MULTI-LLM ROUTER IMPLEMENTATION GUIDE

Build Autonomous, Cost-Efficient Agent Infrastructure

Goal: Replace single-API dependency with intelligent multi-provider routing that minimizes costs while maximizing capability.

Timeline: 2-3 days

Result: 80%+ cost reduction, faster responses, greater autonomy

---

ARCHITECTURE OVERVIEW

PHASE 1: UNIFIED LLM CLIENT

Step 1.1: Create Provider Abstraction

File: backend/app/llm/providers/base.py

```python
```

```python
from abc import ABC, abstractmethod
from typing import Dict, List, Optional
from dataclasses import dataclass

@dataclass
class LLMResponse:
    """Standardized LLM response"""
    text: str
    model: str
    provider: str
    tokens_used: int
    latency_ms: int
    cost_usd: float
    cached: bool = False

@dataclass
class LLMMessage:
    """Standardized message format"""
    role: str  # 'user', 'assistant', 'system'
    content: str

class BaseLLMProvider(ABC):
    """Base class for all LLM providers"""

    def __init__(self, api_key: Optional[str] = None):
        self.api_key = api_key
        self.provider_name = self.__class__.__name__.replace('Provider', '').lower()

    @abstractmethod
    async def complete(
        self,
        messages: List[LLMMessage],
        model: str,
        max_tokens: int = 1000,
        temperature: float = 0.7
    ) -> LLMResponse:
        """Generate completion from messages"""
        pass

    @abstractmethod
    def get_available_models(self) -> List[str]:
        """Return list of available models"""
        pass
```

```python
    @abstractmethod
    def estimate_cost(self, model: str, input_tokens: int, output_tokens: int) -> float:
        """Estimate cost in USD"""
        pass

    @abstractmethod
    async def health_check(self) -> bool:
        """Check if provider is available"""
        pass
```

Step 1.2: Anthropic Provider (Already Have)

File: backend/app/llm/providers/anthropic_provider.py

```python
```

```python
import anthropic
import os
import time
from typing import List
from .base import BaseLLMProvider, LLMResponse, LLMMessage

class AnthropicProvider(BaseLLMProvider):
    """Anthropic Claude provider"""

    MODELS = {
        "claude-opus-4-5": "claude-opus-4-5-20251101",
        "claude-sonnet-4-5": "claude-sonnet-4-5-20250929",
        "claude-haiku-4-5": "claude-haiku-4-5-20251001",
        "claude-3-5-sonnet": "claude-3-5-sonnet-20241022",
    }

    # Pricing per 1M tokens (input, output)
    PRICING = {
        "claude-opus-4-5": (15.00, 75.00),
        "claude-sonnet-4-5": (3.00, 15.00),
        "claude-haiku-4-5": (1.00, 5.00),
        "claude-3-5-sonnet": (3.00, 15.00),
    }

    def __init__(self):
        super().__init__(api_key=os.getenv("ANTHROPIC_API_KEY"))
        self.client = anthropic.Anthropic(api_key=self.api_key)

    async def complete(
        self,
        messages: List[LLMMessage],
        model: str = "claude-sonnet-4-5",
        max_tokens: int = 1000,
        temperature: float = 0.7
    ) -> LLMResponse:

        start_time = time.time()

        # Convert to Anthropic format
        anthropic_messages = [
            {"role": msg.role, "content": msg.content}
            for msg in messages
            if msg.role != "system"
```

```
        ]

        # Extract system message
        system = next(
            (msg.content for msg in messages if msg.role == "system"),
            None
        )

        # Get full model name
        full_model = self.MODELS.get(model, model)

        # Call API
        response = self.client.messages.create(
            model=full_model,
            max_tokens=max_tokens,
            temperature=temperature,
            system=system,
            messages=anthropic_messages
        )

        latency_ms = int((time.time() - start_time) * 1000)

        # Extract text
        text = response.content[0].text

        # Calculate cost
        input_tokens = response.usage.input_tokens
        output_tokens = response.usage.output_tokens
        cost = self.estimate_cost(model, input_tokens, output_tokens)

        return LLMResponse(
            text=text,
            model=full_model,
            provider="anthropic",
            tokens_used=input_tokens + output_tokens,
            latency_ms=latency_ms,
            cost_usd=cost
        )

    def get_available_models(self) -> List[str]:
        return list(self.MODELS.keys())

    def estimate_cost(self, model: str, input_tokens: int, output_tokens: int) -> float:
        if model not in self.PRICING:
```

```
        return 0.0

    input_cost, output_cost = self.PRICING[model]

    return (
        (input_tokens / 1_000_000) * input_cost +
        (output_tokens / 1_000_000) * output_cost
    )

async def health_check(self) -> bool:
    try:
        response = await self.complete(
            messages=[LLMMessage(role="user", content="test")],
            model="claude-haiku-4-5",
            max_tokens=5
        )
        return True
    except:
        return False
```

