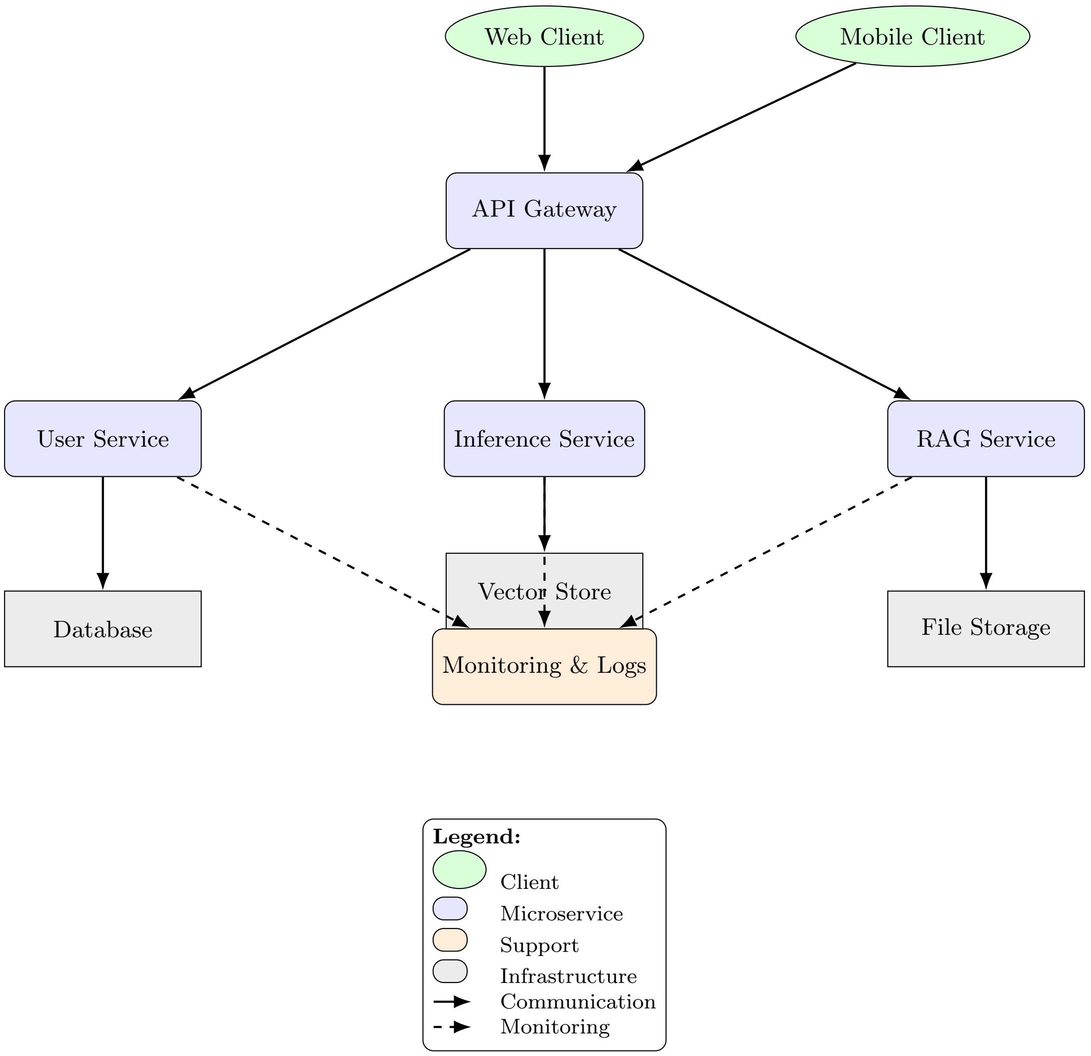
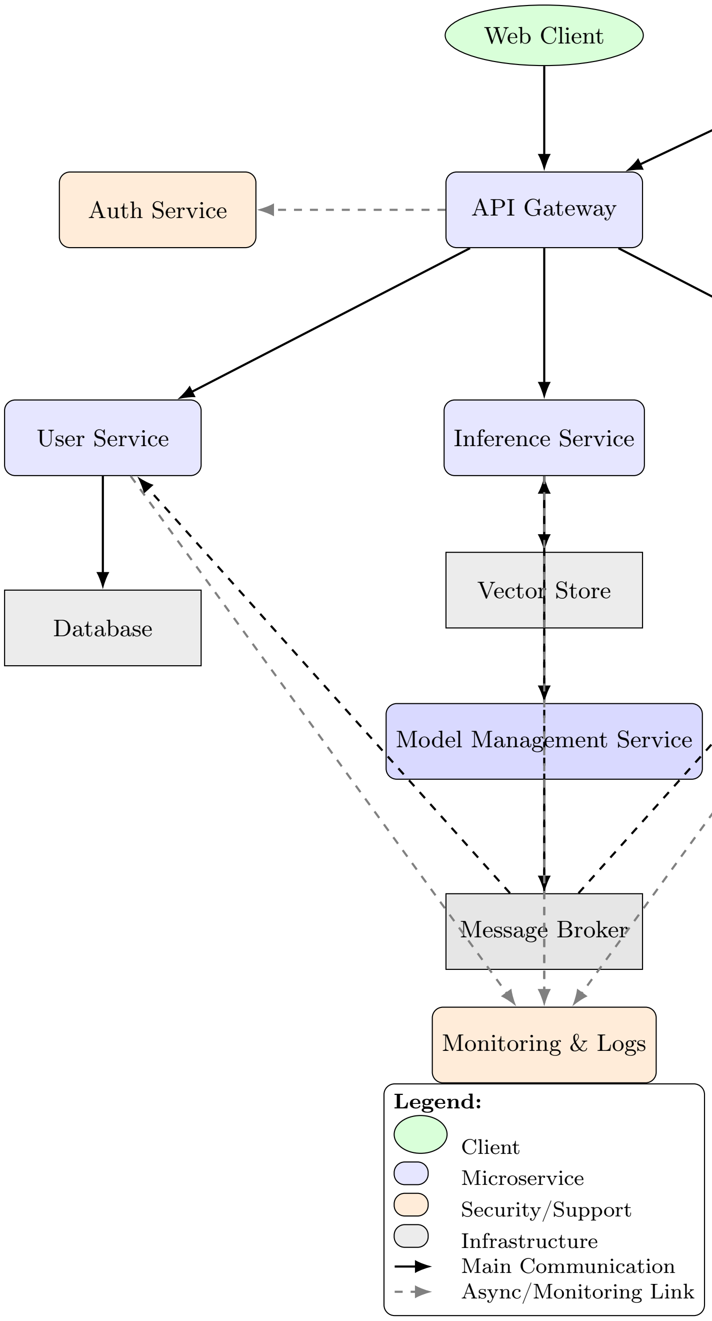
Simplified Microservices Architecture – DeepSeek



