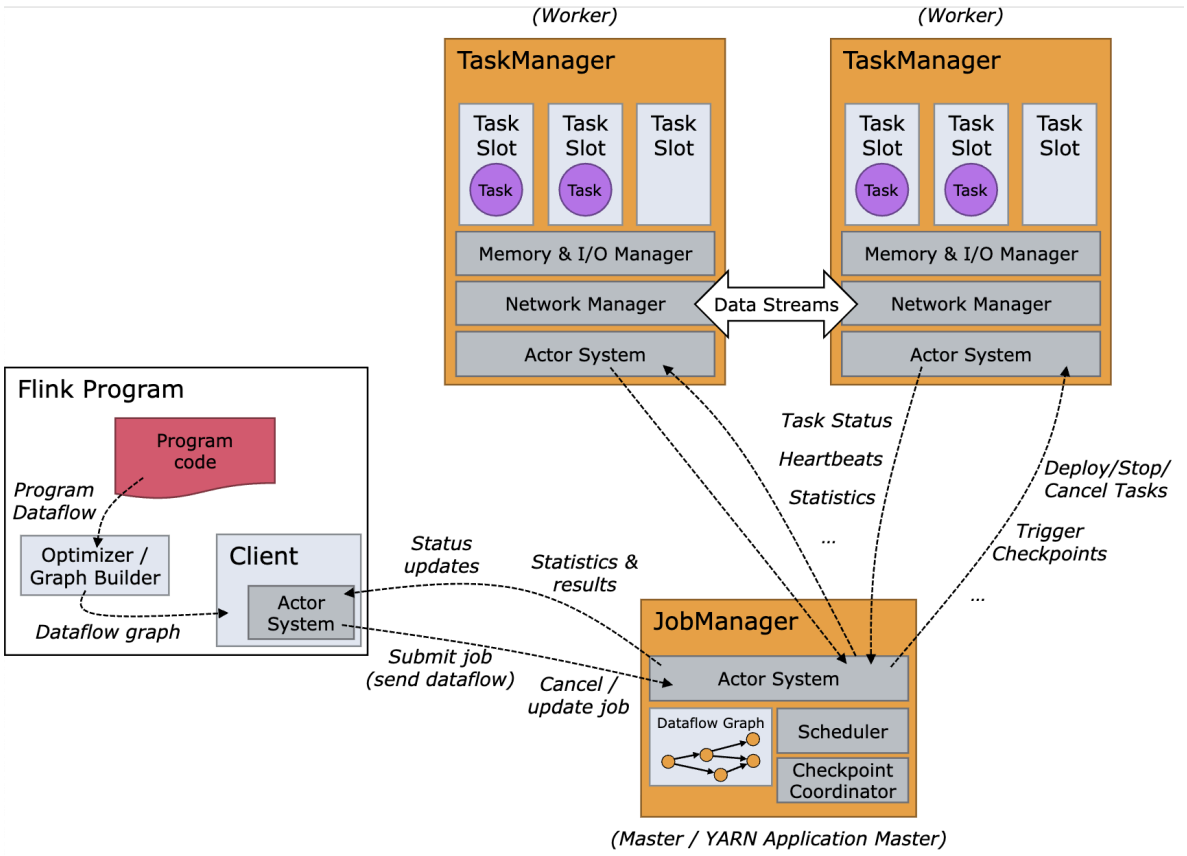
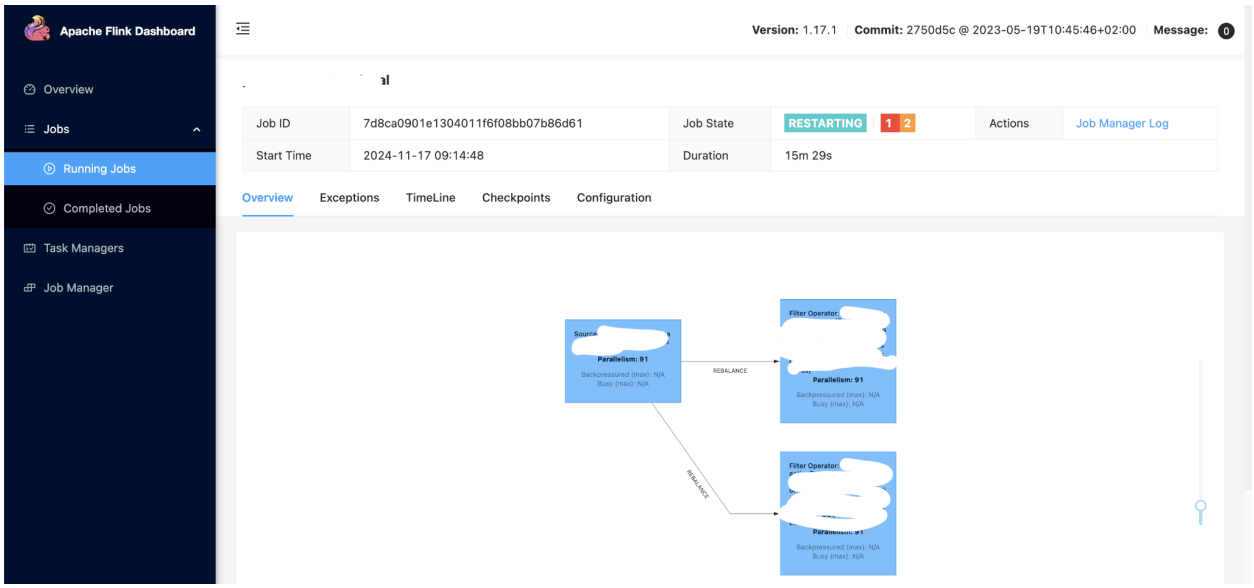
Flink





- True stream processing engine with event-by-event processing, unlike Spark's micro-batch.
- Components:

- (Flink internally uses the Akka actor system for communication between the Job Managers and the Task Managers).
- JobManager: Coordinates distributed execution, Schedules tasks, Manages checkpoints, Tracks job status, Handles failures and recovery.
    - JobManager high availability (HA) works by having multiple JobManager instances, with one acting as the leader and others as standbys, to eliminate the single point of failure.
- TaskManager: Worker nodes that execute tasks, Manage task slots, Buffer and exchange data streams, Maintain state backends
    - Task Slots: Logical fraction of resources in TaskManager or a Container for multiple tasks. Memory isolated, CPU shared.
        - Each slot can execute one parallel instance of each task in the pipeline (via slot sharing)
        - Total available parallelism = number of TaskManagers × slots per TaskManager
    - Task: One or more chained operators running in one thread. All tasks run simultaneously.
        - All tasks run simultaneously (pipelined execution model)
        - Tasks are separated by network shuffles (e.g., keyBy(), rebalance(), broadcast())
        - Compatible operators are chained into a single task to avoid serialization overhead
    - Threads: Each parallel instance of a task = 1 thread.
        - In a Slot: Multiple threads run (one per task stage), all sharing the slot's CPU quota
        - Total threads = Σ(Parallelism of each Task). If all tasks have same parallelism P: Total threads = Number of Tasks × P, where: Number of Tasks ≈ Number of shuffles + 1
        - Parallelism: Number of parallel instances of an operator/task. It can be set per operator using .setParallelism(N) or globally. Match parallelism to bottleneck and throughput requirements of each stage.

