Multi-LLM Implementation Guide

Step-by-Step Integration for Deep Research Agent

🚀 Quick Start Implementation

Immediate Setup (Day 1)

1. Install Required Dependencies

```
# Core LLM client libraries
npm install openai anthropic @google-ai/generativelanguage cohere-ai

# Utility libraries for multi-LLM management
npm install lodash uuid crypto-js

# Caching and optimization
npm install ioredis node-cache

# Performance monitoring
npm install prom-client winston
```

2. Environment Variables Setup

Add to your `.env` file:

```
# Multi-LLM Configuration
ENABLE_MULTI_LLM=true
DEFAULT_LLM_STRATEGY=cost_optimized

# LLM API Keys
OPENAI_API_KEY=sk-your-openai-key
ANTHROPIC_API_KEY=your-anthropic-key
GOOGLE_AI_API_KEY=your-google-ai-key
COHERE_API_KEY=your-cohere-key

# Cost Management
MONTHLY_LLM_BUDGET=500
COST_OPTIMIZATION_MODE=balanced
ENABLE_BUDGET_ALERTS=true

# Performance Settings
ENABLE_LLM_CACHING=true
CACHE_TTL=3600
ENABLE_SEMANTIC_CACHING=true
MAX_CONCURRENT_LLM_CALLS=10
```

3. Basic LLM Service Implementation

```typescript
// src/lib/llm/base-service.ts
export abstract class BaseLLMService {
  abstract readonly id: string;
  abstract readonly provider: string;
  abstract readonly model: string;
  abstract readonly pricing: ModelPricing;

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

// src/lib/llm/openai-service.ts
import { OpenAI } from 'openai';
import { BaseLLMService } from './base-service';

export class OpenAIService extends BaseLLMService {
  readonly id = 'openai-gpt4-turbo';
  readonly provider = 'OpenAI';
  readonly model = 'gpt-4-turbo';
  readonly pricing = {
    inputTokenCost: 0.01,
    outputTokenCost: 0.03
  };

  private client: OpenAI;

  constructor(apiKey: string) {
    super();
    this.client = new OpenAI({ apiKey });
  }

  async generateCompletion(request: CompletionRequest): Promise<CompletionResponse> {
    try {
      const response = await this.client.chat.completions.create({
        model: this.model,
        messages: request.messages,
        temperature: request.temperature || 0.7,
        max_tokens: request.maxTokens || 2000
      });

      return {
        content: response.choices[0].message.content || '',
        usage: {
          inputTokens: response.usage?.prompt_tokens || 0,
          outputTokens: response.usage?.completion_tokens || 0,
          totalTokens: response.usage?.total_tokens || 0
        },
        cost: this.calculateCost(response.usage),
        confidence: this.estimateConfidence(response),
        modelId: this.id
      };
    } catch (error) {
      throw new LLMServiceError(`OpenAI API error: ${error.message}`);
    }
  }

  estimateCost(request: CompletionRequest): CostEstimate {
    const inputTokens = this.estimateInputTokens(request.messages);
    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)
    };
  }
}
```

