Data Engineering Interview Scenario

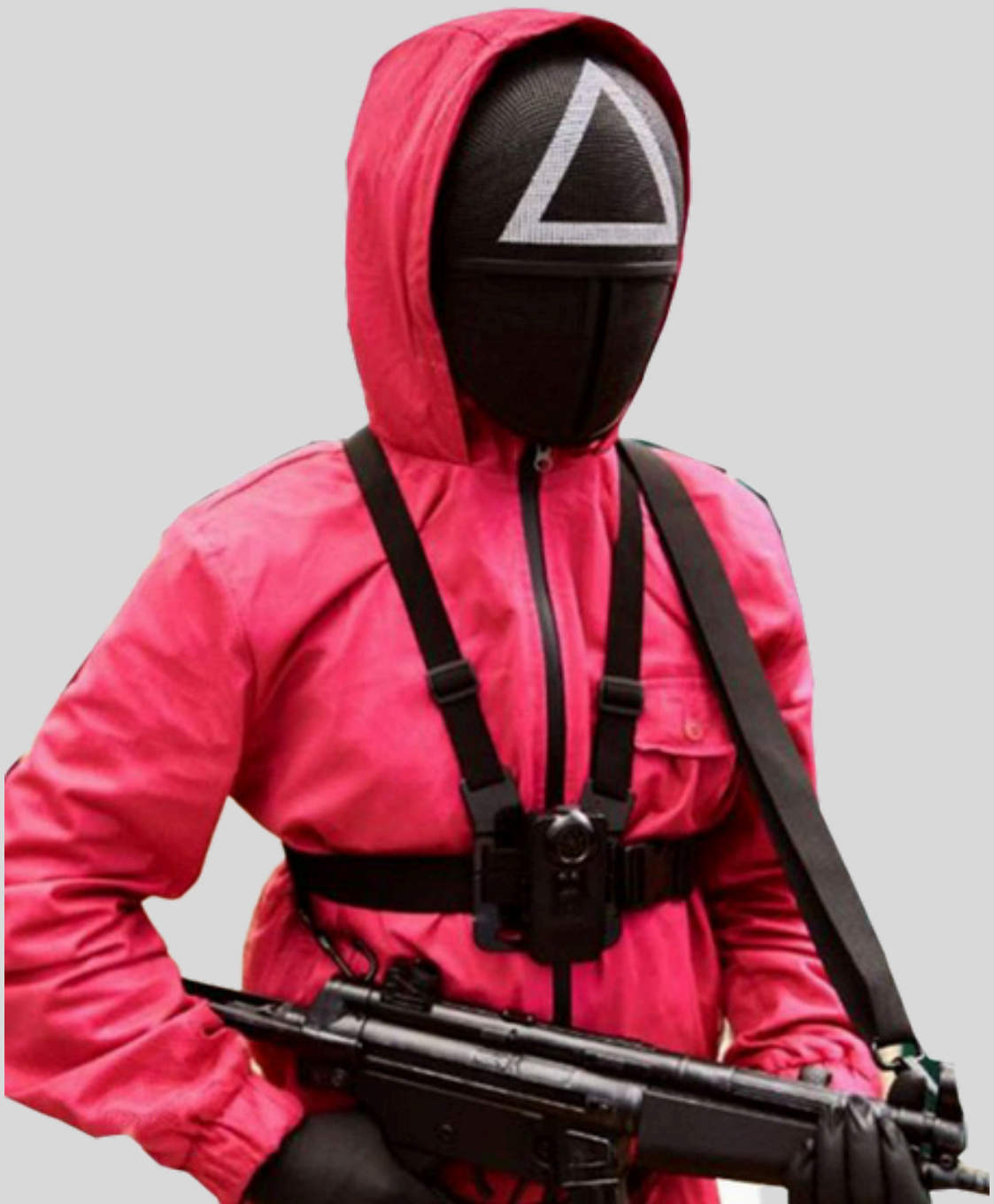
\

Prepared By

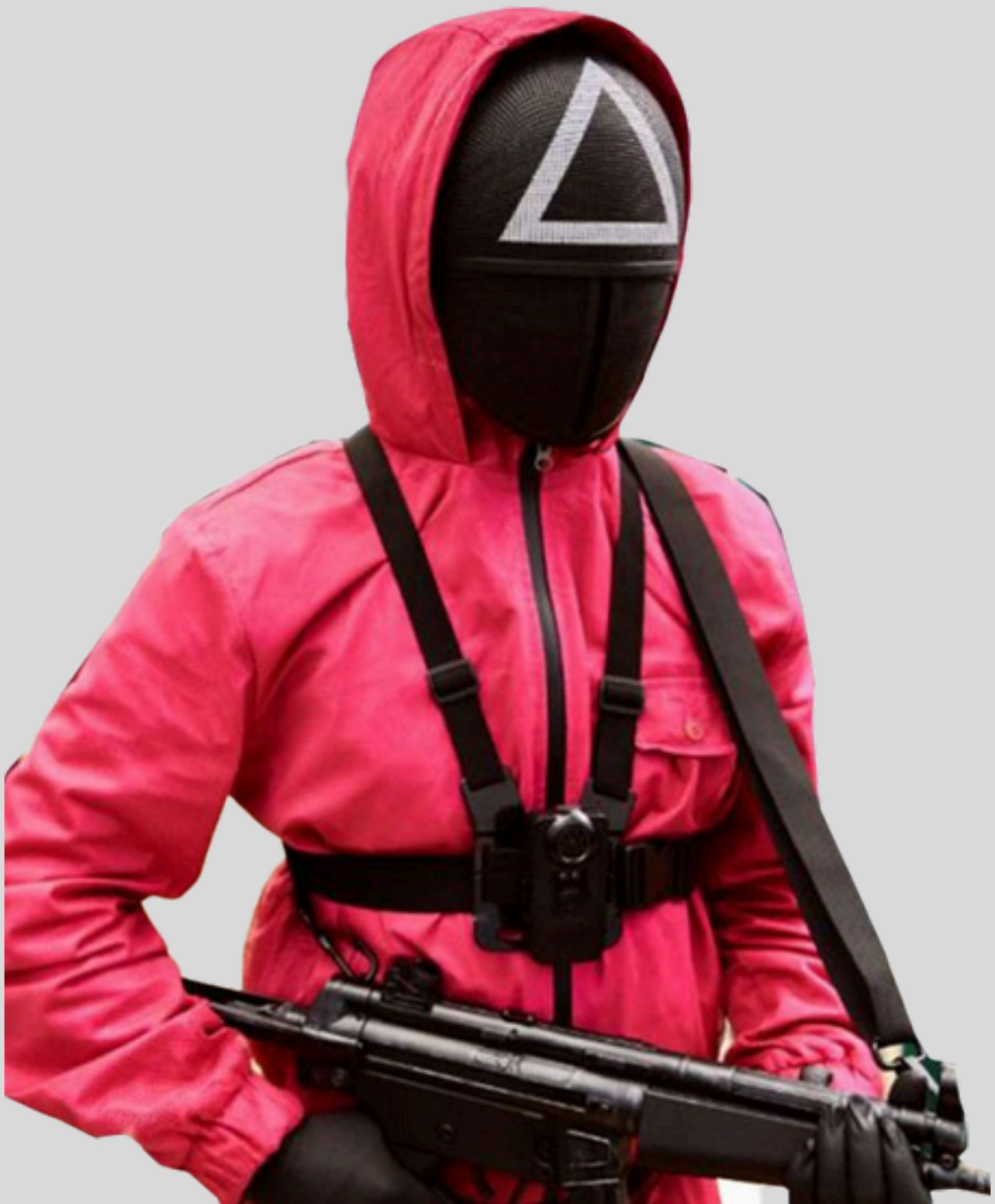
AFRIN AHAMED

Interviewer:

You are running a Spark job on a large dataset, and you notice that some tasks are taking significantly longer due to unbalanced partitions.



Question 1: What steps would you take to address this imbalance and improve performance?

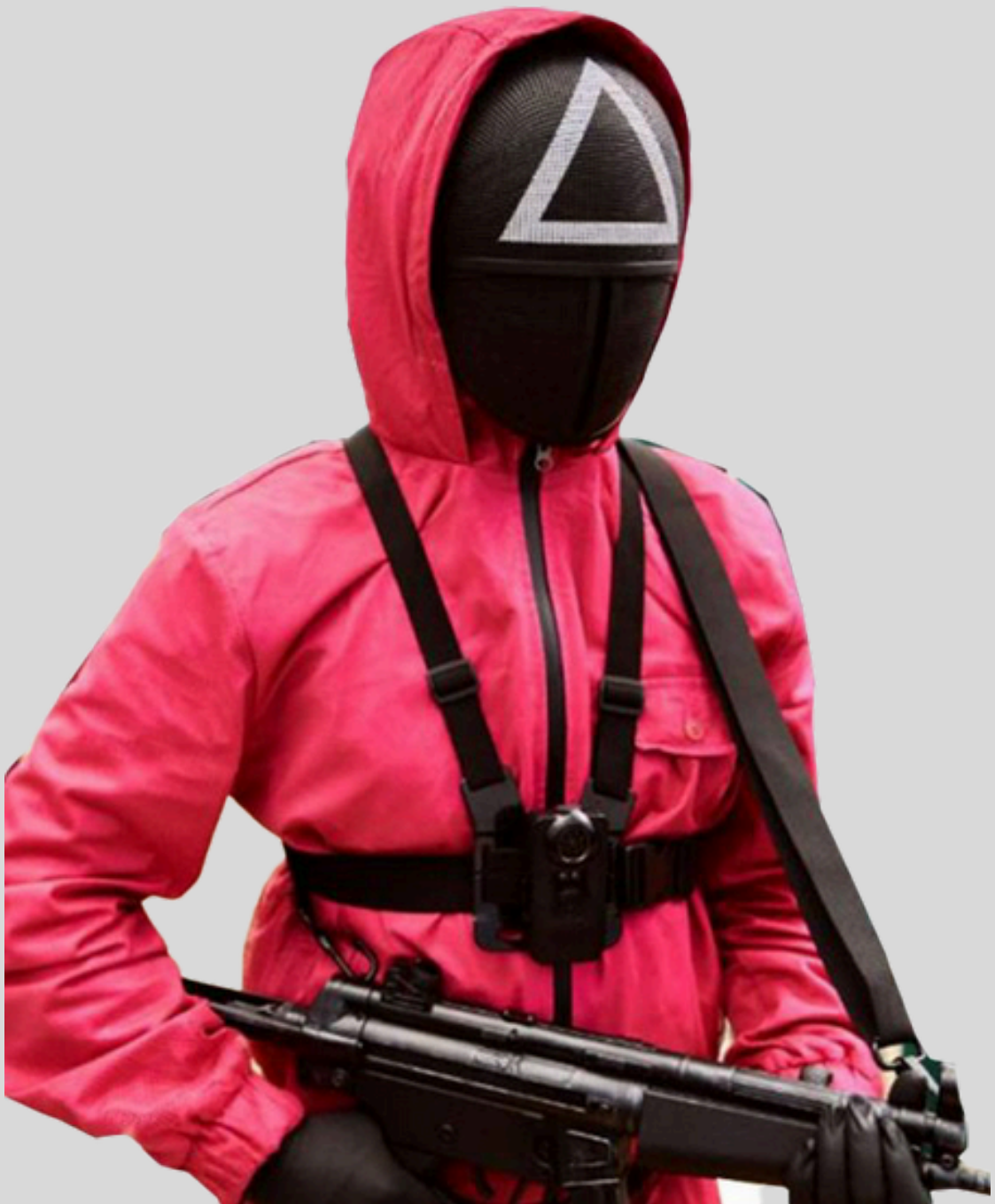


Candidate

:

- To address unbalanced partitions:
  - Check the "Stages" tab to identify tasks with longer execution times.
  - Look for partitions with significantly larger input sizes or higher shuffle read/write metrics.
  - Use repartition(n) to evenly distribute data across partitions.
  - If downstream stages don’t need excessive partitions, use coalesce(n) for efficient partitioning.
  - Apply techniques like salting (adding random prefixes to keys).
  - Redistribute heavy keys across multiple partitions.
  - Leverage broadcast joins for smaller datasets to avoid shuffles.
  - Enable Adaptive Query Execution (AQE):AQE adjusts partition sizes dynamically based on runtime statistics, balancing data distribution.

