SAMPLE DELIVERABLE · PREMIUM TIER

Acme HR SaaS

DynamoDB Schema Review · Premium Tier

Reviewer · Tejovanth N · singletable.dev Delivered · May 2026

Tejovanth has shipped production DynamoDB systems for rasika.life, rekha.app, and several B2B SaaS apps since 2022. Writes single-table design patterns at singletable.dev.

AT A GLANCE

5 entities · 12 access patterns · 1 overloaded GSI · ~$2/month at launch · ~$21/month at 10×. Greenfield design optimised for tenant isolation and time-ordered queries.

- · ULID over UUID for composite sort keys: preserves time-ordered query semantics
- · Relationship item (DeptEmployee) for M:N instead of a GSI: cheaper, no extra index
- · Sparse GSI for open jobs: closed jobs auto-excluded, no FilterExpression
- · Denormalised headcount via Transactions: consistent without scanning
- · GSI1 overloaded across 3 entity types: 1 index covers AP4, AP8, and AP11

TABLE OF CONTENTS

- Project brief
- Entities
- Access patterns
- Schema diagram
- Table design
- Sample data
- ElectroDB entity definitions
- GSI1 overloading: rationale
- Access pattern → query mapping
- Cost analysis
- Trade-off analysis
- Recommended next steps

Project brief

Company: Acme HR SaaS
Use case: Multi-tenant HR platform for small and mid-size companies (10–500 employees per org)
Core features: Employee directory, department management, internal job board, applicant tracking
Stack: SST v3 · Node.js / TypeScript · ElectroDB · DynamoDB on-demand
Scale targets: 500 organisations at launch · up to 1,000 employees per org · 50 concurrent writes/second at peak
Key constraint: Employee email is the SSO identifier. Lookup by email must be sub-millisecond.

Acme HR SaaS is a multi-tenant B2B product. Each paying customer is an Organisation. Employees within an Organisation can browse open job postings, apply internally, and track their application status. HR admins manage departments, headcount, and the hiring pipeline. The auth flow authenticates employees by email address (Google/Okta SSO), so email lookup must be a primary key operation, not a scan.

Entities

Five entity types. All IDs are ULIDs (lexicographically sortable, time-ordered, URL-safe). Choosing ULID over UUID here is intentional: composite sort keys like `JOB#<postedAt>#<jobId>` depend on the ID sorting by creation time. A UUID in that position produces random ordering and breaks time-sorted queries.

