Contents

SECTION 5: LAMBDA FUNCTIONS - ADMIN & BILLING (v2.1.0)

â• â

> **Dependencies:** Sections 0-4 **Creates:** Invitations, Approvals, Metering, Billing handlers

Type Imports

```
import { Administrator, AdminRole, Invitation, ApprovalRequest, Invoice } from '@radiant/shared
```

> **IMPORTANT:** Use canonical table names: administrators, invitations, approval_requests

---

RADIANT v2.2.0 - Prompt 5: Lambda Functions - Admin & Billing

> **Prompt 5 of 9** | Target Size: ~60KB | Version: 3.7.0 | December 2024

OVERVIEW

This prompt creates the Admin & Billing Lambda functions for the RADIANT platform:

1. **Invitations Lambda** - Email-based administrator invitations with secure tokens
2. **Approvals Lambda** - Two-person approval workflow for production deployments
3. **Admin Users Lambda** - Administrator CRUD, roles, and MFA management
4. **Admin Profiles Lambda** - Preferences, notifications, and settings
5. **Metering Lambda** - Real-time usage event collection and tracking
6. **Billing Lambda** - Cost aggregation, invoicing, and payment processing
7. **Audit Lambda** - Comprehensive audit logging and compliance reporting
8. **Notifications Lambda** - Admin notification delivery and management

---

KEY FEATURES

Administrator Management

- **Email Invitations**: Secure token-based invitation system with expiry
- **Role-Based Access**: super_admin, admin, operator, auditor roles
- **MFA Requirement**: Mandatory MFA for production environment access
- **Two-Person Approval**: Production deployments require separate initiator and approver

Billing & Metering

- **Real-Time Tracking**: Every API call metered with token counts and costs

- **Dynamic Pricing**: Costs pulled from Dynamic Model Registry
- **Configurable Margin**: Default 20% markup on provider costs
- **Stripe Integration**: Automatic billing and payment processing
- **Usage Rollups**: Daily aggregation for efficient querying

---

LAMBDA DIRECTORY STRUCTURE

“‘ packages/infrastructure/lambda/ Ã¢€ Å“Ã¢€ â‚¬Ã¢€ â„¢ admin/ Ã¢€ â€šÃ¢€ Å“Ã¢€ â‚¬Ã¢€ â„¢ invitations.ts # Email invitations Ã¢€ â€šÃ¢€ Å“Ã¢€ â‚¬Ã¢€ â„¢ approvals.ts # Two-person approval workflow Ã¢€ â€šÃ¢€ Å“Ã¢€ â‚¬Ã¢€ â„¢ users.ts # Admin user CRUD Ã¢€ â€šÃ¢€ Å“Ã¢€ â‚¬Ã¢€ â„¢ profiles.ts # Admin preferences Ã¢€ â€šÃ¢€ Å“Ã¢€ â‚¬Ã¢€ â„¢ notifications.ts # Notification management Ã¢€ â€šÃ¢€ â€ Ã¢€ â‚¬Ã¢€ â„¢ audit.ts # Audit log queries Ã¢€ Å“Ã¢€ â‚¬Ã¢€ â„¢ billing/ Ã¢€ â€šÃ¢€ Å“Ã¢€ â‚¬Ã¢€ â„¢ metering.ts # Usage event collection Ã¢€ â€šÃ¢€ Å“Ã¢€ â‚¬Ã¢€ â„¢ billing.ts # Cost aggregation & invoices Ã¢€ â€šÃ¢€ Å“Ã¢€ â‚¬Ã¢€ â„¢ pricing.ts # Dynamic pricing sync Ã¢€ â€šÃ¢€ â€ Ã¢€ â‚¬Ã¢€ â„¢ payments.ts # Stripe integration Ã¢€ â€ Ã¢€ â‚¬Ã¢€ â„¢ shared/ Ã¢€ â€ Ã¢€ â‚¬Ã¢€ â„¢ admin/ Ã¢€ Å“Ã¢€ â‚¬Ã¢€ â„¢ index.ts Ã¢€ Å“Ã¢€ â‚¬Ã¢€ â„¢ types.ts # Admin-specific types Ã¢€ Å“Ã¢€ â‚¬Ã¢€ â„¢ email.ts # SES email utilities Ã¢€ Å“Ã¢€ â‚¬Ã¢€ â„¢ tokens.ts # Secure token generation Ã¢€ â€ Ã¢€ â‚¬Ã¢€ â„¢ stripe.ts # Stripe client “‘

PART 1: SHARED ADMIN UTILITIES

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

“‘typescript // Re-export admin utilities export * from ‘./types’; export * from ‘./email’; export * from ‘./tokens’; export * from ‘./stripe’; “‘

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

“‘typescript /** * Admin-specific types for RADIANT v2.2.0 */

import { z } from ‘zod’;

// ==
// ADMINISTRATOR ROLES // ==

export const AdminRole = { SUPER_ADMIN: ‘super_admin’, ADMIN: ‘admin’, OPERATOR: ‘operator’, AUDITOR: ‘auditor’, } as const;

export type AdminRoleType = typeof AdminRole[keyof typeof AdminRole];

export const ROLE_HIERARCHY: Record<AdminRoleType, number> = { [AdminRole.SUPER_ADMIN]: 100, [AdminRole.ADMIN]: 75, [AdminRole.OPERATOR]: 50, [AdminRole.AUDITOR]: 25, };

export const ROLE_PERMISSIONS: Record<AdminRoleType, string[]> = { [AdminRole.SUPER_ADMIN]: [‘admin:’, ’billing:’, ‘settings:’, ’deployments:’, ‘approvals:’, ’audit:’,], [AdminRole.ADMIN]: [‘admin:read’, ‘admin:write’, ‘billing:read’, ‘settings:read’, ‘settings:write’, ‘deployments:read’, ‘deployments:write’, ‘approvals:read’, ‘approvals:initiate’,],

[AdminRole.OPERATOR]: [‘admin:read’, ‘billing:read’, ‘settings:read’, ‘deployments:read’,], [AdminRole.AUDITOR]: [‘admin:read’, ‘billing:read’, ‘audit:read’,], };

// ========== // INVITATION TYPES // ==========

export interface Invitation { id: string; email: string; role: AdminRoleType; invitedBy: string; invitedByName: string; appId: string; tenantId: string; environment: ‘dev’ | ‘staging’ | ‘prod’; token: string; tokenHash: string; expiresAt: string; status: ‘pending’ | ‘accepted’ | ‘expired’ | ‘revoked’; createdAt: string; acceptedAt?: string; acceptedByIp?: string; }

export const createInvitationSchema = z.object({ email: z.string().email(), role: z.enum([‘super_admin’, ‘admin’, ‘operator’, ‘auditor’]), message: z.string().max(500).optional(), expiresInHours: z.number().int().min(1).max(168).default(48), });

export const acceptInvitationSchema = z.object({ token: z.string().min(32), firstName: z.string().min(1).max(50), lastName: z.string().min(1).max(50), password: z.string().min(12).max(128), mfaMethod: z.enum([‘authenticator’, ‘sms’, ‘email’]).default(‘authenticator’), phone: z.string().optional(), });

// ========== // APPROVAL TYPES // ==========

export type ApprovalType = | ‘deployment’ | ‘promotion’ | ‘model_activation’ | ‘provider_change’ | ‘user_role_change’ | ‘billing_change’;

export type ApprovalStatus = | ‘pending’ | ‘approved’ | ‘rejected’ | ‘expired’ | ‘cancelled’;

export interface ApprovalRequest { id: string; type: ApprovalType; appId: string; tenantId: string; environment: ‘dev’ | ‘staging’ | ‘prod’; requestedBy: string; requestedByName: string; requestedAt: string; expiresAt: string; status: ApprovalStatus; approvedBy?: string; approvedByName?: string; approvedAt?: string; rejectedReason?: string; action: string; resourceType: string; resourceId: string; details: Record<string, unknown>; priority: ‘low’ | ‘medium’ | ‘high’ | ‘critical’; notes?: string; }

export const createApprovalSchema = z.object({ type: z.enum([‘deployment’, ‘promotion’, ‘model_activation’, ‘provider_change’, ‘user_role_change’, ‘billing_change’,]), action: z.string(), resourceType: z.string(), resourceId: z.string(), details: z.record(z.unknown()), priority: z.enum([‘low’, ‘medium’, ‘high’, ‘critical’]).default(‘medium’), notes: z.string().max(1000).optional(), expiresInHours: z.number().int().min(1).max(168).default(24), });

export const processApprovalSchema = z.object({ action: z.enum([‘approve’, ‘reject’]), reason: z.string().max(500).optional(), }); “‘

packages/infrastructure/lambda/shared/admin/tokens.ts

“‘typescript /** * Secure token generation and validation */

import { randomBytes, createHash, timingSafeEqual } from ‘crypto’;