Step 1.3: Gemini Provider

File: `backend/app/llm/providers/gemini_provider.py`

```
python
```

```python
import google.generativeai as genai
import os
import time
from typing import List
from .base import BaseLLMProvider, LLMResponse, LLMMessage

class GeminiProvider(BaseLLMProvider):
    """Google Gemini provider"""

    MODELS = {
        "gemini-2-flash": "gemini-2.0-flash-exp",
        "gemini-1.5-pro": "gemini-1.5-pro-latest",
        "gemini-1.5-flash": "gemini-1.5-flash-latest",
    }

    # Pricing per 1M tokens (input, output)
    PRICING = {
        "gemini-2-flash": (0.00, 0.00),  # Free during preview
        "gemini-1.5-pro": (1.25, 5.00),
        "gemini-1.5-flash": (0.075, 0.30),
    }

    def __init__(self):
        super().__init__(api_key=os.getenv("GEMINI_API_KEY"))
        genai.configure(api_key=self.api_key)

    async def complete(
        self,
        messages: List[LLMMessage],
        model: str = "gemini-2-flash",
        max_tokens: int = 1000,
        temperature: float = 0.7
    ) -> LLMResponse:

        start_time = time.time()

        # Get full model name
        full_model = self.MODELS.get(model, model)

        # Initialize model
        gemini_model = genai.GenerativeModel(full_model)

        # Convert messages to Gemini format
```

```
        # Gemini doesn't have system role, so prepend system message to first user message
        system_msg = next(
            (msg.content for msg in messages if msg.role == "system"),
            ""
        )

        chat_messages = []
        for msg in messages:
            if msg.role == "system":
                continue

            # Add system context to first user message
            content = msg.content
            if msg.role == "user" and system_msg and not chat_messages:
                content = f"System instructions: {system_msg}\n\nUser: {content}"

            chat_messages.append({
                "role": "user" if msg.role == "user" else "model",
                "parts": [content]
            })

        # Generate response
        response = gemini_model.generate_content(
            chat_messages,
            generation_config={
                "temperature": temperature,
                "max_output_tokens": max_tokens,
            }
        )

        latency_ms = int((time.time() - start_time) * 1000)

        # Extract text
        text = response.text

        # Estimate tokens (Gemini doesn't provide exact count)
        input_tokens = sum(len(msg.content.split()) * 1.3 for msg in messages)
        output_tokens = len(text.split()) * 1.3

        # Calculate cost
        cost = self.estimate_cost(model, int(input_tokens), int(output_tokens))

        return LLMResponse(
            text=text,
```

```
            model=full_model,
            provider="gemini",
            tokens_used=int(input_tokens + output_tokens),
            latency_ms=latency_ms,
            cost_usd=cost
        )

    def get_available_models(self) -> List[str]:
        return list(self.MODELS.keys())

    def estimate_cost(self, model: str, input_tokens: int, output_tokens: int) -> float:
        if model not in self.PRICING:
            return 0.0

        input_cost, output_cost = self.PRICING[model]

        return (
            (input_tokens / 1_000_000) * input_cost +
            (output_tokens / 1_000_000) * output_cost
        )

    async def health_check(self) -> bool:
        try:
            response = await self.complete(
                messages=[LLMMessage(role="user", content="test")],
                model="gemini-2-flash",
                max_tokens=5
            )
            return True
        except:
            return False
```