1 Enhanced Microservices Architectures - DeepSeek



2 API Gateway Analysis

Role and Definition

The API Gateway acts as a single entry point for clients. It centralizes access to microservices and ensures critical functions such as routing, security, and data aggregation.

Main Responsibilities

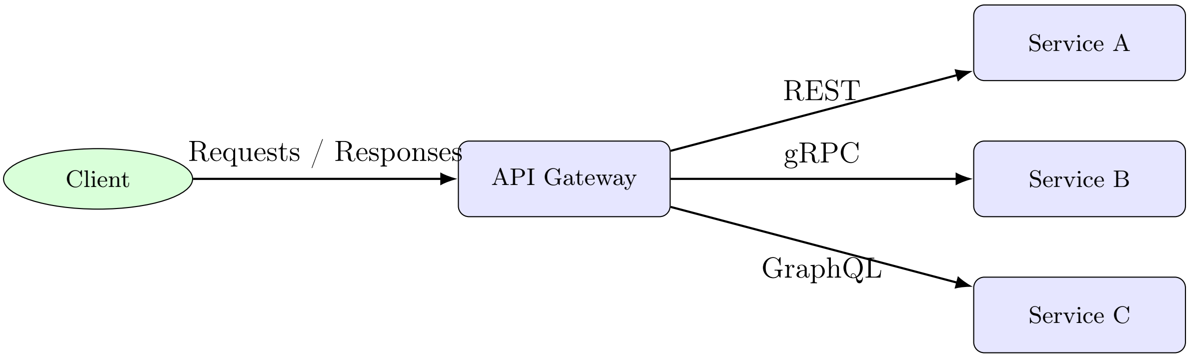
- Request routing to microservices.
- Aggregation of responses from multiple services.
- Authentication and authorization (OAuth2, JWT).
- Protocol transformation (REST, gRPC, WebSocket).
- Metrics and logs collection for monitoring.
- Rate limiting and throttling.

Advantages

- Simplifies client access with a single entry point.
- Centralizes security and global policies.
- Supports response aggregation and improves performance.
- Facilitates scalability and modularity.