export function generateToken(length: number = 32): string { return randomBytes(length).toString(‘base64url’); }

export function generateCode(length: number = 6): string { const chars = ‘0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ’; const bytes = randomBytes(length); let code = ’’; for (let i = 0; i < length; i++) { code += chars[bytes[i] % chars.length]; } return code; }

export function hashToken(token: string): string { return createHash(‘sha256’).update(token).digest(‘hex’); }

export function verifyToken(token: string, hash: string): boolean { const tokenHash = hashToken(token); try { return timingSafeEqual(Buffer.from(tokenHash, ‘hex’), Buffer.from(hash, ‘hex’)); } catch { return false; } }

export function generateInvitationToken(): { token: string; tokenHash: string } { const token = generateToken(48); const tokenHash = hashToken(token); return { token, tokenHash }; }

export function calculateExpiry(hoursFromNow: number): string { const expiry = new Date(); expiry.setHours(expiry.getHours() + hoursFromNow); return expiry.toISOString(); }

export function isExpired(expiresAt: string): boolean { return new Date(expiresAt) < new Date(); } “‘

packages/infrastructure/lambda/shared/admin/email.ts

“‘typescript /** * Email utilities using AWS SES */

import { SESClient, SendEmailCommand } from ‘@aws-sdk/client-ses’; import { Logger } from ‘../logger’;

const sesClient = new SESClient({});

export interface EmailOptions { to: string | string[]; subject: string; html: string; text?: string; replyTo?: string; }

export async function sendEmail(options: EmailOptions, logger: Logger): Promise { const fromEmail = ‘noreply@${process.env.DOMAIN || ‘radiant.cloud’}‘; const toAddresses = Array.isArray(options.to) ? options.to : [options.to];

const command = new SendEmailCommand({ Source: fromEmail, Destination: { ToAddresses: toAddresses }, Message: { Subject: { Data: options.subject, Charset: ‘UTF-8’ }, Body: { Html: { Data: options.html, Charset: ‘UTF-8’ }, …(options.text && { Text: { Data: options.text, Charset: ‘UTF-8’ } }), }, }, ReplyToAddresses: options.replyTo ? [options.replyTo] : undefined, });

try { await sesClient.send(command); logger.info(‘Email sent successfully’, { to: toAddresses, subject: options.subject }); } catch (error) { logger.error(‘Failed to send email’, error as Error, { to: toAddresses }); throw error; } }

export function generateInvitationEmail(params: { inviteeName: string; inviterName: string; role: string; appName: string; environment: string; acceptUrl: string; expiresAt: string; message?: string; }): { html: string; text: string } { const html = ‘ <!DOCTYPE html>

You’ve Been Invited to ${params.appName}

```
<h1 style="color: #1a1a1a; margin-bottom: 20px;">You're Invited!</h1>
<p><strong>\${params.inviterName}</strong> has invited you to join <strong>\${params.appName}</
\${params.message ? \`<blockquote style="background: #f8f9fa; padding: 15px; border-left: 4px s
<div style="text-align: center; margin: 30px 0;">
```

```
  <a href="\${params.acceptUrl}" style="display: inline-block; padding: 14px 32px; background:
</div>
<p style="color: #666; font-size: 14px;">This invitation expires on <strong>\${new Date(params
```

‘;

const text = ‘You’ve been invited to ${params.appName} by ${params.inviterName} as a ${params.role}. Accept: ${params.acceptUrl}‘; return { html, text }; }

export function generateApprovalEmail(params: { approverName: string; requesterName: string; appName: string; environment: string; action: string; resourceType: string; resourceId: string; approveUrl: string; rejectUrl: string; expiresAt: string; }): { html: string; text: string } { const html = ‘ <!DOCTYPE html>

Approval Required - ${params.appName}

```
<div style="background: #ffc107; padding: 20px; text-align: center;">
  <h1 style="margin: 0; color: #1a1a1a;">Ã¢Å¡Ã Ã¯Ã¸Ã  Approval Required</h1>
</div>
<div style="padding: 30px;">
  <p>Hi \${params.approverName},</p>
  <p><strong>\${params.requesterName}</strong> has requested approval for:</p>
  <div style="background: #f8f9fa; padding: 20px; border-radius: 6px; margin: 20px 0;">
    <p><strong>Environment:</strong> \${params.environment.toUpperCase()}</p>
    <p><strong>Action:</strong> \${params.action}</p>
    <p><strong>Resource:</strong> \${params.resourceType} (\${params.resourceId})</p>
  </div>
  <div style="text-align: center; margin: 30px 0;">
    <a href="\${params.approveUrl}" style="display: inline-block; padding: 14px 32px; backgrou
    <a href="\${params.rejectUrl}" style="display: inline-block; padding: 14px 32px; backgroun
  </div>
  <p style="color: #dc3545; font-size: 12px; text-align: center;">Ã¢Å¡Ã Ã¯Ã¸Ã  Production deplo
</div>
```

‘;

const text = ‘APPROVAL REQUIRED: ${params.requesterName} requests approval for ${params.action} on ${params.resourceType}. Approve: ${params.approveUrl} Reject: ${params.rejectUrl}‘; return { html, text }; } “‘

packages/infrastructure/lambda/shared/admin/stripe.ts

“‘typescript /** * Stripe payment integration */

import Stripe from ‘stripe’; import { SecretsManagerClient, GetSecretValueCommand } from ‘@aws-sdk/client-secrets-manager’; import { Logger } from ‘../logger’;

const secretsClient = new SecretsManagerClient({}); let stripeClient: Stripe | null = null;

async function getStripeClient(): Promise { if (stripeClient) return stripeClient;

const secretArn = process.env.STRIPE_SECRET_ARN; if (!secretArn) throw new Error(‘STRIPE_SECRET_ARN not configured’);

const command = new GetSecretValueCommand({ SecretId: secretArn }); const response = await secretsClient.send(command); if (!response.SecretString) throw new Error('Stripe secret not found');

const secrets = JSON.parse(response.SecretString); stripeClient = new Stripe(secrets.apiKey, { apiVersion: '2023-10-16' }); return stripeClient; }

export async function getOrCreateCustomer(tenantId: string, email: string, name: string, metadata: Record<string, string>, logger: Logger): Promise { const stripe = await getStripeClient();

const existing = await stripe.customers.search({ query: 'metadata['tenantId']:'${tenantId}'', limit: 1, });

if (existing.data.length > 0) { logger.info('Found existing Stripe customer', { tenantId, customerId: existing.data[0].id }); return existing.data[0].id; }

const customer = await stripe.customers.create({ email, name, metadata: { tenantId, ...metadata }, });

logger.info('Created Stripe customer', { tenantId, customerId: customer.id }); return customer.id; }

export async function createInvoice(params: { customerId: string; tenantId: string; periodStart: Date; periodEnd: Date; lineItems: Array<{ description: string; amount: number; quantity: number; metadata?: Record<string, string>; }>; metadata?: Record<string, string>; }, logger: Logger): Promise<Stripe.Invoice> { const stripe = await getStripeClient();

for (const item of params.lineItems) { await stripe.invoiceItems.create({ customer: params.customerId, amount: item.amount, currency: 'usd', description: item.description, quantity: item.quantity, metadata: item.metadata, }); }

const invoice = await stripe.invoices.create({ customer: params.customerId, auto_advance: true, collection_method: 'charge_automatically', metadata: { tenantId: params.tenantId, periodStart: params.periodStart.toISOString(), periodEnd: params.periodEnd.toISOString(), ...params.metadata, }, });

logger.info('Created Stripe invoice', { tenantId: params.tenantId, invoiceId: invoice.id }); return invoice; }

export async function getInvoiceStatus(invoiceId: string, logger: Logger): Promise<{ status: string; paid: boolean; amountDue: number; amountPaid: number; }> { const stripe = await getStripeClient(); const invoice = await stripe.invoices.retrieve(invoiceId); return { status: invoice.status || 'unknown', paid: invoice.paid, amountDue: invoice.amount_due, amountPaid: invoice.amount_paid, }; } "'

PART 2: INVITATIONS LAMBDA

packages/infrastructure/lambda/admin/invitations.ts

"'typescript /** * Administrator Invitation Lambda * Handles email-based administrator invitations with secure tokens */

import { APIGatewayProxyEvent, APIGatewayProxyResult, Context } from ‘aws-lambda’; import { v4 as uuidv4 } from ‘uuid’; import { Logger } from ‘../shared/logger’; import { success, created, handleError } from ‘../shared/response’; import { extractAuthContext, requireAdmin, requirePermission } from ‘../shared/auth’; import { ValidationError, NotFoundError, ForbiddenError } from ‘../shared/errors’; import { executeQuery, createAuditLog } from ‘../shared/db’; import { createInvitationSchema, acceptInvitationSchema, ROLE_HIERARCHY } from ‘../shared/admin/types’; import { generateInvitationToken, hashToken, calculateExpiry, isExpired } from ‘../shared/admin/tokens’; import { sendEmail, generateInvitationEmail } from ‘../shared/admin/email’; import { CognitoIdentityProviderClient, AdminCreateUserCommand, AdminAddUserToGroupCommand, AdminSetUserMFAPreferenceCommand, } from ‘@aws-sdk/client-cognito-identity-provider’;

const logger = new Logger({ handler: ‘invitations’ }); const cognitoClient = new CognitoIdentityProviderClient({});