- Client: Submits jobs to JobManager, Can be CLI, web UI, or programmatic
- ResourceManager: Manages task slots, Provisions containers (YARN, Kubernetes, Mesos).

```
Eg:
Infra:
Assume 2 TaskManagers, 1 JobManager
Machine: 64 cores, 256GB RAM per TaskManager
taskmanager.numberOfTaskSlots::4
Parallelism.default::8 -> Global parallelism, i.e for all
operators

Sample Flink Job:
DataStream<SensorEvent> stream = env
    .addSource(new FlinkKafkaConsumer<>(...))  // Read from Kafka
    .name("Kafka Source")
    .filter(event-> event.temperature > 0) // Filter invalid data
    .name("Filter Invalid")
    .map(event-> enrichEvent(event)) // Transform/enrich
    .name("Enrich Data")
    .keyBy(event-> event.sensorId) // Partition by sensor- SHUFFLE
    .process(new ApiCallFunction()) // API call per sensor
    .name("API Call")
    .addSink(new JdbcSink<>(...)) // Write to database
    .name("Database Sink");
```
```
LOGICAL RESOURCE ALLOCATION
Step 1: Job Submission
1. Client submits job to JobManager
2. JobManager receives job and creates ExecutionGraph (logical
plan)
3. JobManager decides how to chain operators together
4. JobManager assigns parallelism to each operator
5. ResourceManager allocates task slots from TaskManagers

Step 2: Operator Chaining
```

Flink chains compatible operators into Tasks to avoid serialization overhead:
Resulting Tasks:
- Task 1: `[Kafka Source → Filter → Map]` (chained together)
- Task 2: `[API Call]` (separated by keyBy shuffle)
- Task 3: `[Database Sink]`

Rules for Chaining: Same parallelism across operators. No shuffle between them (no `keyBy`, `rebalance`, `broadcast`) as shuffle creates task boundary. Connected by forward partitioning (one-to-one). Chaining is enabled (default).

Step 3: Task Distribution
With parallelism = 8:
```
Task 1: [Source → Filter → Map]    Parallelism: 8  → 8 threads
                ↓ (network shuffle via keyBy)
Task 2: [API Call]                 Parallelism: 8  → 8 threads
                ↓
Task 3: [Database Sink]            Parallelism: 8  → 8 threads
> Total threads = 3 tasks × 8 parallelism = 24 threads

Step 4: Slot Allocation
With taskmanager.numberOfTaskSlots: 4 and 2 TaskManagers:
Total slots available: 2 TaskManagers × 4 slots = 8 slots
This means 3 threads per slot, 24 threads total across 8 slots

PHYSICAL RESOURCE ALLOCATION
(image pasted after this content **1)
Each slot gets ~16 cores worth of CPU time
More Slots Than Cores (Oversubscription): Good for: I/O-bound workloads (waiting on network/disk/API calls), Bad for: CPU-bound workloads (heavy computation)
Opposite case with undersubscription.
One slot per core is recommended.

Other notes:
```

```
Pipelining in 3 threads (Multiple events processing simultaneously
at different stages)
(image pasted after this content **2)
Backpressure is created if downstream threads queue is getting
filled up.
—-------------------------------------------------

Other ex:
Job:
env.setParallelism(8);  // Global default
DataStream<Event> stream = env
    .addSource(kafkaSource)
    .setParallelism(10)        // ← Override: 10 Kafka consumers
    .name("Source")
    .filter(e -> e.valid)      // Inherits 10
    .map(e -> enrich(e))       // Inherits 10
    .name("Transform")
    .keyBy(e -> e.key)         // ← SHUFFLE: Task boundary
    .window(...)
    .aggregate(...)
    .setParallelism(20)        // ← Override: 20 window operators
    .name("Windowed Agg")
    .keyBy(e -> e.userId)      // ← SHUFFLE: Task boundary
    .process(new ApiCall())
    .setParallelism(4)         // ← Override: Only 4 API callers
    .name("API")
    .map(e -> format(e))       // Inherits 4
    .keyBy(e -> e.region)      // ← SHUFFLE: Task boundary
    .addSink(jdbcSink)
    .setParallelism(16);       // ← Override: 16 DB writers

Logical Tasks Created
Task 1: [Source → Filter → Map]  Parallelism: 10 → 10 threads
         ↓ (keyBy shuffle)
Task 2: [Window → Aggregate]      Parallelism: 20 → 20 threads
         ↓ (keyBy shuffle)
Task 3: [Process (API) → Map]     Parallelism: 4  → 4 threads
```

```
         ↓ (keyBy shuffle)
Task 4: [Sink]                    Parallelism: 16 → 16 threads

Total threads = 10 + 20 + 4 + 16 = 50 threads
Slots needed = max(10, 20, 4, 16) = 20 slots (with slot sharing)

Physical Slot Distribution

With 20 slots available (5 TaskManagers × 4 slots):
SLOT DISTRIBUTION WITH SLOT SHARING:

Slot 1:  T1-inst0  + T2-inst0  + T3-inst0  + T4-inst0  = 4 threads
Slot 2:  T1-inst1  + T2-inst1  + T3-inst1  + T4-inst1  = 4 threads
Slot 3:  T1-inst2  + T2-inst2  + T3-inst2  + T4-inst2  = 4 threads
Slot 4:  T1-inst3  + T2-inst3  + T3-inst3  + T4-inst3  = 4 threads
Slot 5:  T1-inst4  + T2-inst4  + T3-empty  + T4-inst4  = 3 threads
Slot 6:  T1-inst5  + T2-inst5  + T3-empty  + T4-inst5  = 3 threads
Slot 7:  T1-inst6  + T2-inst6  + T3-empty  + T4-inst6  = 3 threads
Slot 8:  T1-inst7  + T2-inst7  + T3-empty  + T4-inst7  = 3 threads
Slot 9:  T1-inst8  + T2-inst8  + T3-empty  + T4-inst8  = 3 threads
Slot 10: T1-inst9  + T2-inst9  + T3-empty  + T4-inst9  = 3 threads
Slot 11: T1-empty  + T2-inst10 + T3-empty  + T4-inst10 = 2 threads
Slot 12: T1-empty  + T2-inst11 + T3-empty  + T4-inst11 = 2 threads
Slot 13: T1-empty  + T2-inst12 + T3-empty  + T4-inst12 = 2 threads
Slot 14: T1-empty  + T2-inst13 + T3-empty  + T4-inst13 = 2 threads
Slot 15: T1-empty  + T2-inst14 + T3-empty  + T4-inst14 = 2 threads
Slot 16: T1-empty  + T2-inst15 + T3-empty  + T4-inst15 = 2 threads
Slot 17: T1-empty  + T2-inst16 + T3-empty  + T4-empty  = 1 thread
Slot 18: T1-empty  + T2-inst17 + T3-empty  + T4-empty  = 1 thread
Slot 19: T1-empty  + T2-inst18 + T3-empty  + T4-empty  = 1 thread
Slot 20: T1-empty  + T2-inst19 + T3-empty  + T4-empty  = 1 thread

Total: 50 threads across 20 slots
Average: 2.5 threads per slot

Per-TaskManager View
```

```
TaskManager 1 (32 cores, 4 slots):
Slot 1:  4 threads }
Slot 2:  4 threads }  16 threads total
Slot 3:  4 threads }  competing for 32 cores
Slot 4:  4 threads }

TaskManager 2 (32 cores, 4 slots):
Slot 5:  3 threads }
Slot 6:  3 threads }  12 threads total
Slot 7:  3 threads }  competing for 32 cores
Slot 8:  3 threads }

TaskManager 3 (32 cores, 4 slots):
Slot 9:  3 threads }
Slot 10: 3 threads }  12 threads total
Slot 11: 2 threads }  competing for 32 cores
Slot 12: 2 threads }

TaskManager 4 (32 cores, 4 slots):
Slot 13: 2 threads }
Slot 14: 2 threads }  8 threads total
Slot 15: 2 threads }  competing for 32 cores
Slot 16: 2 threads }

TaskManager 5 (32 cores, 4 slots):
Slot 17: 1 thread }
Slot 18: 1 thread }  4 threads total
Slot 19: 1 thread }  competing for 32 cores (underutilized!)
Slot 20: 1 thread }

How Data Flows with Different Parallelism
Network Shuffle Behavior
10 Source instances reading from Kafka
         ↓
    [Rebalance/Hash Partition by key]
         ↓
20 Window instances (each receives from ~0.5 sources on average)
```

