AI-Powered Architecture Assistant — Complete Guide (SaaS MVP)

Target user: Developers and small teams who want automated, accurate architecture documentation and task planning for their codebases.

MVP focus: A polished, consumer-facing SaaS that connects to a GitHub repo and produces: - `architecture.md` — full codebase architecture - `task.md` — granular, testable step-by-step plans for new features

The MVP must be small, reliable, and demonstrably valuable: generate high-quality architecture docs and task plans for small-to-medium repos, with a smooth onboarding flow and GitHub integration.

---

1) Project recap (short)

Idea: A web SaaS where developers connect their GitHub repo and receive automatically generated `architecture.md` and `task.md` files. It can optionally annotate PRs, surface architecture drift, and provide task templates that can be handed to LLM coding agents.

Why it sells: Teams waste time documenting and scoping work. Automating high-quality, consistent docs + task breakdowns is time saved and a measurable productivity win.

MVP scope: - Repo OAuth + read-only access - Parse repo (file tree) and analyze source files (JS/TS/Node, common frameworks) - Generate `architecture.md` (folders, responsibilities, DB models, service layers, where state lives) - Generate `task.md` for a single requested feature using `architecture.md` as context - Web UI to view/save docs and download - Billing: free tier + paid plan (later)

---

2) Should the dev workflow be part of the product?

Yes. The developer-facing workflow you described is *the product's value proposition*. Make these capabilities available as features: - **One-click architecture snapshot** — run anytime, store versions - **Task planner** — produce `task.md` from natural-language feature descriptions - **PR/commit awareness** — surface differences between snapshots and highlight drift - **Agent-friendly outputs** — outputs formatted specifically for downstream LLM agents (minimal hallucination risk, includes file paths and lines-to-change guidance)

This makes the tooling both an internal dev helper and the product that others will pay for.

---

3) Tech stack (recommended for you)

- **Frontend**: React + TypeScript (Vite or Next.js). Use component library (Tailwind + shadcn/ui recommended).

- **Backend**: Node.js + TypeScript, use **NestJS** for structure (optional) or Express + TypeScript if you prefer lighter. Deploy as serverless or containerized service.
- **Database/Auth**: **Supabase** (Postgres + Auth) — fast to iterate, gives you Postgres, Auth, Storage, and row-level policies.
- **LLM provider**: OpenAI APIs (or other providers when needed) — start with GPT-4.1/4o if available for best results. Keep provider-agnostic layering.
- **Storage**: Supabase Storage or S3 for generated docs and snapshots.
- **CI/CD**: GitHub Actions
- **Payments**: Stripe
- **Testing**: Jest + Testing Library (frontend), Vitest or Jest (backend)
- **Infrastructure options**: Vercel for frontend, Supabase Edge Functions / Render / Fly.io for backend as needed.

Rationale: This keeps Node as your core, speeds development via Supabase, and lets you focus on model prompts and UX.

---

4) High-level architecture

```
[ User (browser) ] <-- HTTPS --> [Frontend (React TS) ]
                                        |
                                        v
                                  REST / GraphQL
                                        |
[ GitHub OAuth ] <--> [ Backend (Node TS) + LLM layer ] <--> [Supabase
(Postgres/Auth/Storage) ]
                                        |
                                        v
                                  External LLMs (OpenAI)
```

Components: - **Frontend**: App shell, projects list, repo onboarding flow, document viewer/editor, task generator UI, billing page, settings - **Backend API**: Auth/session handling, GitHub webhook handlers, repo fetch + parsing, LLM orchestration (prompt engineering + caching), doc generation pipeline, task manager - **Worker queue**: For heavy repo analysis and LLM jobs (BullMQ/Redis or Supabase background jobs) - **DB**: Projects, repos, snapshots, users, billing, task templates - **Storage**: Generated `.md` files and snapshot diffs

---

5) Folder + file structure (monorepo suggestion)

```
/ (root)
  /apps
    /web        -> React frontend (Next.js or Vite)
    /api        -> Node backend (NestJS/Express)
  /packages
    /common     -> shared types, prompt templates, utils
    /llm        -> LLM orchestration helpers
  .env
```

```
README.md
docs/
  architecture.md (example)
  task.md
```

---

6) Database models (Postgres / Supabase)

Key tables (simplified):

users - id (uuid) - email - name - role - stripe_customer_id

projects - id - user_id - name - repo_full_name - github_installation_id - visibility (private/public)

snapshots - id - project_id - generated_at - architecture_md_path - data (json metadata: file tree, language stats)

tasks - id - project_id - snapshot_id (optional) - description - task_md_path - status

billing - id - user_id - plan - status

Add indexes on `project.repo_full_name`, `user_id`, and `snapshot.generated_at`.

