TECHNICALSTRAT / VIBECODE SERIES

Enterprise App Vibecode Recipe

Production-grade enterprise app on Next.js 15, PostgreSQL, and Azure Container Apps. Paste into Cursor and build.

A step-by-step plan to bootstrap a production-grade enterprise application on Next.js 15 + PostgreSQL + Azure Container Apps + Microsoft Entra SSO. Paste this entire file into Cursor as the initial plan and iterate phase by phase.

This is the exact stack used to ship two production enterprise apps (one commercial loan origination system, one multi-app internal platform) totaling ~350,000 lines of TypeScript, built solo in 8 months for $13,945 of Cursor spend.

Source: https://technicalstrat.com/articles/enterprise-app-vibecode-recipe

---

> **Note.** This recipe is a reference and a testimony of one IT operator's experience, not professional advice. The Azure architecture is where the author's operational confidence lives; the application-code patterns were vibe-coded with Cursor as starting points to adapt rather than turnkey production code. Set Azure budget alerts before you provision and watch your Cursor spend before any long agent session. Any cloud bills, AI bills, or production outcomes from your build are yours to own. TechnicalStrat and the author assume no liability for them.

Prerequisites (do this before step 1)

- Azure subscription with Owner permissions
- Microsoft Entra ID tenant with Global Admin
- GitHub organization (or personal account) you will configure with OIDC to Azure
- Node.js 20+ and Docker Desktop installed locally
- Cursor with access to Anthropic frontier models
- A domain name (optional but recommended for Entra Application Proxy or custom ingress)
- An honest answer to: "will any non-tenant user need to reach this app?" - drives the exposure-model decision in Phase 8

Phase 1: Azure foundation

1. Create resource group `myapp-prod-rg` in your preferred region (e.g. `westus` if you live near the West Coast; pick the region closest to your Microsoft 365 tenant home region for lower Graph latency).
2. Set an Azure subscription budget of **$300/month** with a **$200 warning** alert routed to your email and a hard alert at 100%. Do this before provisioning anything else.
3. Create a Key Vault `kv-myapp-prod` with **RBAC authentication** enabled (not the legacy access-policy model) and purge protection on.
4. Create an Azure Container Registry `acrmyappprod` (Premium SKU if you want private endpoints and geo-replication; Basic is fine to start). Disable admin user; we will pull via managed identity.
5. Provision an Azure Database for PostgreSQL **Flexible Server** `psql-myapp-prod`, version 16, Burstable B2s for dev, General Purpose D2s_v5 for prod. Disable public access; we will use private endpoints. Set storage autogrow to on.
6. Provision an Azure Cache for Redis `redis-myapp-prod` (Basic C0 for dev, Standard C1 minimum for prod so you get replication).
7. Create an Application Insights workspace `appi-myapp-prod`. Connect it to a Log Analytics workspace in the same resource group.
8. Create a Container Apps Environment `cae-myapp-prod` using the Consumption workload profile. If you need internal-only ingress, set the environment to internal and plan for Entra Application Proxy.
9. Generate the Postgres and Redis connection strings. Store them in Key Vault as `DATABASE-URL` and `REDIS-URL`. Never paste either into a file the AI can see locally.
10. Create a user-assigned managed identity `mi-myapp-prod`. Grant it **AcrPull** on the registry and **Key Vault Secrets User** on the vault. The Container App will use this identity to pull images and read secrets.
11. Create a Storage Account `stmyappprod` with hierarchical namespace off (unless you need ADLS Gen2). Create the containers your app actually needs at launch (common starters: one for user uploads, one for backups). Public access disabled at the account level. Add the account name and primary key to Key Vault as `STORAGE-ACCOUNT` and `STORAGE-KEY`.

Phase 2: Microsoft Entra ID

12. In Entra, create two app registrations:
    - `myapp-sso` for NextAuth-based interactive sign-in
    - `myapp-graph` for Microsoft Graph application access (only if you need Graph API, SharePoint, or Outlook integration)
13. For `myapp-sso`: add redirect URI `https://<your-prod-host>/api/auth/callback/azure-ad` and a localhost URI for dev. Grant delegated permissions `openid`, `profile`, `email`, `User.Read`. Configure ID token claims to include `email` and `groups`.

