**System Design: Flexible Rate Limiter Service**

**1. Requirements & SLAs**

* Default policy for most users (e.g., 100 req/min).
* Special policy for VIP users (e.g., bieber-id → 1,000,000 req/min).
* Must enforce fairness:
  + If a user **hits rate limit**, block further requests until reset.
  + Return **HTTP 402 (Payment Required)** with a **wait\_time** (cool-down).
* Support **different user types**:
  + **Normal users** (strict limits).
  + **VIP users** (looser limits).
* **Leaky Bucket**:
  + Fill 15 tokens/minute.
  + If empty, reject until next refill.
* **Latency goal**:
  + Under **200–500 ms** for requests not hitting limit.
* **Correctness**:
  + Handle multiple clients using the same userId.
  + Shared counters (via Redis) to avoid bypassing limits.

**2. API Design**

**Endpoint**

GET /get\_status?user\_id={id}&token={tkn}

**Response**

{

"user\_id": "123",

"allowed": true,

"remaining": 5,

"wait\_time": 0

}

Or, if limit hit:

{

"user\_id": "123",

"allowed": false,

"remaining": 0,

"wait\_time": 60

}

**3. Data Model (Redis)**

{

"user\_id": "123",

"count": 5,

"last\_bucket\_filled\_time": "2025-09-16T15:55:00Z",

"limit": 10

}

**4. Python Implementation (with Redis)**

import time

import redis

from flask import Flask, request, jsonify

app = Flask(\_\_name\_\_)

r = redis.Redis(host="localhost", port=6379, db=0)

class RateLimiter:

def \_\_init\_\_(self, user\_id, limit, window):

self.user\_id = user\_id

self.limit = limit

self.window = window # seconds

def allow\_request(self):

now = int(time.time())

key = f"user:{self.user\_id}"

user\_data = r.hgetall(key)

if not user\_data:

# First request

r.hmset(key, {"count": 1, "start\_time": now})

r.expire(key, self.window)

return True, self.limit - 1, 0

count = int(user\_data.get(b"count", 0))

start\_time = int(user\_data.get(b"start\_time", now))

if now - start\_time >= self.window:

# Reset window

r.hmset(key, {"count": 1, "start\_time": now})

r.expire(key, self.window)

return True, self.limit - 1, 0

if count < self.limit:

r.hincrby(key, "count", 1)

return True, self.limit - (count + 1), 0

else:

wait\_time = self.window - (now - start\_time)

return False, 0, wait\_time

@app.route("/get\_status")

def serve\_status():

user\_id = request.args.get("user\_id", "guest")

# Example: different policies

if user\_id == "bieber-id":

limiter = RateLimiter(user\_id, limit=1\_000\_000, window=60)

else:

limiter = RateLimiter(user\_id, limit=100, window=60)

allowed, remaining, wait\_time = limiter.allow\_request()

return jsonify({

"user\_id": user\_id,

"allowed": allowed,

"remaining": remaining,

"wait\_time": wait\_time

})

if \_\_name\_\_ == "\_\_main\_\_":

app.run(port=8000, debug=True)

**5. Scalability**

* **Redis**: Centralized counter store (fast in-memory, distributed).
* **Sharding by user\_id**: Multiple Redis instances.
* **Stateless service**: Each rate-limiter instance just queries Redis → easy horizontal scaling.
* **Consistency**: Use atomic Redis ops (INCR, HINCRBY) to avoid race conditions.
* **High availability**: Redis cluster with replication + failover.

**6. Operability**

* **Monitoring**:
  + Track request counts, reject counts, latency.
  + Export Prometheus metrics (rate\_limiter\_rejects, rate\_limiter\_latency).
* **Alerting**:
  + If reject ratio > X% → raise alert.
* **Logging**:
  + Log rejected requests with user\_id and wait\_time.

**7. Diagram**

Clients ---> Load Balancer ---> Rate Limiter Service ---> Redis ---> Decision

| (stateless API)

|---> Backend services (if allowed)

✅ With this, you have: **Requirements + API + Redis schema + Python code + scalability/operability notes.**

👉 Do you want me to also show you how to extend this to a **Token Bucket algorithm** (smoother bursts) instead of the Leaky Bucket?

**You said:**

Is there any shortcomings on this

**ChatGPT said:**

Yes 👍 — while the design and Python + Redis implementation is solid, there are some **shortcomings and trade-offs** you should be aware of.

**🔹 1. Redis Bottlenecks**

* Every request triggers a **Redis read/write**, which can become a bottleneck under very high QPS (queries per second).
* High churn users (millions of API calls/minute) may overload Redis.
* Fixes:
  + Use **Redis pipelines** or **Lua scripts** to batch commands.
  + Introduce **local in-memory caching** for users with low limits.
  + Use a **sharded Redis cluster**.

**🔹 2. Clock Skew & Window Reset**

* Using a **fixed window** (e.g., reset every minute) creates unfairness:
  + A user can send **100 requests at 0:59s and 100 at 1:01s** = **200 requests in 2 seconds**.
* Fixes:
  + Use **sliding window log** or **token bucket** instead of fixed window.
  + Token bucket smooths out bursts better.

**🔹 3. Multiple Clients per User**

* If the same user ID is shared across many clients, each client still increments the same Redis counter.
* That’s correct for fairness, but it could:
  + Cause **hot keys** in Redis (all writes to one user\_id).
  + Create latency spikes.
* Fix:
  + Distribute keys (e.g., user\_id:shard), then aggregate.
  + Or use a **hash partitioning strategy** in Redis Cluster.

**🔹 4. Single Point of Failure**

* If Redis is down, **rate limiting fails**.
* Fixes:
  + Fallback mode (default allow but log violations).
  + Redis replication + failover (Sentinel/Cluster).
  + Use local counters with eventual consistency.

**🔹 5. Operational Limits**

* Redis expire approach may lead to stale data if expiry isn’t set correctly.
* Requires tuning of **memory eviction policies** (volatile-lru, etc.).
* Monitoring must ensure **no silent evictions** of active keys.

**🔹 6. API & Error Handling**

* Currently returns 402 — uncommon for rate limiting (most systems use 429 Too Many Requests).
* Wait time granularity is **per second**. Could be more precise (ms).
* No global fallback if Redis is overloaded.

**🔹 7. Scalability Trade-offs**

* Design is **CP (Consistent + Partition Tolerant)** but may sacrifice **Availability** if Redis is down.
* At Twitter/Facebook scale, centralized Redis may not be enough. They use **distributed counters** with probabilistic data structures (like **Approximate Counters / Sketches**).

**✅ Summary of Shortcomings**

1. Redis dependency = bottleneck + SPOF.
2. Fixed window unfairness (burst at boundaries).
3. Hot keys for popular users.
4. Operational tuning (expiry, memory).
5. Unconventional HTTP code (should be 429).
6. Doesn’t yet handle **global rate limits** (e.g., “all requests system-wide < X QPS”).
7. No defense against **distributed attacks** (botnets spreading requests across IDs).