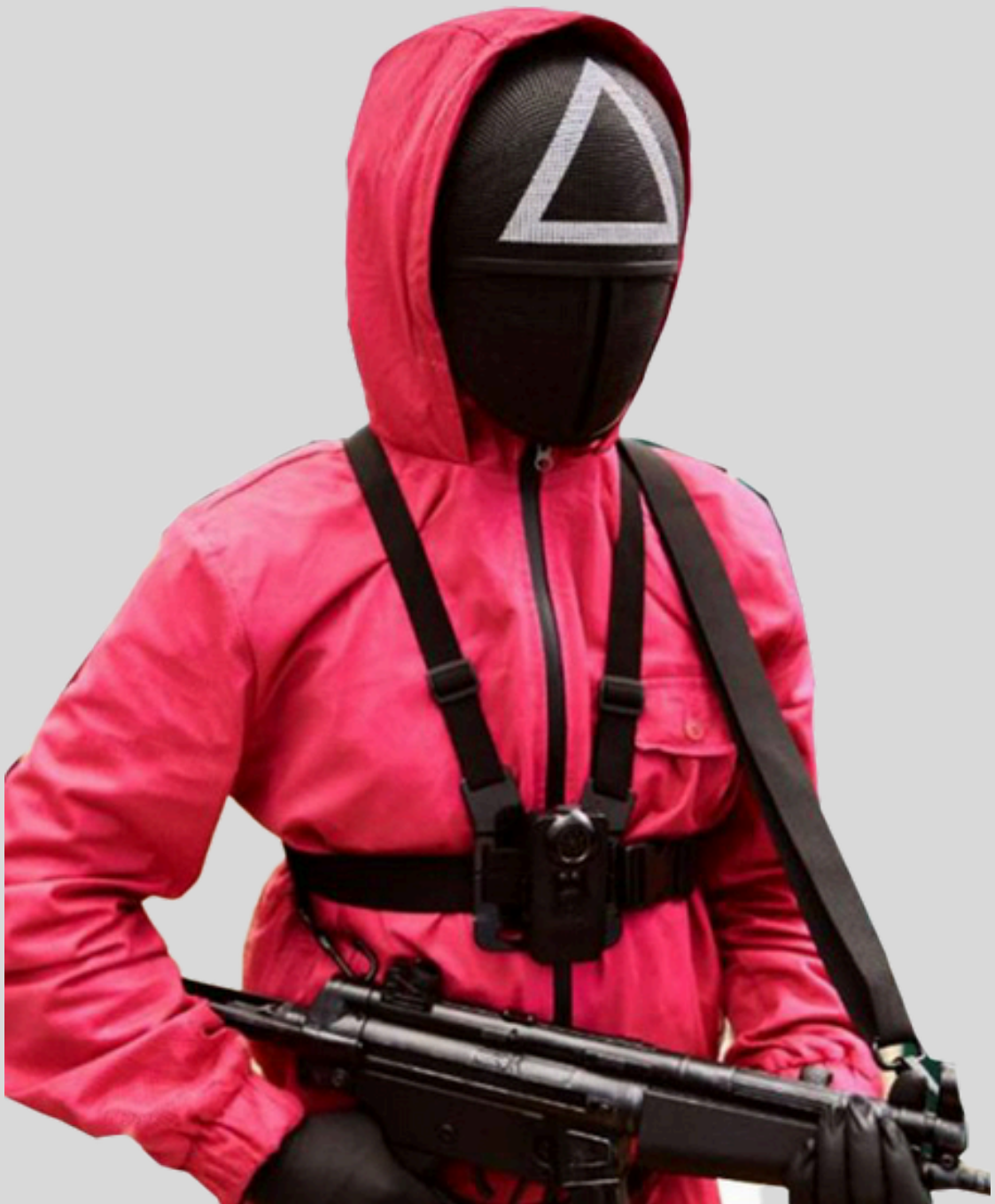
Question 2: How would you identify if the issue is caused by data skew?



Candidate:

- Spark UI Analysis:
  - Review task execution times in the "Stages" tab; tasks on skewed partitions will run significantly longer.
  - Check partition sizes in shuffle read/write metrics—skewed partitions will show disproportionately larger sizes.
- Partition Size Analysis:
  - Use rdd.mapPartitions(iter => Iterator(iter.size)).collect() to directly examine partition sizes.
- Profiling Keys:
  - Run data.groupByKey().count() to identify keys with an unusually large number of records.

Question 3: What are the advantages of using repartition() versus coalesce()?



Candidate:

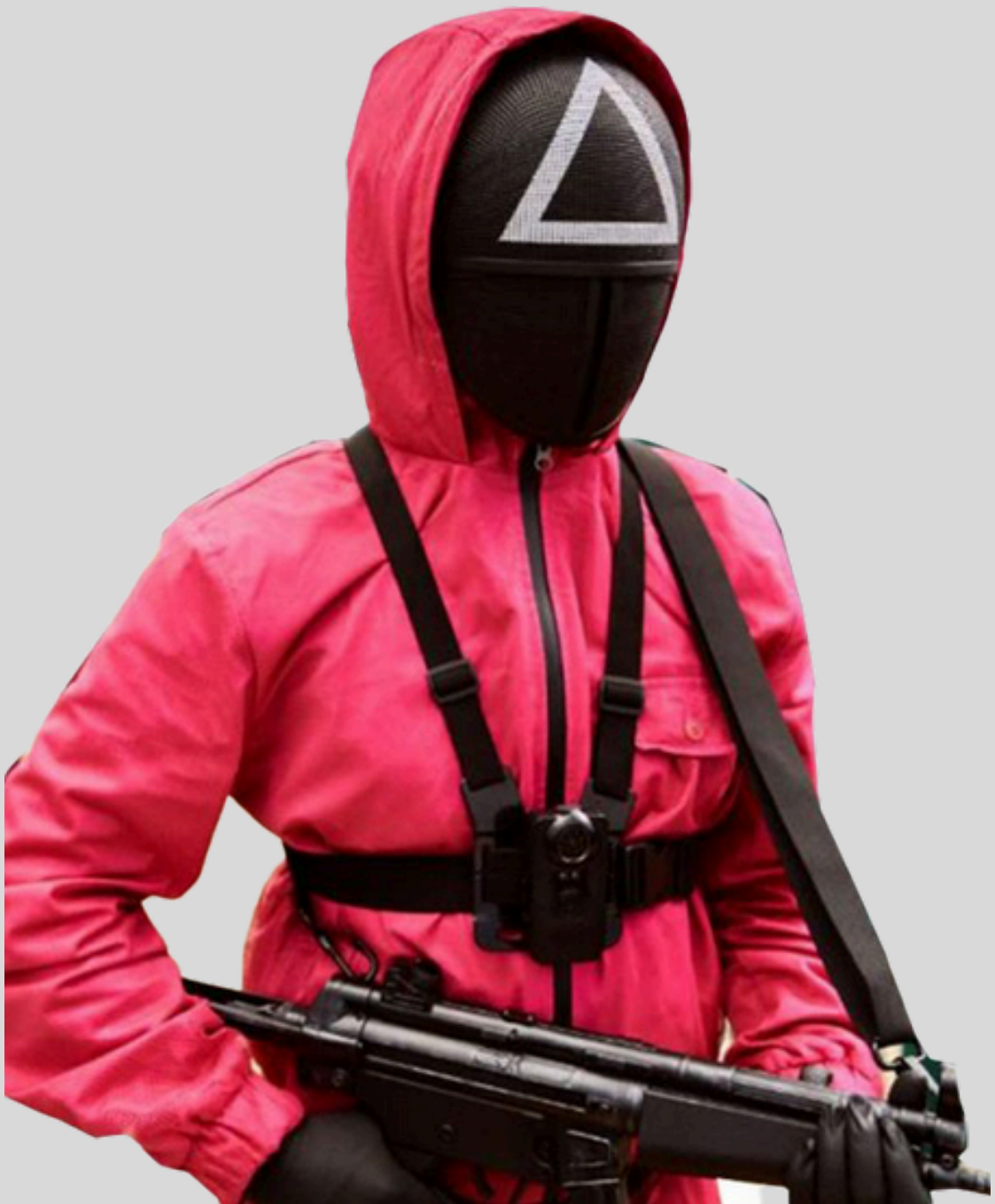
- 1.Repartition():