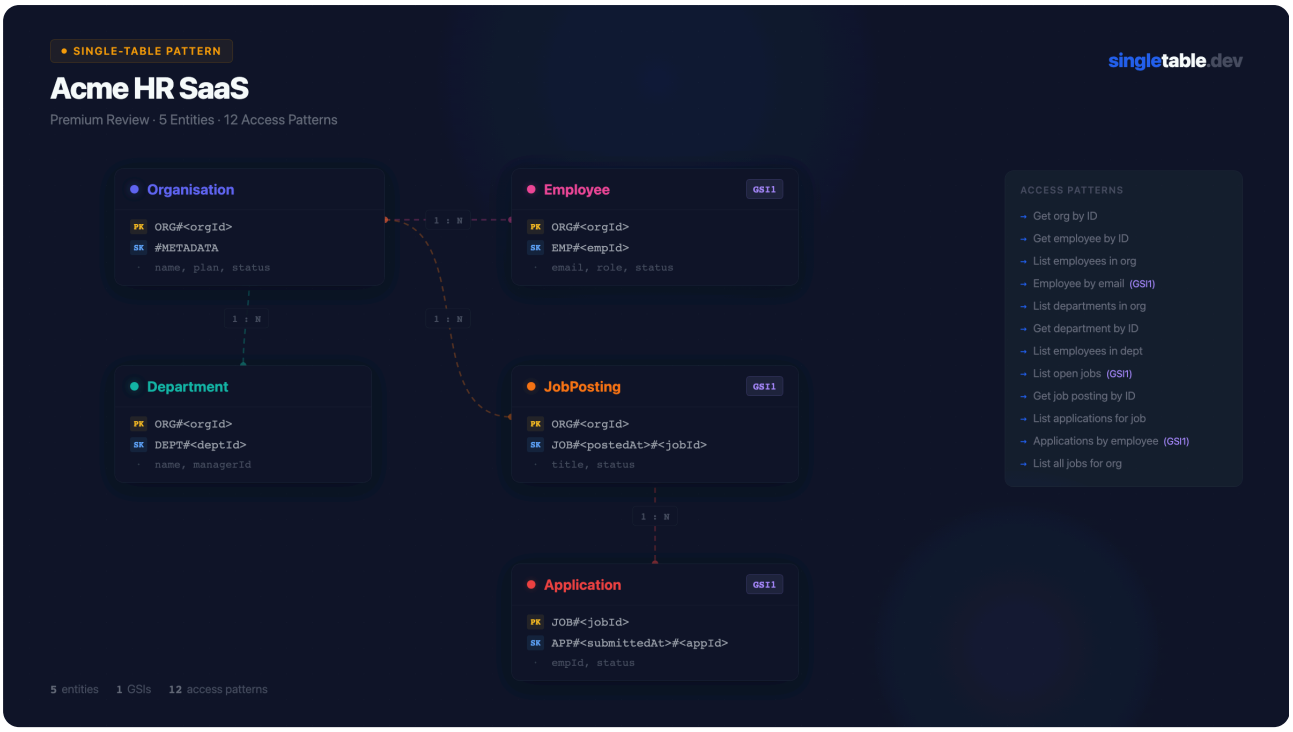
Entity	Key Attributes
Organisation	`orgId`, `name`, `plan` (free / pro / enterprise), `status`, `createdAt`
Employee	`empId`, `orgId`, `email`, `firstName`, `lastName`, `departmentId`, `role` (admin / manager / employee), `status` (active / inactive), `hiredAt`, `terminatedAt?`
Department	`deptId`, `orgId`, `name`, `managerId` (empId), `headcount` (denormalised counter)
JobPosting	`jobId`, `orgId`, `departmentId`, `title`, `status` (draft / open / closed), `postedAt`, `closedAt?`
Application	`appId`, `jobId`, `empId`, `status` (applied / reviewing / rejected / accepted), `submittedAt`

Access patterns

Twelve access patterns.

#	Description	Who triggers it
AP1	Get organisation by ID	Session bootstrap on every login
AP2	Get employee by ID (within org context)	Profile page, permission checks
AP3	List all employees in an org, sorted by hire date	Employee directory
AP4	Get employee by email	SSO authentication, invite deduplication
AP5	List all departments in an org	Org chart, department picker
AP6	Get department by ID	Department detail page
AP7	List all employees in a department	Team roster, manager view
AP8	List all open job postings for an org, newest first	Internal job board
AP9	Get job posting by ID	Job detail page
AP10	List all applications for a job posting, newest first	Applicant review pipeline
AP11	List all applications submitted by an employee	Employee’s application history
AP12	List all job postings for an org (all statuses), newest first	HR admin dashboard

Schema diagram



PK/SK structure — 5 entities plus the DeptEmployee relationship item. GSI1 is overloaded across AP4 (email lookup), AP8 (open jobs), and AP11 (applications by employee).

Table design

Main table: `acme-hr-{stage}`

One table. On-demand billing. One overloaded GSI (GSI1).

