# API Monetization Gateway Solution Design

This document outlines the proposed system architecture and data model for the API Monetization Gateway. The solution is designed to be scalable, maintainable, and to demonstrate key concepts related to API monetization.

**1. System Architecture**

The proposed architecture is a multi-tiered approach. It is designed to be highly available and to handle a large volume of requests efficiently.

* **Client Application:** The external user's application making API requests.
* **API Gateway (Our Solution):** A .NET Core Web API that sits between the client and the internal services. This is the core of our solution. It will perform several key functions:
  + **Authentication & Authorization:** Verify the client's credentials and ensure they have permission to access the requested endpoint.
  + **Rate Limiting Middleware:** A custom middleware that intercepts every request to enforce the rate limit (requests per second) and monthly quota. This middleware will use a distributed cache (like Redis) for high-speed, in-memory tracking of usage, which is critical for real-time rate limiting.
  + **Usage Tracker Service:** A service that logs every successful API request to a persistent database (e.g., PostgreSQL or SQL Server).
  + **Proxy:** Forwards the validated and unblocked requests to the appropriate internal services.
* **Internal Services:** The actual backend services that provide the core functionality.
* **Distributed Cache (e.g., Redis):** Used for storing real-time, short-lived data required for rate limiting. This includes the number of requests per second for a given customer and their current monthly usage count.
* **Database:** A relational database to store permanent data, including:
  + Tier configurations (monthly quota, rate limit, pricing).
  + Customer and user information, including their assigned tier.
  + Historical API usage logs.
  + Monthly billing summaries.
* **Background Job Processor (e.g., Hangfire, ASP.NET Core Background Service):** A service that runs on a schedule (e.g., once a month) to process usage logs, calculate billing, and save the summary to the database.

**High-Level Flow:**

1. A client makes an API call to the gateway.
2. The gateway's rate limiting middleware checks the request against the customer's tier configuration and current usage stored in Redis.
3. If the request exceeds the limits, the middleware returns HTTP 429 Too Many Requests.
4. If the request is within the limits, the middleware allows the request to proceed.
5. The gateway's usage tracker service logs the request to the database.
6. The gateway forwards the request to the internal service.
7. Once a month, the background job processor queries the database, aggregates the usage, calculates the cost, and saves the billing summary.

# Flow Diagram

# Entity Relationship Diagram