FLIP-XXX: Batch Inference Support for ML_Predict Integration

State: Draft
Release: Targeted for Flink 1.xx
Discussion Thread:
Created: November 09, 2025
Authors: Rahul Bhattacharya

1. Motivation

Currently, our ML_Predict operator or sink calls the inference API synchronously for each record.

For large Kafka streams or expensive models, this leads to excessive API calls, slow throughput, and high cost.

Batching significantly improves efficiency:

- Batch API calls are typically **~50% cheaper** than synchronous calls.
- Posting grouped records also improves throughput and reduces API throttling risks.

This FLIP proposes a batch inference mechanism in which a Flink operator:

1. Collects a group of records from the source.
2. Writes them into a temporary JSONL file.
3. Uploads the file to the model's Batch API.
4. Monitors completion status.
5. Emits the results downstream once ready.

This enables efficient integration with ML batch inference APIs while preserving **exactly-once semantics**.

---

2. Public Interfaces

New Parameters in ML_Predict:

Parameter	Type	Description
batch	Boolean	Enables batch inference mode
batch_number	String	Unique identifier for each batch submission
ml.batch.api.endpoint	String	Endpoint for batch inference API
ml.batch.poll.interval	Duration	Interval between status polls
ml.batch.local.dir	Path	Local temporary directory for batch files

These are managed by the Flink operator when batch=true.

3. Proposed Changes

New Operator: BatchInferenceOperator
Responsibilities:
- Buffer incoming records until a threshold is reached.
- Write them to a temporary .jsonl file.
- Submit file to batch_api_endpoint.
- Poll job status until completion.
- Emit results downstream.

Configuration:
- ml.batch.enabled (Boolean) – default: false
- ml.batch.size (Int) – default: 10
- ml.batch.api.endpoint (String) – required
- ml.batch.poll.interval (Duration) – default: 10s
- ml.batch.local.dir (Path) – default: /tmp/flink_batches
- ml.batch.max.wait (Duration) – default: 10m

4. Example Flow

1. Flink reads 10 Kafka records.
2. Write them to /tmp/batch_12345.jsonl.
3. POSTs file to /ml/api/batch_predict.
4. Polls GET /ml/api/batch/status/{job_id}.
5. Once COMPLETED, fetches results and emits downstream.

5. Design Alternatives & Trade-offs

Option 1 – In-Operator Polling (Stateful Approach)

Flink stores the batch_id in operator state, keeps polling the batch API until completion, retrieves results, and emits them downstream.

- ✅ Pros — exactly-once semantics, single-operator ownership, easier recovery
- ❌ Cons — more complex operator, potential backpressure, API polling load

Option 2 – External Polling Process (Decoupled Approach)

Flink only submits the batch job and writes the batch_id + metadata to a Kafka response topic. A separate process (Flink job or another service) later polls the API, retrieves results, and emits them.

- ✅ Pros — simpler operator, scalable polling, flexible architecture
- ❌ Cons — more moving parts, loss of exactly-once, higher latency, needs another process

6. Pseudocode Implementation

Option1:

```
def process_batch(records):
  # Step 1: Write records to JSONL
  with open("/tmp/batch_input.jsonl", "w") as f:
    for r in records:
      f.write(json.dumps(r) + "\n")

  # Step 2: Submit batch
  response = requests.post(BATCH_API_URL, files={"file": open("/tmp/batch_input.jsonl", "rb")})
  batch_id = response.json()["id"]

  # Step 3: Poll until completion
  while True:
    status_resp = requests.get(f"{BATCH_API_URL}/{batch_id}")
```

```
    status_data = status_resp.json()
    if status_data["status"] == "completed":
        break
    elif status_data["status"] == "failed":
        raise Exception("Batch processing failed")
    time.sleep(10)

# Step 4: Retrieve results
results_url = status_data.get("results_url")
results_resp = requests.get(results_url)
results = results_resp.json()

# Step 5: Emit results downstream
for r in results:
    emit_to_flink_sink(r)
```

Option 2:

```
def process_batch(records):
# Step 1: Write records to JSONL
with open("/tmp/batch_input.jsonl", "w") as f:
 for r in records:
            f.write(json.dumps(r) + "\n")

# Step 2: Submit batch
 response = requests.post(BATCH_API_URL, files={"file": open("/tmp/batch_input.jsonl", "rb")})
batch_id = response.json()["id"]

# Step 3: Prepare metadata

metadata = {

            "batch_id": batch_id,

            "submitted_at": time.time(),

            "record_count": len(records),

            "input_file": "/tmp/batch_input.jsonl",

            "results_topic": "ml_batch_results",

            "api_endpoint": BATCH_API_URL,

            "status": "submitted"

}

# Step 4: Emit results downstream
emit_to_flink_sink(metadata)
```

6. Compatibility, Deprecation, Migration Plan

- New operator, not replacing existing ones.
- Synchronous ML_Predict remains unchanged.
- Jobs not enabling ml.batch.enabled continue normal operation.

7. Testing Plan

- Unit tests for batching, file creation, job submission, polling, and results.
- Integration tests with mock batch API.
- Fault injection for timeouts.
- Performance testing for various batch sizes.

8. Metrics and Monitoring

- batch.jobs.submitted
- batch.jobs.completed
- batch.jobs.failed
- batch.job.latency_ms
- batch.records_per_job

9. Rollout and Risk

Risks:

- Temporary file management and cleanup.
- API rate limits or slow polling.

Mitigation:

- Configurable polling interval and job limits.

Rollout:

- Feature flag ml.batch.enabled=false by default.
- Gradual rollout with monitoring.

10. Next Steps

1. Discussion & approval on mailing list.
2. Implement operator in experimental namespace.
3. Add documentation and configuration examples.
4. Integration testing with batch API.
5. Merge and release.