PK/SK structure

Entity	PK	SK	Notes
Organisation	`ORG#<orgId>`	`#METADATA`	`#` sorts before all letters, so it's always first in any `ORG#` partition
Department	`ORG#<orgId>`	`DEPT#<deptId>`	Co-located with Org; AP5 is a single Query with `begins_with DEPT#`
Employee	`ORG#<orgId>`	`EMP#<empId>`	Co-located with Org; AP3 is a single Query with `begins_with EMP#`
DeptEmployee *(relationship item)*	`DEPT#<deptId>`	`EMP#<empId>`	Written atomically alongside the Employee item (same Transaction). Required for AP7.
JobPosting	`ORG#<orgId>`	`JOB#<postedAt>#<jobId>`	Composite SK enables AP12 (all jobs, newest first) without a GSI
Application	`JOB#<jobId>`	`APP#<submittedAt>#<appId>`	Composite SK enables AP10 (applications per job, newest first) without a GSI

The composite sort keys on JobPosting and Application are the load-bearing decision in this schema. Because `postedAt` and `submittedAt` are ULIDs, `ScanIndexForward=false` returns results newest-first with zero client-side sort. See ULIDs vs UUIDs vs Timestamps for Sort Keys for why this breaks with UUID.

Why DeptEmployee is a separate write

AP7 (list employees in a department) requires querying by `DEPT#<deptId>`. Employee items live at `ORG#<orgId>`, the wrong partition for that query. Rather than a GSI, we write a lightweight relationship item at `DEPT#<deptId>` / `EMP#<empId>` alongside every employee create/update. This item holds only `orgId`, enough to hydrate the full Employee record via `BatchGetItem` if needed.

This is the typical approach for M:N in DynamoDB without the GSI cost. See The 5 Most Common Single-Table Design Mistakes. Mistake #2 is reaching for a GSI when a relationship item solves the problem cheaper.

GSI1: overloaded across three entity types

Rather than three separate GSIs (one for email lookup, one for applications-by-employee, one for open-jobs-by-org), this schema uses one GSI with disjoint PK shapes per entity type. DynamoDB charges per GB of projected data in a GSI, not per access pattern. Overloading keeps storage cost flat as the schema grows.

Entity	GSI1PK	GSI1SK	Enables
Employee	`EMAIL#<email>`	`EMP#<empId>`	AP4: employee by email
Application	`EMP#<empId>`	`APP#<submittedAt>#<appId>`	AP11: applications by employee, time-ordered
JobPosting *(open only — sparse)*	`ORG#<orgId>#OPEN`	`JOB#<postedAt>#<jobId>`	AP8: open jobs by org, newest-first

Why the sparse index on JobPosting

When a job is closed, its `GSI1PK` attribute is deleted (set to `undefined`). DynamoDB only writes a record into a GSI when the projected attributes are present, so closed jobs are never indexed in GSI1. The AP8 result set is clean without a `FilterExpression`, and no cleanup Lambda is required. This pattern is covered in Sparse Indexes: Queues, Soft Deletes, Active-Only Views.

Why the key shapes don't collide

- `EMAIL#` prefix belongs exclusively to Employee items
- `EMP#` prefix in GSI1PK belongs exclusively to Application items (Employee items use `EMAIL#` in GSI1PK)
- `ORG#<id>#OPEN` belongs exclusively to JobPosting items

The `SK begins_with` filter in each query provides a second discriminator. The full rationale is in When to Add a GSI vs Reshape Your Sort Key.

Sample data

Ten representative items with realistic attribute values. ULIDs truncated for readability.

PK	SK	type	Key attributes
`ORG#01HXAA`	`#METADATA`	org	name="Acme Corp", plan="pro", status="active"
`ORG#01HXAA`	`DEPT#01HXAB`	dept	name="Engineering", managerId="01HXAD", headcount=12
`ORG#01HXAA`	`DEPT#01HXAC`	dept	name="People Ops", managerId="01HXAE", headcount=4
`ORG#01HXAA`	`EMP#01HXAD`	emp	email="alice@acme.co", firstName="Alice", role="admin", **GSI1PK**="EMAIL#alice@acme.co"
`ORG#01HXAA`	`EMP#01HXAE`	emp	email="bob@acme.co", firstName="Bob", role="manager", **GSI1PK**="EMAIL#bob@acme.co"
`DEPT#01HXAB`	`EMP#01HXAD`	dept_emp	orgId="01HXAA"
`DEPT#01HXAB`	`EMP#01HXAE`	dept_emp	orgId="01HXAA"
`ORG#01HXAA`	`JOB#01HXZZ1#01HXAF`	job	title="Senior Engineer", status="open", **GSI1PK**="ORG#01HXAA#OPEN"
`ORG#01HXAA`	`JOB#01HXZZ0#01HXAG`	job	title="HR Coordinator", status="closed", *(no GSI1PK)*
`JOB#01HXAF`	`APP#01HXZZ9#01HXAH`	app	empId="01HXAD", status="reviewing", **GSI1PK**="EMP#01HXAD"

