Retries Have an Evil Twin: Duplicates

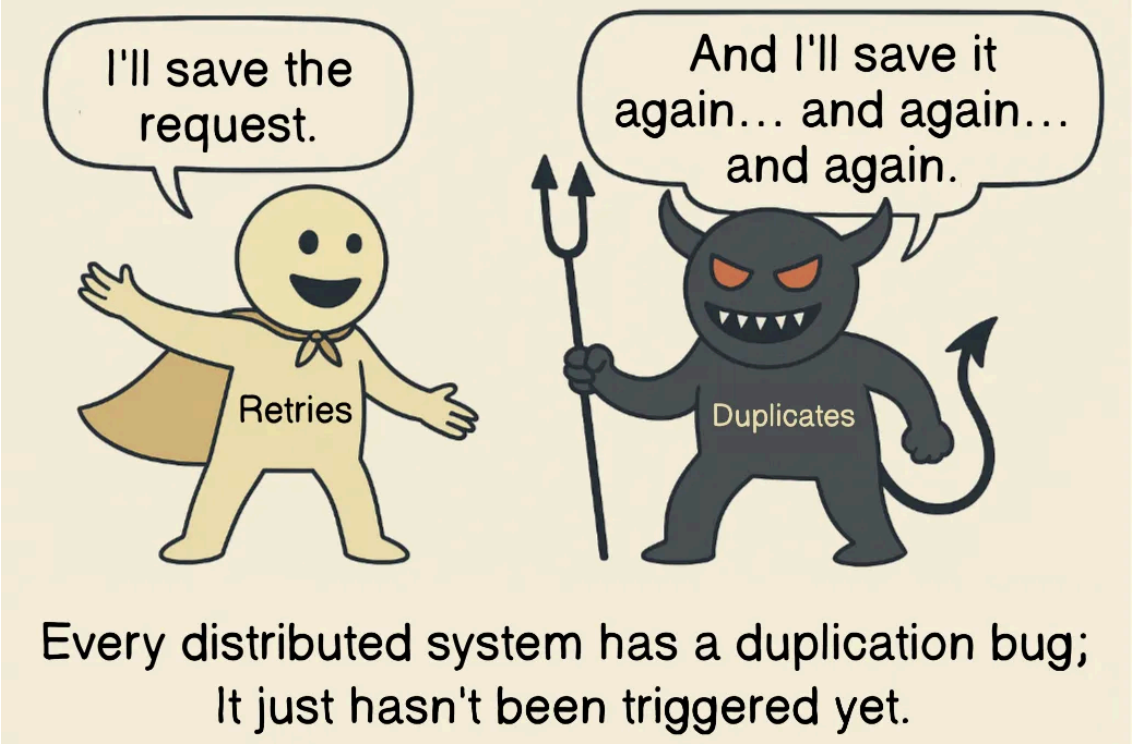
Every distributed system has a duplication bug; it just hasn't been triggered yet.

RAUL JUNCO
JUL 28, 2025

You've built retries.
You've added queues.
You think you've made everything '**resilient**.'

And then a payment went through twice.
Two orders shipped.
The same email got sent three times.

Welcome to the ugly side of retries no one warns you about: **duplication.**



In distributed systems, **At-Least-Once delivery is the default**. Messages get retried. Requests get replayed. But if your app isn't ready for it, you end up with corrupted data, angry customers, and incomprehensible logic

This post breaks down **four battle-tested strategies** to handle duplication at the application level, so your systems stay correct, even when the same work shows up more than once.

The Problem: Same Message, Twice the Effect

Let's say a user submits a payment. The request goes through your API, but the backend crashes right before saving the result.

Your retry logic kicks in.

Now the same request is processed again. And you've:

- Charged the user twice.
- Created two payment records.
- Triggered two downstream events.

This isn't just bad luck. It's **by design**.

