FTS4: Scheduler and FlowController: Scalable Scheduling and Adaptive Transfer Concurrency

Nicola Pace on behalf of the CERN FTS team

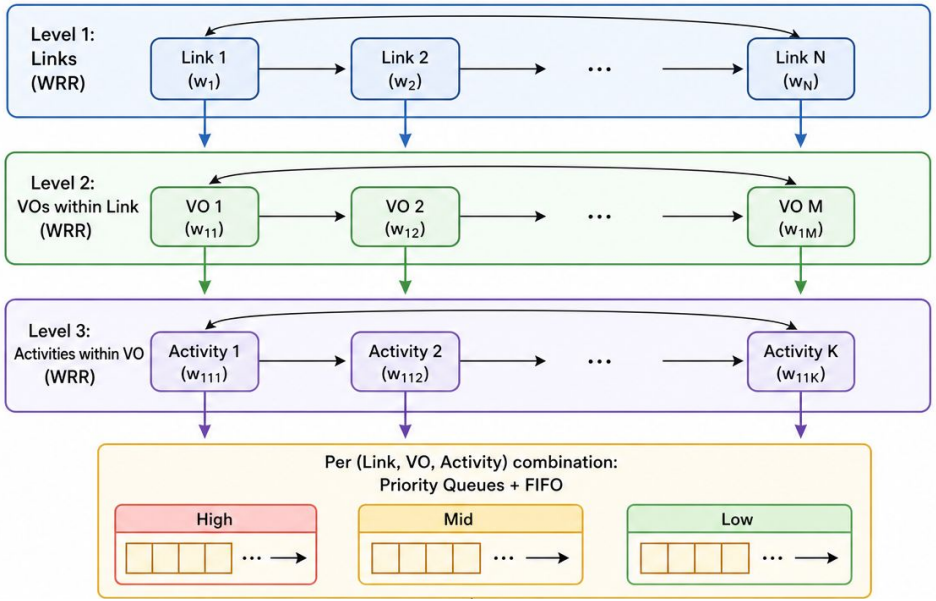
FTS & XRootD Workshop 2026 - 15th Sept. 2026, Lyon

Summary

- Scheduler design
  - Data challenge mode
- DB usage at scale
  - Deadlocks (some kinds of deadlocks)
  - Triggers overhead
  - Shared counter bottleneck
- Flow controller

FTS4 Disk Scheduler design

Multi-Level Round-Robin Architecture



1. Link-Level: Each link gets equal opportunity (RR). No link starvation regardless of queue size.
2. VO-Level: Each VO gets equal opportunity per link (RR). Exp. 1 and Exp. 2 treated fairly on same link.
3. Activity-Level: Activities share capacity proportionally (WRR). Production:Analysis:Default = 50:30:20.

Smart Filtering: Weights calculated only for *active* sources. Idle sources don't steal capacity from working sources.

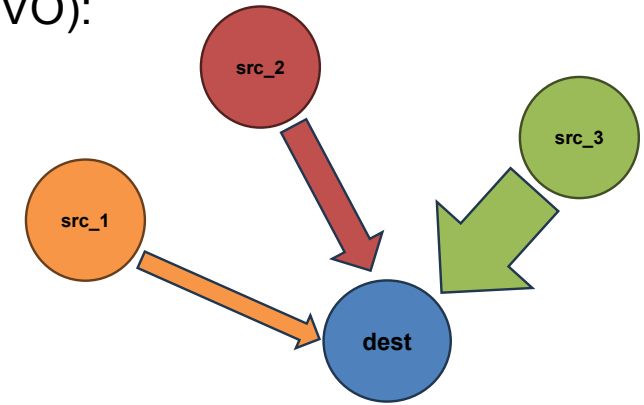
- inbound weights - tape factor - data challenge mode

Achieved required scheduling **speed**
(*10k links -> **1KHz** scheduling rate*)

Inbound Weights

User defined weight Configuration (per destination + VO):

```
"destination": "dest",
"vo": "atlas",
"inbound_weights": {
  "src_1": 20,
  "src_2": 30,
  "src_3": 50,
}
```



Enforcement:

Allowed transfers = (source_weight / total_weight) × destination_max_capacity

Example (destination capacity = 100):

- src_1: max 40 concurrent transfers (20%)
- src_2: max 30 concurrent transfers (30%)
- src_3: max 10 concurrent transfers (50%)