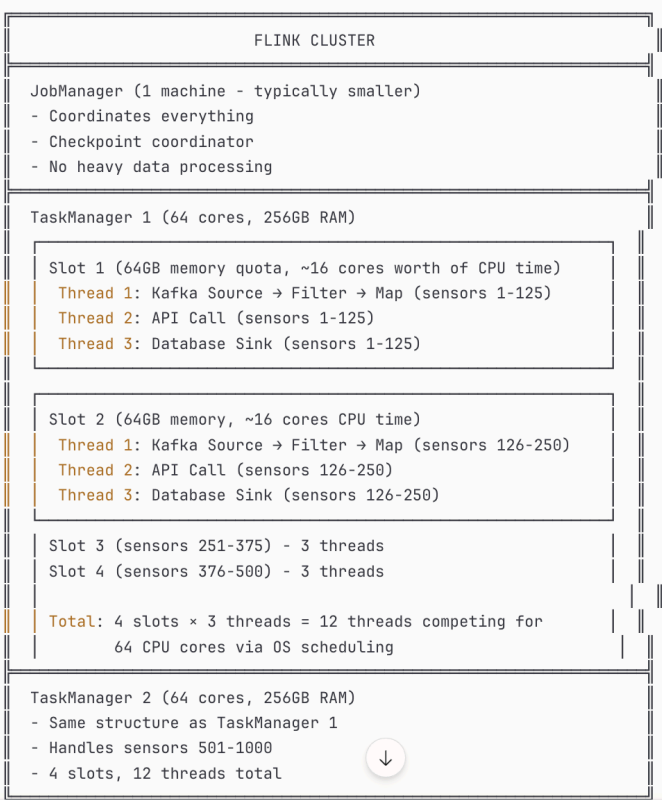
```
         ↓
    [Hash Partition by userId]
         ↓
4 API instances (each receives from ~5 window instances)
         ↓
    [Hash Partition by region]
         ↓
16 Sink instances (each receives from ~0.25 API instances)

Example event flow:
Event with key="sensor123", userId="user456", region="us-east"
1. Read by Source-instance-7 (one of 10)
2. Filter + Map in Source-instance-7
3. Network shuffle: hash("sensor123") % 20 = 13
   → Sent to Window-instance-13
4. Windowing in Window-instance-13
5. Network shuffle: hash("user456") % 4 = 2
   → Sent to API-instance-2
6. API call in API-instance-2
7. Map in API-instance-2
8. Network shuffle: hash("us-east") % 16 = 8
   → Sent to Sink-instance-8
9. DB write in Sink-instance-8

All threads use OS scheduling to compete for the 32 cores
```

(Physical resource allocation **1)

Cluster Setup:



(Pipelining in thread **2)

```
### How Events Flow (Pipelined)

**3 threads are active simultaneously, processing different events:**
```
Time   Thread 1   Thread 2   Thread 3
        (Source)   (API)   (Sink)
───
0ms   Read e1
1ms   Filter e1
3ms   Read e2   Start API(e1)
4ms   Filter e2   API(e1) running
6ms   Read e3   API(e1) running
103ms   Read e35   API(e1) done   Start Write(e1)
104ms   Filter e35   Start API(e2)   Write(e1)
113ms   Read e36   API(e2) running   Write(e1) done
114ms   Filter e36   API(e2) running   Start Write(e2)
```

**Key Insights:**
- Thread 1 doesn't wait for Thread 2 or 3
- Events are **buffered in queues** between operators
- Multiple events are "in flight" at different stages simultaneously
- Maximum throughput achieved through pipelining
```

