```
import multiprocessing
import os
import secrets
import signal
import sys
import threading
import time
import traceback
from concurrent.futures import ThreadPoolExecutor
from dataclasses import dataclass
from datetime import datetime, timedelta
from enum import Enum
from multiprocessing import Lock, Process, Queue, Value
from pathlib import Path
from typing import Any, Dict, List, Optional, Tuple
from uuid import UUID, uuid4

import httpx
import psutil
import uvicorn
from dotenv import load_dotenv
from fastapi import (
    BackgroundTasks,
    Depends,
    FastAPI,
    Header,
```

```
    HTTPException,
    Query,
    Request,
    status,
)
from fastapi.middleware.cors import CORSMiddleware
from loguru import logger
from pydantic import BaseModel, Field

from swarms.structs.agent import Agent

# Load environment variables
load_dotenv()


# # Set start method to 'fork' at the very beginning of the script
# multiprocessing.set_start_method('fork')


@dataclass
class ProcessMetrics:
    """Metrics for each API process."""

    pid: int
    cpu_usage: float
    memory_usage: float
```

```
    request_count: int
    last_heartbeat: float
    port: int


class ProcessManager:
    """Manages multiple API processes and their metrics."""

    def __init__(
        self, num_processes: int = None, start_port: int = 8000
    ):
        self.num_processes = (
            num_processes or multiprocessing.cpu_count()
        )
        self.start_port = start_port
        self.processes: Dict[int, Process] = {}
        self.metrics: Dict[int, ProcessMetrics] = {}
        self.metrics_lock = Lock()
        self.heartbeat_queue = Queue()
        self.shutdown_event = multiprocessing.Event()

    def start_api_process(self, port: int) -> Process:
        """Start a single API process on the specified port."""
        process = Process(
            target=run_api_instance,
            args=(port, self.heartbeat_queue, self.shutdown_event),
```

```
    )
    process.start()
    return process

def start_all_processes(self):
    """Start all API processes."""
    for i in range(self.num_processes):
        port = self.start_port + i + 1
        process = self.start_api_process(port)
        self.processes[process.pid] = process
        self.metrics[process.pid] = ProcessMetrics(
            pid=process.pid,
            cpu_usage=0.0,
            memory_usage=0.0,
            request_count=0,
            last_heartbeat=time.time(),
            port=port,
        )

def monitor_processes(self):
    """Monitor process health and metrics."""
    while not self.shutdown_event.is_set():
        try:
            # Update metrics from heartbeat queue
            while not self.heartbeat_queue.empty():
                pid, cpu, memory, requests = (
```

```
                self.heartbeat_queue.get_nowait()
            )
            with self.metrics_lock:
                if pid in self.metrics:
                    self.metrics[pid].cpu_usage = cpu
                    self.metrics[pid].memory_usage = memory
                    self.metrics[pid].request_count = requests
                    self.metrics[pid].last_heartbeat = (
                        time.time()
                    )

        # Check for dead processes and restart them
        current_time = time.time()
        with self.metrics_lock:
            for pid, metrics in list(self.metrics.items()):
                if (
                    current_time - metrics.last_heartbeat > 30
                ):  # 30 seconds timeout
                    print(
                        f"Process {pid} appears to be dead, restarting..."
                    )
                    if pid in self.processes:
                        self.processes[pid].terminate()
                        del self.processes[pid]
                    new_process = self.start_api_process(
                        metrics.port
```

```
                    )
                    self.processes[new_process.pid] = (
                        new_process
                    )
                    self.metrics[new_process.pid] = (
                        ProcessMetrics(
                            pid=new_process.pid,
                            cpu_usage=0.0,
                            memory_usage=0.0,
                            request_count=0,
                            last_heartbeat=time.time(),
                            port=metrics.port,
                        )
                    )
                    del self.metrics[pid]

            time.sleep(1)
        except Exception as e:
            print(f"Error in process monitoring: {e}")

def shutdown(self):
    """Shutdown all processes gracefully."""
    self.shutdown_event.set()
    for process in self.processes.values():
        process.terminate()
        process.join()
```

```
class AgentStatus(str, Enum):
    """Enum for agent status."""

