Hybrid AI with LLM Fallback

Specification Document v1.0

Executive Summary

Hybrid AI with LLM Fallback is an architectural pattern that combines **template-based generation** (fast, deterministic) with **Large Language Model inference** (intelligent, adaptive) to deliver optimal response quality while maintaining performance and cost efficiency.

---

1. Problem Statement

Pure LLM Approach Limitations

Issue	Impact
High latency	2-15 seconds per request
API costs	$0.01-0.10+ per request
Rate limits	Throttling under load
Availability	External dependency failures
Inconsistency	Variable output quality

Pure Template Approach Limitations

Issue	Impact
Limited adaptability	Cannot handle novel scenarios
Maintenance burden	Manual updates for new cases
No context understanding	Keyword-only matching
Rigid responses	Cannot personalize or elaborate

---

2. Solution Architecture

2.1 Core Concept

```
┌─────────────────────────────────────────────────────────────────┐
│                        User Request                              │
└─────────────────────────────────────────────────────────────────┘
                               │
                               ▼
┌─────────────────────────────────────────────────────────────────┐
│                    Confidence Scoring                            │
│         (Analyze request complexity and template coverage)       │
└─────────────────────────────────────────────────────────────────┘
                               │
                   ┌───────────┴───────────┐
                   │                       │
         confidence ≥ 0.7        confidence < 0.7
                   │                       │
                   ▼                       ▼
┌───────────────────────────┐  ┌───────────────────────────┐
│   Template Generation     │  │      LLM Enhancement      │
│   (< 100ms response)      │  │    (2-15s response)       │
└───────────────────────────┘  └───────────────────────────┘
                   │                       │
                   └───────────┬───────────┘
                               │
                               ▼
┌─────────────────────────────────────────────────────────────────┐
│                      Response + Metadata                         │
│              (source, confidence, processingTime)                │
└─────────────────────────────────────────────────────────────────┘
```

2.2 Decision Flow

1. **Request Analysis** - Parse and categorize incoming request
2. **Confidence Calculation** - Score template coverage (0.0-1.0)
3. **Path Selection** - Route to template or LLM based on threshold
4. **Response Generation** - Execute selected path
5. **Metadata Attachment** - Include source and confidence in response

---

3. Confidence Scoring Algorithm

3.1 Scoring Factors

Factor	Weight	Description
Keyword Match	40%	Known terms found in request
Pattern Match	30%	Request matches known patterns
Context Completeness	20%	Required context data available
Request Complexity	10%	Length and structure analysis

3.2 Calculation Formula

```
confidence = (keywordScore × 0.4) +
             (patternScore × 0.3) +
             (contextScore × 0.2) +
             (complexityScore × 0.1)
```

3.3 Implementation Example

```
interface ConfidenceResult {
  score: number;          // 0.0 - 1.0
  factors: {
    keywordMatch: number;
    patternMatch: number;
    contextCompleteness: number;
    complexityScore: number;
  };
  matchedKeywords: string[];
  matchedPatterns: string[];
}

function calculateConfidence(
  request: string,
  context: Record<string, any>,
  knowledgeBase: KnowledgeBase
): ConfidenceResult {
  const normalizedRequest = request.toLowerCase().trim();

  // Keyword matching
  const matchedKeywords = knowledgeBase.keywords.filter(
    kw => normalizedRequest.includes(kw.toLowerCase())
  );
  const keywordScore = Math.min(matchedKeywords.length / 3, 1.0);

  // Pattern matching
  const matchedPatterns = knowledgeBase.patterns.filter(
    pattern => pattern.regex.test(normalizedRequest)
  );
  const patternScore = matchedPatterns.length > 0 ? 0.8 : 0.2;

  // Context completeness
  const requiredFields = knowledgeBase.requiredContext;
  const presentFields = requiredFields.filter(f => context[f] !== undefined);
  const contextScore = presentFields.length / requiredFields.length;

  // Complexity (inverse - simpler is higher confidence)
  const wordCount = normalizedRequest.split(/\s+/).length;
  const complexityScore = wordCount <= 20 ? 1.0 :
                          wordCount <= 50 ? 0.7 : 0.4;

  const score = (keywordScore * 0.4) +
                (patternScore * 0.3) +
                (contextScore * 0.2) +
                (complexityScore * 0.1);

  return {
    score: Math.min(score, 1.0),
    factors: {
      keywordMatch: keywordScore,
      patternMatch: patternScore,
      contextCompleteness: contextScore,
      complexityScore: complexityScore
    },
    matchedKeywords,
    matchedPatterns: matchedPatterns.map(p => p.name)
  };
}
```

4. Template-Based Generation

4.1 Template Structure

```
interface Template {
  id: string;
  name: string;
  patterns: RegExp[];           // Trigger patterns
  keywords: string[];           // Associated keywords
  requiredContext: string[];    // Required input fields
  generator: (context: any) => GeneratedResponse;
  priority: number;             // For conflict resolution
}

interface GeneratedResponse {
  content: any;                 // Primary response data
  confidence: number;           // Template's confidence
  suggestions?: string[];       // Follow-up suggestions
  references?: string[];        // Supporting documentation
}
```

