Contents

```
â• â• â• â• â• â• â• â• â• â• â• â• â• â• â• â• â• â• â• â• â• â• â• â• â• â• â• â• â• â
â• â• â• â• â• â• â• â• â• â• â• â• â• â• â• â• â• â• â• â• â• â• â• â• â• â• â• â• â• â
```

SECTION 4: LAMBDA FUNCTIONS - CORE (v2.1.0)

â• â

> **Dependencies:** Sections 0-3 **Creates:** Router, Chat, Models, Providers, PHI handlers

Type Imports

```
import { AIModel, AIProvider, ThermalState, PHIConfig, UsageEvent } from '@radiant/shared';
```

RADIANT v2.2.0 - Prompt 4: Lambda Functions - Core

> **Prompt 4 of 9** | Target Size: ~55KB | Version: 3.7.0 | December 2024

OVERVIEW

This prompt creates the core Lambda functions for the RADIANT API:

1. **Router Lambda** - Main API entry point, request routing, health checks
2. **Chat Lambda** - AI completions via LiteLLM, streaming support
3. **Models Lambda** - Dynamic model registry CRUD operations
4. **Providers Lambda** - Provider management and configuration
5. **PHI Lambda** - HIPAA-compliant PHI sanitization and re-identification

Plus shared utilities: - Database client (Aurora PostgreSQL via Data API) - LiteLLM client (HTTP client for AI routing) - Authentication helpers (JWT validation, tenant extraction) - Response helpers (standardized API responses) - Logging and tracing utilities

---

LAMBDA DIRECTORY STRUCTURE

```
packages/infrastructure/lambda/
Ã¢â€ Å"Ã¢â€ â‚¬Ã¢â€ â‚¬ tsconfig.json
Ã¢â€ Å"Ã¢â€ â‚¬Ã¢â€ â‚¬ package.json
Ã¢â€ Å"Ã¢â€ â‚¬Ã¢â€ â‚¬ shared/
Ã¢â€ â€š   Ã¢â€ Å"Ã¢â€ â‚¬Ã¢â€ â‚¬ index.ts
Ã¢â€ â€š   Ã¢â€ Å"Ã¢â€ â‚¬Ã¢â€ â‚¬ config.ts
Ã¢â€ â€š   Ã¢â€ Å"Ã¢â€ â‚¬Ã¢â€ â‚¬ logger.ts
Ã¢â€ â€š   Ã¢â€ Å"Ã¢â€ â‚¬Ã¢â€ â‚¬ errors.ts
Ã¢â€ â€š   Ã¢â€ Å"Ã¢â€ â‚¬Ã¢â€ â‚¬ response.ts
Ã¢â€ â€š   Ã¢â€ Å"Ã¢â€ â‚¬Ã¢â€ â‚¬ auth.ts
```

```
Ã¢â€ â€š   Ã¢â€ Å"Ã¢â€ â‚¬Ã¢â€ â‚¬ db/
Ã¢â€ â€š   Ã¢â€ â€š   Ã¢â€ Å"Ã¢â€ â‚¬Ã¢â€ â‚¬ index.ts
Ã¢â€ â€š   Ã¢â€ â€š   Ã¢â€ Å"Ã¢â€ â‚¬Ã¢â€ â‚¬ client.ts
Ã¢â€ â€š   Ã¢â€ â€š   Ã¢â€ Å"Ã¢â€ â‚¬Ã¢â€ â‚¬ queries.ts
Ã¢â€ â€š   Ã¢â€ â€š   Ã¢â€ â€ Ã¢â€ â‚¬Ã¢â€ â‚¬ types.ts
Ã¢â€ â€š   Ã¢â€ Å"Ã¢â€ â‚¬Ã¢â€ â‚¬ litellm/
Ã¢â€ â€š   Ã¢â€ â€š   Ã¢â€ Å"Ã¢â€ â‚¬Ã¢â€ â‚¬ index.ts
Ã¢â€ â€š   Ã¢â€ â€š   Ã¢â€ Å"Ã¢â€ â‚¬Ã¢â€ â‚¬ client.ts
Ã¢â€ â€š   Ã¢â€ â€š   Ã¢â€ â€ Ã¢â€ â‚¬Ã¢â€ â‚¬ types.ts
Ã¢â€ â€š   Ã¢â€ â€ Ã¢â€ â‚¬Ã¢â€ â‚¬ phi/
Ã¢â€ â€š       Ã¢â€ Å"Ã¢â€ â‚¬Ã¢â€ â‚¬ index.ts
Ã¢â€ â€š       Ã¢â€ Å"Ã¢â€ â‚¬Ã¢â€ â‚¬ sanitizer.ts
Ã¢â€ â€š       Ã¢â€ Å"Ã¢â€ â‚¬Ã¢â€ â‚¬ patterns.ts
Ã¢â€ â€š       Ã¢â€ â€ Ã¢â€ â‚¬Ã¢â€ â‚¬ types.ts
Ã¢â€ Å"Ã¢â€ â‚¬Ã¢â€ â‚¬ api/
Ã¢â€ â€š   Ã¢â€ Å"Ã¢â€ â‚¬Ã¢â€ â‚¬ router.ts
Ã¢â€ â€š   Ã¢â€ Å"Ã¢â€ â‚¬Ã¢â€ â‚¬ chat.ts
Ã¢â€ â€š   Ã¢â€ Å"Ã¢â€ â‚¬Ã¢â€ â‚¬ models.ts
Ã¢â€ â€š   Ã¢â€ â€ Ã¢â€ â‚¬Ã¢â€ â‚¬ providers.ts
Ã¢â€ â€ Ã¢â€ â‚¬Ã¢â€ â‚¬ admin/
    Ã¢â€ â€ Ã¢â€ â‚¬Ã¢â€ â‚¬ [See Prompt 5]
```

PART 1: LAMBDA CONFIGURATION

packages/infrastructure/lambda/package.json

```
{
  "name": "@radiant/lambda",
  "version": "2.2.0",
  "private": true,
  "scripts": {
    "build": "tsc",
    "clean": "rm -rf dist",
    "lint": "eslint . --ext .ts",
    "test": "jest"
  },
  "dependencies": {
    "@aws-sdk/client-dynamodb": "^3.470.0",
    "@aws-sdk/client-rds-data": "^3.470.0",
    "@aws-sdk/client-secrets-manager": "^3.470.0",
    "@aws-sdk/client-ssm": "^3.470.0",
    "@aws-sdk/client-s3": "^3.470.0",
    "@aws-sdk/client-sagemaker-runtime": "^3.470.0",
    "@aws-sdk/lib-dynamodb": "^3.470.0",
    "@aws-sdk/s3-request-presigner": "^3.470.0",
    "uuid": "^9.0.0",
    "zod": "^3.22.0"
  },
  "devDependencies": {
    "@types/aws-lambda": "^8.10.130",
    "@types/node": "^20.10.0",
    "@types/uuid": "^9.0.0",
    "typescript": "^5.3.0",
    "jest": "^29.7.0",
    "@types/jest": "^29.5.11",
    "ts-jest": "^29.1.1"
  }
}
```

packages/infrastructure/lambda/tsconfig.json

```
{
  "compilerOptions": {
    "target": "ES2022",
    "module": "NodeNext",
    "moduleResolution": "NodeNext",
    "lib": ["ES2022"],
    "outDir": "./dist",
    "rootDir": ".",
    "strict": true,
    "esModuleInterop": true,
    "skipLibCheck": true,
    "forceConsistentCasingInFileNames": true,
    "declaration": true,
    "declarationMap": true,
    "sourceMap": true,
    "resolveJsonModule": true,
    "noUnusedLocals": true,
    "noUnusedParameters": true,
    "noImplicitReturns": true,
    "noFallthroughCasesInSwitch": true
  },
  "include": ["**/*.ts"],
  "exclude": ["node_modules", "dist"]
}
```

PART 2: SHARED UTILITIES

packages/infrastructure/lambda/shared/index.ts

```
// Re-export all shared utilities
export * from './config';
export * from './logger';
export * from './errors';
export * from './response';
export * from './auth';
export * from './db';
export * from './litellm';
export * from './phi';
```

packages/infrastructure/lambda/shared/config.ts

```
/**
 * Environment configuration with validation
 */

import { z } from 'zod';

const envSchema = z.object({
  APP_ID: z.string().min(1),
  ENVIRONMENT: z.enum(['dev', 'staging', 'prod']),
  TIER: z.string().transform(Number).pipe(z.number().min(1).max(5)),
  LITELLM_URL: z.string().url(),
  AURORA_SECRET_ARN: z.string().startsWith('arn:aws:secretsmanager:'),
  AURORA_CLUSTER_ARN: z.string().startsWith('arn:aws:rds:'),
  USAGE_TABLE: z.string().min(1),
  SESSIONS_TABLE: z.string().min(1),
  CACHE_TABLE: z.string().min(1),
  MEDIA_BUCKET: z.string().min(1),
  USER_POOL_ID: z.string().min(1),
  LOG_LEVEL: z.enum(['debug', 'info', 'warn', 'error']).default('info'),
  AWS_REGION: z.string().default('us-east-1'),
});

export type Config = z.infer<typeof envSchema>;

let cachedConfig: Config | null = null;

export function getConfig(): Config {
  if (cachedConfig) return cachedConfig;

  const result = envSchema.safeParse(process.env);

  if (!result.success) {
    console.error('Configuration validation failed:', result.error.flatten());
    throw new Error(`Invalid configuration: ${JSON.stringify(result.error.flatten())}`);
  }

  cachedConfig = result.data;
  return cachedConfig;
}
```

```
/**
 * Feature flags based on tier
 */
export interface FeatureFlags {
  multiRegion: boolean;
  waf: boolean;
  guardDuty: boolean;
  sagemaker: boolean;
  elasticache: boolean;
  xray: boolean;
  phiSanitization: boolean;
  advancedMetrics: boolean;
}

export function getFeatureFlags(tier: number): FeatureFlags {
  return {
    multiRegion: tier >= 4,
    waf: tier >= 2,
    guardDuty: tier >= 2,
    sagemaker: tier >= 3,
    elasticache: tier >= 2,
    xray: tier >= 2,
    phiSanitization: true, // Always available
    advancedMetrics: tier >= 3,
  };
}
```

packages/infrastructure/lambda/shared/logger.ts

```
/**
 * Structured logging with correlation IDs
 */

export type LogLevel = 'debug' | 'info' | 'warn' | 'error';

interface LogContext {
  requestId?: string;
  tenantId?: string;
  userId?: string;
  appId?: string;
  environment?: string;
  [key: string]: unknown;
}

interface LogEntry {
  timestamp: string;
  level: LogLevel;
  message: string;
  context?: LogContext;
  error?: {
    name: string;
    message: string;
    stack?: string;
  };
  duration?: number;
  [key: string]: unknown;
}

const LOG_LEVELS: Record<LogLevel, number> = {
  debug: 0,
  info: 1,
  warn: 2,
  error: 3,
};

export class Logger {
  private context: LogContext;
  private minLevel: LogLevel;
  private startTime: number;

  constructor(context: LogContext = {}, minLevel?: LogLevel) {
    this.context = context;
    this.minLevel = minLevel || (process.env.LOG_LEVEL as LogLevel) || 'info';
    this.startTime = Date.now();
  }

  private shouldLog(level: LogLevel): boolean {
    return LOG_LEVELS[level] >= LOG_LEVELS[this.minLevel];
  }

  private formatEntry(level: LogLevel, message: string, extra?: Record<string, unknown>): LogEr
    return {
      timestamp: new Date().toISOString(),
      level,
      message,
      context: this.context,
      duration: Date.now() - this.startTime,
      ...extra,
    };
  }

  private log(level: LogLevel, message: string, extra?: Record<string, unknown>): void {
    if (!this.shouldLog(level)) return;

    const entry = this.formatEntry(level, message, extra);
    const output = JSON.stringify(entry);
    switch (level) {
      case 'error':
        console.error(output);
        break;
      case 'warn':
        console.warn(output);
        break;
      default:
        console.log(output);
    }
  }

  debug(message: string, extra?: Record<string, unknown>): void {
    this.log('debug', message, extra);
  }

  info(message: string, extra?: Record<string, unknown>): void {
    this.log('info', message, extra);
  }

  warn(message: string, extra?: Record<string, unknown>): void {
    this.log('warn', message, extra);
  }

  error(message: string, error?: Error, extra?: Record<string, unknown>): void {
    this.log('error', message, {
      ...extra,
      error: error ? {
        name: error.name,
        message: error.message,
        stack: error.stack,
      } : undefined,
    });
  }

  child(additionalContext: LogContext): Logger {
    return new Logger(
      { ...this.context, ...additionalContext },
      this.minLevel
    );
  }

  setRequestId(requestId: string): void {
    this.context.requestId = requestId;
  }

  setTenantId(tenantId: string): void {
    this.context.tenantId = tenantId;
  }

  setUserId(userId: string): void {
    this.context.userId = userId;
  }
}

// Default logger instance
export const logger = new Logger({
  appId: process.env.APP_ID,
  environment: process.env.ENVIRONMENT,
});
```