Step 1.4: OpenAI Provider

File: backend/app/llm/providers/openai_provider.py

python

```python
from openai import AsyncOpenAI
import os
import time
from typing import List
from .base import BaseLLMProvider, LLMResponse, LLMMessage

class OpenAIProvider(BaseLLMProvider):
    """OpenAI GPT provider"""

    MODELS = {
        "gpt-4-turbo": "gpt-4-turbo-preview",
        "gpt-4": "gpt-4",
        "gpt-3.5-turbo": "gpt-3.5-turbo",
    }

    # Pricing per 1M tokens (input, output)
    PRICING = {
        "gpt-4-turbo": (10.00, 30.00),
        "gpt-4": (30.00, 60.00),
        "gpt-3.5-turbo": (0.50, 1.50),
    }

    def __init__(self):
        super().__init__(api_key=os.getenv("OPENAI_API_KEY"))
        self.client = AsyncOpenAI(api_key=self.api_key)

    async def complete(
        self,
        messages: List[LLMMessage],
        model: str = "gpt-3.5-turbo",
        max_tokens: int = 1000,
        temperature: float = 0.7
    ) -> LLMResponse:

        start_time = time.time()

        # Convert to OpenAI format
        openai_messages = [
            {"role": msg.role, "content": msg.content}
            for msg in messages
        ]

        # Get full model name
```

```
        full_model = self.MODELS.get(model, model)

        # Call API
        response = await self.client.chat.completions.create(
            model=full_model,
            messages=openai_messages,
            max_tokens=max_tokens,
            temperature=temperature
        )

        latency_ms = int((time.time() - start_time) * 1000)

        # Extract text
        text = response.choices[0].message.content

        # Calculate cost
        input_tokens = response.usage.prompt_tokens
        output_tokens = response.usage.completion_tokens
        cost = self.estimate_cost(model, input_tokens, output_tokens)

        return LLMResponse(
            text=text,
            model=full_model,
            provider="openai",
            tokens_used=input_tokens + output_tokens,
            latency_ms=latency_ms,
            cost_usd=cost
        )

    def get_available_models(self) -> List[str]:
        return list(self.MODELS.keys())

    def estimate_cost(self, model: str, input_tokens: int, output_tokens: int) -> float:
        if model not in self.PRICING:
            return 0.0

        input_cost, output_cost = self.PRICING[model]

        return (
            (input_tokens / 1_000_000) * input_cost +
            (output_tokens / 1_000_000) * output_cost
        )

    async def health_check(self) -> bool:
```

```
        try:
            response = await self.complete(
                messages=[LLMMessage(role="user", content="test")],
                model="gpt-3.5-turbo",
                max_tokens=5
            )
            return True
        except:
            return False
```

Step 1.5: Ollama Provider (Local, FREE)

File: backend/app/llm/providers/ollama_provider.py

```
python
```

```python
import aiohttp
import time
from typing import List
from .base import BaseLLMProvider, LLMResponse, LLMMessage

class OllamaProvider(BaseLLMProvider):
    """Ollama local LLM provider (FREE)"""

    MODELS = {
        "llama-3.2": "llama3.2:latest",
        "llama-3.1": "llama3.1:latest",
        "mistral": "mistral:latest",
        "gemma-2": "gemma2:latest",
    }

    def __init__(self, base_url: str = "http://localhost:11434"):
        super().__init__()
        self.base_url = base_url

    async def complete(
        self,
        messages: List[LLMMessage],
        model: str = "llama-3.2",
        max_tokens: int = 1000,
        temperature: float = 0.7
    ) -> LLMResponse:

        start_time = time.time()

        # Get full model name
        full_model = self.MODELS.get(model, model)

        # Convert to Ollama format
        ollama_messages = [
            {"role": msg.role, "content": msg.content}
            for msg in messages
        ]

        # Call local Ollama API
        async with aiohttp.ClientSession() as session:
            async with session.post(
                f"{self.base_url}/api/chat",
                json={
```

```
                "model": full_model,
                "messages": ollama_messages,
                "stream": False,
                "options": {
                    "temperature": temperature,
                    "num_predict": max_tokens
                }
            }
        ) as response:
            data = await response.json()

    latency_ms = int((time.time() - start_time) * 1000)

    # Extract text
    text = data["message"]["content"]

    # Estimate tokens
    input_tokens = sum(len(msg.content.split()) for msg in messages)
    output_tokens = len(text.split())

    return LLMResponse(
        text=text,
        model=full_model,
        provider="ollama",
        tokens_used=input_tokens + output_tokens,
        latency_ms=latency_ms,
        cost_usd=0.0  # FREE!
    )

def get_available_models(self) -> List[str]:
    return list(self.MODELS.keys())

def estimate_cost(self, model: str, input_tokens: int, output_tokens: int) -> float:
    return 0.0  # Always free

async def health_check(self) -> bool:
    try:
        async with aiohttp.ClientSession() as session:
            async with session.get(f"{self.base_url}/api/tags") as response:
                return response.status == 200
    except:
        return False
```

