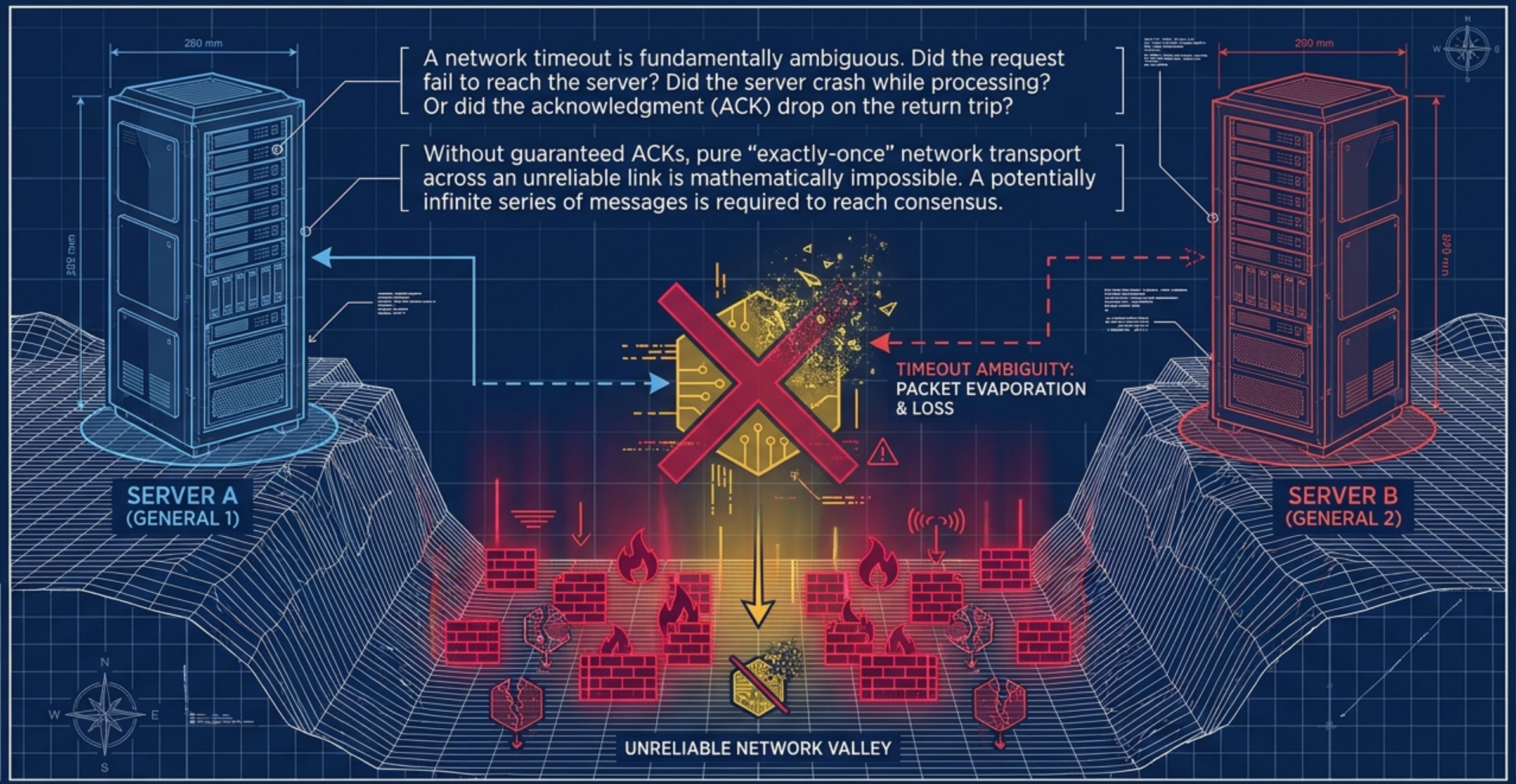
Hard Problems in Computing: S08E02 — Idempotency & 'Exactly-Once' Delivery

Engineering robust distributed state machines in a universe of unreliable networks.

> PREVIOUSLY IN S08E01 ('Randomness & Entropy'):
> We established how cryptographicaly secure PRNG UUIDv4 keys provide the foundation for collision-free identity. We now weaponize this to enable safe, deterministic network retries.

