Serverless AI Agent — Construction Material Estimator

Goal: A modern, secure, serverless AI agent that takes user inputs (e.g., concrete class, volume, mix ratio, material densities) and returns a complete Bill of Materials (BoM). Phase 2: integrate live supplier pricing.

Quick orientation

This document is a complete, end-to-end, **step-by-step blueprint** for building, deploying, and operating a production-ready serverless AI agent for construction material estimation. It includes: requirements, data & formulas, architecture options, code examples, infrastructure-as-code snippets, security controls, testing plans, monitoring, and a Phase-2 pricing-integration plan. Each item is actionable — treat each SR (Step/Stage Requirement) as a checklist entry to complete.

> Note: This guide is written for a serverless-first implementation (AWS + Vercel examples) but includes alternatives (GCP/Firebase, Azure, Supabase) where useful.

SR0 — Project summary & acceptance criteria

- **Primary function:** Accurately compute quantities of cement, sand, and aggregate for any given concrete volume & mix ratio; output BoM in downloadable formats (CSV, XLSX, PDF) and JSON/APIs.
- **Non-functional requirements:** secure, low-latency, horizontally scalable, observable, cost-controlled, mobile-friendly UI, offline-capable minimal features (PWA), i18n-ready.
- **Acceptance criteria:**
- Estimation function passes unit tests and integration tests (±0.5% numeric tolerance).
- API responds < 500ms p50 under normal load and scales with bursts.
- Authenticated users can save, retrieve, and share estimates.
- BoM export to CSV/PDF works and matches UI view.
- Security checklist satisfied (TLS, IAM least privilege, secrets rotated, rate limits configured).

SR1 — Discovery & requirements capture

1. Stakeholders & users: site engineers, contractors, material suppliers, quantity surveyors. Capture personas and what they need (e.g., quick estimate vs detailed procurement list).
2. Inputs to support (initial):
3. Volume (m^3)
4. Mix ratio (parts; e.g., 1:2:4)
5. Material densities (kg/m^3) — allow user override; provide defaults
6. Unit preferences (kg, tonnes, bags, m^3)

7. Waste/over-provision settings (percent or absolute)
8. Project metadata (name, location, client, notes)
9. Outputs (initial):
10. BoM (per-material): volume (m^3), mass (kg), count (e.g., 50 kg bags), cost placeholders
11. JSON API response
12. Exports: CSV, XLSX, PDF
13. Optional: supplier suggestions (Phase 2)
14. Edge cases & validations to capture:
15. Invalid ratios (zero/negative parts)
16. Very small volumes (<0.01 m^3)
17. Missing densities (fall back to defaults with warning)
18. Unit mismatch (user enters liters, etc.)

SR2 — Domain math & exact algorithm (must be authoritative)

Use *volumetric mix method* as baseline.

Core math definitions: - Let parts = (p_cement, p_sand, p_agg). Let S = p_cement + p_sand + p_agg. - For requested concrete volume `V_total` (m^3): - Volume_cement = (p_cement / S) * V_total - Volume_sand = (p_sand / S) * V_total - Volume_agg = (p_agg / S) * V_total - Using densities `ρ_cement`, `ρ_sand`, `ρ_agg` (kg/m^3): - Mass_cement = Volume_cement * ρ_cement - Mass_sand = Volume_sand * ρ_sand - Mass_agg = Volume_agg * ρ_agg - Convert to practical units: - Cement bags (50 kg) = ceil(Mass_cement / 50) - Mass to tonnes = Mass_kg / 1000

Allowable adjustments & business rules (configurable): - Dry-volume factor (optional): multiply `V_total` by `dry_factor` (common default: 1.54) to account for bulking. *Make dry_factor configurable and documented.* - Wastage percentages by material (defaults: cement 2%, sand 3%, aggregate 2%) — also configurable per-project.

Example: 100 m^3 Class 20 concrete, mix 1:2:4 — full digit-by-digit math (We compute step-by-step to avoid arithmetic slip-ups.)