packages/infrastructure/lambda/shared/errors.ts

```
/**
 * Custom error types for API responses
 */

export abstract class AppError extends Error {
  abstract readonly statusCode: number;
  abstract readonly code: string;
  readonly isOperational = true;

  constructor(message: string) {
    super(message);
    this.name = this.constructor.name;
    Error.captureStackTrace(this, this.constructor);
  }

  toJSON() {
    return {
      code: this.code,
      message: this.message,
      statusCode: this.statusCode,
    };
  }
}

// 400 Bad Request
export class ValidationError extends AppError {
  readonly statusCode = 400;
  readonly code = 'VALIDATION_ERROR';
  readonly details?: Record<string, string[]>;

  constructor(message: string, details?: Record<string, string[]>) {
    super(message);
    this.details = details;
  }

  toJSON() {
    return {
      ...super.toJSON(),
      details: this.details,
    };
  }
}

// 401 Unauthorized
export class UnauthorizedError extends AppError {
  readonly statusCode = 401;
  readonly code = 'UNAUTHORIZED';

  constructor(message = 'Authentication required') {
    super(message);
  }
}

// 403 Forbidden
export class ForbiddenError extends AppError {
  readonly statusCode = 403;
  readonly code = 'FORBIDDEN';

  constructor(message = 'Access denied') {
    super(message);
  }
}

// 404 Not Found
export class NotFoundError extends AppError {
  readonly statusCode = 404;
  readonly code = 'NOT_FOUND';
  readonly resource?: string;

  constructor(resource?: string) {
    super(resource ? `${resource} not found` : 'Resource not found');
    this.resource = resource;
  }
}

// 409 Conflict
export class ConflictError extends AppError {
  readonly statusCode = 409;
  readonly code = 'CONFLICT';
```

```typescript
  constructor(message: string) {
    super(message);
  }
}

// 422 Unprocessable Entity
export class UnprocessableError extends AppError {
  readonly statusCode = 422;
  readonly code = 'UNPROCESSABLE_ENTITY';

  constructor(message: string) {
    super(message);
  }
}

// 429 Too Many Requests
export class RateLimitError extends AppError {
  readonly statusCode = 429;
  readonly code = 'RATE_LIMITED';
  readonly retryAfter?: number;

  constructor(retryAfter?: number) {
    super('Rate limit exceeded');
    this.retryAfter = retryAfter;
  }
}

// 500 Internal Server Error
export class InternalError extends AppError {
  readonly statusCode = 500;
  readonly code = 'INTERNAL_ERROR';

  constructor(message = 'An unexpected error occurred') {
    super(message);
  }
}

// 502 Bad Gateway (AI provider errors)
export class ProviderError extends AppError {
  readonly statusCode = 502;
  readonly code = 'PROVIDER_ERROR';
  readonly provider?: string;

  constructor(message: string, provider?: string) {
    super(message);
    this.provider = provider;
  }
}
```

```
// 503 Service Unavailable
export class ServiceUnavailableError extends AppError {
  readonly statusCode = 503;
  readonly code = 'SERVICE_UNAVAILABLE';

  constructor(message = 'Service temporarily unavailable') {
    super(message);
  }
}

/**
 * Check if error is an operational error (expected)
 */
export function isOperationalError(error: unknown): error is AppError {
  return error instanceof AppError && error.isOperational;
}

/**
 * Convert unknown error to AppError
 */
export function toAppError(error: unknown): AppError {
  if (error instanceof AppError) {
    return error;
  }

  if (error instanceof Error) {
    return new InternalError(error.message);
  }

  return new InternalError('Unknown error occurred');
}
```

packages/infrastructure/lambda/shared/response.ts

```
/**
 * Standardized API response helpers
 */

import type { APIGatewayProxyResult } from 'aws-lambda';
import { AppError, toAppError } from './errors';
import { Logger } from './logger';

interface SuccessResponse<T> {
  success: true;
  data: T;
  meta?: ResponseMeta;
}
```

```
interface ErrorResponse {
  success: false;
  error: {
    code: string;
    message: string;
    details?: unknown;
  };
}

interface ResponseMeta {
  requestId?: string;
  pagination?: {
    page: number;
    limit: number;
    total: number;
    hasMore: boolean;
  };
  timing?: {
    duration: number;
  };
}

type ApiResponse<T> = SuccessResponse<T> | ErrorResponse;

const DEFAULT_HEADERS = {
  'Content-Type': 'application/json',
  'Access-Control-Allow-Origin': '*',
  'Access-Control-Allow-Headers': 'Content-Type,Authorization,X-Api-Key,X-Tenant-Id',
  'Access-Control-Allow-Methods': 'GET,POST,PUT,DELETE,OPTIONS',
  'X-Content-Type-Options': 'nosniff',
  'X-Frame-Options': 'DENY',
  'Strict-Transport-Security': 'max-age=31536000; includeSubDomains',
};

/**
 * Create success response
 */
export function success<T>(
  data: T,
  statusCode = 200,
  meta?: ResponseMeta
): APIGatewayProxyResult {
  const response: SuccessResponse<T> = {
    success: true,
    data,
    meta,
  };
  return {
    statusCode,
    headers: DEFAULT_HEADERS,
    body: JSON.stringify(response),
  };
}

/**
 * Create created response (201)
 */
export function created<T>(data: T, meta?: ResponseMeta): APIGatewayProxyResult {
  return success(data, 201, meta);
}

/**
 * Create no content response (204)
 */
export function noContent(): APIGatewayProxyResult {
  return {
    statusCode: 204,
    headers: DEFAULT_HEADERS,
    body: '',
  };
}

/**
 * Create error response
 */
export function error(
  err: AppError,
  logger?: Logger
): APIGatewayProxyResult {
  if (logger) {
    if (err.statusCode >= 500) {
      logger.error('Internal error', err);
    } else {
      logger.warn('Client error', { error: err.toJSON() });
    }
  }

  const response: ErrorResponse = {
    success: false,
    error: {
      code: err.code,
      message: err.message,
      details: (err as any).details,
    },
  };

  const headers = { ...DEFAULT_HEADERS };

  if ('retryAfter' in err && err.retryAfter) {
    headers['Retry-After'] = String(err.retryAfter);
  }

  return {
    statusCode: err.statusCode,
    headers,
    body: JSON.stringify(response),
  };
}

/**
 * Handle errors uniformly
 */
export function handleError(
  err: unknown,
  logger?: Logger
): APIGatewayProxyResult {
  const appError = toAppError(err);
  return error(appError, logger);
}

/**
 * Create streaming response headers
 */
export function streamingHeaders(): Record<string, string> {
  return {
    ...DEFAULT_HEADERS,
    'Content-Type': 'text/event-stream',
    'Cache-Control': 'no-cache',
    Connection: 'keep-alive',
  };
}

/**
 * Format Server-Sent Event
 */
export function formatSSE(data: unknown, event?: string): string {
  let output = '';
  if (event) {
    output += `event: ${event}\n`;
  }
  output += `data: ${JSON.stringify(data)}\n\n`;
  return output;
```

```
}
```

packages/infrastructure/lambda/shared/auth.ts

```
/**
 * Authentication and authorization utilities
 */

import type { APIGatewayProxyEvent } from 'aws-lambda';
import { UnauthorizedError, ForbiddenError } from './errors';
import { Logger } from './logger';

/**
 * Enhanced AuthContext with app isolation (v4.6.0)
 */
export interface AuthContext {
  // Identity
  userId: string;       // Cognito sub
  appUserId: string;    // App-scoped user ID from app_users table
  tenantId: string;
  appId: string;        // Application identifier (thinktank, launchboard, etc.)
  email: string;

  // Roles & permissions
  roles: string[];
  groups: string[];
  isAdmin: boolean;
  isSuperAdmin: boolean;

  // Session
  sessionId?: string;
  tokenExpiry: number;
}

export interface TokenClaims {
  sub: string;
  email?: string;
  'cognito:username'?: string;
  'cognito:groups'?: string[];
  'custom:tenantId'?: string;
  'custom:tenant_id'?: string;
  'custom:appId'?: string;
  'custom:app_id'?: string;
  'custom:appUserId'?: string;
  'custom:app_user_id'?: string;
  'custom:role'?: string;
  iss: string;
  aud: string;
  exp: number;
  iat: number;
}

/**
 * Extract and validate authentication context from API Gateway event
 * Includes app isolation validation (v4.6.0)
 */
export function extractAuthContext(event: APIGatewayProxyEvent): AuthContext {
  const claims = event.requestContext.authorizer?.claims as TokenClaims | undefined;

  if (!claims) {
    throw new UnauthorizedError('No authentication claims found');
  }

  // Check token expiration
  if (claims.exp && claims.exp < Date.now() / 1000) {
    throw new UnauthorizedError('Token has expired');
  }

  // Extract core identifiers
  const userId = claims.sub;
  const tenantId = claims['custom:tenantId'] || claims['custom:tenant_id'];
  const appId = claims['custom:appId'] || claims['custom:app_id'];
  const appUserId = claims['custom:appUserId'] || claims['custom:app_user_id'];
  const email = claims.email || claims['cognito:username'] || '';

  // Validate required claims
  if (!userId) throw new UnauthorizedError('Missing user ID');
  if (!tenantId) throw new UnauthorizedError('Missing tenant ID');

  // App isolation - appId and appUserId required for v4.6.0+
  // Fallback for backward compatibility with pre-v4.6.0 tokens
  const resolvedAppId = appId || extractAppIdFromRoute(event) || 'default';
  const resolvedAppUserId = appUserId || userId; // Fallback to userId for legacy tokens

  // Validate app_id matches route (defense in depth)
  const routeAppId = extractAppIdFromRoute(event);
  if (routeAppId && appId && routeAppId !== appId) {
    throw new ForbiddenError(`Token app_id (${appId}) does not match route (${routeAppId})`);
  }

  // Extract roles and groups
  const groups = claims['cognito:groups'] || [];
  const roles = claims['custom:role'] ? [claims['custom:role']] : [];

  const isAdmin = groups.some(g =>
    ['super_admin', 'admin', 'operator', 'auditor'].includes(g)
  );
  const isSuperAdmin = groups.includes('super_admin');

  return {
    userId,
    appUserId: resolvedAppUserId,
    tenantId,
    appId: resolvedAppId,
    email,
    roles,
    groups,
    isAdmin,
    isSuperAdmin,
    tokenExpiry: claims.exp || 0,
  };
}

/**
 * Extract app_id from route for validation (v4.6.0)
 */
function extractAppIdFromRoute(event: APIGatewayProxyEvent): string | null {
  // Extract from subdomain: thinktank.domain.com -> thinktank
  const host = event.headers.Host || event.headers['host'];
  if (host) {
    const subdomain = host.split('.')[0];
    if (['thinktank', 'launchboard', 'alwaysme', 'mechanicalmaker'].includes(subdomain)) {
      return subdomain;
    }
  }

  // Extract from path: /api/thinktank/... -> thinktank
  const pathMatch = event.path.match(/^\/api\/(thinktank|launchboard|alwaysme|mechanicalmaker)/
  if (pathMatch) {
    return pathMatch[1];
  }

  return null;
}

/**
 * Require specific roles
 */
export function requireRoles(auth: AuthContext, requiredRoles: string[]): void {
  const hasRole = requiredRoles.some(role =>
    auth.roles.includes(role) || auth.groups.includes(role)
  );

  if (!hasRole) {
    throw new ForbiddenError(
      `Required roles: ${requiredRoles.join(', ')}`
    );
  }
}

/**
 * Require admin access
 */
export function requireAdmin(auth: AuthContext): void {
  if (!auth.isAdmin) {
    throw new ForbiddenError('Admin access required');
  }
}

/**
 * Require super admin access
 */
export function requireSuperAdmin(auth: AuthContext): void {
  if (!auth.groups.includes('super_admin')) {
    throw new ForbiddenError('Super admin access required');
  }
}

/**
 * Check if user can access tenant
 */
export function canAccessTenant(auth: AuthContext, tenantId: string): boolean {
  // Super admins can access any tenant
  if (auth.groups.includes('super_admin')) {
    return true;
  }

  // Users can only access their own tenant
  return auth.tenantId === tenantId;
}

/**
 * Require tenant access
 */
export function requireTenantAccess(auth: AuthContext, tenantId: string): void {
  if (!canAccessTenant(auth, tenantId)) {
    throw new ForbiddenError('Access to tenant denied');
  }
}

/**
 * Extract API key from header (for API key auth)
 */
export function extractApiKey(event: APIGatewayProxyEvent): string | undefined {
  return event.headers['X-Api-Key'] || event.headers['x-api-key'];
}

/**
 * Log authentication context (sanitized)
 */
export function logAuthContext(auth: AuthContext, logger: Logger): void {
  logger.info('Authenticated request', {
    userId: auth.userId,
    tenantId: auth.tenantId,
    isAdmin: auth.isAdmin,
    groupCount: auth.groups.length,
  });
}
```