- Watermarks:
    - Special markers in stream indicating event time has progressed to certain point. Tell Flink when to trigger window computations. Events with timestamp < watermark are considered late.
    - Operators forward minimum watermark from all input streams (input unions, joins, etc.). Ensures correctness across parallel operations.
    - Watermarking if: aggressive - data loss, conservative - data lag.
    - Types:
        - Periodic Watermarks:
            - Generated at fixed intervals (default: 200ms)
            - Based on maximum observed timestamp minus allowed lateness

```
Eg: DataStream<Event> stream = env
    .addSource(new FlinkKafkaConsumer<>("events", new
EventDeserializationSchema(), props))
    .assignTimestampsAndWatermarks(
```

```
        WatermarkStrategy

.<Event>forBoundedOutOfOrderness(Duration.ofSeconds(5))
            .withTimestampAssigner((event, ts) ->
event.getEventTime())
            .withIdleness(Duration.ofMinutes(1))  // handle idle
Kafka partitions
    );
   -  Emits watermark every 200ms by default.
   -  forBoundedOutOfOrderness: Allows events that are up to 5s
      late (within watermark delay).
   -  withIdleness(): prevents watermark stalls if one Kafka
      partition stops producing.
   -  withTimestampAssigner(...): To extract event time from
      payload. Pattern: ISO-8601 string → epoch milliseconds, as
      Flink uses epoch millis internally for event time
```

- Punctuated Watermarks:
  - Generated based on special markers in stream. Event-driven watermark generation

```
Eg:
DataStream<Event> stream = env
    .addSource(new CustomSourceFunction())
    .assignTimestampsAndWatermarks(
        new WatermarkStrategy<Event>() {
            @Override
            public WatermarkGenerator<Event>
createWatermarkGenerator(WatermarkGeneratorSupplier.Context
context) {
                return new WatermarkGenerator<Event>() {
                    private long currentMaxTimestamp =
Long.MIN_VALUE;
                    @Override
                    public void onEvent(Event event, long
eventTimestamp, WatermarkOutput output) {
                        currentMaxTimestamp =
```

```
Math.max(currentMaxTimestamp, eventTimestamp);
           // Emit punctuated watermark when special flag is seen
                       if (event.isEndOfBatch()) {
                           output.emitWatermark(new
Watermark(currentMaxTimestamp - 1));
                       }
                   }
                   @Override
                   public void onPeriodicEmit(WatermarkOutput
output) {
                       // No-op for punctuated watermark
                   }
               };
           }
           @Override
           public TimestampAssigner<Event>
createTimestampAssigner(TimestampAssignerSupplier.Context ctx) {
                return (event, recordTimestamp) ->
event.getEventTime(); }});
```

- To handle late arriving data:
  - Use Allowed Lateness: Eg: stream.keyBy(...).window(TumblingEventTimeWindows.of(Time.minutes(1))).allowedLateness(Time.minutes(5)).sideOutputLateData(lateTag.aggregate(...)
  - Side Outputs: Redirect late data to separate stream for special handling, can reprocess, log, or send to dead letter queue.

```
final OutputTag<Event> lateEventsTag = new
OutputTag<Event>("late-events"){};
SingleOutputStreamOperator<AggregatedEvent> result = stream
    .keyBy(Event::getKey)
    .window(TumblingEventTimeWindows.of(Time.minutes(1)))
    .allowedLateness(Time.minutes(5))
    .sideOutputLateData(lateEventsTag)
    .aggregate(new MyAggregateFunction());
// Retrieve late data for special handling
DataStream<Event> lateStream =
```

```
result.getSideOutput(lateEventsTag);
lateStream
    .addSink(new DeadLetterQueueSink<>("late-events-dlq"));
```

        - Watermark Configuration: Increase allowed out-of-orderness; Tradeoff: Higher latency before window triggers

```
env.getConfig().setAutoWatermarkInterval(1000L); // emit every 1 second instead of 200ms
```

        - Window Reassignment: Custom logic to reassign late events to appropriate windows
- Windows: A window defines how data in a stream is grouped for processing over time or count. Since data in streams is unbounded, windows help you create bounded slices of it for aggregation or analysis. Flink decides window boundaries based on
  > Event time (Timestamp when event actually occurred. Embedded in the event record),
  > Processing time (Timestamp when event processed by Flink operator,
  > Non-deterministic (Depends on system speed)), or
  > Ingestion time (Timestamp when event enters Flink source)
    - Types:
        - Tumbling Windows: Fixed-size, non-overlapping. Each event belongs to exactly one window. Eg: TumblingEventTimeWindows.of(Time.minutes(5)).
            - Use case: 5-minute aggregations, Periodic reports (e.g., sales per 5 minutes).
              |----5m----|----5m----|----5m----|
        - Sliding Windows: Fixed size with configurable slide interval. Windows can overlap. Eg: SlidingEventTimeWindows.of(Time.minutes(10), Time.minutes(5)...
            - Use case: Moving averages, rolling metrics
              |------10m------|

              |------10m------|

|------10m------|

        - Session Windows: Dynamic size based on inactivity gap. No fixed duration. Eg: EventTimeSessionWindows.withGap(Time.minutes(10))
            - Use case: User session analysis, burst detection
              User clicks → gap < 10m → same window
              User idle → gap > 10m → new session starts
        - Global Windows: Single window containing all events, Requires custom trigger.
            - Use case: Custom windowing logic
        - Count Windows: Based on number of elements. Eg: countWindow(100) // tumbling. countWindow(100, 10) // sliding
            - Use case: Batch-style aggregation
    - Triggers: Control when window is evaluated
        - EventTimeTrigger: Fire when watermark passes window end
        - ProcessingTimeTrigger: Fire based on processing time
        - CountTrigger: Fire after N elements
        - ContinuousEventTimeTrigger: Fire periodically within window
        - Custom Triggers: Implement TriggerResult (CONTINUE, FIRE, PURGE, FIRE_AND_PURGE)
            - FIRE emits results but keeps data for future updates; FIRE_AND_PURGE emits and clears state to free memory.
    - Evictors: Remove elements from window before/after computation
        - TimeEvictor: Keep only elements within time range
        - CountEvictor: Keep only last N elements
        - DeltaEvictor: Keep elements within delta threshold
        - Custom Evictors: Implement Evictor interface

```
Eg:
stream
  .keyBy(...)
```

```
  .window(GlobalWindows.create())
  .trigger(CountTrigger.of(1000))
  .evictor(TimeEvictor.of(Time.hours(1)))
  .reduce(...)
Visual:
Event Stream:     E1   E2   E3   E4   E5
Timestamps:       1s   2s   4s   7s   9s
Window:           [0 ------------------ 10)
Watermark:                          ↑ (fires trigger at 10s)
Trigger:         CONT → CONT → CONT → CONT → FIRE
Evictor:                           (removes old events before
compute)
```

