THE ILLUSION OF IDEMPOTENCY

Why MCP actions aren't as safe as you think

Prerit Munjal
Product & Engineering GROUPON

Where I build: Groupon

GROUPON

The marketplace for local experiences — connecting customers with the local merchants who power Main Street.

"Get people offline — through quality local experiences at great value."

MY CORNER OF IT

Data platform · Observability — the plumbing under all of it.

Millions

of local merchants powering Main Street

2008

founded in Chicago · NASDAQ: GRPN

AI-native

rebuilding for agentic commerce — the world this talk lives in

Personal views — not an official statement of Groupon, Inc. · Figures from Groupon's public Q1 2026 results (May 2026).

"

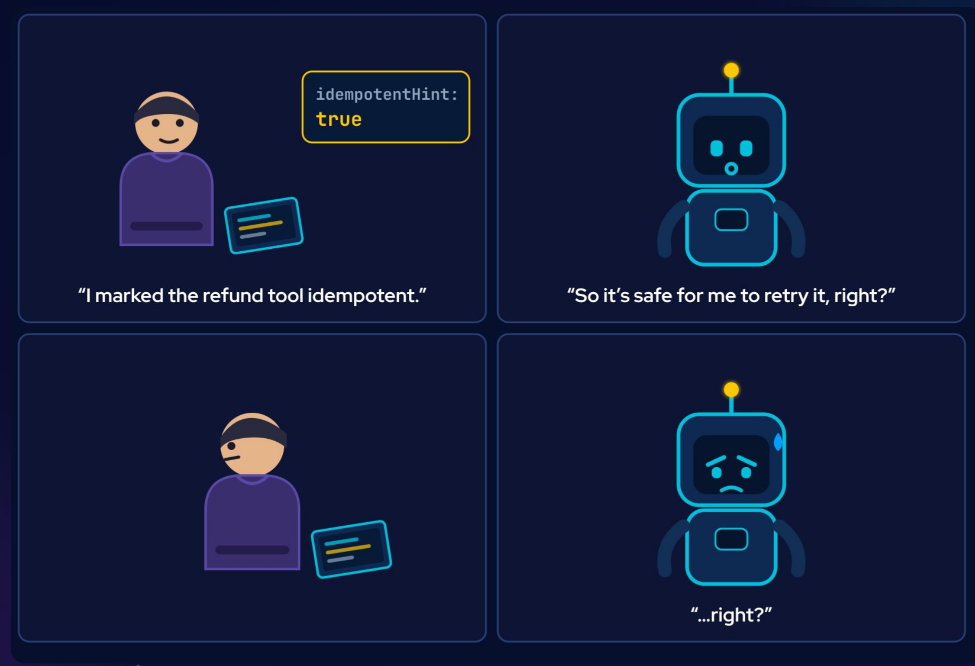
The most dangerous systems aren't the ones that fail.

They're the ones you believe are safe to retry.

— the premise of the next 25 minutes

// MOVEMENT 0 · A TRUE STORY

How every incident in this talk begins



(original art — any resemblance to your last standup is coincidental)

// MOVEMENT I · THE PROMISE

The story we tell ourselves

apply it twice — nothing changes

the contract we silently assume every tool call honors

Retries are free

If a call fails, just send it again. No harm done.

Reconciliation converges

Run the loop until reality matches the spec.

State is declarative

Describe the end state; the system makes it so.

So we wire LLM agents to retry anything, anytime — and call it safe.

// MOVEMENT I · THE PROMISE

What idempotency actually demands

Same effect — not same response

Returning “ok” twice is easy. Causing the change exactly once is the hard part.

Side effects are state too

Emails, webhooks, audit rows, usage meters — they all count toward “nothing changed”.

Holds under concurrency

Two interleaved calls must still land the system on one defensible outcome.

Survives partial failure

Crash after the commit, before the reply — the retry must not re-execute.