PART 3: DATABASE CLIENT

packages/infrastructure/lambda/shared/db/index.ts

```
export * from './client';
export * from './queries';
export * from './types';
```

packages/infrastructure/lambda/shared/db/types.ts

```
/**
 * Database types matching Aurora PostgreSQL schema
 */

// =============================================================================
// PROVIDERS
// =============================================================================

export interface DBProvider {
  id: string;
  name: string;
  type: 'external' | 'self-hosted' | 'mid-tier';
  status: 'active' | 'inactive' | 'deprecated';
  hipaa_compliant: boolean;
  baa_available: boolean;
  base_url: string | null;
  auth_type: 'api_key' | 'oauth' | 'iam' | 'none';
  capabilities: string[];
  config: Record<string, unknown>;
  created_at: string;
  updated_at: string;
}

// ============================================================================
// MODELS
// ============================================================================

export interface DBModel {
  id: string;
  provider_id: string;
  name: string;
  display_name: string;
  description: string | null;
  type: 'external' | 'self-hosted';
  specialty: string;
  capabilities: string[];
  context_window: number | null;
  max_output_tokens: number | null;
  supports_functions: boolean;
  supports_vision: boolean;
  supports_streaming: boolean;
  has_thinking_mode: boolean;
  thinking_budget_tokens: number | null;
  pricing: DBModelPricing;
  thermal_state: string | null;
  thermal_config: DBThermalConfig | null;
  status: 'active' | 'inactive' | 'deprecated' | 'coming_soon';
  release_date: string | null;
  deprecation_date: string | null;
  created_at: string;
  updated_at: string;
}

export interface DBModelPricing {
  input_tokens?: number;
  output_tokens?: number;
  per_image?: number;
  per_minute_audio?: number;
  per_minute_video?: number;
  per_3d_model?: number;
  billed_markup: number;
}

export interface DBThermalConfig {
  state: 'OFF' | 'COLD' | 'WARM' | 'HOT' | 'AUTOMATIC';
  min_instances: number;
  max_instances: number;
  scale_to_zero_after_minutes?: number;
  warmup_time_seconds?: number;
}

// ============================================================================
// TENANTS
// ============================================================================

export interface DBTenant {
  id: string;
  app_id: string;
  name: string;
  domain: string | null;
  status: 'active' | 'suspended' | 'deleted';
  settings: DBTenantSettings;
  phi_config: DBPhiConfig | null;
  created_at: string;
  updated_at: string;
}

export interface DBTenantSettings {
  default_model?: string;
  allowed_providers?: string[];
  max_tokens_per_request?: number;
  rate_limit?: {
    requests_per_minute: number;
    tokens_per_day: number;
  };
}

export interface DBPhiConfig {
  mode: 'auto' | 'manual' | 'disabled';
  categories: Record<string, boolean>;
  reidentification: {
    allowed: boolean;
    requires_approval: boolean;
    mapping_ttl_hours: number;
  };
}

// ============================================================================
// USAGE
// ============================================================================

export interface DBUsageRecord {
  id: string;
  tenant_id: string;
  user_id: string | null;
  session_id: string | null;
  model_id: string;
  provider_id: string;
  input_tokens: number;
  output_tokens: number;
  total_tokens: number;
  cost: number;
  billed_amount: number;
  request_type: string;
  latency_ms: number;
  created_at: string;
}

// ===========================================================================
// AUDIT LOG
// ===========================================================================

export interface DBAuditLog {
  id: string;
  tenant_id: string;
  user_id: string | null;
  admin_id: string | null;
  action: string;
  resource_type: string;
  resource_id: string | null;
  details: Record<string, unknown>;
  ip_address: string | null;
  user_agent: string | null;
  created_at: string;
}
```

packages/infrastructure/lambda/shared/db/client.ts

```
/**
 * Aurora PostgreSQL client using Data API
 */

import {
  RDSDataClient,
  ExecuteStatementCommand,
  BatchExecuteStatementCommand,
  BeginTransactionCommand,
  CommitTransactionCommand,
  RollbackTransactionCommand,
  Field,
  SqlParameter,
} from '@aws-sdk/client-rds-data';
import {
  SecretsManagerClient,
  GetSecretValueCommand,
} from '@aws-sdk/client-secrets-manager';
import { getConfig } from '../config';
import { Logger } from '../logger';
import { InternalError } from '../errors';

// Initialize clients
const rdsClient = new RDSDataClient({});
const secretsClient = new SecretsManagerClient({});

// Cache database credentials
let dbCredentials: { username: string; password: string } | null = null;

export interface QueryResult<T = Record<string, unknown>> {
  rows: T[];
  rowCount: number;
  columnMetadata?: { name: string; type: string }[];
}

export interface TransactionContext {
  transactionId: string;
}

/**
 * Get database credentials from Secrets Manager
 */
async function getDbCredentials(): Promise<{ username: string; password: string }> {
  if (dbCredentials) return dbCredentials;

  const config = getConfig();

  try {
    const command = new GetSecretValueCommand({
      SecretId: config.AURORA_SECRET_ARN,
    });

    const response = await secretsClient.send(command);

    if (!response.SecretString) {
      throw new Error('Secret value is empty');
    }

    dbCredentials = JSON.parse(response.SecretString);
    return dbCredentials!;
  } catch (error) {
    throw new InternalError(`Failed to retrieve database credentials: ${error}`);
  }
}
```

```
/**
 * Convert JavaScript value to SQL parameter
 */
function toSqlParameter(name: string, value: unknown): SqlParameter {
  if (value === null || value === undefined) {
    return { name, value: { isNull: true } };
  }

  if (typeof value === 'string') {
    return { name, value: { stringValue: value } };
  }

  if (typeof value === 'number') {
    if (Number.isInteger(value)) {
      return { name, value: { longValue: value } };
    }
    return { name, value: { doubleValue: value } };
  }

  if (typeof value === 'boolean') {
    return { name, value: { booleanValue: value } };
  }

  if (Array.isArray(value)) {
    return { name, value: { stringValue: JSON.stringify(value) }, typeHint: 'JSON' };
  }

  if (typeof value === 'object') {
    return { name, value: { stringValue: JSON.stringify(value) }, typeHint: 'JSON' };
  }

  return { name, value: { stringValue: String(value) } };
}

/**
 * Convert SQL field to JavaScript value
 */
function fromSqlField(field: Field): unknown {
  if (field.isNull) return null;
  if (field.stringValue !== undefined) return field.stringValue;
  if (field.longValue !== undefined) return field.longValue;
  if (field.doubleValue !== undefined) return field.doubleValue;
  if (field.booleanValue !== undefined) return field.booleanValue;
  if (field.blobValue !== undefined) return field.blobValue;
  if (field.arrayValue !== undefined) {
    return field.arrayValue.stringValues ||
           field.arrayValue.longValues ||
           field.arrayValue.doubleValues ||
           field.arrayValue.booleanValues ||
           [];
  }
  return null;
}

/**
 * Execute a SQL query
 */
export async function query<T = Record<string, unknown>>(
  sql: string,
  params: Record<string, unknown> = {},
  logger?: Logger,
  transactionId?: string
): Promise<QueryResult<T>> {
  const config = getConfig();
  const startTime = Date.now();

  try {
    const command = new ExecuteStatementCommand({
      resourceArn: config.AURORA_CLUSTER_ARN,
      secretArn: config.AURORA_SECRET_ARN,
      database: 'radiant',
      sql,
      parameters: Object.entries(params).map(([name, value]) =>
        toSqlParameter(name, value)
      ),
      includeResultMetadata: true,
      transactionId,
    });

    const response = await rdsClient.send(command);
    const duration = Date.now() - startTime;

    if (logger) {
      logger.debug('Database query executed', {
        sql: sql.substring(0, 100),
        duration,
        rowCount: response.numberOfRecordsUpdated,
      });
    }

    // Convert response to rows
    const rows: T[] = [];

    if (response.records && response.columnMetadata) {
      for (const record of response.records) {
        const row: Record<string, unknown> = {};

        for (let i = 0; i < response.columnMetadata.length; i++) {
          const columnName = response.columnMetadata[i].name || `col${i}`;
          row[columnName] = fromSqlField(record[i]);
        }

        rows.push(row as T);
      }
    }

    return {
      rows,
      rowCount: response.numberOfRecordsUpdated || rows.length,
      columnMetadata: response.columnMetadata?.map(col => ({
        name: col.name || '',
        type: col.typeName || '',
      })),
    };
  } catch (error) {
    const duration = Date.now() - startTime;

    if (logger) {
      logger.error('Database query failed', error as Error, {
        sql: sql.substring(0, 100),
        duration,
      });
    }

    throw new InternalError(`Database query failed: ${error}`);
  }
}

/**
 * Execute a single row query
 */
export async function queryOne<T = Record<string, unknown>>(
  sql: string,
  params: Record<string, unknown> = {},
  logger?: Logger
): Promise<T | null> {
  const result = await query<T>(sql, params, logger);
  return result.rows[0] || null;
}

/**
 * Begin a transaction
 */
```

```
export async function beginTransaction(logger?: Logger): Promise<TransactionContext> {
  const config = getConfig();

  try {
    const command = new BeginTransactionCommand({
      resourceArn: config.AURORA_CLUSTER_ARN,
      secretArn: config.AURORA_SECRET_ARN,
      database: 'radiant',
    });

    const response = await rdsClient.send(command);

    if (!response.transactionId) {
      throw new Error('No transaction ID returned');
    }

    if (logger) {
      logger.debug('Transaction started', { transactionId: response.transactionId });
    }

    return { transactionId: response.transactionId };
  } catch (error) {
    throw new InternalError(`Failed to begin transaction: ${error}`);
  }
}

/**
 * Commit a transaction
 */
export async function commitTransaction(
  ctx: TransactionContext,
  logger?: Logger
): Promise<void> {
  const config = getConfig();

  try {
    const command = new CommitTransactionCommand({
      resourceArn: config.AURORA_CLUSTER_ARN,
      secretArn: config.AURORA_SECRET_ARN,
      transactionId: ctx.transactionId,
    });

    await rdsClient.send(command);

    if (logger) {
      logger.debug('Transaction committed', { transactionId: ctx.transactionId });
    }
  } catch (error) {
    throw new InternalError(`Failed to commit transaction: ${error}`);
  }
}

/**
 * Rollback a transaction
 */
export async function rollbackTransaction(
  ctx: TransactionContext,
  logger?: Logger
): Promise<void> {
  const config = getConfig();

  try {
    const command = new RollbackTransactionCommand({
      resourceArn: config.AURORA_CLUSTER_ARN,
      secretArn: config.AURORA_SECRET_ARN,
      transactionId: ctx.transactionId,
    });

    await rdsClient.send(command);

    if (logger) {
      logger.debug('Transaction rolled back', { transactionId: ctx.transactionId });
    }
  } catch (error) {
    // Log but don't throw - rollback errors are usually not critical
    if (logger) {
      logger.warn('Transaction rollback failed', {
        transactionId: ctx.transactionId,
        error: String(error),
      });
    }
  }
}

/**
 * Execute a callback within a transaction
 */
export async function withTransaction<T>(
  callback: (transactionId: string) => Promise<T>,
  logger?: Logger
): Promise<T> {
  const ctx = await beginTransaction(logger);

  try {
    const result = await callback(ctx.transactionId);
    await commitTransaction(ctx, logger);
    return result;
  } catch (error) {
    await rollbackTransaction(ctx, logger);
    throw error;
  }
}
```