4.2 Template Registry

```
class TemplateRegistry {
  private templates: Map<string, Template> = new Map();

  register(template: Template): void {
    this.templates.set(template.id, template);
  }

  findMatching(request: string, context: any): Template | null {
    const matches = Array.from(this.templates.values())
      .filter(t => this.matchesTemplate(request, context, t))
      .sort((a, b) => b.priority - a.priority);

    return matches[0] || null;
  }

  private matchesTemplate(
    request: string,
    context: any,
    template: Template
  ): boolean {
    // Check patterns
    const patternMatch = template.patterns.some(
      p => p.test(request.toLowerCase())
    );

    // Check required context
    const contextComplete = template.requiredContext.every(
      field => context[field] !== undefined
    );

    return patternMatch && contextComplete;
  }
}
```

4.3 Template Categories

Category	Use Case	Example
Lookup	Static information retrieval	"What is X?"
Calculation	Deterministic computations	"Calculate Y"
Status	System state queries	"Show current Z"
Recommendation	Rule-based suggestions	"Best practice for W"
Analysis	Pattern-based insights	"Analyze V metrics"

---

5. LLM Fallback Integration

5.1 When to Use LLM

Scenario	Confidence	Action
Known pattern, complete context	≥ 0.8	Template only
Partial match, good context	0.5 - 0.8	Template + LLM enhancement
Unknown pattern, any context	< 0.5	LLM only
Explicit depth request	Any	Force LLM

5.2 LLM Enhancement Strategies

Strategy 1: Augmentation

Use template output as LLM context for elaboration.

```
async function augmentWithLLM(
  templateResponse: GeneratedResponse,
  originalRequest: string,
  context: any
): Promise<EnhancedResponse> {
  const prompt = `
    Based on this data:
    ${JSON.stringify(templateResponse.content)}

    User asked: "${originalRequest}"

    Provide additional insights and recommendations.
  `;

  const llmResponse = await callLLM(prompt);

  return {
    ...templateResponse,
    insights: llmResponse.insights,
    elaboration: llmResponse.explanation,
    source: 'hybrid'
  };
}
```

Strategy 2: Validation

Use LLM to validate template response accuracy.

```
async function validateWithLLM(
  templateResponse: GeneratedResponse,
  context: any
): Promise<ValidatedResponse> {
  const prompt = `
    Validate this recommendation:
    ${JSON.stringify(templateResponse.content)}

    Given context:
    ${JSON.stringify(context)}

    Is this accurate? Any corrections needed?
  `;

  const validation = await callLLM(prompt);

  return {
    ...templateResponse,
    validated: validation.isAccurate,
    corrections: validation.corrections,
    source: 'validated'
  };
}
```

Strategy 3: Full Generation

Use LLM for complete response when no template matches.

```
async function generateWithLLM(
  request: string,
  context: any,
  systemPrompt: string
): Promise<LLMResponse> {
  const prompt = `
    ${systemPrompt}

    Context:
    ${JSON.stringify(context)}

    User request: "${request}"
  `;

  return await callLLM(prompt);
}
```

5.3 LLM Configuration

```
interface LLMConfig {
  provider: 'openai' | 'anthropic' | 'custom';
  model: string;
  temperature: number;        // 0.0-1.0, lower = deterministic
  maxTokens: number;
  timeout: number;            // milliseconds
  retries: number;
  fallbackResponse?: any;     // Return if LLM fails
}

const defaultConfig: LLMConfig = {
  provider: 'openai',
  model: 'gpt-4',
  temperature: 0.3,           // Low for consistency
  maxTokens: 2000,
  timeout: 30000,
  retries: 2,
  fallbackResponse: {
    error: 'Unable to process request',
    suggestion: 'Please try a more specific query'
  }
};
```

---

6. Response Metadata

6.1 Metadata Schema

```
interface ResponseMetadata {
  _meta: {
    source: 'template' | 'llm' | 'hybrid';
    confidence: number;
    processingTime: number;     // milliseconds
    templateId?: string;        // If template used
    llmModel?: string;          // If LLM used
    cacheHit?: boolean;         // If response cached
    factors?: {                 // Confidence breakdown
      keywordMatch: number;
      patternMatch: number;
      contextCompleteness: number;
    };
  };
}
```

6.2 Response Structure

```
interface HybridResponse<T> {
  data: T;                      // Primary response content
  _meta: ResponseMetadata;
}

// Example response
{
  data: {
    recommendations: [...],
    analysis: "..."
  },
  _meta: {
    source: "template",
    confidence: 0.85,
    processingTime: 45,
    templateId: "perf-recommendations-v2",
    factors: {
      keywordMatch: 0.9,
      patternMatch: 1.0,
      contextCompleteness: 0.8
    }
  }
}
```

7. Caching Strategy

7.1 Cache Layers

Layer	TTL	Use Case
Request Hash	5 min	Identical requests
Template Output	15 min	Same template + context
LLM Response	30 min	Expensive LLM calls
Context Data	1 min	Frequently accessed context