1. Parts: 1 + 2 + 4 = **7**.
2. Volume of cement = (1 / 7) * 100
3. Compute 1 / 7 = 0.14285714285714285
4. Multiply by 100 → 0.14285714285714285 * 100 = **14.285714285714286 m^3**
5. Volume of sand = (2 / 7) * 100
6. Compute 2 / 7 = 0.2857142857142857
7. Multiply by 100 → 0.2857142857142857 * 100 = **28.571428571428573 m^3**
8. Volume of aggregate = (4 / 7) * 100
9. Compute 4 / 7 = 0.5714285714285714

10. Multiply by 100 → 0.5714285714285714 * 100 = **57.142857142857146 m^3**

11. Cement mass = Volume_cement * ρ_cement

12. ρ_cement = 1440 kg/m^3 (user-provided)
13. Compute mass = 14.285714285714286 * 1440
14. Equivalent exact fraction method: (1440 * 100) / 7 = 144000 / 7 = **20,571.428571428572 kg**

15. Bags (50 kg) = 20,571.428571428572 / 50 = 411.42857142857144 → **round up → 412 bags**

16. Sand mass = Volume_sand * ρ_sand

17. ρ_sand = 1600 kg/m^3
18. Compute mass = 28.571428571428573 * 1600

19. Exact fraction: (1600 * 200) / 7 = 320000 / 7 = **45,714.28571428572 kg**

20. Aggregate mass = Volume_agg * ρ_agg

21. ρ_agg = 1750 kg/m^3
22. Compute mass = 57.142857142857146 * 1750
23. Exact fraction: (1750 * 400) / 7 = 700000 / 7 = **100,000 kg** → **100 tonnes**

BoM (base — no wastage/dry factor):

Material	Volume (m^3)	Mass (kg)	Bags/tonnes
Cement	14.285714	20,571.4286	412 bags (50 kg)
Sand	28.571429	45,714.2857	45.714 t
Aggregate	57.142857	100,000.0000	100 t

Notes: - Always show both mass and volume in BoM because some suppliers sell by m^3, some by weight. - Allow the user to apply wastage/dry factors post-calculation.

SR3 — Data model & storage (serverless-friendly)

Design for simple, schemaless storage (DynamoDB / Firebase / Supabase) and an object store for exports (S3).

Recommended tables/collections: 1. `Users` — userId, name, email (auth provider id), role, settings (default densities, units). 2. `Projects` — projectId, userId, name, location, createdAt, tags. 3. `Estimates` — estimateId, projectId, inputs (volume, mixRatio, densities, options), results (BoM JSON), costPlaceholder, createdAt, updatedAt. 4. `Materials` — materialId, name, defaultDensity, unit, supplierOverrides. 5. `Suppliers` — supplierId, name, contacts, API endpoints (Phase 2) 6. `Pricing` — supplierId, materialId, unitPrice, currency, lastUpdated

Keys & indexes (DynamoDB example): - `Users` PK: userId - `Projects` PK: projectId, GSI by userId for list retrieval - `Estimates` PK: estimateId, GSI by projectId for list retrieval

S3 buckets: - `app-exports-{env}` — CSV/PDF/XLSX stored with prefix /{userId}/{projectId}/{estimateId}/ - `app-logs-{env}` — only if storing logs externally (avoid storing PII)

SR4 — API design & contract (OpenAPI)

Design a small, RESTful API (or GraphQL if you prefer) with clear schema. Start with REST for simplicity.

Endpoints (minimal) - `POST /api/v1/estimate` — compute an estimate - Body: `{ volume_m3, mix_ratio: {cement, sand, agg}, densities: {cement, sand, agg}, options: {dry_factor, wastage_percent}, units }` - Returns: `{estimateId, inputs, results: { perMaterial: [...], totals }, links: {downloadCsv, downloadPdf} }` - `GET /api/v1/estimate/{id}` — retrieve saved estimate - `POST /api/v1/projects` — create project - `GET /api/v1/projects/{id}/estimates` — list estimates

OpenAPI snippet (skeleton):

```
openapi: 3.0.3
info:
  title: Concrete Estimator API
  version: 1.0.0
paths:
  /api/v1/estimate:
    post:
      summary: Compute material estimate
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/EstimateRequest'
      responses:
        '200':
          description: Successful estimate
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/EstimateResponse'
components:
  schemas:
    EstimateRequest:
      type: object
      properties:
        volume_m3:
          type: number
        mix_ratio:
          type: object
          properties:
            cement: {type: number}
            sand: {type: number}
            agg: {type: number}
        densities:
          type: object
          properties:
            cement: {type: number}
            sand: {type: number}
            agg: {type: number}
        options:
          type: object
    EstimateResponse:
      type: object
      properties:
        estimateId: {type: string}
        results: {type: object}
```

API auth: use JWT Bearer tokens with short expiry and refresh tokens; protect endpoints with IAM when internal.

SR5 — Calculation engine: code-first (isolated, testable component)

Write the estimation algorithm as a single, pure function with no external dependencies. This will be used in unit tests and invoked inside serverless functions.