Gemini Notebook

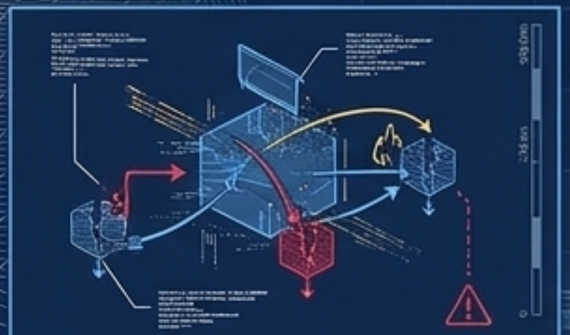
SECTION 1 — TIMEOUT AMBIGUITY & THE TWO GENERALS' PROOF



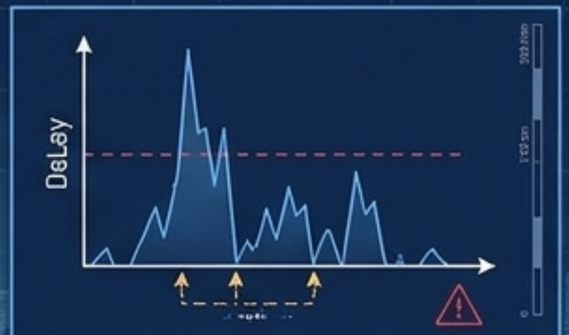
Gemini Notebook

THE DISTRIBUTED-SYSTEMS EQUATION

EFFECTIVE ONCE = **AT-LEAST-ONCE RETRIES** (WITH JITTER) + **STATEFUL IDEMPOTENT RECEIVER**



We accept that the network will drop packets. (Cite: Tyler Treat, "You Cannot Have Exactly-Once Delivery")



Randomness prevents the "Thundering Herd" of synchronized retries. (Cite: AWS Builders' Library)



The destination must be engineered to deduplicate identical payloads.

UNRELIABLE NETWORK VALLEY

Gemini Notebook

SECTION 2 — HTTP SEMANTICS (RFC 9110) & THE STATUS CODE NUANCE

$$f(f(x)) = f(x)$$

Safe: Read-only operations (**GET**).

Idempotent: Applying the operation multiple times yields the same server state as applying it once (**PUT**, **DELETE**).

THE NUANCE: Idempotency guarantees state, not identical responses.

Attempt 1: `DELETE /item/42` → `204 No Content`
Attempt 2: `DELETE /item/42` → `404 Not Found`

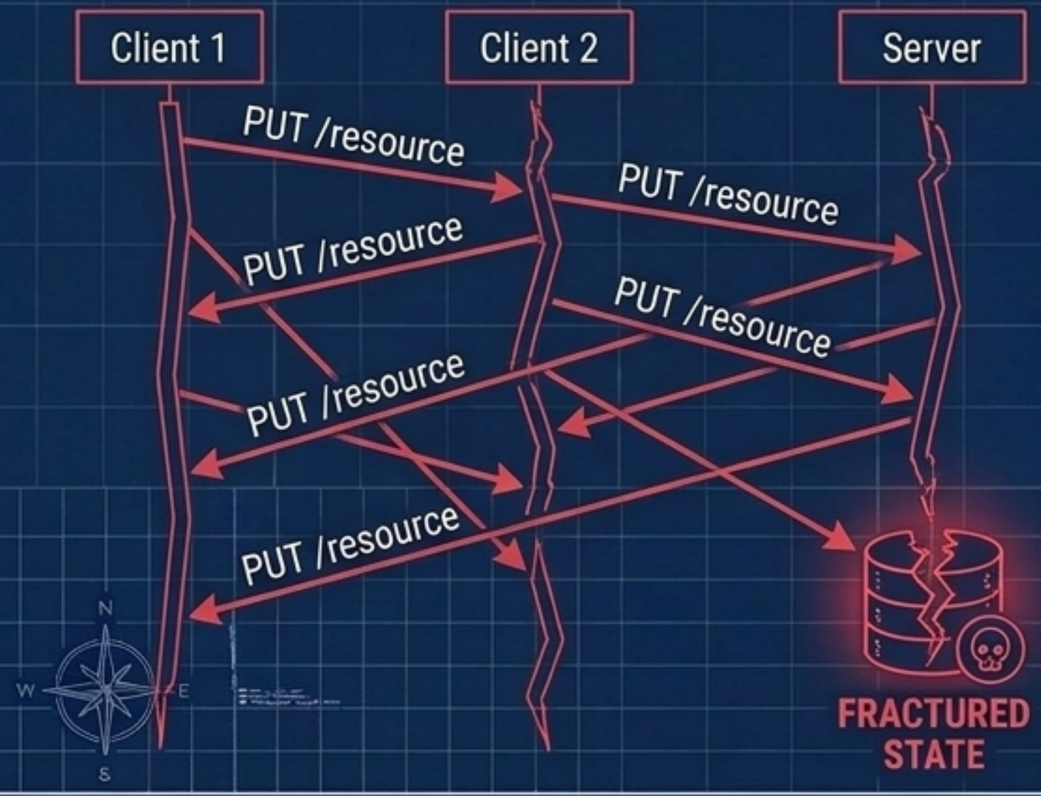
Result: The end state (item 42 is gone) is identical.