Step 1.6: Groq Provider (Cloud, FREE Tier)

File: backend/app/llm/providers/groq_provider.py

```python
```

```python
from groq import AsyncGroq
import os
import time
from typing import List
from .base import BaseLLMProvider, LLMResponse, LLMMessage

class GroqProvider(BaseLLMProvider):
    """Groq cloud LLM provider (FREE tier available)"""

    MODELS = {
        "llama-3.3": "llama-3.3-70b-versatile",
        "llama-3.1": "llama-3.1-70b-versatile",
        "mixtral": "mixtral-8x7b-32768",
    }

    # Pricing per 1M tokens (input, output) - FREE tier: 30 req/min
    PRICING = {
        "llama-3.3": (0.59, 0.79),
        "llama-3.1": (0.59, 0.79),
        "mixtral": (0.24, 0.24),
    }

    def __init__(self):
        super().__init__(api_key=os.getenv("GROQ_API_KEY"))
        self.client = AsyncGroq(api_key=self.api_key)

    async def complete(
        self,
        messages: List[LLMMessage],
        model: str = "llama-3.3",
        max_tokens: int = 1000,
        temperature: float = 0.7
    ) -> LLMResponse:

        start_time = time.time()

        # Convert to Groq format
        groq_messages = [
            {"role": msg.role, "content": msg.content}
            for msg in messages
        ]

        # Get full model name
```

```
        full_model = self.MODELS.get(model, model)

        # Call API
        response = await self.client.chat.completions.create(
            model=full_model,
            messages=groq_messages,
            max_tokens=max_tokens,
            temperature=temperature
        )

        latency_ms = int((time.time() - start_time) * 1000)

        # Extract text
        text = response.choices[0].message.content

        # Calculate cost
        input_tokens = response.usage.prompt_tokens
        output_tokens = response.usage.completion_tokens
        cost = self.estimate_cost(model, input_tokens, output_tokens)

        return LLMResponse(
            text=text,
            model=full_model,
            provider="groq",
            tokens_used=input_tokens + output_tokens,
            latency_ms=latency_ms,
            cost_usd=cost
        )

    def get_available_models(self) -> List[str]:
        return list(self.MODELS.keys())

    def estimate_cost(self, model: str, input_tokens: int, output_tokens: int) -> float:
        if model not in self.PRICING:
            return 0.0

        input_cost, output_cost = self.PRICING[model]

        return (
            (input_tokens / 1_000_000) * input_cost +
            (output_tokens / 1_000_000) * output_cost
        )

    async def health_check(self) -> bool:
```

```
        try:
            response = await self.complete(
                messages=[LLMMessage(role="user", content="test")],
                model="llama-3.3",
                max_tokens=5
            )
            return True
        except:
            return False
```