Day 2-3: Router Implementation

4. Smart LLM Router

```
// src/lib/llm/router.ts
export class LLMRouter {
  private services: Map<string, BaseLLMService> = new Map();
  private taskPreferences: Map<string, ModelPreference[]> = new Map();

  constructor() {
    this.initializeServices();
    this.setupTaskPreferences();
  }

  private initializeServices() {
    // Initialize all LLM services
    if (process.env.OPENAI_API_KEY) {
      this.services.set('openai-gpt4', new OpenAIService(process.env.OPENAI_API_KEY));
      this.services.set('openai-gpt3.5', new OpenAIService(process.env.OPENAI_API_KEY,
'gpt-3.5-turbo'));
    }

    if (process.env.ANTHROPIC_API_KEY) {
      this.services.set('claude-opus', new AnthropicSer-
vice(process.env.ANTHROPIC_API_KEY, 'claude-3-opus'));
      this.services.set('claude-sonnet', new AnthropicSer-
vice(process.env.ANTHROPIC_API_KEY, 'claude-3-sonnet'));
    }

    if (process.env.GOOGLE_AI_API_KEY) {
      this.services.set('gemini-pro', new GoogleSer-
vice(process.env.GOOGLE_AI_API_KEY));
    }
  }

  private setupTaskPreferences() {
    this.taskPreferences.set('metadata_extraction', [
      { modelId: 'openai-gpt3.5', priority: 1, costWeight: 0.8 },
      { modelId: 'gemini-pro', priority: 2, costWeight: 0.9 },
      { modelId: 'claude-sonnet', priority: 3, qualityWeight: 0.7 }
    ]);

    this.taskPreferences.set('claim_analysis', [
      { modelId: 'openai-gpt4', priority: 1, qualityWeight: 0.9 },
      { modelId: 'claude-opus', priority: 2, qualityWeight: 0.95 },
      { modelId: 'claude-sonnet', priority: 3, qualityWeight: 0.8 }
    ]);

    this.taskPreferences.set('prior_art_analysis', [
      { modelId: 'claude-sonnet', priority: 1, contextWeight: 0.9 },
      { modelId: 'openai-gpt4', priority: 2, contextWeight: 0.8 },
      { modelId: 'claude-opus', priority: 3, contextWeight: 0.95 }
    ]);
  }

  async selectOptimalLLM(task: PatentTask, options: SelectionOptions = {}): Promise<Bas
eLLMService> {
    // Get task preferences
    const preferences = this.taskPreferences.get(task.type) || [];

    // Filter available models
    const availableModels = await this.getAvailableModels(preferences);

    // Apply budget constraints
    const budgetFiltered = this.filterByBudget(availableModels, options.maxCost);
```

```
    // Score and select best model
    const scored = await this.scoreModels(budgetFiltered, task, options);

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

    return scored[0].service;
  }

  private async scoreModels(
    services: BaseLLMService[],
    task: PatentTask,
    options: SelectionOptions
  ): Promise<ScoredService[]> {
    const scored = await Promise.all(
      services.map(async (service) => {
        const score = await this.calculateScore(service, task, options);
        return { service, score };
      })
    );

    return scored.sort((a, b) => b.score - a.score);
  }

  private async calculateScore(
    service: BaseLLMService,
    task: PatentTask,
    options: SelectionOptions
  ): Promise<number> {
    const preference = this.getTaskPreference(task.type, service.id);

    // Quality score (based on model capabilities and task fit)
    const qualityScore = this.getQualityScore(service, task) * (preference?.quality-
Weight || 0.5);

    // Cost efficiency score
    const costEstimate = service.estimateCost(this.createEstimationRequest(task));
    const costScore = this.calculateCostScore(costEstimate.totalCost) * (preference?.co
stWeight || 0.3);

    // Speed score (based on typical response times)
    const speedScore = this.getSpeedScore(service) * (preference?.speedWeight || 0.2);

    // Historical performance
    const historyScore = await this.getHistoricalScore(service.id, task.type);

    return (qualityScore + costScore + speedScore + historyScore) / 4;
  }
}
```

