In the following, I explain three potential solutions as well as their trade-offs depending on the current system's status and statistics of timeouts. I **assume** that timed-out jobs eventually finish after a few minutes (they would not be running forever due to a deadlock); I'll put a note at the end for this case.

1) Increase Time Interval (Baseline)

The first solution is to increase the time interval (e.g., 10 minutes). While timeout tolerance increases, computation time does not increase linearly, which results in a fewer number of timeouts. **Downsides: 1)** The latency of the data increases. However, the consequences of this must be checked with the business side. If suppliers are shipping once a day, is this increase problematic for their process? **2)** This change might bother other potential users of this data pipeline. They might need real-time data or might be dependent on 5 minutes granularity. (e.g., BI team or Machine Learning models) **3)** The data partitioning should be revised too. Do we want to keep it as 5-minute intervals and query over two partitions or change it accordingly? **4)** Developers still might need to run ETL pipelines for some failed jobs. **5)** This solution does not help if most timed-out jobs run forever **Upsides: 1)** Relatively easy implementation

2) Queuing

The jobs which finish in 5 minutes and 10 seconds are timed-out in the default design resulting in a huge loss of computations. The second solution is to implement a queuing system that puts the next job in the queue every 5 minutes. The next execution kicks off when the current one is finished. (I would still put a timeout of 15 minutes to prevent jobs from running forever). Assuming that jobs take less than five minutes on average, this solution works perfectly fine. On the other hand, if there are a lot of consecutive jobs taking more than five minutes (e.g., during rush hour), the jobs are queued up until the pressure goes down and the system gradually starts to keep up with the pace of incoming jobs. The default design, however, would have a lot of timeouts in this case. **Downsides: 1)** The system might have less amount of latest/recent data compared to the default design. **2)** More implementation effort compared to baseline **Upsides: 1)** Less computation waste **2)** No need to manually run ETL pipeline anymore **3)** An auto-scaling policy can be set up for the rush hour (when the delay is high) to efficiently use more resources when needed. **4)** The actual delay can be provided to the data users as helpful information.

3) Queuing + Flexible Windowing

In the queuing solution, the system might have a lot of jobs being queued up during rush hour. In this solution, I change the design a way that each job does the computation for all the completed five-minute slots (instead of one). This way when the system is falling behind for more than one slot, it automatically uses the first solution to speed up. **Upsides: 1)** All benefits of the previous two solutions. **Downsides: 1)** More implementation effort. **2)** More changes to the current system that adds to the risk of negative side effects to other services. **3)** "do we need this solution?" If there are only a few non-consecutive timeouts, solution 2 would be enough. I would start with solution 2 and then implement the latter if needed.

Final Note (Forever Running Jobs): If most timed jobs are running forever, the first solution does not help. The second solution works if we add the failed job to the queue again. The result depends on the statistics of the execution times. Timeout time should be also **tuned** accordingly as we might prefer to keep it at 5 minutes as before. That way, we are eliminating the need to manually run the pipeline by automating it. In other words, this is a fourth solution to run failed jobs automatically again using a queuing system.