  Fully shuffles the data, ensuring an even distribution across all partitions.
  Suitable for increasing the number of partitions.

1.Coalesce():

Avoids full shuffling, making it faster but less evenly distributed.
Suitable for reducing the number of partitions.

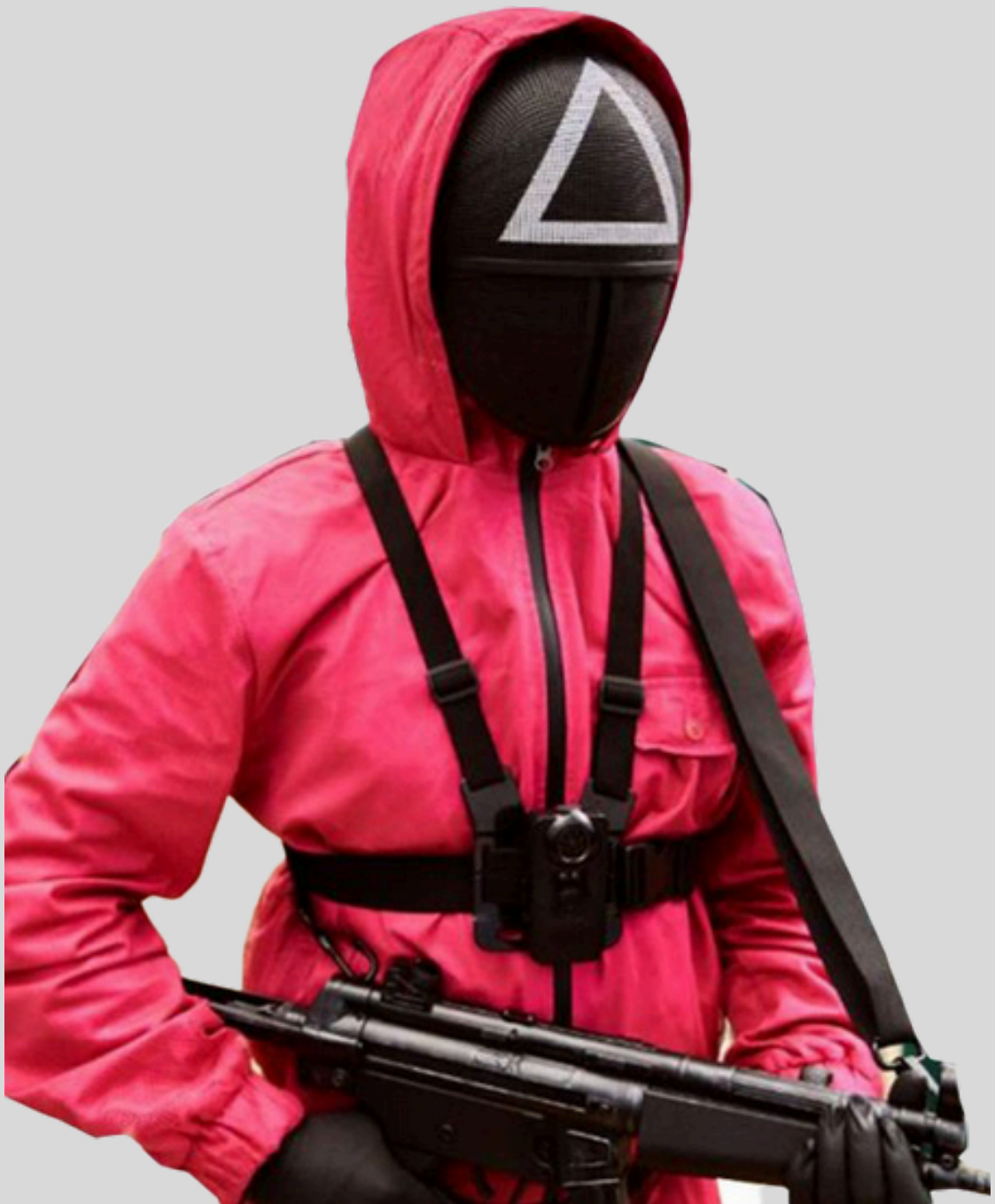
Question 4: If repartitioning doesn't resolve the issue, what alternative strategies would you consider?



Candidate:

- Data Salting:
  Add random prefixes to skewed keys to split them across multiple partitions.
- Custom Partitioners:
  Write a partitioner that redistributes data intelligently based on key distribution.
- Pre-Aggregation:
  Reduce data size by aggregating records before operations like joins or shuffles.

Question 5: What are the implications of too many partitions in a Spark job?



Candidate:

-
  - Task Overhead:

    Excessive partitions lead to an increased number of tasks, causing task scheduling overhead.