ElectroDB entity definitions

Production-ready TypeScript. Set `TABLE_NAME` in your environment. All five entities share one table and one ElectroDB `Service`.

View interactive version — copy-paste ready code at singletable.dev/review/sample

ORGANISATION

```
import { Entity } from 'electrodb'
import { dynamoClient } from './dynamo'

export const Organisation = new Entity(
  {
    model: { entity: 'org', version: '1', service: 'acme-hr' },
    attributes: {
      orgId:     {
        type: 'string',
        required: true
      },
      name:      {
        type: 'string',
        required: true
      },
      plan:      {
        type: 'string',
        enum: ['free', 'pro', 'enterprise'], required: true
      },
      status:    {
        type: 'string',
        enum: ['active', 'suspended'], default: 'active'
      },
      createdAt: {
        type: 'string',
        readOnly: true, default: () => new Date().toISOString()
      },
    },
    indexes: {
      byOrg: {
        pk: {
          field: 'PK',
```

```
          composite: ['orgId'],
          template: 'ORG#${orgId}'
        },
        sk: {
          field: 'SK',
          composite: [],
          template: '#METADATA'
        },
      },
    },
  },
  { table: process.env.TABLE_NAME!, client: dynamoClient },
)
```

EMPLOYEE

```
export const Employee = new Entity(
  {
    model: { entity: 'employee', version: '1', service: 'acme-hr' },
    attributes: {
      empId:        {
        type: 'string',
        required: true
      },
      orgId:        {
        type: 'string',
        required: true
      },
      email:        {
        type: 'string',
        required: true
      },
      firstName:    {
        type: 'string',
        required: true
      },
      lastName:     {
        type: 'string',
        required: true
      },
      departmentId: {
        type: 'string'
      },
      role:         {
        type: 'string',
        enum: ['admin', 'manager', 'employee'], default: 'employee'
      },
      status:       {
        type: 'string',
        enum: ['active', 'inactive'], default: 'active'
      },
      hiredAt:      {
        type: 'string',
        readOnly: true, default: () => new Date().toISOString()
      },
      terminatedAt: {
        type: 'string'
```

```
      },
    },
    indexes: {
      byOrg: {
        pk: {
          field: 'PK',
          composite: ['orgId'],
          template: 'ORG#${orgId}'
        },
        sk: {
          field: 'SK',
          composite: ['empId'],
          template: 'EMP#${empId}'
        },
      },
      byEmail: {
        index: 'GSI1',
        pk: {
          field: 'GSI1PK',
          composite: ['email'],
          template: 'EMAIL#${email}'
        },
        sk: {
          field: 'GSI1SK',
          composite: ['empId'],
          template: 'EMP#${empId}'
        },
      },
    },
  },
  { table: process.env.TABLE_NAME!, client: dynamoClient },
)
```

DEPARTMENT

```
export const Department = new Entity(
  {
    model: { entity: 'department', version: '1', service: 'acme-hr' },
    attributes: {
      deptId:    {
        type: 'string',
        required: true
      },
      orgId:     {
        type: 'string',
        required: true
      },
      name:      {
        type: 'string',
        required: true
      },
      managerId: {
        type: 'string'
      },
      headcount: {
        type: 'number',
        default: 0
      },
    },
    indexes: {
      byOrg: {
        pk: {
          field: 'PK',
          composite: ['orgId'],
          template: 'ORG#${orgId}'
        },
        sk: {
          field: 'SK',
          composite: ['deptId'],
          template: 'DEPT#${deptId}'
        },
      },
    },
  },
  { table: process.env.TABLE_NAME!, client: dynamoClient },
)
```

