Contents

SECTION 11: RADIANT BRAIN - SMART ROUTER (v2.4.0)

11.1 Brain Overview

RADIANT Brain is an intelligent request routing system that selects optimal models based on task analysis, cost constraints, latency requirements, and historical performance.

11.2 Brain Database Schema

```sql
-- migrations/021_radiant_brain.sql

CREATE TABLE brain_routing_rules (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    tenant_id UUID REFERENCES tenants(id),
    name VARCHAR(100) NOT NULL,
    priority INTEGER NOT NULL DEFAULT 100,
    conditions JSONB NOT NULL,
    target_model VARCHAR(100) NOT NULL,
    fallback_models TEXT[],
    is_active BOOLEAN DEFAULT true,
    created_at TIMESTAMPTZ NOT NULL DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE brain_routing_history (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    tenant_id UUID NOT NULL REFERENCES tenants(id),
    user_id UUID NOT NULL REFERENCES users(id),
    task_type VARCHAR(50),
    selected_model VARCHAR(100) NOT NULL,
    selection_reason TEXT,
    input_tokens INTEGER,
    output_tokens INTEGER,
    latency_ms INTEGER,
    cost DECIMAL(10, 6),
    success BOOLEAN,
    created_at TIMESTAMPTZ NOT NULL DEFAULT CURRENT_TIMESTAMP
);

CREATE INDEX idx_brain_history_tenant ON brain_routing_history(tenant_id);
CREATE INDEX idx_brain_history_model ON brain_routing_history(selected_model);

ALTER TABLE brain_routing_rules ENABLE ROW LEVEL SECURITY;
ALTER TABLE brain_routing_history ENABLE ROW LEVEL SECURITY;

CREATE POLICY brain_rules_isolation ON brain_routing_rules
    USING (tenant_id IS NULL OR tenant_id = current_setting('app.current_tenant_id')::UUID);
CREATE POLICY brain_history_isolation ON brain_routing_history
    USING (tenant_id = current_setting('app.current_tenant_id')::UUID);
```