> ⚠️ **At-least-once delivery is the default.**
> Most systems, especially queues, retries, and distributed APIs, guarantee that a message will arrive **at least once**, but may arrive **multiple times**.

Why? Because retries are safer than data loss.
It's your job to make sure duplicates don't break your system.

What's at Stake?

Without duplication control:

- **Your data becomes unreliable.** Systems that rely on exact state, like billing or inventory, get corrupted.
- **Your code becomes defensive.** Every handler needs if-checks, patches, or compensating logic.
- **Your operations team burns out.** They spend hours deduplicating rows, refunding customers, and triaging inconsistencies.

You don't just lose trust. You will lose time.

4 Ways to Stop Duplication at the Application Layer

These approaches aren't theoretical. They show up in real production systems, depending on scale, latency tolerance, and system design.

1. Database Unique Constraints

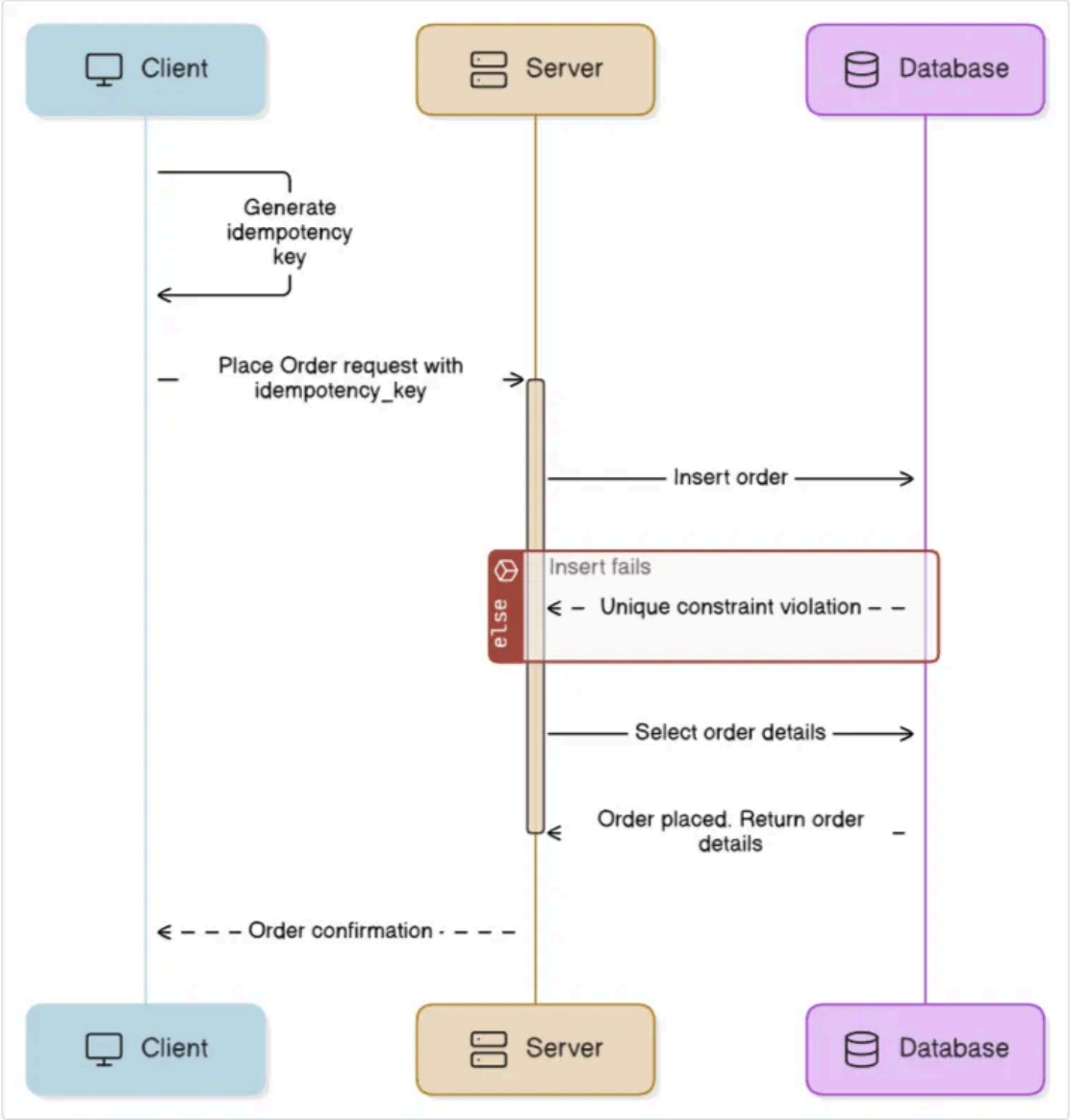
Concept: Use a `UNIQUE` constraint in the database to prevent duplicate operations, backed by an **idempotency key**.

