### 9.1 AI Router Design (Phased)

The AI Router is a sophisticated system that intelligently selects the optimal AI model for each user query based on content analysis, philosophical context, cost optimization, and quality requirements. The implementation follows a three-phase approach to ensure robust functionality while managing complexity.

#### 9.1.1 Phase 1: Foundational Router

**Timeline**: MVP Launch **Goal**: Establish basic model selection logic with reliable fallback mechanisms

**Core Components**:

1. **Model Registry**:

class AIModel:

name: str

provider: str

cost\_per\_token: float

max\_tokens: int

strengths: List[str]

philosophical\_tones: List[str]

availability\_score: float

MODELS = {

"grok": AIModel(

name="grok",

provider="xai",

cost\_per\_token=0.00002,

max\_tokens=4096,

strengths=["creative", "conversational", "humor"],

philosophical\_tones=["socratic", "existential", "pragmatic"],

availability\_score=0.95

),

"claude": AIModel(

name="claude",

provider="anthropic",

cost\_per\_token=0.00003,

max\_tokens=8192,

strengths=["analytical", "structured", "ethical"],

philosophical\_tones=["analytical", "stoic", "kantian"],

availability\_score=0.98

),

"gemini": AIModel(

name="gemini",

provider="google",

cost\_per\_token=0.000015,

max\_tokens=8192,

strengths=["factual", "comprehensive", "multilingual"],

philosophical\_tones=["academic", "historical", "comparative"],

availability\_score=0.97

),

"chatgpt": AIModel(

name="chatgpt",

provider="openai",

cost\_per\_token=0.00001,

max\_tokens=4096,

strengths=["balanced", "accessible", "educational"],

philosophical\_tones=["educational", "dialectical", "modern"],

availability\_score=0.99

)

}

1. **Basic Selection Logic**:

def select\_model\_phase1(question: str, tone: str, user\_tier: str) -> str:

# Rule-based selection

if tone in ["socratic", "existential"]:

return "grok"

elif tone in ["analytical", "stoic"]:

return "claude"

elif tone in ["academic", "historical"]:

return "gemini"

else:

return "chatgpt" # Default fallback

1. **Fallback Mechanism**:
   * Primary model selection based on tone
   * Secondary selection if primary fails (availability/rate limits)
   * Tertiary fallback to most reliable model (ChatGPT)
   * Error handling with graceful degradation

**Phase 1 Features**:

* Static rule-based model selection
* Basic cost tracking
* Simple fallback logic
* Response time monitoring
* Error logging and alerting

#### 9.1.2 Phase 2: Advanced Integration

**Timeline**: 3-6 months post-MVP **Goal**: Implement dynamic selection with user context and performance optimization

**Enhanced Components**:

1. **Context Analyzer**:

class ContextAnalyzer:

def analyze\_question(self, question: str) -> Dict[str, float]:

return {

"complexity\_score": self.\_calculate\_complexity(question),

"philosophical\_depth": self.\_assess\_depth(question),

"concept\_categories": self.\_extract\_concepts(question),

"emotional\_tone": self.\_detect\_emotion(question),

"length\_category": self.\_categorize\_length(question)

}

1. **User Context Integration**:

class UserContextService:

def get\_user\_context(self, user\_id: str) -> Dict[str, Any]:

return {

"subscription\_tier": self.\_get\_subscription(user\_id),

"usage\_history": self.\_get\_recent\_usage(user\_id),

"preferred\_models": self.\_get\_preferences(user\_id),

"concept\_mastery": self.\_get\_mastery\_levels(user\_id),

"interaction\_patterns": self.\_analyze\_patterns(user\_id)

}

1. **Dynamic Selection Algorithm**:

def select\_model\_phase2(

question: str,

tone: str,

user\_context: Dict[str, Any]

) -> Tuple[str, float]:

context = self.analyzer.analyze\_question(question)

scores = {}

for model\_name, model in MODELS.items():

score = self.\_calculate\_model\_score(

model, context, tone, user\_context

)

scores[model\_name] = score

# Select highest scoring available model

selected\_model = max(scores, key=scores.get)

confidence = scores[selected\_model]

return selected\_model, confidence

**Phase 2 Features**:

* Dynamic model selection based on question analysis
* User behavior pattern recognition
* Cost optimization algorithms
* A/B testing framework for model performance
* Response quality scoring
* Adaptive learning from user feedback