11.3 Brain Router Service

```typescript
// packages/core/src/services/brain-router.ts

import { Pool } from 'pg';

interface RoutingContext {
    tenantId: string;
    userId: string;
    taskType: 'chat' | 'code' | 'analysis' | 'creative' | 'vision' | 'audio';
    inputTokenEstimate: number;
    maxLatencyMs?: number;
    maxCost?: number;
    preferredProvider?: string;
    requiresVision?: boolean;
    requiresAudio?: boolean;
}

interface RoutingResult {
    model: string;
    provider: string;
    reason: string;
    estimatedCost: number;
    estimatedLatencyMs: number;
    confidence: number;
}

export class BrainRouter {
    private pool: Pool;
    private modelPerformanceCache: Map<string, ModelPerformance> = new Map();

    constructor(pool: Pool) {
        this.pool = pool;
    }
```

```
async route(context: RoutingContext): Promise<RoutingResult> {
    // 1. Check tenant-specific rules first
    const customRule = await this.checkCustomRules(context);
    if (customRule) return customRule;

    // 2. Get available models for task type
    const candidates = await this.getCandidateModels(context);

    // 3. Score each candidate
    const scored = await Promise.all(
        candidates.map(async (model) => ({
            model,
            score: await this.scoreModel(model, context)
        }))
    );

    // 4. Sort by score and return best match
    scored.sort((a, b) => b.score.total - a.score.total);
    const best = scored[0];

    return {
        model: best.model.model_id,
        provider: best.model.provider,
        reason: this.formatReason(best.score),
        estimatedCost: best.score.estimatedCost,
        estimatedLatencyMs: best.score.estimatedLatency,
        confidence: best.score.total
    };
}

private async checkCustomRules(context: RoutingContext): Promise<RoutingResult | null> {
    const result = await this.pool.query(`
        SELECT * FROM brain_routing_rules
        WHERE (tenant_id IS NULL OR tenant_id = $1)
        AND is_active = true
        ORDER BY priority ASC
    `, [context.tenantId]);

    for (const rule of result.rows) {
        if (this.matchesConditions(rule.conditions, context)) {
            return {
                model: rule.target_model,
                provider: this.getProviderForModel(rule.target_model),
                reason: `Matched rule: ${rule.name}`,
                estimatedCost: 0,
                estimatedLatencyMs: 0,
                confidence: 1.0
            };
        }
    }

    return null;
}

private async getCandidateModels(context: RoutingContext) {
    const capabilities: string[] = [];
    if (context.requiresVision) capabilities.push('vision');
    if (context.requiresAudio) capabilities.push('audio');

    const capabilityFilter = capabilities.length > 0
        ? `AND capabilities @> $2::jsonb`
        : '';

    const result = await this.pool.query(`
        SELECT * FROM external_models
        WHERE is_active = true
        ${capabilityFilter}
        UNION ALL
        SELECT * FROM self_hosted_models
        WHERE is_active = true
        ${capabilityFilter}
    `, capabilities.length > 0 ? [context.tenantId, JSON.stringify(capabilities)] : [conte

    return result.rows;
}

private async scoreModel(model: any, context: RoutingContext): Promise<ModelScore> {
    const perf = await this.getModelPerformance(model.model_id);

    const costScore = this.scoreCost(model, context);
    const latencyScore = this.scoreLatency(perf, context);
    const qualityScore = this.scoreQuality(model, context);
    const reliabilityScore = perf.successRate;

    const estimatedCost = this.estimateCost(model, context.inputTokenEstimate);
    const estimatedLatency = perf.avgLatencyMs;

    return {
        costScore,
        latencyScore,
        qualityScore,
        reliabilityScore,
        estimatedCost,
        estimatedLatency,
        total: (costScore * 0.25) + (latencyScore * 0.25) + (qualityScore * 0.35) + (relia
    };
}

private scoreCost(model: any, context: RoutingContext): number {
    if (!context.maxCost) return 0.5;
    const estimated = this.estimateCost(model, context.inputTokenEstimate);
    if (estimated > context.maxCost) return 0;
    return 1 - (estimated / context.maxCost);
}

private scoreLatency(perf: ModelPerformance, context: RoutingContext): number {
    if (!context.maxLatencyMs) return 0.5;
    if (perf.avgLatencyMs > context.maxLatencyMs) return 0;
    return 1 - (perf.avgLatencyMs / context.maxLatencyMs);
}

private scoreQuality(model: any, context: RoutingContext): number {
    const taskQuality: Record<string, Record<string, number>> = {
        'code': { 'claude-sonnet-4': 0.95, 'gpt-4o': 0.9, 'grok-4': 0.85 },
        'creative': { 'claude-opus-4': 0.95, 'gpt-4o': 0.85, 'gemini-2': 0.8 },
        'analysis': { 'claude-opus-4': 0.95, 'o1': 0.95, 'gemini-2': 0.85 }
    };
    return taskQuality[context.taskType]?.[model.model_id] ?? 0.7;
}

private estimateCost(model: any, inputTokens: number): number {
    const outputEstimate = inputTokens * 1.5;
    return (inputTokens * model.input_cost_per_1k / 1000) +
           (outputEstimate * model.output_cost_per_1k / 1000);
}

private async getModelPerformance(modelId: string): Promise<ModelPerformance> {
    if (this.modelPerformanceCache.has(modelId)) {
        return this.modelPerformanceCache.get(modelId)!;
    }

    const result = await this.pool.query(`
        SELECT
            AVG(latency_ms) as avg_latency,
            COUNT(CASE WHEN success THEN 1 END)::float / COUNT(*)::float as success_rate
        FROM brain_routing_history
        WHERE selected_model = $1
        AND created_at > NOW() - INTERVAL '7 days'
    `, [modelId]);

    const perf = {
        avgLatencyMs: result.rows[0]?.avg_latency ?? 1000,
        successRate: result.rows[0]?.success_rate ?? 0.9
    };
    this.modelPerformanceCache.set(modelId, perf);
    return perf;
}

private formatReason(score: ModelScore): string {
    const factors: string[] = [];
    if (score.costScore > 0.8) factors.push('cost-effective');
    if (score.latencyScore > 0.8) factors.push('fast');
    if (score.qualityScore > 0.8) factors.push('high-quality');
    if (score.reliabilityScore > 0.95) factors.push('reliable');
    return factors.join(', ') || 'balanced choice';
}

private matchesConditions(conditions: any, context: RoutingContext): boolean {
    if (conditions.taskType && conditions.taskType !== context.taskType) return false;
    if (conditions.minTokens && context.inputTokenEstimate < conditions.minTokens) return
    if (conditions.maxTokens && context.inputTokenEstimate > conditions.maxTokens) return
    return true;
}

private getProviderForModel(modelId: string): string {
    const providerMap: Record<string, string> = {
        'claude-opus-4': 'anthropic',
        'claude-sonnet-4': 'anthropic',
        'gpt-4o': 'openai',
        'o1': 'openai',
        'gemini-2': 'google',
        'grok-4': 'xai'
    };
    return providerMap[modelId] ?? 'unknown';
}
}

interface ModelPerformance {
avgLatencyMs: number;
successRate: number;
}

interface ModelScore {
costScore: number;
latencyScore: number;
qualityScore: number;
reliabilityScore: number;
estimatedCost: number;
estimatedLatency: number;
total: number;
}
```