    IDLE = "idle"
    PROCESSING = "processing"
    ERROR = "error"
    MAINTENANCE = "maintenance"


# Security configurations
API_KEY_LENGTH = 32  # Length of generated API keys


class APIKey(BaseModel):
    key: str
    name: str
    created_at: datetime
    last_used: datetime
    is_active: bool = True


class APIKeyCreate(BaseModel):
    name: str  # A friendly name for the API key
```

```
class User(BaseModel):
    id: UUID
    username: str
    is_active: bool = True
    is_admin: bool = False
    api_keys: Dict[str, APIKey] = {}  # key -> APIKey object



class AgentConfig(BaseModel):
    """Configuration model for creating a new agent."""

    agent_name: str = Field(..., description="Name of the agent")
    model_name: str = Field(
        ...,
        description="Name of the llm you want to use provided by litellm",
    )
    description: str = Field(
        default="", description="Description of the agent's purpose"
    )
    system_prompt: str = Field(
        ..., description="System prompt for the agent"
    )
    model_name: str = Field(
        default="gpt-4", description="Model name to use"
    )
```

```
    temperature: float = Field(
        default=0.1,
        ge=0.0,
        le=2.0,
        description="Temperature for the model",
    )
    max_loops: int = Field(
        default=1, ge=1, description="Maximum number of loops"
    )
    autosave: bool = Field(
        default=True, description="Enable autosave"
    )
    dashboard: bool = Field(
        default=False, description="Enable dashboard"
    )
    verbose: bool = Field(
        default=True, description="Enable verbose output"
    )
    dynamic_temperature_enabled: bool = Field(
        default=True, description="Enable dynamic temperature"
    )
    user_name: str = Field(
        default="default_user", description="Username for the agent"
    )
    retry_attempts: int = Field(
        default=1, ge=1, description="Number of retry attempts"
```

```
    )
    context_length: int = Field(
        default=200000, ge=1000, description="Context length"
    )
    output_type: str = Field(
        default="string", description="Output type (string or json)"
    )
    streaming_on: bool = Field(
        default=False, description="Enable streaming"
    )
    tags: List[str] = Field(
        default_factory=list,
        description="Tags for categorizing the agent",
    )


class AgentUpdate(BaseModel):
    """Model for updating agent configuration."""

    description: Optional[str] = None
    system_prompt: Optional[str] = None
    temperature: Optional[float] = 0.5
    max_loops: Optional[int] = 1
    tags: Optional[List[str]] = None
    status: Optional[AgentStatus] = None
```

```
class AgentSummary(BaseModel):
    """Summary model for agent listing."""

    agent_id: UUID
    agent_name: str
    description: str
    created_at: datetime
    last_used: datetime
    total_completions: int
    tags: List[str]
    status: AgentStatus


class AgentMetrics(BaseModel):
    """Model for agent performance metrics."""

    total_completions: int
    average_response_time: float
    error_rate: float
    last_24h_completions: int
    total_tokens_used: int
    uptime_percentage: float
    success_rate: float
    peak_tokens_per_minute: int
```

```
class CompletionRequest(BaseModel):
    """Model for completion requests."""

    prompt: str = Field(..., description="The prompt to process")
    agent_id: UUID = Field(..., description="ID of the agent to use")
    max_tokens: Optional[int] = Field(
        None, description="Maximum tokens to generate"
    )
    temperature_override: Optional[float] = 0.5
    stream: bool = Field(
        default=False, description="Enable streaming response"
    )


class CompletionResponse(BaseModel):
    """Model for completion responses."""

    agent_id: UUID
    response: str
    metadata: Dict[str, Any]
    timestamp: datetime
    processing_time: float
    token_usage: Dict[str, int]
```

```
class AgentStore:
    """Enhanced store for managing agents."""

    def __init__(self):
        self.agents: Dict[UUID, Agent] = {}
        self.agent_metadata: Dict[UUID, Dict[str, Any]] = {}
        self.users: Dict[UUID, User] = {}  # user_id -> User
        self.api_keys: Dict[str, UUID] = {}  # api_key -> user_id
        self.user_agents: Dict[UUID, List[UUID]] = (
            {}
        )  # user_id -> [agent_ids]
        self.executor = ThreadPoolExecutor(max_workers=4)
        self.total_requests = Value(
            "i", 0
        )  # Shared counter for total requests
        self._ensure_directories()

    def increment_request_count(self):
        """Increment the total request counter."""
        with self.total_requests.get_lock():
            self.total_requests.value += 1

    def get_total_requests(self) -> int:
        """Get the total number of requests processed."""
        return self.total_requests.value
```

```
def _ensure_directories(self):
    """Ensure required directories exist."""