- State Management: “state” means remembering information about past events to influence future processing. Flink stores and manages this state fault-tolerantly across operators and keys.
    - Types:
        - Keyed State: Scoped per key (after a keyBy() operation). Only accessible in keyed streams. Each key has its own isolated state partition. Eg: ValueState<T> (Single value per key), ListState<T> (List of values), MapState<K,V>, ReducingState<T>, etc.
        - Operator State: Scoped per operator instance, not per key. Used when partitioning isn’t applied (e.g., non-keyed source/sink). Eg: ListState, UnionListState, BroadcastState<K,V>
    - The State Backend determines how and where state is stored:
        - HashMapStateBackend: Stores state in-memory (JVM heap), fast, snapshots to external storage, only for development testing.
        - EmbeddedRocksDBStateBackend: RocksDB (disk), slower but scalable, best for prod.

```
StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
env.setStateBackend(new EmbeddedRocksDBStateBackend());
```

```
env.getCheckpointConfig().setCheckpointStorage("s3://flink-checkpoints");
```

    - State TTL (Time-To-Live): TTL prevents unbounded state growth. Automatic expiration of state after configured time. Can be incremental, snapshot based, compaction.

```
StateTtlConfig ttlConfig = StateTtlConfig
    .newBuilder(Time.days(7))
    .setUpdateType(StateTtlConfig.UpdateType.OnCreateAndWrite)
.setStateVisibility(StateTtlConfig.StateVisibility.NeverReturnExpired)
    .cleanupInRocksdbCompactFilter()
    .build();
Types:
OnCreateAndWrite: Reset TTL on creation and writes
OnReadAndWrite: Reset TTL on reads and writes
Disabled: TTL never updated
```

    - State Rescaling: When you change job parallelism, Flink redistributes state. Works accordingly for Keyed and Operator states.
- Checkpoints & Recovery: State is snapshotted periodically into durable storage (e.g., S3, HDFS) On recovery, Flink restores state exactly as before the failure. Steps:
    - Aligned Checkpoints:
        - Triggering:
            - The JobManager initiates a checkpoint periodically (as configured).
            - Checkpoint barriers are injected into the source tasks (Kafka, S3, etc.).
        - Barrier Propagation:
            - These barriers flow through the data stream, tagging all records as belonging to a specific checkpoint (say, #42).
            - Records before the barrier → belong to checkpoint 41.
            - Records after → belong to checkpoint 43.

        - State Snapshotting:
            - When an operator has received a barrier for checkpoint #42 from all its inputs, it:
                - Pauses processing new elements.
                - Snapshots its current operator state + keyed state asynchronously.
                - Persists this state to a checkpoint storage backend (e.g., S3, HDFS).
        - Barrier Acknowledgement:
            - Once all operators complete their snapshots, ack messages go to the JobManager.
            - When all are acked → checkpoint considered complete.
            - No global stop — processing continues concurrently (async snapshotting), just locally the operator may pause & faster upstream inputs would be buffered, downstream operators would still be running; Once checkpoint complete, then the local operator resumes.
        - What causes checkpoint alignment delays?
            - Skewed input rates — one Kafka partition slower than others.
            - Large operator state — snapshot takes long.
            - High backpressure — data flow uneven, barriers delayed.
            - Network congestion — delays barrier propagation.
            - Asynchronous snapshot thread saturation in RocksDB.
    - Unaligned checkpoints: Operators don't wait for barrier alignment, In-flight records are included in checkpoint snapshots hence big checkpoint size, Checkpoints complete even under heavy backpressure.
    - Others:
        - Checkpoint Components: Operator state, Keyed state, In-flight records (for exactly-once).
        - Flink ensures exactly-once both in computation *and sinks*.
        - Checkpoint: Automatic, periodic, lightweight, may be deleted AND Savepoint: Manual, for versioning, upgrades, always

retained

```
env.enableCheckpointing(60000); // Every 60 seconds
env.getCheckpointConfig().setCheckpointingMode(CheckpointingMode.EXACTLY_ONCE);
env.getCheckpointConfig().setMinPauseBetweenCheckpoints(30000);
env.getCheckpointConfig().setCheckpointTimeout(600000);
env.getCheckpointConfig().setMaxConcurrentCheckpoints(1);
env.getCheckpointConfig().setTolerableCheckpointFailureNumber(3);
```

- Exactly-once in Flink:
    - Exactly-Once Processing: Internal state updated exactly once per event, Achieved through checkpointing, Flink guarantees this for state updates
    - Exactly-Once Delivery: Each event written to sink exactly once, Requires transactional or idempotent sinks, Harder to achieve, depends on external system. Delivery Guarantees:
        - At-Most-Once: Fire and forget, No checkpointing needed, Data loss possible, Lowest overhead
        - At-Least-Once: Checkpoint and replay on failure, Duplicates possible, Most sinks support this
        - Exactly-Once: Transactional sinks or idempotent writes, No duplicates or loss, Highest overhead.
    - Steps:
        - Source:
            - Must be replayable (Kafka offsets, file positions)
            - Checkpoint source positions
            - Reset to checkpoint on recovery
        - Internal Processing:
            - Barrier snapshotting ensures consistent state
            - All operators see same snapshot of inputs
        - Sink:
            - Transactional Sinks: Two-phase commit (2PC)
            - Idempotent Sinks: Deterministic IDs, overwrite on replay.
              Eg: env.enableCheckpointing(60000);

sink.setDeliveryGuarantee(DeliveryGuarantee.AT_LEAST_ONCE)

