Contents

RADIANT v4.18.0 - Lambda Handlers Export

Component: AWS Lambda Functions **Language**: TypeScript (Node.js 20.x), Python 3.11 **Files**: 100+ Lambda handlers across 48 directories **Runtime**: AWS Lambda

---

Architecture Narrative

RADIANT's backend logic is implemented as **AWS Lambda functions** organized into logical domains. Each Lambda handles specific functionality and communicates through API Gateway, SQS queues, or direct invocation.

Lambda Organization

```
lambda/
  admin/            # 48 admin API handlers
  api/              # 4 public API handlers
  brain/            # 3 AGI Brain handlers
  thinktank/        # 12 Think Tank handlers
  consciousness/    # 9 consciousness engine handlers
  billing/          # 3 billing handlers
  security/         # 3 security handlers
  scheduled/        # 3 scheduled task handlers
  collaboration/    # 4 real-time collaboration handlers
  thermal/          # 5 thermal management handlers
  tier-transition/  # 20 tier upgrade handlers
  shared/           # 344 shared utilities/services
  ...               # 20+ additional domains
```

Handler Patterns

1. **API Gateway Integration** - REST/WebSocket event handling
2. **SQS Event Source** - Queue-triggered processing
3. **EventBridge Scheduled** - Cron-based tasks
4. **Direct Invocation** - Lambda-to-Lambda calls