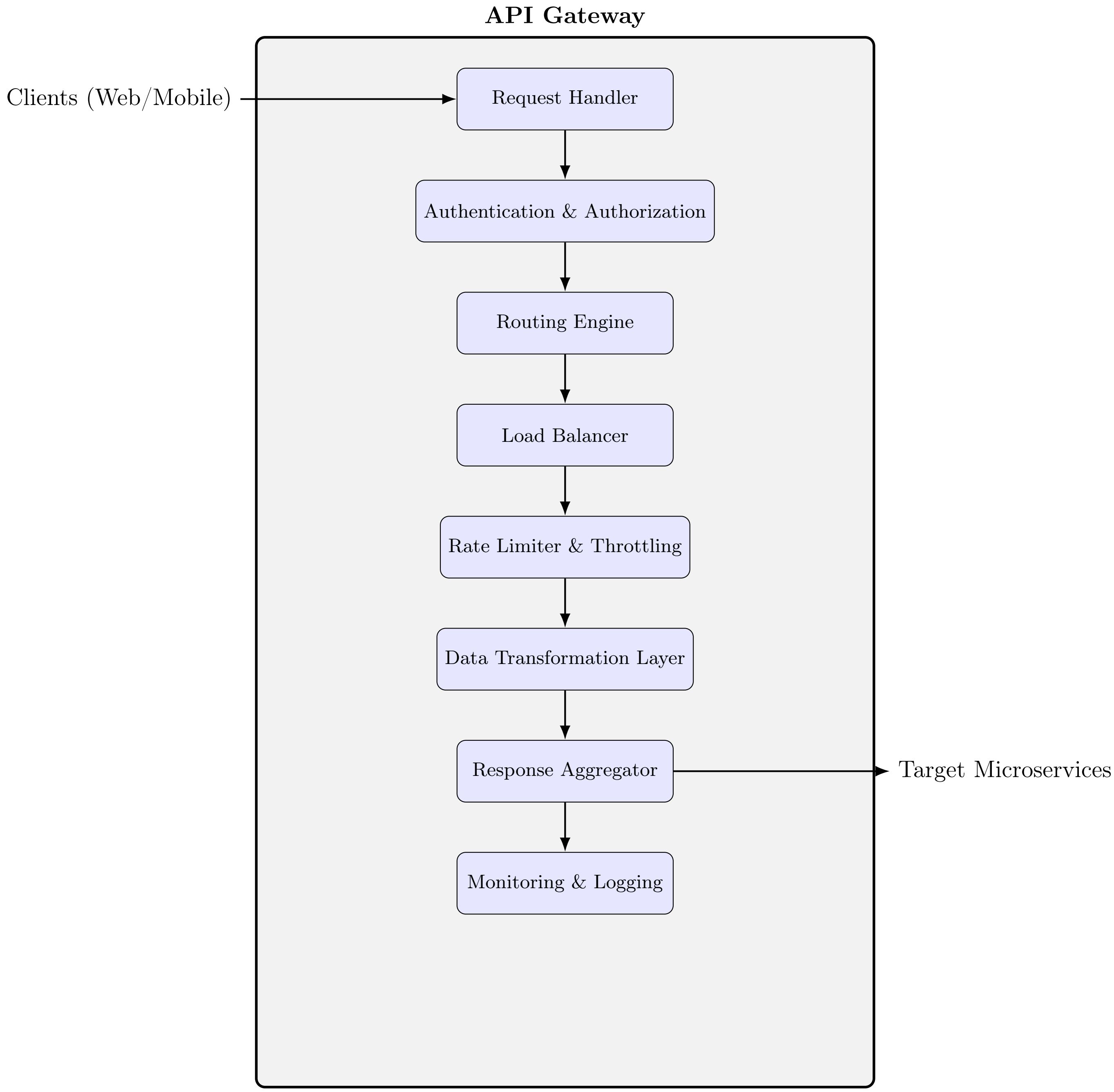
Limitations

- Risk of **Single Point of Failure**.
- Additional latency.
- Complexity in configuration and maintenance.



figureUML diagram of the API Gateway and its interactions.

3 Internal Architecture of the API Gateway



4 Detailed Analysis — Internal Architecture of the API Gateway

4.1 Objectives and Requirements

The API Gateway plays a central role in controlling and orchestrating client access to DeepSeek's microservices.

Functional Objectives:

- Provide a unified and stable entry point for all clients (Web, Mobile, Third-party APIs).
- Authenticate and authorize requests.
- Route requests to the appropriate services and aggregate responses when necessary.
- Perform protocol and data format transformations (REST ↔ gRPC, JSON, GraphQL).

- Enforce policies (rate limiting, quotas, canary routing).