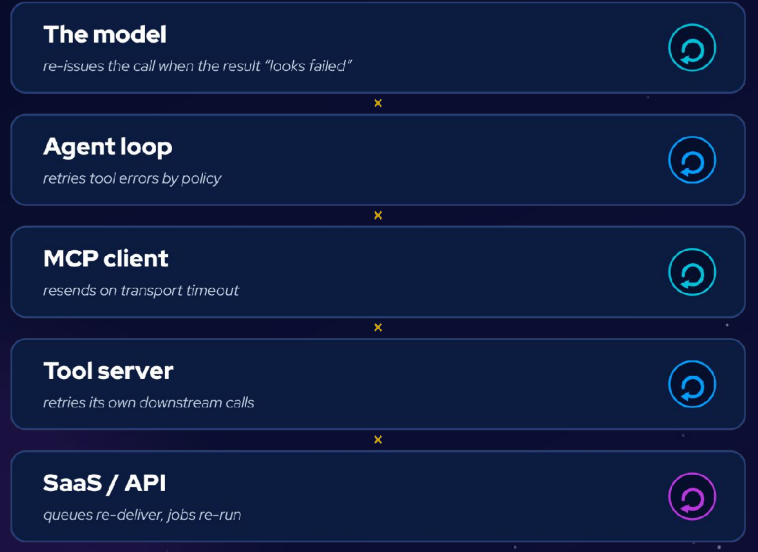
A timeout is not a failure — it's an unknown

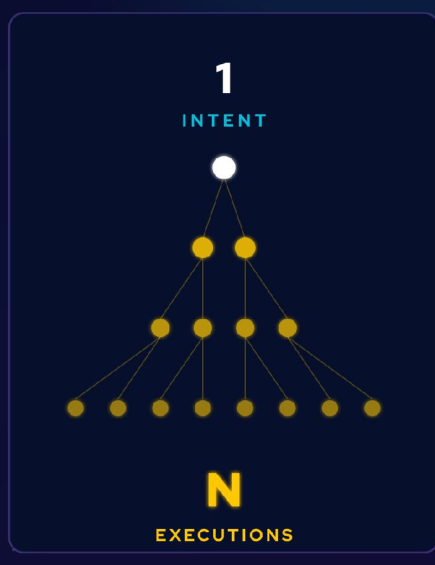
MCP CLIENT

TOOL SERVER

// MOVEMENT III · WHY MCP MAKES IT WORSE

Five layers, each retrying on its own





nobody is counting

Every layer believes some other layer is being careful.

// MOVEMENT III · WHY MCP MAKES IT WORSE

The spec has a flag for it — and fine print

```
"annotations": {
  "title":           "Refund payment",
  "readOnlyHint":    false,
  "destructiveHint": false,
  "idempotentHint":  true,    ← the lie
  "openWorldHint":   true
}
```

MCP tool annotations — what the model is told about your tool

The fine print

- Annotations are hints — not verified, not enforced.
- Clients are told not to rely on them from untrusted servers.
- Nothing in the protocol checks a tool against its own metadata.

Anything can claim idempotentHint: true. Nothing has to earn it.

// MOVEMENT III · WHY MCP MAKES IT WORSE

Your caller is a language model

paraphrases arguments

THE MODEL'S WORLD

- "no reply → it must have failed"
- "I'll phrase it a bit differently this time"
- "safe to try again – the hint said so"

THE ACTUAL WORLD

- the first call committed after the timeout
- "slightly different" = brand-new arguments
- every retry is a fresh mutation

"Same input, same output" assumes the same input. You don't even get that.

// MOVEMENT IV · FOUR WAYS IT BREAKS

01

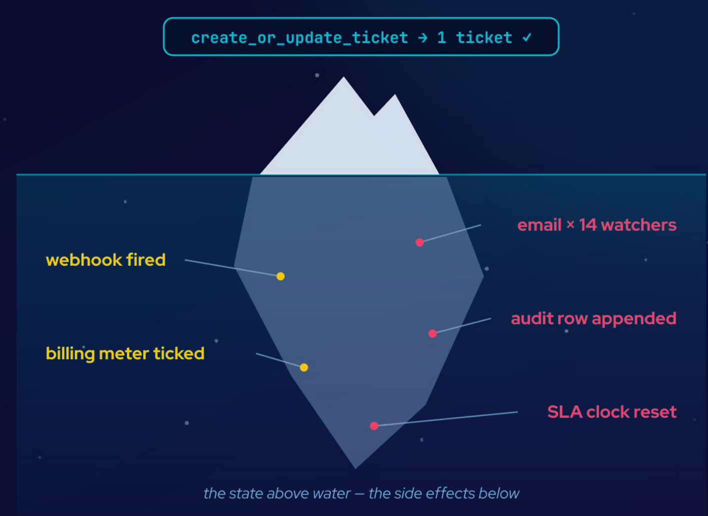
Hidden side effects

WHAT YOU ASSUME

create_or_update_ticket converges state — run it twice, one ticket.

