User Stories

Part A: Initial User & Function Mapping

1) Manual User Brainstorming : A preliminary, comprehensive list of all potential user types.

Direct Users (day-to-day product users)

- **Anchor Developer (solo / indie)** — writes Anchor programs, runs the CLI to instantly scaffold a frontend for testing and demos.
- **Full-stack Web3 Developer** — integrates generated UI into a larger app, customizes components, and commits code to repo.
- **Hackathon Team Member** — uses the generator to spin up quick demos and validate ideas within contest timeframes.
- **Frontend Engineer (web)** — uses generated TSX as starting point, replaces UI with production design, or implements custom components.
- **Product Manager (dApp)** — validates flows with stakeholders using the live local UI and uses LLM suggestions for UX wording.
- **QA Engineer / Integration Tester** — exercises program flows through the generated UI to create test cases and reproduce bugs.
- **Educator / Instructor** — uses the tool to teach Anchor concepts with a working UI for student exercises.
- **Student / Bootcamp Attendee** — runs the CLI to learn how on-chain instructions map to client forms and transactions.

Indirect Users / Beneficiaries

- **End Users of the dApp (non-technical)** — benefit when developers ship apps faster with better UX produced via LLM suggestions.
- **Project Owners / Founders** — get faster prototypes and demo-ready UIs to show investors or partners.
- **Business Analysts / PMs (stakeholder reviewers)** — review flows and approve product decisions using the generated UI.
- **Client Stakeholders (agency clients)** — see functionality and sign off on features before full front-end implementation.

Administrators, Maintainers & Moderators

- **Tool Maintainer / OSS Maintainer (I / core dev)** — maintains the CLI, templates, and LLM adapter and triages issues.
- **Community Moderator / Forum Admin** — moderates community requests, helps map common feature requests into template updates.

- **Repository Maintainer / Release Manager** — manages releases, template versions, and changelogs for the CLI.
- **Template Curator** — maintains and publishes vetted UI templates (Tailwind, Chakra, etc.) for the generator.

Stakeholders (vested but may not use directly)

- **Startup Founders / CTOs** — evaluate tool adoption across engineering teams to speed prototyping.
- **Investors / Grant Providers (Turbine, Solana Foundation)** — monitor adoption metrics and potential impact on developer productivity.
- **Partner Companies (wallets, infra)** — may want integrations or co-marketing (e.g., Phantom, RPC providers).
- **Training Organizations / Bootcamps** — invest in the tool for curriculum efficiency.

Security, Audit & Compliance Roles

- **Smart Contract Auditor** — uses generated UI to exercise program paths and reproduce edge cases during audits.
- **Security Researcher / Bug Bounty Hunter** — leverages local UIs to fuzz or exercise flows and find vulnerabilities.
- **Compliance Officer / Legal Reviewer** — reviews generated UI flows for regulatory red flags in fintech use cases.

AI / LLM-Specific Roles

- **Prompt Engineer / UX-LLM Designer** — crafts prompts or prompt templates to guide the LLM adapter's UI suggestions.
- **LLM Operator / DevOps for AI** — manages API keys, model selection (self-hosted vs API), and cost/latency configurations.
- **AI Reviewer / Trust & Safety Engineer** — reviews LLM recommendations, filters hallucinations, and defines guardrails.
- **Data Privacy Officer** — ensures IDLs and program logic are handled in ways that meet privacy/security requirements when passed to external LLMs.

Integration & Automation Roles

- **CI/CD Engineer** — integrates the generator into pipelines (e.g., auto-generate UI after IDL changes, run tests).
- **DevOps / Infra Engineer** — manages optional hosted preview environments or deploys frontends to Vercel/Netlify.
- **Plugin/Extension Author** — builds VSCode extension or GitHub Action that calls the CLI and commits code.
- **Template / Component Library Author** — provides UI components that the codegen can reference (e.g., design system wrappers).

Documentation, Support & Community

- **Technical Writer / Docs Maintainer** — writes usage guides, example workflows, and LLM adapter docs.
- **Developer Advocate / Community Manager** — runs demos, writes tutorials, and collects feedback from early adopters.
- **Support Engineer / Helpdesk** — triages installation, runtime, and integration issues reported by developers.

Buyers & Business Roles

- **Enterprise Procurement / IT Buyer** — evaluates licensing, security posture, and integration costs for enterprise adoption.
- **Agency Owners** — use the tool as part of client delivery pipelines to speed up MVPs and demos.

External Infrastructure & Partners (not direct users but essential)

- **Wallet Providers (Phantom, Backpack)** — their wallet adapters are integrated into the generated UIs.
- **RPC Providers (QuickNode, Alchemy)** — devnet/mainnet endpoints the UI connects to for transactions.
- **LLM Providers (OpenAI, Anthropic, local models)** — supply models for the LLM adapter when enabled.
- **Hosters (Vercel, Netlify)** — optional public hosting for generated frontends or previews.