- Partitioning Strategy:
  - It defines how records are distributed across parallel subtasks in Flink's dataflow graph. This directly impacts state locality, load balancing, and network shuffle costs.
  - Strategies:
    - KeyBy: stream.keyBy(event -> event.getUserId())
      - Hash partition by key
      - Same key always goes to same parallel instance
      - Enables keyed state
      - Can cause data skew if keys not uniformly distributed
    - Rebalance: stream.rebalance()
      - Round-robin distribution
      - Even load distribution
      - Causes network shuffle
      - Use when dealing with skewed data
    - Rescale: stream.rescale()
      - Local round-robin to subset of downstream tasks
      - Less network overhead than rebalance
      - Use when upstream and downstream parallelism are multiples
    - Forward:
      - One-to-one connection (no shuffle)
      - Preserves partitioning
      - Lowest overhead
    - Broadcast: stream.broadcast()
      - Send all elements to all parallel instances
      - Expensive, use sparingly
      - Useful for small reference data
  - Handle/detect data skew:
    - Flink UI to detect skewness.
    - Key Salting. Eg: .keyBy(event -> event.getUserId() + "_" + (event.getUserId().hashCode() % 4))
    - Rebalance Before Stateful Operator: Helps redistribute load

before keyBy when skewed distribution is known. Eg: rebalance().keyBy(...)
        - Dynamic Load Balancing: Implement custom partitioner. Eg: stream.partitionCustom(new CustomPartitioner(), event -> event.getKey());
        - Scale Up Parallelism: Could operator parallelism. Eg: env.setMaxParallelism(128);
- Process Functions: Process Functions are low-level API for stream processing. They give fine-grained control over: Event-time / processing-time timers, Custom logic beyond windowing / simple transformations, State access (keyed or operator), Side outputs (for late data or special cases). Types:
    - ProcessFunction<IN, OUT>: Operates on non-keyed streams. Gives access to processing time, event time, and side outputs. Cannot use keyed state (only operator state).

```
stream.process(new ProcessFunction<Event, Result>() {
    @Override
    public void processElement(Event event, Context ctx,
Collector<Result> out) {
        out.collect(new Result("Received: " + event));
    }
});
```

    - KeyedProcessFunction<K, IN, OUT>: Operates on keyed streams (after .keyBy()). Can use keyed state (ValueState, ListState, etc.). Can register event-time or processing-time timers per key.
        - Non-keyed stream: A normal DataStream where all events are processed together by each parallel subtask, without partitioning by key. Every operator instance gets a subset of the stream — distributed in parallel — but no key-based grouping is done.
        - Keyed Stream: Created by applying .keyBy(...) on a stream. Partitions the stream by key value, so all records with the same key go to the same parallel subtask. Usecase: If you need to track information per user, session, merchant, device,

etc., like: Counting clicks per user:

```
Example for Keyed stream:
stream
      .keyBy(Event::getUserId)
      .process(new KeyedProcessFunction<String, Event,
UserCount>() {
          private ValueState<Integer> count;
          public void open(Configuration parameters) {
              count = getRuntimeContext().getState(new
ValueStateDescriptor<>("count", Integer.class));
          }
          public void processElement(Event event, Context ctx,
Collector<UserCount> out) throws Exception {
              int newCount = (count.value() == null ? 0 :
count.value()) + 1;
              count.update(newCount);
              out.collect(new UserCount(event.getUserId(),
newCount));
          }
      });

Example for Keyed Process function:
sstream.keyBy(Event::getUserId)
    .process(new KeyedProcessFunction<String, Event, Result>() {
        ...
    }
```

- ProcessWindowFunction<IN, OUT, KEY, W extends Window>: Used inside window operations. Gives access to window metadata (start/end timestamp). Provides all elements of the window for full aggregation.

```
stream.keyBy(Event::getUserId)
    .window(TumblingEventTimeWindows.of(Time.minutes(5)))
    .process(new ProcessWindowFunction<Event, Result, String,
TimeWindow>() {
```

```
        @Override
        public void process(String key, Context ctx,
Iterable<Event> events, Collector<Result> out) {
            long count =
StreamSupport.stream(events.spliterator(), false).count();
            out.collect(new Result(key, count,
ctx.window().getEnd()));
        }
    });
```

- ProcessAllWindowFunction<IN, OUT, W extends Window>: Same as above, but for non-keyed windows (global).

```
stream.windowAll(TumblingProcessingTimeWindows.of(Time.seconds(10)
))
      .process(new ProcessAllWindowFunction<Event, Result,
TimeWindow>() {
          @Override
          public void process(Context ctx, Iterable<Event> events,
Collector<Result> out) {
              out.collect(new Result("Total count = " +
StreamSupport.stream(events.spliterator(), false).count()));
          }
      });
```

- CoProcessFunction<IN1, IN2, OUT>: Processes two connected streams together. Can access state shared between both inputs.

```
DataStream<EventA> streamA = ...
DataStream<EventB> streamB = ...
streamA.connect(streamB)
    .process(new CoProcessFunction<EventA, EventB, Result>() {
        private ValueState<EventA> pendingA;
        ...
    }
```

- Async I/O in Flink: Avoid blocking operators waiting for external service

calls. Increase throughput with concurrent requests. Reduce latency. Use in API calls, DB reqs and lookups, etc.

- orderedWait(): Preserve event order (lower throughput)
- unorderedWait(): Allow reordering (higher throughput)

```
Eg:
class AsyncDatabaseRequest extends RichAsyncFunction<String,
Tuple2<String, String>> {
    private transient DatabaseClient client;

    @Override
    public void open(Configuration parameters) {
        client = new DatabaseClient();
    }

    @Override
    public void asyncInvoke(String key,
ResultFuture<Tuple2<String, String>> resultFuture) {
        CompletableFuture<String> future = client.asyncQuery(key);

        future.whenComplete((result, error) -> {
            if (error != null) {
                resultFuture.completeExceptionally(error);
            } else {
                resultFuture.complete(Collections.singleton(new
Tuple2<>(key, result)));
            }
        });
    }

    @Override
    public void timeout(String input, ResultFuture<Tuple2<String,
String>> resultFuture) {
        resultFuture.complete(Collections.singleton(new
Tuple2<>(input, "TIMEOUT")));
    }
}
```

```
// Usage
AsyncDataStream.unorderedWait(
    stream,
    new AsyncDatabaseRequest(),
    10000, TimeUnit.MILLISECONDS,  // timeout
    100  // max concurrent requests
);
```