    Path("logs").mkdir(exist_ok=True)
    Path("states").mkdir(exist_ok=True)


def create_api_key(self, user_id: UUID, key_name: str) -> APIKey:
    """Create a new API key for a user."""
    if user_id not in self.users:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail="User not found",
        )

    # Generate a secure random API key
    api_key = secrets.token_urlsafe(API_KEY_LENGTH)

    # Create the API key object
    key_object = APIKey(
        key=api_key,
        name=key_name,
        created_at=datetime.utcnow(),
        last_used=datetime.utcnow(),
    )

    # Store the API key
    self.users[user_id].api_keys[api_key] = key_object
```

```
    self.api_keys[api_key] = user_id

    return key_object

async def verify_agent_access(
    self, agent_id: UUID, user_id: UUID
) -> bool:
    """Verify if a user has access to an agent."""
    if agent_id not in self.agents:
        return False
    return (
        self.agent_metadata[agent_id]["owner_id"] == user_id
        or self.users[user_id].is_admin
    )

def validate_api_key(self, api_key: str) -> Optional[UUID]:
    """Validate an API key and return the associated user ID."""
    user_id = self.api_keys.get(api_key)
    if not user_id or api_key not in self.users[user_id].api_keys:
        return None

    key_object = self.users[user_id].api_keys[api_key]
    if not key_object.is_active:
        return None

    # Update last used timestamp
```

```
        key_object.last_used = datetime.utcnow()
        return user_id

    async def create_agent(
        self, config: AgentConfig, user_id: UUID
    ) -> UUID:
        """Create a new agent with the given configuration."""
        try:

            agent = Agent(
                agent_name=config.agent_name,
                system_prompt=config.system_prompt,
                model_name=config.model_name,
                max_loops=config.max_loops,
                autosave=config.autosave,
                dashboard=config.dashboard,
                verbose=config.verbose,
                dynamic_temperature_enabled=True,

saved_state_path=f"states/{config.agent_name}_{datetime.now().strftime('%Y%m%d_%H%M%S')}.j
son",
                user_name=config.user_name,
                retry_attempts=config.retry_attempts,
                context_length=config.context_length,
                return_step_meta=True,
                output_type="str",
```

```
    streaming_on=config.streaming_on,
)

agent_id = uuid4()
self.agents[agent_id] = agent
self.agent_metadata[agent_id] = {
    "description": config.description,
    "created_at": datetime.utcnow(),
    "last_used": datetime.utcnow(),
    "total_completions": 0,
    "tags": config.tags,
    "total_tokens": 0,
    "error_count": 0,
    "response_times": [],
    "status": AgentStatus.IDLE,
    "start_time": datetime.utcnow(),
    "downtime": timedelta(),
    "successful_completions": 0,
}

# Add to user's agents list
if user_id not in self.user_agents:
    self.user_agents[user_id] = []
self.user_agents[user_id].append(agent_id)

return agent_id
```

```
    except Exception as e:
        logger.error(f"Error creating agent: {str(e)}")
        raise HTTPException(
            status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
            detail=f"Failed to create agent: {str(e)}",
        )

async def get_agent(self, agent_id: UUID) -> Agent:
    """Retrieve an agent by ID."""
    agent = self.agents.get(agent_id)
    if not agent:
        logger.error(f"Agent not found: {agent_id}")
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail=f"Agent {agent_id} not found",
        )
    return agent

async def update_agent(
    self, agent_id: UUID, update: AgentUpdate
) -> None:
    """Update agent configuration."""
    agent = await self.get_agent(agent_id)
    metadata = self.agent_metadata[agent_id]
```

```
    if update.system_prompt:
        agent.system_prompt = update.system_prompt
    if update.max_loops is not None:
        agent.max_loops = update.max_loops
    if update.tags is not None:
        metadata["tags"] = update.tags
    if update.description is not None:
        metadata["description"] = update.description
    if update.status is not None:
        metadata["status"] = update.status
        if update.status == AgentStatus.MAINTENANCE:
            metadata["downtime"] += (
                datetime.utcnow() - metadata["last_used"]
            )

    logger.info(f"Updated agent {agent_id}")

async def list_agents(
    self,
    tags: Optional[List[str]] = None,
    status: Optional[AgentStatus] = None,
) -> List[AgentSummary]:
    """List all agents, optionally filtered by tags and status."""
