Handling LLM API Failures

linkedin.com/in/dimplesharma21

Retry + Failover Strategies for Production Systems

Dimple Sharma

LLM-Specific Challenges

Challenge #1 Multiple Rate Limit Types

Traditional APIs: Single request limit

LLM APIs: *Dual limits* tracked separately

RPM (Requests Per Minute)
Number of API calls

TPM (Tokens Per Minute)
Total input + output tokens

→ Can hit token limit while under request limit (and vice versa)
→ Must monitor two different quotas

RPM	50/min (API calls)
TPM	38,000/min (tokens)

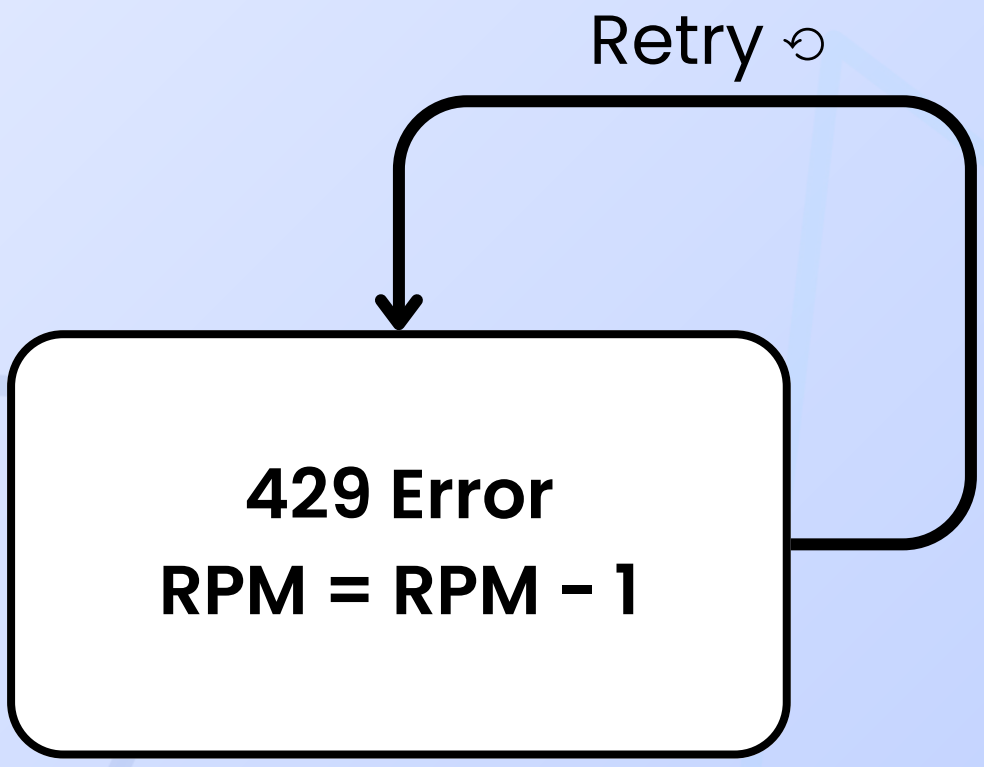
Eg: LLM API limits for Claude Sonnet 4 (Tier 1)

LLM-Specific Challenges

Challenge #2 Failed Requests Consume Quota

429 errors still count toward the request limit even though the request failed

→ Rapid retries drain RPM quota faster
→ Each failed retry = one request counted
→ Creates "death spiral" without proper strategy



