

SQL Traffic Routing Without Breaking Consistency

Nerdearla 2026 · Buenos Aires

René Cannaò · ProxySQL

Who am I?



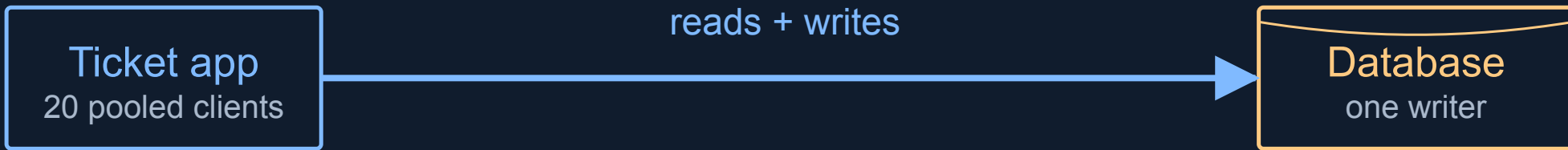
René Cannaò

Author and CEO of ProxySQL

Building ProxySQL since 2013

Over two decades working with systems,
networks, and database performance

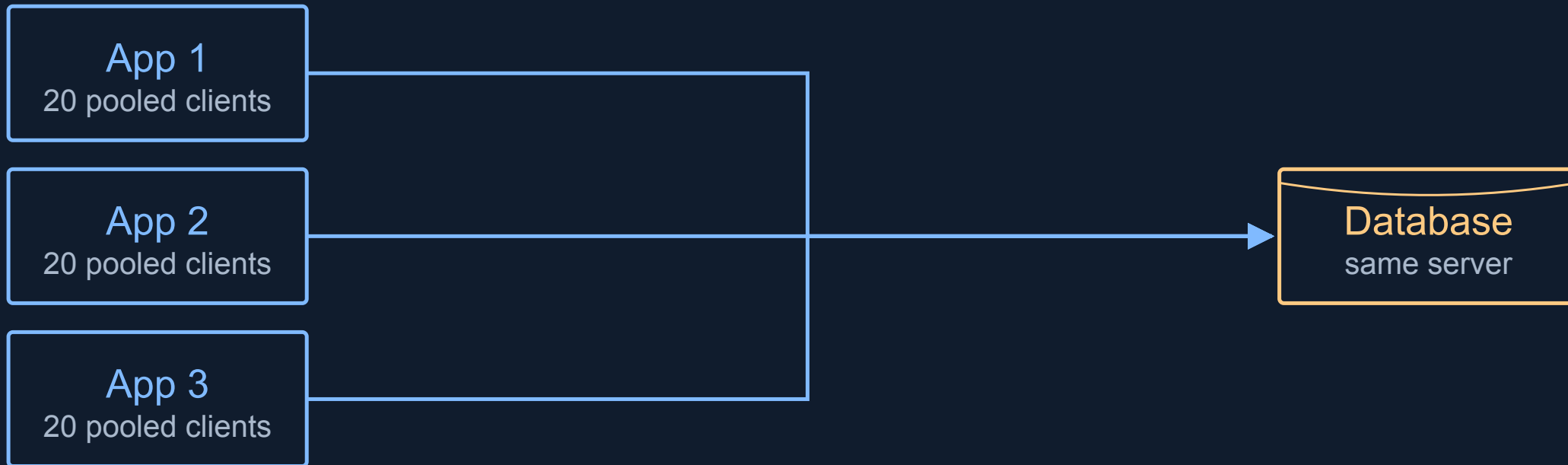
We launch a ticket-booking service



A customer books a ticket, then opens the confirmation.

Book a ticket. Commit. Show the confirmation.

More customers. More application instances.



The application scales out. Every instance brings its own pool.

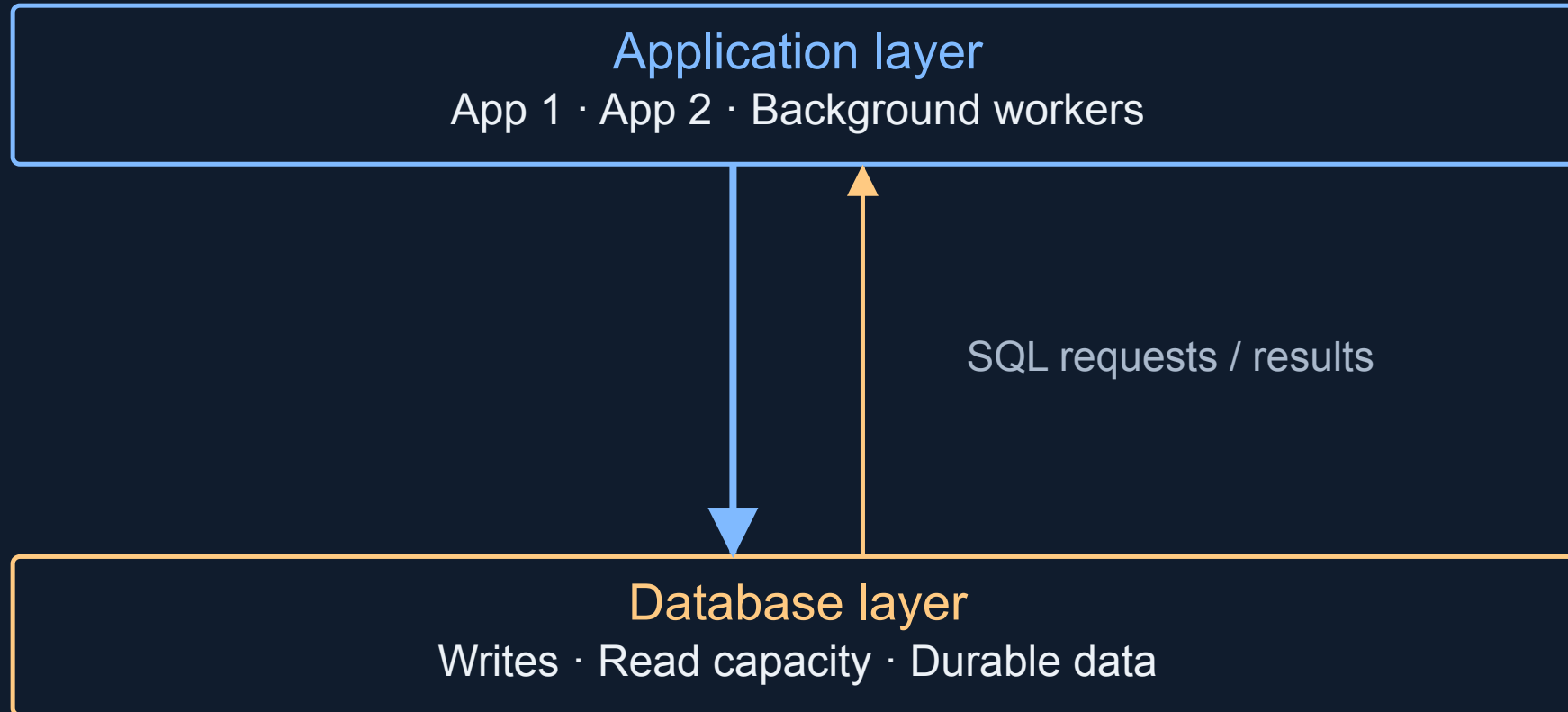
Application pools add up



Illustration: 20 open connections per instance; not a benchmark

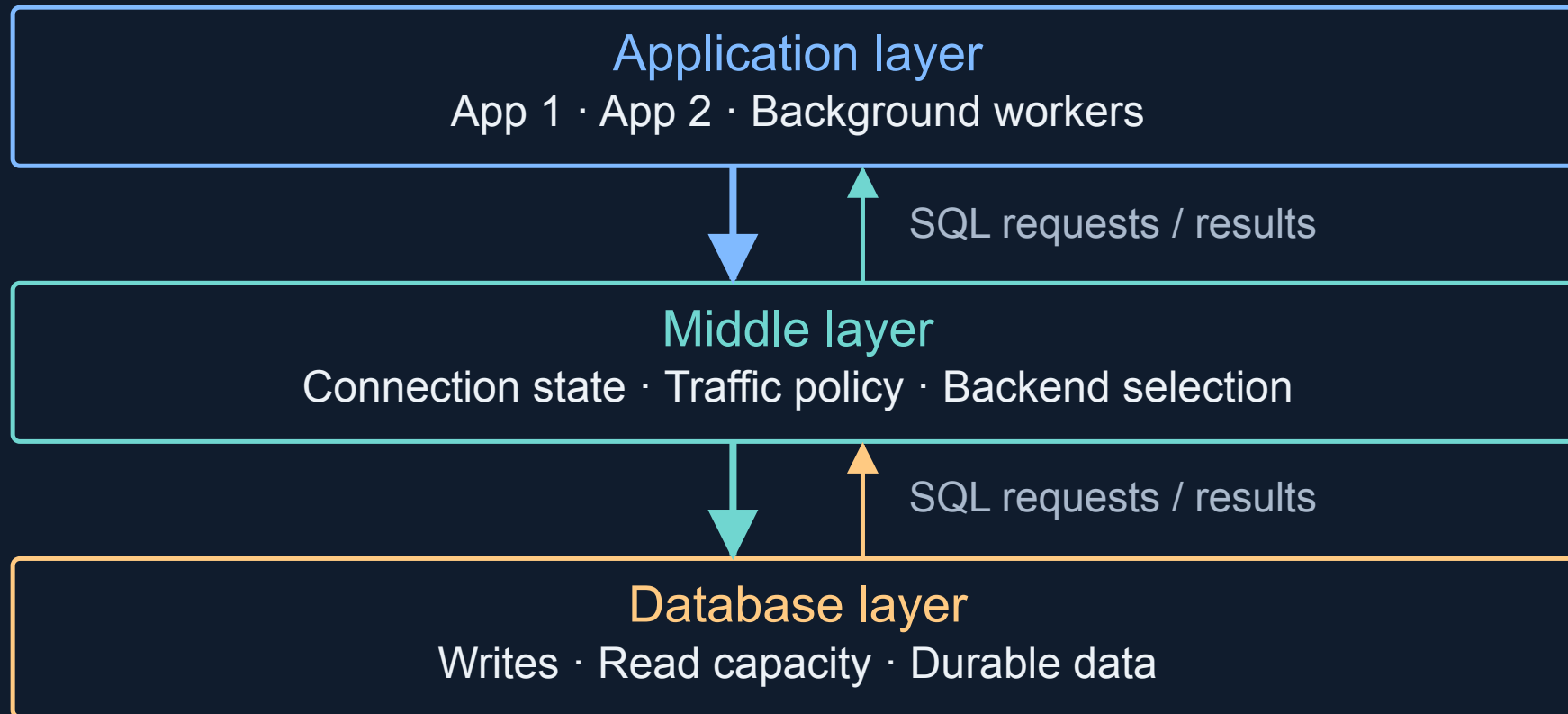
Open sessions and concurrent SQL work are different quantities.