packages/infrastructure/lambda/shared/db/queries.ts

```
/**
 * Pre-built database queries
 */

import { query, queryOne, QueryResult } from './client';
import { DBProvider, DBModel, DBTenant, DBUsageRecord, DBAuditLog } from './types';
import { Logger } from '../logger';
import { NotFoundError } from '../errors';

// ============================================================================
// PROVIDERS
// ============================================================================

export async function getProviders(
  filters: {
    type?: string;
    status?: string;
    hipaaCompliant?: boolean;
    limit?: number;
    offset?: number;
  } = {},
  logger?: Logger
): Promise<QueryResult<DBProvider>> {
  let sql = `
    SELECT * FROM providers
    WHERE 1=1
  `;
  const params: Record<string, unknown> = {};

  if (filters.type) {
    sql += ` AND type = :type`;
    params.type = filters.type;
  }

  if (filters.status) {
    sql += ` AND status = :status`;
    params.status = filters.status;
  }
  if (filters.hipaaCompliant !== undefined) {
    sql += ` AND hipaa_compliant = :hipaaCompliant`;
    params.hipaaCompliant = filters.hipaaCompliant;
  }

  sql += ` ORDER BY name ASC`;

  if (filters.limit) {
    sql += ` LIMIT :limit`;
    params.limit = filters.limit;
  }

  if (filters.offset) {
    sql += ` OFFSET :offset`;
    params.offset = filters.offset;
  }

  return query<DBProvider>(sql, params, logger);
}

export async function getProviderById(
  id: string,
  logger?: Logger
): Promise<DBProvider> {
  const result = await queryOne<DBProvider>(
    `SELECT * FROM providers WHERE id = :id`,
    { id },
    logger
  );

  if (!result) {
    throw new NotFoundError(`Provider ${id}`);
  }

  return result;
}

export async function updateProvider(
  id: string,
  updates: Partial<Pick<DBProvider, 'status' | 'hipaa_compliant' | 'config'>>,
  logger?: Logger
): Promise<DBProvider> {
  const setClauses: string[] = ['updated_at = NOW()'];
  const params: Record<string, unknown> = { id };

  if (updates.status !== undefined) {
    setClauses.push('status = :status');
    params.status = updates.status;
  }

  if (updates.hipaa_compliant !== undefined) {
    setClauses.push('hipaa_compliant = :hipaaCompliant');
    params.hipaaCompliant = updates.hipaa_compliant;
  }

  if (updates.config !== undefined) {
    setClauses.push('config = :config');
    params.config = updates.config;
  }

  const sql = `
    UPDATE providers
    SET ${setClauses.join(', ')}
    WHERE id = :id
    RETURNING *
  `;

  const result = await queryOne<DBProvider>(sql, params, logger);

  if (!result) {
    throw new NotFoundError(`Provider ${id}`);
  }

  return result;
}

// ============================================================================
// MODELS
// ============================================================================

export async function getModels(
  filters: {
    providerId?: string;
    specialty?: string;
    status?: string;
    type?: string;
    supportsVision?: boolean;
    supportsStreaming?: boolean;
    limit?: number;
    offset?: number;
  } = {},
  logger?: Logger
): Promise<QueryResult<DBModel>> {
  let sql = `
    SELECT * FROM models
    WHERE 1=1
  `;
  const params: Record<string, unknown> = {};

  if (filters.providerId) {
    sql += ` AND provider_id = :providerId`;
    params.providerId = filters.providerId;
  }

  if (filters.specialty) {
    sql += ` AND specialty = :specialty`;
    params.specialty = filters.specialty;
  }

  if (filters.status) {
    sql += ` AND status = :status`;
    params.status = filters.status;
  }

  if (filters.type) {
    sql += ` AND type = :type`;
    params.type = filters.type;
  }

  if (filters.supportsVision !== undefined) {
    sql += ` AND supports_vision = :supportsVision`;
    params.supportsVision = filters.supportsVision;
  }

  if (filters.supportsStreaming !== undefined) {
    sql += ` AND supports_streaming = :supportsStreaming`;
    params.supportsStreaming = filters.supportsStreaming;
  }

  sql += ` ORDER BY display_name ASC`;

  if (filters.limit) {
    sql += ` LIMIT :limit`;
    params.limit = filters.limit;
  }

  if (filters.offset) {
    sql += ` OFFSET :offset`;
    params.offset = filters.offset;
  }

  return query<DBModel>(sql, params, logger);
}
```

```
export async function getModelById(
  id: string,
  logger?: Logger
): Promise<DBModel> {
  const result = await queryOne<DBModel>(
    `SELECT * FROM models WHERE id = :id`,
    { id },
    logger
  );

  if (!result) {
    throw new NotFoundError(`Model ${id}`);
  }

  return result;
}

export async function getModelByName(
  name: string,
  logger?: Logger
): Promise<DBModel | null> {
  return queryOne<DBModel>(
    `SELECT * FROM models WHERE name = :name AND status = 'active'`,
    { name },
    logger
  );
}

export async function updateModel(
  id: string,
  updates: Partial<Pick<DBModel, 'status' | 'thermal_state' | 'thermal_config' | 'display_name
  logger?: Logger
): Promise<DBModel> {
  const setClauses: string[] = ['updated_at = NOW()'];
  const params: Record<string, unknown> = { id };

  if (updates.status !== undefined) {
    setClauses.push('status = :status');
    params.status = updates.status;
  }

  if (updates.thermal_state !== undefined) {
    setClauses.push('thermal_state = :thermalState');
    params.thermalState = updates.thermal_state;
  }

  if (updates.thermal_config !== undefined) {
    setClauses.push('thermal_config = :thermalConfig');
    params.thermalConfig = updates.thermal_config;
  }

  if (updates.display_name !== undefined) {
    setClauses.push('display_name = :displayName');
    params.displayName = updates.display_name;
  }

  if (updates.description !== undefined) {
    setClauses.push('description = :description');
    params.description = updates.description;
  }

  const sql = `
    UPDATE models
    SET ${setClauses.join(', ')}
    WHERE id = :id
    RETURNING *
  `;

  const result = await queryOne<DBModel>(sql, params, logger);

  if (!result) {
    throw new NotFoundError(`Model ${id}`);
  }

  return result;
}

// ============================================================================
// TENANTS
// ============================================================================

export async function getTenantById(
  id: string,
  logger?: Logger
): Promise<DBTenant> {
  const result = await queryOne<DBTenant>(
    `SELECT * FROM tenants WHERE id = :id AND status = 'active'`,
    { id },
    logger
  );

  if (!result) {
    throw new NotFoundError(`Tenant ${id}`);
  }

  return result;
```

```
}

export async function getTenantPhiConfig(
  tenantId: string,
  logger?: Logger
): Promise<DBTenant['phi_config']> {
  const tenant = await getTenantById(tenantId, logger);
  return tenant.phi_config;
}

// ============================================================================
// USAGE
// ============================================================================

export async function recordUsage(
  usage: Omit<DBUsageRecord, 'id' | 'created_at'>,
  logger?: Logger
): Promise<DBUsageRecord> {
  const sql = `
    INSERT INTO usage_records (
      id, tenant_id, user_id, session_id, model_id, provider_id,
      input_tokens, output_tokens, total_tokens, cost, billed_amount,
      request_type, latency_ms
    ) VALUES (
      gen_random_uuid(), :tenantId, :userId, :sessionId, :modelId, :providerId,
      :inputTokens, :outputTokens, :totalTokens, :cost, :billedAmount,
      :requestType, :latencyMs
    )
    RETURNING *
  `;

  const result = await queryOne<DBUsageRecord>(sql, {
    tenantId: usage.tenant_id,
    userId: usage.user_id,
    sessionId: usage.session_id,
    modelId: usage.model_id,
    providerId: usage.provider_id,
    inputTokens: usage.input_tokens,
    outputTokens: usage.output_tokens,
    totalTokens: usage.total_tokens,
    cost: usage.cost,
    billedAmount: usage.billed_amount,
    requestType: usage.request_type,
    latencyMs: usage.latency_ms,
  }, logger);

  return result!;
}
```

```
// ============================================================================
// AUDIT LOG
// ============================================================================

export async function createAuditLog(
  log: Omit<DBAuditLog, 'id' | 'created_at'>,
  logger?: Logger
): Promise<DBAuditLog> {
  const sql = `
    INSERT INTO audit_logs (
      id, tenant_id, user_id, admin_id, action, resource_type,
      resource_id, details, ip_address, user_agent
    ) VALUES (
      gen_random_uuid(), :tenantId, :userId, :adminId, :action, :resourceType,
      :resourceId, :details, :ipAddress, :userAgent
    )
    RETURNING *
  `;

  const result = await queryOne<DBAuditLog>(sql, {
    tenantId: log.tenant_id,
    userId: log.user_id,
    adminId: log.admin_id,
    action: log.action,
    resourceType: log.resource_type,
    resourceId: log.resource_id,
    details: log.details,
    ipAddress: log.ip_address,
    userAgent: log.user_agent,
  }, logger);

  return result!;
}
```

PART 4: LITELLM CLIENT

packages/infrastructure/lambda/shared/litellm/index.ts

```
export * from './client';
export * from './types';
```

packages/infrastructure/lambda/shared/litellm/types.ts

```
/**
 * LiteLLM API types
 */
```

```
// =============================================================================
// CHAT COMPLETION
// =============================================================================

export interface ChatCompletionRequest {
  model: string;
  messages: ChatMessage[];
  max_tokens?: number;
  temperature?: number;
  top_p?: number;
  stream?: boolean;
  stop?: string | string[];
  presence_penalty?: number;
  frequency_penalty?: number;
  user?: string;
  metadata?: Record<string, unknown>;
}

export interface ChatMessage {
  role: 'system' | 'user' | 'assistant' | 'function' | 'tool';
  content: string | ContentPart[];
  name?: string;
  function_call?: FunctionCall;
  tool_calls?: ToolCall[];
}

export interface ContentPart {
  type: 'text' | 'image_url';
  text?: string;
  image_url?: {
    url: string;
    detail?: 'low' | 'high' | 'auto';
  };
}

export interface FunctionCall {
  name: string;
  arguments: string;
}

export interface ToolCall {
  id: string;
  type: 'function';
  function: FunctionCall;
}

export interface ChatCompletionResponse {
  id: string;
  object: 'chat.completion';
  created: number;
  model: string;
  choices: ChatChoice[];
  usage: TokenUsage;
  system_fingerprint?: string;
}

export interface ChatChoice {
  index: number;
  message: ChatMessage;
  finish_reason: 'stop' | 'length' | 'function_call' | 'tool_calls' | 'content_filter' | null;
  logprobs?: unknown;
}

export interface TokenUsage {
  prompt_tokens: number;
  completion_tokens: number;
  total_tokens: number;
}

// ============================================================================
// STREAMING
// ============================================================================

export interface ChatCompletionChunk {
  id: string;
  object: 'chat.completion.chunk';
  created: number;
  model: string;
  choices: StreamChoice[];
  usage?: TokenUsage;
}

export interface StreamChoice {
  index: number;
  delta: {
    role?: string;
    content?: string;
    function_call?: Partial<FunctionCall>;
    tool_calls?: Partial<ToolCall>[];
  };
  finish_reason: string | null;
}

// ============================================================================
// EMBEDDINGS
// =============================================================================

export interface EmbeddingRequest {
  model: string;
  input: string | string[];
  encoding_format?: 'float' | 'base64';
  dimensions?: number;
  user?: string;
}

export interface EmbeddingResponse {
  object: 'list';
  data: EmbeddingData[];
  model: string;
  usage: {
    prompt_tokens: number;
    total_tokens: number;
  };
}

export interface EmbeddingData {
  object: 'embedding';
  index: number;
  embedding: number[];
}

// =============================================================================
// MODELS
// =============================================================================

export interface LiteLLMModel {
  id: string;
  object: 'model';
  created: number;
  owned_by: string;
}

export interface LiteLLMModelList {
  object: 'list';
  data: LiteLLMModel[];
}

// =============================================================================
// HEALTH
// =============================================================================

export interface HealthResponse {
  status: 'healthy' | 'unhealthy';
  version?: string;
  models?: string[];
}

// ============================================================================
// ERRORS
// ============================================================================

export interface LiteLLMError {
  error: {
    message: string;
    type: string;
    param?: string;
    code?: string;
  };
}
```