"Death Spiral": Quota depletes with each failed attempt

LLM-Specific Challenges

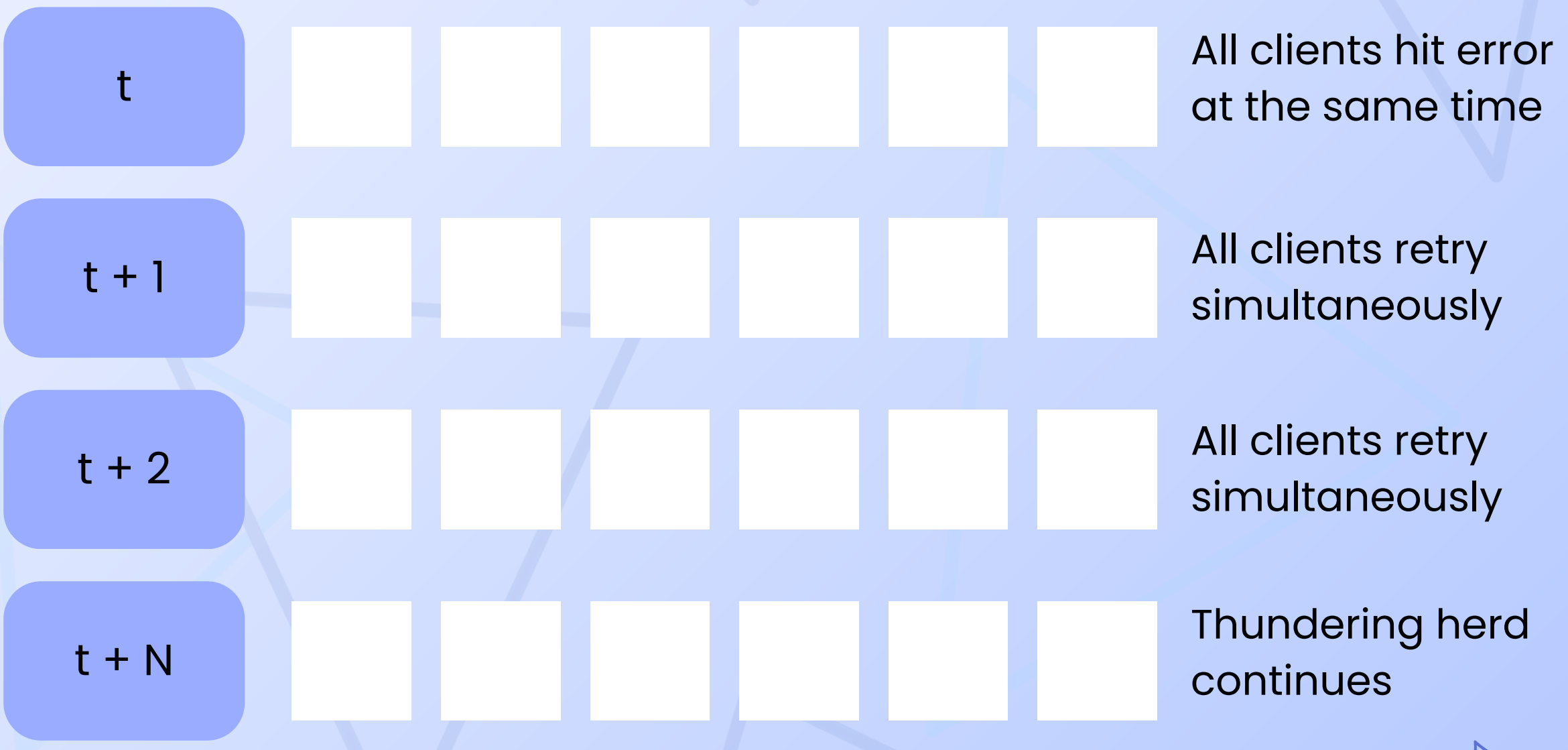
Challenge #3 Thundering Herd Problem

When failures happen:

Clients encounter error at same time (due to rate limit window resets, provider outage recovery, etc.)

All retry at the same instant, i.e., after same delay

→ Synchronized traffic spike
→ Server gets hit with concentrated load
→ More failures occur



Server gets overwhelmed with synchronised retries

LLM-Specific Challenges

Challenge #4 Variable Response Times

Traditional REST APIs:

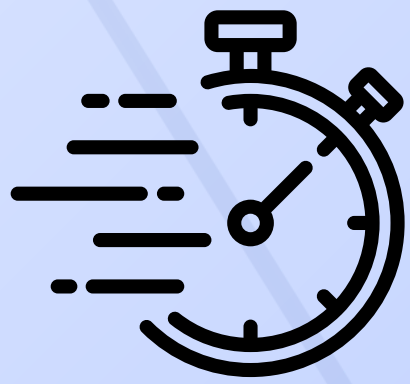
Predictable ≤ 1 second response time

LLM APIs:

Simple query: 2-5 seconds

Complex reasoning: 30+ seconds

(wide, unpredictable range)



Problem:

→ Can't set fixed timeout values

→ Simple queries would timeout unnecessarily

→ Complex queries might fail prematurely

Retry

Strategy #1: Exponential Backoff with Jitter

How it works:
→ Calculate retry delay on the client side
→ Increase delay exponentially after each failed attempt
→ Add random variance (jitter) to break synchronization
→ Set maximum delay cap to prevent indefinite waits

Formula:

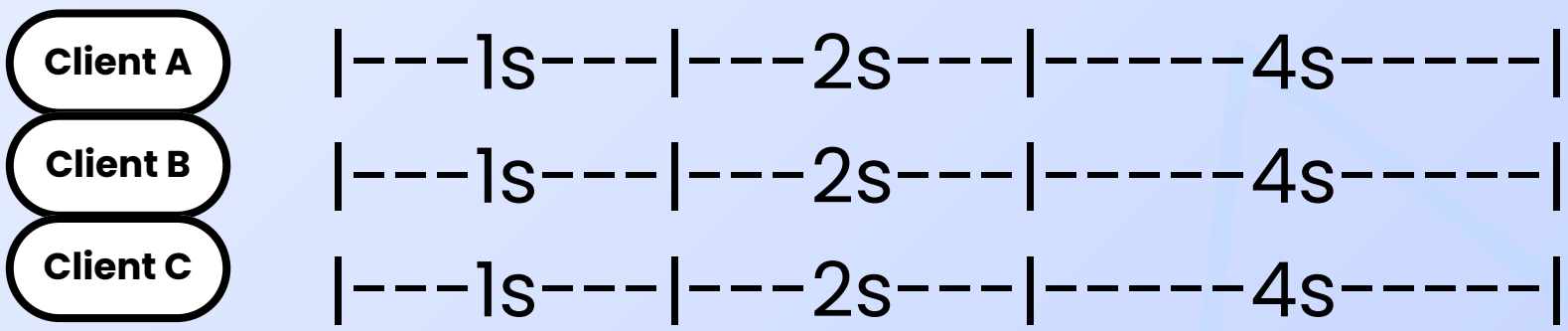
Wait = base_delay × (2^attempt) + jitter