Let's say a user places an order.
The client generates a unique `idempotency_key` and sends it with the request:

```
CREATE TABLE orders (
  id UUID PRIMARY KEY,
  idempotency_key TEXT UNIQUE,
  ...
);
```

The server attempts to insert the order using this key. If the same request is retried (e.g. due to a timeout), the database will reject the duplicate with a constraint violation.

You catch that error and fetch the original order.



✅ Pros

- Ensures **only one successful insert** per key. Potential performance impact from high-concurrency insert conflicts or deadlocks.

- Clean fallback path for retries, return the existing result.
- Ideal for operations like **checkout, registration, or payment initiation**.

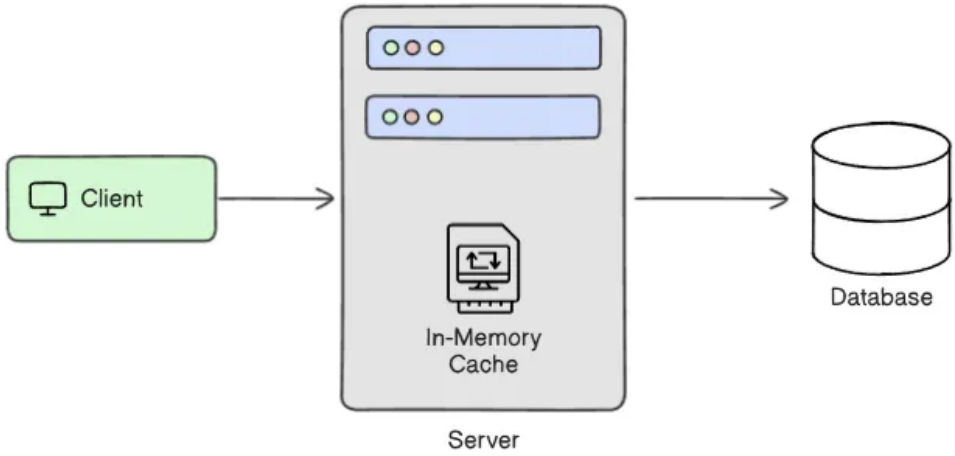
⚠️ Cons

- Adds write pressure to the database. In
- Doesn't protect downstream effects unless guarded (e.g. emails, inventory).
- Needs good **key generation hygiene** on the client or gateway.

📌 **Key insight:** The database isn't just for persistence. It's a **gatekeeper** that protects your system from replayed requests, *as long as you give it something to gate on.*

2. In-Memory Deduplication

Concept: Track processed request IDs in memory using a `Set`, Map, or LRU cache. On each request, check if the ID was seen before.



It's fast. It's simple. But it's **only safe if your service is single-instance** and doesn't restart frequently.

Why it works: The process itself remembers what it's already done, no I/O needed.

✅ Pros

- Blazing fast.
- No infrastructure dependency.
- Great for **short-lived** tools and **batch processes.**

⚠️ Cons

- **Volatile memory**: everything is wiped on restart or crash.