Gemini Notebook

CONDITIONAL REQUESTS & OPTIMISTIC CAS

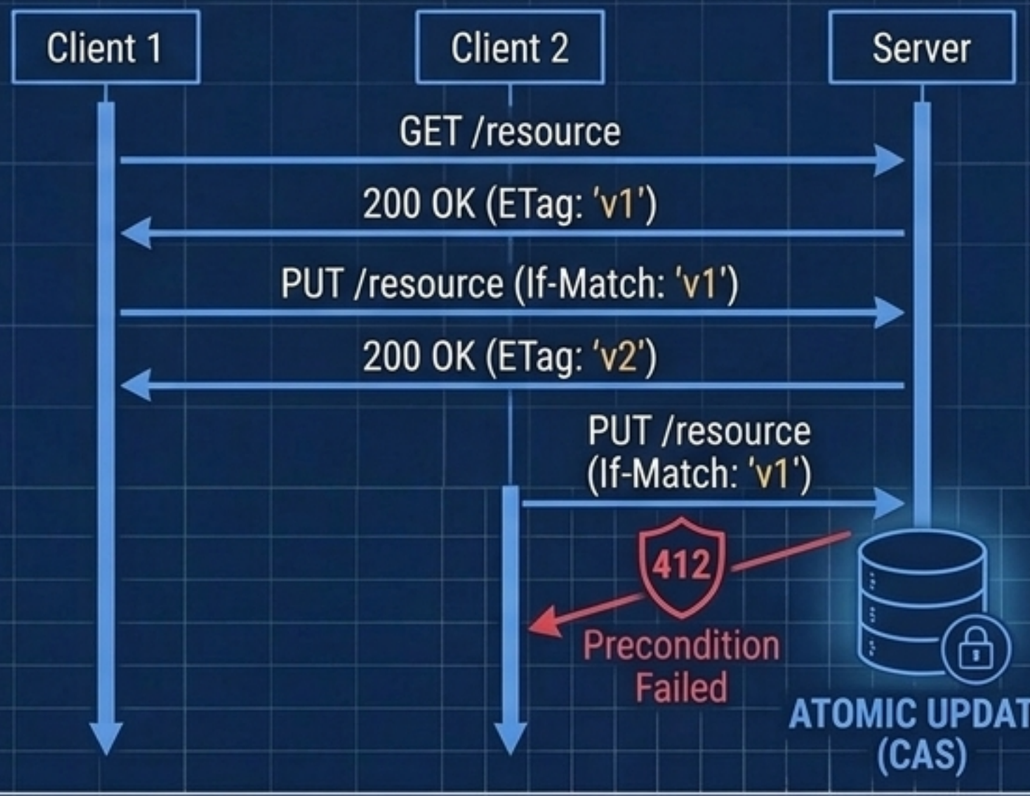
LOST UPDATE

Concurrent blind PUT requests result in lost updates when clients race to overwrite state.



OPTIMISTIC LOCKING

Server issues an ETag (Entity Tag) representing resource version. Client sends PUT with **If-Match: "etag_value"**. Turns an unsafe overwrite into an atomic **Compare-And-Swap** (**CAS**) mutation. Mismatches safely reject with a **412 Precondition Failed**.