Non-Functional Requirements:

- High availability (replicas, multi-zone deployment).
- Low latency and ability to handle traffic spikes.
- Observability (metrics, logs, traces).
- Security (TLS, validation, WAF).
- Easy configuration and policy enforcement.

4.2 Internal Components — Detailed Description

Request Handler : entry point receiving HTTP/WS requests, extracting headers, body, correlation IDs; performs syntactic validations (size, JSON schema) and applies basic protections (payload limits).

Authentication & Authorization : verifies JWT/OAuth2 (signature via JWKS) or opaque tokens (introspection with an Auth Service). Enforces scope/role checks (RBAC/ABAC).

Rate Limiter & Throttling : applies quotas per key (IP, API key, userId). Common implementations: token-bucket or leaky-bucket, atomic counters in Redis (Lua scripts) for cluster safety.

Input Validation & Sanitization : schema validation, header sanitization, normalization (unicode, trim), injection protection.

Routing Engine / Service Discovery : maps requests (path, host, header) to target services. Integrates discovery (Consul / Kubernetes API) and routing rules (versions, canary, A/B testing).

Load Balancer / Upstream Manager : selects upstream instance (RoundRobin, least-connections, weighted), manages persistent connections (keep-alive), handles timeouts.

Circuit Breaker & Retry Policy : protects against degraded upstreams. Key parameters: failure threshold, time window, reset timeout, exponential backoff retry logic.

Protocol / Data Transformation : converts REST ↔ gRPC, shapes JSON, maps schemas, enriches responses, strips sensitive fields before sending.

Response Aggregator / Composer : orchestrates parallel service calls, merges results, supports partial responses and fallback strategies.

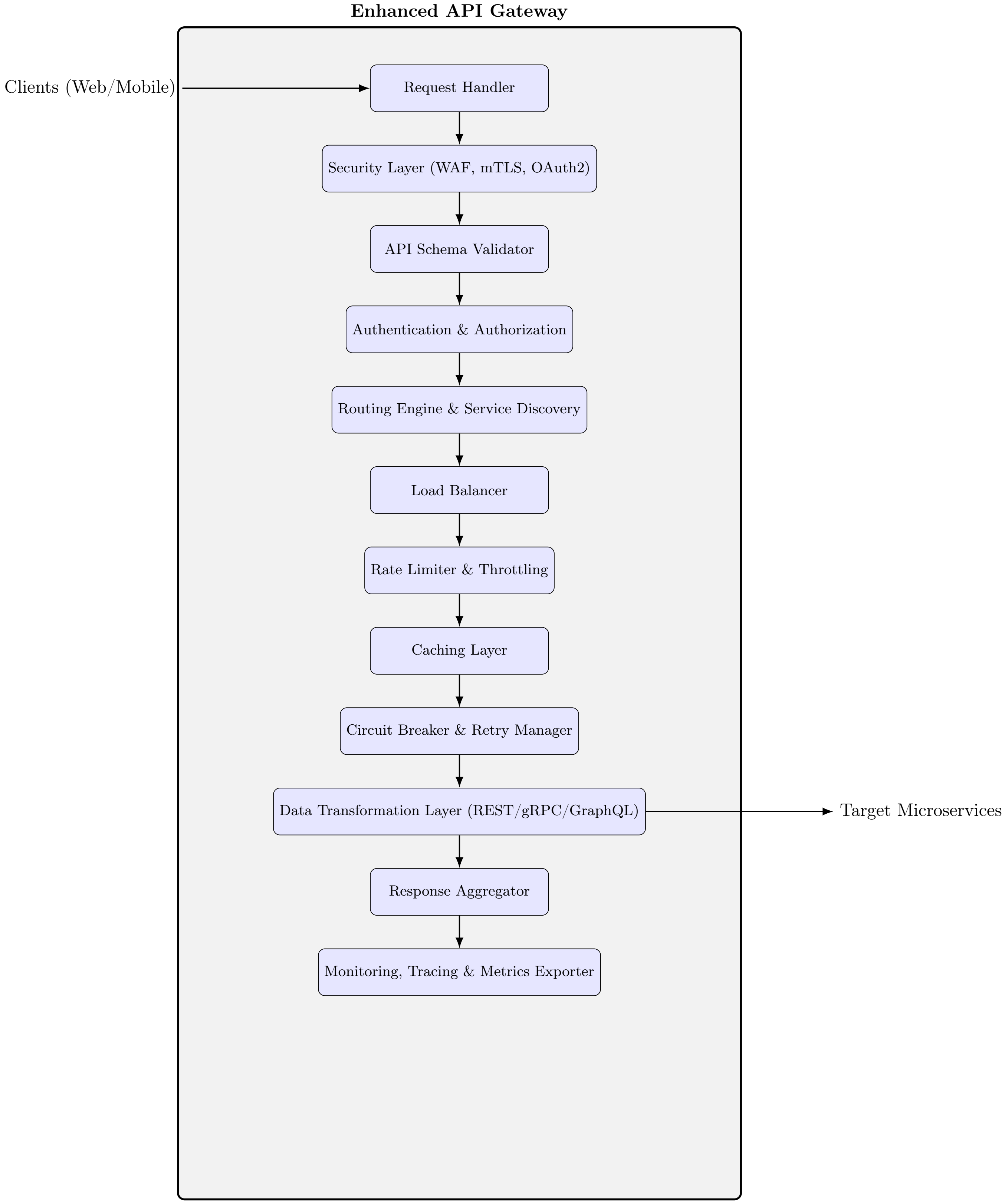
Monitoring & Logging : exports structured logs, traces, and metrics (Prometheus, OpenTelemetry) for observability.