WHAT ACTUALLY HAPPENS

Each run also emails 14 watchers, fires the webhook, appends an audit row, ticks billing.



FIELD EXAMPLE

An agent retried a “safe” upsert three times during a flaky deploy ⟶ three “ticket updated” emails, SLA clock reset twice.

The state converged. The world didn’t.

// MOVEMENT IV · FOUR WAYS IT BREAKS

02

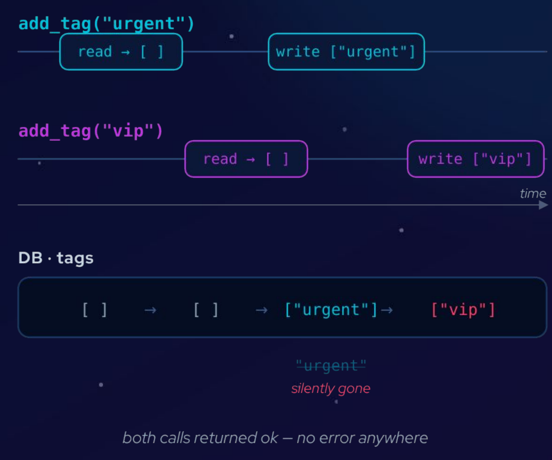
Ordering that doesn't exist

WHAT YOU ASSUME

The plan executes step 1 → 2 → 3, exactly as the model wrote it.

WHAT ACTUALLY HAPPENS

Clients parallelize tool calls. No transaction across tools. Two read-modify-writes interleave — last writer wins.



FIELD EXAMPLE add_tag("urgent") and add_tag("vip") run concurrently → both read [], both write one tag → "urgent" silently vanishes.

Plans are sequential. Execution isn't.

Divergence under retries

WHAT YOU ASSUME

A timeout means it failed — so retrying just means trying again.

WHAT ACTUALLY HAPPENS

The first call committed after you stopped waiting. The retry commits again. Agent records one refund; ledger holds two.

FIELD EXAMPLE

Every step after the retry now reasons from a false balance — and the agent acts on that belief with total confidence.

Retrying an unknown forks reality.

// MOVEMENT IV · FOUR WAYS IT BREAKS

04

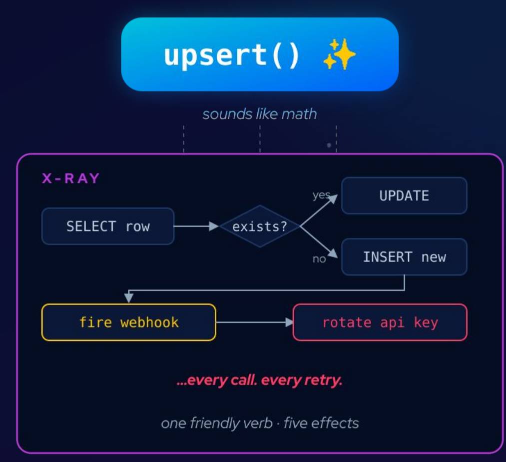
Safe-looking abstractions

WHAT YOU ASSUME

upsert / ensure / sync — the verb sounds like math, so it must be safe.

WHAT ACTUALLY HAPPENS

upsert = SELECT-then-branch (a race window). ensure resets ACLs. set_config notifies. upgrade_plan is a delta in costume.



FIELD EXAMPLE

"ensure_user" looked read-mostly — it also rotated the API key on every call. Three retries, three locked-out integrations.

The verb promises math. The implementation does I/O.

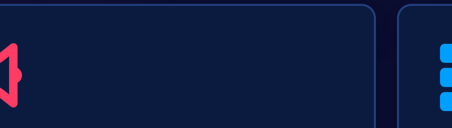
What this costs when it ships

Double money movement

Refunds, payouts, charges — issued twice.

Duplicate work objects

Tickets, orders, branches, resources ×2.