Key Lambda Implementations

1. brain/inference.ts

Purpose: Main entry point for AGI Brain inference requests. Orchestrates Ghost Vector loading, SOFAI routing, context assembly, and LLM calls.

```
/**
 * RADIANT v6.0.4 - Brain Inference Lambda
 * Main entry point for AGI Brain inference requests
 *
 * Orchestrates:
 * - Ghost Vector loading/saving
 * - SOFAI routing
 * - Context assembly (Compliance Sandwich)
 * - Flash fact detection
 * - Async re-anchoring
 */

import { APIGatewayProxyEvent, APIGatewayProxyResult } from 'aws-lambda';
import { v4 as uuidv4 } from 'uuid';
import Redis from 'ioredis';
import { enhancedLogger as logger } from '../shared/logging/enhanced-logger';
import { executeStatement } from '../shared/db/client';
import { brainConfigService } from '../shared/services/brain-config.service';
import { ghostManagerService } from '../shared/services/ghost-manager.service';
import { flashBufferService } from '../shared/services/flash-buffer.service';
import { sofaiRouterService } from '../shared/services/sofai-router.service';
import { contextAssemblerService } from '../shared/services/context-assembler.service';
import { oversightService } from '../shared/services/oversight.service';
import {
  BrainInferenceRequest,
  BrainInferenceResponse,
  SystemLevel,
  ConversationMessage,
} from '@radiant/shared';
import { embeddingService } from '../shared/services/embedding.service';
import { ecdVerificationService } from '../shared/services/ecd-verification.service';

let redisClient: Redis | null = null;
let servicesInitialized = false;

async function initializeServices(): Promise<void> {
  if (servicesInitialized) return;

  const redisUrl = process.env.REDIS_URL || process.env.REDIS_ENDPOINT;
  if (redisUrl && !redisClient) {
    try {
      redisClient = new Redis(redisUrl);
      redisClient.on('error', (err) => {
        logger.error('Redis connection error', { error: String(err) });
      });

      // Initialize services with Redis client
      const redisAdapter = {
        get: async (key: string) => redisClient?.get(key) ?? null,
        set: async (key: string, value: string, options?: { EX?: number }) => {
          if (options?.EX) {
            await redisClient?.setex(key, options.EX, value);
          } else {
            await redisClient?.set(key, value);
          }
        },
        lpush: async (key: string, ...values: string[]) => redisClient?.lpush(key, ...values)
        lrange: async (key: string, start: number, stop: number) => redisClient?.lrange(key, s
        ltrim: async (key: string, start: number, stop: number) => { await redisClient?.ltrim(
        expire: async (key: string, seconds: number) => { await redisClient?.expire(key, secon
        del: async (key: string) => { await redisClient?.del(key); },
      };

      flashBufferService.initialize(redisAdapter);
      ghostManagerService.initialize(redisAdapter);
      logger.info('Brain services initialized with Redis');
    } catch (err) {
      logger.warn('Redis unavailable, services running without cache', { error: String(err) })
    }
  }

  servicesInitialized = true;
}
```

```
export async function handler(event: APIGatewayProxyEvent): Promise<APIGatewayProxyResult> {
  const startTime = Date.now();
  const requestId = uuidv4();

  await initializeServices();

  try {
    if (!event.body) {
      return error(400, 'Request body is required');
    }

    const request: BrainInferenceRequest = JSON.parse(event.body);
    const { userId, tenantId, prompt, conversationHistory, domain, forceSystemLevel, options }

    if (!userId || !tenantId || !prompt) {
      return error(400, 'userId, tenantId, and prompt are required');
    }

    logger.info('Brain inference started', { requestId, userId, tenantId, promptLength: prompt

    // Set tenant context for RLS
    await executeStatement(`SELECT set_config('app.current_tenant_id', $1, true)`, [
      { name: 'tenantId', value: { stringValue: tenantId } },
    ]);

    // Step 1: Load Ghost Vector
    let ghostVector: Float32Array | null = null;
    let ghostUpdated = false;

    if (options?.includeGhost !== false) {
      const ghostResult = await ghostManagerService.loadGhost(userId, tenantId);
      if (ghostResult.found && ghostResult.versionMatch) {
        ghostVector = ghostResult.vector;
      }
    }

    // Step 2: SOFAI Routing
    const routingDecision = await sofaiRouterService.route({
      prompt,
      userId,
      tenantId,
      domain,
      forceLevel: forceSystemLevel,
    });

    // Step 3: Assemble Context (Compliance Sandwich)
    const assembledContext = await contextAssemblerService.assemble({
      userId,
      tenantId,
      prompt,
      conversationHistory: conversationHistory as ConversationMessage[],
      ghostVector,
      domain: domain || sofaiRouterService.detectDomain(prompt),
    });

    // Step 4: Call LLM with ECD Verification (Truth Engine )
    const verificationResult = await ecdVerificationService.executeWithVerification({
      userId,
      tenantId,
      requestId,
      prompt,
      sourceContext: assembledContext.userContext,
      flashFacts: assembledContext.flashFacts.map(f => f.fact),
      retrievedDocs: [],
      domain: domain || sofaiRouterService.detectDomain(prompt),
      generateResponse: async (refinedPrompt: string) => {
        const result = await callLLM(
          contextAssemblerService.formatForModel({
            ...assembledContext,
            userContext: refinedPrompt,
          }),
          routingDecision.level,
          options
        );
        return result.response;
      },
    });

    // Step 5: Detect Flash Facts
    const flashFactsDetected = flashBufferService.detectFlashFacts(prompt);
    if (flashFactsDetected.detected) {
      for (const fact of flashFactsDetected.facts) {
        await flashBufferService.storeFact(userId, tenantId, fact);
      }
    }

    // Step 6: Check for Re-anchoring (async)
    if (options?.includeGhost !== false) {
      const needsReanchor = await ghostManagerService.checkReAnchorNeeded(userId, tenantId);
      if (needsReanchor) {
        ghostManagerService.reAnchorAsync(userId, tenantId, [prompt], async (history) => {
          const combinedText = history.join(' ').slice(0, 8000);
          const result = await embeddingService.generateEmbedding(combinedText);
          return new Float32Array(result.embedding);
        });
        ghostUpdated = true;
```

```
      }
    }

    // Build Response
    const response: BrainInferenceResponse = {
      response: verificationResult.finalResponse,
      systemLevel: routingDecision.level,
      routingDecision,
      budget: assembledContext.budget,
      ghostUpdated,
      flashFactsDetected,
      latencyMs: Date.now() - startTime,
      verification: {
        passed: verificationResult.passed,
        ecdScore: verificationResult.ecdScore.score,
        refinementAttempts: verificationResult.refinementAttempts,
        blocked: verificationResult.blocked,
      },
    };

    return success(response);
  } catch (err) {
    logger.error('Brain inference failed', { requestId, error: String(err) });
    return error(500, `Inference failed: ${String(err)}`);
  }
}
```

---

2. api/router.ts

Purpose: Main API router that dispatches requests to appropriate handlers based on HTTP method and path.

```
/**
 * API Router Lambda Handler
 * Main entry point for all API requests
 */

import type { APIGatewayProxyEvent, APIGatewayProxyResult, Context } from 'aws-lambda';
import { Logger } from '../shared/logger';
import { successResponse, errorResponse } from '../shared/response';
import { UnauthorizedError, NotFoundError, ValidationError } from '../shared/errors';
import { extractUserFromEvent } from '../shared/auth';
import { handleChat } from './chat';
import { handleModels } from './models';
import { handleProviders } from './providers';

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