Two layers communicate directly



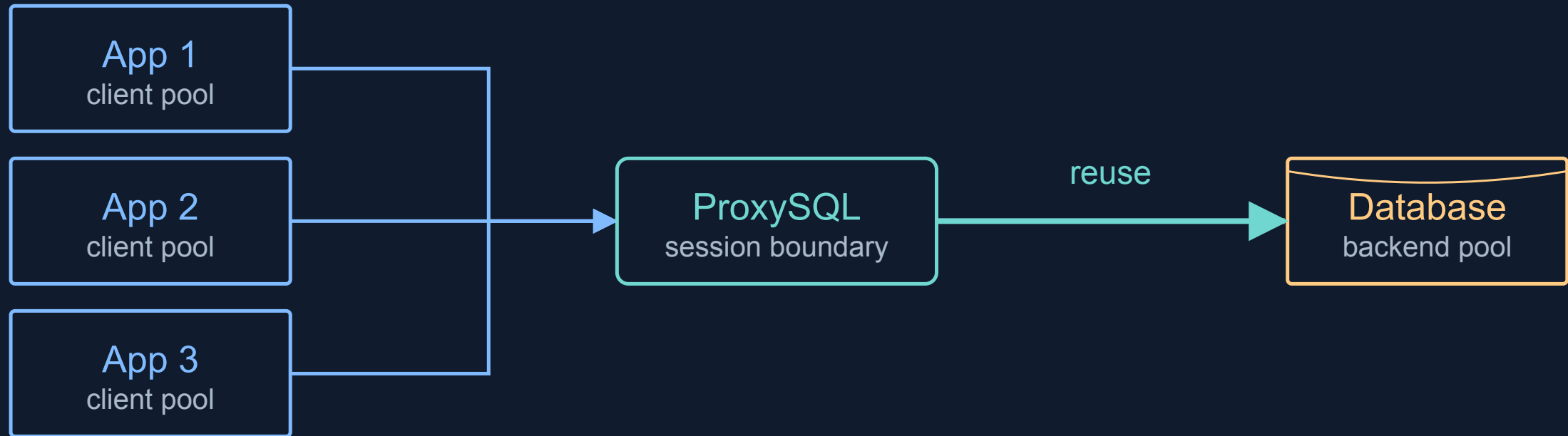
Every application pool connects straight to the database layer.

A middle layer manages the database conversation



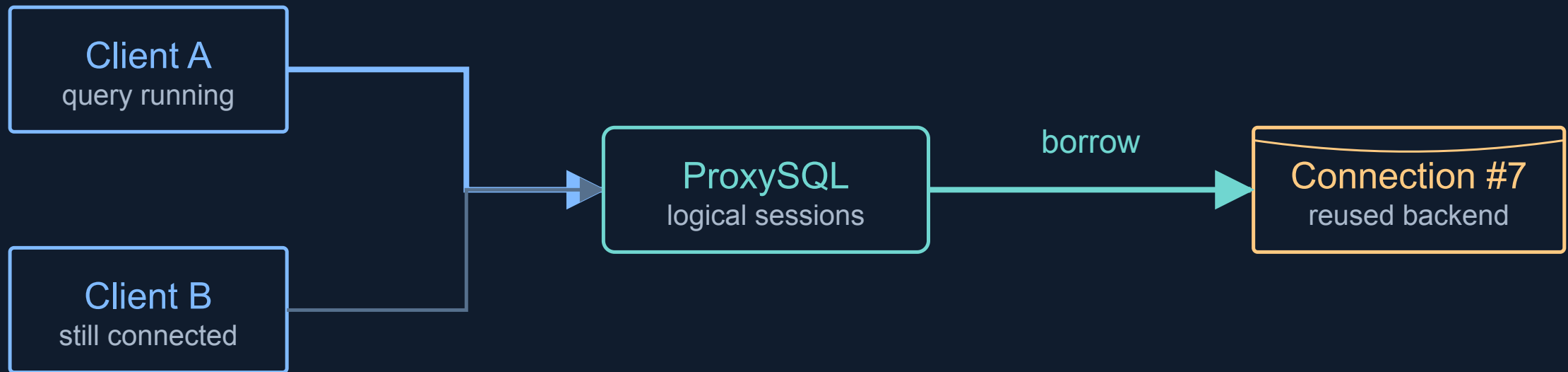
A shared place to manage connection state and traffic policy.

We introduce ProxySQL



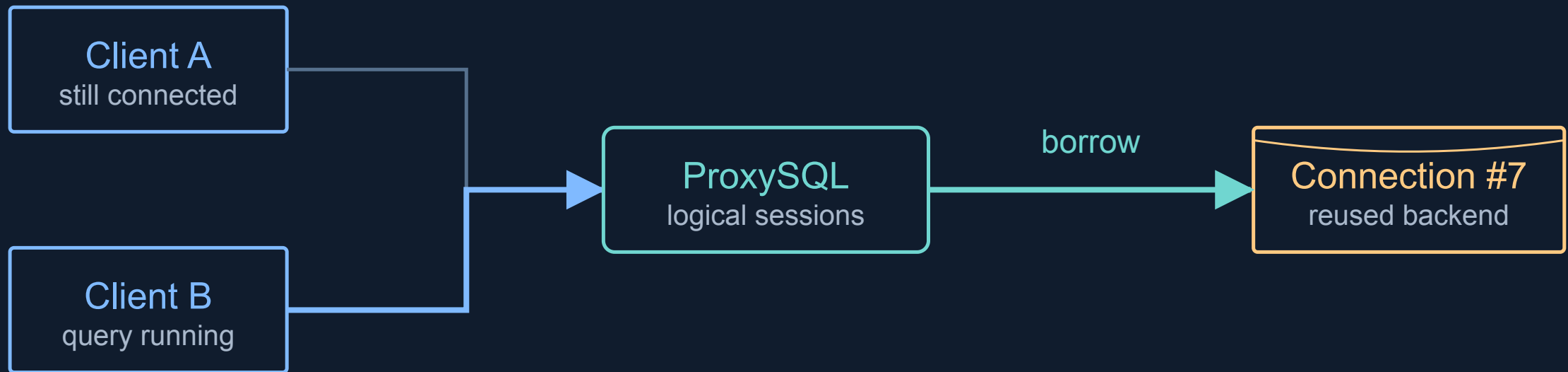
Frontend sessions can outlive the backend connections they borrow.

Client A borrows a backend



A query needs a backend. An idle client may not.

Then client B reuses it



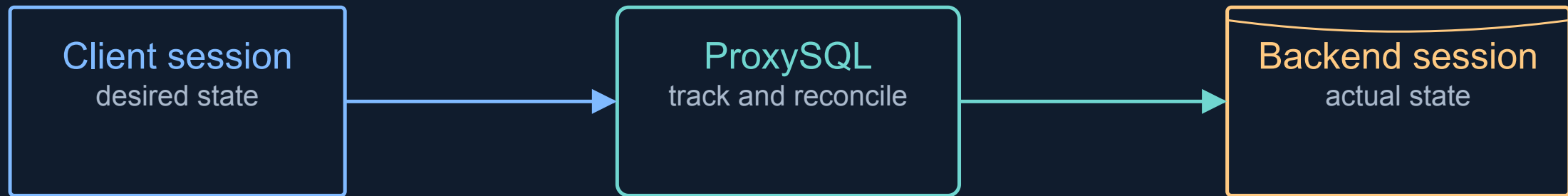
The client connections remain open; backend ownership changes.

Then A opens a transaction



BEGIN ... statements ... COMMIT/ROLLBACK stay in one backend context.

The application brings session state



Restore supported state before execution; retain affinity for other state.

Transactions, variables, temporary objects, locks, and protocol state

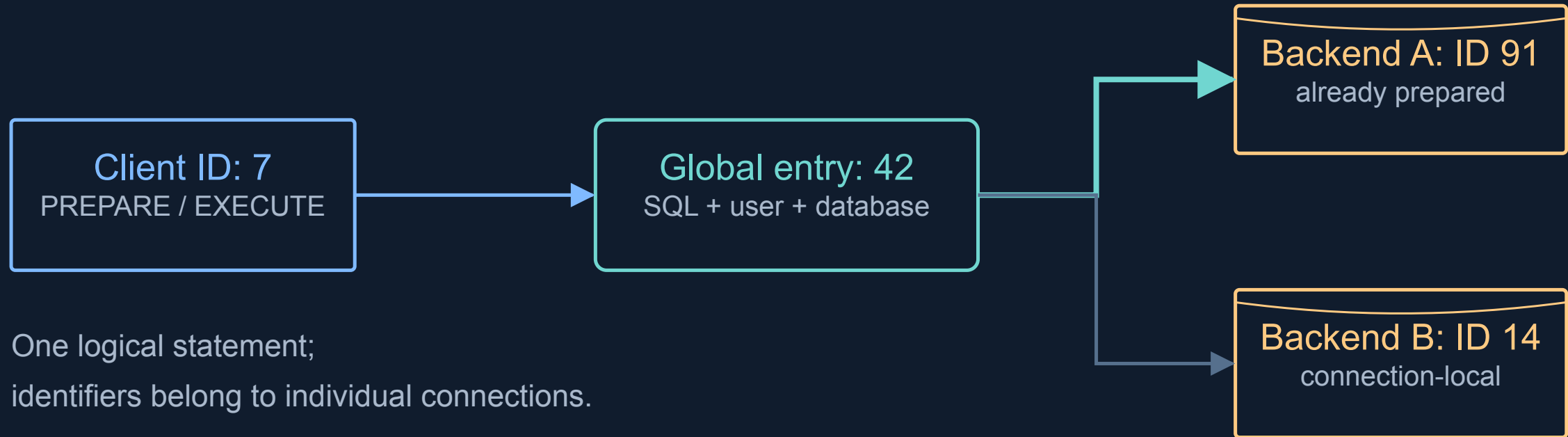
A reused connection needs the right variables



MySQL example; PostgreSQL has its own tracked settings and protocol state.