Design rules: - Pure function: `estimateMaterials(inputs) -> results`. - Strict input validation: ensure numbers, non-negative, ratio parts > 0. - Deterministic rounding rules: define rounding strategy (e.g., ceil for bags, round to 3 decimals for masses). - Exported in multiple languages (TypeScript + Python) so both frontend (edge) and backend (lambda) can reuse.

Minimal TypeScript example (library)

```
export type MixRatio = {cement:number; sand:number; agg:number}
export type Densities = {cement:number; sand:number; agg:number}
export function estimateMaterials(volume_m3:number, mix:MixRatio,
densities:Densities, options?:{dryFactor?:number, wastagePercent?:
{cement?:number,sand?:number,agg?:number}}) {
  if (volume_m3 <= 0) throw new Error('volume must be > 0')
  const s = mix.cement + mix.sand + mix.agg
  const dryFactor = options?.dryFactor ?? 1.0
  const V = volume_m3 * dryFactor
  const v_c = (mix.cement / s) * V
  const v_s = (mix.sand / s) * V
  const v_a = (mix.agg / s) * V
  const m_c = v_c * densities.cement
  const m_s = v_s * densities.sand
  const m_a = v_a * densities.agg
  const bags_c = Math.ceil(m_c / 50)
  return {
    volumes:{cement:v_c, sand:v_s, agg:v_a},
    masses:{cement:m_c, sand:m_s, agg:m_a},
    bags:{cement:bags_c}
  }
}
```

Testing: - Unit tests for known cases (including the 100 m^3 sample). Use Jest (TS) / pytest (Python). - Property tests: random ratio & volume values; assert mass conservation and monotonicity.

SR6 — Serverless backend architecture (detailed)

Goal: serverless, low ops, secure. Core components (AWS example):

1. **API layer:** AWS API Gateway (HTTP API) or AWS App Runner with edge if needed. Use HTTP API for cost-effectiveness.
2. **Compute:** AWS Lambda (Node.js or Python) to host endpoints; consider Lambda@Edge or Cloudflare Workers for lower latency if global.
3. **Storage:** DynamoDB for metadata; S3 for exports
4. **Auth:** Amazon Cognito for user sign-up/sign-in, or external OIDC provider
5. **Secrets:** AWS Secrets Manager or Parameter Store + KMS
6. **Observability:** CloudWatch (logs, metrics), X-Ray for traces; optional Datadog/Sentry
7. **CDN/Frontend:** CloudFront + S3 or Vercel for Next.js
8. **Optional:** Step Functions for long-running processes (bulk exports), EventBridge for scheduled jobs (price sync)

Detailed steps to provision (AWS): - Create Cognito User Pool & App Client; configure OAuth flows. - Create API Gateway with JWT authorizer pointing at Cognito. - Implement Lambda function for `/estimate` with IAM role minimal permissions: `dynamodb:PutItem` (Estimates), `s3:PutObject` (exports) — no blanket *. - DynamoDB tables and S3 buckets with server-side encryption (SSE-KMS). - Configure CloudWatch Logs and log retention policy (30/90/365 days depending on compliance).

Cost controls: - Use concurrency limits or provisioned concurrency for critical endpoints. - Use DynamoDB on-demand for early stages; switch to provisioned capacity with autoscaling when predictable.

SR7 — Frontend: modern, friendly, accessible UI

Stack recommendation: Next.js (App Router) + TypeScript + Tailwind CSS + shadcn UI components + React Query (TanStack) + Vercel for hosting.

UX features: - Clear input form with presets (Class 20, Class 25, common mix ratios); helpful tooltips explaining each input. - Real-time estimate preview as user types (debounced) using the same estimation library compiled to WASM/TS so calculation runs client-side fast. - BoM table with toggles for units, wastage, and dry factor. - Save/Load projects, export buttons (CSV, PDF) and shareable links (signed URL with ttl). - Accessibility: keyboard navigation, ARIA labels, color contrast. - PWA support: manifest + service worker for offline viewing of saved projects.

Implementation notes: - Bundle the calculation engine as an npm package used by both frontend and backend to ensure parity. - For serverless functions, prefer to call a compute endpoint for saved projects and cost calculations; allow offline quick estimates purely client-side.

SR8 — Exports, reporting & formats

- Generate CSV client-side (fast) and PDF server-side for stable layout (use Puppeteer / headless chrome in a serverless container or use a specialized PDF microservice).
- XLSX export using `exceljs` on Node.js (lambda).
- Provide downloadable Smartsheet/CSV template compatible with Smartsheet import (user provided link earlier).