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

try { const invitationId = event.pathParameters?.invitationId; const action = event.path.split(‘/’).pop();

```
switch (event.httpMethod) {
  case 'GET':
    if (invitationId) return await handleGetInvitation(invitationId, event, requestLogger);
    return await handleListInvitations(event, requestLogger);

  case 'POST':
    if (action === 'accept') return await handleAcceptInvitation(event, requestLogger);
    if (action === 'resend' && invitationId) return await handleResendInvitation(invitationId,
    return await handleCreateInvitation(event, requestLogger);

  case 'DELETE':
    if (!invitationId) throw new ValidationError('Invitation ID required');
    return await handleRevokeInvitation(invitationId, event, requestLogger);

  default:
    throw new ValidationError(\`Method \${event.httpMethod} not allowed\`);
}
```

} catch (error) { return handleError(error, requestLogger); } }

async function handleCreateInvitation(event: APIGatewayProxyEvent, logger: Logger): Promise { const auth = extractAuthContext(event); requireAdmin(auth); requirePermission(auth, ‘admin:write’);

const body = event.body ? JSON.parse(event.body) : {}; const parseResult = createInvitationSchema.safeParse(body); if (!parseResult.success) { throw new ValidationError(‘Invalid request body’, parseResult.error.flatten().fieldErrors as Record<string, string[]>); }

const { email, role, message, expiresInHours } = parseResult.data;

// Validate role hierarchy if (ROLE_HIERARCHY[role] > ROLE_HIERARCHY[auth.role as keyof typeof ROLE_HIERARCHY]) { throw new ForbiddenError(‘Cannot invite administrator with

higher role than your own'); }

// Check existing admin/invitation const existingAdmin = await executeQuery('SELECT id FROM administrators WHERE email = $1 AND tenant_id = $2', [email, auth.tenantId], logger); if (existingAdmin.rowCount > 0) { throw new ValidationError('An administrator with this email already exists'); }

const pendingInvitation = await executeQuery('SELECT id FROM invitations WHERE email = $1 AND tenant_id = $2 AND status = 'pending' AND expires_at > NOW()', [email, auth.tenantId], logger); if (pendingInvitation.rowCount > 0) { throw new ValidationError('A pending invitation already exists for this email'); }

// Generate token and create invitation const { token, tokenHash } = generateInvitationToken(); const expiresAt = calculateExpiry(expiresInHours); const invitationId = uuidv4();

const inviterResult = await executeQuery('SELECT first_name, last_name FROM administrators WHERE id = $1', [auth.userId], logger); const inviterName = inviterResult.rows[0] ? '${inviterResult.rows[0].first_name} ${inviterResult.rows[0].last_name}' : 'An administrator';

const appResult = await executeQuery('SELECT name FROM apps WHERE id = $1', [auth.appId], logger); const appName = appResult.rows[0]?.name || auth.appId;

await executeQuery('INSERT INTO invitations (id, email, role, invited_by, app_id, tenant_id, environment, token_hash, expires_at, status, message, created_at) VALUES ($1, $2, $3, $4, $5, $6, $7, $8, $9, 'pending', $10, NOW())', [invitationId, email, role, auth.userId, auth.appId, auth.tenantId, auth.environment, tokenHash, expiresAt, message || null], logger);

// Send email const acceptUrl = '${process.env.ADMIN_URL}/invite/accept?token=${token}'; const emailContent = generateInvitationEmail({ inviteeName: '', inviterName, role, appName, environment: auth.environment, acceptUrl, expiresAt, message, }); await sendEmail({ to: email, subject: 'You've been invited to ${appName}', html: emailContent.html, text: emailContent.text }, logger);

await createAuditLog({ tenant_id: auth.tenantId, user_id: null, admin_id: auth.userId, action: 'invitation.create', resource_type: 'invitation', resource_id: invitationId, details: { email, role, expiresAt }, ip_address: event.requestContext.identity?.sourceIp || null, user_agent: event.headers['User-Agent'] || null, }, logger);

logger.info('Invitation created and sent', { invitationId, email, role }); return created({ invitation: { id: invitationId, email, role, status: 'pending', expiresAt, createdAt: new Date().toISOString() } }); }

async function handleAcceptInvitation(event: APIGatewayProxyEvent, logger: Logger): Promise { const body = event.body ? JSON.parse(event.body) : {}; const parseResult = acceptInvitationSchema.safeParse(body); if (!parseResult.success) { throw new ValidationError('Invalid request body', parseResult.error.flatten().fieldErrors as Record<string, string[]>); }

const { token, firstName, lastName, password, mfaMethod, phone } = parseResult.data;

const tokenHash = hashToken(token); const inviteResult = await executeQuery('SELECT * FROM invitations WHERE token_hash = $1 AND status = 'pending'', [tokenHash], logger); if (inviteResult.rowCount === 0) throw new NotFoundError('Invalid or expired invitation');

const invitation = inviteResult.rows[0]; if (isExpired(invitation.expires_at)) { await executeQuery(‘UPDATE invitations SET status = ‘expired’ WHERE id = $1‘, [invitation.id], logger); throw new ValidationError(‘This invitation has expired’); }

if (invitation.environment === ‘prod’ && mfaMethod === ‘sms’ && !phone) { throw new ValidationError(‘Phone number required for SMS MFA’); }

const userPoolId = process.env.ADMIN_USER_POOL_ID; const adminUserId = uuidv4();

// Create Cognito user await cognitoClient.send(new AdminCreateUserCommand({ UserPoolId: userPoolId, Username: invitation.email, TemporaryPassword: password, UserAttributes: [{ Name: ‘email’, Value: invitation.email }, { Name: ‘email_verified’, Value: ‘true’ }, { Name: ‘given_name’, Value: firstName }, { Name: ‘family_name’, Value: lastName }, { Name: ‘custom:adminId’, Value: adminUserId }, { Name: ‘custom:tenantId’, Value: invitation.tenant_id }, { Name: ‘custom:role’, Value: invitation.role },], MessageAction: ‘SUPPRESS’, }));

await cognitoClient.send(new AdminAddUserToGroupCommand({ UserPoolId: userPoolId, Username: invitation.email, GroupName: invitation.role, }));

if (invitation.environment === ‘prod’) { await cognitoClient.send(new AdminSetUserMFAPreferenceCommand({ UserPoolId: userPoolId, Username: invitation.email, SoftwareTokenMfaSettings: mfaMethod === ‘authenticator’ ? { Enabled: true, PreferredMfa: true } : undefined, SMSMfaSettings: mfaMethod === ‘sms’ ? { Enabled: true, PreferredMfa: true } : undefined, })); }

// Create admin user record await executeQuery(‘INSERT INTO administrators (id, cognito_user_id, email, first_name, last_name, display_name, role, app_id, tenant_id, mfa_enabled, mfa_method, status, created_at, updated_at, created_by, invitation_id) VALUES ($1, $2, $3, $4, $5, $6, $7, $8, $9, $10, $11, ‘active’, NOW(), NOW(), $12, $13)‘, [adminUserId, invitation.email, invitation.email, firstName, lastName, ‘${firstName} ${lastName}‘, invitation.role, invitation.app_id, invitation.tenant_id, invitation.environment === ‘prod’, mfaMethod, invitation.invited_by, invitation.id], logger);