DB relations should be straightforward foreign keys (user → projects → snapshots → tasks).

---

7) Where state lives + how services connect

- **Frontend state**: ephemeral UI state in React (use React Query / TanStack Query for server state). Keep no long-lived secrets client-side.
- **Server state**: persistent state lives in Postgres (Supabase). LLM jobs and long-running processes are stored as `snapshots` and `jobs` records.
- **Connections**:
- GitHub OAuth → backend stores installation id and uses GitHub REST API to fetch files.
- Backend fetches repo contents (file tree) → stores metadata in `snapshots` → orchestrates LLM calls to generate files → saves `architecture.md` and `task.md` to Storage and snapshot record.
- Webhooks notify on pushes/PRs to trigger snapshot diffs.

---

8) Service layers (back-end breakdown)

- **AuthService**: handles JWT sessions + Supabase auth bridging.
- **GitHubService**: handles OAuth, installation tokens, repo read operations.
- **RepoParserService**: traverses files, detects languages/frameworks, extracts key files (package.json, tsconfig, prisma/schema, dockerfile, dbcontext.cs if C# exists), and generates a normalized file-tree JSON.

- **LLMService**: central orchestrator for prompts, rate-limiting, retries, and response caching. Accepts higher-level tasks (generateArchitecture, generateTaskPlan).
- **SnapshotService**: stores and versions generated artifacts.
- **TaskService**: CRUD for tasks, and interfaces to LLMService for task generation.
- **BillingService**: Stripe webhooks and plan logic.

---

9) Prompt design & orchestration

Design canonical prompt templates stored in `/packages/common/prompts`. Example: `architecture_prompt.v1` and `task_prompt.v1`. Always include structured instructions and a strict output format (markdown with specified headings). Example instructs LLM to: list folders, describe each file, derive DB models, list services, and mark uncertainties.

Safety against hallucinations: include file paths and code snippets as evidence in the prompt. Ask LLM to output `UNCERTAINTIES` section where it lists things it couldn't be certain about.

---

10) Testing strategy

- **Unit tests**: Jest (backend), Vitest (frontend) — test parsing logic, prompt template rendering, and small utilities.
- **Integration tests**: Mock GitHub API flows (nock) and stub LLM responses for deterministic tests. Test that `RepoParserService` → `LLMService` → snapshot persistence works.
- **E2E**: Cypress or Playwright for main user flows (connect repo, generate doc, download).

Testing should be required for any production commit.

---

11) Coding conventions & rules (naming, style)

- TypeScript strict mode on (`tsconfig.strict = true`).
- Files: `kebab-case` for folders, `camelCase` for functions, `PascalCase` for React components and types/interfaces.
- API routes in `api/` use `snake_case` for DB columns and `camelCase` for JS objects with clear mapping.
- Commit messages: Conventional Commits (feat/fix/docs/chore).
- PRs: small, one-concern, include unit tests.

Add these rules to `architecture.md` as part of the project conventions.

---

12) `architecture.md` template (what your agent should output)

Include a template in repo `docs/architecture-template.md` that the service uses. Example headings:

- Project name + summary
- Tech stack
- File & folder structure (tree)
- Responsibilities (what each folder/file does)
- DB models & relations (ER diagrams or textual tables)
- Service layers & API surface
- Where state lives (client vs server vs DB)
- Testing framework & CI
- Deployment & infra notes
- Uncertainties / next steps

---

13) `task.md` template (what your agent should output)

Task must be granular and testable. Example structure:

- Title
- Scope (start & end)
- Preconditions (what must exist before starting)
- Files to change (paths)
- Step-by-step plan (numbered)
- Tests to add (unit/integration)
- Acceptance criteria (pass tests + manual checks)

---

14) Minimum viable feature list (MVP)

1. User sign-up/login (Supabase) + project onboarding (connect GitHub repo)
2. Repo parsing (small repos) + small language detection
3. Generate `architecture.md` and display it in UI
4. Generate `task.md` for one feature request
5. Store snapshots and allow download
6. Basic billing UI (placeholder for Stripe) — optional for MVP
7. Tests & CI

---

15) Example first task (auto-generated `task.md`)

Feature: "Project onboarding and generate initial `architecture.md`"

- Scope: Connect a GitHub repo and generate `architecture.md` for it.
- Start: User has OAuth access and selects a repo.