JOBPOSTING

```
export const JobPosting = new Entity(
  {
    model: { entity: 'job', version: '1', service: 'acme-hr' },
    attributes: {
      jobId:        {
        type: 'string',
        required: true
      },
      orgId:        {
        type: 'string',
        required: true
      },
      departmentId: {
        type: 'string'
      },
      title:        {
        type: 'string',
        required: true
      },
      status:       {
        type: 'string',
        enum: ['draft', 'open', 'closed'], default: 'draft'
      },
      postedAt:     {
        type: 'string'
      },
      closedAt:     {
        type: 'string'
      },
      // Sparse GSI1 — only set when status = 'open'
      gsi1pk: {
        type: 'string',
        watch: ['orgId', 'status'],
        get: (_, { orgId, status }) =>
          status === 'open' ? `ORG#${orgId}#OPEN` : undefined,
        readOnly: true,
      },
    },
    indexes: {
      byOrg: {
        pk: {
```

```
        field: 'PK',
        composite: ['orgId'],
        template: 'ORG#${orgId}'
      },
        sk: {
        field: 'SK',
        composite: ['postedAt', 'jobId'],
        template: 'JOB#${postedAt}#${jobId}'
      },
      },
      openByOrg: {
        index: 'GSI1',
        pk: {
          field: 'GSI1PK',
          composite: ['orgId'],
          template: 'ORG#${orgId}#OPEN'
        },
        sk: {
          field: 'GSI1SK',
          composite: ['postedAt', 'jobId'],
          template: 'JOB#${postedAt}#${jobId}'
        },
      },
    },
  },
  { table: process.env.TABLE_NAME!, client: dynamoClient },
)
```

APPLICATION

```
export const Application = new Entity(
  {
    model: { entity: 'application', version: '1', service: 'acme-hr' },
    attributes: {
      appId:       {
        type: 'string',
        required: true
      },
      jobId:       {
        type: 'string',
        required: true
      },
      empId:       {
        type: 'string',
        required: true
      },
      status: {
        type: 'string',
        enum: ['applied', 'reviewing', 'rejected', 'accepted'],
        default: 'applied',
      },
      submittedAt: {
        type: 'string',
        readOnly: true, default: () => new Date().toISOString()
      },
    },
    indexes: {
      byJob: {
        pk: {
          field: 'PK',
          composite: ['jobId'],
          template: 'JOB#${jobId}'
        },
        sk: {
          field: 'SK',
          composite: ['submittedAt', 'appId'],
          template: 'APP#${submittedAt}#${appId}'
        },
      },
      byEmployee: {
        index: 'GSI1',
```

```
        pk: {
          field: 'GSI1PK',
          composite: ['empId'],
          template: 'EMP#${empId}'
        },
        sk: {
          field: 'GSI1SK',
          composite: ['submittedAt', 'appId'],
          template: 'APP#${submittedAt}#${appId}'
        },
      },
    },
  },
  { table: process.env.TABLE_NAME!, client: dynamoClient },
)
```

GSI1 overloading: rationale

GSI1 serves three different query shapes. Each uses a distinct PK prefix, so they can't collide.

Query	GSI1PK pattern	GSI1SK prefix	FilterExpression needed?
Employee by email (AP4)	`EMAIL#<email>`	`EMP#`	No
Applications by employee (AP11)	`EMP#<empId>`	`APP#`	No
Open jobs by org (AP8)	`ORG#<orgId>#OPEN`	`JOB#`	No — sparse index handles it

The obvious alternative is three separate GSIs, one per access pattern. It works, but costs more: GSIs are billed on the data they store, and each additional index multiplies storage cost proportionally. All three access patterns here are served from one GSI with disjoint key shapes, and there's no latency difference between querying one GSI or three.