    summaries = []
    for agent_id, agent in self.agents.items():
        metadata = self.agent_metadata[agent_id]
```

```
        # Apply filters
        if tags and not any(
            tag in metadata["tags"] for tag in tags
        ):
            continue
        if status and metadata["status"] != status:
            continue

        summaries.append(
            AgentSummary(
                agent_id=agent_id,
                agent_name=agent.agent_name,
                description=metadata["description"],
                created_at=metadata["created_at"],
                last_used=metadata["last_used"],
                total_completions=metadata["total_completions"],
                tags=metadata["tags"],
                status=metadata["status"],
            )
        )
    return summaries

async def get_agent_metrics(self, agent_id: UUID) -> AgentMetrics:
    """Get performance metrics for an agent."""
    metadata = self.agent_metadata[agent_id]
```

```
response_times = metadata["response_times"]

# Calculate metrics
total_time = datetime.utcnow() - metadata["start_time"]
uptime = total_time - metadata["downtime"]
uptime_percentage = (
    uptime.total_seconds() / total_time.total_seconds()
) * 100

success_rate = (
    metadata["successful_completions"]
    / metadata["total_completions"]
    * 100
    if metadata["total_completions"] > 0
    else 0
)

return AgentMetrics(
    total_completions=metadata["total_completions"],
    average_response_time=(
        sum(response_times) / len(response_times)
        if response_times
        else 0
    ),
    error_rate=(
        metadata["error_count"]
```

```
            / metadata["total_completions"]
            if metadata["total_completions"] > 0
            else 0
        ),
        last_24h_completions=sum(
            1
            for t in response_times
            if (datetime.utcnow() - t).days < 1
        ),
        total_tokens_used=metadata["total_tokens"],
        uptime_percentage=uptime_percentage,
        success_rate=success_rate,
        peak_tokens_per_minute=max(
            metadata.get("tokens_per_minute", [0])
        ),
    )

async def clone_agent(
    self, agent_id: UUID, new_name: str
) -> UUID:
    """Clone an existing agent with a new name."""
    original_agent = await self.get_agent(agent_id)
    original_metadata = self.agent_metadata[agent_id]

    config = AgentConfig(
        agent_name=new_name,
```

```
        description=f"Clone of {original_agent.agent_name}",
        system_prompt=original_agent.system_prompt,
        model_name=original_agent.model_name,
        temperature=0.5,
        max_loops=original_agent.max_loops,
        tags=original_metadata["tags"],
    )

    return await self.create_agent(config)

async def delete_agent(self, agent_id: UUID) -> None:
    """Delete an agent."""
    if agent_id not in self.agents:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail=f"Agent {agent_id} not found",
        )

    # Clean up any resources
    agent = self.agents[agent_id]
    if agent.autosave and os.path.exists(agent.saved_state_path):
        os.remove(agent.saved_state_path)

    del self.agents[agent_id]
    del self.agent_metadata[agent_id]
    logger.info(f"Deleted agent {agent_id}")
```

```
async def process_completion(
    self,
    agent: Agent,
    prompt: str,
    agent_id: UUID,
    max_tokens: Optional[int] = None,
    temperature_override: Optional[float] = None,
) -> CompletionResponse:
    """Process a completion request using the specified agent."""