- Doesn't work in **multi-threaded** or **multi-node** systems without coordination.
- You need to **manually clean up** old keys or use a TTL mechanism.

📌 **Best for:** One-off scripts, CLI tools, test environments, or internal utilities where reliability isn't mission-critical, monoliths with low/moderate operation frequency.

3. Distributed Cache (Redis)

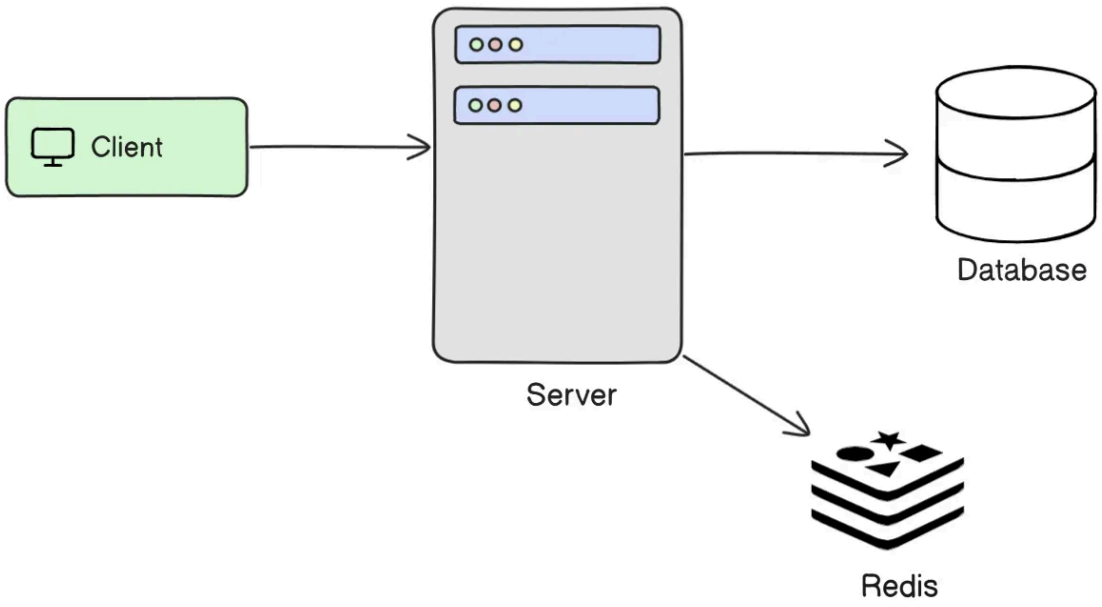
Concept: Use Redis to track processed operations across distributed nodes. Store a key for each request ID with a TTL:

```
SETNX request:<id> "processed"    # Only set if not exists
EXPIRE request:<id> 600           # Auto-expire in 10 minutes
```

This ensures:

- Only one node processes the request.
- Future retries are blocked.
- The de-dup record eventually expires to free memory.

Why it works: Redis becomes a **shared memory layer** across services and instances.



✅ Pros

- Fast lookup and write.
- Survives restarts and horizontal scaling.

- TTL provides automatic cleanup.

⚠️ Cons

- TTL tuning is **non-trivial**. Too short and legit retries slip through; too long and Redis bloats.
- **Network partitioning or Redis downtime** means you may process duplicates.
- You'll need to deploy and monitor Redis reliably. It can become a single point of failure.

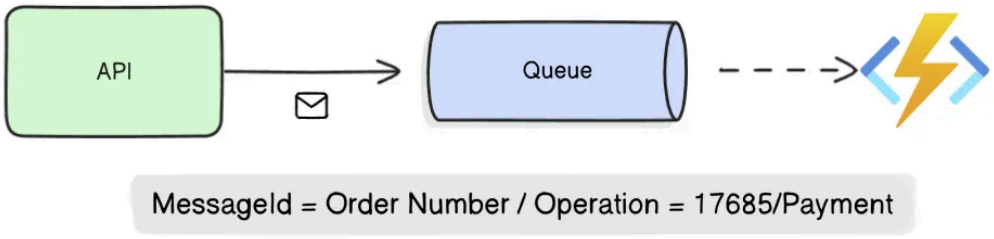
📌 **Tradeoff tip:**
Pair Redis deduplication with a **fallback to DB uniqueness** to catch any misses due to cache failures or TTL gaps.