```typescript
interface RouteHandler {
  (event: APIGatewayProxyEvent, context: Context): Promise<APIGatewayProxyResult>;
}

const routes: Map<string, Map<string, RouteHandler>> = new Map();

function registerRoute(method: string, pathPattern: string, handler: RouteHandler): void {
  if (!routes.has(method)) {
    routes.set(method, new Map());
  }
  routes.get(method)!.set(pathPattern, handler);
}

// Register routes
registerRoute('GET', '/api/v2/health', handleHealth);
registerRoute('POST', '/api/v2/chat/completions', handleChat);
registerRoute('GET', '/api/v2/models', handleModels);
registerRoute('GET', '/api/v2/models/{modelId}', handleModels);
registerRoute('GET', '/api/v2/providers', handleProviders);
registerRoute('POST', '/api/v2/usage', handleUsage);

export async function handler(
  event: APIGatewayProxyEvent,
  context: Context
): Promise<APIGatewayProxyResult> {
  const requestId = event.requestContext.requestId;
  logger.setRequestId(requestId);

  const startTime = Date.now();
  const method = event.httpMethod;
  const path = event.path;

  logger.info('Request received', { method, path, requestId });

  try {
    // Skip auth for health and usage endpoints
    if (path !== '/api/v2/health' && path !== '/api/v2/usage') {
      const user = await extractUserFromEvent(event);
      if (!user) {
        throw new UnauthorizedError('Invalid or missing authorization token');
      }
      logger.setTenantId(user.tenantId);
      logger.setUserId(user.userId);
    }

    const routeHandler = findRouteHandler(method, path);
    if (!routeHandler) {
      throw new NotFoundError(`Route not found: ${method} ${path}`);
    }

    const result = await routeHandler(event, context);

    logger.info('Request completed', {
      method,
      path,
      statusCode: result.statusCode,
      durationMs: Date.now() - startTime,
    });

    return result;
  } catch (error) {
    logger.error('Request failed', error as Error, {
      method,
      path,
      durationMs: Date.now() - startTime,
    });

    return errorResponse(error as Error);
  }
}

function findRouteHandler(method: string, path: string): RouteHandler | null {
  const methodRoutes = routes.get(method);
  if (!methodRoutes) return null;

  for (const [pattern, handler] of methodRoutes) {
    if (matchPath(pattern, path)) {
      return handler;
    }
  }
  return null;
}

function matchPath(pattern: string, path: string): boolean {
  const patternParts = pattern.split('/');
  const pathParts = path.split('/');

  if (patternParts.length !== pathParts.length) return false;

  for (let i = 0; i < patternParts.length; i++) {
    const patternPart = patternParts[i];
    const pathPart = pathParts[i];

    // Skip path parameters (e.g., {modelId})
    if (patternPart.startsWith('{') && patternPart.endsWith('}')) continue;
    if (patternPart !== pathPart) return false;
  }

  return true;
}

async function handleHealth(): Promise<APIGatewayProxyResult> {
  return successResponse({
    status: 'healthy',
    version: process.env.RADIANT_VERSION || 'unknown',
    timestamp: new Date().toISOString(),
    region: process.env.AWS_REGION,
  });
}

