Data Engineer take-home assignment: Real-Time Streaming ETL (≈ 1 MD)

Task statement: This assessment evaluates your ability to design a production-grade streaming data pipeline using open-source technologies, integrate cloud services, and implement live observability. The implementation must be modular, testable, and deployable from a Unix-based environment. Your pipeline must produce structured data and sink it into a Spark-based storage system (e.g., on EMR or local Spark batch job).

Objective: Implement a modular, observable streaming ETL pipeline using open-source components on the AWS stack free-tier. Use Spark (EMR) as the target data-sink and GitHub to deploy the main stack; any other tool that eases your work is encouranged but optional. Near real-time monitoring is mandatory.

AI-based coding is encouraged: This task enforces responsible LLM usage, leveraging productivity without compromising security, reliability, or code quality. It is very hard to do it in all its parts without assisted coding within the indicated timeframes. This is because in Yard Reaas we encourage all cost-effective AI accelerations for doing your work.

1. Ingestion (~1 hr)

Choose and deploy a streaming ingestion system. It should:

- Support at-least-once or exactly-once delivery
- Run locally (e.g., via Docker Compose)
- Include a custom JSON producer that emits records periodically (e.g., simulated IoT or event data)

Deliverables:

- Producer source code
- Docker-based or CLI deployment for message broker
- Schema description of emitted records

Choose one:

- **Apache Kafka** — industry-standard messaging system offering high throughput and low latency.
- **Apache Pulsar** — decouples storage and compute, supports multi-tenancy and geo-replication.
- **RabbitMQ** or **Redis Streams** — lightweight, push-based message buses with simple low-latency delivery.
- **Amazon Kinesis** — AWS-native streaming service with simplified setup and scaling.
- **Apache Beam** — unified programming model for both batch and streaming, portable across different execution engines.

- **Apache Storm** — distributed, fault-tolerant stream processor with a spout-and-bolt topology for low-latency workloads.
- **Apache Flink** — high-performance engine providing event-time awareness, stateful streaming, and both batch and real-time processing.
- **Apache Samza** — scalable, stateful stream processor with tight integration with Kafka and incremental checkpointing.

2. Processing Engine (~1.5 hours)

Choose any streaming framework to consume from the broker and perform ETL:

- Parsing, filtering, and windowed transformations
- Aggregation or deduplication logic (e.g., average value per device per minute)

Note: Spark Structured Streaming may be used, but is not required.

Deliverables:

- Streaming job code and configuration
- Fault-tolerance considerations and retry/error handling

3. Sink to Spark-based Storage (~0.5 hours)

Processed data must be written to AWS S3 in Parquet format. You must:

- Provision storage and credentials via AWS CLI or SDK
- Demonstrate data landing and structure

Deliverables:

- S3 bucket setup script
- Spark read/test script for validation

4. Observability and Live Monitoring (~1 hour)

Set up real-time dashboards with metrics on ingestion, processing, and data throughput.

Your dashboard must show:

- Message volume over time
- Lag or latency (by partition or topic if applicable)
- Processing failures or retries

Deliverables:

- One of Grafana dashboard, Spark UI integration, or custom interface
- Dockerized Prometheus stack or similar setup

5. Repository and Automation (~0.5 hour)

- All code must be version-controlled in a GitHub repository
- Extensive use of GitHub Actions are optional but strongly encouraged

Deliverables:

- GitHub repository with modular structure
- Shell scripts or Makefile for one-command setup

6. Spark Query (~2.0 hours)

Once the pipeline has landed data in Parquet format, provide a Spark query that answers the following:

> "Compute the 95th percentile of `event_duration` per device type per day, excluding outliers that fall outside 3 standard deviations from the daily mean. Only include device types that had at least 500 distinct events per day."

Deliverables:

- Spark SQL or PySpark job to run the above query
- Output validation file (CSV)
- Resource usage considerations (e.g., memory tuning, shuffling note)

Task Overall Deliverables

GitHub repository including:

```
1. Ingestion system and producer
2. Streaming processing job
3. Spark batch query and results on S3
4. Monitoring stack (Grafana, Prometheus, SparkUI or alternative)
5. AWS provisioning scripts
6. Runbook and README (Italian acceptable; English preferred)
```

Evaluation Matrix

You will be evaluated by YR's team with the following weights by task subsection:

Component	Task Scope	Weight
Ingestion Layer	Messaging system deployment, producer implementation	5%
Processing Layer	ETL logic, filtering, fault handling	20%
Storage Integration (Spark)	S3 output, Spark compatibility, data format validation	20%

Component	Task Scope	Weight
Monitoring & Observability	Metrics export, dashboard, real-time visibility	10%
Spark Query	Logic correctness, shuffle management, percentile logic	10%
Code Quality & CI/CD	Modularity, test coverage, optional CI/CD setup	15%
Documentation & Runbook	Clarity, reproducibility, architectural rationale	20%

Note on LLM Usage

- **Workflow acceleration with LLM-based coding is strongly suggested** for boilerplate generation, tests suggestions, or prototyping complex logic.
- **Verify dependencies and imports** before using them. Almost 20% of LLM-generated code references non-existent packages, creating supply-chain risks (e.g., "slopsquatting") (infosecurity-magazine.com, techradar.com).
- **Test generated code immediately** to catch syntax errors from hallucinated methods; logical edge cases often require deeper review.
- **Refine iteratively**: use prompts to enhance performance, edge-case handling, and clarity—review each output manually.
- **Annotate LLM-generated sections** clearly in comments or commit history to aid peer review and ensure transparency.
- **Apply standard engineering safeguards**: run static analysis, dependency scans, peer code reviews, and CI tests—LLM-generated code should not bypass these.