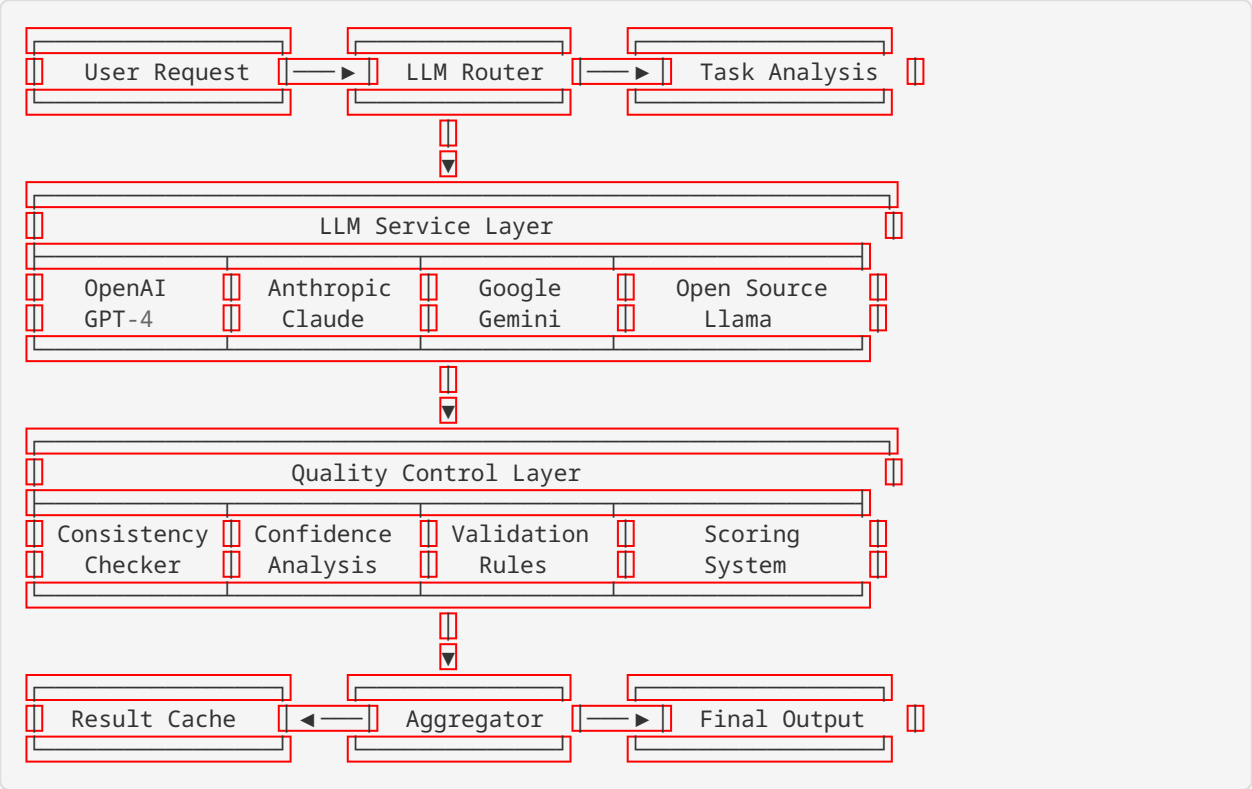
Multi-LLM Integration Architecture

Technical Implementation Guide

🏗️ System Architecture Overview



🔧 Core Implementation Components

1. LLM Service Abstraction Layer

```typescript
// Base LLM Service Interface
interface LLMService {
  readonly id: string;
  readonly provider: string;
  readonly model: string;
  readonly pricing: ModelPricing;
  readonly capabilities: ModelCapabilities;
  readonly limits: ModelLimits;

  generateCompletion(request: CompletionRequest): Promise<CompletionResponse>;
  estimateCost(request: CompletionRequest): CostEstimate;
  checkAvailability(): Promise<boolean>;
  getUsageStats(): UsageStatistics;
}

// Model Pricing Structure
interface ModelPricing {
  inputTokenCost: number;  // Cost per 1K input tokens
  outputTokenCost: number; // Cost per 1K output tokens
  requestCost?: number;    // Fixed cost per request
  monthlyMinimum?: number; // Monthly minimum charges
}

// Model Capabilities
interface ModelCapabilities {
  maxContextLength: number;
  supportedLanguages: string[];
  specializations: string[];
  qualityScore: number;
  speedScore: number;
  reasoningScore: number;
}

// Implementation for OpenAI
class OpenAIService implements LLMService {
  readonly id = 'openai-gpt4-turbo';
  readonly provider = 'OpenAI';
  readonly model = 'gpt-4-turbo';
  readonly pricing: ModelPricing = {
    inputTokenCost: 0.01,
    outputTokenCost: 0.03
  };

  private client: OpenAI;
  private rateLimiter: RateLimiter;

  constructor(apiKey: string) {
    this.client = new OpenAI({ apiKey });
    this.rateLimiter = new RateLimiter({
      tokensPerMinute: 150000,
      requestsPerMinute: 500
    });
  }

  async generateCompletion(request: CompletionRequest): Promise<CompletionResponse> {
    await this.rateLimiter.waitForCapacity(request.estimatedTokens);

    try {
      const response = await this.client.chat.completions.create({
        model: this.model,
        messages: request.messages,
        temperature: request.temperature || 0.7,
```

```
        max_tokens: request.maxTokens,
        response_format: request.responseFormat
      });

      return this.formatResponse(response, request);
    } catch (error) {
      throw new LLMServiceError(`OpenAI API error: ${error.message}`, error);
    }
  }

  estimateCost(request: CompletionRequest): CostEstimate {
    const inputTokens = this.estimateInputTokens(request);
    const outputTokens = request.maxTokens || 1000;

    return {
      inputCost: (inputTokens / 1000) * this.pricing.inputTokenCost,
      outputCost: (outputTokens / 1000) * this.pricing.outputTokenCost,
      totalCost: ((inputTokens / 1000) * this.pricing.inputTokenCost) +
                 ((outputTokens / 1000) * this.pricing.outputTokenCost)
    };
  }
}

// Implementation for Anthropic Claude
class AnthropicService implements LLMService {
  readonly id = 'anthropic-claude3-sonnet';
  readonly provider = 'Anthropic';
  readonly model = 'claude-3-sonnet';
  readonly pricing: ModelPricing = {
    inputTokenCost: 0.003,
    outputTokenCost: 0.015
  };

  private client: AnthropicClient;

  async generateCompletion(request: CompletionRequest): Promise<CompletionResponse> {
    const response = await this.client.messages.create({
      model: this.model,
      max_tokens: request.maxTokens || 4000,
      temperature: request.temperature || 0.7,
      messages: this.formatMessages(request.messages)
    });

    return this.formatResponse(response, request);