PHASE 2: SMART ROUTER

Step 2.1: Complexity Analyzer

File: backend/app/llm/complexity_analyzer.py

```
python
```

```python
from typing import List
from enum import Enum

class QueryComplexity(Enum):
    CACHED = 0      # Exact cache hit
    TRIVIAL = 1     # Greetings, simple Q&A
    SIMPLE = 2      # FAQs, basic info
    MEDIUM = 3      # Explanations, summaries
    COMPLEX = 4     # Analysis, creative writing
    EXPERT = 5      # Advanced reasoning, critical tasks

class ComplexityAnalyzer:
    """Determines query complexity for routing decisions"""

    # Keywords that indicate complexity
    TRIVIAL_KEYWORDS = ["hi", "hello", "hey", "thanks", "bye", "ok"]
    SIMPLE_KEYWORDS = ["what", "when", "where", "who", "how much", "price"]
    COMPLEX_KEYWORDS = ["analyze", "compare", "evaluate", "design", "create", "write"]
    EXPERT_KEYWORDS = ["expert", "advanced", "professional", "critical", "urgent"]

    def analyze(self, message: str, context: List[str] = None) -> QueryComplexity:
        """
        Determine complexity of a user query

        Args:
            message: User's message
            context: Previous messages in conversation

        Returns:
            QueryComplexity enum
        """

        message_lower = message.lower()
        message_length = len(message.split())

        # Very short messages are usually trivial
        if message_length <= 3:
            if any(kw in message_lower for kw in self.TRIVIAL_KEYWORDS):
                return QueryComplexity.TRIVIAL

        # Check for expert-level indicators
        if any(kw in message_lower for kw in self.EXPERT_KEYWORDS):
            return QueryComplexity.EXPERT
```

```
        # Check for complex task indicators
        if any(kw in message_lower for kw in self.COMPLEX_KEYWORDS):
            return QueryComplexity.COMPLEX

        # Long messages with technical content
        if message_length > 100:
            return QueryComplexity.COMPLEX

        # Simple information requests
        if any(kw in message_lower for kw in self.SIMPLE_KEYWORDS):
            return QueryComplexity.SIMPLE

        # Medium-length messages
        if message_length > 20:
            return QueryComplexity.MEDIUM

        # Default to simple
        return QueryComplexity.SIMPLE

    def should_use_cache(self, message: str) -> bool:
        """Check if query is likely to have cached response"""
        message_lower = message.lower()

        # Common questions that can be cached
        cache_indicators = [
            "how much",
            "what is your",
            "do you offer",
            "what are your hours",
            "where are you located"
        ]

        return any(indicator in message_lower for indicator in cache_indicators)
```

Step 2.2: Smart Router

File: backend/app/llm/router.py

```
python
```

```python
from typing import List, Optional, Dict
from .complexity_analyzer import ComplexityAnalyzer, QueryComplexity
from .providers.base import BaseLLMProvider, LLMMessage, LLMResponse
from .providers.anthropic_provider import AnthropicProvider
from .providers.gemini_provider import GeminiProvider
from .providers.openai_provider import OpenAIProvider
from .providers.ollama_provider import OllamaProvider
from .providers.groq_provider import GroqProvider
import logging

logger = logging.getLogger(__name__)

class SmartRouter:
    """Intelligent routing to optimal LLM provider"""

    def __init__(self):
        self.analyzer = ComplexityAnalyzer()

        # Initialize all providers
        self.providers: Dict[str, BaseLLMProvider] = {
            "anthropic": AnthropicProvider(),
            "gemini": GeminiProvider(),
            "openai": OpenAIProvider(),
            "ollama": OllamaProvider(),
            "groq": GroqProvider(),
        }

        # Routing rules by complexity
        self.routing_rules = {
            QueryComplexity.TRIVIAL: [
                ("ollama", "llama-3.2"),
                ("groq", "llama-3.3"),
                ("gemini", "gemini-2-flash"),
            ],
            QueryComplexity.SIMPLE: [
                ("ollama", "llama-3.2"),
                ("groq", "llama-3.3"),
                ("gemini", "gemini-2-flash"),
            ],
            QueryComplexity.MEDIUM: [
                ("groq", "llama-3.3"),
                ("gemini", "gemini-2-flash"),
                ("openai", "gpt-3.5-turbo"),
```

```
            ],
            QueryComplexity.COMPLEX: [
                ("gemini", "gemini-1.5-pro"),
                ("anthropic", "claude-sonnet-4-5"),
                ("openai", "gpt-4-turbo"),
            ],
            QueryComplexity.EXPERT: [
                ("anthropic", "claude-opus-4-5"),
                ("openai", "gpt-4-turbo"),
                ("anthropic", "claude-sonnet-4-5"),
            ],
        }

    async def route_and_complete(
        self,
        messages: List[LLMMessage],
        agent_role: Optional[str] = None,
        force_provider: Optional[str] = None,
        force_model: Optional[str] = None
    ) -> LLMResponse:
        """
        Intelligently route request to optimal provider

        Args:
            messages: Conversation messages
            agent_role: Optional role for context (e.g., "Technical Support")
            force_provider: Force specific provider (for testing)
            force_model: Force specific model (for testing)

        Returns:
            LLMResponse with result and metadata
        """

        # Extract user message for analysis
        user_message = next(
            (msg.content for msg in reversed(messages) if msg.role == "user"),
            ""
        )

        # Determine complexity
        complexity = self.analyzer.analyze(user_message)

        logger.info(f"Query complexity: {complexity.name} - '{user_message[:50]}...'")
```

```
        # Get routing options for this complexity
        routing_options = self.routing_rules.get(complexity, [])

        # Try providers in order of preference
        for provider_name, model_name in routing_options:
            # Skip if forced provider specified
            if force_provider and provider_name != force_provider:
                continue

            provider = self.providers.get(provider_name)
            if not provider:
                continue

            # Use forced model if specified
            if force_model:
                model_name = force_model

            try:
                # Check if provider is available
                is_available = await provider.health_check()
                if not is_available:
                    logger.warning(f"Provider {provider_name} unavailable, trying next...")
                    continue

                # Make request
                response = await provider.complete(
                    messages=messages,
                    model=model_name,
                    max_tokens=2000,
                    temperature=0.7
                )

                logger.info(
                    f"Routed to {provider_name}/{model_name} - "
                    f"${response.cost_usd:.4f}, {response.latency_ms}ms"
                )

                return response

            except Exception as e:
                logger.error(f"Error with {provider_name}/{model_name}: {e}")
                continue

        # Fallback to Claude if all else fails
```

```
        logger.warning("All providers failed, falling back to Claude Sonnet")
        return await self.providers["anthropic"].complete(
            messages=messages,
            model="claude-sonnet-4-5"
        )

    async def get_provider_status(self) -> Dict[str, bool]:
        """Check health of all providers"""
        status = {}
        for name, provider in self.providers.items():
            status[name] = await provider.health_check()
        return status
```