packages/infrastructure/lambda/shared/litellm/client.ts

```
/**
 * LiteLLM HTTP client
 */

import { getConfig } from '../config';
import { Logger } from '../logger';
import { ProviderError, ServiceUnavailableError, RateLimitError } from '../errors';
import {
  ChatCompletionRequest,
  ChatCompletionResponse,
  ChatCompletionChunk,
  EmbeddingRequest,
  EmbeddingResponse,
  LiteLLMModelList,
  HealthResponse,
  LiteLLMError,
} from './types';

const DEFAULT_TIMEOUT = 120000; // 2 minutes for AI requests

interface FetchOptions {
  method?: string;
  body?: unknown;
  timeout?: number;
  stream?: boolean;
}

/**
 * Make HTTP request to LiteLLM
 */
async function fetchLiteLLM<T>(
  path: string,
  options: FetchOptions = {},
  logger?: Logger
): Promise<T> {
  const config = getConfig();
  const url = `${config.LITELLM_URL}${path}`;
  const timeout = options.timeout || DEFAULT_TIMEOUT;

  const controller = new AbortController();
  const timeoutId = setTimeout(() => controller.abort(), timeout);

  try {
    const startTime = Date.now();

    const response = await fetch(url, {
      method: options.method || 'GET',
      headers: {
        'Content-Type': 'application/json',
        Accept: options.stream ? 'text/event-stream' : 'application/json',
      },
      body: options.body ? JSON.stringify(options.body) : undefined,
      signal: controller.signal,
    });

    const duration = Date.now() - startTime;

    if (logger) {
      logger.debug('LiteLLM request completed', {
        path,
        status: response.status,
        duration,
      });
    }

    if (!response.ok) {
      const errorBody = await response.text();
      let errorMessage = `LiteLLM error: ${response.status}`;

      try {
        const errorJson = JSON.parse(errorBody) as LiteLLMError;
        errorMessage = errorJson.error?.message || errorMessage;
      } catch {
        errorMessage = errorBody || errorMessage;
      }

      if (response.status === 429) {
        const retryAfter = parseInt(response.headers.get('Retry-After') || '60');
        throw new RateLimitError(retryAfter);
      }

      if (response.status >= 500) {
        throw new ServiceUnavailableError(errorMessage);
      }

      throw new ProviderError(errorMessage, 'litellm');
    }

    if (options.stream) {
      return response as unknown as T;
    }

    return await response.json() as T;
  } catch (error) {
    if (error instanceof Error && error.name === 'AbortError') {
      throw new ServiceUnavailableError('LiteLLM request timed out');
    }

    if (error instanceof ProviderError ||
        error instanceof ServiceUnavailableError ||
        error instanceof RateLimitError) {
      throw error;
    }

    throw new ServiceUnavailableError(`LiteLLM connection failed: ${error}`);
  } finally {
    clearTimeout(timeoutId);
  }
}

/**
 * Create chat completion
 */
export async function createChatCompletion(
  request: ChatCompletionRequest,
  logger?: Logger
): Promise<ChatCompletionResponse> {
  return fetchLiteLLM<ChatCompletionResponse>(
    '/v1/chat/completions',
    {
      method: 'POST',
      body: request,
    },
    logger
  );
```

```
}

/**
 * Create streaming chat completion
 */
export async function* streamChatCompletion(
  request: ChatCompletionRequest,
  logger?: Logger
): AsyncGenerator<ChatCompletionChunk> {
  const streamRequest = { ...request, stream: true };

  const response = await fetchLiteLLM<Response>(
    '/v1/chat/completions',
    {
      method: 'POST',
      body: streamRequest,
      stream: true,
    },
    logger
  );

  const reader = response.body?.getReader();
  if (!reader) {
    throw new ServiceUnavailableError('No response body for streaming');
  }

  const decoder = new TextDecoder();
  let buffer = '';

  try {
    while (true) {
      const { done, value } = await reader.read();

      if (done) break;

      buffer += decoder.decode(value, { stream: true });
      const lines = buffer.split('\n');
      buffer = lines.pop() || '';

      for (const line of lines) {
        const trimmed = line.trim();

        if (!trimmed || !trimmed.startsWith('data: ')) continue;

        const data = trimmed.slice(6);

        if (data === '[DONE]') {
          return;
```

```typescript
        }

        try {
          const chunk = JSON.parse(data) as ChatCompletionChunk;
          yield chunk;
        } catch {
          if (logger) {
            logger.warn('Failed to parse SSE chunk', { data });
          }
        }
      }
    }
  } finally {
    reader.releaseLock();
  }
}

/**
 * Create embeddings
 */
export async function createEmbedding(
  request: EmbeddingRequest,
  logger?: Logger
): Promise<EmbeddingResponse> {
  return fetchLiteLLM<EmbeddingResponse>(
    '/v1/embeddings',
    {
      method: 'POST',
      body: request,
    },
    logger
  );
}

/**
 * List available models
 */
export async function listModels(logger?: Logger): Promise<LiteLLMModelList> {
  return fetchLiteLLM<LiteLLMModelList>('/v1/models', {}, logger);
}

/**
 * Health check
 */
export async function checkHealth(logger?: Logger): Promise<HealthResponse> {
  try {
    const response = await fetchLiteLLM<HealthResponse>(
      '/health',
      { timeout: 5000 },
      logger
    );
    return response;
  } catch {
    return { status: 'unhealthy' };
  }
}

/**
 * Calculate cost for a completion
 */
export function calculateCost(
  usage: { prompt_tokens: number; completion_tokens: number },
  pricing: { input_tokens?: number; output_tokens?: number; billed_markup: number }
): { cost: number; billed: number } {
  const inputCost = (usage.prompt_tokens / 1_000_000) * (pricing.input_tokens || 0);
  const outputCost = (usage.completion_tokens / 1_000_000) * (pricing.output_tokens || 0);
  const cost = inputCost + outputCost;
  const billed = cost * (1 + pricing.billed_markup);

  return { cost, billed };
}
```

PART 5: PHI SANITIZATION

packages/infrastructure/lambda/shared/phi/index.ts

```
export * from './sanitizer';
export * from './patterns';
export * from './types';
```

packages/infrastructure/lambda/shared/phi/types.ts

```
/**
 * PHI (Protected Health Information) types
 */

export type PHICategory =
  | 'NAME'
  | 'SSN'
  | 'DOB'
  | 'ADDRESS'
  | 'PHONE'
  | 'EMAIL'
  | 'DIAGNOSIS'
  | 'TREATMENT'
  | 'MEDICAL_RECORD'
  | 'INSURANCE_ID';

export interface PHIConfig {
  mode: 'auto' | 'manual' | 'disabled';
  categories: Record<PHICategory, boolean>;
  reidentification: {
    allowed: boolean;
    requires_approval: boolean;
    mapping_ttl_hours: number;
  };
}

export interface PHIMatch {
  category: PHICategory;
  original: string;
  placeholder: string;
  startIndex: number;
  endIndex: number;
  confidence: number;
}

export interface SanitizationResult {
  sanitizedText: string;
  matches: PHIMatch[];
  mappingId: string;
}

export interface ReidentificationResult {
  originalText: string;
  matches: PHIMatch[];
}

export interface PHIMapping {
  id: string;
  tenant_id: string;
  session_id: string | null;
  mappings: Record<string, string>; // placeholder -> original
  created_at: string;
  expires_at: string;
}

export const DEFAULT_PHI_CONFIG: PHIConfig = {
  mode: 'auto',
  categories: {
    NAME: true,
    SSN: true,
    DOB: true,
    ADDRESS: true,
    PHONE: true,
    EMAIL: true,
    DIAGNOSIS: false, // Often needed for AI analysis
    TREATMENT: false, // Often needed for AI analysis
    MEDICAL_RECORD: true,
    INSURANCE_ID: true,
  },
  reidentification: {
    allowed: true,
    requires_approval: true,
    mapping_ttl_hours: 24,
  },
};
```

packages/infrastructure/lambda/shared/phi/patterns.ts

```
/**
 * PHI detection patterns
 */

import { PHICategory } from './types';

export interface PHIPattern {
  category: PHICategory;
  patterns: RegExp[];
  validator?: (match: string) => boolean;
  confidence: number;
}

/**
 * PHI detection patterns by category
 */
export const PHI_PATTERNS: PHIPattern[] = [
  // Social Security Numbers
  {
    category: 'SSN',
    patterns: [
      /\b\d{3}-\d{2}-\d{4}\b/g,
      /\b\d{3}\s\d{2}\s\d{4}\b/g,
      /\bSSN[:\s]*\d{3}[-\s]?\d{2}[-\s]?\d{4}\b/gi,
    ],
    validator: (match) => {
      const digits = match.replace(/\D/g, '');
      if (digits.length !== 9) return false;
      // Invalid SSN patterns
      if (digits.startsWith('000') || digits.startsWith('666')) return false;
      if (digits.substring(0, 3) === '900' && parseInt(digits.substring(0, 3)) <= 999) return f
```

```
        return true;
      },
      confidence: 0.95,
    },

    // Dates of Birth
    {
      category: 'DOB',
      patterns: [
        /\b(?:DOB|Date of Birth|Birthday|Born)[:\s]*(\d{1,2}[-\/]\d{1,2}[-\/]\d{2,4})\b/gi,
        /\b(?:DOB|Date of Birth|Birthday|Born)[:\s]*([A-Z][a-z]+\s+\d{1,2},?\s+\d{4})\b/gi,
      ],
      confidence: 0.90,
    },

    // Phone Numbers
    {
      category: 'PHONE',
      patterns: [
        /\b\(?[2-9]\d{2}\)?[-.\s]?\d{3}[-.\s]?\d{4}\b/g,
        /\b(?:Phone|Tel|Mobile|Cell)[:\s]*\(?[2-9]\d{2}\)?[-.\s]?\d{3}[-.\s]?\d{4}\b/gi,
        /\b\+1[-.\s]?\(?[2-9]\d{2}\)?[-.\s]?\d{3}[-.\s]?\d{4}\b/g,
      ],
      confidence: 0.85,
    },

    // Email Addresses
    {
      category: 'EMAIL',
      patterns: [
        /\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}\b/g,
      ],
      validator: (match) => {
        // Exclude common system emails
        const systemDomains = ['YOUR_DOMAIN.com', 'test.com', 'localhost'];
        const domain = match.split('@')[1]?.toLowerCase();
        return !systemDomains.includes(domain);
      },
      confidence: 0.90,
    },

    // Medical Record Numbers
    {
      category: 'MEDICAL_RECORD',
      patterns: [
        /\b(?:MRN|Medical Record|Patient ID)[:\s#]*([A-Z0-9]{6,12})\b/gi,
        /\bMRN[:\s#]*\d{6,12}\b/gi,
      ],
      confidence: 0.85,
    },

    // Insurance IDs
    {
      category: 'INSURANCE_ID',
      patterns: [
        /\b(?:Insurance ID|Policy Number|Member ID)[:\s#]*([A-Z0-9]{8,15})\b/gi,
        /\b(?:Group|Plan)\s*(?:Number|ID|#)[:\s]*([A-Z0-9]{6,12})\b/gi,
      ],
      confidence: 0.80,
    },

    // Street Addresses
    {
      category: 'ADDRESS',
      patterns: [
        /\b\d{1,5}\s+[A-Za-z]+(?:\s+[A-Za-z]+)*\s+(?:Street|St|Avenue|Ave|Road|Rd|Boulevard|Blvd
        /\b(?:P\.?O\.?\s*Box|PO Box)\s+\d+\b/gi,
      ],
      confidence: 0.75,
    },

    // Names (context-dependent)
    {
      category: 'NAME',
      patterns: [
        /\b(?:Patient|Client|Name)[:\s]*([A-Z][a-z]+(?:\s+[A-Z][a-z]+)+)\b/gi,
        /\b(?:Dr\.|Doctor|Mr\.|Mrs\.|Ms\.)\s+([A-Z][a-z]+(?:\s+[A-Z][a-z]+)*)\b/g,
      ],
      confidence: 0.70,
    },

    // Diagnoses (ICD codes and common conditions)
    {
      category: 'DIAGNOSIS',
      patterns: [
        /\b(?:ICD-?10)[:\s]*([A-Z]\d{2}(?:\.\d{1,4})?)\b/gi,
        /\b(?:Diagnosis|Dx)[:\s]*([A-Za-z\s]+(?:syndrome|disease|disorder|condition))\b/gi,
      ],
      confidence: 0.80,
    },

    // Treatments
    {
      category: 'TREATMENT',
      patterns: [
        /\b(?:Rx|Prescription|Medication)[:\s]*([A-Za-z]+(?:\s+\d+\s*mg)?)\b/gi,
        /\b(?:Treatment|Procedure)[:\s]*([A-Za-z\s]+(?:ectomy|plasty|scopy|therapy))\b/gi,
      ],
      confidence: 0.75,
    },
  ];

  /**
   * Get patterns for enabled categories
   */
  export function getEnabledPatterns(
    categories: Record<PHICategory, boolean>
  ): PHIPattern[] {
    return PHI_PATTERNS.filter(pattern => categories[pattern.category]);
  }
```