  }
}
```

2. Intelligent LLM Router

```
class LLMRouter {
  private services: Map<string, LLMService>;
  private taskModelMap: Map<string, LLMPreference[]>;
  private costOptimizer: CostOptimizer;
  private qualityAnalyzer: QualityAnalyzer;

  constructor() {
    this.initializeTaskModelMapping();
    this.costOptimizer = new CostOptimizer();
    this.qualityAnalyzer = new QualityAnalyzer();
  }

  async selectOptimalLLM(task: AnalysisTask, options: SelectionOptions = {}): Promise<L
LMService> {
    // Get candidate models for this task type
    const candidates = this.getCandidateModels(task.type);

    // Apply budget constraints
    const budgetFiltered = this.costOptimizer.filterByBudget(candidates, op-
tions.maxCost);

    // Apply quality requirements
    const qualityFiltered = this.qualityAnalyzer.filterByQuality(budgetFiltered, op-
tions.minQuality);

    // Apply availability check
    const availableModels = await this.checkAvailability(qualityFiltered);

    // Select optimal model based on scoring
    return this.scoreAndSelect(availableModels, task, options);
  }

  private initializeTaskModelMapping(): void {
    this.taskModelMap = new Map([
      ['metadata_extraction', [
        { modelId: 'openai-gpt3.5-turbo', priority: 1, costWeight: 0.8 },
        { modelId: 'google-gemini-pro', priority: 2, costWeight: 0.9 },
        { modelId: 'openai-gpt4-turbo', priority: 3, costWeight: 0.3 }
      ]],
      ['claim_analysis', [
        { modelId: 'openai-gpt4-turbo', priority: 1, qualityWeight: 0.9 },
        { modelId: 'anthropic-claude3-opus', priority: 2, qualityWeight: 0.95 },
        { modelId: 'anthropic-claude3-sonnet', priority: 3, qualityWeight: 0.8 }
      ]],
      ['prior_art_analysis', [
        { modelId: 'anthropic-claude3-sonnet', priority: 1, contextWeight: 0.9 },
        { modelId: 'openai-gpt4-turbo', priority: 2, contextWeight: 0.8 },
        { modelId: 'anthropic-claude3-opus', priority: 3, contextWeight: 0.95 }
      ]],
      ['competitive_intelligence', [
        { modelId: 'anthropic-claude3-opus', priority: 1, analysisWeight: 0.95 },
        { modelId: 'openai-gpt4-turbo', priority: 2, analysisWeight: 0.9 },
        { modelId: 'anthropic-claude3-sonnet', priority: 3, analysisWeight: 0.85 }
      ]],
      ['report_generation', [
        { modelId: 'openai-gpt4-turbo', priority: 1, writingWeight: 0.9 },
        { modelId: 'anthropic-claude3-sonnet', priority: 2, writingWeight: 0.85 },
        { modelId: 'openai-gpt3.5-turbo', priority: 3, writingWeight: 0.7 }
      ]]
    ]);
  }
```

```
  private async scoreAndSelect(
    models: LLMService[],
    task: AnalysisTask,
    options: SelectionOptions
  ): Promise<LLMService> {
    const scores = await Promise.all(
      models.map(async (model) => ({
        model,
        score: await this.calculateModelScore(model, task, options)
      }))
    );

    // Sort by score (descending) and return the best model
    scores.sort((a, b) => b.score - a.score);

    if (scores.length === 0) {
      throw new Error(`No suitable LLM found for task: ${task.type}`);
    }

    return scores[0].model;
  }

  private async calculateModelScore(
    model: LLMService,
    task: AnalysisTask,
    options: SelectionOptions
  ): Promise<number> {
    const preferences = this.getTaskPreferences(task.type, model.id);

    // Base capability scores
    const qualityScore = model.capabilities.qualityScore * (preferences?.qualityWeight
|| 0.5);