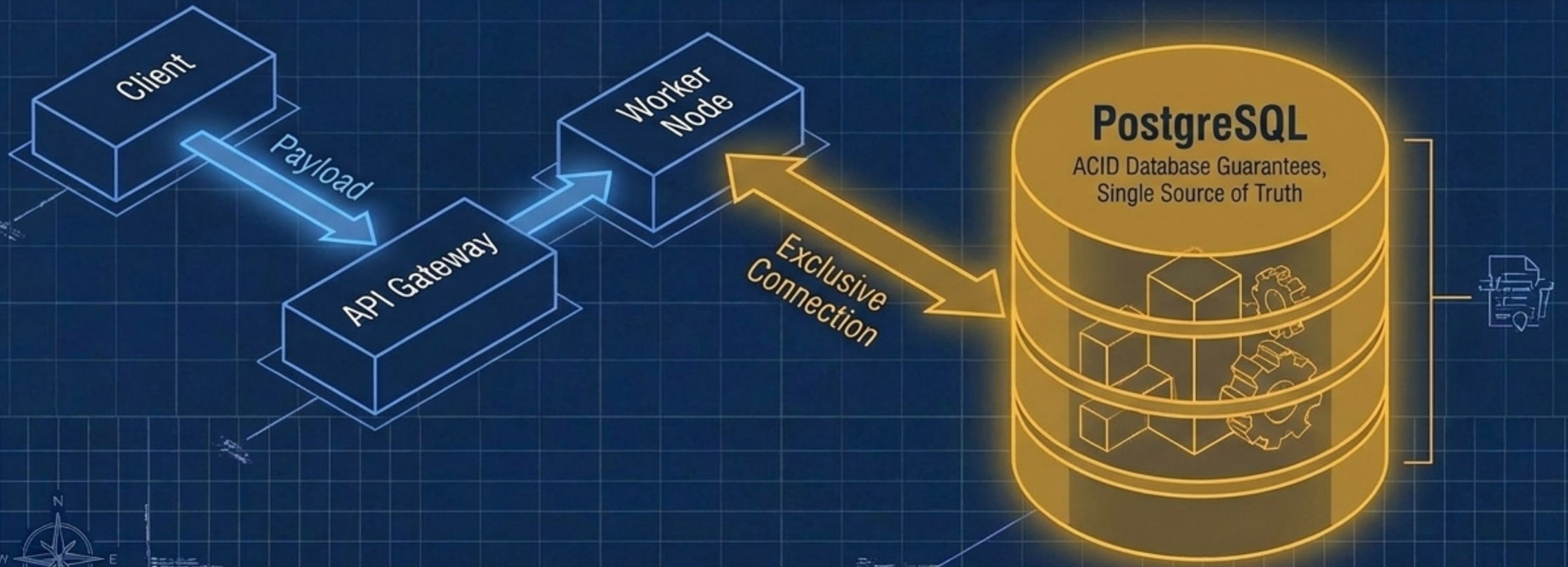
UNRELIABLE NETWORK VALLEY

Gemini Notebook

SECTION 3 — STRIPE-GRADE IDEMPOTENCY IN POSTGRESQL

The Standard: Based on the IETF Idempotency-Key draft, Google AIP-155, and Brandur Leach's atomic phases.

The Goal: Provide Stripe-level safety where clients can securely retry POST mutations without double-charging users or corrupting state, relying entirely on ACID database guarantees.



UNRELIABLE NETWORK VALLEY

Gemini Notebook

MULTI-PHASE ACID RECOVERY POINTS

STARTED

- Client-session-scoped UUIDv4 arrives in Idempotency-Key header.
- Atomic `INSERT ... ON CONFLICT DO NOTHING` into the idempotency_keys table.

IN_PROGRESS

Evaluate recovery point:
If newly STARTED → proceed with mutation and handle active lock.

FINISHED

If FINISHED → return cached response body and status code.



Gemini Notebook

FINGERPRINTING, LOCKS, & HORIZONS

Panel 1: Request Handling & Concurrency

```
1  // Handle Incoming Request
2  func handle_request(key, payload, client_id):
3      // 1. Generate Payload Fingerprint
4      payload_hash = sha256(payload)
5
6      // 2. Check/Create Idempotency Key
7      key_record = db.get_or_create_key(key, payload_hash,
       client_id)
8
9      // 3. Concurrency Check
10     if key_record.status == IN_PROGRESS:
11         throw 409 Conflict // Concurrent Retry Rejected
12     elif key_record.status == FINISHED:
13         return key_record.cached_response
14     else:
15         // Proceed with Mutation
16         process_and_store(key, payload)
```

Payload Fingerprinting: Hash the request body (SHA-256). If a client reuses an idempotency key with a different payload, throw 422 Unprocessable Content.

Concurrency Locks: If the key is actively processing (IN_PROGRESS), throw 409 Conflict to reject concurrent retries.

Panel 2: Background Tasks & Lifecycle

```
1  // Background Cleanup Task
2  func background_cleanup_task():
3      // 4. Stale Lock Recovery
4      stale_keys = db.get_stale_keys(timeout_threshold)
5      for key in stale_keys:
6          if key.locked_at < (now - timeout_threshold):
7              // Assume Worker OOM/Crash
8              key.unlock_and_recover()
9
10     // 5. Data Lifecycle Management
11     expired_keys = db.get_expired_keys(ttl=24h)
12     for key in expired_keys:
13         // Enforce TTL Horizon
14         db.delete_key(key)
15
```

Stale Lock Recovery: If locked_at is older than the timeout threshold, assume a worker OOM/crash. Allow lock steal and recovery.

Data Lifecycle: Enforce 24h TTL horizons. Keys are not permanent archives; they are temporary safety nets.

Gemini Notebook

Section 4 — Delivery & Deduplication Matrix

The Reality Check: There is no universal “exactly-once” tool.

Context Dictates Tooling: The correct mechanism depends entirely on failure boundaries, whether you are calling internal DBs or external APIs, and your tolerance for operational cost.