// Create default profile await executeQuery(‘INSERT INTO admin_profiles (admin_id, notifications, timezone, language, date_format, time_format, currency, theme, default_environment, sidebar_collapsed, table_rows_per_page, updated_at) VALUES ($1, $2, ‘America/New_York’, ‘en’, ‘MM/DD/YYYY’, ‘12h’, ‘USD’, ‘system’, $3, false, 25, NOW())‘, [adminUserId, JSON.stringify({ method: ‘email’, frequency: ‘immediate’, categories: { security: true, billing: true, deployments: true, approvals: true, system: true } }), invitation.environment], logger);

await executeQuery(‘UPDATE invitations SET status = ‘accepted’, accepted_at = NOW(), accepted_by_ip = $2 WHERE id = $1‘, [invitation.id, event.requestContext.identity?.sourceIp || null], logger);

await createAuditLog({ tenant_id: invitation.tenant_id, user_id: null, admin_id: adminUserId, action: ‘invitation.accept’, resource_type: ‘invitation’, resource_id: invitation.id, details: { email: invitation.email, role: invitation.role, firstName, lastName }, ip_address: event.requestContext.identity?.sourceIp || null, user_agent: event.headers[‘User-Agent’] || null, }, logger);

logger.info(‘Invitation accepted’, { invitationId: invitation.id, adminUserId, email: invitation.email }); return success({ message: ‘Invitation accepted successfully’, adminUser: { id:

adminUserId, email: invitation.email, firstName, lastName, role: invitation.role, mfaRequired: invitation.environment === 'prod' }, }); }

async function handleListInvitations(event: APIGatewayProxyEvent, logger: Logger): Promise { const auth = extractAuthContext(event); requireAdmin(auth); requirePermission(auth, 'admin:read');

const status = event.queryStringParameters?.status; const limit = parseInt(event.queryStringParameters?.limit || '50'); const offset = parseInt(event.queryStringParameters?.offset || '0');

let query = 'SELECT i.*, a.first_name as inviter_first_name, a.last_name as inviter_last_name FROM invitations i LEFT JOIN administrators a ON i.invited_by = a.id WHERE i.tenant_id = $1'; const params: any[] = [auth.tenantId];

if (status) { params.push(status); query += ' AND i.status = $${params.length}'; } query += ' ORDER BY i.created_at DESC LIMIT $${params.length + 1} OFFSET $${params.length + 2}'; params.push(limit, offset);

const result = await executeQuery(query, params, logger); const invitations = result.rows.map(row => ({ id: row.id, email: row.email, role: row.role, status: row.status, environment: row.environment, invitedBy: { id: row.invited_by, name: row.inviter_first_name ? '${row.inviter_first_name} ${row.inviter_last_name}' : 'Unknown' }, message: row.message, expiresAt: row.expires_at, createdAt: row.created_at, acceptedAt: row.accepted_at, }));

return success({ invitations, pagination: { limit, offset, hasMore: invitations.length === limit } }); }

async function handleGetInvitation(invitationId: string, event: APIGatewayProxyEvent, logger: Logger): Promise { const auth = extractAuthContext(event); requireAdmin(auth); requirePermission(auth, 'admin:read');

const result = await executeQuery('SELECT i.*, a.first_name as inviter_first_name, a.last_name as inviter_last_name FROM invitations i LEFT JOIN administrators a ON i.invited_by = a.id WHERE i.id = $1 AND i.tenant_id = $2', [invitationId, auth.tenantId], logger); if (result.rowCount === 0) throw new NotFoundError('Invitation not found');

const row = result.rows[0]; return success({ invitation: { id: row.id, email: row.email, role: row.role, status: row.status, environment: row.environment, invitedBy: { id: row.invited_by, name: '${row.inviter_first_name} ${row.inviter_last_name}' }, message: row.message, expiresAt: row.expires_at, createdAt: row.created_at, acceptedAt: row.accepted_at, }, }); }

async function handleResendInvitation(invitationId: string, event: APIGatewayProxyEvent, logger: Logger): Promise { const auth = extractAuthContext(event); requireAdmin(auth); requirePermission(auth, 'admin:write');

const result = await executeQuery('SELECT * FROM invitations WHERE id = $1 AND tenant_id = $2', [invitationId, auth.tenantId], logger); if (result.rowCount === 0) throw new NotFoundError('Invitation not found');

const invitation = result.rows[0]; if (invitation.status !== 'pending') throw new ValidationError('Cannot resend ${invitation.status} invitation');

const { token, tokenHash } = generateInvitationToken(); const expiresAt = calculateExpiry(48);

await executeQuery(‘UPDATE invitations SET token_hash = $2, expires_at = $3 WHERE id = $1‘, [invitationId, tokenHash, expiresAt], logger);

const inviterResult = await executeQuery(‘SELECT first_name, last_name FROM administrators WHERE id = $1‘, [auth.userId], logger); const inviterName = inviterResult.rows[0] ? ‘${inviterResult.rows[0].first_name} ${inviterResult.rows[0].last_name}‘ : ‘An administrator’;

const appResult = await executeQuery(‘SELECT name FROM apps WHERE id = $1‘, [auth.appId], logger); const appName = appResult.rows[0]?.name || auth.appId;

const acceptUrl = ‘${process.env.ADMIN_URL}/invite/accept?token=${token}‘; const emailContent = generateInvitationEmail({ inviteeName: ’’, inviterName, role: invitation.role, appName, environment: invitation.environment, acceptUrl, expiresAt, message: invitation.message }); await sendEmail({ to: invitation.email, subject: ‘Reminder: You’ve been invited to ${appName}‘, html: emailContent.html, text: emailContent.text }, logger);

logger.info(‘Invitation resent’, { invitationId, email: invitation.email }); return success({ message: ‘Invitation resent successfully’, expiresAt }); }

async function handleRevokeInvitation(invitationId: string, event: APIGatewayProxyEvent, logger: Logger): Promise { const auth = extractAuthContext(event); requireAdmin(auth); requirePermission(auth, ‘admin:write’);

const result = await executeQuery(‘SELECT * FROM invitations WHERE id = $1 AND tenant_id = $2‘, [invitationId, auth.tenantId], logger); if (result.rowCount === 0) throw new NotFoundError(‘Invitation not found’);

const invitation = result.rows[0]; if (invitation.status !== ‘pending’) throw new ValidationError(‘Cannot revoke ${invitation.status} invitation‘);

await executeQuery(‘UPDATE invitations SET status = ‘revoked’ WHERE id = $1‘, [invitationId], logger);

await createAuditLog({ tenant_id: auth.tenantId, user_id: null, admin_id: auth.userId, action: ‘invitation.revoke’, resource_type: ‘invitation’, resource_id: invitationId, details: { email: invitation.email }, ip_address: event.requestContext.identity?.sourceIp || null, user_agent: event.headers[‘User-Agent’] || null, }, logger);

logger.info(‘Invitation revoked’, { invitationId, email: invitation.email }); return success({ message: ‘Invitation revoked successfully’ }); } “‘

PART 3: TWO-PERSON APPROVALS LAMBDA

packages/infrastructure/lambda/admin/approvals.ts

“‘typescript /** * Two-Person Approval Workflow Lambda * Production deployments require separate initiator and approver */

import { APIGatewayProxyEvent, APIGatewayProxyResult, Context } from ‘aws-lambda’; import { v4 as uuidv4 } from ‘uuid’; import { Logger } from ‘../shared/logger’; import { success, created, handleError } from ‘../shared/response’; import { extractAuthContext, requireAdmin, requirePermission } from ‘../shared/auth’; import { ValidationError, NotFoundError, ForbiddenError } from

'../shared/errors'; import { executeQuery, createAuditLog } from '../shared/db'; import { createApprovalSchema, processApprovalSchema, ApprovalStatus, AdminRole, ROLE_HIERARCHY } from '../shared/admin/types'; import { calculateExpiry, isExpired } from '../shared/admin/tokens'; import { sendEmail, generateApprovalEmail } from '../shared/admin/email';

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

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

try { const auth = extractAuthContext(event); requireAdmin(auth);

```
const approvalId = event.pathParameters?.approvalId;
const action = event.path.split('/').pop();

switch (event.httpMethod) {
  case 'GET':
    if (approvalId) return await handleGetApproval(approvalId, auth, requestLogger);
    return await handleListApprovals(event, auth, requestLogger);
  case 'POST':
    if (action === 'process' && approvalId) return await handleProcessApproval(approvalId, eve
    return await handleCreateApproval(event, auth, requestLogger);
  case 'DELETE':
    if (!approvalId) throw new ValidationError('Approval ID required');
    return await handleCancelApproval(approvalId, event, auth, requestLogger);
  default:
    throw new ValidationError(\`Method \${event.httpMethod} not allowed\`);
}
```