    const speedScore = model.capabilities.speedScore * (preferences?.speedWeight ||
0.3);
    const contextScore = this.calculateContextScore(model, task) * (preferences?.con-
textWeight || 0.2);

    // Cost efficiency score (inverse of cost)
    const costEstimate = model.estimateCost(this.createCompletionRequest(task));
    const costScore = (1 / (costEstimate.totalCost + 0.001)) *
(preferences?.costWeight || 0.4);

    // Historical performance score
    const historyScore = await this.getHistoricalPerformance(model.id, task.type);

    // Weighted final score
    return (qualityScore + speedScore + contextScore + costScore + historyScore) / 5;
  }
}
```

3. Multi-Model Task Execution Engine

```
class TaskExecutionEngine {
  private router: LLMRouter;
  private qualityController: QualityController;
  private fallbackManager: FallbackManager;
  private cacheManager: CacheManager;

  async executeTask(task: AnalysisTask, options: ExecutionOptions = {}): Promise<Ana-
lysisResult> {
    // Check cache first
    const cacheKey = this.generateCacheKey(task);
    const cachedResult = await this.cacheManager.get(cacheKey);
    if (cachedResult && !options.bypassCache) {
      return cachedResult;
    }

    // Select optimal LLM
    const primaryLLM = await this.router.selectOptimalLLM(task, options);

    let result: AnalysisResult;
    let attempts = 0;
    const maxAttempts = options.maxRetries || 3;

    while (attempts < maxAttempts) {
      try {
        // Execute task with selected LLM
        result = await this.executeWithLLM(task, primaryLLM);

        // Quality check
        const qualityCheck = await this.qualityController.validateResult(result, task);

        if (qualityCheck.isAcceptable) {
          // Cache successful result
          await this.cacheManager.set(cacheKey, result, { ttl: 3600 });
          return result;
        } else if (qualityCheck.requiresFallback) {
          // Try fallback model
          const fallbackLLM = await this.fallbackManager.getFallbackModel(primaryLLM, t
ask);
          result = await this.executeWithLLM(task, fallbackLLM);

          const fallbackQuality = await this.qualityController.validateResult(result, t
ask);
          if (fallbackQuality.isAcceptable) {
            await this.cacheManager.set(cacheKey, result, { ttl: 1800 });
            return result;
          }
        }

        attempts++;
      } catch (error) {
        attempts++;
        if (attempts >= maxAttempts) {
          throw new TaskExecutionError(`Task failed after ${maxAttempts} attempts: ${er
ror.message}`);
        }

        // Wait before retry with exponential backoff
        await this.delay(Math.pow(2, attempts) * 1000);
      }
    }

    throw new TaskExecutionError(`Task execution failed after ${maxAttempts} attempts`)
```

```
;
  }

  private async executeWithLLM(task: AnalysisTask, llm: LLMService): Promise<Analys-
isResult> {
    const request = this.buildCompletionRequest(task, llm);
    const response = await llm.generateCompletion(request);

    return {
      taskId: task.id,
      modelUsed: llm.id,
      content: response.content,
      confidence: response.confidence,
      tokensUsed: response.usage.totalTokens,
      cost: response.cost,
      timestamp: new Date(),
      metadata: {
        model: llm.model,
        provider: llm.provider,
        processingTime: response.processingTime
      }
    };
  }

  private buildCompletionRequest(task: AnalysisTask, llm: LLMService): CompletionRe-