> The `gsi1pk` computed attribute uses ElectroDB's `watch` + `get` to set `GSI1PK` only when `status === 'open'`. When a job closes, `GSI1PK` becomes `undefined` and the item drops out of GSI1 automatically. No cleanup Lambda needed. See Sparse Indexes for the full pattern.

See When to Add a GSI vs Reshape Your Sort Key for a full treatment.

Access pattern → query mapping

#	Description	Operation	Key condition
AP1	Get org by ID	`GetItem`	`PK=ORG#<orgId>`, `SK=#METADATA`
AP2	Get employee by ID	`GetItem`	`PK=ORG#<orgId>`, `SK=EMP#<empId>`
AP3	List employees in org	`Query`	`PK=ORG#<orgId>`, `SK begins_with EMP#`, `ScanIndexForward=false`
AP4	Get employee by email	`Query` on GSI1	`GSI1PK=EMAIL#<email>`
AP5	List departments in org	`Query`	`PK=ORG#<orgId>`, `SK begins_with DEPT#`
AP6	Get department by ID	`GetItem`	`PK=ORG#<orgId>`, `SK=DEPT#<deptId>`
AP7	List employees in dept	`Query`	`PK=DEPT#<deptId>`, `SK begins_with EMP#`
AP8	List open jobs (org)	`Query` on GSI1	`GSI1PK=ORG#<orgId>#OPEN`, `ScanIndexForward=false`
AP9	Get job posting by ID	`GetItem`	`PK=ORG#<orgId>`, `SK=JOB#<postedAt>#<jobId>` (see note)
AP10	List applications for job	`Query`	`PK=JOB#<jobId>`, `SK begins_with APP#`, `ScanIndexForward=false`
AP11	List applications by employee	`Query` on GSI1	`GSI1PK=EMP#<empId>`, `GSI1SK begins_with APP#`, `ScanIndexForward=false`
AP12	List all jobs for org	`Query`	`PK=ORG#<orgId>`, `SK begins_with JOB#`, `ScanIndexForward=false`

> AP9: `GetItem` on a JobPosting requires both `orgId` and the exact SK (`JOB#<postedAt>#<jobId>`). If job pages are accessed by `jobId` alone: (a) embed `orgId` and `postedAt` in the URL slug for a direct `GetItem`, or (b) store a lookup item at `PK=JOB#<jobId>`, `SK=#META` containing `orgId` and `postedAt` for a two-step fetch. Option (a) is simpler; (b) is better if the URL must be opaque.

Cost analysis

Estimates based on DynamoDB on-demand pricing ($0.25/million RCU, $1.25/million WCU). Baseline: 500 active orgs, 200 employees per org, 20 open jobs per org.

See On-Demand vs Provisioned: The Real Math for when provisioned capacity becomes worth evaluating.

#	Access Pattern	RCU / call	WCU / call	Est. calls/day	Daily cost
AP1	Get org	0.5	—	50,000	$0.006
AP2	Get employee	0.5	—	200,000	$0.025
AP3	List employees in org	4	—	5,000	$0.005
AP4	Get employee by email	0.5	—	100,000	$0.013
AP5	List departments	0.5	—	10,000	$0.001
AP6	Get department	0.5	—	20,000	$0.003
AP7	List employees in dept	1	—	15,000	$0.002
AP8	List open jobs	1	—	30,000	$0.004
AP9	Get job posting	0.5	—	25,000	$0.003
AP10	List applications for job	0.5	—	8,000	$0.001
AP11	List applications by employee	0.5	—	5,000	$0.001
AP12	List all jobs for org	1.5	—	2,000	~$0.001
Employee write	Create / update (main + DeptEmp items)	—	2	2,000	$0.005
Job status change	Open → closed (removes from GSI1)	—	1	500	$0.001
Application submit	New application	—	1	500	$0.001
Total					**≈ $0.071/day**

Monthly DynamoDB cost at baseline: ≈ $2.15.