} catch (error) { return handleError(error, requestLogger); } }

async function handleCreateApproval(event: APIGatewayProxyEvent, auth: ReturnType, logger: Logger): Promise { requirePermission(auth, 'approvals:initiate');

const body = event.body ? JSON.parse(event.body) : {}; const parseResult = createApprovalSchema.safeParse(body); if (!parseResult.success) throw new ValidationError('Invalid request body', parseResult.error.flatten().fieldErrors as Record<string, string[]>);

const { type, action, resourceType, resourceId, details, priority, notes, expiresInHours } = parseResult.data; const requiresTwoPersonApproval = auth.environment === 'prod';

const requesterResult = await executeQuery('SELECT first_name, last_name FROM administrators WHERE id = $1', [auth.userId], logger); const requesterName = requesterResult.rows[0] ? '${requesterResult.rows[0].first_name} ${requesterResult.rows[0].last_name}' : 'Unknown';

const approvalId = uuidv4(); const expiresAt = calculateExpiry(expiresInHours);

await executeQuery('INSERT INTO approval_requests (id, type, app_id, tenant_id, environment, requested_by, requested_at, expires_at, status, action, resource_type, resource_id, details, priority, notes, requires_two_person, created_at) VALUES ($1, $2, $3, $4, $5, $6, NOW(), $7, 'pending', $8, $9, $10, $11, $12, $13, $14, NOW())', [approvalId, type, auth.appId, auth.tenantId, auth.environment, auth.userId, expiresAt, action, resourceType, resourceId, JSON.stringify(details), priority, notes || null, requiresTwoPersonApproval], logger);

if (requiresTwoPersonApproval) { await notifyApprovers(approvalId, auth, requesterName, logger); }

await createAuditLog({ tenant_id: auth.tenantId, user_id: null, admin_id: auth.userId, action: 'approval.create', resource_type: 'approval_request', resource_id: approvalId, details: { type, resourceType, resourceId, requiresTwoPersonApproval }, ip_address: event.requestContext.identity?.sourceIp || null, user_agent: event.headers['User-Agent'] || null, }, logger);

logger.info('Approval request created', { approvalId, type, requestedBy: auth.userId, requiresTwoPersonApproval }); return created({ approval: { id: approvalId, type, status: 'pending', action, resourceType, resourceId, priority, expiresAt, requiresTwoPersonApproval, createdAt: new Date().toISOString() }, }); }

async function handleProcessApproval(approvalId: string, event: APIGatewayProxyEvent, auth: ReturnType, logger: Logger): Promise { requirePermission(auth, 'approvals:*');

const body = event.body ? JSON.parse(event.body) : {}; const parseResult = processApprovalSchema.safeParse(body); if (!parseResult.success) throw new ValidationError('Invalid request body', parseResult.error.flatten().fieldErrors as Record<string, string[]>);

const { action, reason } = parseResult.data;

const result = await executeQuery('SELECT * FROM approval_requests WHERE id = $1 AND tenant_id = $2', [approvalId, auth.tenantId], logger); if (result.rowCount === 0) throw new NotFoundError('Approval request not found');

const approval = result.rows[0]; if (approval.status !== 'pending') throw new ValidationError('Cannot process ${approval.status} approval request'); if (isExpired(approval.expires_at)) { await executeQuery('UPDATE approval_requests SET status = 'expired' WHERE id = $1', [approvalId], logger); throw new ValidationError('This approval request has expired'); }

// Two-person approval: cannot approve own request if (approval.requires_two_person && approval.requested_by === auth.userId) { throw new ForbiddenError('You cannot approve your own request. Production deployments require approval from a different administrator.'); }

const newStatus: ApprovalStatus = action === 'approve' ? 'approved' : 'rejected'; await executeQuery('UPDATE approval_requests SET status = $2, approved_by = $3, approved_at = NOW(), rejected_reason = $4 WHERE id = $1', [approvalId, newStatus, auth.userId, action === 'reject' ? reason : null], logger);

if (action === 'approve') { await executeApprovedAction(approval, logger); }

await createAuditLog({ tenant_id: auth.tenantId, user_id: null, admin_id: auth.userId, action: 'approval.${action}', resource_type: 'approval_request', resource_id: approvalId, details: { type: approval.type, resourceType: approval.resource_type, resourceId: approval.resource_id, reason }, ip_address: event.requestContext.identity?.sourceIp || null, user_agent: event.headers['User-Agent'] || null, }, logger);

logger.info('Approval request processed', { approvalId, action, processedBy: auth.userId }); return success({ approval: { id: approvalId, status: newStatus, processedBy: auth.userId, processedAt: new Date().toISOString(), reason: action === 'reject' ? reason : undefined } }); }

async function executeApprovedAction(approval: any, logger: Logger): Promise { logger.info('Executing approved action', { type: approval.type, resourceType: approval.resource_type,

resourceId: approval.resource_id });

switch (approval.type) { case 'deployment': // Trigger deployment workflow via Step Functions or CodePipeline break; case 'promotion': await executeQuery('UPDATE deployments SET promoted_to = $2, promoted_at = NOW() WHERE id = $1', [approval.resource_id, approval.details?.targetEnv], logger); break; case 'model_activation': await executeQuery('UPDATE ai_models SET status = $2, thermal_state = $3, updated_at = NOW() WHERE id = $1', [approval.details?.modelId, approval.details?.newStatus, approval.details?.thermalState], logger); break; case 'provider_change': await executeQuery('UPDATE ai_providers SET config = $2, updated_at = NOW() WHERE id = $1', [approval.details?.providerId, JSON.stringify(approval.details?.config)], logger); break; case 'user_role_change': await executeQuery('UPDATE administrators SET role = $2, updated_at = NOW() WHERE id = $1', [approval.details?.userId, approval.details?.newRole], logger); break; case 'billing_change': await executeQuery('UPDATE billing_settings SET margin_percent = COALESCE($2, margin_percent), tax_percent = COALESCE($3, tax_percent), updated_at = NOW() WHERE tenant_id = $1', [approval.details?.tenantId, approval.details?.settings?.marginPercent, approval.details?.settings?.taxPercent], logger); break; default: logger.warn('Unknown approval type', { type: approval.type }); } }

async function notifyApprovers(approvalId: string, auth: ReturnType, requesterName: string, logger: Logger): Promise { const result = await executeQuery('SELECT id, email, first_name, last_name FROM administrators WHERE tenant_id = $1 AND id != $2 AND status = 'active' AND role IN ('super_admin', 'admin')', [auth.tenantId, auth.userId], logger); if (result.rowCount === 0) { logger.warn('No other admins available to approve'); return; }

const appResult = await executeQuery('SELECT name FROM apps WHERE id = $1', [auth.appId], logger); const appName = appResult.rows[0]?.name || auth.appId;

const approvalResult = await executeQuery('SELECT * FROM approval_requests WHERE id = $1', [approvalId], logger); const approval = approvalResult.rows[0];

for (const admin of result.rows) { const approveUrl = '${process.env.ADMIN_URL}/approvals/${approvalId}?acti
const rejectUrl = '${process.env.ADMIN_URL}/approvals/${approvalId}?action=reject'; const emailContent = generateApprovalEmail({ approverName: admin.first_name, requesterName, appName, environment: auth.environment, action: approval.action, resourceType: approval.resource_type, resourceId: approval.resource_id, approveUrl, rejectUrl, expiresAt: approval.expires_at, }); await sendEmail({ to: admin.email, subject: 'Ã¢Å¡Â Ã¯Â¸Â Approval Required: ${approval.action} - ${appName}', html: emailContent.html, text: emailContent.text }, logger); } logger.info('Approvers notified', { approvalId, notifiedCount: result.rowCount }); }

async function handleListApprovals(event: APIGatewayProxyEvent, auth: ReturnType, logger: Logger): Promise { requirePermission(auth, 'approvals:read');

const status = event.queryStringParameters?.status; const pendingForMe = event.queryStringParameters?.pending
=== 'true'; const limit = parseInt(event.queryStringParameters?.limit || '50'); const offset = parseInt(event.queryStringParameters?.offset || '0');

let query = 'SELECT ar.*, req.first_name as requester_first_name, req.last_name as requester_last_name FROM approval_requests ar LEFT JOIN administrators req ON ar.requested_by = req.id WHERE ar.tenant_id = $1'; const params: any[] = [auth.tenantId];

if (status) { params.push(status); query += ' AND ar.status = $${params.length}'; } if (pending-