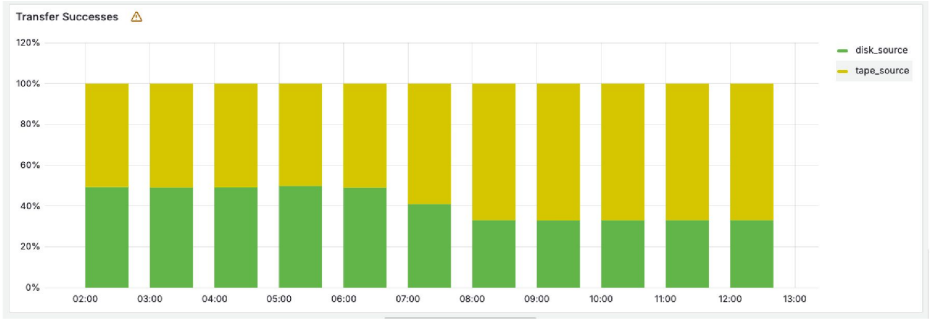
Auto Tape Multiplier

Problem: Endpoints don't favor data recall from tape, causing **delays** and **premature garbage collection**. Goal: avoid retrieving the same file twice.

NOT STARVING + BOOST TO TAPE

AUTOMATIC Tape Multiplier

- Introduce **tape_multiplier = 2** (configurable)
- Tape-B effective weight = 50 × 2 = **100**
- Disk-A (weight=50) + Tape-B (weight=100)
- Result: **33% Disk — 67% Tape**
- Tape gets priority to maintain **efficient streaming**

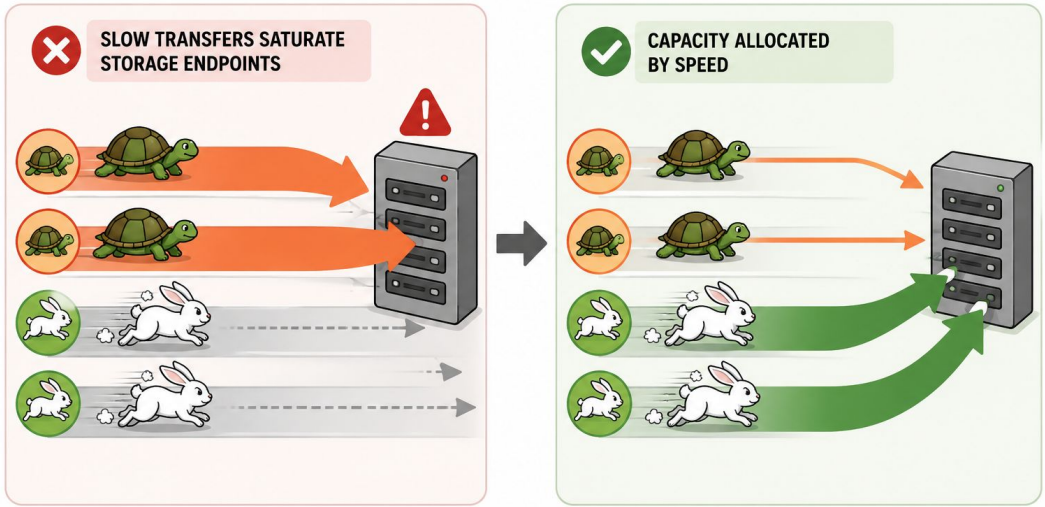


Impact: Tape-staged files are read out faster, preventing premature garbage collection. **No manual operator intervention needed.** Configurable multiplier allows tuning per storage endpoint.

Throughput-Based Allocation: Data Challenge Mode

FTS
File Transfer Service

DC24 Lesson: Slow transfers saturated storage endpoints, preventing fast transfers from getting through. We must **"correct the mistakes of DC24."**



FTS3: Nothing

- Weights are **manually configured**
- Don't adapt to **network conditions**
- Don't reflect **actual performance**
- Slow links **saturate** FTS capacity

FTS4: Dynamic Throughput-Based

- can be tested with mock
- throughput sampled -> link ranking
- Slower sources get **less capacity**

Work in Progress: Throughput sampler implementation and validation ongoing.
Target: automatic regulation balancing fairness with performance under DC27 load.