PHASE 3: RESPONSE CACHING

Step 3.1: Cache System

File: backend/app/llm/cache.py

```
python
```

```python
from typing import Optional, List
import hashlib
import json
from datetime import datetime, timedelta
from .providers.base import LLMResponse, LLMMessage

class ResponseCache:
    """Cache LLM responses to reduce costs"""

    def __init__(self, db):
        self.db = db

    def _generate_cache_key(self, messages: List[LLMMessage]) -> str:
        """Generate unique cache key from messages"""
        # Create hash of message content
        content = json.dumps([
            {"role": msg.role, "content": msg.content}
            for msg in messages
        ], sort_keys=True)

        return hashlib.sha256(content.encode()).hexdigest()

    async def get(self, messages: List[LLMMessage]) -> Optional[LLMResponse]:
        """Get cached response if available"""

        cache_key = self._generate_cache_key(messages)

        # Look up in database
        result = await self.db.fetchrow(
            """
            SELECT response_text, model, provider, tokens_used, latency_ms, cost_usd
            FROM response_cache
            WHERE cache_key = $1
            AND created_at > NOW() - INTERVAL '7 days'
            """,
            cache_key
        )

        if not result:
            return None

        # Update hit count
        await self.db.execute(
```

```
            """
            UPDATE response_cache
            SET hits = hits + 1, last_hit = NOW()
            WHERE cache_key = $1
            """,
            cache_key
        )

        return LLMResponse(
            text=result['response_text'],
            model=result['model'],
            provider=result['provider'],
            tokens_used=result['tokens_used'],
            latency_ms=result['latency_ms'],
            cost_usd=result['cost_usd'],
            cached=True
        )

    async def set(
        self,
        messages: List[LLMMessage],
        response: LLMResponse
    ):
        """Cache a response"""

        cache_key = self._generate_cache_key(messages)

        await self.db.execute(
            """
            INSERT INTO response_cache (
                cache_key,
                response_text,
                model,
                provider,
                tokens_used,
                latency_ms,
                cost_usd
            ) VALUES ($1, $2, $3, $4, $5, $6, $7)
            ON CONFLICT (cache_key) DO UPDATE SET
                response_text = EXCLUDED.response_text,
                hits = response_cache.hits + 1,
                last_hit = NOW()
            """,
            cache_key,
```

```
            response.text,
            response.model,
            response.provider,
            response.tokens_used,
            response.latency_ms,
            response.cost_usd
        )
```