Gemini Notebook

Comparison Matrix: Synchronous & DB State

Architecture	Scope	External-API Safety	Operational Cost
At-Most-Once	Local	N/A (High data-loss risk)	Low
At-Least-Once + DB UNIQUE Key	Local DB inserts	Fails if side-effects exist (e.g., sending emails)	Low
HTTP If-Match Optimistic CAS	Resource updates	High (requires client read-before-write)	Medium

Gemini Notebook

Comparison Matrix: Asynchronous & Eventing

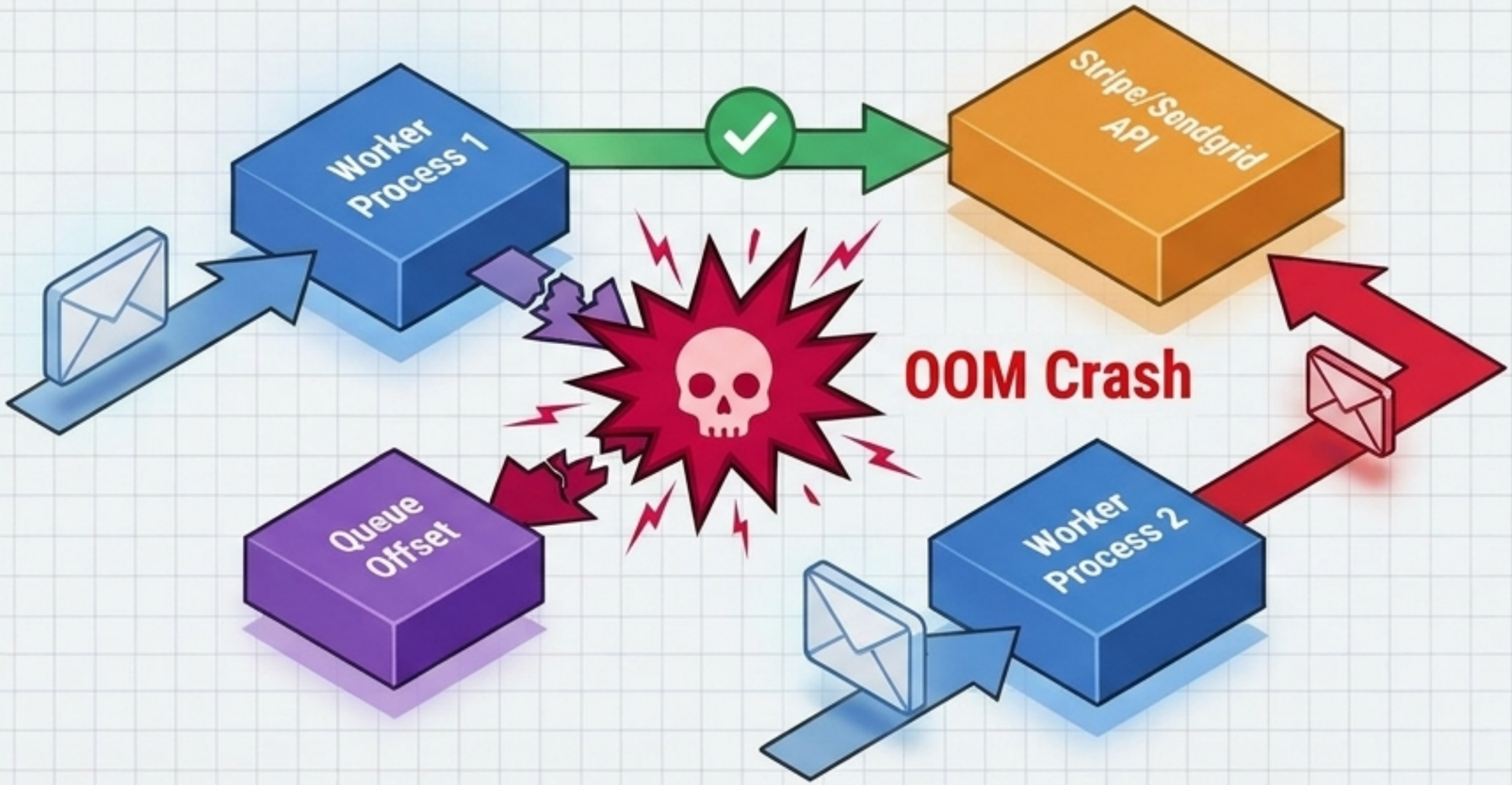
Architecture	Scope	External-API Safety	Operational Cost
Kafka KIP-98 Transactions	Strictly Kafka-in to Kafka-out	Zero protection for external API calls	High
AWS SQS FIFO	Queue deduplication	Relies on a strict 5-min MessageDeduplicationId window	Medium
Transactional Outbox + Debezium CDC	DB to Event Bus	Guarantees DB and message publish atomicity	High
Temporal Durable Execution	Code execution replay	Ultimate safety via event sourcing; paradigm shift	Very High

Gemini Notebook

Section 5 — Where ‘Exactly-Once’ Marketing Fails

The Trap: Systems like Kafka guarantee exactly-once internal state, but they cannot protect your external side-effects.