- Broadcast State pattern: Broadcasting rules/config to all parallel instances for enrichment. Each task (operator instance) maintains its own replica of the broadcast state, ensuring every event is processed with the latest rules — without network lookups. Broadcast state is fully replicated across all parallel subtasks → memory pressure grows linearly with number of subtasks × state size. To Manage Broadcast State Growth: Evict stale entries, State TTL, Compaction, Shard/split broadcast stream (multiple streams), Use rocksdb backend.

```
// Stream 1: Rules/config (low volume)
BroadcastStream<Rule> broadcastStream = rulesStream
    .broadcast(rulesStateDescriptor);

// Stream 2: Events (high volume)
DataStream<Event> eventStream = ...;

// Connect and process
eventStream
    .connect(broadcastStream)
    .process(new BroadcastProcessFunction<Event, Rule, Result>() {
        @Override
        public void processElement(Event event, ReadOnlyContext
ctx, Collector<Result> out) {
            // Read broadcast state
            for (Map.Entry<String, Rule> entry :

ctx.getBroadcastState(rulesStateDescriptor).immutableEntries()) {
                Rule rule = entry.getValue();
```

```
                if (rule.matches(event)) {
                    out.collect(new Result(event, rule));
                }
            }
        }

        @Override
        public void processBroadcastElement(Rule rule, Context
ctx, Collector<Result> out) {
            // Update broadcast state

ctx.getBroadcastState(rulesStateDescriptor).put(rule.getId(),
rule);
        }
    });
```

- Side Output: Additional output streams from operators for specific use cases. Used for late data handling, error streams, etc.

```
final OutputTag<Event> lateDataTag = new
OutputTag<Event>("late-data"){};
final OutputTag<Event> errorTag = new
OutputTag<Event>("errors"){};

SingleOutputStreamOperator<Result> mainStream = stream
    .process(new ProcessFunction<Event, Result>() {
        @Override
        public void processElement(Event event, Context ctx,
Collector<Result> out) {
            try {
                if (event.getTimestamp() <
ctx.timerService().currentWatermark()) {
                    // Late data
                    ctx.output(lateDataTag, event);
                } else if (!event.isValid()) {
                    // Invalid data
                    ctx.output(errorTag, event);
```

```
                } else {
                    // Normal processing
                    out.collect(process(event));
                }
            } catch (Exception e) {
                ctx.output(errorTag, event);
            }
        }
    });

// Get side outputs
DataStream<Event> lateData =
mainStream.getSideOutput(lateDataTag);
DataStream<Event> errors = mainStream.getSideOutput(errorTag);
```

- A SingleOutputStreamOperator<T> represents a transformation in your Flink job that produces one output stream of type T. This operator produces exactly one output stream. For transformations that produce multiple output types (like side outputs), Flink uses: ProcessFunction + OutputTag to produce multiple outputs (main + side).
    - DataStream → the base abstraction.
    - SingleOutputStreamOperator → a DataStream that comes from an operation producing one output.
    - KeyedStream, WindowedStream → specializations for keyed/windowed data.
- Async Functions:
    - AsyncFunction<IN, OUT>: Lightweight - just asyncInvoke()
    - RichAsyncFunction<IN, OUT>: + lifecycle hooks (open, close, getRuntimeContext). Any "Rich" prefix = Lifecycle + RuntimeContext access Examples: RichMapFunction, RichFlatMapFunction, RichAsyncFunction RuntimeContext: Provides metadata & utilities about running task
- Joins:
    - For event-time Join between two streams, can use intervalJoin.

```
SingleOutputStreamOperator<JoinedEvent> joinedStream =
```

```
    leftStream
        .keyBy(LeftEvent::getJoinKey)
        .intervalJoin(rightStream.keyBy(RightEvent::getJoinKey))
        .between(Time.seconds(-60), Time.seconds(60))
        .process(new ProcessJoinFunction<LeftEvent, RightEvent,
JoinedEvent>() {
            @Override
            public void processElement(
                LeftEvent left,
                RightEvent right,
                Context ctx,
                Collector<JoinedEvent> out
            ) throws Exception {
                JoinedEvent joined = new JoinedEvent();
                joined.setId(UUID.randomUUID().toString());
                joined.setTimestamp(right.getEventTimestamp());
                // Combine/enrich from both sides
                out.collect(joined);
            }
        });

How .between() Works
For each LEFT event at time T:
  Match RIGHT events where:
    rightTime ∈ [leftTime - 60s, leftTime + 60s]
Example:
  Left event at 12:00:30
  Matches right events from 11:59:30 to 12:01:30

Key Points
Requirement: Both streams must be keyed on join key
Time-based: Uses event time (not processing time)
Symmetric window: -60s to +60s means 2-minute total window
ProcessJoinFunction: Defines join logic & output type
```

- Sinks (S3/Kafka, etc):
    - FileSink API to write in S3. Kafka methods to publish in Kafka.

```
// FileSink API:
OutputFileConfig fileConfig = OutputFileConfig.builder()
    .withPartPrefix("user-data-")
    .withPartSuffix(".json")
    .build();
FileSink<T> sink = FileSink
    .forRowFormat(new Path(s3BasePath), new Encoder<T>() {
        @Override
        public void encode(T element, OutputStream stream) throws
IOException {
            String json =
objectMapper.writeValueAsString(element);
            stream.write(json.getBytes(StandardCharsets.UTF_8));
            stream.write('\n');  // Newline-delimited JSON
        }
    })
    .withBucketAssigner(new
DateTimeBucketAssigner<>("yyyy-MM-dd"))
    .withRollingPolicy(
        DefaultRollingPolicy.builder()
            .withRolloverInterval(5000)      // Roll every 5s
            .withInactivityInterval(5000)    // Roll after 5s idle
            .withMaxPartSize(1024)           // Roll after 1KB
            .build()
    )
    .withOutputFileConfig(fileConfig)
    .build();
Info:
BucketAssigner: Organizes files by date/time
RollingPolicy: When to close & start new files. Like: Rollover
interval: Time-based, Inactivity: Idle period, Max size: File size
threshold.
Encoder: Custom serialization logic

—-----------------------------------
// Kafka Sink:
KafkaRecordSerializationSchema<Event> schema =
```

```
    KafkaRecordSerializationSchema.<Event>builder()
        .setTopic("output-topic")
        .setValueSerializationSchema(new EventSchema())
        .build();
KafkaSink<Event> kafkaSink = KafkaSink.<Event>builder()
    .setBootstrapServers("broker1:9092,broker2:9092")
    .setRecordSerializer(schema)
    .build();
// Usage
stream.sinkTo(kafkaSink);
Info:
KafkaRecordSerializationSchema: How to serialize + which topic
ValueSerializationSchema: Custom serializer for your type
setRecordSerializer: Wires serialization logic to sink
```