packages/infrastructure/lambda/shared/phi/sanitizer.ts

```
/**
 * PHI sanitization engine
 */

import { v4 as uuid } from 'uuid';
import {
  PHIConfig,
  PHICategory,
  PHIMatch,
  SanitizationResult,
  ReidentificationResult,
  PHIMapping,
  DEFAULT_PHI_CONFIG,
} from './types';
import { getEnabledPatterns, PHIPattern } from './patterns';
import { Logger } from '../logger';

// Placeholder format: [PHI_CATEGORY_INDEX]
const PLACEHOLDER_PREFIX = '[PHI_';
const PLACEHOLDER_SUFFIX = ']';

/**
 * Generate a placeholder for a PHI match
 */
function generatePlaceholder(category: PHICategory, index: number): string {
  return `${PLACEHOLDER_PREFIX}${category}_${index}${PLACEHOLDER_SUFFIX}`;
}

/**
 * Parse a placeholder back to its parts
 */
```

```
function parsePlaceholder(placeholder: string): { category: PHICategory; index: number } | nul
  const match = placeholder.match(/\[PHI_([A-Z_]+)_(\d+)\]/);
  if (!match) return null;
  return {
    category: match[1] as PHICategory,
    index: parseInt(match[2]),
  };
}

/**
 * Detect PHI in text
 */
export function detectPHI(
  text: string,
  config: PHIConfig = DEFAULT_PHI_CONFIG,
  logger?: Logger
): PHIMatch[] {
  if (config.mode === 'disabled') {
    return [];
  }

  const matches: PHIMatch[] = [];
  const patterns = getEnabledPatterns(config.categories);
  const categoryCounters: Record<string, number> = {};

  for (const patternDef of patterns) {
    for (const regex of patternDef.patterns) {
      // Reset regex lastIndex
      regex.lastIndex = 0;
      let match: RegExpExecArray | null;

      while ((match = regex.exec(text)) !== null) {
        const original = match[1] || match[0];

        // Skip if validator fails
        if (patternDef.validator && !patternDef.validator(original)) {
          continue;
        }

        // Check for overlapping matches
        const startIndex = text.indexOf(original, match.index);
        const endIndex = startIndex + original.length;

        const isOverlapping = matches.some(
          m => (startIndex >= m.startIndex && startIndex < m.endIndex) ||
               (endIndex > m.startIndex && endIndex <= m.endIndex)
        );
        if (isOverlapping) continue;

        // Generate unique placeholder
        categoryCounters[patternDef.category] = (categoryCounters[patternDef.category] || 0) +
        const placeholder = generatePlaceholder(
          patternDef.category,
          categoryCounters[patternDef.category]
        );

        matches.push({
          category: patternDef.category,
          original,
          placeholder,
          startIndex,
          endIndex,
          confidence: patternDef.confidence,
        });
      }
    }
  }

  // Sort by start index (descending) for safe replacement
  matches.sort((a, b) => b.startIndex - a.startIndex);

  if (logger && matches.length > 0) {
    logger.info('PHI detected', {
      matchCount: matches.length,
      categories: [...new Set(matches.map(m => m.category))],
    });
  }

  return matches;
}

/**
 * Sanitize PHI in text
 */
export function sanitizePHI(
  text: string,
  config: PHIConfig = DEFAULT_PHI_CONFIG,
  logger?: Logger
): SanitizationResult {
  const matches = detectPHI(text, config, logger);

  if (matches.length === 0) {
    return {
      sanitizedText: text,
      matches: [],
      mappingId: '',
    };
  }

  let sanitizedText = text;

  // Replace in reverse order to preserve indices
  for (const match of matches) {
    sanitizedText =
      sanitizedText.substring(0, match.startIndex) +
      match.placeholder +
      sanitizedText.substring(match.endIndex);
  }

  // Generate mapping ID for re-identification
  const mappingId = uuid();

  return {
    sanitizedText,
    matches: matches.reverse(), // Return in original order
    mappingId,
  };
}

/**
 * Re-identify PHI in text
 */
export function reidentifyPHI(
  sanitizedText: string,
  mapping: PHIMapping,
  logger?: Logger
): ReidentificationResult {
  let originalText = sanitizedText;
  const matches: PHIMatch[] = [];

  // Find all placeholders in the text
  const placeholderRegex = /\[PHI_[A-Z_]+_\d+\]/g;
  let match: RegExpExecArray | null;

  while ((match = placeholderRegex.exec(sanitizedText)) !== null) {
    const placeholder = match[0];
    const original = mapping.mappings[placeholder];

    if (original) {
      const parsed = parsePlaceholder(placeholder);

      if (parsed) {
        matches.push({
          category: parsed.category,
          original,
          placeholder,
          startIndex: match.index,
          endIndex: match.index + placeholder.length,
          confidence: 1.0,
        });
      }
    }
  }

  // Replace in reverse order
  matches.sort((a, b) => b.startIndex - a.startIndex);

  for (const m of matches) {
    originalText =
      originalText.substring(0, m.startIndex) +
      m.original +
      originalText.substring(m.endIndex);
  }

  if (logger) {
    logger.info('PHI re-identified', {
      matchCount: matches.length,
    });
  }

  return {
    originalText,
    matches: matches.reverse(),
  };
}

/**
 * Create a mapping record for storage
 */
export function createMappingRecord(
  tenantId: string,
  sessionId: string | null,
  result: SanitizationResult,
  ttlHours: number
): PHIMapping {
  const mappings: Record<string, string> = {};

  for (const match of result.matches) {
    mappings[match.placeholder] = match.original;
  }
  const now = new Date();
  const expiresAt = new Date(now.getTime() + ttlHours * 60 * 60 * 1000);

  return {
    id: result.mappingId,
    tenant_id: tenantId,
    session_id: sessionId,
    mappings,
    created_at: now.toISOString(),
    expires_at: expiresAt.toISOString(),
  };
}

/**
 * Check if text contains any PHI
 */
export function containsPHI(
  text: string,
  config: PHIConfig = DEFAULT_PHI_CONFIG
): boolean {
  const matches = detectPHI(text, config);
  return matches.length > 0;
}

/**
 * Get PHI summary for audit logging
 */
export function getPHISummary(
  matches: PHIMatch[]
): Record<PHICategory, number> {
  const summary: Record<string, number> = {};

  for (const match of matches) {
    summary[match.category] = (summary[match.category] || 0) + 1;
  }

  return summary as Record<PHICategory, number>;
}
```

PART 6: ROUTER LAMBDA

packages/infrastructure/lambda/api/router.ts

```
/**
 * Router Lambda - Main API entry point
 *
 * Handles:
```

```
 * - Health checks
 * - Request routing
 * - CORS preflight
 * - Error handling
 */

import type {
  APIGatewayProxyEvent,
  APIGatewayProxyResult,
  Context,
} from 'aws-lambda';
import { Logger } from '../shared/logger';
import { getConfig } from '../shared/config';
import { success, handleError } from '../shared/response';
import { NotFoundError } from '../shared/errors';
import { checkHealth as checkLiteLLMHealth } from '../shared/litellm';
import { query } from '../shared/db';

// Initialize logger
const logger = new Logger({ handler: 'router' });

/**
 * Main handler
 */
export async function handler(
  event: APIGatewayProxyEvent,
  context: Context
): Promise<APIGatewayProxyResult> {
  // Set request context
  const requestLogger = logger.child({
    requestId: context.awsRequestId,
    path: event.path,
    method: event.httpMethod,
  });

  try {
    const config = getConfig();

    requestLogger.info('Request received', {
      queryParams: event.queryStringParameters,
      hasBody: !!event.body,
    });

    // Route based on path
    const path = event.path.replace(/^\/api\/v2/, '');

    switch (true) {
      // Health check
      case path === '/health' || path === '/':
        return await handleHealthCheck(requestLogger);

      // Ready check (deep health)
      case path === '/ready':
        return await handleReadyCheck(requestLogger);

      // Version info
      case path === '/version':
        return handleVersionInfo();

      // Metrics (admin only)
      case path === '/metrics':
        return await handleMetrics(event, requestLogger);

      default:
        throw new NotFoundError(`Route ${event.httpMethod} ${event.path}`);
    }
  } catch (error) {
    return handleError(error, requestLogger);
  }
}

/**
 * Basic health check
 */
async function handleHealthCheck(logger: Logger): Promise<APIGatewayProxyResult> {
  const config = getConfig();

  return success({
    status: 'healthy',
    timestamp: new Date().toISOString(),
    environment: config.ENVIRONMENT,
    tier: config.TIER,
  });
}

/**
 * Deep health check (ready probe)
 */
async function handleReadyCheck(logger: Logger): Promise<APIGatewayProxyResult> {
  const config = getConfig();
  const checks: Record<string, { status: string; latency?: number }> = {};
  const startTime = Date.now();

  // Check database
  try {
    const dbStart = Date.now();
    await query('SELECT 1', {}, logger);
    checks.database = {
      status: 'healthy',
      latency: Date.now() - dbStart,
    };
  } catch (error) {
    checks.database = { status: 'unhealthy' };
    logger.error('Database health check failed', error as Error);
  }

  // Check LiteLLM
  try {
    const llmStart = Date.now();
    const health = await checkLiteLLMHealth(logger);
    checks.litellm = {
      status: health.status,
      latency: Date.now() - llmStart,
    };
  } catch (error) {
    checks.litellm = { status: 'unhealthy' };
    logger.error('LiteLLM health check failed', error as Error);
  }

  // Determine overall status
  const allHealthy = Object.values(checks).every(c => c.status === 'healthy');
  const overallStatus = allHealthy ? 'ready' : 'degraded';

  logger.info('Ready check completed', {
    status: overallStatus,
    checks,
    totalLatency: Date.now() - startTime,
  });

  return success({
    status: overallStatus,
    timestamp: new Date().toISOString(),
    environment: config.ENVIRONMENT,
    tier: config.TIER,
    checks,
  }, allHealthy ? 200 : 503);
}

/**
 * Version information
 */
function handleVersionInfo(): APIGatewayProxyResult {
  const config = getConfig();
  return success({
    version: '2.2.0',
    appId: config.APP_ID,
    environment: config.ENVIRONMENT,
    tier: config.TIER,
    apiVersion: 'v2',
    buildDate: '2024-12',
  });
}

/**
 * Metrics endpoint (admin only)
 */
async function handleMetrics(
  event: APIGatewayProxyEvent,
  logger: Logger
): Promise<APIGatewayProxyResult> {
  // Metrics collection - use Section 12 MetricsCollector service
  // See Section 12.2 for implementation details
  // await metricsCollector.recordUsage({ ... });
  // This would aggregate data from CloudWatch, DynamoDB usage table, etc.

  return success({
    message: 'Metrics endpoint - coming in Prompt 5',
    timestamp: new Date().toISOString(),
  });
}
```