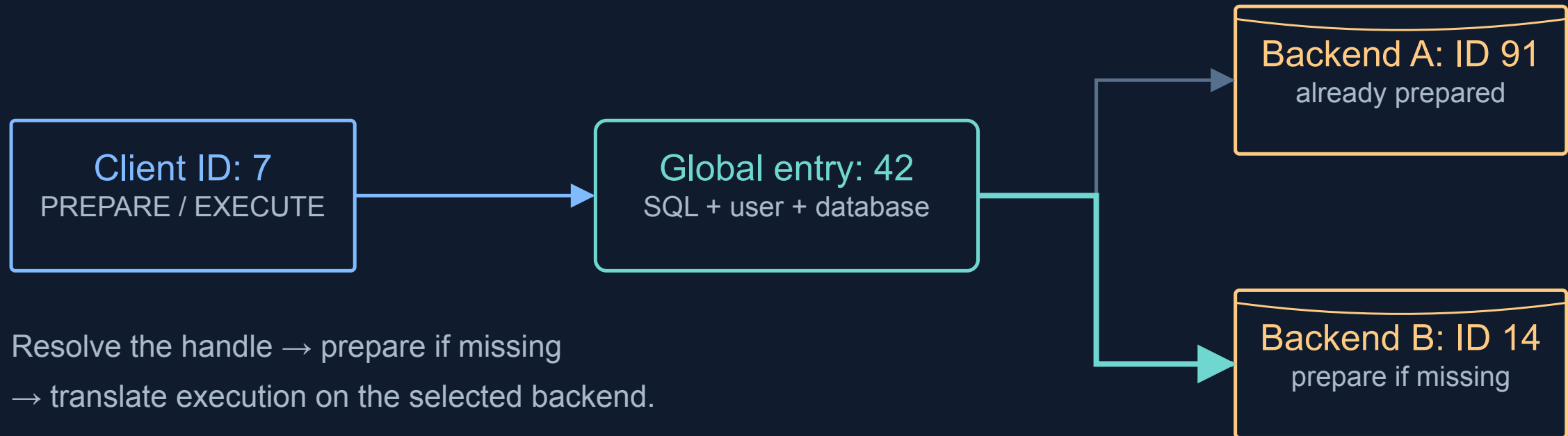
Track the supported setting; synchronize the backend before execution.

The driver prepares a MySQL statement



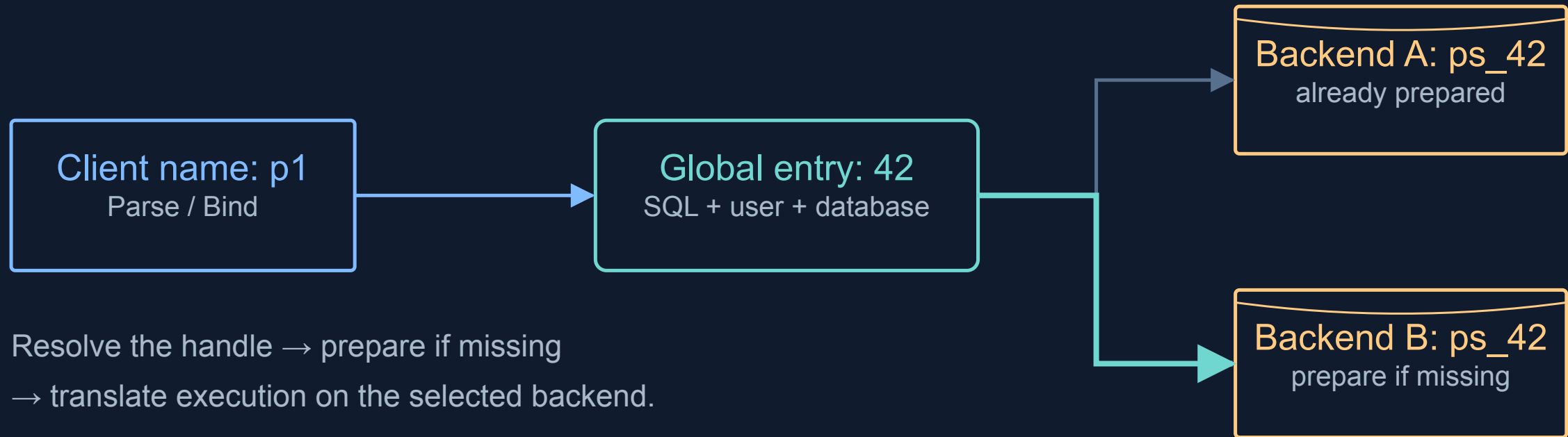
Client handle $\neq$ global statement entry $\neq$ backend handle

Execution reaches another backend



Prepare there if necessary, then translate the execution.

PostgreSQL adds Parse, Bind, portals, and Sync



A statement definition can be mapped; an in-flight execution still has state.

This is why an SQL proxy needs more context

A stateless HTTP deployment

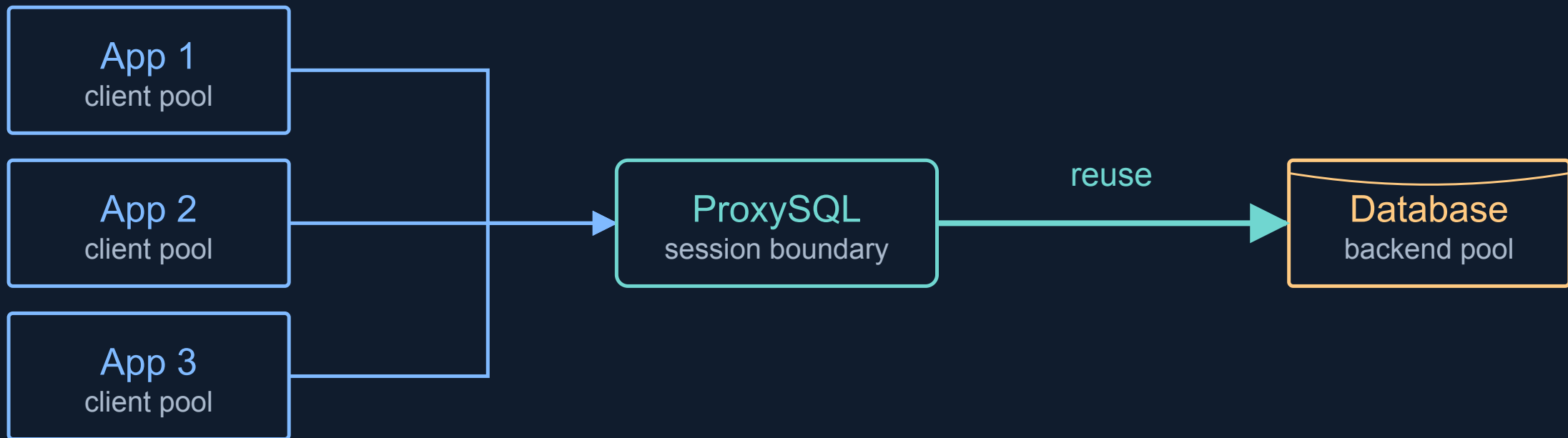
A request can carry enough context to use another worker.

A SQL session

The next message can refer to state already held by a backend.

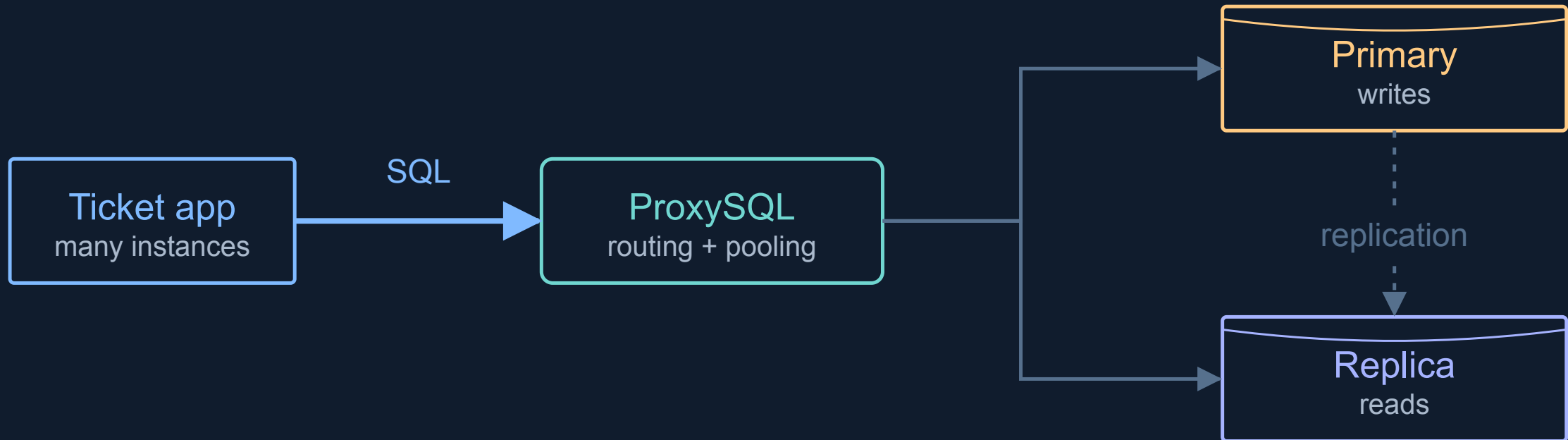
ProxySQL routes work while preserving the conversation.

Connections are under control. Reads keep growing.



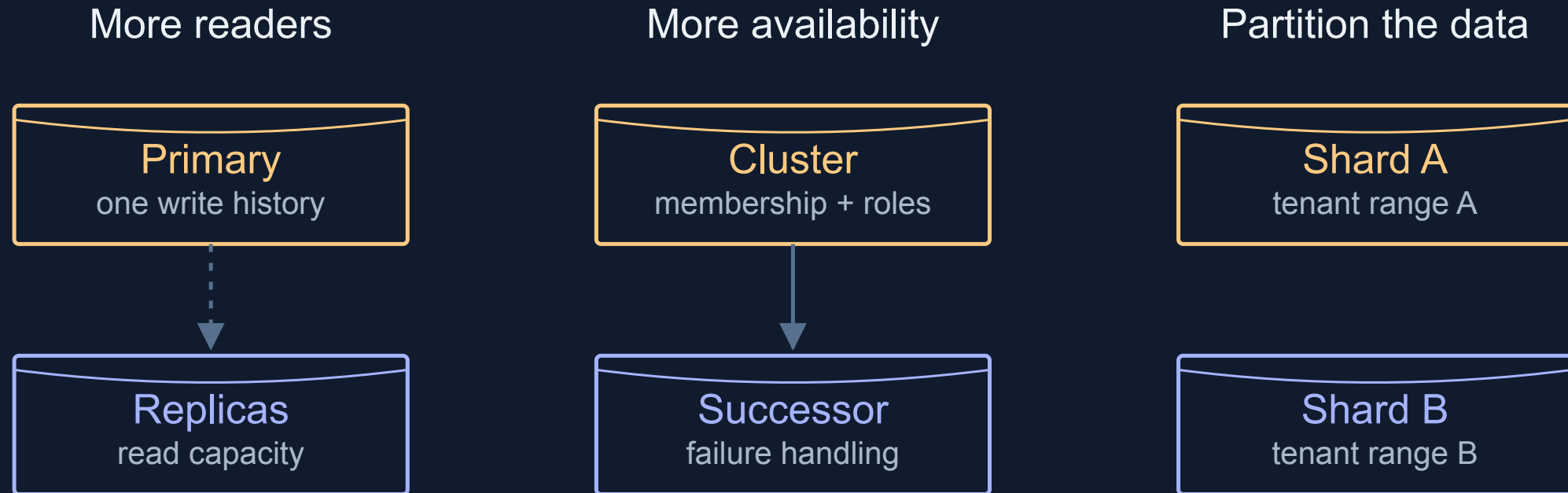
Pooling reuses connections. It does not create database execution capacity.

