AI Agent – Actionable Build Plan (Two-Repo Model)

*This document now assumes **Option A**: the existing company-website repo on **Vercel** stays untouched, while a **new backend repo** (**`accessibility-scanner-backend`****) is created to host the API service, background worker, queue and database—all deployed to **Railway**. The steps below describe only the **backend repo**** o****. A short integration guide for calling the API from the website is included at the end.*

Phase 0 – Meta Setup (½ day)

#	Task	Output	Acceptance Criteria
0.1	**Create GitHub repo** `accessibility-scanner-backend` (public/private).	Repo initialised	MIT LICENSE & `README.md` scaffolded
0.2	**Enable Issues & project board** (Backlog → In Progress → Done).	Project board	Board seeded with this plan's tasks
0.3	**Configure Husky** with lint-staged & Conventional Commits.	`.husky/` scripts	`git commit` blocked on lint failure

Phase 1 – Repo Scaffolding (1 day)

#	Task	Directory	Key Files	Acceptance Criteria
1.1	Init **pnpm workspace**	`/`	`package.json`, `pnpm-workspace.yaml`	`pnpm install` succeeds
1.2	Create apps: `apps/api`, `apps/worker`	folders	baseline `package.json`, TS config	`pnpm -r build` passes
1.3	Add root ESLint, Prettier, TypeScript v5 strict	`/.eslintrc.js`	`pnpm lint` 0 errors	

Phase 2 – Infrastructure as Code (1½ days)

#	Task	Files	Acceptance Criteria
2.1	**Docker Compose (local)**: `api`, `worker`, `redis`, `postgres`	`docker-compose.yml`	`docker compose up` healthy

#	Task	Files	Acceptance Criteria
2.2	**Railway Terraform** module provisioning Service (api), Job (worker), Postgres, Redis.	`infra/railway.tf`	`terraform plan` no drift
2.3	**GitHub Actions CI** – on push:• install deps• type-check & jest• docker build api & worker• push to **GHCR**• deploy to Railway via `railway up`.	`.github/workflows/ci.yml`	PR green checks

Phase 3 – Backend API (3 days)

#	Task	Endpoint	Acceptance Criteria
3.1	Scaffold **Express** w/ `ts-node` in `apps/api`. Health `/health`.	Returns `{ ok: true }`	
3.2	Add **Zod** schemas for request/response.	`src/validation/`	Invalid URL → 400
3.3	**POST **`/api/scan-website` → enqueue BullMQ job, return `{ jobId }` (UUID).	Unit test passes	
3.4	**GET **`/api/scan-status/:jobId` returns `QUEUED	RUNNING	COMPLETED	FAILED`.	Enum validated
3.5	**GET **`/api/scan-results/:jobId` returns aggregated results from Postgres.	Mock until Phase 4	
3.6	Add **rate-limiter** 10 req/min/IP using Redis backend.	Exceeding limit → 429	

Phase 4 – Worker Service (4 days)

#	Task	Description	Acceptance Criteria
4.1	Bootstrap `apps/worker` – connect to BullMQ queue.	Worker logs ‘ready’.	
4.2	On job: launch **Puppeteer** (headless Chromium) with 30 s nav timeout.	`page.title()` non-empty	

#	Task	Description	Acceptance Criteria
4.3	Inject **axe-core**; run scan.	Violations array returned	
4.4	Compute **score** 0-100 via severity weights.	Unit test: 0 violations → 100	
4.5	Persist to Postgres (`jobs`, `scans`, `issues`) via Prisma.	Rows exist	
4.6	ACK or mark FAILED; auto-retry ≤2.	Failed job visible in Bull UI	
4.7	Concurrency via env `MAX_WORKERS` (default 2).	Parallelism scales	

Phase 5 – Observability (¾ day)

#	Task	Tech	Acceptance Criteria
5.1	**Pino** structured logs to Railway Log viewer.	JSON lines displayed	
5.2	**Prometheus exporter** (`/metrics`) exposing job latency, queue depth.	Curl returns metrics	
5.3	**Grafana Cloud** free tier: scrape/export; dashboard in `observability/`.	Graph shows `scan_duration_seconds`	

Phase 6 – Documentation & DX (¾ day)

#	Task	Output	Acceptance Criteria
6.1	Generate **OpenAPI 3** spec (`openapi.yaml`).	Swagger UI renders	
6.2	Update `README.md` with local dev, Railway deploy, env vars.	New dev up in ≤15 min	
6.3	Seed **Changelog.md** (Keep-a-Changelog).	Version 0.1.0 entry	

Phase 7 – Deployment & Smoke Test (½ day)

#	Task	Acceptance Criteria
7.1	**CI deploy**: build & push images → Railway (api & worker).	Services healthy
7.2	Run smoke scan against `https://example.com`.	Status completes; JSON returned

Website-Repo Integration Guide (½ day)

Step	File	Action
1	`src/routes/+page.svelte` (landing)	`fetch("https://api.your-scanner.com/api/scan-website", { method:"POST", body: JSON.stringify({ url }) })` → receive `jobId` → navigate to `/scanResults/[jobId]`.
2	`src/routes/scanResults/+page.svelte`	On load, poll `/api/scan-status/{jobId}` every 3 s; once `COMPLETED`, fetch `/api/scan-results/{jobId}` and render dashboard.
3	Add `.env.public` key `VITE_SCANNER_API=https://api.your-scanner.com`.	Build passes; runtime uses env

No backend code lives in the website repo; only these two fetch calls are added.

Additional Non-Functional Notes

- **Hosting:** API & worker on Railway containers; Redis & Postgres Railway add-ons. Frontend remains on Vercel (no changes to its pipeline).
- **Queue:** BullMQ on Redis Streams.
- **Data retention:** 30-day rolling delete cron (Railway).
- **Concurrency limit:** Start 2 workers (≈50 scans max). Scale via Railway slider.
- **Budget:** Railway starter + existing Vercel plan; no new cost on website repo.

Definition of Done

- User visits company site → submits URL → sees real scan results in ≤60 s.
- Backend repo: tests & lint pass; automatic deploy to Railway.
- Website repo: env var points to API; simple fetch integration works.