Day 4-5: Task Execution Engine

5. Multi-LLM Task Execution

```
// src/lib/llm/execution-engine.ts
export class TaskExecutionEngine {
  private router: LLMRouter;
  private cache: LLMCache;
  private costTracker: CostTracker;
  private qualityController: QualityController;

  async executeTask(task: PatentTask, options: ExecutionOptions = {}): Promise<Pat-
entAnalysisResult> {
    // Check cache first
    const cached = await this.cache.get(task);
    if (cached && !options.bypassCache) {
      return cached;
    }

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

    // Execute with fallback strategy
    const result = await this.executeWithFallback(task, llm, options);

    // Quality validation
    const validated = await this.qualityController.validate(result, task);

    if (validated.requiresImprovement) {
      // Try with higher quality model
      const premiumLLM = await this.router.selectOptimalLLM(task, {
        ...options,
        priorityQuality: true
      });

      if (premiumLLM.id !== llm.id) {
        const improvedResult = await this.executeWithLLM(task, premiumLLM);
        if (improvedResult.confidence > result.confidence) {
          result = improvedResult;
        }
      }
    }

    // Cache successful result
    await this.cache.set(task, result);

    // Track costs and performance
    await this.costTracker.record(result);

    return result;
  }

  private async executeWithFallback(
    task: PatentTask,
    primaryLLM: BaseLLMService,
    options: ExecutionOptions
  ): Promise<PatentAnalysisResult> {
    const maxAttempts = options.maxRetries || 3;
    let currentLLM = primaryLLM;

    for (let attempt = 0; attempt < maxAttempts; attempt++) {
      try {
        return await this.executeWithLLM(task, currentLLM);
      } catch (error) {
        console.warn(`LLM execution failed (attempt ${attempt + 1}):`, error.message);
```

```
        if (attempt < maxAttempts - 1) {
          // Get fallback model
          currentLLM = await this.getFallbackModel(currentLLM, task);
          await this.delay(Math.pow(2, attempt) * 1000); // Exponential backoff
        }
      }
    }

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

  private async executeWithLLM(task: PatentTask, llm: BaseLLMService): Promise<Pat-
entAnalysisResult> {
    const request = this.buildRequest(task, llm);
    const startTime = Date.now();

    const response = await llm.generateCompletion(request);

    const executionTime = Date.now() - startTime;

    return {
      taskId: task.id,
      taskType: task.type,
      modelUsed: llm.id,
      content: response.content,
      confidence: response.confidence,
      cost: response.cost,
      executionTime,
      usage: response.usage,
      timestamp: new Date()
    };
  }
}
```

Week 2: Advanced Features

6. Cost Management System

```
// src/lib/llm/cost-manager.ts
export class CostManager {
  private budgetLimits: BudgetConfiguration;
  private usageTracker: UsageTracker;
  private alertManager: AlertManager;

  async checkBudgetConstraints(estimatedCost: number, userId?: string): Promise<BudgetC
heckResult> {
    const currentUsage = await this.usageTracker.getCurrentUsage(userId);

    // Check monthly limit
    if (currentUsage.monthlyTotal + estimatedCost > this.budgetLimits.monthlyLimit) {
      return {
        allowed: false,
        reason: 'Monthly budget limit exceeded',
        suggestion: 'Use economy tier models or increase budget'
      };
    }

    // Check per-task limit
    if (estimatedCost > this.budgetLimits.maxPerTask) {
      return {
        allowed: false,
        reason: 'Per-task cost limit exceeded',
        suggestion: 'Break task into smaller parts or use lower-cost models'
      };
    }

    // Check if approaching limits (send alerts)
    const budgetUsedPercent = (currentUsage.monthlyTotal / this.budgetLim-
its.monthlyLimit) * 100;
    if (budgetUsedPercent > 80) {
      await this.alertManager.sendBudgetAlert(budgetUsedPercent, userId);
    }

    return { allowed: true };
  }

  async optimizeForBudget(task: PatentTask, maxCost: number): Promise<OptimizationSug-
gestion> {
    const models = await this.getAllAvailableModels();
    const suitable = models.filter(model => {
      const estimate = model.estimateCost(this.createEstimationRequest(task));
      return estimate.totalCost <= maxCost;
    });

    if (suitable.length === 0) {
      return {
        canOptimize: false,
        reason: 'No models available within budget constraint'
      };
    }

    // Find best quality within budget
    const bestModel = suitable.reduce((best, current) => {
      const bestQuality = this.getQualityScore(best, task);
      const currentQuality = this.getQualityScore(current, task);
      return currentQuality > bestQuality ? current : best;
    });

    return {
      canOptimize: true,
```

```
      suggestedModel: bestModel.id,
      estimatedCost: bestModel.estimateCost(this.createEstimationRe-
quest(task)).totalCost,
      qualityScore: this.getQualityScore(bestModel, task)
    };
  }
}
```