5 Proposed Improvements for the API Gateway

The **DeepSeek API Gateway** can be enhanced to improve security, performance, and governance. Key additions include:

- **Security:** WAF (Web Application Firewall), mTLS support, OAuth2 validation.
- **Performance:** distributed caching, intelligent load balancing, circuit breaker and retry logic.
- **Observability:** distributed tracing (OpenTelemetry), advanced monitoring, Prometheus metrics.
- **Flexibility:** multi-protocol support (REST, gRPC, GraphQL), data aggregation and advanced transformation.
- **Governance:** API schema validation (OpenAPI/JSON Schema), quotas, versioning, and service discovery.

Enhanced Internal Architecture of the API Gateway


Enhanced API Gateway
Clients (Web/Mobile)
Request Handler
Security Layer (WAF, mTLS, OAuth2)
API Schema Validator
Authentication & Authorization
Routing Engine & Service Discovery
Load Balancer
Rate Limiter & Throttling
Caching Layer
Circuit Breaker & Retry Manager
Data Transformation Layer (REST/gRPC/GraphQL)
Target Microservices
Response Aggregator
Monitoring, Tracing & Metrics Exporter

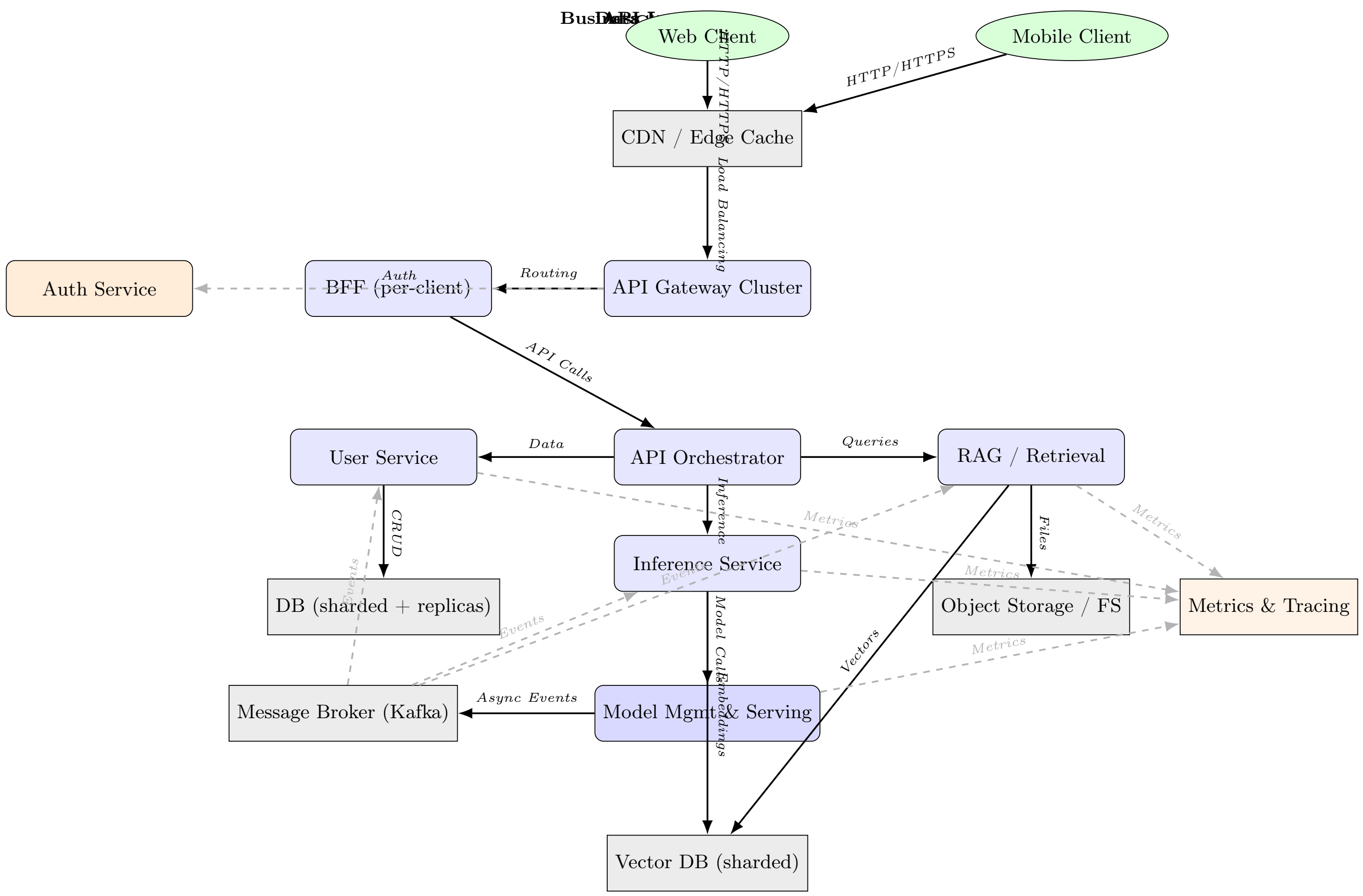
6 Client Request Execution Scenario - DeepSeek API Gateway