#### 9.1.3 Phase 3: ML Enhancement

**Timeline**: 6-12 months post-MVP **Goal**: Implement machine learning for intelligent model selection and quality prediction

**ML Components**:

1. **Model Selection ML Pipeline**:

class ModelSelectionML:

def \_\_init\_\_(self):

self.feature\_extractor = FeatureExtractor()

self.model\_selector = MLModelSelector()

self.quality\_predictor = QualityPredictor()

def predict\_optimal\_model(

self,

question: str,

tone: str,

user\_features: Dict

) -> Tuple[str, float, Dict[str, float]]:

features = self.feature\_extractor.extract(

question, tone, user\_features

)

model\_probabilities = self.model\_selector.predict(features)

quality\_scores = self.quality\_predictor.predict\_all(features)

optimal\_model = self.\_select\_with\_quality\_cost\_tradeoff(

model\_probabilities, quality\_scores

)

return optimal\_model, confidence, quality\_scores

1. **Training Data Collection**:
   * User satisfaction ratings
   * Response quality metrics
   * Engagement measurements
   * Cost efficiency tracking
   * Model performance analytics
2. **Continuous Learning**:
   * Online learning algorithms
   * Feedback loop integration
   * Model performance drift detection
   * Automatic retraining pipelines

**Phase 3 Features**:

* ML-driven model selection
* Predictive quality scoring
* Personalized model preferences
* Real-time performance optimization
* Advanced cost-quality trade-off algorithms

### 9.2 Model Selection Logic (Grok, Claude, Gemini, ChatGPT)

Each AI model has distinct strengths that make them optimal for different types of philosophical inquiries:

#### Model Characteristics

**Grok (xAI)**:

* **Strengths**: Creative thinking, conversational tone, humor integration, unconventional perspectives
* **Optimal For**: Socratic questioning, existential exploration, creative philosophical thought experiments
* **Philosophical Tones**: Socratic, Existential, Pragmatic, Nietzschean
* **Cost**: Medium ($0.00002/token)
* **Use Cases**:
  + "What if" scenarios
  + Paradox exploration
  + Creative philosophical analogies
  + Challenging conventional wisdom

**Claude (Anthropic)**:

* **Strengths**: Analytical rigor, structured reasoning, ethical considerations, nuanced arguments
* **Optimal For**: Complex ethical dilemmas, structured philosophical analysis, academic-style discussions
* **Philosophical Tones**: Analytical, Stoic, Kantian, Utilitarian
* **Cost**: High ($0.00003/token)
* **Use Cases**:
  + Ethical decision-making frameworks
  + Logical argument construction
  + Moral philosophy discussions
  + Academic philosophical analysis

**Gemini (Google)**:

* **Strengths**: Factual accuracy, comprehensive knowledge, historical context, multilingual capabilities
* **Optimal For**: Historical philosophy, comparative analysis, factual philosophical information
* **Philosophical Tones**: Academic, Historical, Comparative, Aristotelian
* **Cost**: Low ($0.000015/token)
* **Use Cases**:
  + Historical philosophical movements
  + Comparative philosophy (East vs West)
  + Factual philosophical information
  + Timeline and context questions

**ChatGPT (OpenAI)**:

* **Strengths**: Balanced responses, educational clarity, accessibility, general versatility
* **Optimal For**: Educational content, beginner-friendly explanations, general philosophical discussions
* **Philosophical Tones**: Educational, Dialectical, Modern, Platonic
* **Cost**: Very Low ($0.00001/token)
* **Use Cases**:
  + Introduction to philosophical concepts
  + Balanced perspective presentations
  + Educational explanations
  + General philosophical guidance

#### Selection Algorithm

def calculate\_model\_score(

model: AIModel,

question\_context: Dict[str, float],

tone: str,

user\_context: Dict[str, Any]

