API Rate Limiting & Performance Optimization

Strategies for handling rate limits and optimizing API usage

Version: 1.0

Author: Platform Engineering Team

Category: API Performance | Rate Limiting

Last Updated: December 12, 2025

This guide explains Monzo API rate limiting mechanisms, provides strategies for handling rate limits gracefully, and offers performance optimization techniques to minimize API calls while maintaining excellent user experience. Understanding rate limits is critical for building production-grade integrations that scale reliably.

Rate Limit Specifications

Standard Rate Limits

Monzo enforces the following rate limits per access token:

Endpoint Category	Rate Limit	Window	Applies To
Account queries	1,000 req/hour	60 minutes	GET /accounts, /balance
Transaction queries	2,000 req/hour	60 minutes	GET /transactions
Payment creation	100 req/hour	60 minutes	POST /payments
Webhook registration	10 req/hour	60 minutes	POST /webhooks
OAuth token refresh	50 req/hour	60 minutes	POST /oauth2/token
Sandbox endpoints	Unlimited	N/A	All sandbox requests

Rate Limit Headers

Every API response includes rate limit information in HTTP headers:

```
X-RateLimit-Limit: 1000
X-RateLimit-Remaining: 847
X-RateLimit-Reset: 1702389600
Retry-After: 3600
```

X-RateLimit-Limit: Maximum requests allowed in current window

X-RateLimit-Remaining: Number of requests left in current window

X-RateLimit-Reset: Unix timestamp when rate limit resets

Retry-After: Seconds to wait before retrying (only when rate limited)

Handling Rate Limit Errors

HTTP 429 Response

When rate limit is exceeded, Monzo returns HTTP 429 Too Many Requests:

```
{
  "error": "too_many_requests",
  "message": "Rate limit exceeded for endpoint /transactions",
  "retry_after": 3600,
  "reset_time": "2024-12-12T15:00:00Z"
}
```

Exponential Backoff Implementation

Implement exponential backoff with jitter to handle rate limits gracefully:

```
import time
import random
import requests

def make_api_request_with_backoff(url, headers, max_retries=5):
    """
    Make API request with exponential backoff on rate limits.
    """
    for attempt in range(max_retries):
        response = requests.get(url, headers=headers)

        if response.status_code == 200:
            return response.json()

        elif response.status_code == 429:
            # Rate limited
            retry_after = int(
                response.headers.get('Retry-After', 60)
            )

            # Add jitter (random 0-30% of wait time)
            jitter = random.uniform(0, retry_after * 0.3)
            wait_time = retry_after + jitter

            print(f"Rate limited. Waiting {wait_time:.1f}s")
            time.sleep(wait_time)

        else:
            # Other error
            response.raise_for_status()

    raise Exception("Max retries exceeded")
```

Client-Side Rate Limiter

Token Bucket Algorithm

Implement client-side rate limiting to prevent hitting API limits:

```
import time
from threading import Lock

class RateLimiter:
    """
    Token bucket rate limiter.
    """
    def __init__(self, rate, capacity):
        self.rate = rate  # tokens per second
        self.capacity = capacity  # max tokens
        self.tokens = capacity
        self.last_update = time.time()
        self.lock = Lock()

    def acquire(self, tokens=1):
        """
        Acquire tokens. Blocks until available.
        """
        with self.lock:
            while True:
                now = time.time()
                elapsed = now - self.last_update

                # Refill tokens
                self.tokens = min(
                    self.capacity,
                    self.tokens + elapsed * self.rate
                )
                self.last_update = now

                if self.tokens >= tokens:
                    self.tokens -= tokens
                    return

                # Wait for tokens to refill
                sleep_time = (tokens - self.tokens) / self.rate
                time.sleep(sleep_time)

# Usage
limiter = RateLimiter(rate=1000/3600, capacity=1000)

def fetch_transactions():
    limiter.acquire()  # Wait if necessary
    response = requests.get(
        'https://api.monzo.com/transactions',
        headers={'Authorization': f'Bearer {token}'}
    )
    return response.json()
```