    start_time = datetime.utcnow()
    metadata = self.agent_metadata[agent_id]

    try:
        # Update agent status
        metadata["status"] = AgentStatus.PROCESSING
        metadata["last_used"] = start_time

        # Process the completion
        response = agent.run(prompt)

        # Update metrics
        processing_time = (
            datetime.utcnow() - start_time
        ).total_seconds()
        metadata["response_times"].append(processing_time)
```

```
metadata["total_completions"] += 1
metadata["successful_completions"] += 1

# Estimate token usage (this is a rough estimate)
prompt_tokens = len(prompt.split()) * 1.3
completion_tokens = len(response.split()) * 1.3
total_tokens = int(prompt_tokens + completion_tokens)
metadata["total_tokens"] += total_tokens

# Update tokens per minute tracking
current_minute = datetime.utcnow().replace(
    second=0, microsecond=0
)
if "tokens_per_minute" not in metadata:
    metadata["tokens_per_minute"] = {}
metadata["tokens_per_minute"][current_minute] = (
    metadata["tokens_per_minute"].get(current_minute, 0)
    + total_tokens
)

return CompletionResponse(
    agent_id=agent_id,
    response=response,
    metadata={
        "agent_name": agent.agent_name,
        # "model_name": agent.llm.model_name,
```

```
                # "temperature": 0.5,
            },
            timestamp=datetime.utcnow(),
            processing_time=processing_time,
            token_usage={
                "prompt_tokens": int(prompt_tokens),
                "completion_tokens": int(completion_tokens),
                "total_tokens": total_tokens,
            },
        )

    except Exception as e:
        metadata["error_count"] += 1
        metadata["status"] = AgentStatus.ERROR
        logger.error(
            f"Error in completion processing: {str(e)}\n{traceback.format_exc()}"
        )
        raise HTTPException(
            status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
            detail=f"Error processing completion: {str(e)}",
        )
    finally:
        metadata["status"] = AgentStatus.IDLE


class StoreManager:
```

```
    _instance = None

    @classmethod
    def get_instance(cls) -> "AgentStore":
        if cls._instance is None:
            cls._instance = AgentStore()
        return cls._instance


# Modify the dependency function
def get_store() -> AgentStore:
    """Dependency to get the AgentStore instance."""
    return StoreManager.get_instance()


# Security utility function using the new dependency
async def get_current_user(
    api_key: str = Header(
        ..., description="API key for authentication"
    ),
    store: AgentStore = Depends(get_store),
) -> User:
    """Validate API key and return current user."""
    user_id = store.validate_api_key(api_key)
    if not user_id:
        raise HTTPException(
```

```
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Invalid or expired API key",
            headers={"WWW-Authenticate": "ApiKey"},
        )
    return store.users[user_id]


class SwarmsAPI:
    """Enhanced API class for Swarms agent integration."""

    def __init__(self):
        self.app = FastAPI(
            title="Swarms Agent API",
            description="Production-grade API for Swarms agent interaction",
            version="1.0.0",
            docs_url="/v1/docs",
            redoc_url="/v1/redoc",
        )
        # Initialize the store using the singleton manager
        self.store = StoreManager.get_instance()

        # Configure CORS
        self.app.add_middleware(
            CORSMiddleware,
            allow_origins=[
                "*"
```

```
        ],  # Configure appropriately for production
        allow_credentials=True,
        allow_methods=["*"],
        allow_headers=["*"],
    )

    self._setup_routes()

def _setup_routes(self):
    """Set up API routes."""

    # In your API code
    @self.app.post("/v1/users", response_model=Dict[str, Any])
    async def create_user(request: Request):
        """Create a new user and initial API key."""
        try:
            body = await request.json()
            username = body.get("username")
            if not username or len(username) < 3:
                raise HTTPException(
                    status_code=400, detail="Invalid username"
                )

            user_id = uuid4()
            user = User(id=user_id, username=username)
            self.store.users[user_id] = user
```

```
        initial_key = self.store.create_api_key(
            user_id, "Initial Key"
        )
        return {
            "user_id": user_id,
            "api_key": initial_key.key,
        }
    except Exception as e:
        logger.error(f"Error creating user: {str(e)}")
        raise HTTPException(status_code=400, detail=str(e))


@self.app.post(
    "/v1/users/{user_id}/api-keys", response_model=APIKey
)
async def create_api_key(
    user_id: UUID,
    key_create: APIKeyCreate,
    current_user: User = Depends(get_current_user),
):
    """Create a new API key for a user."""
    if (
        current_user.id != user_id
        and not current_user.is_admin
    ):
        raise HTTPException(
            status_code=status.HTTP_403_FORBIDDEN,
```

```
            detail="Not authorized to create API keys for this user",
        )

    return self.store.create_api_key(user_id, key_create.name)

@self.app.get(
    "/v1/users/{user_id}/api-keys",
    response_model=List[APIKey],
)
async def list_api_keys(
    user_id: UUID,
    current_user: User = Depends(get_current_user),
):
    """List all API keys for a user."""
    if (
        current_user.id != user_id
        and not current_user.is_admin
    ):
        raise HTTPException(
            status_code=status.HTTP_403_FORBIDDEN,
            detail="Not authorized to view API keys for this user",
        )

    return list(self.store.users[user_id].api_keys.values())

@self.app.delete("/v1/users/{user_id}/api-keys/{key}")
```

```
async def revoke_api_key(
    user_id: UUID,
    key: str,
    current_user: User = Depends(get_current_user),
):
    """Revoke an API key."""