ForMe) { params.push(auth.userId); query += ‘ AND ar.status = ‘pending’ AND ar.requested_by != $${params.length}‘; } query += ‘ ORDER BY ar.created_at DESC LIMIT $${params.length + 1} OFFSET $${params.length + 2}‘; params.push(limit, offset);

const result = await executeQuery(query, params, logger); const approvals = result.rows.map(row => ({ id: row.id, type: row.type, status: row.status, environment: row.environment, action: row.action, resourceType: row.resource_type, resourceId: row.resource_id, priority: row.priority, requestedBy: { id: row.requested_by, name: ‘${row.requester_first_name} ${row.requester_last_name}‘ }, requestedAt: row.requested_at, expiresAt: row.expires_at, requiresTwoPersonApproval: row.requires_two_person, canApprove: row.status === ‘pending’ && row.requested_by !== auth.userId, }));

return success({ approvals, pagination: { limit, offset, hasMore: approvals.length === limit } }); }

async function handleGetApproval(approvalId: string, auth: ReturnType, logger: Logger): Promise { requirePermission(auth, ‘approvals:read’);

const result = await executeQuery(‘SELECT ar.*, req.first_name as requester_first_name, req.last_name as requester_last_name, req.email as requester_email FROM approval_requests ar LEFT JOIN administrators req ON ar.requested_by = req.id WHERE ar.id = $1 AND ar.tenant_id = $2‘, [approvalId, auth.tenantId], logger); if (result.rowCount === 0) throw new NotFoundError(‘Approval request not found’);

const row = result.rows[0]; return success({ approval: { id: row.id, type: row.type, status: row.status, environment: row.environment, action: row.action, resourceType: row.resource_type, resourceId: row.resource_id, details: row.details, priority: row.priority, notes: row.notes, requestedBy: { id: row.requested_by, name: ‘${row.requester_first_name} ${row.requester_last_name}‘, email: row.requester_email }, requestedAt: row.requested_at, expiresAt: row.expires_at, requiresTwoPersonApproval: row.requires_two_person, canApprove: row.status === ‘pending’ && row.requested_by !== auth.userId, }, }); }

async function handleCancelApproval(approvalId: string, event: APIGatewayProxyEvent, auth: ReturnType, logger: Logger): Promise { const result = await executeQuery(‘SELECT * FROM approval_requests WHERE id = $1 AND tenant_id = $2‘, [approvalId, auth.tenantId], logger); if (result.rowCount === 0) throw new NotFoundError(‘Approval request not found’);

const approval = result.rows[0]; if (approval.requested_by !== auth.userId && auth.role !== AdminRole.SUPER_ADMIN) { throw new ForbiddenError(‘Only the requester or a super admin can cancel this request’); } if (approval.status !== ‘pending’) throw new ValidationError(‘Cannot cancel ${approval.status} approval request‘);

await executeQuery(‘UPDATE approval_requests SET status = ‘cancelled’ WHERE id = $1‘, [approvalId], logger); await createAuditLog({ tenant_id: auth.tenantId, user_id: null, admin_id: auth.userId, action: ‘approval.cancel’, resource_type: ‘approval_request’, resource_id: approvalId, details: { type: approval.type }, ip_address: event.requestContext.identity?.sourceIp || null, user_agent: event.headers[‘User-Agent’] || null, }, logger);

logger.info(‘Approval request cancelled’, { approvalId }); return success({ message: ‘Approval request cancelled’ }); } “‘

PART 4: METERING LAMBDA

packages/infrastructure/lambda/billing/metering.ts

“‘typescript /** * Usage Metering Lambda * Collects and stores usage events for billing calculations */

import { APIGatewayProxyEvent, APIGatewayProxyResult, Context } from ‘aws-lambda’; import { DynamoDBClient } from ‘@aws-sdk/client-dynamodb’; import { DynamoDBDocumentClient, PutCommand, QueryCommand, UpdateCommand } from ‘@aws-sdk/lib-dynamodb’; import { v4 as uuidv4 } from ‘uuid’; import { z } from ‘zod’; import { Logger } from ‘../shared/logger’; import { success, created, handleError } from ‘../shared/response’; import { extractAuthContext } from ‘../shared/auth’; import { ValidationError } from ‘../shared/errors’; import { executeQuery } from ‘../shared/db’;

const logger = new Logger({ handler: ‘metering’ }); const ddbClient = DynamoDBDocument-Client.from(new DynamoDBClient({}), { marshallOptions: { removeUndefinedValues: true } });

const USAGE_TABLE = process.env.USAGE_TABLE || ‘radiant-usage-events’; const ROLLUP_TABLE = process.env.ROLLUP_TABLE || ‘radiant-usage-rollups’;

const recordUsageSchema = z.object({ requestId: z.string(), providerId: z.string(), modelId: z.string(), modelName: z.string(), requestType: z.enum([‘chat’, ‘embedding’, ‘image’, ‘audio’, ‘video’]), inputTokens: z.number().int().min(0), outputTokens: z.number().int().min(0), latencyMs: z.number().int().min(0), cached: z.boolean().default(false), phiDetected: z.boolean().default(false), phiSanitized: z.boolean().default(false), userId: z.string().optional(), });

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

try { const auth = extractAuthContext(event); const action = event.path.split(‘/’).pop();

```
switch (event.httpMethod) {
  case 'POST':
    if (action === 'record') return await handleRecordUsage(event, auth, requestLogger);
    if (action === 'batch') return await handleBatchRecord(event, auth, requestLogger);
    break;
  case 'GET':
    if (action === 'summary') return await handleGetSummary(event, auth, requestLogger);
    if (action === 'rollups') return await handleGetRollups(event, auth, requestLogger);
    break;
}
throw new ValidationError(\`Unknown action: \${action}\`);
```

} catch (error) { return handleError(error, requestLogger); } }

async function handleRecordUsage(event: APIGatewayProxyEvent, auth: ReturnType, logger: Logger): Promise { const body = event.body ? JSON.parse(event.body) : {}; const parseResult = recordUsageSchema.safeParse(body); if (!parseResult.success) throw new ValidationError(‘Invalid usage data’, parseResult.error.flatten().fieldErrors as Record<string, string[]>);

const data = parseResult.data;

// Get pricing from model registry const pricing = await getModelPricing(data.modelId, logger);

const inputCost = (data.inputTokens / 1000000) * pricing.inputPricePerMillion; const outputCost = (data.outputTokens / 1000000) * pricing.outputPricePerMillion; const providerCost = inputCost + outputCost;

// Apply margin const marginPercent = await getTenantMargin(auth.tenantId, logger); const billedCost = providerCost * (1 + marginPercent / 100);

const usageEvent = { id: uuidv4(), timestamp: new Date().toISOString(), tenantId: auth.tenantId, userId: data.userId || auth.userId, adminId: auth.isAdmin ? auth.userId : undefined, appId: auth.appId, environment: auth.environment, providerId: data.providerId, modelId: data.modelId, modelName: data.modelName, requestType: data.requestType, inputTokens: data.inputTokens, outputTokens: data.outputTokens, totalTokens: data.inputTokens + data.outputTokens, providerCost, billedCost, currency: 'USD', requestId: data.requestId, latencyMs: data.latencyMs, cached: data.cached, phiDetected: data.phiDetected, phiSanitized: data.phiSanitized, };

await ddbClient.send(new PutCommand({ TableName: USAGE_TABLE, Item: { pk: 'TENANT#${auth.tenantId}', sk: 'EVENT#${usageEvent.timestamp}#${usageEvent.id}', ...usageEvent, ttl: Math.floor(Date.now() / 1000) + (90 * 24 * 60 * 60) }, }));

await updateDailyRollup(usageEvent, logger);

logger.info('Usage recorded', { eventId: usageEvent.id, modelId: data.modelId, tokens: usageEvent.totalTokens, cost: billedCost }); return created({ event: { id: usageEvent.id, billedCost, providerCost } }); }

async function handleBatchRecord(event: APIGatewayProxyEvent, auth: ReturnType, logger: Logger): Promise { const body = event.body ? JSON.parse(event.body) : {}; if (!Array.isArray(body.events) || body.events.length === 0) throw new ValidationError('events array is required'); if (body.events.length > 100) throw new ValidationError('Maximum 100 events per batch');

const results: Array<{ id: string; success: boolean; error?: string }> = [];

for (const eventData of body.events) { try { const parseResult = recordUsageSchema.safeParse(eventData); if (!parseResult.success) { results.push({ id: eventData.requestId || 'unknown', success: false, error: 'Invalid data' }); continue; }

```
  const data = parseResult.data;
  const pricing = await getModelPricing(data.modelId, logger);
  const providerCost = ((data.inputTokens / 1000000) * pricing.inputPricePerMillion) + ((data.o
  const marginPercent = await getTenantMargin(auth.tenantId, logger);
  const billedCost = providerCost * (1 + marginPercent / 100);

  const usageEvent = {
    id: uuidv4(), timestamp: new Date().toISOString(), tenantId: auth.tenantId, userId: data.us
    appId: auth.appId, environment: auth.environment, providerId: data.providerId, modelId: dat
    requestType: data.requestType, inputTokens: data.inputTokens, outputTokens: data.outputToke
    providerCost, billedCost, currency: 'USD', requestId: data.requestId, latencyMs: data.later
  };

  await ddbClient.send(new PutCommand({ TableName: USAGE_TABLE, Item: { pk: \`TENANT#\${auth.te
  await updateDailyRollup(usageEvent, logger);
```

```
  results.push({ id: usageEvent.id, success: true });
} catch (error: any) {
  results.push({ id: eventData.requestId || 'unknown', success: false, error: error.message })
}

}
```

logger.info(‘Batch usage recorded’, { total: body.events.length, successful: results.filter(r => r.success).length }); return success({ results, summary: { total: results.length, successful: results.filter(r => r.success).length, failed: results.filter(r => !r.success).length } }); }