PART 7: CHAT LAMBDA

packages/infrastructure/lambda/api/chat.ts

```
/**
 * Chat Lambda - AI completions handler
 *
 * Handles:
 * - Chat completions via LiteLLM
 * - Streaming responses
 * - PHI sanitization
 * - Usage tracking
 */

import type {
  APIGatewayProxyEvent,
  APIGatewayProxyResult,
  Context,
} from 'aws-lambda';
```

```
import { z } from 'zod';
import { v4 as uuid } from 'uuid';
import { Logger } from '../shared/logger';
import { getConfig, getFeatureFlags } from '../shared/config';
import { success, handleError, formatSSE, streamingHeaders } from '../shared/response';
import { extractAuthContext, logAuthContext } from '../shared/auth';
import { ValidationError, NotFoundError } from '../shared/errors';
import {
  createChatCompletion,
  streamChatCompletion,
  calculateCost,
  ChatCompletionRequest,
  ChatMessage,
} from '../shared/litellm';
import { getModelByName, getTenantPhiConfig, recordUsage } from '../shared/db';
import { sanitizePHI, createMappingRecord, DEFAULT_PHI_CONFIG } from '../shared/phi';
import { DynamoDBClient } from '@aws-sdk/client-dynamodb';
import { DynamoDBDocumentClient, PutCommand } from '@aws-sdk/lib-dynamodb';

// Initialize clients
const ddbClient = DynamoDBDocumentClient.from(new DynamoDBClient({}));
const logger = new Logger({ handler: 'chat' });

// Request validation schema
const chatRequestSchema = z.object({
  model: z.string().min(1),
  messages: z.array(z.object({
    role: z.enum(['system', 'user', 'assistant', 'function', 'tool']),
    content: z.union([
      z.string(),
      z.array(z.object({
        type: z.enum(['text', 'image_url']),
        text: z.string().optional(),
        image_url: z.object({
          url: z.string(),
          detail: z.enum(['low', 'high', 'auto']).optional(),
        }).optional(),
      })),
    ]),
    name: z.string().optional(),
  })).min(1),
  max_tokens: z.number().int().positive().max(128000).optional(),
  temperature: z.number().min(0).max(2).optional(),
  stream: z.boolean().optional().default(false),
  session_id: z.string().uuid().optional(),
  enable_phi: z.boolean().optional(),
  phi_categories: z.array(z.string()).optional(),
});
```

```
type ChatRequest = z.infer<typeof chatRequestSchema>;

/**
 * Main handler
 */
export async function handler(
  event: APIGatewayProxyEvent,
  context: Context
): Promise<APIGatewayProxyResult> {
  const requestLogger = logger.child({
    requestId: context.awsRequestId,
    path: event.path,
  });

  try {
    // Extract authentication
    const auth = extractAuthContext(event);
    logAuthContext(auth, requestLogger);
    requestLogger.setTenantId(auth.tenantId);
    requestLogger.setUserId(auth.userId);

    // Parse and validate request
    const body = event.body ? JSON.parse(event.body) : {};
    const parseResult = chatRequestSchema.safeParse(body);

    if (!parseResult.success) {
      throw new ValidationError(
        'Invalid request body',
        parseResult.error.flatten().fieldErrors as Record<string, string[]>
      );
    }

    const request = parseResult.data;
    const config = getConfig();
    const features = getFeatureFlags(config.TIER);

    // Get model information
    const model = await getModelByName(request.model, requestLogger);
    if (!model) {
      throw new NotFoundError(`Model ${request.model}`);
    }

    // Get tenant PHI config
    let phiConfig = DEFAULT_PHI_CONFIG;
    if (features.phiSanitization && request.enable_phi !== false) {
      try {
        const tenantConfig = await getTenantPhiConfig(auth.tenantId, requestLogger);
        if (tenantConfig) {
          phiConfig = tenantConfig;
        }
      } catch {
        // Use default if tenant config not found
      }
    }

    // Sanitize PHI in messages
    const sanitizedMessages = await sanitizeMessages(
      request.messages,
      phiConfig,
      auth.tenantId,
      request.session_id || null,
      requestLogger
    );

    // Build LiteLLM request
    const litellmRequest: ChatCompletionRequest = {
      model: model.name,
      messages: sanitizedMessages.messages,
      max_tokens: request.max_tokens,
      temperature: request.temperature,
      stream: request.stream,
      user: auth.userId,
      metadata: {
        tenant_id: auth.tenantId,
        session_id: request.session_id,
        app_id: config.APP_ID,
      },
    };

    // Handle streaming vs non-streaming
    if (request.stream) {
      return await handleStreamingRequest(
        litellmRequest,
        model,
        auth,
        request,
        sanitizedMessages.mappingId,
        requestLogger
      );
    } else {
      return await handleNonStreamingRequest(
        litellmRequest,
        model,
        auth,
        request,
        sanitizedMessages.mappingId,
        requestLogger
      );
    }
  } catch (error) {
    return handleError(error, requestLogger);
  }
}

/**
 * Sanitize PHI in messages
 */
async function sanitizeMessages(
  messages: ChatRequest['messages'],
  phiConfig: typeof DEFAULT_PHI_CONFIG,
  tenantId: string,
  sessionId: string | null,
  logger: Logger
): Promise<{ messages: ChatMessage[]; mappingId: string }> {
  if (phiConfig.mode === 'disabled') {
    return {
      messages: messages as ChatMessage[],
      mappingId: '',
    };
  }

  const config = getConfig();
  const sanitizedMessages: ChatMessage[] = [];
  let combinedMappingId = '';
  const allMappings: Record<string, string> = {};

  for (const message of messages) {
    if (typeof message.content === 'string') {
      const result = sanitizePHI(message.content, phiConfig, logger);

      if (result.matches.length > 0) {
        combinedMappingId = combinedMappingId || result.mappingId;

        // Collect mappings
        for (const match of result.matches) {
          allMappings[match.placeholder] = match.original;
        }
      }

      sanitizedMessages.push({
        ...message,
        content: result.sanitizedText,
      } as ChatMessage);
    } else {
      // Handle content parts (images, etc.)
      const sanitizedParts = [];

      for (const part of message.content) {
        if (part.type === 'text' && part.text) {
          const result = sanitizePHI(part.text, phiConfig, logger);

          if (result.matches.length > 0) {
            combinedMappingId = combinedMappingId || result.mappingId;
            for (const match of result.matches) {
              allMappings[match.placeholder] = match.original;
            }
          }

          sanitizedParts.push({
            ...part,
            text: result.sanitizedText,
          });
        } else {
          sanitizedParts.push(part);
        }
      }

      sanitizedMessages.push({
        ...message,
        content: sanitizedParts,
      } as ChatMessage);
    }
  }

  // Store PHI mapping in DynamoDB for re-identification
  if (combinedMappingId && Object.keys(allMappings).length > 0) {
    const ttlHours = phiConfig.reidentification.mapping_ttl_hours;
    const expiresAt = Math.floor(Date.now() / 1000) + (ttlHours * 60 * 60);

    await ddbClient.send(new PutCommand({
      TableName: config.CACHE_TABLE,
      Item: {
        pk: `phi#${combinedMappingId}`,
        tenant_id: tenantId,
        session_id: sessionId,
        mappings: allMappings,
        created_at: new Date().toISOString(),
        ttl: expiresAt,
      },
    }));
```

```
    logger.info('PHI mappings stored', {
      mappingId: combinedMappingId,
      mappingCount: Object.keys(allMappings).length,
    });
  }

  return {
    messages: sanitizedMessages,
    mappingId: combinedMappingId,
  };
}

/**
 * Handle non-streaming request
 */
async function handleNonStreamingRequest(
  request: ChatCompletionRequest,
  model: Awaited<ReturnType<typeof getModelByName>>,
  auth: ReturnType<typeof extractAuthContext>,
  originalRequest: ChatRequest,
  phiMappingId: string,
  logger: Logger
): Promise<APIGatewayProxyResult> {
  const startTime = Date.now();

  // Call LiteLLM
  const response = await createChatCompletion(request, logger);
  const latency = Date.now() - startTime;

  // Calculate costs
  const pricing = model!.pricing;
  const { cost, billed } = calculateCost(response.usage, {
    input_tokens: pricing.input_tokens || 0,
    output_tokens: pricing.output_tokens || 0,
    billed_markup: pricing.billed_markup,
  });

  // Record usage
  await recordUsage({
    tenant_id: auth.tenantId,
    user_id: auth.userId,
    session_id: originalRequest.session_id || null,
    model_id: model!.id,
    provider_id: model!.provider_id,
    input_tokens: response.usage.prompt_tokens,
    output_tokens: response.usage.completion_tokens,
    total_tokens: response.usage.total_tokens,
    cost,
    billed_amount: billed,
    request_type: 'chat.completion',
    latency_ms: latency,
  }, logger);

  // Store in sessions table if session_id provided
  if (originalRequest.session_id) {
    const config = getConfig();
    await ddbClient.send(new PutCommand({
      TableName: config.SESSIONS_TABLE,
      Item: {
        pk: originalRequest.session_id,
        gsi1pk: auth.userId,
        gsi1sk: new Date().toISOString(),
        messages: [
          ...originalRequest.messages,
          response.choices[0]?.message,
        ],
        model: request.model,
        tenant_id: auth.tenantId,
        phi_mapping_id: phiMappingId || undefined,
        created_at: new Date().toISOString(),
        updated_at: new Date().toISOString(),
        ttl: Math.floor(Date.now() / 1000) + (30 * 24 * 60 * 60), // 30 days
      },
    }));
  }

  logger.info('Chat completion successful', {
    model: request.model,
    inputTokens: response.usage.prompt_tokens,
    outputTokens: response.usage.completion_tokens,
    latency,
    cost,
  });

  return success({
    id: response.id,
    object: response.object,
    created: response.created,
    model: response.model,
    choices: response.choices,
    usage: response.usage,
    session_id: originalRequest.session_id,
    phi_mapping_id: phiMappingId || undefined,
  });
}
```

```
/**
 * Handle streaming request
 * Note: API Gateway doesn't support true streaming, so we buffer and return
 * For true streaming, use WebSocket API or Lambda Function URLs
 */
async function handleStreamingRequest(
  request: ChatCompletionRequest,
  model: Awaited<ReturnType<typeof getModelByName>>,
  auth: ReturnType<typeof extractAuthContext>,
  originalRequest: ChatRequest,
  phiMappingId: string,
  logger: Logger
): Promise<APIGatewayProxyResult> {
  const startTime = Date.now();
  const chunks: string[] = [];
  let totalContent = '';
  let finishReason: string | null = null;
  let usage = { prompt_tokens: 0, completion_tokens: 0, total_tokens: 0 };

  try {
    for await (const chunk of streamChatCompletion(request, logger)) {
      chunks.push(formatSSE(chunk));

      if (chunk.choices[0]?.delta?.content) {
        totalContent += chunk.choices[0].delta.content;
      }

      if (chunk.choices[0]?.finish_reason) {
        finishReason = chunk.choices[0].finish_reason;
      }

      if (chunk.usage) {
        usage = chunk.usage;
      }
    }

    // Add done marker
    chunks.push(formatSSE('[DONE]'));
  } catch (error) {
    logger.error('Streaming error', error as Error);
    throw error;
  }

  const latency = Date.now() - startTime;

  // Record usage if we have it
  if (usage.total_tokens > 0) {
    const pricing = model!.pricing;
    const { cost, billed } = calculateCost(usage, {
      input_tokens: pricing.input_tokens || 0,
      output_tokens: pricing.output_tokens || 0,
      billed_markup: pricing.billed_markup,
    });

    await recordUsage({
      tenant_id: auth.tenantId,
      user_id: auth.userId,
      session_id: originalRequest.session_id || null,
      model_id: model!.id,
      provider_id: model!.provider_id,
      input_tokens: usage.prompt_tokens,
      output_tokens: usage.completion_tokens,
      total_tokens: usage.total_tokens,
      cost,
      billed_amount: billed,
      request_type: 'chat.completion.stream',
      latency_ms: latency,
    }, logger);
  }

  logger.info('Streaming completion successful', {
    model: request.model,
    chunkCount: chunks.length,
    contentLength: totalContent.length,
    latency,
  });

  // Return SSE formatted response
  // Note: For true streaming, implement Lambda Function URL or WebSocket
  return {
    statusCode: 200,
    headers: streamingHeaders(),
    body: chunks.join(''),
  };
}
```