Bulk DB Updates & Inserts at Scale

FTS
File Transfer Service

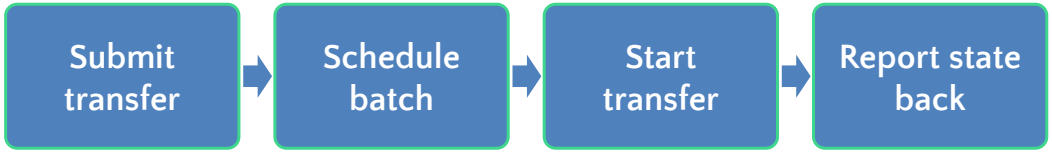
Players in the system

- Transfers submission
- Scheduling
- Getting transfers to start
- Reporting state of transfers

Idea: speed things up with bulk updates and bulk inserts — batch instead of row-by-row. Postgres guarantees data consistency given MVCC.

Several components now write concurrently.

The write pipeline



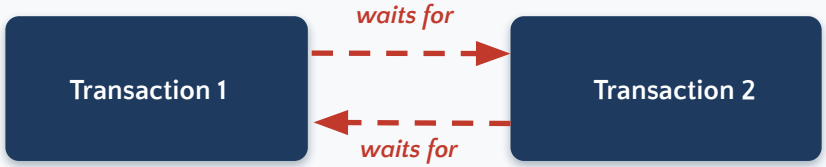
Bulk inserts/updates make each step fast but many concurrent writers touching shared rows is where trouble begins.

Result: Deadlocks

Concurrent bulk writers block each other on the same rows. Something has to give.

What Exactly Is a Deadlock?

- Two or more transactions each **hold a lock** the other one needs, and **wait forever**.
- Postgres uses **row-level + table-level locks**, acquired automatically as you run.
- Locks are held until COMMIT or ROLLBACK: there is no manual unlock of a row lock.
- Waiting is normal; **a cycle in the waits** is a deadlock.
- Postgres does **not prevent** deadlocks — it only **detects** and breaks them (by killing one transaction).



The classes of DB deadlocks

FTS
File Transfer Service

1 Locking Order Deadlocks

- Wrong `t_queue` row-order
- Wrong `t_file` row-order
- Wrong cross-table order

2 BEFORE TRIGGER Deadlocks

- Trigger on `t_file`
- Modifying `t_queue`
- Creates implicit lock contention

3 UNIQUE CONSTRAINT Deadlocks

- Implicit locks on `t_file` and `t_queue`
- Acquired in non-deterministic order
- Unique index contention

4 Cancellation Deadlocks

- Wrong `t_queue` row-order
- During transfer cancellation
- Cross-transaction contention

5 `get_transfers_to_start()`

- Wrong `t_queue` row-order
- In the work-pulling query
- Scheduler selecting transfers

6 `schedule_next_replica()`

- Wrong `t_queue` row-order
- In replica scheduling
- Multi-source job contention

Table-level deadlock: two tables, opposite order

FTS
File Transfer Service

Scenario: every UPDATE locks the target table (ROW EXCLUSIVE table lock) plus the rows it touches. Two transactions touch t_a then t_b — in opposite order.

Time	Transaction One	Transaction Two
t1	`UPDATE t_a ...` ✅ `lock on t_a rows`	`UPDATE t_b ...` ✅ `lock on t_b rows`
t2	`(work, other statements ...)`	`(work, other statements ...)`
t3	`UPDATE t_b ...` ⏳ `blocked by 'Two'`	`UPDATE t_a ...` ⏳ `blocked by 'One'`
t4	`waits for 'Two' ...`	`waits for 'One' ...`
t5	💀 `deadlock detected — one is aborted`	💀 `deadlock detected — one is aborted`

At t3 both are stuck waiting on each other — the wait-for graph has a cycle.
Rule of thumb: always touch shared tables in the same canonical order.

INSERT ... ON CONFLICT: how it locks with unique constraint

FTS
File Transfer Service

Both transactions upsert different rows, then each touches the row the other inserted — each waits on a **row the other holds**.

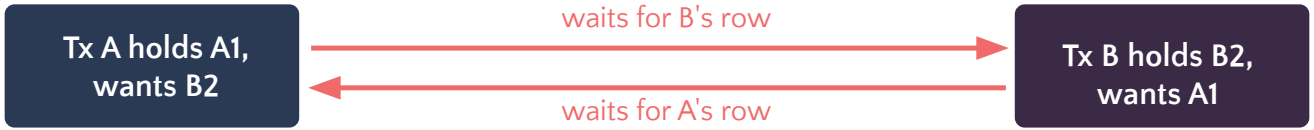
```
-- Tx A
INSERT INTO t_queue (id, count)
VALUES ('A1', 1)
ON CONFLICT (id) DO UPDATE
  SET count = t_queue.count + 1;
-- owns row A1 ✅

INSERT INTO t_queue (id, count)
VALUES ('B2', 1)
ON CONFLICT (id) DO UPDATE
  SET count = t_queue.count +1;
-- owned by B → waits ⏳
```

```
-- Tx B
INSERT INTO t_queue (id, count)
VALUES ('B2', 1)
ON CONFLICT (id) DO UPDATE
  SET count = t_queue.count + 1;
-- owns row B2 ✅

INSERT INTO t_queue (id, count)
VALUES ('A1', 1)
ON CONFLICT (id) DO UPDATE
  SET count = t_queue.count + 1;
-- owned by A → waits ⏳
```

