API & Queue Management Service

Python

```python
# api_service.py
from fastapi import FastAPI, WebSocket, Depends
from sqlalchemy.orm import Session
import asyncio
import httpx
import logging

class APIService:
    def __init__(self):
        self.app = FastAPI(title="Baker Group LLM API")
        self.inference_client = InferenceClient("http://localhost:8001")
        self.queue_manager = SQLiteQueueManager("queue.db")
        self.websocket_manager = WebSocketManager()

    @self.app.post("/api/v1/submit-request")
    async def submit_request(self, request: RequestModel,
                             user: UserContext = Depends(get_current_user)):
        """Submit request to processing queue"""
        try:
            # Validate request and check user limits
            await self._validate_request(request, user)

            # Queue request with priority
            request_id = await self.queue_manager.enqueue_request(
                user_context=user,
                request_data=request,
                priority=PRIORITY_LEVELS[user.role]
            )

            # Notify inference service of new request
            await self.inference_client.notify_new_request()

            # Send real-time update to user
            await self.websocket_manager.send_user_update(
                user.id, {"request_id": request_id, "status": "queued"}
            )

            return {"request_id": request_id, "estimated_wait": await self._estimate_wait_time(user.role)}

        except Exception as e:
            logger.error(f"Request submission failed: {e}")
            raise HTTPException(status_code=500, detail="Request processing failed")

    @self.app.get("/api/v1/request-status/{request_id}")
    async def get_request_status(self, request_id: str,
                                user: UserContext = Depends(get_current_user)):
        """Get current request status and results"""
        request_status = await self.queue_manager.get_request_status(request_id, user.id)

        if request_status["status"] == "completed":
            # Retrieve result from response storage
            result = await self.queue_manager.get_response_data(request_id)
            return {"status": "completed", "result": result}

        return request_status

    @self.app.websocket("/ws/{user_id}")
    async def websocket_endpoint(self, websocket: WebSocket, user_id: str):
        """Real-time updates for request status"""
        await self.websocket_manager.connect(websocket, user_id)
        try:
            while True:
                # Keep connection alive and handle ping/pong
                await websocket.receive_text()
        except WebSocketDisconnect:
            self.websocket_manager.disconnect(websocket, user_id)

class InferenceClient:
    """Client for communicating with inference service"""
    def __init__(self, inference_url: str):
        self.client = httpx.AsyncClient(base_url=inference_url, timeout=300.0)
```

```python

    async def notify_new_request(self):
        """Notify inference service of new queued requests"""
        try:
            await self.client.post("/internal/process-queue")
        except httpx.RequestError as e:
            logger.warning(f"Failed to notify inference service: {e}")

    async def health_check(self) -> bool:
        """Check if inference service is available"""
        try:
            response = await self.client.get("/health")
            return response.status_code == 200
        except:
            return False