PART 8: MODELS LAMBDA

packages/infrastructure/lambda/api/models.ts

```
/**
 * Models Lambda - Dynamic model registry
 *
 * Handles:
 * - List all models
 * - Get model by ID
 * - Filter by specialty, provider, status
 * - Admin: Update model status, thermal state
 */

import type {
  APIGatewayProxyEvent,
  APIGatewayProxyResult,
  Context,
} from 'aws-lambda';
import { z } from 'zod';
import { Logger } from '../shared/logger';
import { getConfig } from '../shared/config';
import { success, handleError, noContent } from '../shared/response';
import { extractAuthContext, requireAdmin, logAuthContext } from '../shared/auth';
import { ValidationError, NotFoundError } from '../shared/errors';
import { getModels, getModelById, updateModel as dbUpdateModel } from '../shared/db';
import { createAuditLog } from '../shared/db';

// Initialize logger
const logger = new Logger({ handler: 'models' });

// Query parameters schema
const listQuerySchema = z.object({
  provider_id: z.string().optional(),
  specialty: z.string().optional(),
  status: z.enum(['active', 'inactive', 'deprecated', 'coming_soon']).optional(),
  type: z.enum(['external', 'self-hosted']).optional(),
  supports_vision: z.string().transform(v => v === 'true').optional(),
  supports_streaming: z.string().transform(v => v === 'true').optional(),
  limit: z.string().transform(Number).pipe(z.number().int().min(1).max(100)).optional(),
  offset: z.string().transform(Number).pipe(z.number().int().min(0)).optional(),
});

// Update request schema (admin only)
const updateModelSchema = z.object({
  status: z.enum(['active', 'inactive', 'deprecated', 'coming_soon']).optional(),
  thermal_state: z.enum(['OFF', 'COLD', 'WARM', 'HOT', 'AUTOMATIC']).optional(),
  display_name: z.string().min(1).max(200).optional(),
  description: z.string().max(2000).optional(),
});

/**
 * Main handler
 */
export async function handler(
  event: APIGatewayProxyEvent,
  context: Context
```

```
): Promise<APIGatewayProxyResult> {
  const requestLogger = logger.child({
    requestId: context.awsRequestId,
    path: event.path,
    method: event.httpMethod,
  });

  try {
    // Extract authentication
    const auth = extractAuthContext(event);
    logAuthContext(auth, requestLogger);

    // Parse path parameters
    const modelId = event.pathParameters?.modelId;

    switch (event.httpMethod) {
      case 'GET':
        if (modelId) {
          return await handleGetModel(modelId, requestLogger);
        }
        return await handleListModels(event, requestLogger);

      case 'PUT':
      case 'PATCH':
        if (!modelId) {
          throw new ValidationError('Model ID required for update');
        }
        requireAdmin(auth);
        return await handleUpdateModel(modelId, event, auth, requestLogger);

      default:
        throw new ValidationError(`Method ${event.httpMethod} not allowed`);
    }
  } catch (error) {
    return handleError(error, requestLogger);
  }
}

/**
 * List models with filtering
 */
async function handleListModels(
  event: APIGatewayProxyEvent,
  logger: Logger
): Promise<APIGatewayProxyResult> {
  // Parse and validate query parameters
  const queryResult = listQuerySchema.safeParse(event.queryStringParameters || {});
  if (!queryResult.success) {
    throw new ValidationError(
      'Invalid query parameters',
      queryResult.error.flatten().fieldErrors as Record<string, string[]>
    );
  }

  const filters = queryResult.data;

  // Query database
  const result = await getModels({
    providerId: filters.provider_id,
    specialty: filters.specialty,
    status: filters.status,
    type: filters.type,
    supportsVision: filters.supports_vision,
    supportsStreaming: filters.supports_streaming,
    limit: filters.limit || 50,
    offset: filters.offset || 0,
  }, logger);

  // Transform to API response format
  const models = result.rows.map(transformModel);

  logger.info('Models listed', {
    count: models.length,
    filters,
  });

  return success({
    models,
    pagination: {
      limit: filters.limit || 50,
      offset: filters.offset || 0,
      total: result.rowCount,
      hasMore: result.rowCount > (filters.offset || 0) + models.length,
    },
  });
}

/**
 * Get single model by ID
 */
async function handleGetModel(
  modelId: string,
  logger: Logger
): Promise<APIGatewayProxyResult> {
  const model = await getModelById(modelId, logger);
  logger.info('Model retrieved', { modelId });

  return success({
    model: transformModel(model),
  });
}

/**
 * Update model (admin only)
 */
async function handleUpdateModel(
  modelId: string,
  event: APIGatewayProxyEvent,
  auth: ReturnType<typeof extractAuthContext>,
  logger: Logger
): Promise<APIGatewayProxyResult> {
  // Parse and validate request body
  const body = event.body ? JSON.parse(event.body) : {};
  const parseResult = updateModelSchema.safeParse(body);

  if (!parseResult.success) {
    throw new ValidationError(
      'Invalid request body',
      parseResult.error.flatten().fieldErrors as Record<string, string[]>
    );
  }

  const updates = parseResult.data;

  if (Object.keys(updates).length === 0) {
    throw new ValidationError('No updates provided');
  }

  // Get current model
  const currentModel = await getModelById(modelId, logger);

  // Update model
  const updatedModel = await dbUpdateModel(modelId, updates, logger);

  // Create audit log
  await createAuditLog({
    tenant_id: auth.tenantId,
    user_id: null,
    admin_id: auth.userId,
    action: 'model.update',
    resource_type: 'model',
    resource_id: modelId,
    details: {
      before: {
        status: currentModel.status,
        thermal_state: currentModel.thermal_state,
        display_name: currentModel.display_name,
      },
      after: updates,
    },
    ip_address: event.requestContext.identity?.sourceIp || null,
    user_agent: event.headers['User-Agent'] || null,
  }, logger);

  logger.info('Model updated', {
    modelId,
    updates,
    adminId: auth.userId,
  });

  return success({
    model: transformModel(updatedModel),
  });
}

/**
 * Transform database model to API response format
 */
function transformModel(dbModel: any) {
  return {
    id: dbModel.id,
    providerId: dbModel.provider_id,
    name: dbModel.name,
    displayName: dbModel.display_name,
    description: dbModel.description,
    type: dbModel.type,
    specialty: dbModel.specialty,
    capabilities: dbModel.capabilities,
    contextWindow: dbModel.context_window,
    maxOutputTokens: dbModel.max_output_tokens,
    supportsFunctions: dbModel.supports_functions,
    supportsVision: dbModel.supports_vision,
    supportsStreaming: dbModel.supports_streaming,
    hasThinkingMode: dbModel.has_thinking_mode,
    thinkingBudgetTokens: dbModel.thinking_budget_tokens,
    pricing: {
      inputTokens: dbModel.pricing?.input_tokens,
      outputTokens: dbModel.pricing?.output_tokens,
      perImage: dbModel.pricing?.per_image,
      perMinuteAudio: dbModel.pricing?.per_minute_audio,
      perMinuteVideo: dbModel.pricing?.per_minute_video,
      per3DModel: dbModel.pricing?.per_3d_model,
      billedMarkup: dbModel.pricing?.billed_markup,
    },
    thermalState: dbModel.thermal_state,
    thermalConfig: dbModel.thermal_config,
    status: dbModel.status,
    releaseDate: dbModel.release_date,
    deprecationDate: dbModel.deprecation_date,
    createdAt: dbModel.created_at,
    updatedAt: dbModel.updated_at,
  };
}
```

PART 9: PROVIDERS LAMBDA

packages/infrastructure/lambda/api/providers.ts

```
/**
 * Providers Lambda - AI provider management
 *
 * Handles:
 * - List all providers
 * - Get provider by ID
 * - Get provider models
 * - Admin: Update provider status
 */

import type {
  APIGatewayProxyEvent,
  APIGatewayProxyResult,
  Context,
} from 'aws-lambda';
import { z } from 'zod';
import { Logger } from '../shared/logger';
import { getConfig } from '../shared/config';
import { success, handleError } from '../shared/response';
import { extractAuthContext, requireAdmin, logAuthContext } from '../shared/auth';
import { ValidationError } from '../shared/errors';
import {
  getProviders,
  getProviderById,
  updateProvider as dbUpdateProvider,
  getModels,
  createAuditLog,
} from '../shared/db';
```

```
// Initialize logger
const logger = new Logger({ handler: 'providers' });

// Query parameters schema
const listQuerySchema = z.object({
  type: z.enum(['external', 'self-hosted', 'mid-tier']).optional(),
  status: z.enum(['active', 'inactive', 'deprecated']).optional(),
  hipaa_compliant: z.string().transform(v => v === 'true').optional(),
  limit: z.string().transform(Number).pipe(z.number().int().min(1).max(100)).optional(),
  offset: z.string().transform(Number).pipe(z.number().int().min(0)).optional(),
});

// Update request schema (admin only)
const updateProviderSchema = z.object({
  status: z.enum(['active', 'inactive', 'deprecated']).optional(),
  hipaa_compliant: z.boolean().optional(),
  config: z.record(z.unknown()).optional(),
});

/**
 * Main handler
 */
export async function handler(
  event: APIGatewayProxyEvent,
  context: Context
): Promise<APIGatewayProxyResult> {
  const requestLogger = logger.child({
    requestId: context.awsRequestId,
    path: event.path,
    method: event.httpMethod,
  });

  try {
    // Extract authentication
    const auth = extractAuthContext(event);
    logAuthContext(auth, requestLogger);

    // Parse path
    const providerId = event.pathParameters?.providerId;
    const subPath = event.path.split('/').pop();

    switch (event.httpMethod) {
      case 'GET':
        if (providerId) {
          if (subPath === 'models') {
            return await handleGetProviderModels(providerId, requestLogger);
          }
          return await handleGetProvider(providerId, requestLogger);
        }
        return await handleListProviders(event, requestLogger);

      case 'PUT':
      case 'PATCH':
        if (!providerId) {
          throw new ValidationError('Provider ID required for update');
        }
        requireAdmin(auth);
        return await handleUpdateProvider(providerId, event, auth, requestLogger);

      default:
        throw new ValidationError(`Method ${event.httpMethod} not allowed`);
    }
  } catch (error) {
    return handleError(error, requestLogger);
  }
}

/**
 * List providers with filtering
 */
async function handleListProviders(
  event: APIGatewayProxyEvent,
  logger: Logger
): Promise<APIGatewayProxyResult> {
  // Parse and validate query parameters
  const queryResult = listQuerySchema.safeParse(event.queryStringParameters || {});

  if (!queryResult.success) {
    throw new ValidationError(
      'Invalid query parameters',
      queryResult.error.flatten().fieldErrors as Record<string, string[]>
    );
  }

  const filters = queryResult.data;

  // Query database
  const result = await getProviders({
    type: filters.type,
    status: filters.status,
    hipaaCompliant: filters.hipaa_compliant,
    limit: filters.limit || 50,
    offset: filters.offset || 0,
  }, logger);

  // Transform to API response format
  const providers = result.rows.map(transformProvider);

  logger.info('Providers listed', {
    count: providers.length,
    filters,
  });

  return success({
    providers,
    pagination: {
      limit: filters.limit || 50,
      offset: filters.offset || 0,
      total: result.rowCount,
      hasMore: result.rowCount > (filters.offset || 0) + providers.length,
    },
  });
}

/**
 * Get single provider by ID
 */
async function handleGetProvider(
  providerId: string,
  logger: Logger
): Promise<APIGatewayProxyResult> {
  const provider = await getProviderById(providerId, logger);

  logger.info('Provider retrieved', { providerId });

  return success({
    provider: transformProvider(provider),
  });
}

/**
 * Get models for a provider
 */
async function handleGetProviderModels(
  providerId: string,
  logger: Logger
): Promise<APIGatewayProxyResult> {
  // Verify provider exists
  await getProviderById(providerId, logger);

  // Get provider's models
  const result = await getModels({
    providerId,
    status: 'active',
  }, logger);

  const models = result.rows.map(model => ({
    id: model.id,
    name: model.name,
    displayName: model.display_name,
    specialty: model.specialty,
    status: model.status,
    thermalState: model.thermal_state,
  }));

  logger.info('Provider models retrieved', {
    providerId,
    modelCount: models.length,
  });

  return success({
    providerId,
    models,
  });
}

/**
 * Update provider (admin only)
 */
async function handleUpdateProvider(
  providerId: string,
  event: APIGatewayProxyEvent,
  auth: ReturnType<typeof extractAuthContext>,
  logger: Logger
): Promise<APIGatewayProxyResult> {
  // Parse and validate request body
  const body = event.body ? JSON.parse(event.body) : {};
  const parseResult = updateProviderSchema.safeParse(body);

  if (!parseResult.success) {
    throw new ValidationError(
      'Invalid request body',
      parseResult.error.flatten().fieldErrors as Record<string, string[]>
    );
  }

  const updates = parseResult.data;

  if (Object.keys(updates).length === 0) {
    throw new ValidationError('No updates provided');
  }
  // Get current provider
  const currentProvider = await getProviderById(providerId, logger);

  // Update provider
  const updatedProvider = await dbUpdateProvider(providerId, {
    status: updates.status,
    hipaa_compliant: updates.hipaa_compliant,
    config: updates.config,
  }, logger);

  // Create audit log
  await createAuditLog({
    tenant_id: auth.tenantId,
    user_id: null,
    admin_id: auth.userId,
    action: 'provider.update',
    resource_type: 'provider',
    resource_id: providerId,
    details: {
      before: {
        status: currentProvider.status,
        hipaa_compliant: currentProvider.hipaa_compliant,
      },
      after: updates,
    },
    ip_address: event.requestContext.identity?.sourceIp || null,
    user_agent: event.headers['User-Agent'] || null,
  }, logger);

  logger.info('Provider updated', {
    providerId,
    updates,
    adminId: auth.userId,
  });

  return success({
    provider: transformProvider(updatedProvider),
  });
}

/**
 * Transform database provider to API response format
 */
function transformProvider(dbProvider: any) {
  return {
    id: dbProvider.id,
    name: dbProvider.name,
    type: dbProvider.type,
    status: dbProvider.status,
    hipaaCompliant: dbProvider.hipaa_compliant,
    baaAvailable: dbProvider.baa_available,
    baseUrl: dbProvider.base_url,
    authType: dbProvider.auth_type,
    capabilities: dbProvider.capabilities,
    createdAt: dbProvider.created_at,
    updatedAt: dbProvider.updated_at,
  };
}
```

DEPLOYMENT VERIFICATION

After deploying the Lambda functions, verify they work:

```
# 1. Health check
curl https://api.thinktank.YOUR_DOMAIN.com/api/v2/health

# Expected response:
# {
#   "success": true,
#   "data": {
#     "status": "healthy",
#     "timestamp": "2024-12-21T...",
#     "environment": "dev",
#     "tier": 1
#   }
# }

# 2. List models (requires auth token)
curl -H "Authorization: Bearer $TOKEN" \
  https://api.thinktank.YOUR_DOMAIN.com/api/v2/models

# 3. List providers
curl -H "Authorization: Bearer $TOKEN" \
  https://api.thinktank.YOUR_DOMAIN.com/api/v2/providers

# 4. Chat completion
curl -X POST \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{
    "model": "gpt-4",
    "messages": [{"role": "user", "content": "Hello!"}]
  }' \
  https://api.thinktank.YOUR_DOMAIN.com/api/v2/chat/completions
```

NEXT PROMPTS

Continue with: - **Prompt 5**: Lambda Functions - Admin & Billing (Invitations, Approvals, Metering) - **Prompt 6**: Self-Hosted Models & Mid-Level Services Configuration - **Prompt 7**: External Providers & Database Schema/Migrations - **Prompt 8**: Admin Web Dashboard (Next.js) - **Prompt 9**: Assembly & Deployment Guide

---

End of Prompt 4: Lambda Functions - Core RADIANT v2.2.0 - December 2024

â•

END OF SECTION 4

â•

â•