Result: wait-for graph



INSERT ... ON CONFLICT: how it locks with unique constraint

- INSERT with a **unique / PK constraint** must check for duplicates before inserting.
- For each conflicting key, the row insert takes a **speculative insertion** lock on the index entry.
- ON CONFLICT DO UPDATE then locks the **existing row** in that table (FOR UPDATE-like).
- So one statement can lock **two things**: the new (speculative) tuple and the pre-existing conflicting row.
- Two concurrent transactions inserting the **same key** create the classic mirrored-wait deadlock.
- Same deadlock appears with plain INSERT racing an UPDATE on the same key, or two inserts of the same new key.

What one statement locks

1. Unique index: speculative insert lock (new key)
2. Conflict found → lock the EXISTING row (DO UPDATE)
3. Both held until COMMIT

The unique constraint is the meeting point where two writers on the same key are forced to serialize.

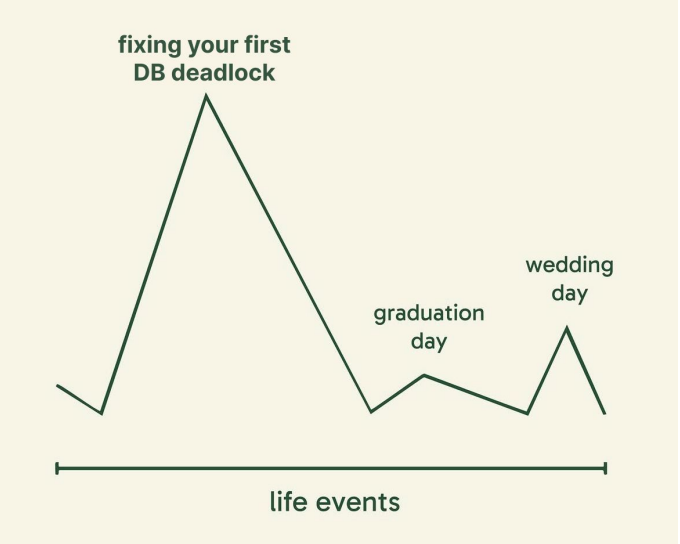
Fixing deadlocks

- **Surfaced During ATLAS & CMS Testing**

All deadlocks identified and fixed
BEFORE TRIGGER, UNIQUE CONSTRAINT, cancellation, get_transfers_to_start(), schedule_next_replica()

- **Deadlock Avoidance Strategy**

Use the **same row-order everywhere** · Decouple unrelated locks · Replace multi-threaded tasks with single-threaded ones · Decouple the architecture further



Deadlock-Free Lock Ordering

A transaction locking more than one row must always take locks in the same order

Golden rule: consistent lock order ⇒ no circular wait ⇒ no deadlock

t_file
INSERT · UPDATE
```
ORDER BY file_id
```

t_queue_submitted
UPDATE
```
ORDER BY queue_id
```

t_queue_submitted
INSERT
```
ORDER BY vo,
 source_se,
 dest_se,
 activity
```

t_job
UPDATE · SELECT FOR UPDATE
```
1 row per COMMIT
multi-row:
ORDER BY job_id
```

Cross-table transactions — lock ALL rows of the first table before ANY row of the second:

t_queue_submitted + t_file
```
1. t_queue_submitted  ←  ALL locks first
2. t_file  ←  then ALL locks
```

t_file + t_job
```
1. t_file  ←  ALL locks first
2. t_job  ←  then ALL locks
```

Cross-table rules are provisional: they may change if keeping the order requires unjustified code or complexity.

Triggers to Update Counters

Approach: let the database do the bookkeeping. Every INSERT/UPDATE fires a trigger that keeps counters up to date automatically.

...repeated for every single file in the batch.

PROBLEM — HUGE OVERHEAD

Submit 100 files → trigger fires 100 times → 100 separate **+1** counter updates. Per-row work multiplied across the whole batch.