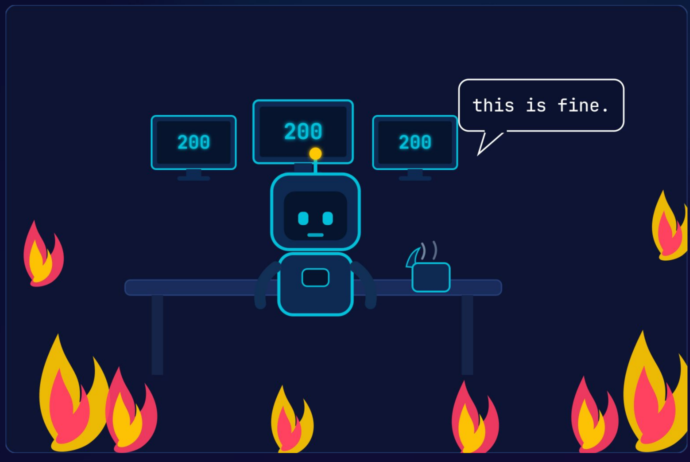


Notification storms

Every retry re-fires email, SMS, webhooks.

Zombie infrastructure

Re-provisioned, untracked — billed monthly.



Silent divergence. *No error. No alert. Everything returned 200 — the books just stop balancing.*

The expensive failures are the quiet ones.

// MOVEMENT V · PROOF

The demo, X-rayed — what you're about to watch

response: confirmation LOST after commit

demo_retry.py
terminal · zero deps

MCP Inspector
browser · mcp dev

Claude Desktop
agent · prompt-driven

MCP · JSON-RPC
stdio

payments_server.py
FastMCP · in-process "bank"

- refund_payment — idempotentHint:true ⚠
- get_ledger
- reset_demo

⚡ first fresh call: COMMIT, then raise
"confirmation lost" · DEMO_FAIL_FIRST=1

commit

LEDGER
in-memory · ground truth

PAY_7F2A $40.00
refunds []

double_paid → watch this

FIXED VARIANT · payments_server_fixed.py

agent mints key ONCE ▶ retry carries SAME key ▶ seen{key}? → replay stored result ▶ ledger: R1 only

same timeout · same retry · one payout

// MOVEMENT V · PROOF

Demo — the retry that paid twice

WATCH FOR

- One human intent: “refund this payment.”
- The only “failure” is a lost response.
- Then the same flow, fixed — same retry, one payout.

Everything from the slides — live, in one terminal.

// MOVEMENT V · THE FIX

One parameter. Three rules.

```
- def refund(payment_id):
-     refunds.append(payout)      # no key, no memory

+ def refund(payment_id, idempotency_key):
+     if key in seen:
+         return seen[key]    # replay, don’t re-run
+     refunds.append(payout)
+     seen[key] = result      # persist BEFORE reply
```

Caller mints the key

before attempt #1 — and reuses it on every retry.

Server replays

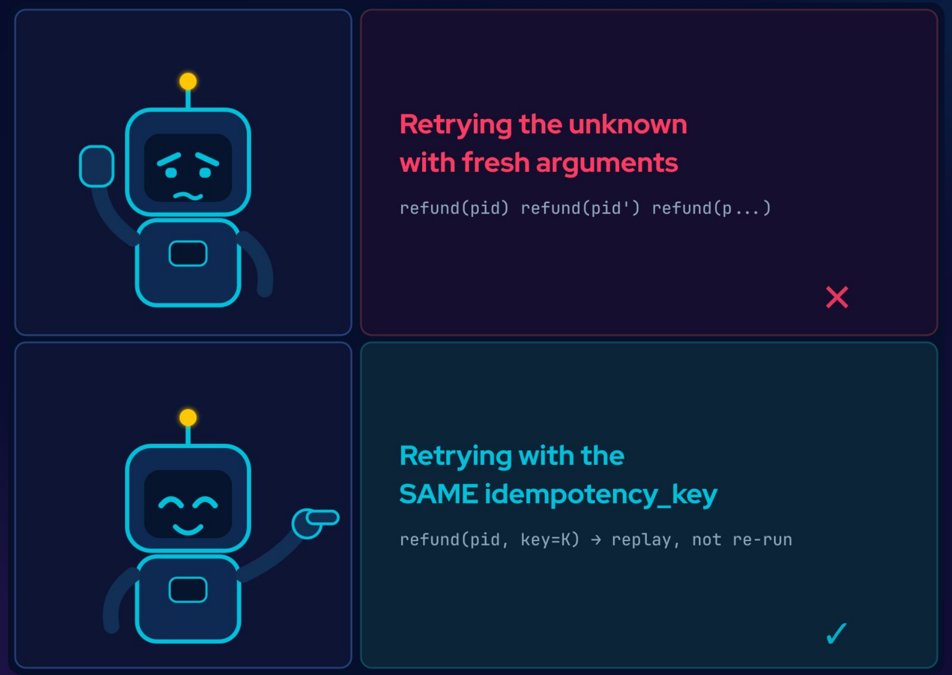
same key → return the stored result. Never re-execute.