    if (
        current_user.id != user_id
        and not current_user.is_admin
    ):
        raise HTTPException(
            status_code=status.HTTP_403_FORBIDDEN,
            detail="Not authorized to revoke API keys for this user",
        )

    if key in self.store.users[user_id].api_keys:
        self.store.users[user_id].api_keys[
            key
        ].is_active = False
        del self.store.api_keys[key]
        return {"status": "API key revoked"}

    raise HTTPException(
        status_code=status.HTTP_404_NOT_FOUND,
        detail="API key not found",
    )
```

```
@self.app.get(
    "/v1/users/me/agents", response_model=List[AgentSummary]
)
async def list_user_agents(
    current_user: User = Depends(get_current_user),
    tags: Optional[List[str]] = Query(None),
    status: Optional[AgentStatus] = None,
):
    """List all agents owned by the current user."""
    user_agents = self.store.user_agents.get(
        current_user.id, []
    )
    return [
        agent
        for agent in await self.store.list_agents(
            tags, status
        )
        if agent.agent_id in user_agents
    ]


@self.app.middleware("http")
async def count_requests(request: Request, call_next):
    """Middleware to count all incoming requests."""
    self.store.increment_request_count()
    response = await call_next(request)
```

```
    return response

# Modify existing routes to use API key authentication
@self.app.post("/v1/agent", response_model=Dict[str, UUID])
async def create_agent(
    config: AgentConfig,
    current_user: User = Depends(get_current_user),
):
    """Create a new agent with the specified configuration."""
    agent_id = await self.store.create_agent(
        config, current_user.id
    )
    return {"agent_id": agent_id}

@self.app.get("/v1/agents", response_model=List[AgentSummary])
async def list_agents(
    tags: Optional[List[str]] = Query(None),
    status: Optional[AgentStatus] = None,
):
    """List all agents, optionally filtered by tags and status."""
    return await self.store.list_agents(tags, status)

@self.app.patch(
    "/v1/agent/{agent_id}", response_model=Dict[str, str]
)
async def update_agent(agent_id: UUID, update: AgentUpdate):
```

```
    """Update an existing agent's configuration."""
    await self.store.update_agent(agent_id, update)
    return {"status": "updated"}

@self.app.get(
    "/v1/agent/{agent_id}/metrics",
    response_model=AgentMetrics,
)
async def get_agent_metrics(agent_id: UUID):
    """Get performance metrics for a specific agent."""
    return await self.store.get_agent_metrics(agent_id)

@self.app.post(
    "/v1/agent/{agent_id}/clone",
    response_model=Dict[str, UUID],
)
async def clone_agent(agent_id: UUID, new_name: str):
    """Clone an existing agent with a new name."""
    new_id = await self.store.clone_agent(agent_id, new_name)
    return {"agent_id": new_id}

@self.app.delete("/v1/agent/{agent_id}")
async def delete_agent(agent_id: UUID):
    """Delete an agent."""
    await self.store.delete_agent(agent_id)
    return {"status": "deleted"}
```

```
@self.app.post(
    "/v1/agent/completions", response_model=CompletionResponse
)
async def create_completion(
    request: CompletionRequest,
    background_tasks: BackgroundTasks,
):
    """Process a completion request with the specified agent."""
    try:
        agent = await self.store.get_agent(request.agent_id)

        # Process completion
        response = await self.store.process_completion(
            agent,
            request.prompt,
            request.agent_id,
            request.max_tokens,
            0.5,
        )

        # Schedule background cleanup
        background_tasks.add_task(
            self._cleanup_old_metrics, request.agent_id
        )
```

```
        return response

    except Exception as e:
        logger.error(f"Error processing completion: {str(e)}")
        raise HTTPException(
            status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
            detail=f"Error processing completion: {str(e)}",
        )

@self.app.get("/v1/agent/{agent_id}/status")
async def get_agent_status(agent_id: UUID):
    """Get the current status of an agent."""
    metadata = self.store.agent_metadata.get(agent_id)
    if not metadata:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail=f"Agent {agent_id} not found",
        )
    return {
        "agent_id": agent_id,
        "status": metadata["status"],
        "last_used": metadata["last_used"],
        "total_completions": metadata["total_completions"],
        "error_count": metadata["error_count"],
    }
```

```
    async def _cleanup_old_metrics(self, agent_id: UUID):
        """Clean up old metrics data to prevent memory bloat."""