) -> float:

score = 0.0

# Tone compatibility (40% weight)

if tone in model.philosophical\_tones:

score += 0.4

# Question complexity match (25% weight)

complexity = question\_context["complexity\_score"]

if model.name == "claude" and complexity > 0.7:

score += 0.25

elif model.name == "grok" and 0.3 < complexity < 0.7:

score += 0.25

elif model.name == "chatgpt" and complexity < 0.5:

score += 0.25

# Cost optimization (20% weight)

user\_tier = user\_context["subscription\_tier"]

if user\_tier == "free":

# Prefer lower cost models for free users

cost\_score = 1.0 - (model.cost\_per\_token / 0.00003)

score += 0.2 \* cost\_score

elif user\_tier == "premium":

# Balanced cost-quality for premium users

score += 0.15

else: # pro

# Quality over cost for pro users

score += 0.2

# Availability (10% weight)

score += 0.1 \* model.availability\_score

# User preference (5% weight)

preferred\_models = user\_context.get("preferred\_models", [])

if model.name in preferred\_models:

score += 0.05

return min(score, 1.0)

### 9.3 Context-Specific Tone Prompt Workflow

The Context-Specific Tone Prompt Workflow ensures consistent, high-quality philosophical responses across all AI models by using carefully crafted prompts tailored to each model's strengths and each philosophical tone's requirements.

#### Prompt Template Structure

class PromptTemplate:

def \_\_init\_\_(self, tone: str, model: str):

self.tone = tone

self.model = model

self.base\_prompt = self.\_get\_base\_prompt()

self.tone\_instructions = self.\_get\_tone\_instructions()

self.model\_optimizations = self.\_get\_model\_optimizations()

def format\_prompt(self, question: str, user\_context: Dict) -> str:

return f"""

{self.base\_prompt}

{self.tone\_instructions}

{self.model\_optimizations}

User Question: {question}

Response Guidelines:

- Maintain the {self.tone} philosophical tone throughout

- Provide insights that encourage further reflection

- Include 2-3 relevant philosophical concepts

- Keep response length appropriate for mobile reading

Please respond:

"""

#### Tone-Specific Prompt Templates

**Socratic Tone**:

SOCRATIC\_PROMPTS = {

"grok": """

You are embodying the Socratic method of philosophical inquiry. Your role is to guide the user toward deeper understanding through thoughtful questioning rather than providing direct answers. Use Socrates' approach of claiming to know nothing while helping others discover knowledge through careful examination of their beliefs.

Grok-specific optimizations:

- Embrace intellectual humility with a touch of wit

- Use creative analogies and thought experiments

- Challenge assumptions with gentle humor

- Encourage the user to question their own certainties

""",

"claude": """

You are applying the Socratic method with analytical precision. Your responses should demonstrate the systematic questioning approach that Socrates used to expose contradictions and guide people toward truth through rigorous examination of their beliefs.

Claude-specific optimizations:

- Structure your questions logically and progressively

- Identify and examine underlying assumptions methodically

- Use precise philosophical terminology

- Build arguments step-by-step through questioning

"""

}

**Analytical Tone**:

ANALYTICAL\_PROMPTS = {

"claude": """

You are providing analytical philosophical analysis with rigorous logical structure. Break down complex philosophical problems into their component parts, examine relationships between concepts, and construct clear, well-reasoned arguments.

Claude-specific optimizations:

- Use formal logical structures (premises, conclusions)

- Identify and address potential counterarguments

- Provide systematic analysis of philosophical positions

- Maintain academic rigor while remaining accessible

""",

"gemini": """

You are conducting analytical philosophical examination with comprehensive factual grounding. Provide thorough analysis that draws from the breadth of philosophical tradition while maintaining logical rigor.

Gemini-specific optimizations:

- Reference relevant historical philosophical positions

- Compare and contrast different analytical approaches

- Provide comprehensive coverage of the topic

- Ground analysis in established philosophical frameworks

"""

}

#### Dynamic Prompt Enhancement

class DynamicPromptEnhancer:

def enhance\_prompt(

self,

base\_prompt: str,

question\_analysis: Dict,

user\_context: Dict

) -> str:

enhancements = []

# Add complexity-appropriate guidance

if question\_analysis["complexity\_score"] > 0.8:

enhancements.append(

"This is a complex philosophical question. Break it down into manageable components."