async function handleGetSummary(event: APIGatewayProxyEvent, auth: ReturnType, logger: Logger): Promise { const startDate = event.queryStringParameters?.startDate || getDefaultStartDate(); const endDate = event.queryStringParameters?.endDate || getTodayDate();

const response = await ddbClient.send(new QueryCommand({ TableName: ROLLUP_TABLE, KeyConditionExpression: ‘pk = :pk AND sk BETWEEN :start AND :end’, ExpressionAttributeValues: { ‘:pk’: ‘TENANT#${auth.tenantId}‘, ‘:start’: ‘DATE#${startDate}‘, ‘:end’: ‘DATE#${endDate}#~‘ }, }));

const rollups = response.Items || []; const totals = rollups.reduce((acc, r) => ({ requests: acc.requests + (r.requestCount || 0), inputTokens: acc.inputTokens + (r.inputTokens || 0), outputTokens: acc.outputTokens + (r.outputTokens || 0), providerCost: acc.providerCost + (r.providerCost || 0), billedCost: acc.billedCost + (r.billedCost || 0), }), { requests: 0, inputTokens: 0, outputTokens: 0, providerCost: 0, billedCost: 0 });

return success({ period: { startDate, endDate }, totals }); }

async function handleGetRollups(event: APIGatewayProxyEvent, auth: ReturnType, logger: Logger): Promise { const startDate = event.queryStringParameters?.startDate || getDefaultStartDate(); const endDate = event.queryStringParameters?.endDate || getTodayDate();

const response = await ddbClient.send(new QueryCommand({ TableName: ROLLUP_TABLE, KeyConditionExpression: ‘pk = :pk AND sk BETWEEN :start AND :end’, ExpressionAttributeValues: { ‘:pk’: ‘TENANT#${auth.tenantId}‘, ‘:start’: ‘DATE#${startDate}‘, ‘:end’: ‘DATE#${endDate}#~‘ }, }));

const rollups = (response.Items || []).map(item => ({ date: item.date, modelId: item.modelId, providerId: item.providerId, requestCount: item.requestCount, inputTokens: item.inputTokens, outputTokens: item.outputTokens, totalTokens: item.totalTokens, providerCost: item.providerCost, billedCost: item.billedCost, avgLatencyMs: item.avgLatencyMs, }));

return success({ rollups, period: { startDate, endDate } }); }

async function updateDailyRollup(event: any, logger: Logger): Promise { const date = event.timestamp.split(‘T’)[0]; try { await ddbClient.send(new UpdateCommand({ TableName: ROLLUP_TABLE, Key: { pk: ‘TENANT#${event.tenantId}‘, sk: ‘DATE#${date}#MODEL#${event.modelId} }, UpdateExpression: ‘SET #date = :date, modelId = :modelId, providerId = :providerId, requestCount = if_not_exists(requestCount, :zero) + :one, inputTokens = if_not_exists(inputTokens, :zero) + :inputTokens, outputTokens = if_not_exists(outputTokens, :zero) + :outputTokens, totalTokens = if_not_exists(totalTokens, :zero) + :totalTokens, providerCost = if_not_exists(providerCost, :zero) + :providerCost, billedCost = if_not_exists(billedCost, :zero) + :billedCost, cachedRequests = if_not_exists(cachedRequests, :zero) + :cached, phiRequests =

if_not_exists(phiRequests, :zero) + :phi, updatedAt = :updatedAt‘, ExpressionAttributeNames: { ‘#date’: ‘date’ }, ExpressionAttributeValues: { ‘:date’: date, ‘:modelId’: event.modelId, ‘:providerId’: event.providerId, ‘:zero’: 0, ‘:one’: 1, ‘:inputTokens’: event.inputTokens, ‘:outputTokens’: event.outputTokens, ‘:totalTokens’: event.totalTokens, ‘:providerCost’: event.providerCost, ‘:billedCost’: event.billedCost, ‘:cached’: event.cached ? 1 : 0, ‘:phi’: event.phiDetected ? 1 : 0, ‘:updatedAt’: new Date().toISOString(), }, })); } catch (error) { logger.error(‘Failed to update rollup’, error as Error, { tenantId: event.tenantId, date, modelId: event.modelId }); } }

async function getModelPricing(modelId: string, logger: Logger): Promise<{ inputPricePerMillion: number; outputPricePerMillion: number }> { const result = await executeQuery(‘SELECT input_price_per_million, output_price_per_million FROM ai_models WHERE id = $1‘, [modelId], logger); if (result.rowCount === 0) return { inputPricePerMillion: 1.0, outputPricePerMillion: 2.0 }; return { inputPricePerMillion: parseFloat(result.rows[0].input_price_per_million) || 1.0, outputPricePerMillion: parseFloat(result.rows[0].output_price_per_million) || 2.0 }; }

async function getTenantMargin(tenantId: string, logger: Logger): Promise { const result = await executeQuery(‘SELECT margin_percent FROM billing_settings WHERE tenant_id = $1‘, [tenantId], logger); return result.rowCount > 0 ? parseFloat(result.rows[0].margin_percent) : 20; }

function getDefaultStartDate(): string { const d = new Date(); d.setDate(d.getDate() - 30); return d.toISOString().split(‘T’)[0]; } function getTodayDate(): string { return new Date().toISOString().split(‘T’)[0]; } “‘

PART 5: DATABASE SCHEMA ADDITIONS

packages/infrastructure/migrations/005_admin_billing.sql

“‘sql – ==========
– RADIANT v2.2.0 - Admin & Billing Schema – ==========

– Admin Invitations CREATE TABLE IF NOT EXISTS invitations (id UUID PRIMARY KEY DEFAULT gen_random_uuid(), email VARCHAR(255) NOT NULL, role VARCHAR(50) NOT NULL, invited_by UUID NOT NULL REFERENCES administrators(id), app_id VARCHAR(100) NOT NULL, tenant_id VARCHAR(100) NOT NULL, environment VARCHAR(20) NOT NULL, token_hash VARCHAR(64) NOT NULL, expires_at TIMESTAMP WITH TIME ZONE NOT NULL, status VARCHAR(20) NOT NULL DEFAULT ‘pending’, message TEXT, accepted_at TIMESTAMP WITH TIME ZONE, accepted_by_ip VARCHAR(45), created_at TIMESTAMP WITH TIME ZONE NOT NULL DEFAULT NOW(), CONSTRAINT valid_invitation_status CHECK (status IN (‘pending’, ‘accepted’, ‘expired’, ‘revoked’)));

CREATE INDEX idx_invitations_email ON invitations(email); CREATE INDEX idx_invitations_tenant ON invitations(tenant_id); CREATE INDEX idx_invitations_token ON invitations(token_hash); CREATE INDEX idx_invitations_status ON invitations(status);

– Approval Requests CREATE TABLE IF NOT EXISTS approval_requests (id UUID PRIMARY KEY DEFAULT gen_random_uuid(), type VARCHAR(50) NOT NULL, app_id VARCHAR(100) NOT NULL, tenant_id VARCHAR(100) NOT NULL, environment VARCHAR(20) NOT NULL, requested_by UUID NOT NULL REFERENCES administrators(id), requested_at TIMESTAMP WITH TIME ZONE NOT NULL DEFAULT NOW(), expires_at TIMESTAMP WITH TIME ZONE NOT NULL, status VARCHAR(20) NOT NULL DEFAULT ‘pending’, approved_by

UUID REFERENCES administrators(id), approved_at TIMESTAMP WITH TIME ZONE, rejected_reason TEXT, action VARCHAR(100) NOT NULL, resource_type VARCHAR(100) NOT NULL, resource_id VARCHAR(255), details JSONB, priority VARCHAR(20) NOT NULL DEFAULT 'medium', notes TEXT, requires_two_person BOOLEAN NOT NULL DEFAULT false, created_at TIMESTAMP WITH TIME ZONE NOT NULL DEFAULT NOW(), CONSTRAINT valid_approval_status CHECK (status IN ('pending', 'approved', 'rejected', 'expired', 'cancelled')), CONSTRAINT valid_priority CHECK (priority IN ('low', 'medium', 'high', 'critical')));

CREATE INDEX idx_approvals_tenant ON approval_requests(tenant_id); CREATE INDEX idx_approvals_status ON approval_requests(status); CREATE INDEX idx_approvals_requested_by ON approval_requests(requested_by);