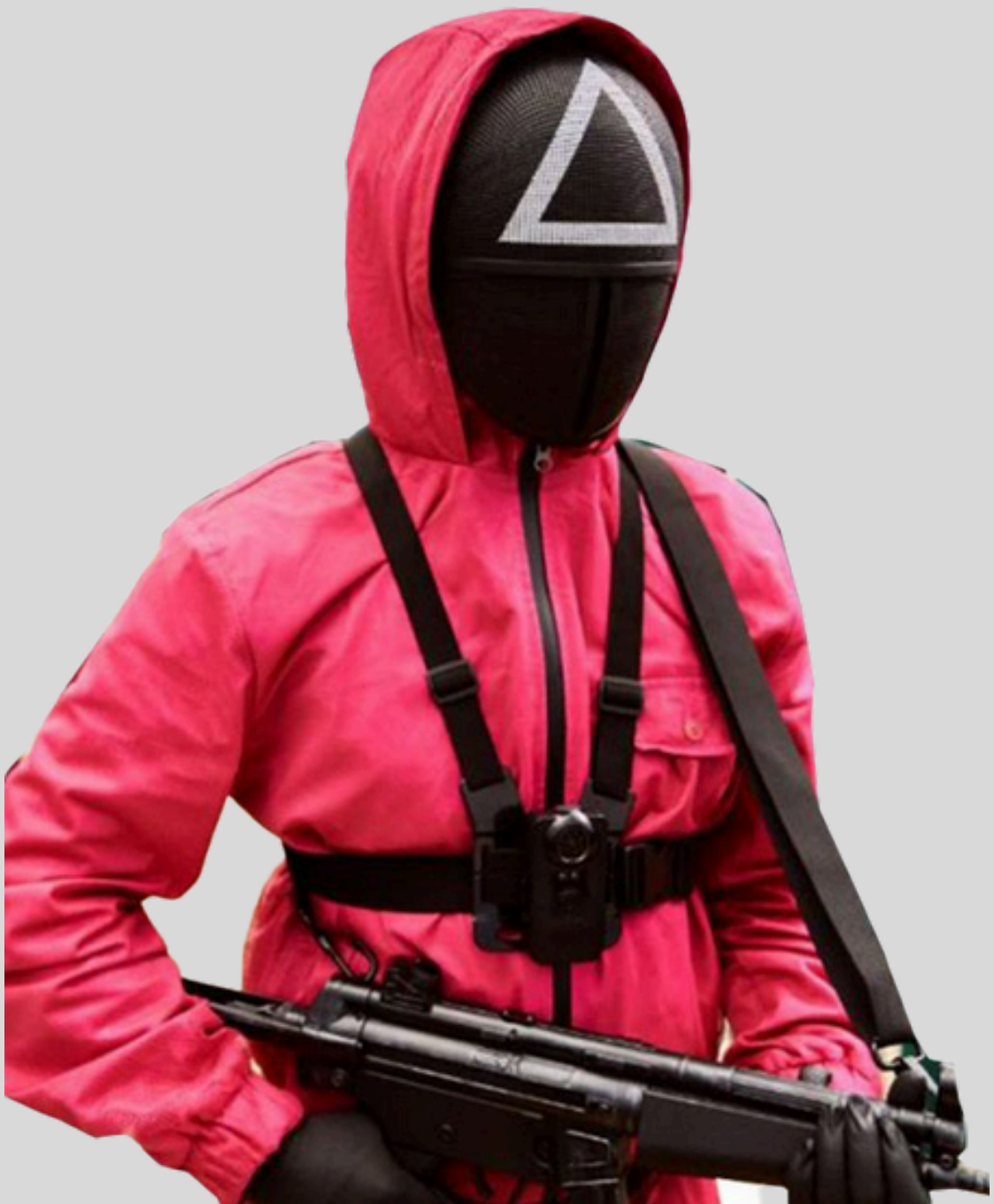
    Small File Problem:

    Writing results to storage may create many small files, impacting downstream processing efficiency.

    Network Costs:

    More partitions increase shuffle operations, leading to higher network I/O.

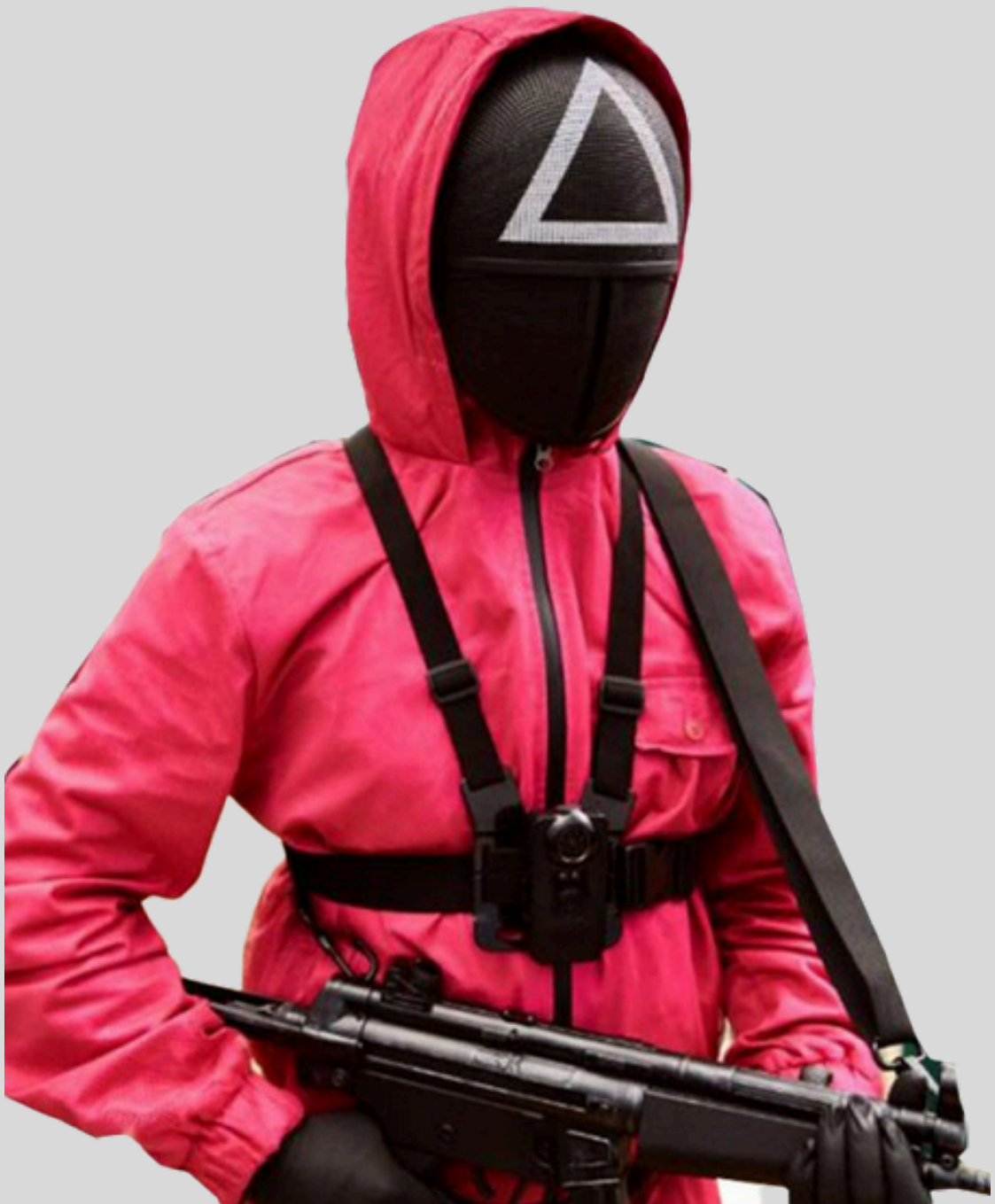
Question 6: How does Adaptive Query Execution (AQE) mitigate partition imbalance?