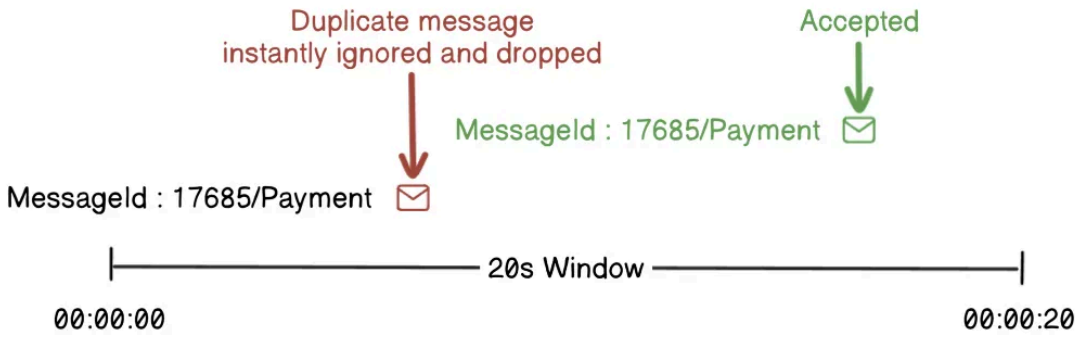
📌 **Best for:** API services, job processors, microservices with retry logic.

4. Message Broker Deduplication

Concept: Some brokers (e.g., Azure Service Bus, Pulsar) support built-in duplicate detection.

You assign a `MessageId` to each message. The broker keeps a **deduplication window** (e.g. 10 minutes). If the same ID appears again, it discards the message.





Why it works: The **message infrastructure** becomes the gatekeeper; duplicates don't even hit your app.

✅ Pros

- **No application logic** required.
- Prevents duplicate **delivery**, not just processing.
- Works great with **at-least-once** brokers.

⚠️ Cons

- Requires proper configuration (de-dup window, clock sync).
- Doesn't help if your app publishes **duplicate downstream events**.
- Broker-specific — not portable across infra.

📌 **Best for:** Event-driven architectures, high-throughput pipelines, or systems with built-in broker support.

Trade-Off Comparison

Method	Scope	Persistence	Infra Needed	Good For
DB Unique Constraints	Single-node	Strong	None	Registration, payments, orders
In-Memory Set	Single-node	Volatile	None	CLI tools, simple workers
Redis Dedup	Distributed	TTL-based	Redis	APIs, microservices
Broker Deduplication	Messaging	Broker-limited	Message broker	Event-driven architectures

Final Takeaways

- **Duplication is a design reality**, not a bug. Plan for it.
- **At-least-once delivery** means your systems will get the same input more than once. Act accordingly.
- **Use the database for strong guarantees**, Redis for fast coordination, and brokers for message-level protection.
- **TTL-based deduplication** is usually good enough. Start there.
- **Combine layers** (e.g., broker de-dup + Redis + DB constraint) for critical flows.

> You can't prevent retries, but you can prevent rework.
>
> Resilience without deduplication is just chaos with retries.

Until next time,
— Raul

System Design Classroom is a reader-supported publication. To receive new posts and support my work, consider becoming a paid subscriber.

Type your email... Subscribe

5 Likes · 2 Restacks

Discussion about this post

Comments Restacks

Write a comment...

Neo Kim 3h

nicely explained, Raul.

Do you have a go-to approach to implement idempotency & why?

LIKE REPLY SHARE