We add database readers



Now a destination decision can change what the customer sees.

Scale-out solves different problems

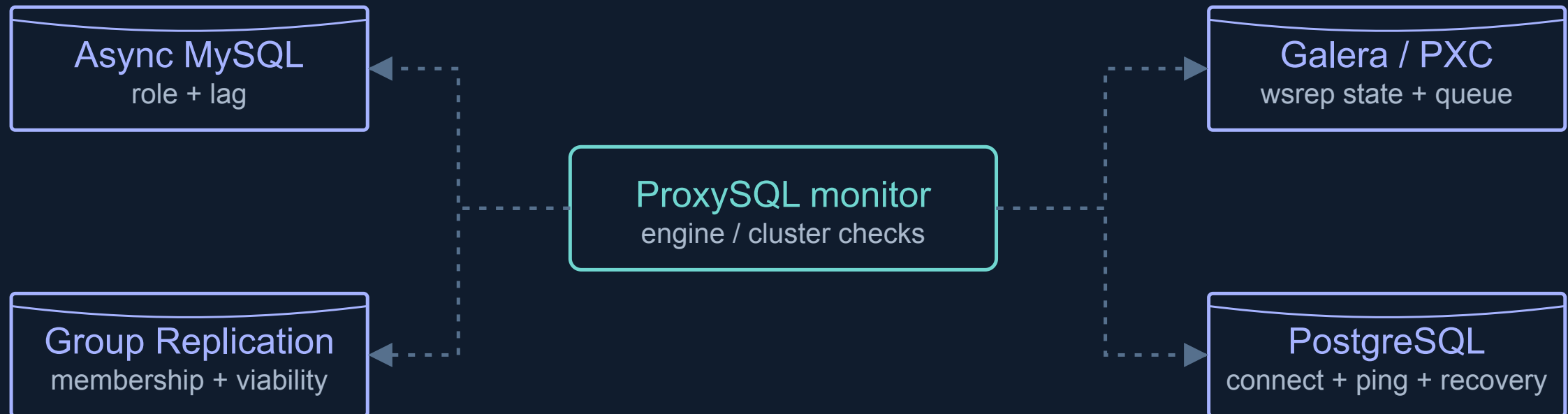


Read capacity, failover, and write/data partitioning are separate goals.

The cluster determines the meaning of “ready”

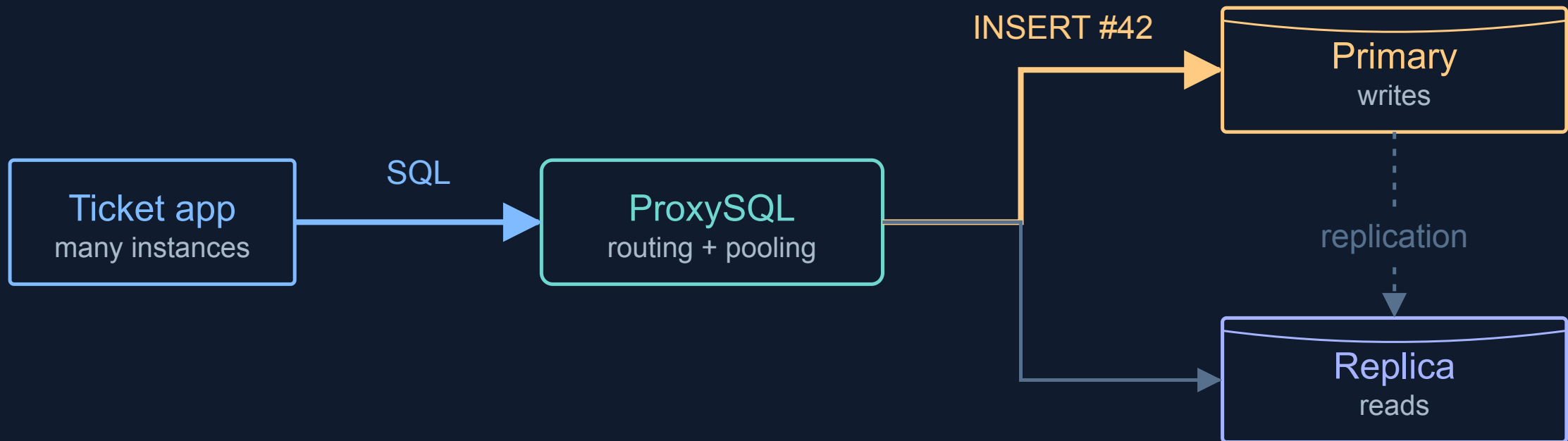
Architecture	What changes for routing?
MySQL async / semisync	Role and apply progress; receipt can precede execution
Group Replication / InnoDB Cluster	Membership, role, viable state, consistency settings
Galera / Percona XtraDB Cluster	Component and node state, receive queue, read guarantees
PostgreSQL streaming / Aurora MySQL	Engine-specific role and progress signals

ProxySQL monitors the topology it is routing



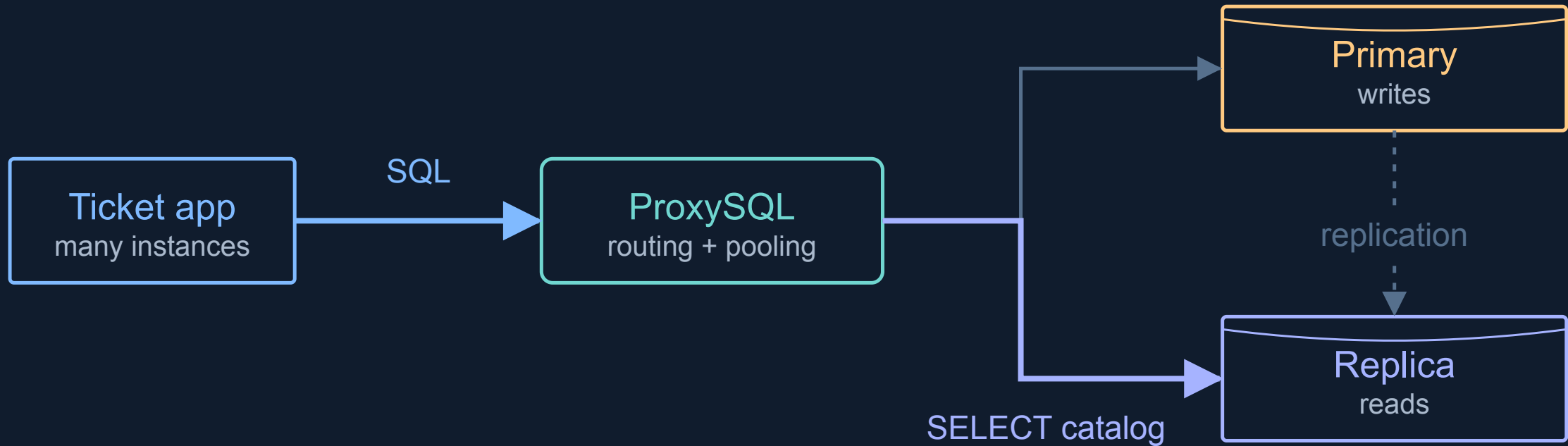
Connect / ping checks are only the beginning.

The usual routing rule: writes to the primary



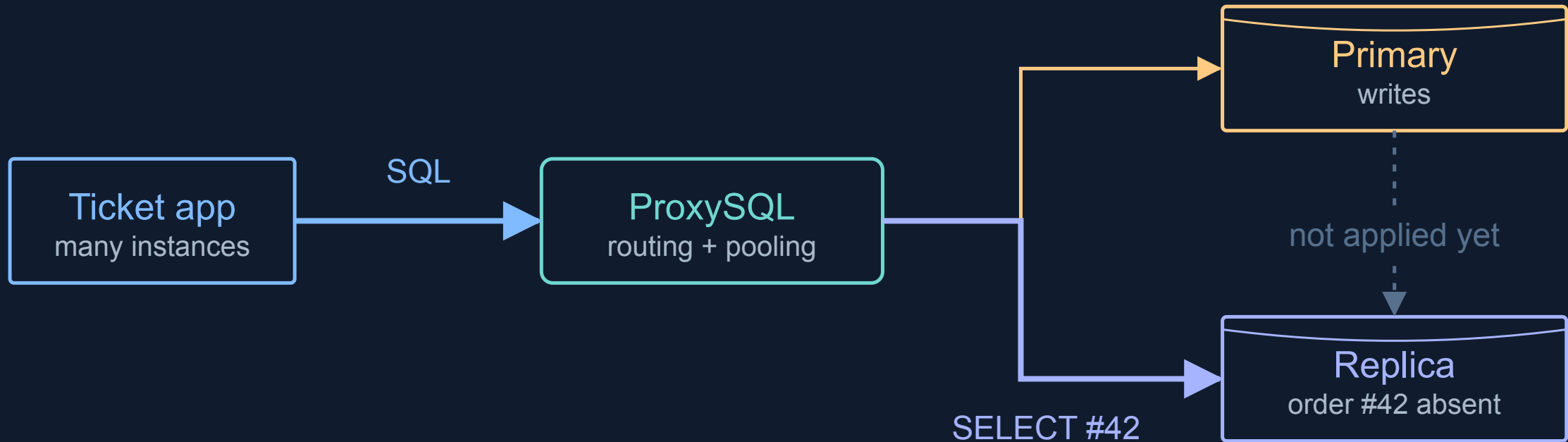
The booking is committed on the write authority.

...and eligible reads to readers



A catalog read can use a reader if its contract allows it.

The customer's confirmation disappears



COMMIT succeeded. Replication has not applied the booking yet.