Performance Optimization Strategies

1. Implement Caching

Cache frequently accessed, slowly-changing data with appropriate TTL:

```
import redis
import json

redis_client = redis.Redis(host='localhost', port=6379)

def get_account_balance_cached(account_id, token):
    """
    Get balance with 60-second cache.
    """
    cache_key = f"balance:{account_id}"

    # Try cache first
    cached = redis_client.get(cache_key)
    if cached:
        return json.loads(cached)

    # Fetch from API
    response = requests.get(
        f'https://api.monzo.com/balance?account_id={account_id}',
        headers={'Authorization': f'Bearer {token}'}
    )
    data = response.json()

    # Cache for 60 seconds
    redis_client.setex(cache_key, 60, json.dumps(data))

    return data
```

2. Batch Requests

Combine multiple operations into single API calls when possible:

```
# Bad: Multiple individual requests
for tx_id in transaction_ids:
    tx = get_transaction(tx_id)
    process(tx)

# Good: Single request with pagination
params = {
    'account_id': account_id,
    'since': since_date,
    'limit': 100
}
response = get_transactions(params)
transactions = response['transactions']
for tx in transactions:
    process(tx)
```

3. Use Webhooks Instead of Polling

Replace polling patterns with webhooks for real-time updates:

```
# Bad: Poll every 30 seconds
while True:
    transactions = fetch_new_transactions()
    process(transactions)
    time.sleep(30)  # Wastes 2,880 API calls/day

# Good: Use webhooks
@app.post('/webhooks/monzo')
```

```
def handle_webhook(request):
    event = verify_and_parse_webhook(request)
    if event.type == 'transaction.created':
        process_transaction(event.data)
    return {'status': 'ok'}
# Uses 0 polling API calls
```

4. Pagination Best Practices

Fetch data efficiently with proper pagination:

```
def fetch_all_transactions(account_id, token):
    """
    Fetch all transactions with efficient pagination.
    """
    all_transactions = []
    params = {
        'account_id': account_id,
        'limit': 100  # Max page size
    }

    while True:
        response = requests.get(
            'https://api.monzo.com/transactions',
            headers={'Authorization': f'Bearer {token}'},
            params=params
        )
        data = response.json()

        all_transactions.extend(data['transactions'])

        # Check if more pages exist
        if not data.get('has_more'):
            break

        # Update cursor for next page
        params['since'] = data['transactions'][-1]['created']

    return all_transactions
```

Monitoring & Alerting

Track Rate Limit Usage

```
import logging
from datetime import datetime

class RateLimitMonitor:
    """
    Monitor rate limit usage and alert when close to limits.
    """
    def __init__(self, threshold=0.8):
        self.threshold = threshold
        self.logger = logging.getLogger(__name__)

    def check_rate_limit(self, response):
        """
        Check rate limit headers and log warnings.
        """
```

```
limit = int(response.headers.get('X-RateLimit-Limit', 0))
remaining = int(
    response.headers.get('X-RateLimit-Remaining', 0)
)

if limit == 0:
    return

usage = (limit - remaining) / limit

if usage >= self.threshold:
    reset_time = response.headers.get('X-RateLimit-Reset')
    self.logger.warning(
        f"High rate limit usage: {usage:.1%}. "
        f"Remaining: {remaining}/{limit}. "
        f"Resets at: {reset_time}"
    )

    # Alert operations team
    if usage >= 0.95:
        self.send_alert(
            f"CRITICAL: Rate limit nearly exhausted. "
            f"Only {remaining} requests remaining."
        )
```

Best Practices Summary

1. Always respect Retry-After headers in 429 responses

2. Implement client-side rate limiting to stay under limits

3. Cache data with appropriate TTL to reduce API calls

4. Use webhooks instead of polling for real-time updates

5. Batch operations when possible to reduce request count

6. Monitor rate limit usage and alert at 80% threshold

7. Use exponential backoff with jitter for retry logic

8. Fetch only required data fields to reduce response time

9. Implement proper pagination for large datasets

10. Test rate limit handling in development/staging environments

> **WARNING: Rate limits are enforced per access token. If you manage multiple customers, each customer needs separate token management to avoid one user exhausting shared rate limits.**