14. For `myapp-graph`: grant **application** permissions for only the Graph scopes you actually need (e.g. `Mail.Send`, `Sites.Selected`, `User.Read.All`). Admin-consent each one. Use `Sites.Selected` instead of `Sites.ReadWrite.All` whenever possible.
15. Generate client secrets for both app registrations. Store them in Key Vault as `AZURE-AD-CLIENT-SECRET` and `GRAPH-CLIENT-SECRET`. Set secret expiry to 24 months and put the rotation date on your calendar.
16. Create two cloud-only break-glass Global Admin accounts (e.g. `breakglass-1@<tenant>.onmicrosoft.com`). 64-character random passwords printed and stored in a physical safe. Two FIDO2 keys registered to each. Exclude both accounts from every Conditional Access policy you will ever write.
17. Create a `BreakGlass-Exclude` Entra security group containing both break-glass accounts. Add this group to the Exclude tab of every Conditional Access policy.

Phase 3: Repository scaffold

18. Initialize a Next.js 15 project with App Router and TypeScript: `npx create-next-app@latest myapp --typescript --app --tailwind --turbopack`.
19. Add core dependencies: `prisma`, `@prisma/client`, `next-auth@4`, `@azure/identity`, `@azure/keyvault-secrets`, `@azure/storage-blob`, `ioredis`, `zod`, `@microsoft/applicationinsights-web`, `@microsoft/applicationinsights-react-js`, `winston`, `class-variance-authority`, `lucide-react`.
20. Add dev dependencies: `@types/node`, `eslint`, `prettier`, `prettier-plugin-tailwindcss`, `vitest` (or `jest` if you prefer; the AI is fluent in both), `playwright`.
21. Initialize Prisma: `npx prisma init --datasource-provider postgresql`. Define a `User` model in `prisma/schema.prisma` with the fields you actually need (`id`, `email`, `name`, `role`, `entraObjectId`, timestamps). Skip the `Account`, `Session`, and `VerificationToken` models unless you decide to use the NextAuth Prisma adapter in step 27. Then start on your domain models.
22. Run `npx prisma migrate dev --name init` against your local Postgres (Docker Compose for local dev is fine).
23. Create `lib/db.ts` as a Prisma client singleton that survives Next.js hot reload. Configure connection pooling for serverless if you ever move off Container Apps.
24. Create `lib/redis.ts` as an ioredis singleton with `lazyConnect: true` and a graceful shutdown handler.
25. Create `lib/keyvault.ts` that uses `DefaultAzureCredential` to pull secrets at boot in production, and falls back to `process.env` locally. Cache pulled secrets in memory for the process lifetime.
26. Create `lib/env.ts` that validates required environment variables with Zod at startup. Fail loud and fail fast if anything is missing.
27. Configure NextAuth at `app/api/auth/[...nextauth]/route.ts` with the Azure AD provider and the **JWT session strategy** (no Prisma adapter). In the `signIn` callback, upsert the `User` row in Postgres on first login (keyed by Entra object ID), set the role from your default-role logic, and reject sign-ins from outside your allowed tenant. In the `jwt` callback, copy `userId` and `role` from the database onto the token so every request has them without a DB hit. Only add the Prisma adapter later if you specifically need DB-

persisted `Account` / `Session` / `VerificationToken` rows (most apps do not - pure JWT plus a manual upsert is simpler and three fewer tables to keep in sync).

Phase 4: Authorization and audit

28. Create `lib/permissions.ts` with a `Role` enum (e.g. `ADMIN`, `MANAGER`, `USER`) and a `can(user, action, resource)` function. Drive permissions from a single capability map; do not scatter `if (user.role === ...)` checks across handlers.
29. Add an `AuditLog` Prisma model with fields: `id`, `actorId`, `action`, `resourceType`, `resourceId`, `before` (Json), `after` (Json), `requestId`, `createdAt`.
30. Write `lib/with-auth.ts` exporting `withAuth(handler, requiredPermission)` that authenticates the request via NextAuth, checks the permission, attaches the user to the request, writes an audit log entry on successful mutations, and returns 401/403 with structured errors otherwise.
31. Apply `withAuth` to every route handler. The AI will try to skip it on "internal" routes. There are no internal routes.