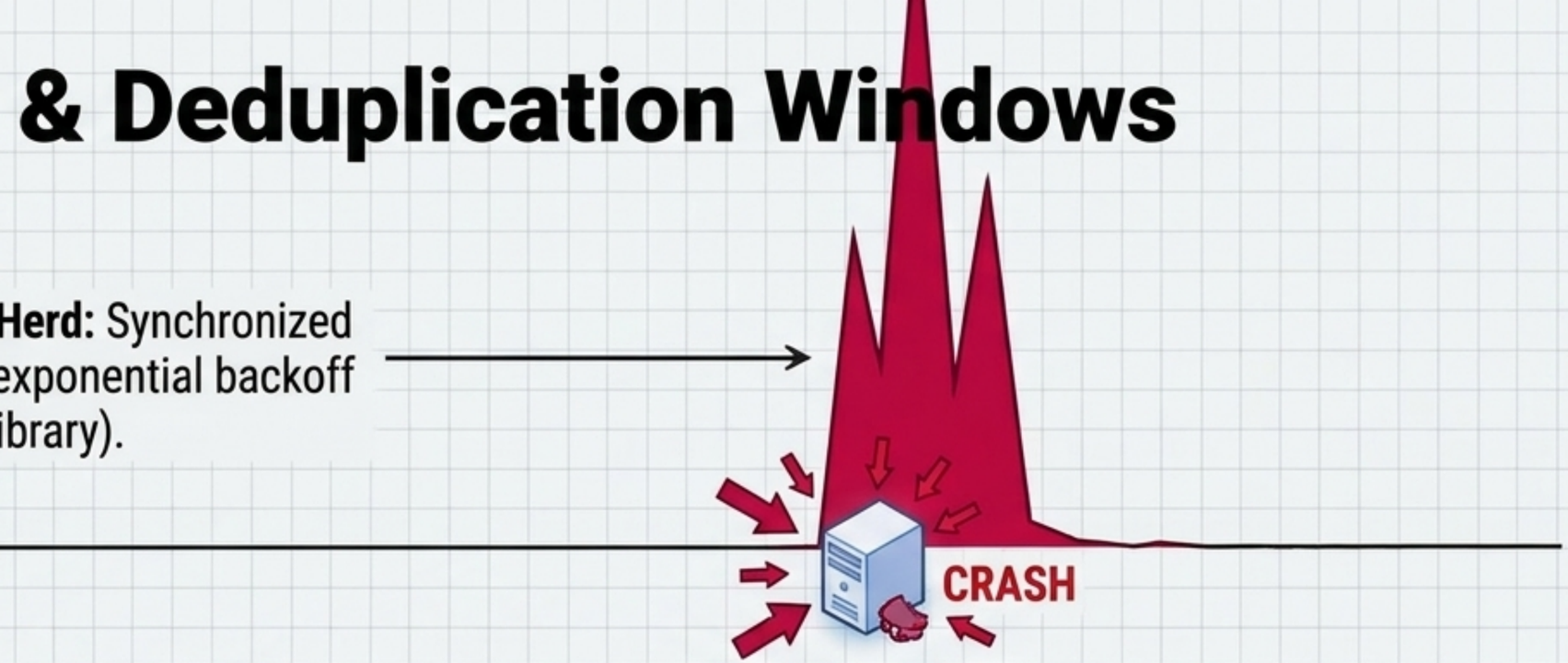
Failure Mode 1 (The Zombie Retry): A worker successfully charges a customer via Stripe, but crashes before committing the queue offset. Another worker picks up the uncommitted message and double-charges the customer.

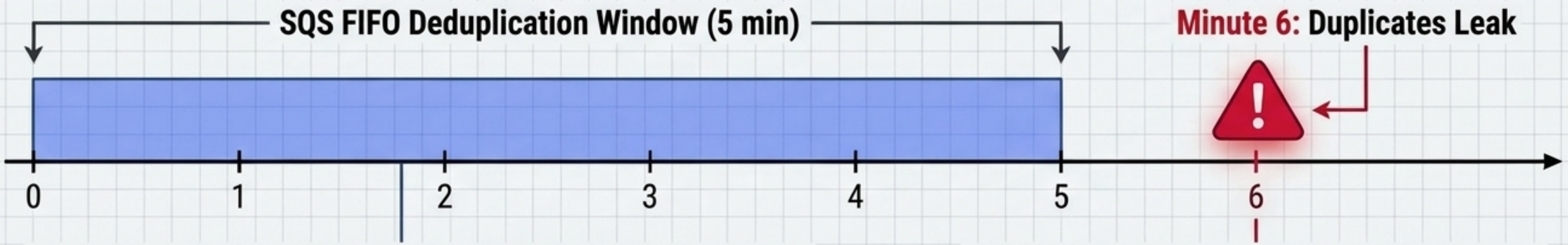


Gemini Notebook

Retry Storms & Deduplication Windows

Failure Mode 4: The Thundering Herd: Synchronized client retry floods occur without exponential backoff and “Full Jitter” (AWS Builders’ Library).