        metadata = self.store.agent_metadata.get(agent_id)
        if metadata:
            # Keep only last 24 hours of response times
            cutoff = datetime.utcnow() - timedelta(days=1)
            metadata["response_times"] = [
                t
                for t in metadata["response_times"]
                if isinstance(t, (int, float))
                and t > cutoff.timestamp()
            ]

            # Clean up old tokens per minute data
            if "tokens_per_minute" in metadata:
                metadata["tokens_per_minute"] = {
                    k: v
                    for k, v in metadata["tokens_per_minute"].items()
                    if k > cutoff
                }


def run_api_instance(
    port: int, heartbeat_queue: Queue, shutdown_event: any
):
    """Run a single API instance and report metrics."""
```

```
try:
    # Initialize API
    api = SwarmsAPI()
    process = psutil.Process()

    # Start metrics reporting
    def report_metrics():
        while not shutdown_event.is_set():
            try:
                cpu_percent = process.cpu_percent()
                memory_percent = process.memory_percent()
                heartbeat_queue.put(
                    (
                        process.pid,
                        cpu_percent,
                        memory_percent,
                        api.store.get_total_requests(),
                    )
                )
                time.sleep(5)
            except Exception as e:
                logger.error(f"Error reporting metrics: {e}")

    metrics_thread = threading.Thread(target=report_metrics)
    metrics_thread.daemon = True
    metrics_thread.start()
```

```
        # Run API
        uvicorn.run(
            api.app, host="0.0.0.0", port=port, log_level="info"
        )

    except Exception as e:
        logger.error(f"Error in API instance: {e}")
        sys.exit(1)


class MultiProcessManager:
    """Manages multiple API processes."""

    def __init__(
        self, base_port: int = 8000, num_processes: int = None
    ):
        self.base_port = base_port
        self.num_processes = (
            num_processes or multiprocessing.cpu_count()
        )
        self.processes: Dict[int, Process] = {}
        self.metrics: Dict[int, ProcessMetrics] = {}
        self.active = Value("b", True)

    def start_process(self, port: int) -> Process:
```

```
    """Start a single API process."""
    process = Process(target=run_api_instance, args=(port,))
    process.start()
    self.metrics[process.pid] = ProcessMetrics(process.pid, port)
    self.processes[process.pid] = process
    return process

def monitor_processes(self):
    """Monitor process health and metrics."""
    while self.active.value:
        for pid, metrics in list(self.metrics.items()):
            try:
                # Update process metrics
                process = psutil.Process(pid)
                metrics.cpu_usage = process.cpu_percent()
                metrics.memory_usage = process.memory_percent()
                metrics.last_heartbeat = time.time()
            except psutil.NoSuchProcess:
                # Restart dead process
                logger.warning(
                    f"Process {pid} died, restarting..."
                )
                if pid in self.processes:
                    self.processes[pid].terminate()
                    del self.processes[pid]
                self.start_process(metrics.port)
```

```
                del self.metrics[pid]
        time.sleep(5)

def start(self):
    """Start all API processes."""
    logger.info(f"Starting {self.num_processes} API processes...")

    # Start worker processes
    for i in range(self.num_processes):
        port = self.base_port + i + 1
        self.start_process(port)

    # Start monitoring thread
    monitor_thread = threading.Thread(
        target=self.monitor_processes
    )
    monitor_thread.daemon = True
    monitor_thread.start()

    logger.info("All processes started successfully")

def shutdown(self):
    """Shutdown all processes."""
    self.active.value = False
    for process in self.processes.values():
        process.terminate()
```

```
            process.join()


def create_app() -> FastAPI:
    """Create and configure the FastAPI application."""
    logger.info("Creating FastAPI application")
    api = SwarmsAPI()
    app = api.app
    logger.info("FastAPI application created successfully")
    return app


class LoadBalancer:
    """Load balancer for distributing requests across API instances."""