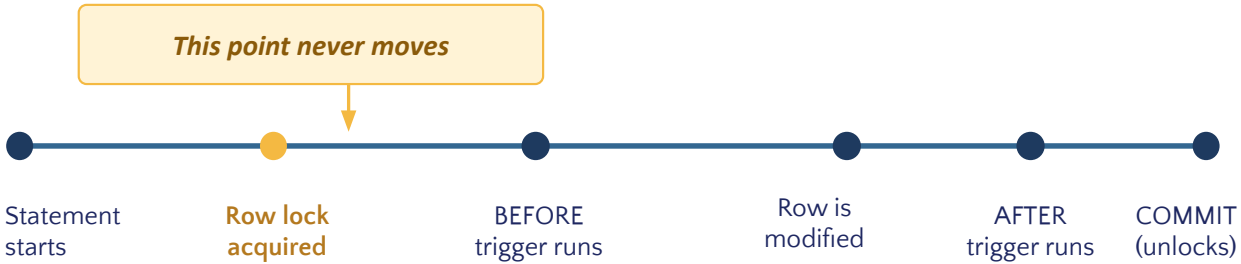
PROBLEM — DEADLOCKING

100 writers all race to lock and update the **same counter rows** at the same time. Locks pile up, transactions wait, deadlock again.

Solution: disable all the triggers with a session variable - triggers now are only for human users of the db

BEFORE vs. AFTER Triggers Same Deadlock Risk

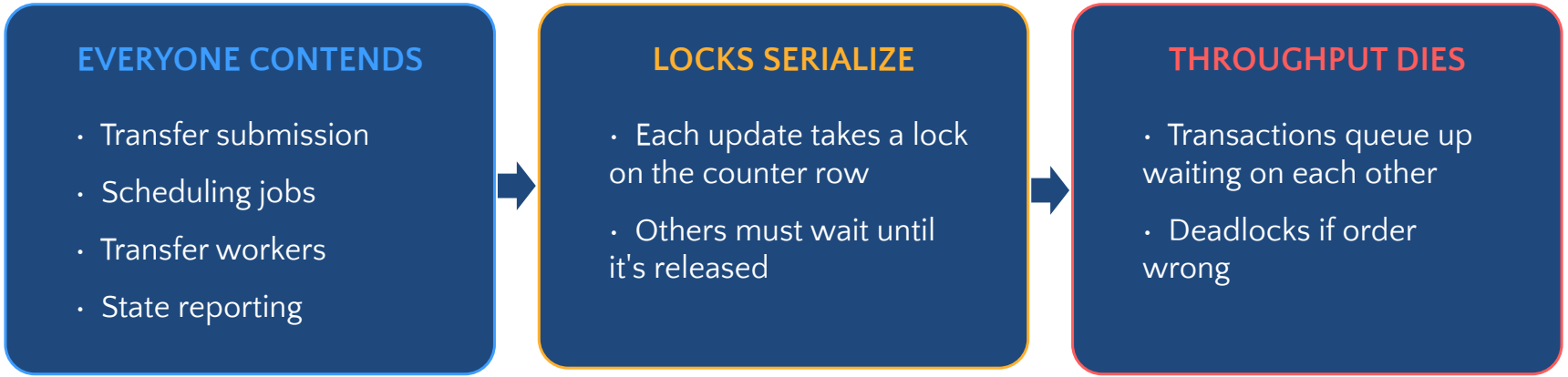
- The row lock is taken by the UPDATE/INSERT/DELETE itself, before any trigger runs. **Postgres must lock the row to fetch and validate it.**
- A **BEFORE** trigger fires after that lock is already held; an **AFTER** trigger fires later in the same command, with the lock still held the whole time.
- Switching BEFORE ↔ AFTER **only moves the trigger's own logic earlier** or later inside that already-locked window.
- It never changes when the row lock is acquired or released, **so it never changes what the trigger can deadlock with**.
- Any extra lock the trigger itself takes (e.g. on a second table) happens at the same relative point either way.