Failure Mode 2: SQS Windows: SQS FIFO deduplicates for exactly 5 minutes. If visibility timeout redeliveries or delayed retries occur at minute 6, duplicates leak through.

Failure Mode 3: Webhook Storms: Multi-day webhook retries from Stripe/Svix will brutally overwhelm receivers lacking idempotency.

Gemini Notebook

Section 6 — The Vibe-Coder Trap (CWE-367 TOCTOU)

Frontend Illusion

Classic Bug 1: Relying solely on useFormStatus or button disabling to prevent double-clicks. Network latency defeats UI blocking instantly.

```
function MyForm() {
  const { pending } = useFormStatus();

  const handleSubmit = async () => {
    // submit logic
  };

  return (
    <form action={handleSubmit}>
      <button type="submit" disabled={pending}>
        Submit
      </button>
    </form>
  );
}
```

The Useless Key

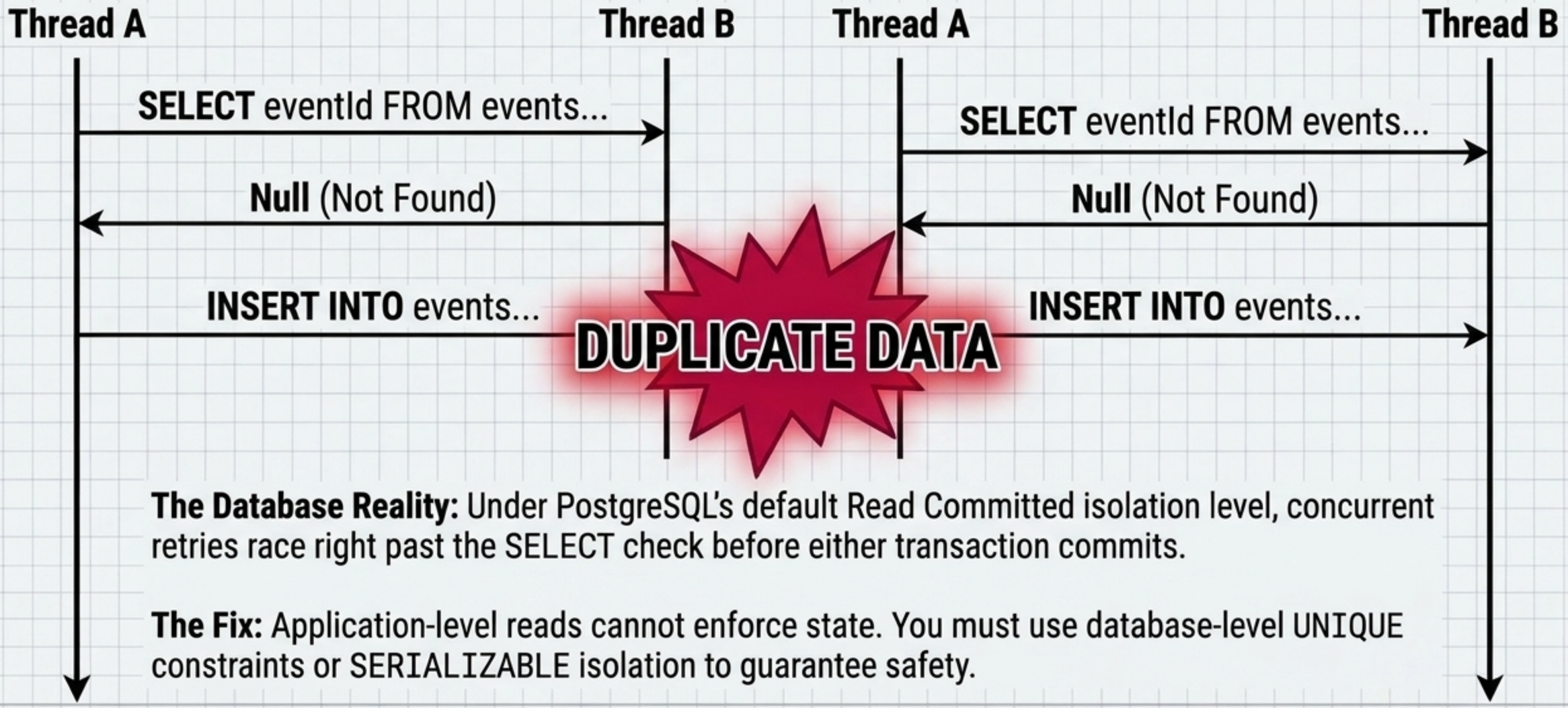
Classic Bug 2: Generating a brand-new uuidv4() inside the fetchWithRetry() loop on every attempt. This destroys the entire concept of an idempotency key, as the server sees every retry as a uniquely new request.

```
async function fetchWithRetry(url, data) {
  let attempts = 0;
  while (attempts < 3) {
    try {
      const idempotencyKey = uuidv4(); // <- ERROR!
      const response = await fetch(url, {
        method: 'POST',
        headers: {
          'Idempotency-Key': idempotencyKey
        },
        body: JSON.stringify(data)
      });
      if (response.ok) return response.json();
    } catch (error) {
      attempts++;
      await delay(1000 * attempts);
    }
  }
  throw new Error('Failed after 3 attempts');
}
```


Gemini Notebook

The Read Committed Race Condition

Classic Bug 3 (Backend): Writing `if (!await db.find(eventId)) { insert() }`



The Database Reality: Under PostgreSQL's default Read Committed isolation level, concurrent retries race right past the SELECT check before either transaction commits.