SR9 — Phase 2: Pricing integration (supplier data)

Goals: attach live prices to BoM and generate procurement-ready purchase lists.

Data sources: - Supplier APIs (preferred): fetch unit prices, lead times, MOQ - Manual CSV upload by supplier - Aggregator pricing (if available)

Design: - `Pricing` table storing {supplierId, materialId, unitPrice, currency, unit, lastUpdated} - Price refresh jobs: scheduled Lambda or EventBridge rule running every X minutes/hours. - For each estimate, compute total cost by matching material units and applying quantity discounts (if supplied). - UI: show cheapest supplier, nearest supplier (geolocation), and estimated delivery cost.

Security & business logic: - Supplier APIs: require secure API keys, store in Secrets Manager, rotate keys regularly. - Validate supplier data before accepting (sanity checks on prices, timestamps).

SR10 — Security hardening (every step)

Secure by default checklist: 1. TLS everywhere (HTTPS enforced), HSTS header via CDN (CloudFront). 2. Auth + Authorization: Cognito/Firebase with role-based access. 3. Input validation & sanitization — reject invalid numbers early. 4. Principle of least privilege for IAM roles. 5. Secrets management — use KMS & Secrets Manager. 6. WAF rules to block common attacks; rate-limiting via API Gateway usage plans. 7. Logging + audit trails; redact PII in logs. 8. Data encryption at rest (DynamoDB encryption or SSE-KMS for S3). 9. Security testing: SAST (GitHub CodeQL), DAST (e.g., OWASP ZAP in CI), and scheduled pen tests. 10. Compliance: store user data in-region (e.g., Kenya region if required) and maintain data retention policies.

SR11 — CI/CD and infra-as-code (complete pipeline)

CI/CD: GitHub Actions recommended.

Pipeline steps: - PR lint (ESLint/Prettier), typecheck (TS), unit tests. - Build (frontend), run integration tests against ephemeral environment. - Deploy using IaC tool (Serverless Framework, Terraform, or AWS SAM).

IaC examples: - **Serverless Framework** (simple): declare functions, resources (DynamoDB, S3, Cognito), deploy to AWS. - **Terraform**: provision full infra with modules for `api-gateway`, `lambda`, `dynamodb`, `cognito`, `s3`.

Recommended approach: Use Terraform for infra + GitHub Actions invoking `terraform apply` for production; Serverless Framework or `sam` for lambda code deploys in CI for quicker iteration.

SR12 — Testing strategy (comprehensive)

1. **Unit tests** for calculation library (TS & Py)
2. **Integration tests** for API endpoints (use localstack for DynamoDB & S3 emulation in CI)
3. **Contract tests** (OpenAPI schema validation)
4. **End-to-end tests** for critical user flows (Cypress / Playwright)
5. **Performance tests**: benchmark Lambda cold/warm starts, throughput using k6.
6. **Security tests**: automated SAST and DAST in CI.

SR13 — Observability & operations

- Logs: structured JSON logs (timestamp, requestId, userId anonymized, action, duration) to CloudWatch; optionally ship to centralized logs (Datadog/Elasticsearch).
- Traces: instrument main request path with AWS X-Ray or OpenTelemetry to see where latency is.
- Metrics: estimate requests per minute, errors, median response times, cost per request.
- Alerts: set up alerts for error rate > 1% for 15m, high latencies, and unusual traffic spikes.
- Playbooks: rollback procedures, CI rollback, and failover guidance.

SR14 — Cost optimization & scaling

- Use on-demand resources early (DynamoDB on-demand, Lambda pay-per-use).
- Set concurrency limits to prevent runaway costs; scale up with provisioned concurrency for critical endpoints.
- Use CloudFront for caching static exports.
- Monitor execution durations and DDB read/write units.

SR15 — UX polish, onboarding & docs

- Provide a guided onboarding wizard that demonstrates example estimates (including the 100 m³ case).
- Provide tooltips and engineering notes explaining assumptions (e.g., density defaults, dry factor).
- Offer export templates and an FAQ explaining differences between mass and volume and common local practices in Kenya.