The Bottleneck: shared queues counters

FTS
File Transfer Service

Every writer must update the same counters. (number of submitted files)



Takeaway: hot, shared mutable counters are a single point of contention

Contention & double queues

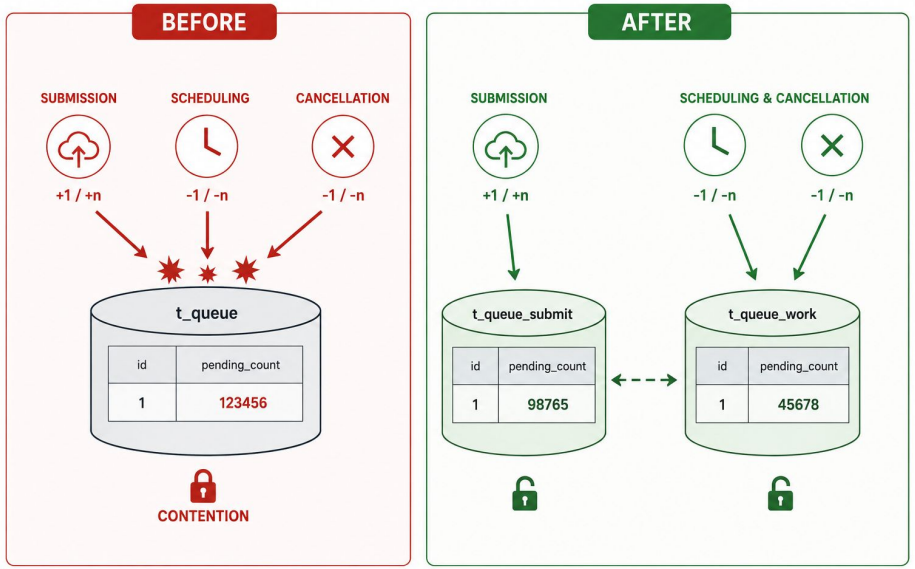
Discovered during ATLAS & CMS load testing: Contention between job-submission and scheduling

The Problem

- Scheduler logic updates the same counters as the submission/canceling
- Load and locks taken on submission can slow down scheduling
- Real-time scheduling is only as fast as the **slowest DB operation**

The Solution

- Multiple queues so that each process has his own table



Recap: solutions we implemented

- **deadlocks**: find and remove all the deadlocks
- **triggers overhead**: disable all the triggers using a session variable - triggers now are only for human users of the db
- **queue counter bottleneck**: remove unnecessary counters (ACTIVE transfers are in memory anyway) and de-normalize the remaining one (SUBMITTED)

The FTS FlowController

A new dedicated component that replaces the retired FTS3 Optimizer, delivering automatic concurrency tuning after DC27.

Protect overloaded storages

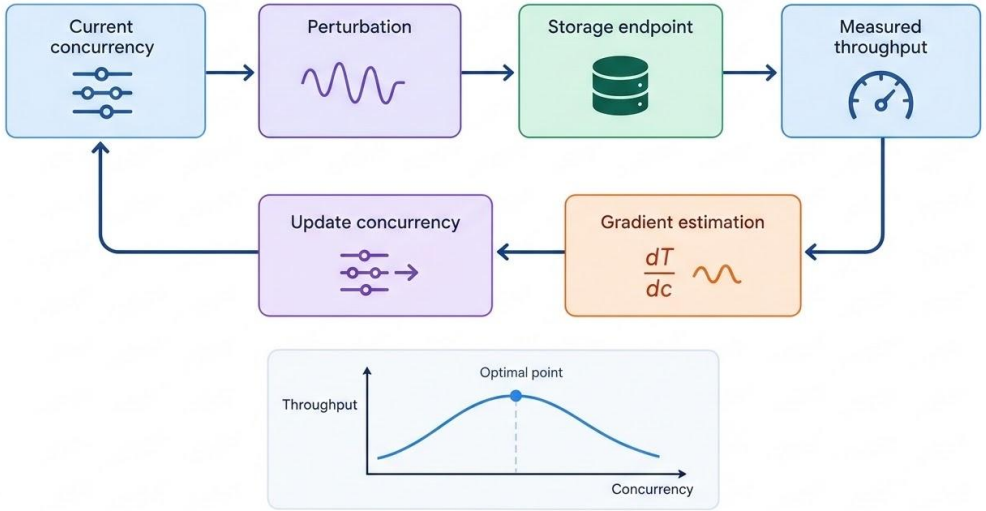
Watches ongoing transfers and backs off concurrency immediately when a storage endpoint shows errors or overload, before the situation compounds.

Auto-tune unbounded limits

For storage endpoints with no static concurrency limit configured, the FlowController continuously adjusts the ceiling itself, removing the need for manual expert tuning.

Design approach

Loosely based on Extremum Seeking Control (ESC): a model-free feedback method that continuously nudges concurrency toward the operating point of maximum throughput, adapting online without needing an explicit model of the storage endpoint.



Thank you !

home.cern