)

# Add user-level appropriate language

user\_level = user\_context.get("level", 1)

if user\_level < 5:

enhancements.append(

"Use accessible language suitable for someone new to philosophy."

)

elif user\_level > 15:

enhancements.append(

"Feel free to use advanced philosophical terminology and concepts."

)

# Add concept mastery context

mastered\_concepts = user\_context.get("mastered\_concepts", [])

if mastered\_concepts:

enhancements.append(

f"The user has demonstrated understanding of: {', '.join(mastered\_concepts[:3])}"

)

enhanced\_prompt = base\_prompt

if enhancements:

enhanced\_prompt += "\n\nAdditional Context:\n" + "\n".join(enhancements)

return enhanced\_prompt

### 9.4 Cost and Performance Optimization

The AI Router implements sophisticated cost and performance optimization strategies to balance quality, speed, and expense:

#### Cost Optimization Strategies

1. **Tiered Cost Management**:

class CostOptimizer:

def \_\_init\_\_(self):

self.daily\_budgets = {

"free": 0.50, # $0.50 per user per day

"premium": 2.00, # $2.00 per user per day

"pro": 10.00 # $10.00 per user per day

}

def select\_cost\_optimal\_model(

self,

user\_tier: str,

current\_usage: float,

quality\_requirements: Dict

) -> str:

remaining\_budget = self.daily\_budgets[user\_tier] - current\_usage

if remaining\_budget < 0.01: # Less than 1 cent remaining

return "chatgpt" # Cheapest option

elif remaining\_budget < 0.05: # Less than 5 cents

return "gemini" # Low cost, good quality

else:

# Select based on quality requirements

return self.\_quality\_based\_selection(quality\_requirements)

1. **Intelligent Caching**:

class ResponseCache:

def \_\_init\_\_(self):

self.redis\_client = redis.Redis()

self.cache\_ttl = 86400 # 24 hours

def get\_cache\_key(self, question: str, tone: str) -> str:

# Normalize question for better cache hits

normalized = self.\_normalize\_question(question)

return f"ai\_response:{tone}:{hash(normalized)}"

def should\_cache(self, question: str, response: str) -> bool:

# Cache responses for common philosophical questions

return (

len(question.split()) > 5 and # Substantial questions

len(response) > 100 and # Substantial responses

self.\_is\_general\_philosophical\_question(question)

)

1. **Token Usage Optimization**:

class TokenOptimizer:

def optimize\_prompt(self, prompt: str, max\_tokens: int) -> str:

# Remove unnecessary whitespace and formatting

optimized = re.sub(r'\s+', ' ', prompt.strip())

# Truncate if necessary while preserving key information

if self.\_count\_tokens(optimized) > max\_tokens \* 0.7:

optimized = self.\_intelligent\_truncate(optimized, max\_tokens \* 0.7)

return optimized

def set\_response\_limits(self, model: str, question\_complexity: float) -> int:

base\_limits = {

"grok": 500,

"claude": 800,

"gemini": 600,

"chatgpt": 400

}

# Adjust based on complexity

limit = base\_limits[model]

if question\_complexity > 0.8:

limit = int(limit \* 1.5)

elif question\_complexity < 0.3:

limit = int(limit \* 0.7)

return limit

#### Performance Optimization

1. **Parallel Processing**:

async def get\_ai\_response\_with\_fallback(

question: str,

tone: str,

user\_context: Dict

) -> Dict[str, Any]:

primary\_model = await self.router.select\_model(question, tone, user\_context)

# Start primary request

primary\_task = asyncio.create\_task(

self.\_call\_ai\_model(primary\_model, question, tone)

)

# Start fallback request with delay

fallback\_model = self.router.get\_fallback\_model(primary\_model)

fallback\_task = asyncio.create\_task(

asyncio.sleep(2.0) # 2-second delay

.then(lambda: self.\_call\_ai\_model(fallback\_model, question, tone))

)

# Return first successful response

done, pending = await asyncio.wait(

[primary\_task, fallback\_task],

return\_when=asyncio.FIRST\_COMPLETED

)