The same SELECT can have a different contract

Application operation	Requirement
Browse the event catalog	Stale data may be acceptable
Show my booking confirmation	Include the booking I just committed
Continue browsing after seeing an update	Do not move backward to an older version
Reserve the last ticket	Preserve transaction and locking semantics

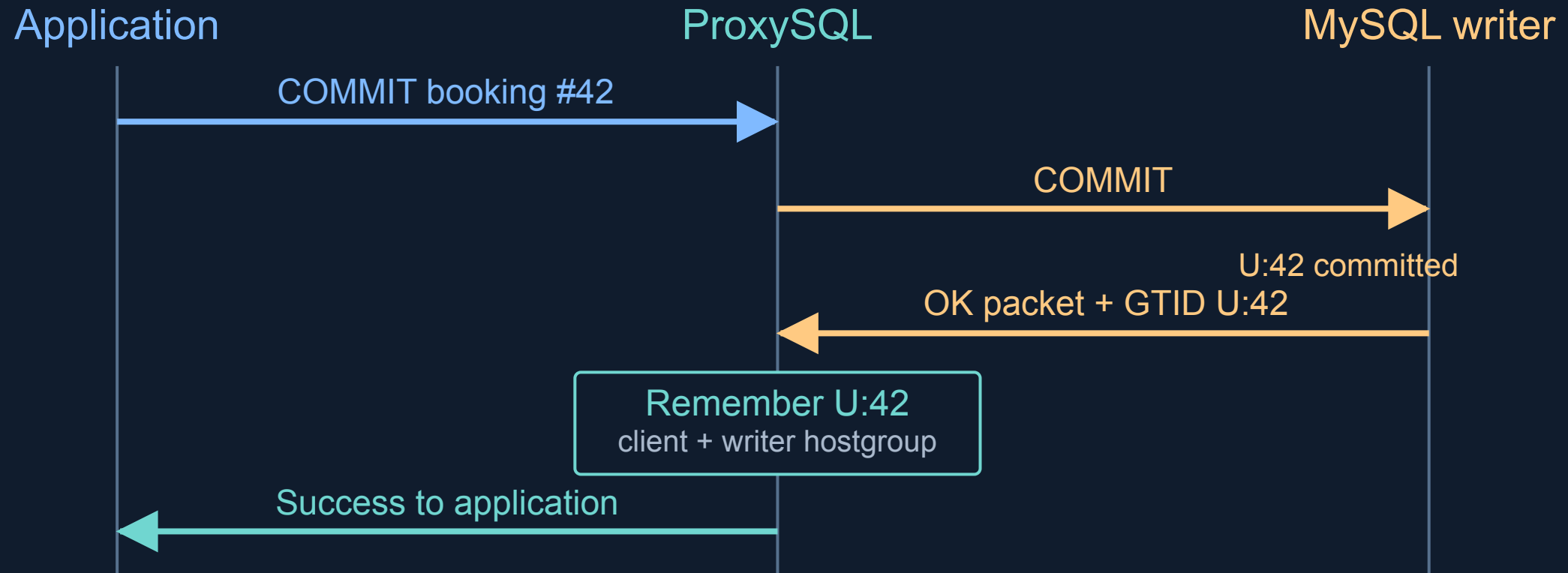
SQL classification helps. Application intent completes the policy.

We choose a consistency strategy per workflow

Strategy	Use and limitation
Writer for dependent reads	Simple baseline; writer capacity and a fresh snapshot matter
Sticky writer for a time window	Useful heuristic; elapsed time does not prove replay
Reader with a causal dependency	Select/check a reader known to include the required write
Explicitly allow stale data	Appropriate only when the product accepts that outcome

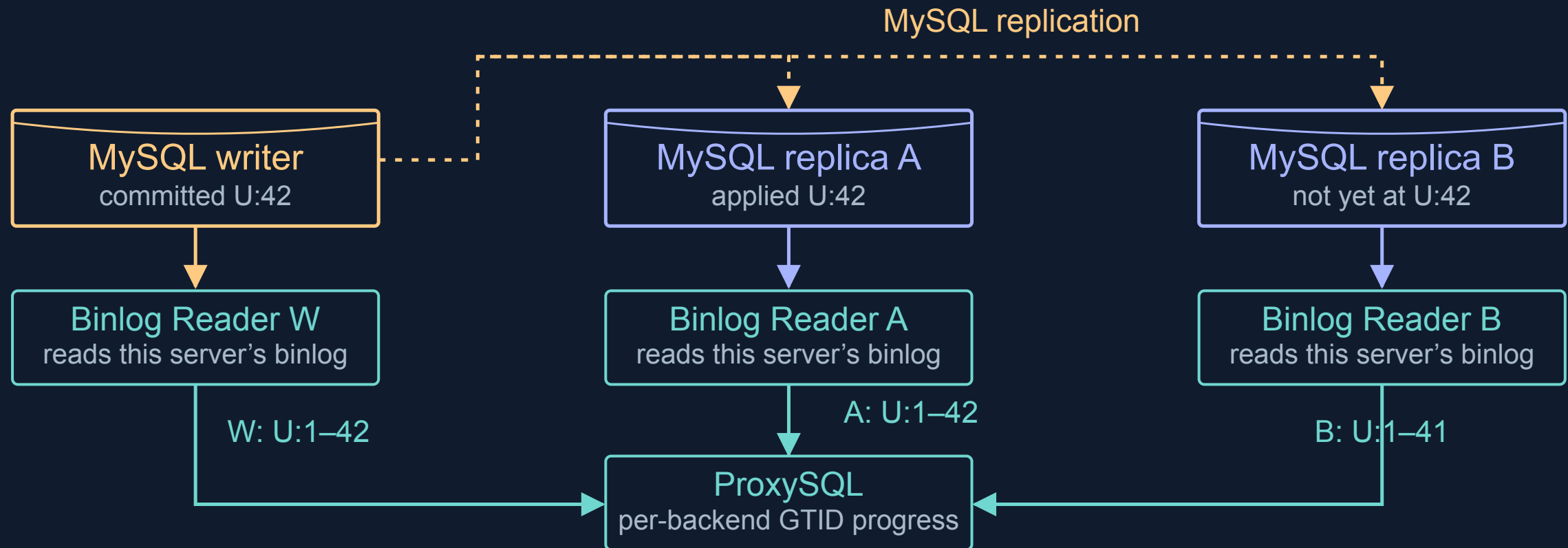
An unsatisfied dependency needs a deadline and an explicit fallback.

MySQL today: remember the write's GTID



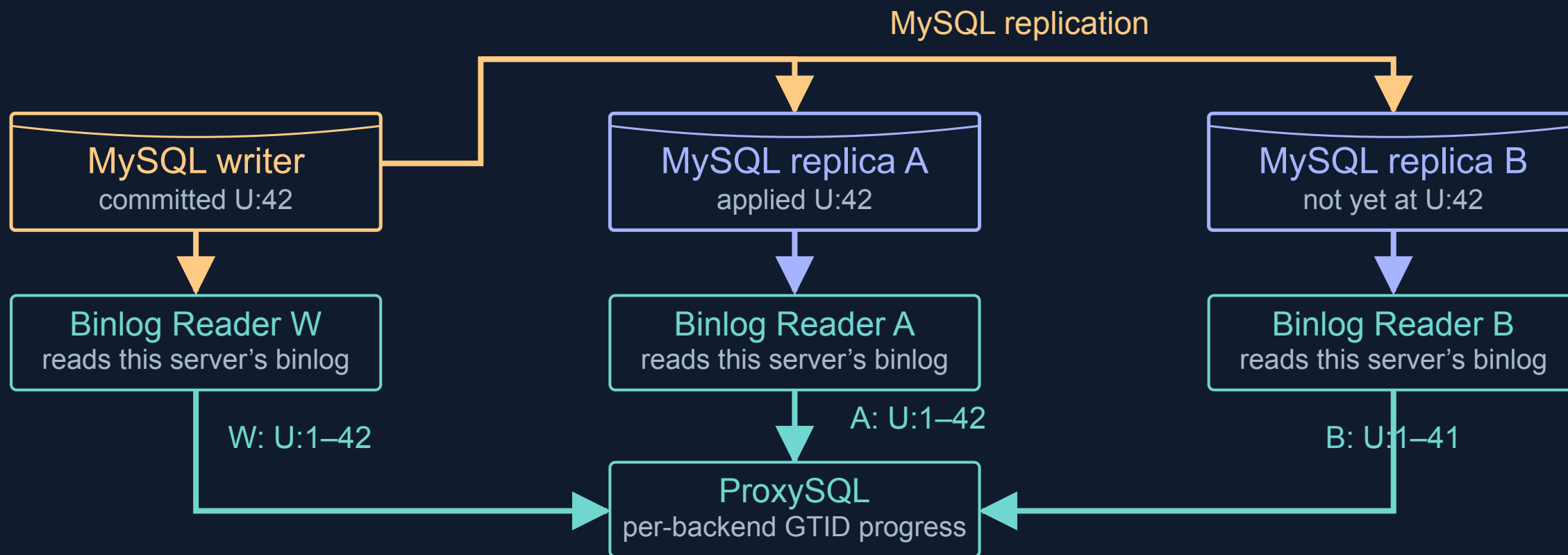
The successful response carries U:42; ProxySQL keeps the dependency.

Each MySQL backend has its own Binlog Reader



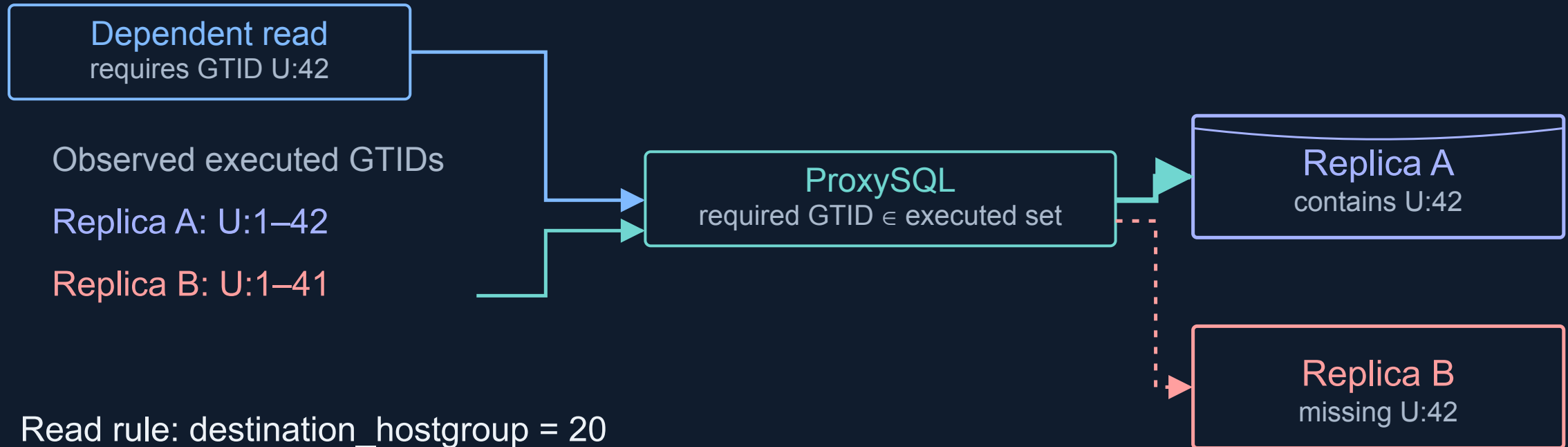
Replication moves data. Each reader observes its own server's binlog.

Every Binlog Reader streams progress to ProxySQL



ProxySQL knows the history reported by W, A, and B separately.

The read needs U:42. Replica A has it.