Phase 5: API layer

32. Create your first API route under `app/api/<resource>/route.ts`. Export `GET`, `POST`, `PUT`, `PATCH`, `DELETE` as needed. Each handler is wrapped in `withAuth` and `withErrorHandling`.
33. Use Zod schemas in `lib/schemas/<resource>.ts` for both request validation and response shaping. Share these schemas with the frontend; do not duplicate them.
34. Add `lib/api-error.ts` with a typed `ApiError` class and a `withErrorHandling()` middleware that catches it, logs to App Insights with the request ID, and returns a structured error response.
35. For every third-party API you call out to, create `lib/integrations/<provider>/client.ts` with a single client module that owns auth, retries (use `p-retry` or write your own with exponential backoff), timeouts, and a typed surface.
36. For every webhook you receive, create `app/api/webhooks/<provider>/route.ts`. Validate HMAC signatures or vendor-provided signatures before doing any work. Persist the raw payload to an `IngestedWebhook` table for replay. Process asynchronously via an outbox/drain pattern, not inline.
37. Build a generic outbox pattern: every external side-effect writes an `OutboxEvent` row in the same database transaction as the business mutation. A scheduled Container Apps Job drains the outbox every minute, delivers the event, and marks it sent. Failed events get exponential backoff and end up in a dead-letter view after N retries.

Phase 6: Frontend

38. Build the application shell: top nav, optional sidebar, content area, toast notifications. Use Radix UI primitives or shadcn/ui. Skip CSS modules and styled-components.
39. Wire a TanStack Query (or SWR) wrapper around `fetch` with cache keys per resource and optimistic updates for mutations.
40. For forms, use `react-hook-form` with the same Zod schemas as your API routes (via `@hookform/resolvers/zod`). Errors render from the same source of truth on both ends.
41. Add a `<DataTable>` component built on TanStack Table with server-side pagination, sort, and filter via search params. Reuse it across every list view in the app; do not let the AI build a bespoke table per page.
42. Add an in-app feedback widget on the layout that posts to `/api/feedback`. Persist submissions to an admin queue table with severity, route, the submitting user, and an optional screenshot. Review the queue weekly. This widget pays for itself the first time a real user reports a real bug from the page they were on.

Phase 7: Containerization

43. Write a multi-stage `Dockerfile`:
    - Stage 1 (deps): `npm ci --omit=dev`, then a separate `npm ci` for build
    - Stage 2 (builder): `npx prisma generate`, `npm run build` (Next.js with `output: 'standalone'`)
    - Stage 3 (runtime): Node 20 slim, copy `.next/standalone`, `.next/static`, `public`, `prisma`. Run as a non-root user.
44. Write `Dockerfile.migrations` that copies only `prisma/`, `package.json`, and `package-lock.json`, runs `npm ci`, then `CMD ["npx", "prisma", "migrate", "deploy"]`.
45. Add `/api/health` that returns 200 with a JSON body containing the result of: a `SELECT 1` against Postgres, a `PING` against Redis, and any critical-integration readiness probe. Return 503 with the failing dependency on failure.
46. Build both images locally to verify: `docker buildx build --platform linux/amd64 -t acrmyappprod.azurecr.io/myapp:local .` and the equivalent for migrations.
47. Push to ACR using the Azure CLI: `az acr login --name acrmyappprod` then `docker push`.

Phase 8: Container Apps deployment

48. Create the Container App `ca-myapp-prod` referencing the image in ACR. Set scale rules: min replicas 1, max replicas 5, HTTP scaler at 10 concurrent requests per replica. CPU 0.5, memory 1.0Gi to start.
49. Configure the Container App to authenticate to ACR using the user-assigned managed identity `mi-myapp-prod` (not admin credentials, not static service principal).

50. Bind every Container App secret as a Key Vault reference: `secretref:database-url` resolves to the Key Vault secret via the managed identity. The container env vars then reference the secret names. No secret values leave Key Vault.

51. Create a Container Apps Job `job-migrations-prod` that runs the migrations image once on demand. CI will trigger this with `az containerapp job start` before each app deploy and poll until success or timeout (3 min).

52. For each scheduled background task (outbox drain, cron jobs, periodic syncs), create a Container Apps Job with a cron trigger. Define them in a single `config/cron-jobs.json` file. Write a small bash or PowerShell script (`scripts/deploy-cron-jobs.sh`) that PUTs each job idempotently via `az rest` . Run this script from CI whenever the file changes.