SR16 — Deliverables & iteration plan (milestones)

MVP (4–6 weeks typical) — deliver: 1. Calculation library (TS + Py) with unit tests 2. Serverless `/estimate` API + DynamoDB storage 3. Next.js frontend with input form + BoM view + CSV export 4. Auth (Cognito) and basic security (WAF + TLS) 5. CI pipeline and basic monitoring

Post-MVP (Phase 2, 2–4 weeks) - Pricing integration (supplier APIs), cost summary - PDF export & printed-ready layout - More UX polish (PWA, offline)

Ongoing - Add supplier onboarding, analytics, and enterprise features (multi-user, roles, SSO)

SR17 — Example code snippets (production-ready starters)

Minimal Python Lambda handler (estimation only)

```python
import json
from decimal import Decimal

# Pure calculation function
def estimate(volume, mix, densities, dry_factor=1.0):
    if volume <= 0:
        raise ValueError('volume must be > 0')
    s = mix['cement'] + mix['sand'] + mix['agg']
    V = Decimal(volume) * Decimal(dry_factor)
    v_c = (Decimal(mix['cement'])/Decimal(s)) * V
    v_s = (Decimal(mix['sand'])/Decimal(s)) * V
    v_a = (Decimal(mix['agg'])/Decimal(s)) * V
    m_c = v_c * Decimal(densities['cement'])
    m_s = v_s * Decimal(densities['sand'])
    m_a = v_a * Decimal(densities['agg'])
    bags_c = (m_c / Decimal(50)).to_integral_value(rounding='ROUND_CEILING')
    return {
        'volumes': {'cement': float(v_c), 'sand': float(v_s), 'agg':
float(v_a)},
        'masses': {'cement': float(m_c), 'sand': float(m_s), 'agg': float(m_a)},
        'bags': {'cement': int(bags_c)}
    }

# Lambda handler
def lambda_handler(event, context):
    body = json.loads(event.get('body') or '{}')
    volume = body.get('volume_m3')
    mix = body.get('mix_ratio')
    densities = body.get('densities', {'cement':1440,'sand':1600,'agg':1750})
    res = estimate(volume, mix, densities, dry_factor=body.get('options',
{}).get('dry_factor',1.0))
    return {'statusCode':200, 'body': json.dumps({'estimate': res})}
```

Minimal Next.js API route (TypeScript)

```typescript
import { NextApiRequest, NextApiResponse } from 'next'
import { estimateMaterials } from '../../lib/estimator'
```

```
export default function handler(req:NextApiRequest, res:NextApiResponse) {
  if(req.method !== 'POST') return res.status(405).end()
  const { volume_m3, mix_ratio, densities, options } = req.body
  try {
    const result = estimateMaterials(volume_m3, mix_ratio, densities, options)
    return res.status(200).json({result})
  } catch(e:any) {
    return res.status(400).json({error: e.message})
  }
}
```

SR18 — Operational checklist before launch

1. IAM review & least-privilege enforcement
2. Rate limiting & WAF rules enabled
3. TLS certificate provisioned + HSTS & CSP headers
4. CI/CD pipeline configured for infra and app
5. Automated smoke tests post-deploy
6. Monitoring & alerts configured
7. Cost alarms for monthly spend

SR19 — Documentation & handover

- Developer docs: codebase README, architecture diagram, API docs (OpenAPI), infra docs (Terraform state location & secrets), runbooks.
- User docs: Quick start, BoM explanation, FAQ, pricing info for Phase 2.

SR20 — Optional advanced features (later phases)

- ML suggestions: recommend optimized mixes given local supplier materials & prices (requires historical data)
- Auto-detect local densities by user-submitted sample tests
- Generate procurement schedules by supplier lead time & project timeline
- Mobile-first offline app for field engineers (React Native or PWAs)
- Multi-language support (Swahili + English) and regional defaults (KES currency)

SR21 — Deliverable checklist (what you will receive when built)

1. Calculation library (TS & Py) with unit tests.
2. Serverless API + deployed Lambdas + DynamoDB + S3 resources (Terraform or Serverless config).
3. Next.js frontend deployed (Vercel/CloudFront).
4. GitHub repo with CI/CD and test coverage.
5. API docs (OpenAPI) and user docs.
6. Operational runbook (monitoring, rollback, scaling).

Appendix A — Quick reference defaults

- Default densities: cement 1440 kg/m^3, sand 1600 kg/m^3, aggregate 1750 kg/m^3
- Default bag size: 50 kg
- Dry volume factor (suggested default): 1.0 (user controlled), common practice uses 1.54 — *document this and make it configurable*

Appendix B — Next steps I can produce for you immediately

- A deployable starter repo (Next.js frontend + serverless estimate API + GitHub Actions).
- Terraform module for the AWS infra (API Gateway, Lambda, DynamoDB, S3, Cognito).
- The standalone calculation library as an npm package + pypi-ready package.

End of guide.

If you want, I can now generate one of the concrete artifacts from Appendix B (pick a starter repo, Terraform module, or the calculation library).