- End: `architecture.md` generated, saved to project snapshots, and UI shows generated file.
- Steps (high level):
- Implement GitHub OAuth flow + store installation id.
- Backend endpoint to fetch file tree + important files.
- RepoParserService that normalizes file list.
- Implement `LLMService.generateArchitecture(snapshot)` using prompt template.
- Save output to Storage and DB snapshot record.
- UI: onboarding flow and doc viewer.
- Tests: unit tests for parser and prompt formatting; integration test mocking LLM.

---

16) Deployment checklist (MVP)

- Supabase project ready (DB + Auth + Storage)
- GitHub OAuth app configured
- Vercel (or Netlify) project for frontend
- Backend deployed (Edge functions / Render / Fly.io) with env vars: GitHub App keys, OpenAI API key, Supabase keys, Stripe keys
- CI: GitHub Actions for tests and deployments

---

17) Monetization & pricing ideas

- **Free tier**: 1 project, 5 snapshots/mo, basic prompts
- **Pro tier** (€10–25/mo): unlimited small repos, priority queue, advanced prompt templates
- **Team tier**: per-seat billing, audit logs, SSO, PR annotations (enterprise features)
- Offer add-on credits for extra LLM calls (pay-as-you-go) or on-demand deep analysis

---

18) Security & privacy notes

- Treat repo content as sensitive. Offer explicit consent screens.
- Store only metadata by default — only fetch file contents when user requests deep analysis. (This reduces liability and cost.)
- Provide data deletion on request, and clear retention policies.
- Use service accounts and least privilege for GitHub API tokens.
- Add rate limits, abuse detection, and cost controls for LLM usage.

---

19) Growth & go-to-market (short)

- Launch to developer communities: Reddit r/programming, r/webdev, Hacker News, and dev Discords
- Share before/after examples: show a messy repo and the generated `architecture.md` as a baseline.
- Offer early free credits to open-source repos and gather testimonials.
- Partner with coding bootcamps and university dev labs (your UTAD network).

---

20) Next recommended steps (concrete)

1. Initialize repo (monorepo with `apps/web` + `apps/api` + `packages/common`).
2. Set up Supabase project and GitHub OAuth app (you can do this quickly via docs).
3. Implement GitHub OAuth + store project record — minimal flow.
4. Implement RepoParserService for small repo and create a mocked LLM response pipeline.
5. Wire LLMService to a real LLM and produce `architecture.md` using the template.
6. Polish UI to show generated doc and allow download.
7. Add tests and CI.

---

21) Example prompts (short)

Architecture prompt (trimmed)

```
You will be given a repository file tree and selected file snippets. Output a
markdown document named architecture.md with these headings: Project Summary,
Tech Stack, File & Folder Structure, Responsibilities (per folder/file), DB
Models (tables + relations), Service Layers, Where state lives, Testing & CI,
Deployment notes, Uncertainties.
Use file paths as evidence. If uncertain, add an UNCERTAINTIES section
listing exact lines/files you couldn't analyze.
```

Task prompt (trimmed)

```
Using the provided architecture.md, write a granular step-by-step plan to
implement: [feature description]. The plan must include start+end, test
cases, file paths to change, acceptance criteria, and a list of unit tests to
add.
```

---

22) Final notes & philosophy

- Keep changes small and testable. The smallest useful feature is better than a big half-finished one.
- Design outputs to be *agent-friendly* but *human-verifiable* — always show the evidence (file paths, snippets) so users can check the LLM claims.
- Start with a conservative set of languages/frameworks (Node/TS, Python, Java) — expanding later.

If you want, I can now: - Generate a starter `architecture.md` template and `task.md` for the first onboarding feature (based on the example above), ready to drop into your repo; **or** - Produce a step-by-step list of first Git commits (1–8) you should make to build the MVP.

Tell me which of those two you want next, or I can pick one and proceed.