quest {
    const prompts = this.getTaskPrompts(task.type);
    const systemPrompt = prompts.getSystemPrompt(llm.capabilities);
    const userPrompt = prompts.getUserPrompt(task.data);

    return {
      messages: [
        { role: 'system', content: systemPrompt },
        { role: 'user', content: userPrompt }
      ],
      temperature: task.parameters?.temperature || 0.7,
      maxTokens: this.calculateOptimalTokenLimit(task, llm),
      responseFormat: task.parameters?.responseFormat || 'text',
      estimatedTokens: this.estimateTokenUsage(systemPrompt + userPrompt)
    };
  }
}
```

4. Quality Control System

```
class QualityController {
  private validators: Map<string, Validator[]>;
  private consistencyChecker: ConsistencyChecker;
  private confidenceAnalyzer: ConfidenceAnalyzer;

  async validateResult(result: AnalysisResult, task: AnalysisTask): Promise<Qual-
ityAssessment> {
    const validations: ValidationResult[] = [];

    // Run task-specific validators
    const taskValidators = this.validators.get(task.type) || [];
    for (const validator of taskValidators) {
      const validation = await validator.validate(result, task);
      validations.push(validation);
    }

    // Check confidence levels
    const confidenceAssessment = this.confidenceAnalyzer.analyze(result);

    // Aggregate results
    const overallScore = this.calculateOverallQuality(validations, confidenceAssess-
ment);

    return {
      overallScore,
      isAcceptable: overallScore >= task.qualityThreshold,
      requiresFallback: overallScore < task.fallbackThreshold,
      requiresHumanReview: overallScore < task.humanReviewThreshold,
      validations,
      confidenceAssessment,
      recommendations: this.generateRecommendations(validations, confidenceAssessment)
    };
  }

  // Cross-model consistency validation
  async validateConsistency(results: AnalysisResult[]): Promise<ConsistencyReport> {
    if (results.length < 2) {
      return { score: 1.0, issues: [] };
    }

    const consistencyChecks = [
      this.checkContentSimilarity(results),
      this.checkKeyFactsAlignment(results),
      this.checkConclusionConsistency(results)
    ];

    const checks = await Promise.all(consistencyChecks);

    return {
      score: checks.reduce((sum, check) => sum + check.score, 0) / checks.length,
      issues: checks.flatMap(check => check.issues),
      recommendations: this.generateConsistencyRecommendations(checks)
    };
  }
}

// Specific validators for different task types
class PatentClaimValidator implements Validator {
  async validate(result: AnalysisResult, task: AnalysisTask):
Promise<ValidationResult> {
    const content = result.content;
```

```
    // Check for required elements in patent claim analysis
    const requiredElements = [
      'independent_claims',
      'dependent_claims',
      'novelty_assessment',
      'infringement_analysis'
    ];

    const missingElements = requiredElements.filter(element =>
      !this.hasElement(content, element)
    );

    const structureScore = (requiredElements.length - missingElements.length) / re-
quiredElements.length;

    // Check legal terminology accuracy
    const terminologyScore = await this.validateLegalTerminology(content);

    // Check citation accuracy
    const citationScore = await this.validateCitations(content);

    const overallScore = (structureScore + terminologyScore + citationScore) / 3;

    return {
      validatorName: 'PatentClaimValidator',