Candidate:

-
  - Dynamic Partition Adjustment:
    AQE adjusts the number of partitions at runtime based on shuffle statistics.
-
  - Skewed Join Optimization:
    Splits large partitions into smaller ones or broadcasts smaller datasets to balance join performance.
-
  - Fine-Tuning:
    Enabled with spark.sql.adaptive.enabled=true. Configure parameters like spark.sql.adaptive.shuffle.targetPostShuffleInputSize to control partition sizes.

Question 7: When would you prefer salting over repartitioning?



Candidate:

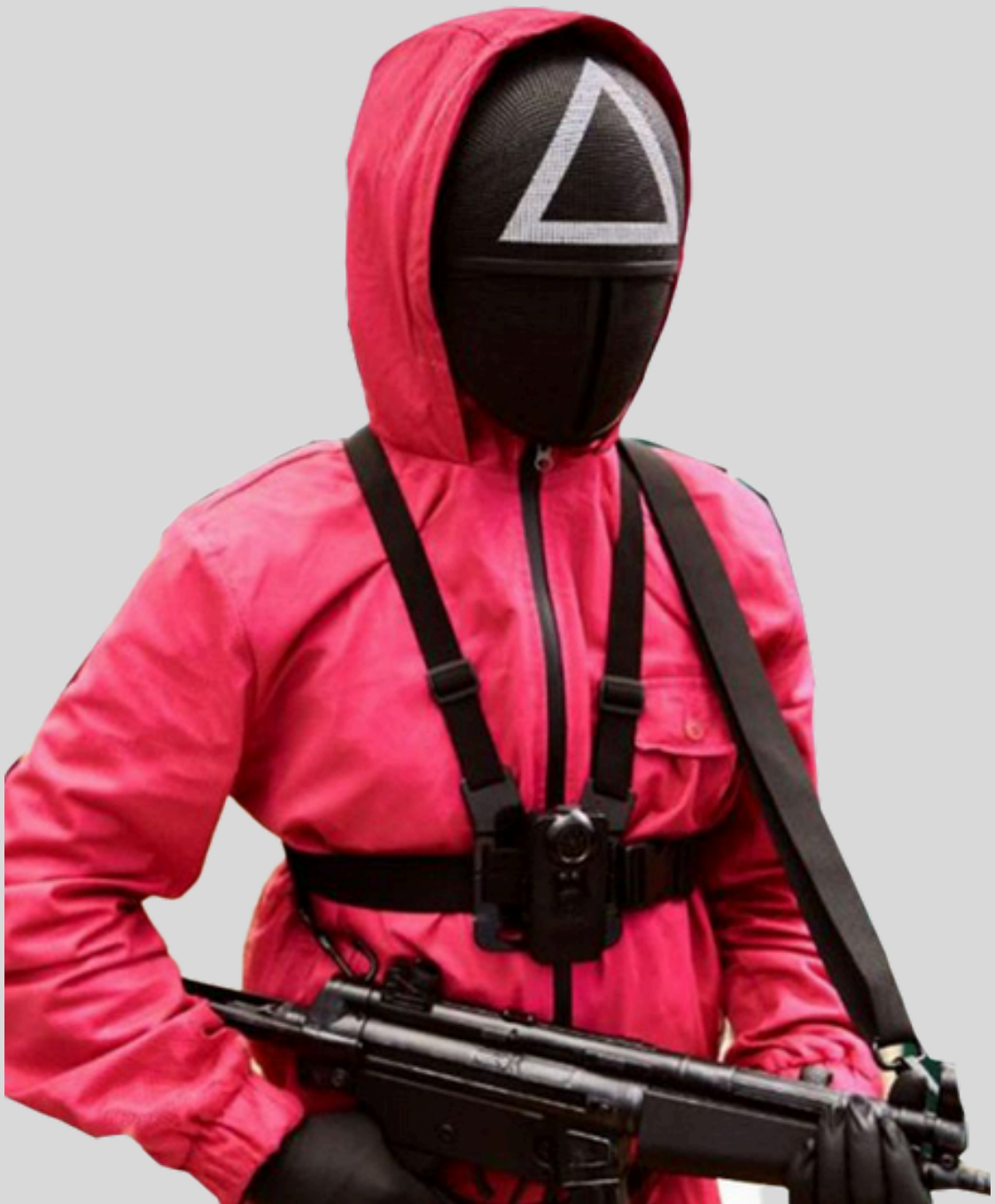
1.Highly Skewed Keys:

Repartitioning works well for moderately uneven data, but for extreme skew, salting is more effective.

1.Custom Data Needs:

When certain keys are so large that they require splitting into multiple partitions to distribute processing.

Question 8: How does the choice of partitioning scheme impact shuffle operations?