Step-by-Step Scenario

1. **Client Request:** Web/Mobile client sends a request (REST/GraphQL) with JWT/OAuth2 token.

2. **Request Handler:** Validates headers, payload, and assigns correlation ID.
3. **Security Layer:** WAF blocks malicious patterns, mTLS ensures client identity, OAuth2 token validation.
4. **API Schema Validator:** Validates request against OpenAPI/JSON Schema.
5. **Authentication & Authorization:** Checks RBAC/ABAC, denies unauthorized requests.
6. **Routing Engine & Service Discovery:** Maps request to appropriate microservices.
7. **Load Balancer:** Selects upstream instances, manages connections and timeouts.
8. **Rate Limiter & Throttling:** Applies per-user/API key quotas.
9. **Caching Layer:** Returns cached response if available, otherwise forwards request.
10. **Circuit Breaker & Retry:** Protects upstream services, retries failed calls, applies fallback if needed.
11. **Data Transformation Layer:** Converts REST ↔ gRPC/GraphQL, enriches request.
12. **Microservice Calls:** Calls User, Inference, and RAG services in parallel.
13. **Response Aggregator:** Combines results into a unified response, handles partial failures.
14. **Monitoring & Metrics:** Logs request/response, updates traces and metrics.
15. **Client Response:** Sends aggregated response back to the client securely (TLS).

7 Parallel Microservices Architecture in DeepSeek



The illustrated architecture represents the **Parallel Microservices Architecture of DeepSeek**, designed to optimize concurrency, scalability, and fault tolerance. The main goal of this design is to process a high volume of client requests simultaneously by distributing workloads across independent and parallelized service layers.

At the top of the architecture, two types of clients—**Web Client** and **Mobile Client**—interact with the system through secure **HTTP/HTTPS** requests. These requests are first routed through a global **CDN (Content Delivery Network) / Edge Cache**, which accelerates content delivery and reduces latency by caching responses near the user.

Next, all incoming traffic is processed by the **API Gateway Cluster**, the central entry point of the system. The gateway performs load balancing, routing, and authentication by forwarding requests either to the **BFF (Backend-for-Frontend)** components—optimized for each type of client—or directly to the **Auth Service** for user validation and token verification. This ensures that every client request is efficiently authenticated before being distributed to backend services.

Once authorized, the requests reach the **API Orchestrator**, which coordinates the execution of multiple backend microservices in parallel. For example, user-related requests are sent to the **User Service**, knowledge retrieval tasks to the **RAG (Retrieval-Augmented Generation) Service**, and inference requests to the **Inference Service**. These services run concurrently, independently accessing their required resources and returning results asynchronously. The orchestrator then aggregates all partial outputs to build the final response, reducing total response time through parallel execution.

At the model layer, the **Model Management & Serving** component handles model versioning, deployment, and serving. It communicates asynchronously with the **Message Broker (Kafka)**, which broadcasts events across services using a publish/subscribe mechanism. This enables event-driven parallel processing, allowing multiple microservices (e.g., User, Inference, and RAG) to react to the same event simultaneously, improving throughput and scalability.