      score: overallScore,
      passed: overallScore >= 0.8,
      issues: this.identifyIssues(missingElements, terminologyScore, citationScore),
      recommendations: this.generateValidationRecommendations(overallScore)
    };
  }
}
```

5. Cost Management & Optimization

```
class CostOptimizer {
  private budgetManager: BudgetManager;
  private usageTracker: UsageTracker;
  private modelPricer: ModelPricer;

  async optimizeModelSelection(
    candidates: LLMService[],
    task: AnalysisTask,
    constraints: CostConstraints
  ): Promise<LLMService[]> {
    // Filter by absolute cost limits
    let filtered = candidates.filter(model => {
      const estimate = model.estimateCost(this.createEstimationRequest(task));
      return estimate.totalCost <= constraints.maxCostPerTask;
    });

    // Apply budget considerations
    if (constraints.monthlyBudget) {
      const currentUsage = await this.usageTracker.getCurrentMonthUsage();
      const remainingBudget = constraints.monthlyBudget - currentUsage.totalCost;

      filtered = filtered.filter(model => {
        const estimate = model.estimateCost(this.createEstimationRequest(task));
        return estimate.totalCost <= remainingBudget * constraints.budgetBufferRatio;
      });
    }

    // Sort by cost efficiency (quality per dollar)
    filtered.sort((a, b) => {
      const efficiencyA = this.calculateCostEfficiency(a, task);
      const efficiencyB = this.calculateCostEfficiency(b, task);
      return efficiencyB - efficiencyA;
    });

    return filtered;
  }

  private calculateCostEfficiency(model: LLMService, task: AnalysisTask): number {
    const costEstimate = model.estimateCost(this.createEstimationRequest(task));
    const qualityScore = model.capabilities.qualityScore;
    const taskFitScore = this.getTaskFitScore(model, task.type);

    const effectiveQuality = (qualityScore * taskFitScore) / 2;
    return effectiveQuality / (costEstimate.totalCost + 0.001); // Avoid division by
zero
  }

  async trackAndOptimize(execution: TaskExecution): Promise<OptimizationInsight> {
    // Track actual usage vs estimates
    await this.usageTracker.record({
      taskId: execution.taskId,
      modelId: execution.modelId,
      estimatedCost: execution.estimatedCost,
      actualCost: execution.actualCost,
      tokensUsed: execution.tokensUsed,
      qualityScore: execution.qualityScore,
      executionTime: execution.executionTime
    });

    // Generate optimization insights
    const insights = await this.generateInsights(execution);
```

```
    // Update model preferences based on performance
    await this.updateModelPreferences(execution);

    return insights;
  }
}

class BudgetManager {
  private budgets: Map<string, Budget>;
  private alerts: AlertManager;

  async checkBudgetConstraints(
    userId: string,
    estimatedCost: number,
    taskType: string
  ): Promise<BudgetCheckResult> {
    const userBudget = this.budgets.get(userId);
    if (!userBudget) {
      return { allowed: true, reason: 'No budget constraints' };
    }

    const currentUsage = await this.getCurrentUsage(userId);
    const projectedTotal = currentUsage.totalCost + estimatedCost;

    // Check various budget limits
    const checks = [
      this.checkMonthlyLimit(projectedTotal, userBudget.monthlyLimit),
      this.checkDailyLimit(currentUsage.dailyUsage + estimatedCost, user-
Budget.dailyLimit),
      this.checkPerTaskLimit(estimatedCost, userBudget.maxPerTask),