Community & Advocacy / Secondary Beneficiaries

- **Hackathon Judges / Demo Day Reviewers** — see polished demos more often because teams can scaffold UIs rapidly.
- **Open-source Contributors** — extend templates, add LLM prompt packs, and improve IDL parsing.
- **Technical Bloggers / Educators** — produce tutorials and case studies that raise adoption.

Edge / Niche Users (special cases)

- **Accessibility Engineer** — adapts generated UI for accessibility compliance (a11y).
- **Localization Manager** — uses LLM to generate translations/localized labels for generated forms.
- **Researcher (UX/Dev Productivity)** — measures time savings and UX improvements from LLM suggestions.
- **Regulatory Sandbox Operator** — uses generated UIs to test compliance scenarios for new fintech products.

2) AI-Assisted User Prioritization

Final prioritized user list (POC focus — top 4)

1. **Anchor Developer (Primary)**
2. **Frontend Engineer (Integrator)**
3. **Hackathon Team Member (Rapid Prototyper / Demo)**
4. **Auditor / Security Researcher**

Why these 4 (decision summary)

I chose these four because together they prove the three core claims our MVP must demonstrate:

- **Technical correctness** (Anchor Developer) — Can the CLI parse a real Anchor IDL and produce a working Next.js app that actually calls a program on **devnet**? If this fails, the project has no foundation.
- **Practical adoptability / code quality** (Frontend Engineer) — Is the generated code *usable* in real projects? If frontend engineers cannot integrate or extend the output quickly, the generator is only a toy.
- **Speed & viral adoption potential** (Hackathon Team Member) — Can teams ship demos rapidly? Success here proves time-to-value and fuels word-of-mouth adoption.
- **Trust & utility for QA/security** (Auditor / Security Researcher) — Can the tool be used for serious testing and auditing workflows (exercise instructions, fetch accounts, reproduce issues)? This proves the tool's value beyond prototyping and increases confidence for professional use.

These four together validate that the product is: **correct**, **usable**, **fast**, and **trustworthy** — the four axes most important for developer tooling.

Agreement / divergence vs. prior AI suggestions

- **Agreement:** I mostly agree with the AI's earlier recommendation (it suggested Anchor Developer, Frontend Engineer, Hackathon Team, Auditor). Those four are the strongest signals to prove the core value and to get rapid adoption.
- **Divergence / deprioritizations:** I *deliberately deprioritized* personas like **Educator**, **Product Manager**, and **End Users** for the initial POC. Not because they're unimportant, but because:
  - Educators and classroom use are high-leverage later, once the generator consistently produces correct, editable code.
  - Product Managers and end users are downstream beneficiaries — they matter for product-market scaling, but they don't validate the technical claim that the generator produces correct, integratable frontends.
- **LLM/AI role:** I agree the LLM/AI features are valuable, but they are **V2**. For POC we must prove the deterministic generator first; LLMs should be introduced later as an **optional assistant** (suggestions that must be reviewed), not as the core mechanism in POC.

Short validation checklist per persona (what POC must prove)

- **Anchor Developer:** `npx anchor-ui generate` → `./anchor-ui` appears → run app → connect wallet → submit an instruction to devnet and receive a confirmed tx.
- **Frontend Engineer:** Import generated TSX into a host repo, swap primitive components for a design system, and build without breaking transaction behavior (≤1 hour of integration effort).
- **Hackathon Team Member:** Using example IDL, produce a demo (1 instruction + account read) and record demo assets within 2 hours.
- **Auditor:** Exercise all instructions via the UI and fetch at least one program account state; reproduce one non-trivial path/bug if test data exists.

---

3) Core Function Mapping

Top 1–2 critical user stories / interaction paths for the POC:

1. **Anchor Developer — Generate & run UI, call instruction (Primary POC path)**

   Given an Anchor project with `target/idl/my_program.json` and a program deployed on devnet,

   When the developer runs `npx anchor-ui generate` and starts the generated app,

   Then they can connect a wallet, select an instruction, provide required scalar args (and manual account pubkeys if needed), submit the transaction, and receive a confirmed devnet tx signature and visible account state update.

2. **Frontend Engineer — Integrate generated code into an existing repo with minimal effort (Secondary POC path)**

   Given a generated `./anchor-ui` scaffold,

   When a frontend engineer imports the generated components into a host Next.js app and swaps primitive inputs for design-system components,

   Then the integrated app builds and the instruction calls functionally succeed with ≤1 hour of integration edits.

