that the agent was successfully learning from this reward system; its average reward improved significantly from **-216** to **-118** over the course of training iterations.

This demonstrates that the agent can learn a policy, but the real test is how that learned policy stacks up against a much simpler, rule-based approach.

3. Evaluation: Did the RL Agent Outperform a Simple Rule?

To determine its real-world effectiveness, the fully trained RL policy was compared against a simple, hand-coded baseline rule.

3.1. The Naive Baseline

The baseline policy was a straightforward threshold rule designed to mimic a basic, non-learning system:

The "Naive baseline" alerts if `rolling_mean > 0.6` OR `rolling_conc > 0.5`.

3.2. Final Performance

Both the RL policy and the naive baseline were evaluated using the same reward function on the same test data stream. The results were telling.

Final Reward Comparison

Policy	Total reward
Naive threshold	-19.8
PPO (RL)	-19.8

The main conclusion from this specific run is clear: the trained PPO policy **did not** outperform the naive threshold baseline.

3.3. Key Insight: Why This Result is Still Valuable

This outcome provides a critical insight into a common pitfall when applying reinforcement learning: an agent can easily learn to "game" a single, predictable environment. Because the data stream and raid injection were identical in every training run, the PPO agent essentially memorized a sequence of actions that maximized its reward on that specific timeline. It showed learning progress, but it didn't learn a *generalizable* policy. The simple, robust logic of the naive baseline proved just as effective in this static scenario, highlighting the critical need for environmental variety in RL training.

To likely achieve superior performance, the methodology would need to be enhanced with several key improvements:

- **Data Variation:** Train on multiple randomized episodes, such as varying the location and timing of the simulated raids, to force the agent to learn a more robust strategy.
- **Smarter Rewards:** Implement more sophisticated reward shaping, such as adding penalties for "alert fatigue" (issuing too many alerts in a row), to teach the agent about the real-world costs of an overly sensitive system.

- **Richer Actions:** Expand the action space beyond a simple alert/no-alert decision to include options like "warn," "timeout," or "escalate."

4. Conclusion and Future Directions

This project demonstrates a powerful, four-step philosophy for real-time monitoring: start by **encoding** individual messages into a common language (embeddings), then **aggregate** them to understand group-level dynamics, **construct** a concise set of features from that aggregation, and finally, use an RL agent to learn a **decision** policy based on those features.

The current implementation has some key limitations, including its reliance on a **proxy ground truth** for toxicity and its simplified **binary alert decision**.

However, this framework serves as a strong foundation for future work. Potential expansions include:

- Building models that are aware of individual conversation threads for more precise context.
- Expanding the RL agent's action space to enable more nuanced moderation (e.g., warn, timeout, escalate).
- Incorporating explicit time-window features for more accurate density estimates.
- Developing capabilities to identify *who* is being targeted and the specific *type* of harm involved.