    def __init__(self, process_manager: ProcessManager):
        self.process_manager = process_manager
        self.last_selected_pid = None
        self._lock = Lock()

    def get_best_instance(self) -> Tuple[int, int]:
        """Select the best instance to handle the next request based on load."""
        with self.process_manager.metrics_lock:
            valid_instances = [
                (pid, metrics)
                for pid, metrics in self.process_manager.metrics.items()
```

```
    if time.time() - metrics.last_heartbeat < 30
]

if not valid_instances:
    raise RuntimeError(
        "No healthy API instances available"
    )

# Calculate load score for each instance
scores = []
for pid, metrics in valid_instances:
    cpu_score = metrics.cpu_usage / 100.0
    memory_score = metrics.memory_usage / 100.0
    request_score = (
        metrics.request_count / 1000.0
    )  # Normalize request count
    total_score = (
        cpu_score + memory_score + request_score
    ) / 3
    scores.append((pid, metrics.port, total_score))

# Select instance with lowest load score
selected_pid, selected_port, _ = min(
    scores, key=lambda x: x[2]
)
return selected_pid, selected_port
```

```
class LoadBalancedAPI(SwarmsAPI):
    """Enhanced API class with load balancing capabilities."""

    def __init__(
        self,
        process_manager: ProcessManager,
        load_balancer: LoadBalancer,
    ):
        super().__init__()
        self.process_manager = process_manager
        self.load_balancer = load_balancer
        self.request_count = Value("i", 0)
        self.add_middleware()

    def add_middleware(self):
        """Add middleware for request routing and metrics collection."""

        @self.app.middleware("http")
        async def route_request(request: Request, call_next):
            try:
                # Increment request count
                with self.request_count.get_lock():
                    self.request_count.value += 1
```

```
    # Get best instance for processing
    pid, port = self.load_balancer.get_best_instance()

    # Forward request if not already on the best instance
    if request.url.port != port:
        async with httpx.AsyncClient() as client:
            forwarded_url = f"http://localhost:{port}{request.url.path}"
            response = await client.request(
                request.method,
                forwarded_url,
                headers=dict(request.headers),
                content=await request.body(),
            )
            return httpx.Response(
                content=response.content,
                status_code=response.status_code,
                headers=dict(response.headers),
            )

    # Process request locally if already on the best instance
    response = await call_next(request)
    return response

except Exception as e:
    logger.error(f"Error routing request: {e}")
    raise HTTPException(
```

```
                status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
                detail=str(e),
            )


def run_worker(port: int):
    """Run a single worker instance."""
    try:
        api = SwarmsAPI()
        uvicorn.run(
            api.app, host="0.0.0.0", port=port, log_level="info"
        )
        logger.info(f"Worker started on port {port}")
    except Exception as e:
        logger.error(f"Worker error: {e}")


def main():
    """Main entry point for the multi-process API."""
    # Initialize processes list before any potential exceptions
    processes = []

    try:
        # Try to get current method, only set if not already set
        try:
            current_method = multiprocessing.get_start_method()
```

```
    logger.info(
        f"Using existing start method: {current_method}"
    )
except RuntimeError:
    try:
        multiprocessing.set_start_method("fork")
        logger.info("Set start method to fork")
    except RuntimeError:
        logger.warning("Using default start method")

# Calculate number of workers
num_workers = max(1, multiprocessing.cpu_count() - 1)
base_port = 8000

# Start worker processes
for i in range(num_workers):
    port = base_port + i + 1
    process = Process(target=run_worker, args=(port,))
    process.start()
    processes.append(process)
    logger.info(f"Started worker on port {port}")

# Run main instance
api = SwarmsAPI()

def shutdown_handler(signum, frame):
```

```
        logger.info("Shutting down workers...")
        for p in processes:
            try:
                p.terminate()
                p.join(timeout=5)
                logger.info(f"Worker {p.pid} terminated")
            except Exception as e:
                logger.error(f"Error shutting down worker: {e}")
        sys.exit(0)

    signal.signal(signal.SIGINT, shutdown_handler)
    signal.signal(signal.SIGTERM, shutdown_handler)

    # Run main instance
    uvicorn.run(
        api.app, host="0.0.0.0", port=base_port, log_level="info"
    )
    logger.info(f"Main instance started on port {base_port}")

except Exception as e:
    logger.error(f"Startup error: {e}")
    # Clean up any started processes
    for p in processes:
        try:
            p.terminate()
            p.join(timeout=5)
```

```
                logger.info(
                    f"Worker {p.pid} terminated during cleanup"
                )
            except Exception as cleanup_error:
                logger.error(f"Error during cleanup: {cleanup_error}")
        sys.exit(1)


if __name__ == "__main__":
    main()
```