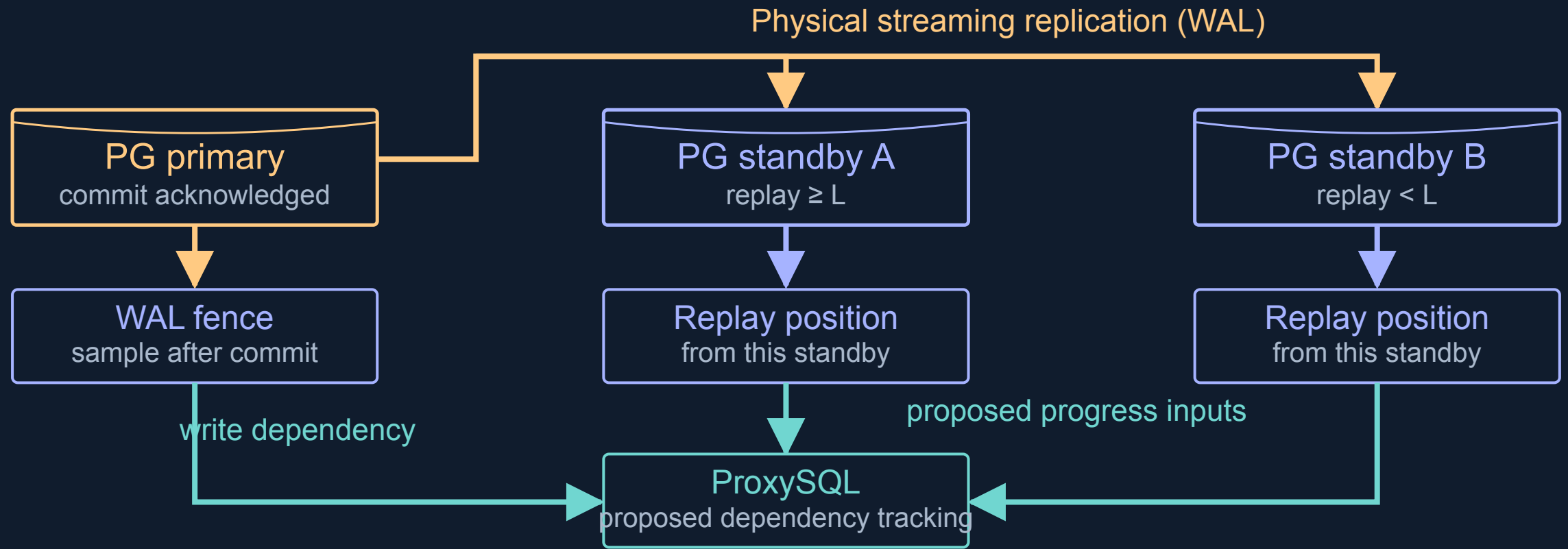


Read rule: destination_hostgroup = 20

gtid_from_hostgroup = 10 → use the write dependency from HG 10

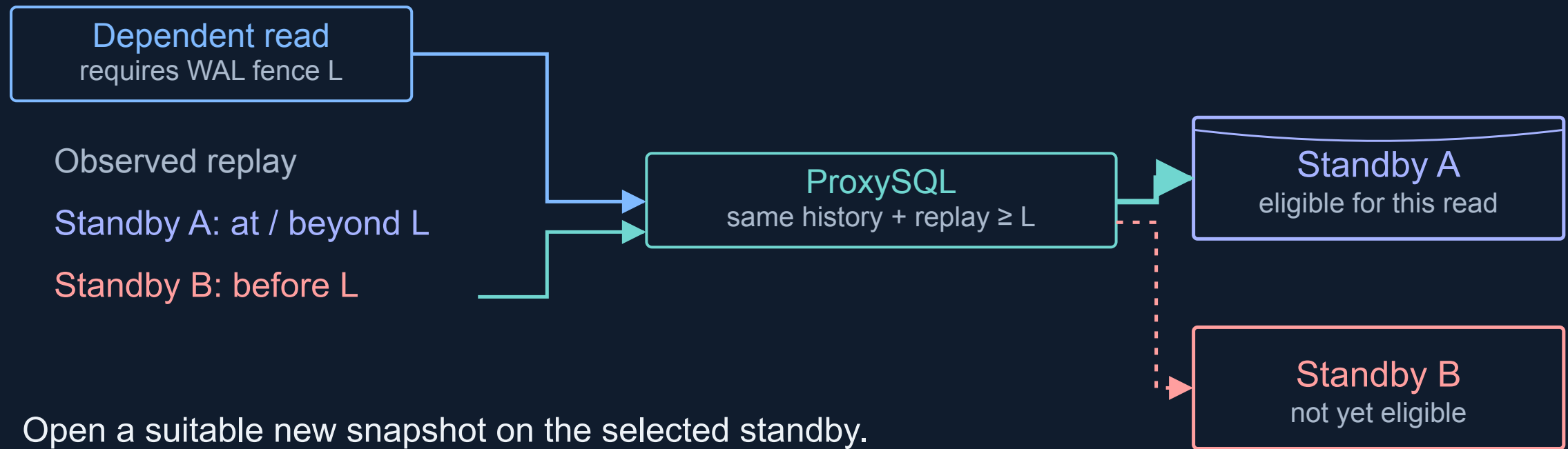
Client dependency + per-backend progress → an eligible destination

PostgreSQL: write dependency and replay progress



Planned direction · per-standby evidence · no release date

A PostgreSQL reader must replay the dependency first



Open a suitable new snapshot on the selected standby.

Cluster / timeline ancestry and deadlines remain part of the design.

Validate the history, check replay, then open the dependent snapshot.

Which traffic should we move to replicas?

The writer is busy.

We have reader capacity.

Which queries should move first?

“All SELECTs” does not tell us the benefit or the consistency cost.

Query shape, frequency, and total time reveal candidates



Illustrative workload · equal one-minute windows · elapsed query time, not CPU

The busiest shape and the most expensive call are not always the same query.

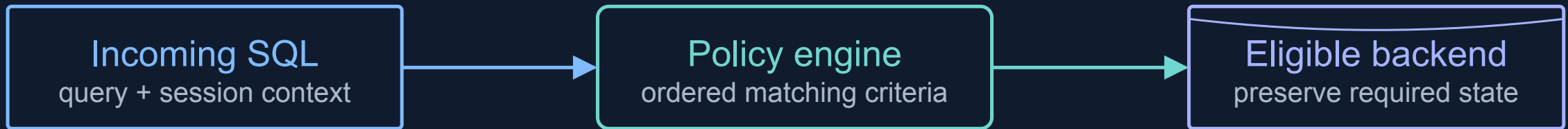
ProxySQL exposes the workload while it runs

```
SELECT hostgroup, digest, count_star,  
       ROUND(sum_time / 1000000.0, 2) AS total_seconds,  
       digest_text  
FROM stats_mysql_query_digest  
ORDER BY sum_time DESC LIMIT 5;
```

Find the query shapes, compare their frequency and accumulated time.

Admin interface · PostgreSQL: stats_pgsql_query_digest

We need a routing engine that understands the criteria



User / schema / digest / pattern → destination + behavior

The same query shape can require a different route in a different workflow.

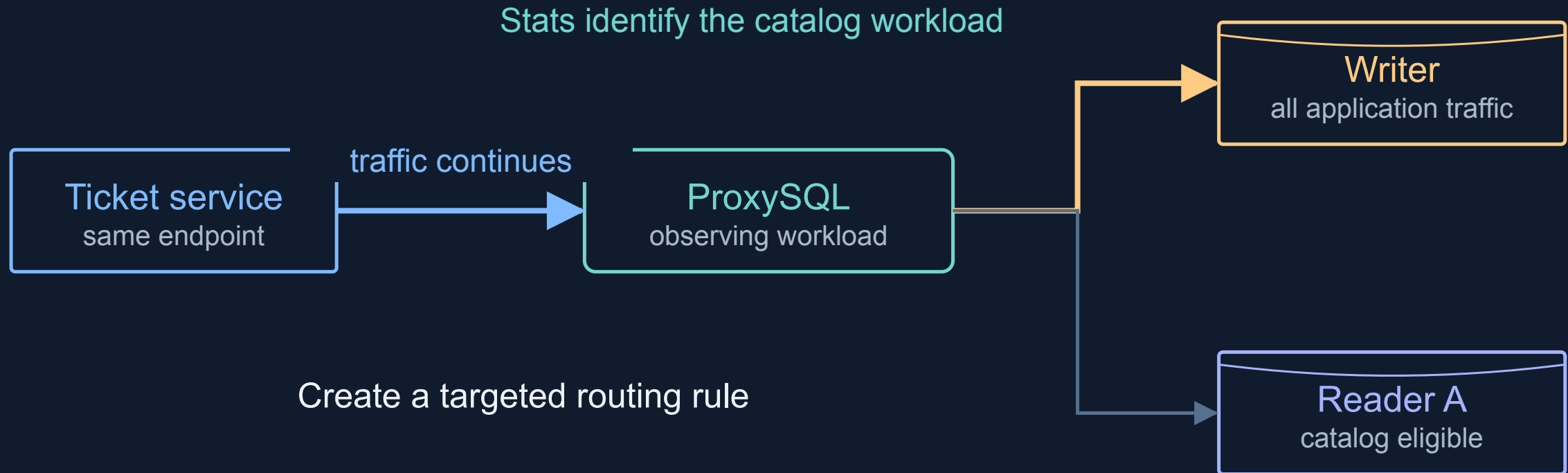
Recognize the workload on every request, then enforce its routing contract.