Candidate:

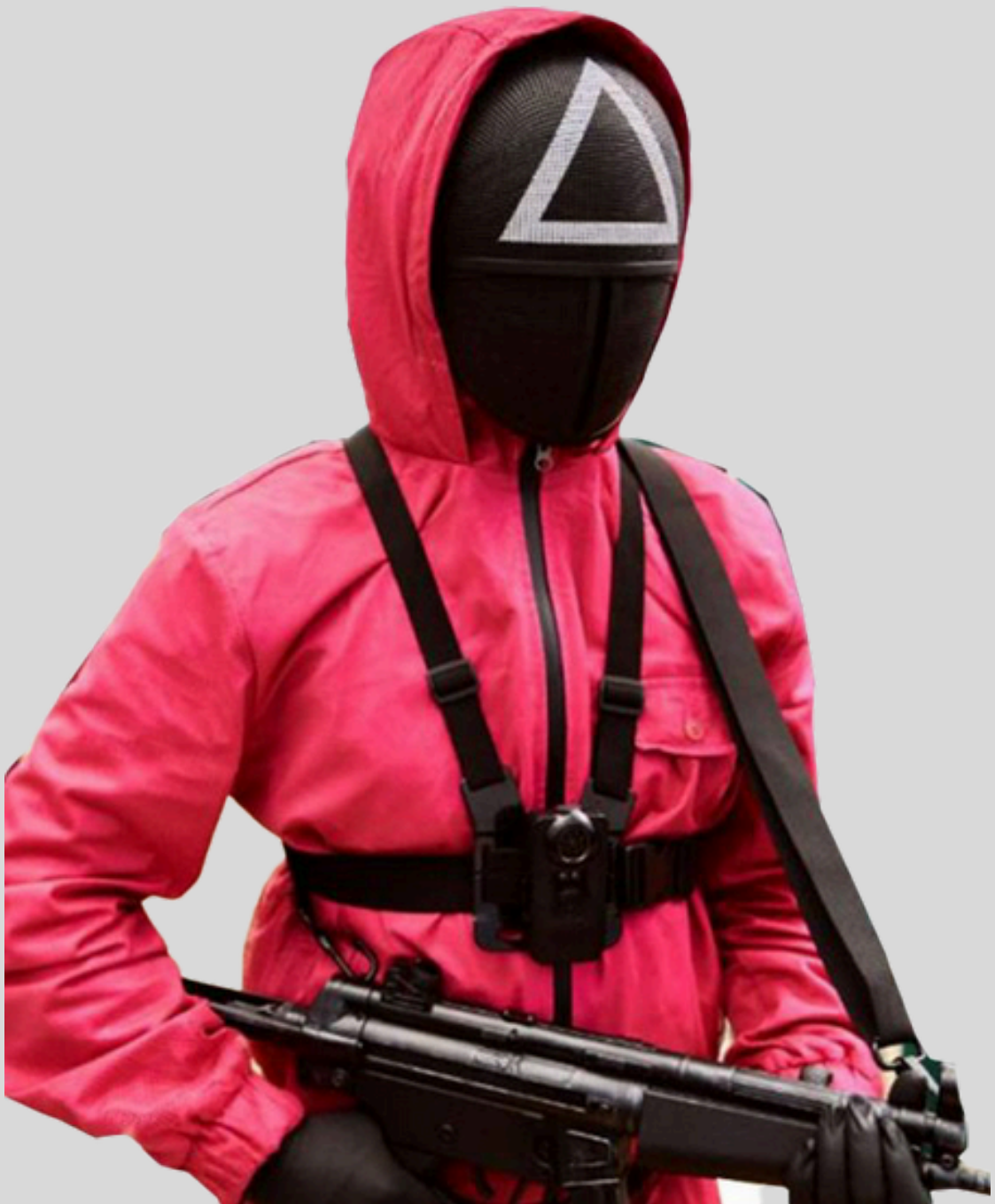
Default Hash Partitioning:

May lead to uneven data distribution for datasets with highly skewed keys.

Custom Partitioning:

Allows control over how data is distributed, reducing shuffle data size and improving performance.

Question 9: What tools or metrics would you monitor to verify your fixes?



Candidate:

- 1.Spark UI:

  Check task execution times and shuffle read/write sizes to confirm balanced partition sizes.

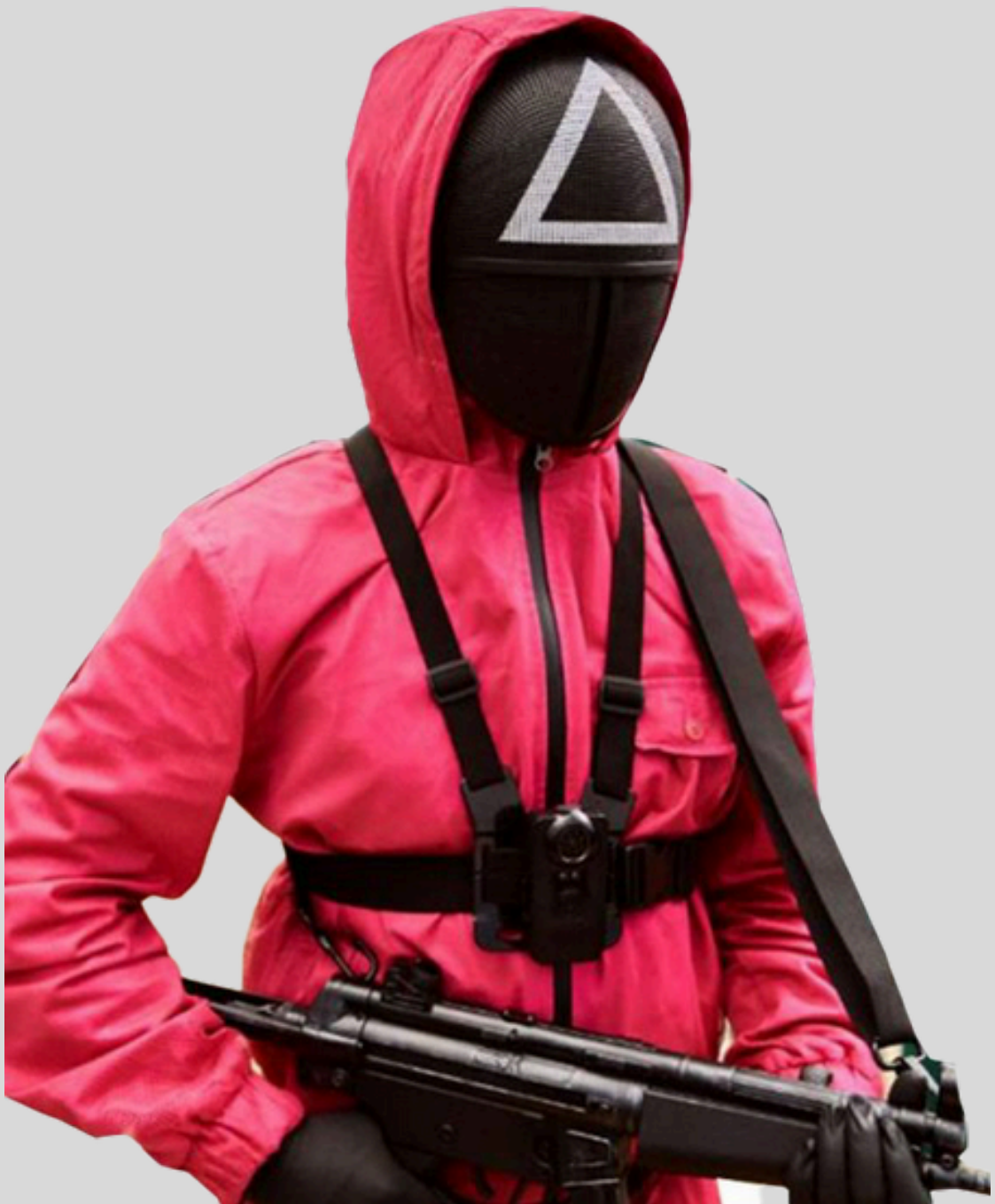
2.Logs:

Look for reduced shuffle spill warnings and fewer memory-related errors.