      this.checkTaskTypeLimit(currentUsage.byTaskType[taskType] + estimatedCost, user-
Budget.taskTypeLimits[taskType])
    ];

    const failedChecks = checks.filter(check => !check.passed);

    if (failedChecks.length > 0) {
      return {
        allowed: false,
        reason: failedChecks.map(check => check.reason).join('; '),
        suggestedAlternatives: this.suggestAlternatives(estimatedCost, taskType)
      };
    }

    // Check if approaching limits (send alerts)
    this.checkAndSendBudgetAlerts(userId, projectedTotal, userBudget);

    return { allowed: true };
  }
}
```

6. Caching & Performance Optimization

```
class LLMCacheManager {
  private promptCache: Redis;
  private resultCache: Redis;
  private semanticCache: VectorDatabase;

  async getCachedResult(
    prompt: string,
    modelId: string,
    parameters: TaskParameters
  ): Promise<AnalysisResult | null> {
    // Try exact match first (fastest)
    const exactKey = this.generateExactCacheKey(prompt, modelId, parameters);
    const exactMatch = await this.promptCache.get(exactKey);
    if (exactMatch) {
      return JSON.parse(exactMatch);
    }

    // Try semantic similarity match
    const semanticMatch = await this.findSemanticMatch(prompt, modelId, parameters);
    if (semanticMatch && semanticMatch.similarity > 0.95) {
      return semanticMatch.result;
    }

    return null;
  }

  async cacheResult(
    prompt: string,
    modelId: string,
    parameters: TaskParameters,
    result: AnalysisResult,
    ttl: number = 3600
  ): Promise<void> {
    // Cache exact match
    const exactKey = this.generateExactCacheKey(prompt, modelId, parameters);
    await this.promptCache.setex(exactKey, ttl, JSON.stringify(result));

    // Cache semantic embedding for similarity search
    const embedding = await this.generateEmbedding(prompt);
    await this.semanticCache.upsert({
      id: exactKey,
      vector: embedding,
      metadata: {
        modelId,
        parameters,
        result,
        timestamp: Date.now()
      }
    });

    // Update cache statistics
    await this.updateCacheStats(modelId, 'cache_write');
  }

  async findSemanticMatch(
    prompt: string,
    modelId: string,
    parameters: TaskParameters,
    threshold: number = 0.9
  ): Promise<SemanticMatch | null> {
    const embedding = await this.generateEmbedding(prompt);
```

```
    const matches = await this.semanticCache.query({
      vector: embedding,
      topK: 5,
      filter: { modelId },
      includeMetadata: true
    });

    const bestMatch = matches.find(match =>
      match.score >= threshold &&
      this.parametersMatch(parameters, match.metadata.parameters)
    );

    if (bestMatch) {
      await this.updateCacheStats(modelId, 'semantic_hit');
      return {
        similarity: bestMatch.score,
        result: bestMatch.metadata.result,
        originalPrompt: bestMatch.metadata.originalPrompt
      };
    }

    return null;
  }

  // Intelligent cache warming based on common patterns
  async warmCache(): Promise<void> {
    const commonPatterns = await this.analyzeCommonPatterns();

    for (const pattern of commonPatterns) {
      if (pattern.frequency > 10 && pattern.cacheHitRate < 0.5) {
        await this.precomputePattern(pattern);
      }
    }
  }
}
```

This comprehensive architecture provides a robust foundation for multi-LLM integration that optimizes for cost, quality, and performance while maintaining flexibility and scalability.