async function handleUsage(event: APIGatewayProxyEvent): Promise<APIGatewayProxyResult> {
  const body = JSON.parse(event.body || '{}');
  logger.info('Usage event recorded', {
    tenantId: body.tenant_id,
    modelId: body.model_id,
    tokens: body.total_tokens,
  });
  return successResponse({ recorded: true });
}
```

Lambda Handler Inventory

Admin Handlers (48 files)

Handler	Purpose
`admin/users.ts`	User management CRUD
`admin/tenants.ts`	Tenant management
`admin/models.ts`	AI model configuration
`admin/providers.ts`	Provider management
`admin/billing.ts`	Billing administration
`admin/analytics.ts`	Usage analytics
`admin/security.ts`	Security settings
`admin/compliance.ts`	Compliance controls
`admin/domains.ts`	Domain configuration
`admin/tiers.ts`	Tier management
…	38 more handlers

Think Tank Handlers (12 files)

Handler	Lines	Purpose
`thinktank/artifact-engine.ts`	17K	Artifact generation and management
`thinktank/brain-plan.ts`	15K	Brain planning and orchestration
`thinktank/conversations.ts`	9K	Conversation management
`thinktank/derivation-history.ts`	11K	Derivation tracking
`thinktank/domain-modes.ts`	11K	Domain mode handling
`thinktank/file-conversion.ts`	13K	File format conversion
`thinktank/ideas.ts`	6K	Idea management
`thinktank/model-categories.ts`	7K	Model categorization
`thinktank/models.ts`	20K	Model operations
`thinktank/ratings.ts`	13K	Rating system
`thinktank/user-context.ts`	15K	User context management
`thinktank/users.ts`	11K	Think Tank user operations

Consciousness Handlers (9 files)

Handler	Purpose
`consciousness/mcp-server.ts`	Model Context Protocol server
`consciousness/sleep-cycle.ts`	Weekly consciousness evolution
`consciousness/deep-research.ts`	Browser automation research
`consciousness/thinking-session.ts`	Async thinking sessions
`consciousness/budget-monitor.ts`	Cost control
`consciousness/admin-api.ts`	Consciousness admin APIs
`consciousness/memory-consolidation.ts`	Memory processing
`consciousness/dream-generator.ts`	Dream state generation
`consciousness/parameter-sync.ts`	Parameter synchronization

Billing Handlers (3 files)

Handler	Purpose
`billing/credits.ts`	Credit management
`billing/subscriptions.ts`	Subscription handling
`billing/invoices.ts`	Invoice generation

Security Handlers (3 files)

Handler	Purpose
`security/audit.ts`	Audit log management
`security/rbac.ts`	Role-based access control
`security/compliance.ts`	Compliance checking

Thermal Handlers (5 files)

Handler	Purpose
`thermal/monitor.ts`	System thermal monitoring
`thermal/throttle.ts`	Throttling control
`thermal/cooldown.ts`	Cooldown management
`thermal/alerts.ts`	Thermal alerting
`thermal/metrics.ts`	Thermal metrics

Tier Transition Handlers (20 files)

Handler	Purpose
`tier-transition/evaluate.ts`	Tier eligibility evaluation
`tier-transition/migrate.ts`	Migration execution
`tier-transition/validate.ts`	Transition validation
`tier-transition/rollback.ts`	Rollback handling
…	16 more handlers

Shared Utilities (344 files)

The `shared/` directory contains reusable code:

Database (`shared/db/`)

- `client.ts` - Aurora Data API client
- `pool.ts` - Connection pooling
- `migrations.ts` - Migration utilities

Services (`shared/services/`)

- `brain-config.service.ts` - Brain configuration
- `ghost-manager.service.ts` - Ghost vector management
- `flash-buffer.service.ts` - Flash fact buffer
- `sofai-router.service.ts` - SOFAI routing
- `context-assembler.service.ts` - Context assembly
- `oversight.service.ts` - Human oversight
- `embedding.service.ts` - Embedding generation
- `ecd-verification.service.ts` - ECD truth verification
- `governor/economic-governor.ts` - Cost optimization

Utilities (`shared/utils/`)

- `response.ts` - API response helpers
- `errors.ts` - Error types
- `auth.ts` - Authentication helpers

- `validation.ts` - Input validation
- `db-helpers.ts` - Database helpers

Logging (`shared/logging/`)

- `enhanced-logger.ts` - Structured logging
- `metrics.ts` - CloudWatch metrics

Handler Patterns

Standard API Handler

```
import { APIGatewayProxyEvent, APIGatewayProxyResult } from 'aws-lambda';
import { success, error } from '../shared/response';
import { withSecureDBContext } from '../shared/services/db-context.service';

export async function handler(event: APIGatewayProxyEvent): Promise<APIGatewayProxyResult> {
  try {
    return await withSecureDBContext(event, async (db, authContext) => {
      // Handler logic with RLS context
      const result = await db.query('SELECT * FROM table WHERE tenant_id = $1', [authContext.te
      return success(result);
    });
  } catch (err) {
    return error(500, err.message);
  }
}
```

SQS Event Handler

```
import { SQSEvent, SQSRecord } from 'aws-lambda';
import { logger } from '../shared/logging/enhanced-logger';

export async function handler(event: SQSEvent): Promise<void> {
  for (const record of event.Records) {
    await processRecord(record);
  }
}

async function processRecord(record: SQSRecord): Promise<void> {
  const body = JSON.parse(record.body);
  logger.info('Processing message', { messageId: record.messageId });
  // Process message
}
```

Scheduled Event Handler

```
import { ScheduledEvent } from 'aws-lambda';
import { logger } from '../shared/logging/enhanced-logger';

export async function handler(event: ScheduledEvent): Promise<void> {
  logger.info('Scheduled task started', {
    source: event.source,
    time: event.time,
  });

  // Execute scheduled task
  await runScheduledTask();

  logger.info('Scheduled task completed');
}
```

This concludes the Lambda handlers export. See other export files for additional components.