3.Job Completion Time:

Measure the total execution time for stages and compare before/after the fix.

Question 10: How does the choice of file format affect partitioning?



Candidate:

-
  - Columnar Formats:

    Parquet and ORC support partitioning at the storage level, enabling efficient reads for Spark jobs.

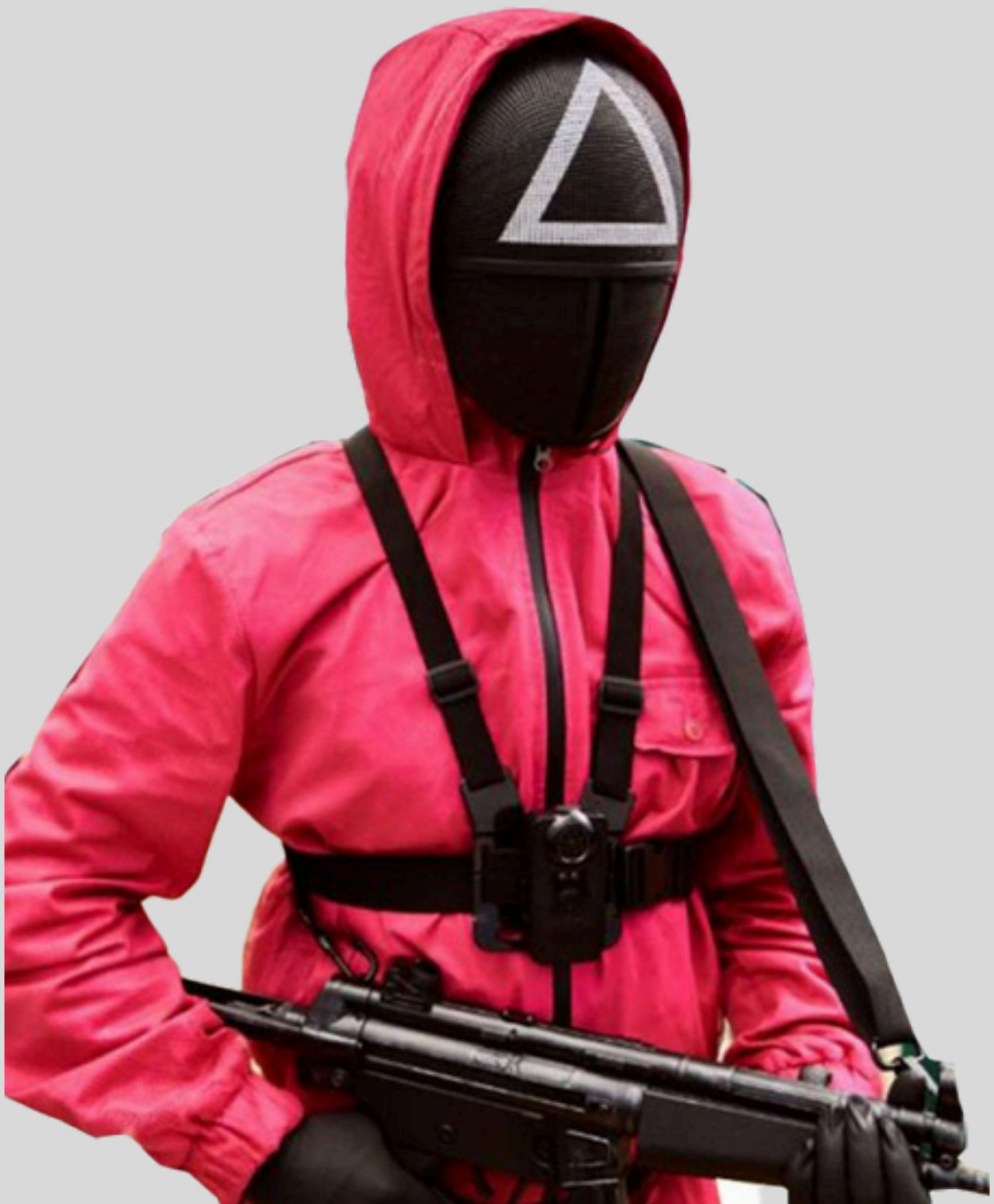
  Non-Columnar Formats:

    Formats like CSV and JSON don’t inherently support partitioning, leading to less efficient I/O operations.

  Partition Pruning:

    Using partition-aware formats allows Spark to skip unnecessary partitions, optimizing performance.

Question 11: Tell me more about Parquet and ORC. How do they differ, and how are they beneficial in a Spark workflow?



Parquet

1. Features:
   - Columnar Storage: Data is stored in columns, enabling efficient compression and query performance for analytics workloads.
   - Schema Evolution: Supports schema evolution (adding/removing columns) with backward and forward compatibility.
   - Compression: Default compression uses Snappy, but other algorithms (e.g., Gzip, Brotli) are also supported.
   - Splittable: Parquet files can be split for parallel processing, making it ideal for distributed systems.
   -
2. Advantages in Spark:
   - Supports partition pruning for optimized reads.
   - Works seamlessly with Spark's DataFrame API.
   - Efficient for aggregation queries, as columnar storage minimizes unnecessary reads.

- ORC (Optimized Row Columnar)
  - Features:
    - Columnar Storage: Similar to Parquet, but optimized for Hadoop-based ecosystems like Hive.
    - Advanced Indexing: Includes built-in indexes like min/max and bloom filters for faster data scans.
    - Compression: Default compression uses Zlib, which often results in smaller file sizes than Parquet.
    - Splittable: Supports split processing for distributed systems.
-
  -
  - Advantages in Spark:
    - Great for heavy read-intensive workloads with its indexing features.
    - Efficient for operations involving filtering or range queries due to its min/max index.
    - Supports ACID operations in Hive environments.

THANK
YOU