7. Quality Control System

```
// src/lib/llm/quality-controller.ts
export class QualityController {
  async validate(result: PatentAnalysisResult, task: PatentTask): Promise<QualityValid-
ation> {
    const validations = [];

    // Content structure validation
    validations.push(await this.validateStructure(result, task));

    // Domain-specific validation
    validations.push(await this.validateDomainRules(result, task));

    // Confidence threshold validation
    validations.push(this.validateConfidence(result));

    // Consistency validation (if multiple results available)
    if (task.previousResults) {
      validations.push(await this.validateConsistency(result, task.previousResults));
    }

    const overallScore = validations.reduce((sum, v) => sum + v.score, 0) / valida-
tions.length;

    return {
      overallScore,
      isAcceptable: overallScore >= task.qualityThreshold || 0.8,
      requiresImprovement: overallScore < 0.7,
      requiresHumanReview: overallScore < 0.6,
      validations,
      recommendations: this.generateRecommendations(validations)
    };
  }

  private async validateStructure(result: PatentAnalysisResult, task: PatentTask): Prom
ise<ValidationResult> {
    const requiredFields = this.getRequiredFields(task.type);
    const content = JSON.parse(result.content || '{}');

    const missingFields = requiredFields.filter(field => !content[field]);
    const score = (requiredFields.length - missingFields.length) / required-
Fields.length;

    return {
      type: 'structure',
      score,
      passed: score >= 0.9,
      issues: missingFields.map(field => `Missing required field: ${field}`),
      suggestions: missingFields.length > 0 ? ['Retry with more specific prompts'] : []
    };
  }
}
```

🔧 Integration with Existing Code

Updating Existing Agents

Modify Patent Analysis Agent

```
// src/agents/patent-analysis-agent.ts
export class PatentAnalysisAgent {
  private executionEngine: TaskExecutionEngine;

  constructor() {
    this.executionEngine = new TaskExecutionEngine();
  }

  async analyzePatent(patent: Patent, analysisType: AnalysisType): Promise<AnalysisRes-
ult> {
    const task: PatentTask = {
      id: generateId(),
      type: analysisType,
      data: patent,
      qualityThreshold: 0.8,
      maxCost: this.getMaxCostForTask(analysisType)
    };

    // Use multi-LLM execution engine instead of direct OpenAI call
    return await this.executionEngine.executeTask(task, {
      enableFallback: true,
      cacheResults: true,
      maxRetries: 3
    });
  }

  private getMaxCostForTask(analysisType: AnalysisType): number {
    const costLimits = {
      'metadata_extraction': 0.01,
      'claim_analysis': 0.05,
      'prior_art_analysis': 0.10,
      'competitive_intelligence': 0.15,
      'report_generation': 0.20
    };

    return costLimits[analysisType] || 0.05;
  }
}
```

Update API Routes

```
// src/pages/api/analyze.ts
import { TaskExecutionEngine } from '@/lib/llm/execution-engine';

export default async function handler(req: NextApiRequest, res: NextApiResponse) {
  const { patentData, analysisType, options = {} } = req.body;

  const executionEngine = new TaskExecutionEngine();

  try {
    const task = {
      id: generateId(),
      type: analysisType,
      data: patentData,
      qualityThreshold: options.qualityThreshold || 0.8
    };

    const result = await executionEngine.executeTask(task, {
      maxCost: options.maxCost,
      enableFallback: true,
      priorityQuality: options.priorityQuality
    });

    res.status(200).json({
      success: true,
      result,
      modelUsed: result.modelUsed,
      cost: result.cost,
      confidence: result.confidence
    });
  } catch (error) {
    res.status(500).json({
      success: false,
      error: error.message
    });
  }
}
```