Cost at 10× scale (5,000 orgs, 2,000 employees per org) scales roughly linearly to ≈ $21/month. The on-demand → provisioned crossover for this access pattern mix is around 50–100M requests/month. See Cost Optimization: The 5 Levers That Actually Matter for when that math starts to matter.

Trade-off analysis

What this schema supports

- All 12 access patterns with single-digit millisecond reads
- Complete tenant isolation: you can't reach cross-org data without `orgId`
- Time-ordered results for jobs (AP8, AP12) and applications (AP10, AP11) via composite sort keys, with no client-side sort needed
- Clean open-jobs query (AP8) with zero `FilterExpression` cost via sparse GSI
- Internal applicant tracking: employees can apply to postings across departments

What this schema doesn't handle well

1. Employee search by name AP3 lists employees sorted by hire date (ULID order). Alphabetical sort requires a client-side sort, which is negligible at 200 employees per org. Free-text search (e.g., name autocomplete) isn't possible with DynamoDB queries. For admin-facing search, use OpenSearch Serverless or a DynamoDB export to S3 + Athena. This is a DynamoDB constraint, not a schema design problem. See What Queries Your DynamoDB Schema Explicitly Doesn't Support.

2. Real-time headcount without counter drift The `headcount` attribute on Department is a denormalised counter maintained via atomic increment/decrement in Transactions. It stays accurate as long as every employee-department mutation goes through the transaction path. A direct `Employee.upsert()` will silently drift the count. Wrap all employee-department mutations in a single service function that enforces the Transaction. See DynamoDB Transactions vs Conditional Writes for when atomic counters are the right tool.

3. Org-wide analytics queries "All employees hired in Q1 across all orgs" or "job postings with more than 10 applications" require a full table scan or a secondary analytics store. Enable DynamoDB Streams and pipe events to S3 via Kinesis Firehose. Athena or DuckDB can answer arbitrary queries on the raw data at near-zero cost.

4. Employee lookup by empId alone (no orgId) AP2 requires `orgId` to construct the full primary key. This is correct for a multi-tenant SaaS: `orgId` comes from the session JWT. If a future external integration needs `empId`-only lookup, add a lookup item at `PK=EMP#<empId>`, `SK=#META` containing just `orgId`. Only add this if the use case materialises.

5. Sorting employees alphabetically across large orgs At 200 employees per org, client-side sort on AP3 is negligible. If orgs grow beyond 5,000 employees, consider adding `lastName` to the Employee SK: `EMP#<lastName>#<empId>`. That's a schema migration; see Schema Migrations: The Guide Nobody Wrote for how to approach it.

Recommended next steps

1. **DeptEmployee writes must go through a Transaction.** Every `Employee.upsert()` that sets or changes `departmentId` must atomically write the relationship item too. A missed write silently breaks AP7. It won't surface as an error, just wrong data.

2. Before you ship, write an integration test for the sparse GSI: create an open job, confirm AP8 returns it, close it, confirm AP8 excludes it. The sparse index is correct by construction, but that test will catch any regression if someone touches the job-close logic later.

3. **Set `default: () => ulid()` on every `*Id` attribute** and validate ULID format at the service boundary. Swapping to UUID or nanoid silently breaks time-ordered queries. The symptom is random-ordered pagination, which is hard to trace back to an ID format change.

4. **Add optimistic locking to Employee.** Two concurrent admin updates produce silent last-write-wins. Add a `version` counter and `ConditionExpression: version = :expected`. See DynamoDB Transactions vs Conditional Writes.

5. Do the migration planning now, before you have production data. Greenfield schemas are cheap to change. Once you have millions of items, a sort key restructure requires a full live migration, and nobody wants to run that in production. See Schema Migrations: The Guide Nobody Wrote.

Tejovanth N · singletable.dev