Retry

Strategy #1: Exponential Backoff with Jitter

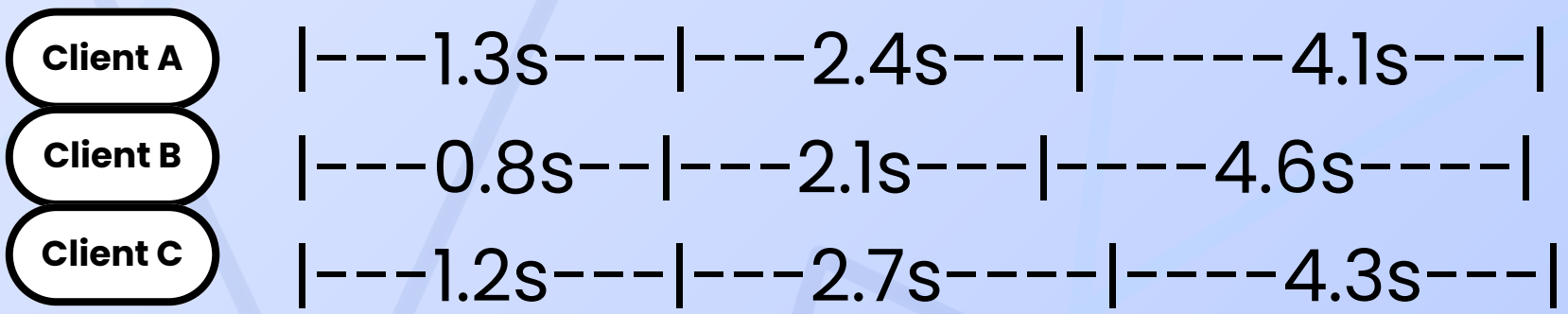
Benefits:
→ Breaks synchronization between clients
→ Prevents thundering herd (Challenge #3) and retry storms
→ Distributes load and avoids overwhelming recovering servers

Synchronized retries (without jitter)



All retry together → Traffic spikes

Randomized retries (with jitter)



Retries spread out → Smooth traffic

Retry

Strategy #2: Retry-After Header

How it works:
→ Provider returns retry-after header in 429 response
→ Specifies exact wait time in seconds

Benefits:
→ Server knows exact recovery time and reset window
→ Eliminates guessing (prevents wasted quota or unnecessary wait)
→ Both OpenAI and Anthropic provide this header

HTTP/1.1 429 Too Many Requests

```
retry-after: 20
anthropic-ratelimit-requests-limit: 50
anthropic-ratelimit-requests-remaining: 0
anthropic-ratelimit-requests-reset: 2024-12-08T10:05:00Z
```

Anthropic HTTP Response Header

Failover

Strategy #1 Provider Failover

Cross-provider switching for reliability

Primary → Backup : OpenAI → Anthropic

When to use:
⚡ Provider outage
⚡ Quota exhausted
⚡ Persistent failures (max retries failed)

Strategy #2 Model Category Failover

Maintain capability level within OR across providers

Fast → Fast
GPT-4o Mini → Claude 3.5 Haiku → Gemini Flash

Advanced → Advanced
GPT-4o → Claude Opus → Gemini Pro

When to use:
⚡ Ensure consistent user experience
⚡ Match performance tier during failover

Failover

Strategy #3 Context Window Fallback

If input exceeds model limit, auto-switch to larger context model

GPT-4o (128K) → GPT-4.1 (1M)

When to use:

⚡ Processing large documents or codebases
⚡ User uploads exceed current model capacity
⚡ Avoid request failures due to token limits

Strategy #4 Latency-Based Routing

Route to fastest available provider

Based on continuous monitoring of response time, error rates, availability of various models

When to use:

⚡ Real-time applications (chatbots, voice AI, etc.)
⚡ Performance-critical workflows

Failover Trade-offs

What to Consider

QUALITY
Output consistency across providers varies
Different models = different styles = different responses

LATENCY
Every failover adds delay
Switching models = new API call
Applies within or across providers

COST
Different providers = different rates
Track spend across fallback chain
Example: GPT-4 vs Claude pricing

Dimple Sharma

Building Production Grade LLM Systems

Rate limits !

Quota exhaustion !

Context overflow !

Latency spikes !

Server errors !

LLM systems fail in dozens of ways !

Failure handling isn't optional, it's what separates prototypes from production

Retry for transient failures
Failover for persistent failures

What's your take on building resilient systems?
💬 Share in the comments

Dimple Sharma