7.2 Cache Key Generation

```
function generateCacheKey(
  request: string,
  context: any,
  source: 'template' | 'llm'
): string {
  const normalizedRequest = request.toLowerCase().trim();
  const contextHash = hashObject(context);
  return `${source}:${hash(normalizedRequest)}:${contextHash}`;
}
```

7.3 Cache Invalidation

- **Time-based**: Automatic expiration via TTL
- **Event-based**: Invalidate on data changes
- **Manual**: Admin-triggered cache clear

8. Error Handling

8.1 Graceful Degradation

```
async function processRequest(request: string, context: any): Promise<Response> {
  const confidence = calculateConfidence(request, context);

  // Try template first (fast path)
  if (confidence.score >= TEMPLATE_THRESHOLD) {
    try {
      return await generateFromTemplate(request, context);
    } catch (templateError) {
      console.warn('Template failed, falling back to LLM');
      // Fall through to LLM
    }
  }

  // Try LLM (slow path)
  try {
    return await generateFromLLM(request, context);
  } catch (llmError) {
    console.error('LLM failed, using fallback');
    // Return graceful fallback
    return getFallbackResponse(request, confidence);
  }
}
```

8.2 Fallback Responses

```
function getFallbackResponse(
  request: string,
  confidence: ConfidenceResult
): Response {
  return {
    data: {
      message: 'Unable to fully process your request',
      partialMatch: confidence.matchedKeywords,
      suggestions: [
        'Try rephrasing your question',
        'Provide more specific details',
        'Check documentation for supported queries'
      ]
    },
    _meta: {
      source: 'fallback',
      confidence: 0,
      processingTime: 0,
      error: true
    }
  };
}
```

9. Performance Benchmarks

9.1 Target Metrics

Metric	Template	LLM	Hybrid Target
P50 Latency	< 50ms	3-5s	< 100ms
P99 Latency	< 200ms	10-15s	< 500ms
Cost/Request	$0	$0.01-0.10	< $0.002
Accuracy	85%	95%	92%
Availability	99.99%	99.5%	99.9%

9.2 Expected Distribution

```
Template Path: 80% of requests
Hybrid Path:   15% of requests
LLM-Only Path:  5% of requests
```

---

10. Monitoring & Observability

10.1 Key Metrics to Track

```
interface HybridMetrics {
  // Volume
  totalRequests: number;
  templateRequests: number;
  llmRequests: number;
  hybridRequests: number;

  // Performance
  avgTemplateLatency: number;
  avgLLMLatency: number;
  p99Latency: number;

  // Quality
  avgConfidence: number;
  lowConfidenceRate: number;    // % below threshold
  fallbackRate: number;         // % using fallback

  // Cost
  llmTokensUsed: number;
  estimatedCost: number;
}
```

10.2 Alerting Thresholds

Metric	Warning	Critical
LLM Latency	> 5s	> 15s
Fallback Rate	> 5%	> 15%
Low Confidence Rate	> 20%	> 40%
LLM Error Rate	> 1%	> 5%

---

11. Implementation Checklist

Phase 1: Foundation

- [ ] Define knowledge base structure
- [ ] Implement confidence scoring
- [ ] Create template registry
- [ ] Build basic templates (10-20)

Phase 2: Integration

- [ ] Integrate LLM provider
- [ ] Implement routing logic
- [ ] Add response metadata
- [ ] Set up caching layer

Phase 3: Optimization

- [ ] Tune confidence thresholds
- [ ] Expand template coverage
- [ ] Implement monitoring
- [ ] Add A/B testing capability

Phase 4: Maintenance

- [ ] Template update workflow
- [ ] Confidence threshold tuning
- [ ] Cost optimization
- [ ] Quality feedback loop

---

12. Best Practices

Do’s

- ✅ Start with high-frequency, well-defined queries as templates
- ✅ Include confidence metadata in all responses
- ✅ Cache aggressively for repeated queries

- ✅ Monitor template coverage and expand iteratively
- ✅ Use LLM to validate template accuracy periodically

Don’ts

- ❌ Don’t force template responses for ambiguous queries
- ❌ Don’t call LLM for every request “just in case”
- ❌ Don’t ignore low-confidence patterns (add templates)
- ❌ Don’t cache LLM responses without TTL limits
- ❌ Don’t expose raw LLM errors to users

---

13. Glossary

Term	Definition
Template	Pre-defined response generator for known patterns
Confidence Score	Numerical measure (0-1) of template applicability
Fallback	Graceful degradation when both paths fail
Hybrid Response	Response combining template + LLM output
Knowledge Base	Collection of keywords, patterns, and context definitions
LLM Enhancement	Using LLM to augment template-generated responses

---

14. References

- OpenAI Best Practices (https://platform.openai.com/docs/guides/gpt-best-practices)
- Anthropic Prompt Engineering (https://docs.anthropic.com/claude/docs/prompt-engineering)
- Caching Strategies for AI (https://www.microsoft.com/en-us/research/publication/semantic-caching-for-llm-queries/)

Document Version: 1.0
Last Updated: February 2026