```

LLM Inference Service

Python

```python
# inference_service.py
from fastapi import FastAPI
import asyncio
import sqlite3
from contextlib import asynccontextmanager
import time

class InferenceService:
    def __init__(self):
        self.app = FastAPI(title="Baker Group Inference Service")
        self.queue_manager = SQLiteQueueManager("queue.db")
        self.model_manager = ModelResourceManager()
        self.processing_loop_task = None

    @asynccontextmanager
    async def lifespan(self, app: FastAPI):
        """Manage service lifecycle"""
        # Startup
        await self.model_manager.load_models()
        self.processing_loop_task = asyncio.create_task(self._processing_loop())
        yield
        # Shutdown
        if self.processing_loop_task:
            self.processing_loop_task.cancel()
        await self.model_manager.cleanup()

    async def _processing_loop(self):
        """Main inference processing loop"""
        while True:
            try:
                # Get next highest priority request
                request = await self.queue_manager.get_next_request()

                if request:
                    await self._process_inference_request(request)
                else:
                    await asyncio.sleep(1)  # Brief pause if no requests

            except Exception as e:
                logger.error(f"Processing loop error: {e}")
                await asyncio.sleep(5)  # Error recovery delay

    async def _process_inference_request(self, request: QueuedRequest):
        """Process individual inference request"""
        try:
            # Update request status to processing
            await self.queue_manager.update_request_status(request.id, "processing")

            # Determine appropriate model and acquire resource lock
            if request.model_target == "gpt-oss-20b":
                async with self.model_manager.gpt_oss_lock:
                    result = await self._execute_text_inference(request)
            else:  # llama3.2-vision-11b
                async with self.model_manager.vision_lock:
                    result = await self._execute_vision_inference(request)

            # Store result and update status
            await self.queue_manager.store_response(request.id, result)
            await
```

```python
self.queue_manager.update_request_status(request.id, "completed")

            # Notify API service of completion (optional webhook)
            await self._notify_completion(request.id, request.user_id)

        except Exception as e:
            await self.queue_manager.update_request_status(request.id, "failed", str(e))
            logger.error(f"Inference request {request.id} failed: {e}")

    @self.app.post("/internal/process-queue")
    async def trigger_queue_processing(self):
        """Internal endpoint to trigger queue processing"""
        # This endpoint allows API service to notify of new requests
        return {"status": "processing triggered"}

    @self.app.get("/health")
    async def health_check(self):
        """Health check endpoint"""
        model_status = await self.model_manager.get_model_status()
        gpu_status = await self.model_manager.get_gpu_status()

        return {
            "status": "healthy" if model_status["loaded"] else "degraded",
            "models": model_status,
            "gpu": gpu_status,
            "queue_depth": await self.queue_manager.get_queue_depth()
        }

class ModelResourceManager:
    """Manages Ollama models and GPU resources"""
    def __init__(self):
        self.gpt_oss_lock = asyncio.Semaphore(1)
        self.vision_lock = asyncio.Semaphore(1)
        self.ollama_client = OllamaClient()

    async def load_models(self):
        """Initialize and load both models"""
        await self.ollama_client.pull_model("gpt-oss:20b")
        await self.ollama_client.pull_model("llama3.2-
vision:11b")
        logger.info("Models loaded successfully")

    async def get_model_status(self) -> dict:
        """Get current model loading status"""
        return {
            "loaded": True,  # Check actual model status
            "gpt_oss_memory": "14GB",
            "vision_memory": "12GB",
            "available_vram": "6GB"
        }



```

inter-service API

Python

```python
# Communication between API service and Inference service
class ServiceCommunication:
    def __init__(self):
        self.api_service_url = "http://localhost:8000"
        self.inference_service_url = "http://localhost:8001"

    # API Service -> Inference Service
    async def notify_new_request(self):
        """API notifies inference of new queued requests"""
        endpoint = f"{self.inference_service_url}/internal/process-queue"

    async def check_inference_health(self):
        """API checks if inference service is available"""
        endpoint = f"{self.inference_service_url}/health"

    # Inference Service -> API Service (optional webhook)
    async def notify_completion(self, request_id: str, user_id: str):
        """Inference notifies API when request completes"""
        endpoint = f"{self.api_service_url}/internal/request-completed"
        payload = {"request_id": request_id, "user_id": user_id}

```

healthcheck service

Python

```python
class ServiceHealthMonitor:
    def __init__(self):
        self.api_service = "http://localhost:8000"
        self.inference_service = "http://localhost:8001"

    async def monitor_services(self):
        """Continuous service health monitoring"""
        while True:
            api_health = await self._check_service_health(self.api_service)
            inference_health = await self._check_service_health(self.inference_service)

            if not api_health:
                await self._alert_service_down("API Service")
            if not inference_health:
                await self._alert_service_down("Inference Service")

            await asyncio.sleep(30)  # Check every 30 seconds

    async def get_system_status(self):
        """Get overall system health status"""
        return {
            "api_service": await self._detailed_health_check(self.api_service),
            "inference_service": await self._detailed_health_check(self.inference_service),
            "queue_metrics": await self._get_queue_metrics(),
            "gpu_status": await self._get_gpu_metrics()
        }



```