Step 3.2: Cache Table Schema

sql

```
-- Add to backend/app/schema.py

CREATE TABLE IF NOT EXISTS response_cache (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    cache_key VARCHAR(64) UNIQUE NOT NULL,
    response_text TEXT NOT NULL,
    model VARCHAR(100) NOT NULL,
    provider VARCHAR(50) NOT NULL,
    tokens_used INTEGER,
    latency_ms INTEGER,
    cost_usd DECIMAL(10,6),
    hits INTEGER DEFAULT 1,
    created_at TIMESTAMPTZ DEFAULT NOW(),
    last_hit TIMESTAMPTZ DEFAULT NOW()
);

CREATE INDEX idx_cache_key ON response_cache(cache_key);
CREATE INDEX idx_cache_created ON response_cache(created_at DESC);
```

PHASE 4: INTEGRATION

Step 4.1: Update LiteLLM Client

File: `backend/app/llm/litellm_client.py` (REPLACE)

python

```python
from typing import Dict, Optional, List
from app.llm.router import SmartRouter
from app.llm.cache import ResponseCache
from app.llm.providers.base import LLMMessage
from app.database import get_db
import logging

logger = logging.getLogger(__name__)

# Initialize router (singleton)
router = SmartRouter()
cache = None

async def execute_via_litellm(
    agent_code: str,
    org_id: str,
    message: str,
    conversation_history: Optional[List[Dict]] = None,
    use_cache: bool = True
) -> Dict:
    """
    Execute LLM request with smart routing

    Args:
        agent_code: Agent identifier
        org_id: Organization ID
        message: User message
        conversation_history: Previous messages
        use_cache: Whether to use cache (default True)

    Returns:
        Dict with response, model info, cost, latency
    """

    global cache
    if cache is None:
        db = await get_db()
        cache = ResponseCache(db)

    # Get agent details
    db = await get_db()
    agent = await db.fetchrow(
        "SELECT system_prompt, role FROM agent_catalog WHERE agent_code = $1",
```

```
    agent_code
)

if not agent:
    raise ValueError(f"Agent {agent_code} not found")

# Build messages
messages = [
    LLMMessage(role="system", content=agent['system_prompt'])
]

# Add conversation history
if conversation_history:
    for msg in conversation_history:
        messages.append(LLMMessage(
            role=msg["role"],
            content=msg["content"]
        ))

# Add current message
messages.append(LLMMessage(role="user", content=message))

# Try cache first
if use_cache:
    cached_response = await cache.get(messages)
    if cached_response:
        logger.info(f"Cache HIT for {agent_code} - Saved ${cached_response.cost_usd:.4f}")

        # Store interaction with cache indicator
        interaction_id = await _store_interaction(
            db, org_id, agent_code, message,
            cached_response.text, cached_response.model,
            cached_response.latency_ms, cached=True
        )

        return {
            "response": cached_response.text,
            "model": cached_response.model,
            "provider": cached_response.provider,
            "latency_ms": cached_response.latency_ms,
            "cost_usd": 0.0,  # No cost for cached
            "tokens_used": cached_response.tokens_used,
            "cached": True,
            "interaction_id": interaction_id
```

```
        }

    # Route to optimal provider
    response = await router.route_and_complete(
        messages=messages,
        agent_role=agent['role']
    )

    # Cache successful response
    if use_cache:
        await cache.set(messages, response)

    # Store interaction
    interaction_id = await _store_interaction(
        db, org_id, agent_code, message,
        response.text, response.model,
        response.latency_ms, cached=False
    )

    return {
        "response": response.text,
        "model": response.model,
        "provider": response.provider,
        "latency_ms": response.latency_ms,
        "cost_usd": response.cost_usd,
        "tokens_used": response.tokens_used,
        "cached": False,
        "interaction_id": interaction_id
    }

async def _store_interaction(
    db, org_id, agent_code, message, response,
    model, latency_ms, cached
):
    """Store interaction in database"""

    result = await db.fetchrow(
        """
        INSERT INTO interaction_logs (
            org_id, agent_code, message, response,
            model_used, latency_ms, cached
        ) VALUES ($1, $2, $3, $4, $5, $6, $7)
        RETURNING id
        """,
```

```
        org_id, agent_code, message, response,
        model, latency_ms, cached
    )

    return str(result['id'])
```