📊 Monitoring & Analytics Dashboard

Real-time Performance Tracking

```tsx
// src/components/admin/LLMDashboard.tsx
export const LLMDashboard: React.FC = () => {
  const [metrics, setMetrics] = useState<LLMMetrics>();
  const [costs, setCosts] = useState<CostAnalytics>();

  useEffect(() => {
    // Real-time updates
    const interval = setInterval(async () => {
      const [metricsData, costData] = await Promise.all([
        fetch('/api/admin/llm-metrics').then(r => r.json()),
        fetch('/api/admin/cost-analytics').then(r => r.json())
      ]);

      setMetrics(metricsData);
      setCosts(costData);
    }, 30000); // Update every 30 seconds

    return () => clearInterval(interval);
  }, []);

  return (
    <div className="grid grid-cols-1 md:grid-cols-2 lg:grid-cols-4 gap-6">
      {/* Cost Overview */}
      <Card>
        <CardHeader>
          <CardTitle>Monthly Spend</CardTitle>
        </CardHeader>
        <CardContent>
          <div className="text-2xl font-bold">${costs?.monthlyTotal?.toFixed(2)}</div>
          <div className="text-sm text-gray-500">
            Budget: ${costs?.monthlyBudget?.toFixed(2)}
          </div>
          <Progress
            value={(costs?.monthlyTotal / costs?.monthlyBudget) * 100}
            className="mt-2"
          />
        </CardContent>
      </Card>

      {/* Model Performance */}
      <Card>
        <CardHeader>
          <CardTitle>Model Performance</CardTitle>
        </CardHeader>
        <CardContent>
          <div className="space-y-2">
            {metrics?.modelPerformance?.map(model => (
              <div key={model.id} className="flex justify-between">
                <span className="text-sm">{model.name}</span>
                <span className="text-sm font-medium">{model.successRate}%</span>
              </div>
            ))}
          </div>
        </CardContent>
      </Card>

      {/* Cost Savings */}
      <Card>
        <CardHeader>
          <CardTitle>Cost Savings</CardTitle>
        </CardHeader>
        <CardContent>
```

```
            <div className="text-2xl font-bold text-green-600">
              {costs?.savingsPercent}%
            </div>
            <div className="text-sm text-gray-500">
              vs Single Premium Model
            </div>
            <div className="text-lg font-semibold text-green-600 mt-2">
              ${costs?.totalSaved?.toFixed(2)} saved
            </div>
          </CardContent>
        </Card>

        {/* Quality Metrics */}
        <Card>
          <CardHeader>
            <CardTitle>Quality Score</CardTitle>
          </CardHeader>
          <CardContent>
            <div className="text-2xl font-bold">{metrics?.overallQuality}/10</div>
            <div className="text-sm text-gray-500">Average Quality</div>
            <div className="mt-2">
              <Badge variant={metrics?.qualityTrend === 'up' ? 'success' : 'warning'}>
                {metrics?.qualityTrend === 'up' ? '↗ Improving' : '→ Stable'}
              </Badge>
            </div>
          </CardContent>
        </Card>
      </div>
    );
};
```

🚀 Deployment Checklist

Pre-Production Validation

- [ ] All LLM services properly configured with API keys
- [ ] Cost tracking and budget alerts functional
- [ ] Quality validation rules implemented
- [ ] Fallback mechanisms tested
- [ ] Caching system operational
- [ ] Monitoring dashboard displaying real-time data
- [ ] Load testing completed with multiple LLMs
- [ ] Error handling and logging comprehensive

Production Deployment

```
# Build and deploy with multi-LLM support
npm run build
npm run test:llm-integration
npm run deploy:production

# Verify deployment
curl -X POST /api/test/multi-llm \
  -H "Content-Type: application/json" \
  -d '{"task": "test_analysis", "models": ["gpt-4", "claude-sonnet"]}'
```

Post-Deployment Monitoring

- [ ] Monitor cost trends vs projections
- [ ] Track quality scores across models
- [ ] Validate cost savings achieve 60%+ target
- [ ] Ensure 99.9%+ uptime with fallbacks
- [ ] Collect user feedback on analysis quality

📈 Expected Results

Week 1 Results

- **Basic multi-LLM routing functional**
- **30-40% cost reduction vs single premium model**
- **Improved availability through redundancy**

Week 2 Results

- **Advanced optimization features active**
- **50-65% cost reduction achieved**
- **Quality scores maintained or improved**
- **Real-time monitoring operational**

Month 1 Results

- **Fully optimized multi-LLM system**
- **65-70% cost reduction sustained**
- **95%+ quality scores across all tasks**
- **Sub-second cached response times**
- **Predictable monthly costs within ±5%**

This implementation guide provides a clear path to integrate multiple LLMs into the Deep Research Agent, achieving significant cost savings while improving quality and reliability through intelligent routing and validation systems.