53. **Decide your production exposure model.** This is the single most consequential decision after the stack itself. Pick exactly one of the two paths and commit to it:

    Path A - Internal-only behind Microsoft Entra Application Proxy (recommended if every user authenticates against your Entra tenant):

    - Set Container Apps Environment ingress to **internal**.
    - Set the Container App ingress to **internal** and confirm it has no public DNS entry.
    - Provision a small Windows Server VM in the same VNet to run the App Proxy connector. Join it to the connector group for your tenant.
    - In Entra: **Enterprise Applications > Application Proxy > Configure an app**. Set the internal URL to the Container Apps internal hostname, set the external URL to a subdomain of `<tenant>.msappproxy.net` (or a verified custom domain), set pre-authentication to **Microsoft Entra ID**.
    - Apply a Conditional Access policy to the App Proxy application: require MFA, require compliant device, restrict to known IPs if appropriate.
    - Add the per-user Application Proxy license to every user who needs to reach the app. Budget the recurring cost.
    - Result: VPN-grade isolation, pre-authentication at the edge, conditional access enforced before any request touches your code. The right call when you are not certain of your app's security grade and every user is inside your tenant.

    Path B - Public ingress with a managed certificate (required if any non-tenant users, customer portals, magic-link flows, or partner access must reach the app):

    - Set Container Apps Environment ingress to **external** (or use a workload profile with external ingress).
    - Set the Container App ingress to **external**, allow insecure connections to **false**, traffic split 100% to latest revision.
    - Configure a custom domain and let Container Apps provision a managed certificate. Add the verification TXT record to your DNS.

- Document every unauthenticated route in `docs/public-surface.md`. Keep this file under version control. Any PR that adds a new unauthenticated route must update it.
- **Budget a paid third-party security review before launch**. A senior developer should sign off on the auth flows, the magic-link or portal flows, the rate limits, the input validation, and the audit posture. If you cannot afford the review, do not ship publicly.

Phase 9: GitHub Actions CI/CD with OIDC

54. Configure GitHub-to-Azure federated OIDC credentials on a service principal scoped to the resource group. Add `AZURE_CLIENT_ID`, `AZURE_TENANT_ID`, `AZURE_SUBSCRIPTION_ID` as plain GitHub variables (not secrets, they are not sensitive). No `AZURE_CLIENT_SECRET` anywhere.
55. Write `.github/workflows/ci-cd.yml` with these jobs:
    - **Quality (parallel matrix)**: lint, type-check, unit tests with coverage, `npm audit --omit=dev --audit-level=high`, Prettier check, Prisma format check.
    - **Build**: `docker buildx build` with GitHub Actions cache for both app and migrations images. Tag with `${{ github.sha }}`. Save as artifacts.
    - **E2E (Playwright)**: smoke tests against a temporary preview deployment or a local server. Required for prod, optional for dev.
    - **Deploy (branch-gated)**: dev branch deploys to dev environment; main branch deploys to prod and requires E2E green. OIDC login, push images to ACR, run the migrations job and poll, update the Container App image, health-probe `/api/health` 3 times on dev and 5 times on prod, redeploy cron jobs if `config/cron-jobs.json` changed.
56. Protect `main`: require all checks green, require at least one review (even if you are solo: review your own PR before merging - this is the moment to read the diff carefully).
57. Configure a non-blocking CodeQL security scan on push and weekly schedule. Triage findings before launch and at every sprint boundary.

Phase 10: Observability

58. Wire the Application Insights server SDK in a root server component or a `lib/telemetry-server.ts` module that initializes once. Wire the browser SDK in `app/layout.tsx` via a client component boundary.
59. Add a Winston logger with an Application Insights transport. Every log line carries `requestId`, `userId`, `route`, and (for mutations) `auditId`.
60. Create App Insights workbooks for: request rate by route, P95 latency by route, exception rate, top custom events, outbox queue depth, dead-letter count, Conditional Access challenge rate.
61. Configure alerts: 5xx error rate > 1% over 15 minutes, P95 latency > 2 seconds over 15 minutes, outbox dead-letter count > 0 over 5 minutes, health endpoint failing for > 5 minutes. Route alerts to an email and

an on-call Teams channel.