ProxySQL query rules implement the routing policy

Matching criteria	Routing decision
Catalog digest + catalog application user	Eligible reader hostgroup
Booking lookup + valid MySQL GTID context	Reader that contains the write dependency
Existing transaction or session dependency	Preserve the required backend context
Unclassified work	Compatible writer path

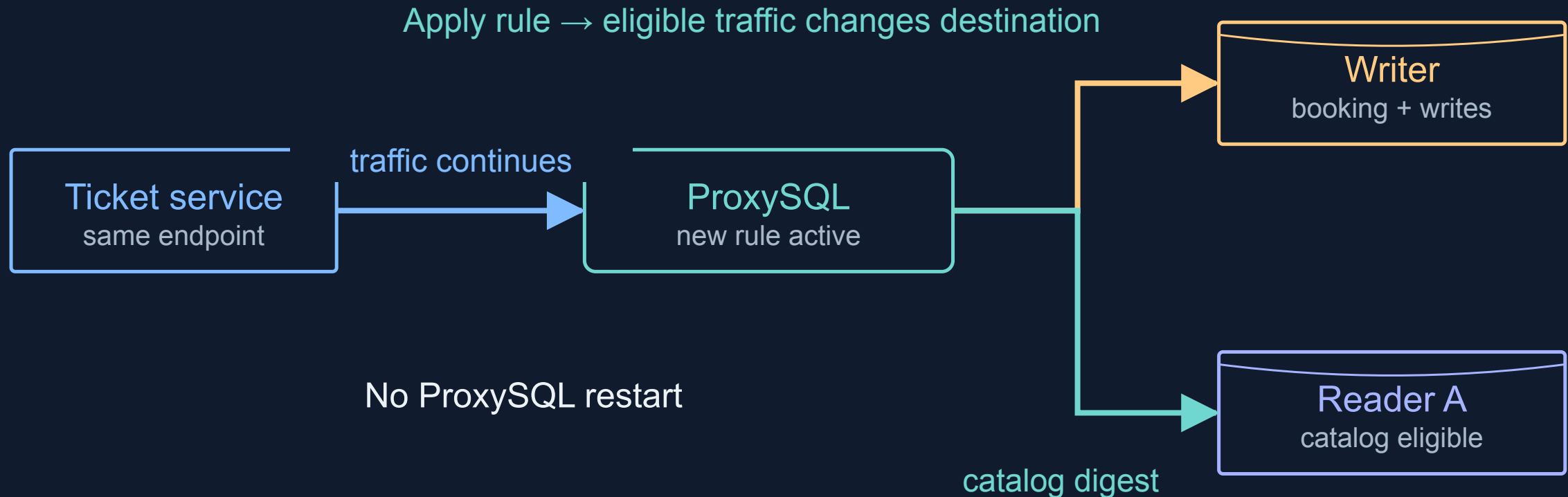
`mysql_query_rules` / `pgsql_query_rules` · ordered matching and actions

Traffic keeps running while we build the rule



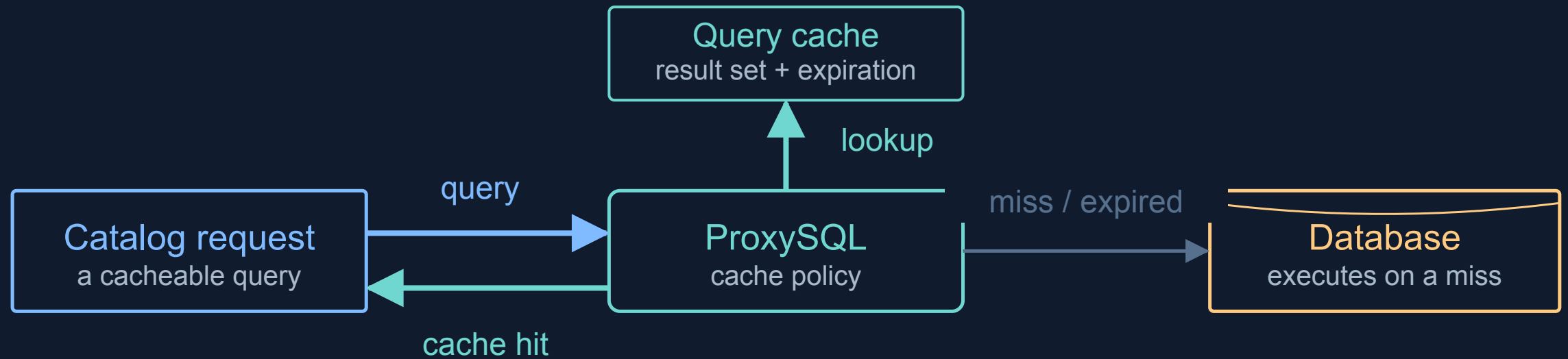
Observe the catalog digest → check its contract → create a targeted rule

Apply the rule. Catalog traffic moves immediately.



No restart. Newly matched, eligible reads now use the reader pool.

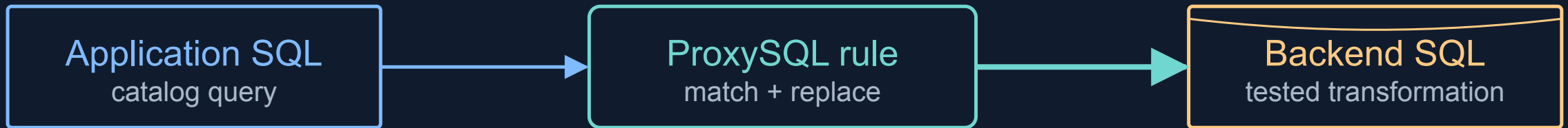
Query cache can answer a repeated catalog read



Repeated eligible read → cached result → no backend query on a hit

A cache hit avoids backend work. Its freshness policy must fit the read.

Query rewrite can transform SQL before execution

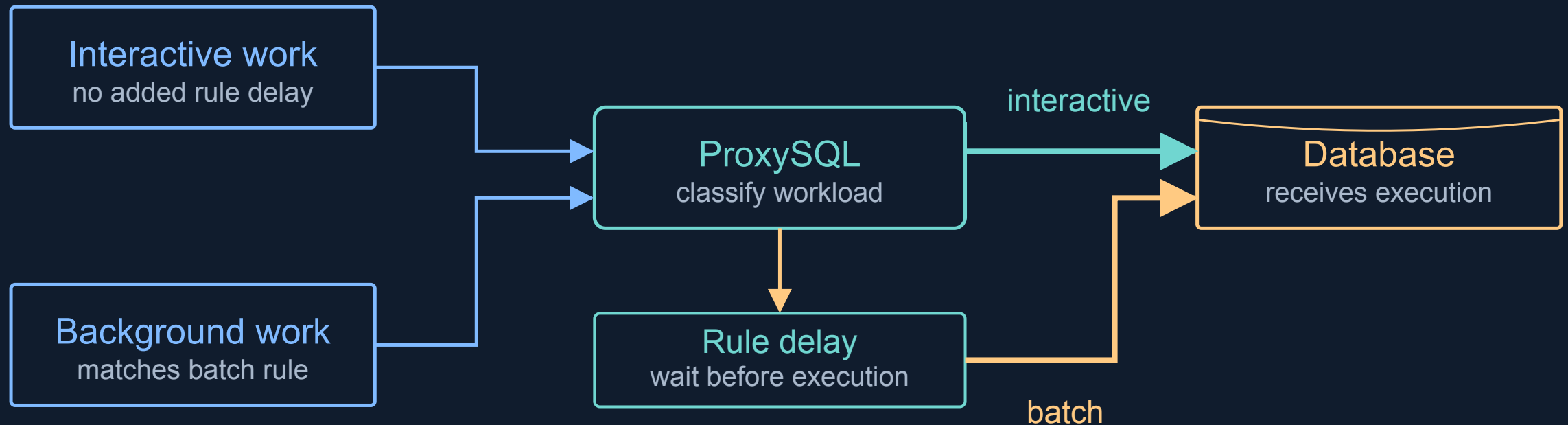


Example: add a tested MySQL index hint to one catalog query.

`match_pattern` → `replace_pattern`

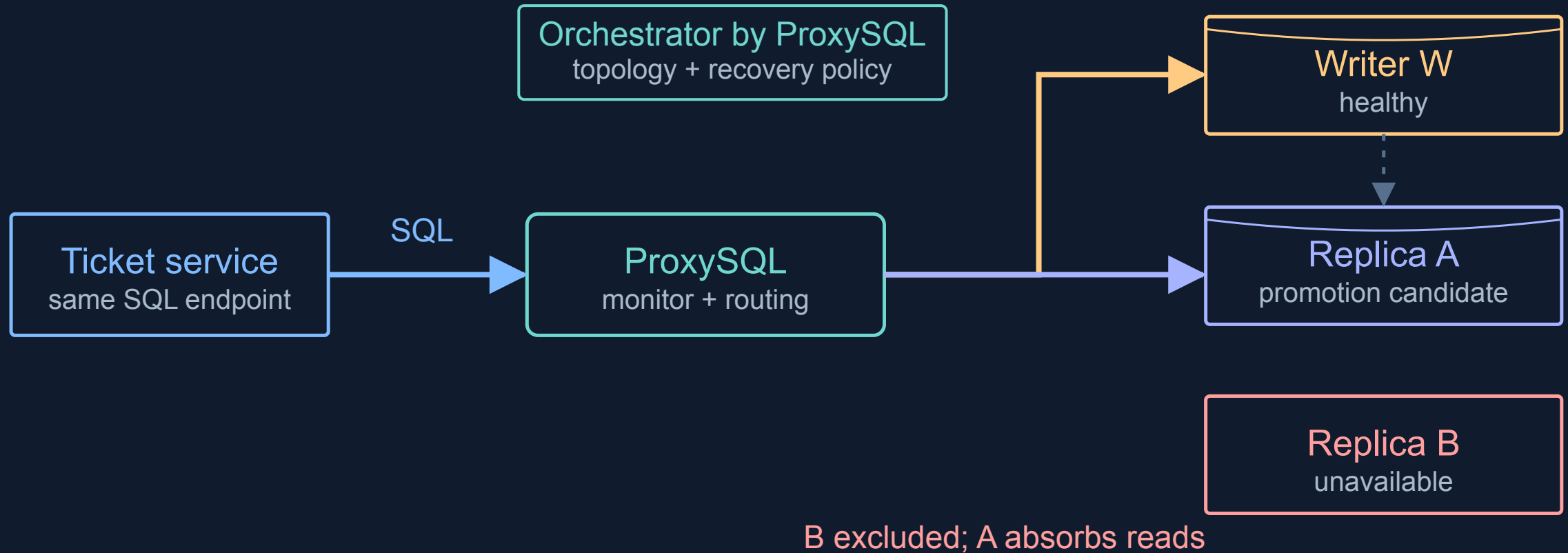
Change a selected query at the middle layer without redeploying the application.

Query throttling can slow a selected workload



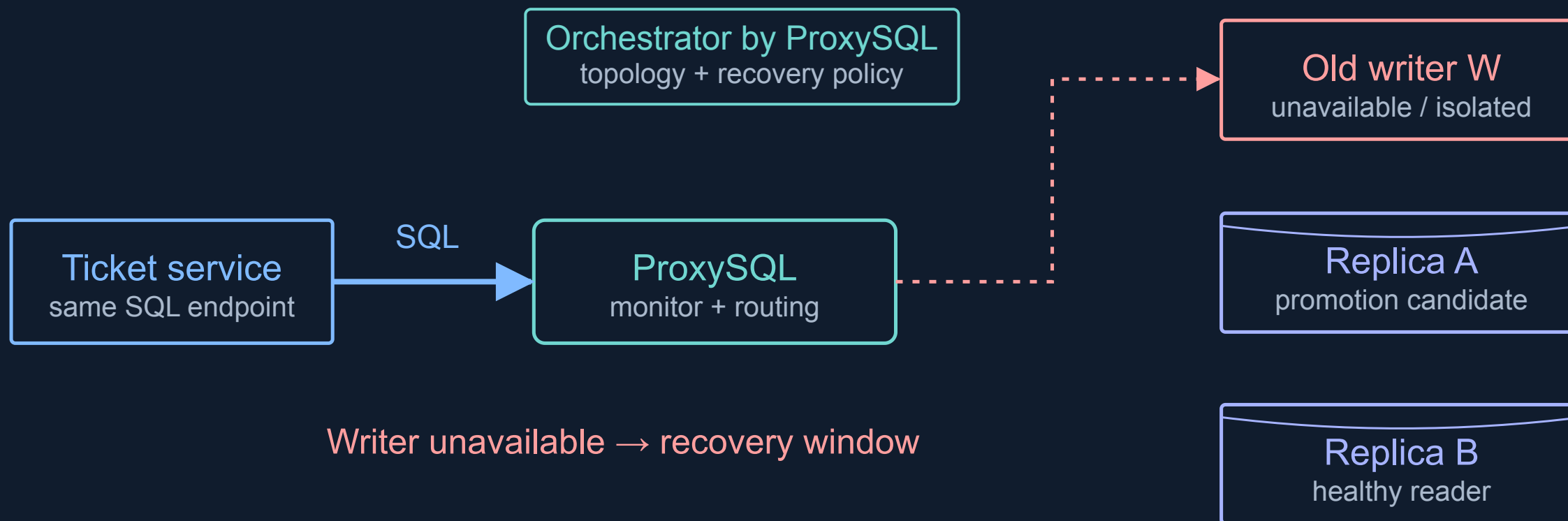
Delay matching background work; observe the effect on load and latency.