Commit together

effect + key persist in one transaction (outbox).

Idempotency is enforced server-side — or it doesn’t exist.

tl;dr



Screenshot this one. Put it in the team channel.

// MOVEMENT V · DESIGN RULES

If you build MCP tools

Demand a key

Every effectful tool takes idempotency_key. Dedupe and replay.

Box the side effects

Email, webhooks, meters commit with the record — or not at all.

Absolute, not delta

set_quantity(7) survives retries. increment() never will.

Expose preconditions

Versions / ETags → compare-and-swap, not blind writes.

Split plan from apply

Dry-run returns the diff. Apply executes exactly that diff.

Document the blast

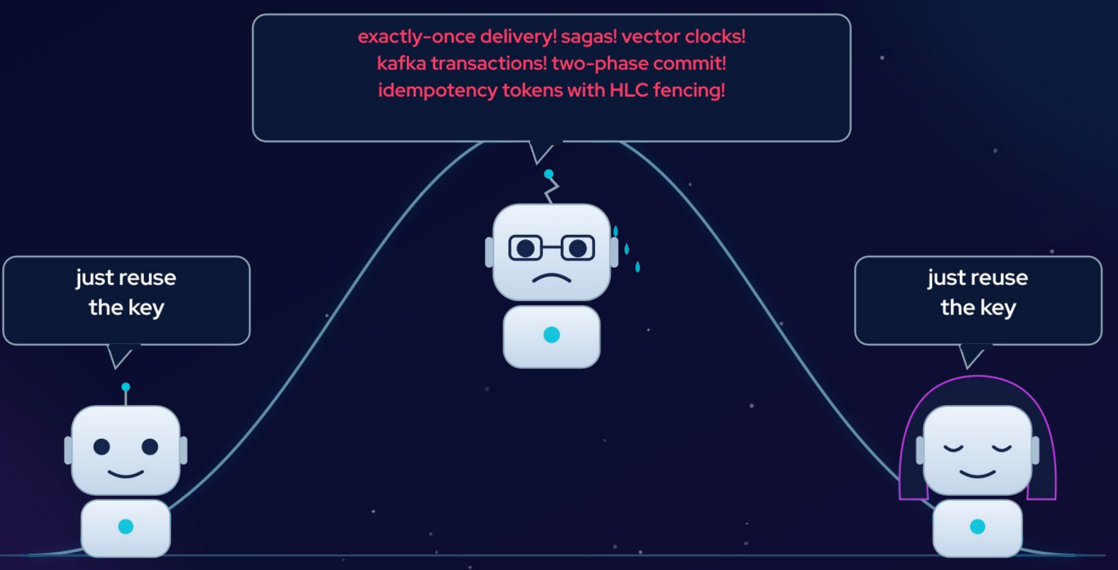
List every effect in the description — the model reads it.

// MOVEMENT V · DESIGN RULES

If you build the agent side

1	**Mint the key first**	generate it before attempt #1 — every retry carries the same key.
2	**Read before re-acting**	on timeout, query state; never blind-fire a write you can't account for.
3	**One owner for retries**	pick a layer; every other layer passes failures up, untouched.
4	**Gate the destructive**	destructiveHint or openWorldHint ⇒ a human confirms, with the diff in front of them.
5	**Reconcile to detect**	drift is a signal to investigate — not a license to stomp concurrent writers.

Both ends of the curve agree



the bell curve of exactly-once

// TAKE A PHOTO

The six-line contract

01 Effectful? It needs a key.

02 A timeout is an unknown — not a no.

03 Deltas are landmines. Set absolute state.

04 Side effects count as state.

05 Hints are not guarantees.

06 Can’t make it idempotent? Make it confirmable.

Systems don’t fail because they break.

They fail because we believed the retry was free.

Idempotency is not a property you assume.
It’s a contract you enforce.

Thank you · धन्यवाद

Prerit Munjal · Product + Engineering GROUPON