# Cancel pending tasks

for task in pending:

task.cancel()

return done.pop().result()

1. **Response Time Monitoring**:

class PerformanceMonitor:

def \_\_init\_\_(self):

self.response\_times = defaultdict(list)

self.error\_rates = defaultdict(int)

async def monitor\_request(

self,

model: str,

request\_func: Callable

) -> Tuple[Any, float]:

start\_time = time.time()

try:

result = await request\_func()

response\_time = time.time() - start\_time

self.response\_times[model].append(response\_time)

return result, response\_time

except Exception as e:

self.error\_rates[model] += 1

raise e

def get\_model\_performance(self, model: str) -> Dict[str, float]:

times = self.response\_times[model]

return {

"avg\_response\_time": sum(times) / len(times) if times else 0,

"p95\_response\_time": np.percentile(times, 95) if times else 0,

"error\_rate": self.error\_rates[model] / len(times) if times else 0

}

### 9.5 Quality Evaluation Metrics

The AI Router continuously evaluates response quality to optimize model selection and improve user experience:

#### Quality Metrics

1. **Automated Quality Scoring**:

class QualityEvaluator:

def \_\_init\_\_(self):

self.concept\_extractor = ConceptExtractor()

self.sentiment\_analyzer = SentimentAnalyzer()

self.coherence\_checker = CoherenceChecker()

def evaluate\_response(

self,

question: str,

response: str,

tone: str

) -> Dict[str, float]:

return {

"relevance\_score": self.\_calculate\_relevance(question, response),

"philosophical\_depth": self.\_assess\_depth(response),

"tone\_consistency": self.\_check\_tone\_consistency(response, tone),

"concept\_accuracy": self.\_verify\_concepts(response),

"coherence\_score": self.coherence\_checker.score(response),

"engagement\_potential": self.\_predict\_engagement(response)

}

1. **User Feedback Integration**:

class UserFeedbackCollector:

def collect\_implicit\_feedback(self, interaction\_id: str) -> Dict[str, float]:

interaction = self.db.get\_interaction(interaction\_id)

return {

"time\_spent\_reading": interaction.reading\_time,

"saved\_response": 1.0 if interaction.is\_saved else 0.0,

"expanded\_insight": 1.0 if interaction.is\_expanded else 0.0,

"follow\_up\_questions": len(interaction.follow\_ups),

"concept\_exploration": len(interaction.concepts\_clicked)

}

def collect\_explicit\_feedback(self, interaction\_id: str) -> Dict[str, float]:

feedback = self.db.get\_feedback(interaction\_id)

if not feedback:

return {}

return {

"helpfulness\_rating": feedback.helpfulness / 5.0,

"accuracy\_rating": feedback.accuracy / 5.0,

"clarity\_rating": feedback.clarity / 5.0,

"engagement\_rating": feedback.engagement / 5.0

}

1. **Continuous Improvement Loop**:

class QualityImprovementEngine:

def \_\_init\_\_(self):

self.feedback\_aggregator = FeedbackAggregator()

self.model\_performance\_tracker = ModelPerformanceTracker()

async def update\_model\_preferences(self):

# Analyze recent performance data

performance\_data = await self.model\_performance\_tracker.get\_recent\_data()

# Update model selection weights

for model in MODELS:

quality\_score = performance\_data[model]["avg\_quality\_score"]

cost\_efficiency = performance\_data[model]["cost\_per\_quality\_point"]

# Adjust model weights based on performance

MODELS[model].selection\_weight = self.\_calculate\_weight(

quality\_score, cost\_efficiency

)

def generate\_improvement\_recommendations(self) -> List[str]:

recommendations = []

# Analyze underperforming areas

low\_quality\_tones = self.\_identify\_low\_quality\_tones()

for tone in low\_quality\_tones:

recommendations.append(

f"Improve prompt templates for {tone} tone"

)

# Analyze cost inefficiencies

high\_cost\_models = self.\_identify\_cost\_inefficient\_models()

for model in high\_cost\_models:

recommendations.append(

f"Optimize token usage for {model} model"

)

return recommendations