- Flink's Table API and SQL API are high-level, declarative APIs built on top of the DataStream API. You can register DataStreams as Tables, perform Table or SQL queries, and then convert the results back to streams.

```
StreamTableEnvironment tableEnv =
StreamTableEnvironment.create(env);

// Register a stream as a table with schema and watermark
tableEnv.createTemporaryView(
    "Events",
    eventStream,
    Schema.newBuilder()
        .column("userId", DataTypes.STRING())
        .column("amount", DataTypes.DOUBLE())
        .column("eventTime", DataTypes.TIMESTAMP(3))
        .watermark("eventTime", "eventTime - INTERVAL '5' SECOND")
        .build()
);

// Table API query
Table result = tableEnv.from("Events")
```

```
    .groupBy($("userId"))
    .select($("userId"), $("amount").sum().as("totalAmount"));

// Convert back to DataStream
DataStream<Row> resultStream = tableEnv.toDataStream(result);

// Equivalent SQL ex:
Table result = tableEnv.sqlQuery(
    "SELECT userId, SUM(amount) AS totalAmount " +
    "FROM Events GROUP BY userId"
);
```

- CEP (Complex Event Processing): Detect patterns in event streams. Used for fraud detection.

```
Pattern<Event, ?> pattern = Pattern.<Event>begin("start")
    .where(new SimpleCondition<Event>() {
        @Override
        public boolean filter(Event event) {
            return event.getType().equals("login");
        }
    })
    .next("middle")
    .where(event -> event.getType().equals("payment"))
    .followedBy("end")
    .where(event -> event.getType().equals("logout"))
    .within(Time.minutes(10));
```

- Other notes:
    - Operations in Flink:

```
  -  Transformations:
       -  map: One-to-one transformation that applies a function
          to each element. Eg: dataStream.map(event ->
          event.toUpperCase());
       -  flatMap: One-to-many transformation that can produce
          zero, one, or multiple elements for each input. Eg:
          dataStream.flatMap((event, out) -> {for (String word :
```

```
        event.split(" ")) { out.collect(word); }});
    - filter: Keeps elements that satisfy a condition. Eg:
      dataStream.filter(event -> event.contains("error"));
- Keying & Partitioning:
    - keyBy: Groups the stream by a key (creates a
      KeyedStream). Eg: dataStream.keyBy(event ->
      event.getUserId());
    - shuffle: Redistributes elements randomly. Eg:
      dataStream.shuffle();
    - rebalance: Evenly distributes elements across tasks. Eg:
      dataStream.rebalance();
    - rescale: Redistributes elements in a round-robin fashion
      among downstream tasks. Eg: dataStream.rescale();
    - broadcast: Replicates each element to all parallel
      tasks. Eg: dataStream.broadcast()
- Aggregations & Windows:
    - reduce: Combines elements of a KeyedStream using a
      reduce function. Eg: keyedStream.reduce((a, b) -> new
      Sum(a.value + b.value));
    - aggregate: Applies an aggregation function (like sum,
      min, max). Eg: keyedStream.sum("amount");
      keyedStream.min("latency"); keyedStream.max("value");
    - window: Groups elements of a stream into finite sets
      based on time or count. Eg:
      keyedStream.window(TumblingEventTimeWindows.of(Time.seco
      nds(5)));
      keyedStream.window(SlidingProcessingTimeWindows.of(Time.
      seconds(10), Time.seconds(5)));
      keyedStream.countWindow(100, 10); // sliding count
      window
    - windowAll: Creates a window on a non-keyed stream. Eg:
      dataStream.windowAll(TumblingProcessingTimeWindows.of(Ti
      me.minutes(1)));
- Joining & Combining
    - join: Joins two data streams based on a key. Eg:
      stream1.join(stream2).where(e1 ->
      e1.getKey()).equalTo(e2 ->
```

```
    e2.getKey()).window(TumblingEventTimeWindows.of(Time.sec
    onds(5))).apply((e1, e2) -> new Tuple2<>(e1, e2));
- coGroup: Groups and joins two streams in a more flexible
    way than join. Eg: stream1.coGroup(stream2).where(e1 ->
    e1.getKey()).equalTo(e2 ->
    e2.getKey()).window(TumblingEventTimeWindows.of(Time.sec
    onds(5))).apply(new CoGroupFunction<...>() {...});
- union: Combines multiple streams of the same type. Eg:
    stream1.union(stream2, stream3);
- connect: Combines two streams of potentially different
    types. Eg: stream1.connect(stream2).map(new
    CoMapFunction<Type1, Type2, ResultType>() {...});
```

- Flink Performance Optimization: Parallelism Tuning (setParallelism), State Optimization (ValueState or MapState, Implement state TTL, Incremental Checkpointing, RocksDB Backend state), Checkpoint Optimization, Resource Configs, Network Optimize (Chain operators, reduce shuffle, local aggregations before shuffle), Operator Optimization (Use AggregateFunction instead of ProcessWindowFunction, Batch operations if possible).
- Backpressure reasons: Slow operators, External system latency, Insufficient resources, Data skew (hot keys), GC pressure, Checkpoint alignment delays.
- In eks yaml deployment config file you may use: upgradeMode key. The upgradeMode setting determines how Flink jobs behave during upgrades or restarts:
    - last-state mode: When a job is upgraded or restarted, it will attempt to restore from the most recent successful checkpoint. This preserves the processing state and allows the job to continue from where it left off.
    - stateless mode: The job will start from scratch after an upgrade, without attempting to restore any previous state. This is like a completely fresh deployment.
    - Note that if you have checkpointing related configs in yaml; The checkpointing configuration remains active in both

modes - it's just that the stateless mode ignores existing checkpoints when pipeline starting up.

- Deployment modes in Flink:
  - Session Mode: Long-running cluster, Multiple jobs share same cluster, Resources shared across jobs, Use case: Development, multiple small jobs
  - Application Mode: One cluster per application, Main method runs on cluster, Better resource isolation, Use case: Production applications
  - Per-Job Mode (Deprecated): One cluster per job, Cluster terminated after job finishes, Use case: Batch jobs, isolation

---