**Multi-Provider Model Router API Specification**

**1. Overview**

The Multi-Provider Model Router is a server application built with FastAPI that routes requests to multiple large language model (LLM) providers based on tenant-specific policies, available models, modalities, and provider health, latency, and cost. It supports multi-tenant authentication, model validation, failover, logging, and metrics monitoring.

**2. Key Components**

* **FastAPI service:** API server exposing endpoints (e.g., /v1/chat/completions) for chat completions.
* **Tenant Authentication:** Validates tenants using API keys via Authorization: Bearer <api\_key> headers.
* **Model Catalog:** YAML-based catalog defining models (e.g., GPT, Claude, Gemini) and their providers, modalities, pricing, and parameters.
* **Routing Policy Engine:** Encapsulates tenant policies that specify primary providers and failover order.
* **Model Router:** Core routing logic that selects the optimal provider considering health, latency, cost, and tenant policy.
* **Provider Adapters:** Abstract connections to different providers, implementing request/response translation.
* **Logging:** Structured logging setup with console, rotating file, and JSON format handlers for production diagnostics.
* **Metrics:** Prometheus integration providing application and routing metrics for usage, latency, failures.
* **Failover:** Automatic retry failover to secondary providers if primary fails.

**3. API Endpoints**

**3.1 Chat Completions**

* **URL:** /v1/chat/completions
* **Method:** POST
* **Description:** Accepts chat completion requests, authenticates tenant, validates model, routes to provider, and returns response.
* **Headers:** Authorization: Bearer <api\_key>
* **Request Body:** JSON containing
  + model: String model name (e.g., gpt-3.5-turbo)
  + messages: List of messages in OpenAI chat format (role + content)
  + Optional parameters: temperature, max\_tokens, top\_p, etc.
* **Response:** JSON response from selected provider routed through the service.
* **Error Codes:**
  + 401 Unauthorized: Missing or invalid API key
  + 403 Forbidden: Tenant not allowed to use requested model
  + 503 Service Unavailable: No primary provider configured
  + 502 Bad Gateway: All providers failed to respond

**4. Tenant Authentication and Authorization**

* Tenants are identified and authenticated via API keys loaded from a YAML config.
* Each tenant's allowed models and providers are defined in tenant configuration.
* Authorization enforced via dependency injection in FastAPI on each request.
* Requests are rejected early if authentication or model authorization fails.

**5. Models Catalog Specification**

* Models and their providers are defined in models\_catalog.yaml.
* Each model entry includes:
  + name: Model identifier (e.g., gpt-3.5-turbo)
  + description: Human-readable description
  + categories: Model classification tags
  + providers: List of providers supporting this model
    - Each provider defines:
      * name: Provider identifier (e.g., openai)
      * base\_url: Provider API base URL
      * modalities: Supported request types (chat, embedding, image, etc.)
      * context\_length: Min and max token limits
      * prompt\_pricing: Cost per token for different modalities
      * supported\_parameters: Request params supported
* Router filters by modality and tenant policies when selecting providers.

**6. Routing Policy Engine**

* Loads tenant-specific routing policies from YAML.
* Policies include:
  + primary\_providers: Ordered list of preferred providers per tenant.
  + failover\_order: List of fallback providers in priority.
* Retrieves tenant policy or defaults to global policy.
* Provides methods to get failover providers excluding the failed one.

**7. Model Router Logic**

* Combines tenant policy, models catalog, and provider health/latency/cost metrics.
* Filters candidate providers by:
  + Tenant primary provider preferences
  + Provider support for requested model and modality
  + Provider health (stub implementation currently always healthy)
* Scores providers by weighted sum of latency and cost.
* Selects the best provider with lowest score.
* Supports failover to secondary providers on request failure.
* Abstracts provider-specific request sending via adapter interface.

**8. Provider Adapters**

* Implement communication logic for each supported provider (OpenAI, Anthropic, Google).
* Translate incoming standardized requests to provider-specific API calls.
* Support sending requests asynchronously and returning parsed responses.

**9. Logging and Observability**

* Structured logging configured with rotating files and JSON formatting.
* Logs include contextual fields (tenant\_id, model\_name, provider, latency, error).
* Middleware logs all incoming requests and outgoing responses with timings and status.
* Provides detailed traceability for routing decisions and failures.

**10. Metrics and Monitoring**

* Prometheus middleware added to FastAPI app to expose /metrics.
* Custom Prometheus metrics include:
  + Request count (model\_router\_requests\_total) by tenant, provider, and model.
  + Request latencies histogram (model\_router\_request\_latency\_seconds).
  + Failure counters (model\_router\_failures\_total).
* Metrics exported for scraping by Prometheus for monitoring and alerting.

**11. Security Considerations**

* API keys securely validated on each request.
* No API secrets exposed externally.
* Sanitize inputs and enforce strict JSON schemas recommended (optional).
* Use HTTPS for all production traffic.
* Optionally implement rate limiting and quotas (quota enforcement currently planned).

**12. Deployment and Configuration**

* Environment-driven configuration supports debug logs, reload, and log level toggles.
* Config files for tenants and models via YAML for easy modifications.
* Providers’ API keys and secrets managed securely outside of this catalog.
* Prometheus scraping enabled via /metrics endpoint for performance and availability monitoring.

**13. Future Extensions**

* Integrate embedding models and dedicated embedding endpoints.
* Extend adapter plugins for new providers and modalities.
* Enhance health checks with real-time provider status and circuit breakers.
* Add comprehensive unit and integration tests.
* **Implement a database and migrations to maintain the models, tenants without config files to get the more and ease of control.**