A reader fails. The service still has another route.



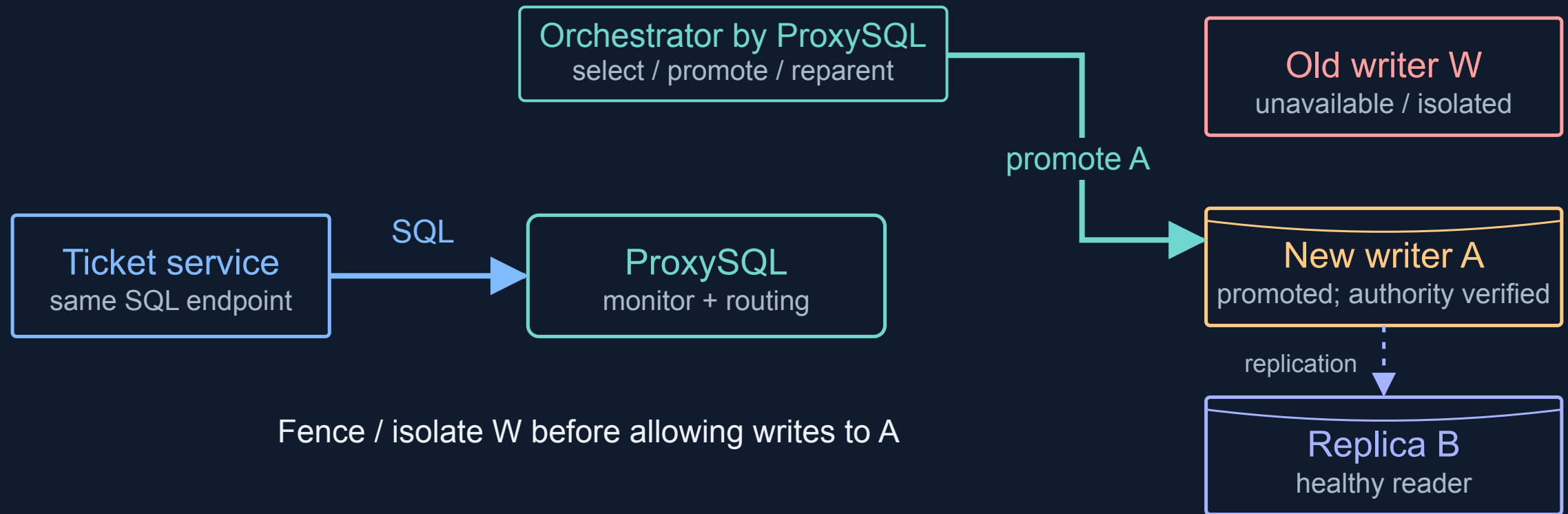
ProxySQL excludes the failed reader and uses eligible survivors.

Later, the writer fails. Who becomes the new writer?



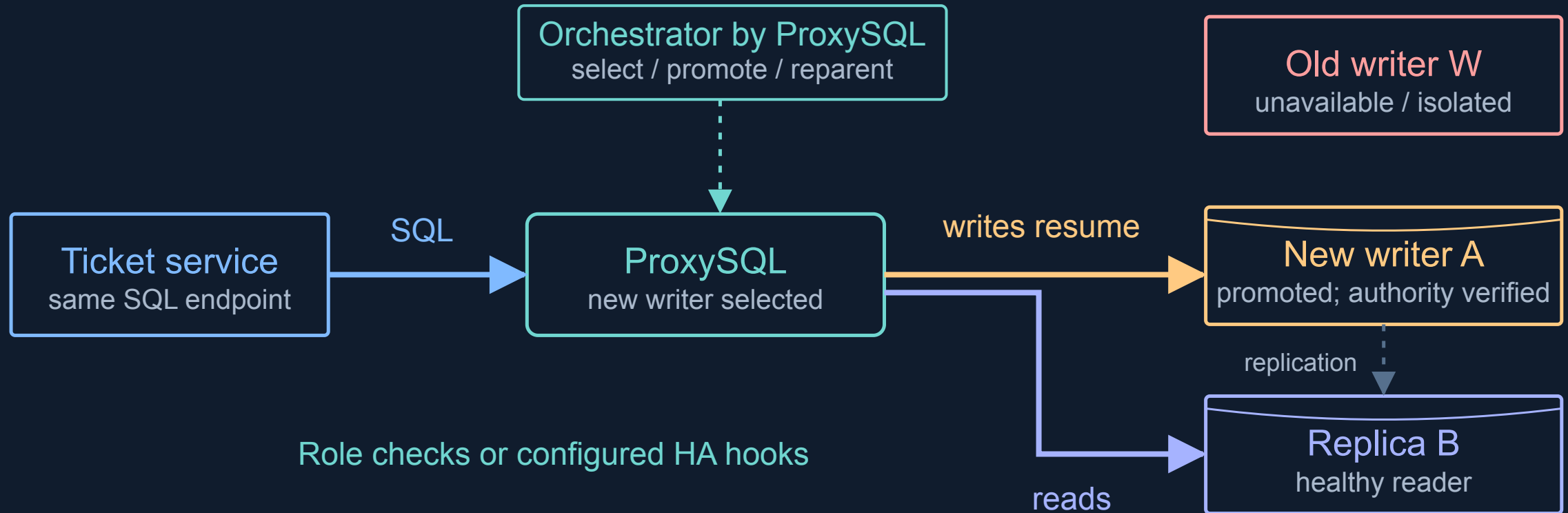
Detection opens a recovery window; promotion needs a topology authority.

Orchestrator promotes a candidate and repairs the topology



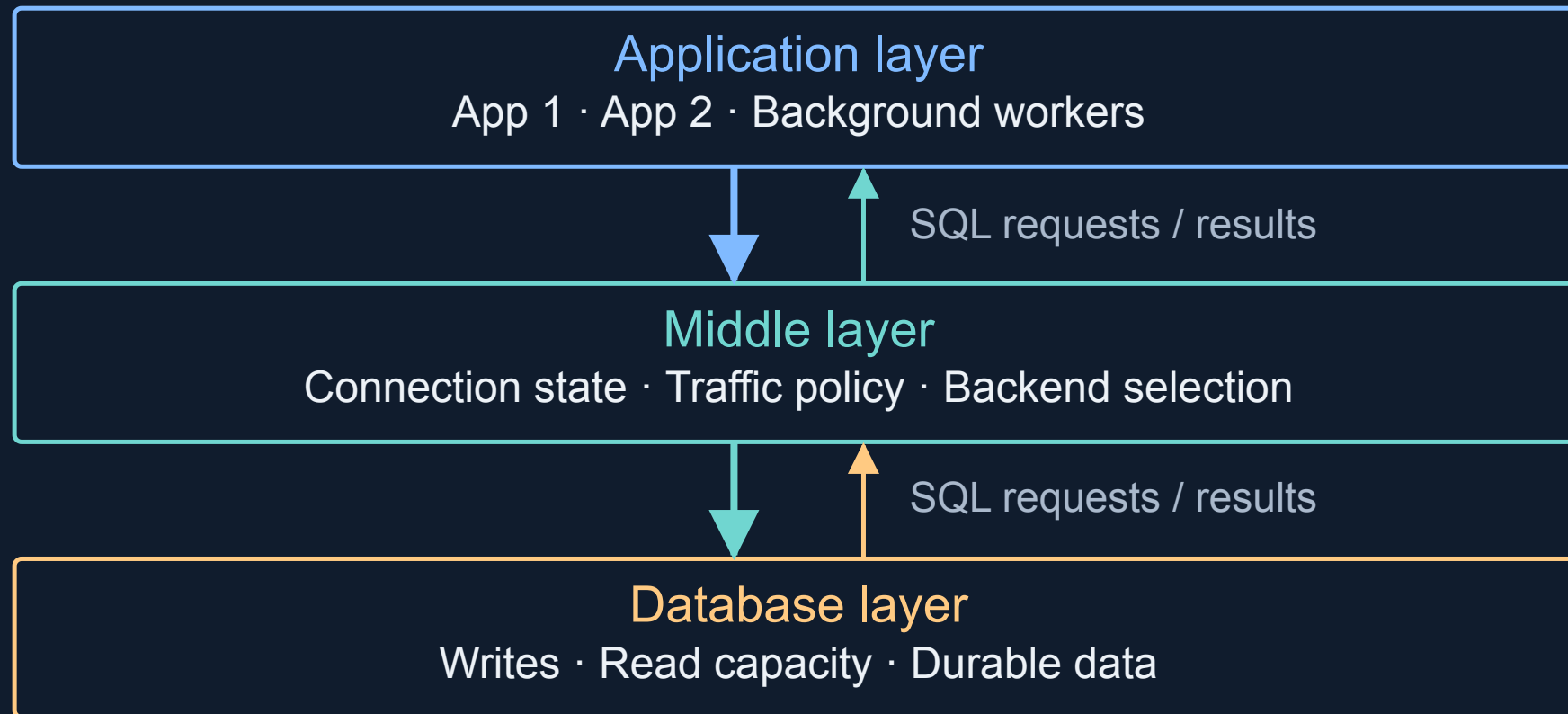
Recovery policy authorizes A; the remaining replica follows the new writer.

ProxySQL follows the new writer. The endpoint stays the same.



Automatic redirection for new work; interrupted transactions still need recovery.

The middle layer gives us one place to manage SQL traffic



Reuse connections · Observe and shape traffic · Adapt backend routing

The architecture grew. So did the routing contract.

More apps	Reuse connections without losing session state
More databases	Choose readers that satisfy the operation
Production changes	Observe, reconfigure, and recover deliberately

ProxySQL supplies the traffic mechanisms.
The application defines what correctness means.

Questions

René Cannaò · ProxySQL
proxysql.com