Step 4.2: Update Database Schema

Add cached column to interaction_logs:

```sql
ALTER TABLE interaction_logs
ADD COLUMN IF NOT EXISTS cached BOOLEAN DEFAULT FALSE;
```

PHASE 5: TESTING & VALIDATION

Step 5.1: Test All Providers

File: `backend/scripts/test_providers.py`

```python
```

```python
import asyncio
from app.llm.providers.base import LLMMessage
from app.llm.router import SmartRouter

async def test_all_providers():
    router = SmartRouter()

    test_message = [
        LLMMessage(role="user", content="Hello, can you tell me what 2+2 equals?")
    ]

    print("Testing all LLM providers...\n")

    providers_to_test = [
        ("ollama", "llama-3.2"),
        ("groq", "llama-3.3"),
        ("gemini", "gemini-2-flash"),
        ("openai", "gpt-3.5-turbo"),
        ("anthropic", "claude-haiku-4-5"),
    ]

    for provider, model in providers_to_test:
        print(f"Testing {provider}/{model}...")

        try:
            response = await router.route_and_complete(
                messages=test_message,
                force_provider=provider,
                force_model=model
            )

            print(f"✅ SUCCESS")
            print(f"   Response: {response.text[:100]}...")
            print(f"   Cost: ${response.cost_usd:.6f}")
            print(f"   Latency: {response.latency_ms}ms")
            print(f"   Tokens: {response.tokens_used}")

        except Exception as e:
            print(f"❌ FAILED: {e}")

        print()
```

```
if __name__ == "__main__":
    asyncio.run(test_all_providers())
```

Step 5.2: Test Smart Routing

File: backend/scripts/test_routing.py

```python
import asyncio
from app.llm.providers.base import LLMMessage
from app.llm.router import SmartRouter

async def test_routing():
    router = SmartRouter()

    test_queries = [
        ("Hi", "TRIVIAL"),
        ("What are your hours?", "SIMPLE"),
        ("Can you explain how photosynthesis works?", "MEDIUM"),
        ("Write a detailed analysis comparing React and Vue.js for enterprise applications", "COMPLEX"),
        ("I need expert advice on optimizing database queries for a high-traffic application", "EXPERT"),
    ]

    print("Testing smart routing...\n")

    for query, expected_complexity in test_queries:
        print(f"Query: {query}")
        print(f"Expected: {expected_complexity}")

        messages = [LLMMessage(role="user", content=query)]

        response = await router.route_and_complete(messages)

        print(f"Routed to: {response.provider}/{response.model}")
        print(f"Cost: ${response.cost_usd:.6f}")
        print(f"Latency: {response.latency_ms}ms")
        print()

if __name__ == "__main__":
    asyncio.run(test_routing())
```

DELIVERABLES CHECKLIST

- [ ] All provider classes created (Anthropic, Gemini, OpenAI, Ollama, Groq)
- [ ] Complexity analyzer implemented
- [ ] Smart router implemented
- [ ] Response caching system
- [ ] Database schema updated
- [ ] LiteLLM client updated to use router
- [ ] Test scripts pass
- [ ] All providers health check pass
- [ ] Routing logic validated
- [ ] Cache working correctly

EXPECTED RESULTS

Cost Reduction:

- 70-80% of queries → Free providers (Ollama, Groq)
- 15-20% of queries → Mid-tier ($)
- 5-10% of queries → Premium ($$$)
- **Overall: 80%+ cost reduction**

Performance:

- Ollama: 200-500ms (local)
- Groq: 200-800ms (very fast cloud)
- Gemini Flash: 500-1500ms
- Others: 2-5 seconds
- **Average improvement: 2-3x faster**

Autonomy:

- Not dependent on any single provider
- Automatic failover if provider down
- Cost-optimized routing
- Scalable to millions of requests

NEXT STEPS AFTER COMPLETION

1. **Add Web Search** (agents can find info)
2. **Add Document Retrieval** (search knowledge base)
3. **Real-User Learning** (improve from feedback)
4. **Advanced Routing** (learn which model best for each agent)

This is the foundation for truly autonomous agents.