– Admin Profiles CREATE TABLE IF NOT EXISTS admin_profiles (admin_id UUID PRIMARY KEY REFERENCES administrators(id) ON DELETE CASCADE, notifications JSONB NOT NULL DEFAULT '{}', timezone VARCHAR(50) NOT NULL DEFAULT 'America/New_York', language VARCHAR(10) NOT NULL DEFAULT 'en', date_format VARCHAR(20) NOT NULL DEFAULT 'MM/DD/YYYY', time_format VARCHAR(10) NOT NULL DEFAULT '12h', currency VARCHAR(3) NOT NULL DEFAULT 'USD', theme VARCHAR(20) NOT NULL DEFAULT 'system', default_environment VARCHAR(20) NOT NULL DEFAULT 'dev', sidebar_collapsed BOOLEAN NOT NULL DEFAULT false, table_rows_per_page INTEGER NOT NULL DEFAULT 25, updated_at TIMESTAMP WITH TIME ZONE NOT NULL DEFAULT NOW());

– Billing Settings CREATE TABLE IF NOT EXISTS billing_settings (tenant_id VARCHAR(100) PRIMARY KEY, margin_percent DECIMAL(5,2) NOT NULL DEFAULT 20.00, margin_type VARCHAR(20) NOT NULL DEFAULT 'fixed', tiers JSONB, tax_enabled BOOLEAN NOT NULL DEFAULT false, tax_percent DECIMAL(5,2) NOT NULL DEFAULT 0.00, tax_id VARCHAR(50), stripe_customer_id VARCHAR(100), default_payment_method_id VARCHAR(100), auto_pay BOOLEAN NOT NULL DEFAULT false, billing_cycle_day INTEGER NOT NULL DEFAULT 1, currency VARCHAR(3) NOT NULL DEFAULT 'USD', budget_limit DECIMAL(12,2), alert_thresholds INTEGER[] NOT NULL DEFAULT '{50, 75, 90, 100}', created_at TIMESTAMP WITH TIME ZONE NOT NULL DEFAULT NOW(), updated_at TIMESTAMP WITH TIME ZONE NOT NULL DEFAULT NOW(), CONSTRAINT valid_margin_type CHECK (margin_type IN ('fixed', 'tiered')), CONSTRAINT valid_billing_day CHECK (billing_cycle_day BETWEEN 1 AND 28));

– Invoices CREATE TABLE IF NOT EXISTS invoices (id UUID PRIMARY KEY DEFAULT gen_random_uuid(), tenant_id VARCHAR(100) NOT NULL, app_id VARCHAR(100) NOT NULL, period_start DATE NOT NULL, period_end DATE NOT NULL, subtotal DECIMAL(12,2) NOT NULL, margin_percent DECIMAL(5,2) NOT NULL, tax DECIMAL(12,2) NOT NULL DEFAULT 0, tax_percent DECIMAL(5,2) NOT NULL DEFAULT 0, total DECIMAL(12,2) NOT NULL, currency VARCHAR(3) NOT NULL DEFAULT 'USD', status VARCHAR(20) NOT NULL DEFAULT 'draft', due_date TIMESTAMP WITH TIME ZONE NOT NULL, paid_at TIMESTAMP WITH TIME ZONE, line_items JSONB NOT NULL DEFAULT '[]', stripe_invoice_id VARCHAR(100), stripe_payment_intent_id VARCHAR(100), created_at TIMESTAMP WITH TIME ZONE NOT NULL DEFAULT NOW(), updated_at TIMESTAMP WITH TIME ZONE NOT NULL DEFAULT NOW(), CONSTRAINT valid_invoice_status CHECK (status IN ('draft', 'pending', 'paid', 'overdue', 'cancelled')));

CREATE INDEX idx_invoices_tenant ON invoices(tenant_id); CREATE INDEX idx_invoices_status

ON invoices(status); CREATE INDEX idx_invoices_period ON invoices(period_start, period_end);

– Admin Notifications CREATE TABLE IF NOT EXISTS admin_notifications (id UUID PRIMARY KEY DEFAULT gen_random_uuid(), admin_id UUID NOT NULL REFERENCES administrators(id) ON DELETE CASCADE, tenant_id VARCHAR(100) NOT NULL, type VARCHAR(50) NOT NULL, priority VARCHAR(20) NOT NULL DEFAULT 'medium', title VARCHAR(255) NOT NULL, message TEXT NOT NULL, action_url VARCHAR(500), action_label VARCHAR(100), read BOOLEAN NOT NULL DEFAULT false, read_at TIMESTAMP WITH TIME ZONE, dismissed BOOLEAN NOT NULL DEFAULT false, dismissed_at TIMESTAMP WITH TIME ZONE, email_sent BOOLEAN NOT NULL DEFAULT false, email_sent_at TIMESTAMP WITH TIME ZONE, created_at TIMESTAMP WITH TIME ZONE NOT NULL DEFAULT NOW(), expires_at TIMESTAMP WITH TIME ZONE, CONSTRAINT valid_notification_type CHECK (type IN ('security', 'billing', 'deployment', 'approval', 'system', 'alert')), CONSTRAINT valid_notification_priority CHECK (priority IN ('low', 'medium', 'high', 'critical')));

CREATE INDEX idx_notifications_admin ON admin_notifications(admin_id); CREATE INDEX idx_notifications_read ON admin_notifications(admin_id, read); "'

API ROUTES SUMMARY

Admin Routes

Method	Path	Description	Permission
POST	/admin/invitations	Create invitation	admin:write
GET	/admin/invitations	List invitations	admin:read
GET	/admin/invitations/:id	Get invitation	admin:read
POST	/admin/invitations/:id/resend	Resend invitation	admin:write
DELETE	/admin/invitations/:id	Revoke invitation	admin:write
POST	/admin/invitations/accept	Accept invitation	(public)
POST	/admin/approvals	Create approval request	approvals:initiate
GET	/admin/approvals	List approvals	approvals:read
GET	/admin/approvals/:id	Get approval	approvals:read
POST	/admin/approvals/:id/process	Approve/reject	approvals:*
DELETE	/admin/approvals/:id	Cancel approval	approvals:initiate
GET	/admin/users	List admins	admin:read
GET	/admin/users/:id	Get admin	admin:read
PUT	/admin/users/:id	Update admin	admin:write
DELETE	/admin/users/:id	Delete admin	admin:*

Billing Routes

Method	Path	Description	Permission
GET	/billing/settings	Get settings	billing:read
PUT	/billing/settings	Update settings	billing:*

Method	Path	Description	Permission
GET	/billing/current	Current period usage	billing:read
GET	/billing/projections	Cost projections	billing:read
GET	/billing/invoices	List invoices	billing:read
GET	/billing/invoices/:id	Get invoice	billing:read
POST	/billing/generate	Generate invoice	billing:*

Metering Routes

Method	Path	Description	Permission
POST	/metering/record	Record usage event	(internal)
POST	/metering/batch	Batch record events	(internal)
GET	/metering/summary	Usage summary	billing:read
GET	/metering/rollups	Daily rollups	billing:read

DEPLOYMENT VERIFICATION

“‘bash # 1. Create Invitation curl -X POST -H “Authorization: Bearer $ADMIN_TOKEN” -H “Content-Type: application/json” \ -d ‘{“email”: “new.admin@company.com”, “role”: “operator”, “message”: “Welcome!”, “expiresInHours”: 48}’ \ https://admin-api.thinktank.YOUR_DOMAIN.com/api/v2/adı

2. List Pending Approvals (for me to approve)

curl -H “Authorization: Bearer $ADMIN_TOKEN” \ “https://admin-api.thinktank.YOUR_DOMAIN.com/api/v

3. Get Current Billing Period

curl -H “Authorization: Bearer $ADMIN_TOKEN” \ https://admin-api.thinktank.YOUR_DOMAIN.com/api/v2

4. Get Usage Rollups

curl -H “Authorization: Bearer $ADMIN_TOKEN” \ “https://admin-api.thinktank.YOUR_DOMAIN.com/api/v 12-01&endDate=2024-12-21”

5. Generate Invoice

curl -X POST -H “Authorization: Bearer $ADMIN_TOKEN” -H “Content-Type: application/json” \ -d ‘{“periodStart”: “2024-12-01”, “periodEnd”: “2024-12-31”, “sendToStripe”: true}’ \ https://admin-api.thinktank.YOUR_DOMAIN.com/api/v2/billing/generate “‘

ESTIMATED COSTS BY TIER

Component	Tier 1	Tier 2	Tier 3	Tier 4	Tier 5
Lambda (Admin)	$5	$15	$50	$150	$500
Lambda (Billing)	$5	$20	$75	$200	$600
DynamoDB (Usage)	$10	$50	$200	$500	$1,500
SES (Emails)	$1	$5	$20	$50	$150
Stripe Fees	Variable	Variable	Variable	Variable	Variable
Prompt 5 Total	~$21	~$90	~$345	~$900	~$2,750

Note: Stripe fees are 2.9% + $0.30 per transaction, not included in estimates.

NEXT PROMPTS

Continue with: - **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 5: Lambda Functions - Admin & Billing RADIANT v2.2.0 - December 2024

â• â

END OF SECTION 5

â• â

â• â