The Fix: Application-level reads cannot enforce state. You must use database-level `UNIQUE` constraints or `SERIALIZABLE` isolation to guarantee safety.

Gemini Notebook

Section 7 — Judicious AI Prompting

CLAUDE.md / AGENTS.md

Agent Invariants to Enforce:

> All POST mutations MUST accept and validate an 'Idempotency-Key' header.

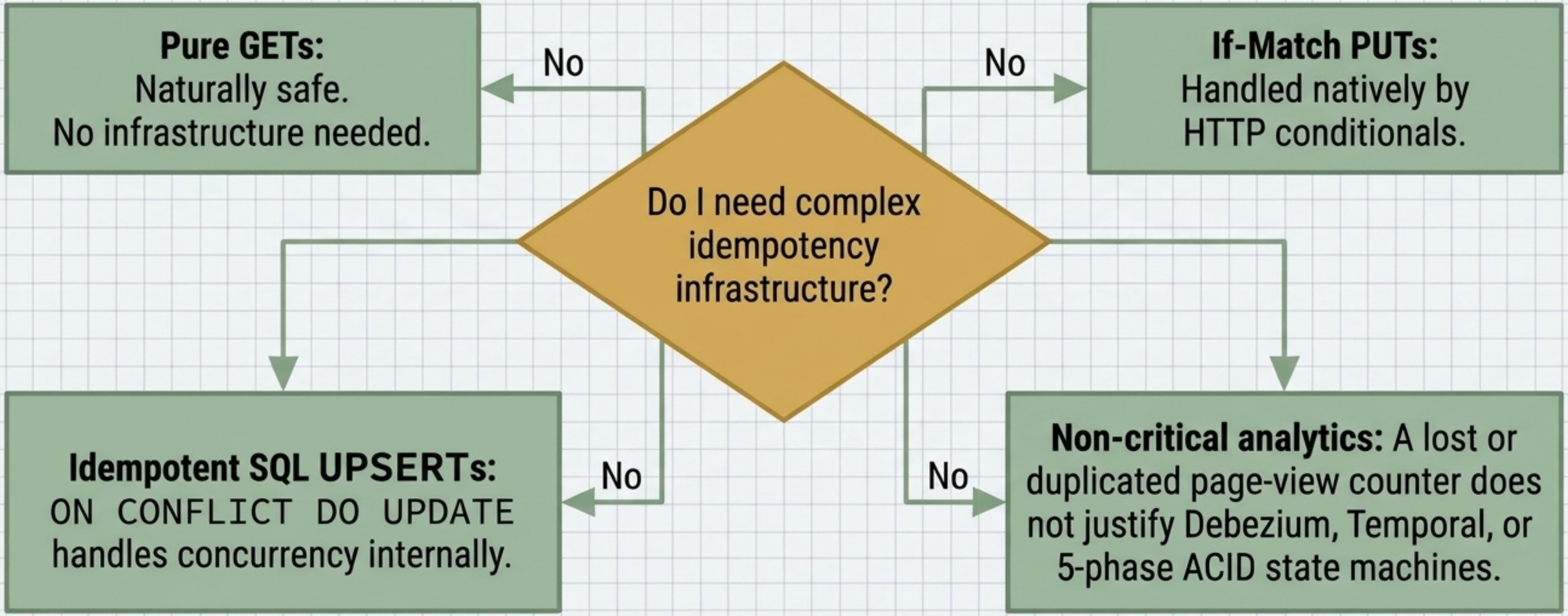
> Webhook handlers MUST deduplicate using 'event.id' via strict DB Unique Constraints, never application-level reads.

> Integration tests MUST fire parallel, single-packet duplicate-retry requests to assert constraint safety.

Gemini Notebook

The YAGNI Guardrails

When NOT to over-engineer (You Aren't Gonna Need It):



Gemini Notebook

Conclusion & Next Up

Final Takeaway: True system resilience is achieved by expecting every packet to fail, duplicate, or arrive late. We do not try to fix the network; we make the application immune to it.

Next Up in S08E03: Recursion & Tail-Call Optimization: Escaping the Stack Overflow.

Effective Once = At-Least-Once + Jitter + Idempotence



Gemini Notebook