The **Data Layer** includes multiple specialized storage systems operating in parallel. A **Sharded Database** handles transactional data, the **Vector Database** stores embeddings for semantic search and retrieval, and the **Object Storage** manages large unstructured files. These distributed data systems enable concurrent read and write operations, supporting the high throughput demands of parallel microservices.

To ensure system observability, all services continuously send logs, metrics, and traces to a centralized **Monitoring and Tracing** infrastructure. This allows developers and operators to track performance bottlenecks, identify latency issues, and optimize the parallel execution flow in real time.

Overall, this architecture emphasizes:

- **Parallel Execution:** Requests are handled by multiple services at once, reducing total processing time.
- **Scalability:** Each component can scale independently based on workload intensity.
- **Asynchronous Communication:** Event-driven processing ensures non-blocking interactions between services.
- **High Availability:** Redundant components and clusters ensure fault tolerance and resilience under heavy load.

This design enables DeepSeek to efficiently manage large-scale workloads, providing rapid and reliable responses even under high concurrency, which is essential for real-time AI inference and research-intensive operations.

8 Optimality of the Parallel Microservices Architecture

The proposed architecture for DeepSeek aims to maximize throughput and minimize latency under high-concurrency workloads. We formalize its optimality based on parallel execution principles and queueing theory.

8.1 Latency Reduction via Parallelism

Let a request be decomposed into independent tasks $T_1, T_2, \dots, T_n$. In a sequential design, total latency is:

$$L_{seq} = \sum_{i=1}^{n} T_i$$

With parallel execution, independent tasks can run concurrently, reducing latency to:

$$L_{par} = \max_i T_i + L_{agg}$$

where L_{agg} is a small aggregation overhead. Clearly,

$$L_{par} \leq L_{seq}.$$

8.2 Throughput Scaling

Let k be the number of identical service replicas. By Little's Law, for a stable system:

$$\bar{N} = X \cdot \bar{T}$$

where $\bar{N}$ is average number of requests in the system, $\bar{T}$ is mean processing time per request, and X is throughput. Increasing k increases service capacity and thus X, while keeping $\bar{T}$ bounded.

8.3 Amdahl's Law for Parallel Fraction

If p fraction of work is parallelizable and $1 - p$ is serial:

$$S_{\max} = \frac{1}{1-p}.$$

The DeepSeek architecture maximizes p by:

- Decoupling services via message broker (Kafka)
- Making microservices stateless and horizontally scalable
- Offloading heavy inference and vector search tasks asynchronously

Thus, speedup approaches $S_{\max}$ under realistic workloads.

8.4 Conclusion

The architecture achieves near-optimal latency and throughput for high-concurrency AI workloads:

- Critical path latency is minimized via parallel execution
- Throughput scales with replicas until saturation
- Asynchronous communication decouples heavy tasks and avoids blocking

Empirical validation through load testing and monitoring can confirm that the architecture meets target Service Level Objectives (SLOs) and sustains high throughput under peak load.

9 Utility of Distributed Architecture for DeepSeek

DeepSeek is designed to handle **large-scale search queries** and **machine learning inference** in real time. Using a **distributed architecture** provides several critical advantages:

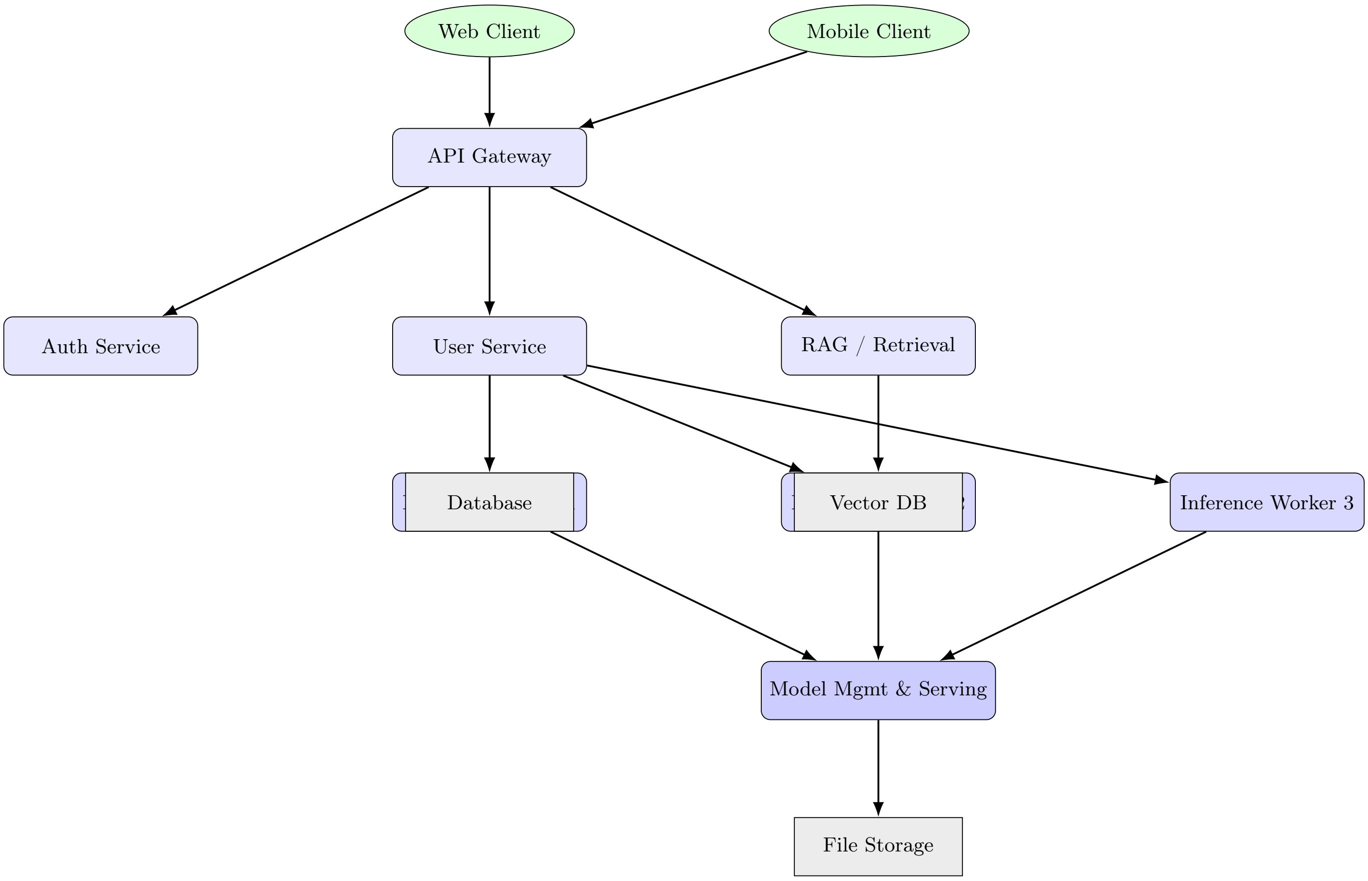
1. **Parallel Processing of Requests:** Each service (User Service, RAG Service, Inference Service, etc.) can run on **multiple nodes simultaneously**. Requests from clients are distributed across these nodes, allowing **many queries to be processed in parallel**, which drastically reduces latency.
2. **Fault Tolerance and Resilience:** In a distributed system, there is **no single point of failure**. If one node fails, other nodes continue to handle requests without disrupting service. This is crucial for DeepSeek, where high availability is required for search and ML inference.

3. **Scalability without Centralized Clock:** DeepSeek's ML models and vector databases can grow independently on separate servers. The system does **not rely on a single global clock**; nodes synchronize via **distributed protocols** instead of a central timing mechanism. This allows the platform to **scale horizontally** as more users or larger models are added.

4. **Efficient Resource Utilization:** Tasks like model inference, data retrieval, and preprocessing can be **distributed across multiple nodes** based on workload. This reduces bottlenecks and ensures that no single server becomes overloaded, improving overall performance.

In summary, a distributed architecture enables DeepSeek to handle **high-throughput, low-latency searches**, scale easily, and remain reliable, even without relying on a central system clock or a single point of control.

10 Local Parallel Architecture of DeepSeek



10.1 Local Parallel Architecture Overview

The local parallel architecture of DeepSeek allows multiple inference requests to be processed concurrently on a single machine. This design is optimized for development, testing, and small-scale deployment.

- **Clients:** Web and mobile clients send requests through the API Gateway.
- **API Gateway:** Routes requests to appropriate services and inference workers.
- **Business Services:** Include User Service, Auth Service, and RAG Service.
- **Parallel Inference Workers:** Multiple instances of the inference service handle requests simultaneously, reducing latency and increasing throughput.
- **Model Management & Serving:** Centralized management of models, shared by all inference workers.
- **Data Storage:** Local database, vector store, and file storage for efficient retrieval and persistence.

This parallel architecture maximizes CPU/GPU utilization on a single machine, simulating a distributed system's performance. It allows DeepSeek to process multiple requests quickly while maintaining simplicity for local testing.