Phase 11: Cost discipline

62. Use Container Apps consumption pricing for dev. Only move to a dedicated workload profile when load justifies it.
63. Set the dev Postgres to auto-pause after 1 hour idle. Schedule the dev Container App to scale to 0 replicas overnight if your dev usage allows.
64. Review the Cost Management blade weekly for the first month. Annotate anything anomalous in `docs/cost-log.md` so the next surprise has historical context.
65. Tag every resource with `env`, `app`, `owner`, `cost-center`. Filter the Cost Management view by tag and you have per-feature cost attribution.

Phase 12: The vibe coding loop

66. Write a `CLAUDE.md` (or `AGENTS.md`) at the repo root. Two pages, max. Cover:
    - The stack and its rationale
    - Naming conventions for files, directories, models, and routes
    - The `withAuth` wrapper, the `lib/permissions.ts` model, the audit log pattern
    - The outbox pattern and where to add new integrations
    - The "do not" list: do not skip Zod validation, do not write raw SQL, do not add a new background job without registering it in `config/cron-jobs.json`, do not bypass `withAuth`.
67. For every new feature, write the spec in plain English at the top of `docs/specs/<feature>.md` before opening a Cursor chat. One paragraph of intent, one bullet list of acceptance criteria. Reference it in the chat.
68. Use Claude Opus in extended thinking mode for architecture, refactors, schema changes, and anything touching auth or permissions. Use Sonnet for boilerplate, commit messages, and small typed refactors. Use a non-thinking model for trivia.
69. After every significant feature: run `npx tsc --noEmit`, run your test suite, run `npm audit --omit=dev --audit-level=high`. Do not trust the model's "this should work."
70. Commit small, commit often. Use feature branches. Let the model draft commit messages, then read them and edit.

Phase 13: Hardening before launch

71. Configure Conditional Access policies in Entra:

- **Block legacy authentication** (Microsoft-managed)
- **MFA for admins** (Microsoft-managed)
- **Phishing-resistant MFA for any user touching your app's admin surface** (custom)
- **Require compliant device for production users** (custom; requires Intune compliance policies) Ship each in report-only mode for a week, watch the Insights dashboard, then enforce. Always exclude `BreakGlass-Exclude`.

72. Run a manual auth penetration pass: missing access controls, IDOR via predictable IDs, missing CSRF on POST routes that bypass NextAuth's protection, session fixation, cookie security flags.
73. Run `npm audit --omit=dev --audit-level=high` as a blocking CI gate. Resolve every high or critical before launch.
74. Review every external endpoint for: input validation, rate limiting (a sliding-window limiter backed by Redis works well; many npm options exist, or write a small Lua script if you want zero dependencies), audit logging, structured errors.
75. Write the runbook at `docs/runbook.md`:
    - How to roll back a deploy (revert the Container App revision pointer)
    - How to rotate a secret (Key Vault new version, restart the app)
    - How to add a user (admin UI flow + the manual fallback)
    - Where every alert routes and who is on call
    - The break-glass procedure (where the safe is, who has access)
76. Launch.

What you have when you finish

- Next.js 15 app on Azure Container Apps with managed-identity ACR pulls
- PostgreSQL Flexible Server with Prisma migrations gated through a Container Apps Job in CI
- Redis cache for sessions, rate limits, and lightweight queues
- Microsoft Entra ID SSO with role-based authorization, audit logging, and Conditional Access
- GitHub Actions CI/CD with OIDC federation, branch protection, and Playwright E2E
- Application Insights distributed tracing with custom workbooks and alerts
- Key Vault for every secret, no long-lived credentials in CI or in code
- Cost ceiling enforced by a real Azure budget with alerts
- A `CLAUDE.md` that makes every future Cursor session productive on day one

The recipe in numbers

Two real apps built on this exact stack:

- **App A**: a commercial loan origination system, ~150,000 to 180,000 lines of TypeScript. Public ingress with a paid third-party security review before launch (Path B above).
- **App B**: a multi-app internal platform, ~250,000 lines of TypeScript. Deployed behind Microsoft Entra Application Proxy with internal ingress (Path A above).

Build window: October 2025 to May 2026, single developer, $13,945 in Cursor spend, 450 active coding hours at an effective rate of ~$31 per active hour. The majority of the spend went to frontier Anthropic Opus-thinking models.

The recipe is the path. Yours to run.

Source article: https://technicalstrat.com/articles/enterprise-app-vibecode-recipe

---