---

4) Deriving Core POC Requirements

Core (must-have for POC — HIGH priority)

1. **CLI entrypoint**
   - Node-based CLI (commander.js / simple node script) with flags: `-idl`, `-out`, `-network`, `-open`.
   - Acceptance: running `npx anchor-ui generate` writes `./anchor-ui` and exits 0; `-idl` overrides autodetection.
2. **IDL detection & parser**
   - Read `target/idl/*.json`, parse Anchor IDL into a normalized `ProgramSpec` structure: instructions (name, args types), accounts (names, isSigner, isMut), types (simple structs/enums flagged).
   - Acceptance: spec JSON contains all instruction metadata; parser logs unsupported types.
3. **Minimal codegen / template engine**
   - Small templating system to emit:

    - `package.json`, `tsconfig.json`, `next.config.js` (minimal).
    - `pages/index.tsx` listing instructions and rendering forms.
    - `components/InstructionForm.tsx` handling input rendering, serialization, submit handler.
    - `lib/anchor.ts` (AnchorProvider + Program initialization using IDL + programId).
    - Copy IDL into `idl/` of generated app.
  - Acceptance: generated app compiles and starts (`npm run dev`).
4. **Form generation for scalar types**
  - Support automatic form fields for common scalar arg types: `u8/u16/u32/u64`, `i64`, `bool`, `string`, `publicKey` (pubkey input), `bytes` (base64/text).
  - Acceptance: each scalar maps to an input with proper parsing/serialization before calling `program.methods.<inst>(...).rpc()`.
5. **Anchor client wiring & wallet integration**
  - Use `@coral-xyz/anchor` and `@solana/wallet-adapter-react` to initialize provider and program (devnet default).
  - Include wallet-connect UI and prompt to add SOL-faucet guidance if wallet empty.
  - Acceptance: after connecting wallet, `program.methods.<inst>().rpc()` can be invoked.
6. **Transaction submission, logging, and basic account fetch**
  - Capture tx signature and show confirmation; display `program.account.<type>.fetch(pubkey)` for at least one account type.
  - Acceptance: after submission, tx sig displayed and account viewer shows updated state (if account exists).
7. **Manual pubkey fallback for accounts**
  - When an account required by instruction cannot be auto-derived, show an input for the developer to paste a pubkey.
  - Acceptance: transaction succeeds when developer provides correct pubkeys.
8. **Devnet support & simple deploy script for sample program**
  - Provide a small example Anchor program (counter/escrow) and instructions/docs to deploy to devnet (`anchor deploy --provider.cluster devnet`) or include an already-deployed sample programId for immediate demo.
  - Acceptance: README demonstrates steps to run end-to-end against devnet.

Quality & ergonomics (MEDIUM priority)

1. **Readable, small, human-editable code**
  - Templates should be modular, small components, with clear TODO markers for customization.
  - Acceptance: a Frontend Engineer can understand structure in <15 minutes.
2. **Basic validation & error handling**
  - Input validation for pubkey format, numeric parse errors, and clear UI error states from RPC.

- Acceptance: malformed pubkey shows inline error; RPC errors surface in log panel.

3. **Generator safety: non-destructive regenerate**
   - CLI can output to a new folder or warn before overwriting; `-merge` or `-force` optional.
   - Acceptance: default behavior avoids overwriting existing user edits.

Extensibility (LOW priority — for V2 but design now)

1. **Modular templates & `ProgramSpec` output**
   - Emit canonical `ProgramSpec.json` used by templates and future LLM adapter.
   - Acceptance: `ProgramSpec.json` contains normalized metadata and is used by templates.
2. **Hook points for LLM adapter**
   - Provide a clear stage: `parse → (optional) llmAdapter → render` allowing later insertion of AI-driven transformations (labeling, grouping, control recommendations).
   - Acceptance: `noop` llmAdapter exists and is importable.
3. **Sample tests / smoke tests**
   - Basic script to call one instruction using Anchor JS for CI smoke checks (devnet flakiness caveat).
   - Acceptance: smoke script runs locally and demonstrates program call.

Part B: Adversarial Analysis & Granularity Check

1) Critique & Refine User Stories / Requirements:

Quick critique — do the stories hit the mark?

What they do well

- Focus on the right personas (Anchor Developer + Frontend Engineer + Hackathon + Auditor).
- Capture the two core POC paths: (A) IDL → generated app → call program, and (B) integrate generated code into a host repo.
- Surface essential requirements: CLI, IDL parser, templates, wallet, tx logging, manual pubkey fallback.

What's missing / not granular enough

- **Spec/schema details**: The `ProgramSpec` shape used by templates was described conceptually but not in schema form (fields, types).
- **Exact file/api contract**: No explicit function names, file paths, or minimal code contract for `lib/anchor.ts`, `InstructionForm.tsx`, or the CLI.
- **Serialization rules**: How numeric types and Borsh-related types are mapped and validated in the UI is underspecified.
- **Account handling**: PDA derivation behavior, default accounts (signer/system), and how to present `isSigner/isMut` to the user need rules.

- **Regeneration behavior**: How to avoid overwriting custom edits when re-generating (merge strategy) needs explicit behavior.
- **Testing**: No concrete unit/integration test cases or smoke scripts were specified.
- **Operational config**: Which RPC endpoints, environment variables, and package versions are assumed? (affects reproducibility)

Conclusion: the stories are directionally correct but need refinement to map to exact components and tests.

Part C: Granularity & Clarity Refinement

1) Final Manual Review & Refinement :

Notes about language:

- I replaced technical shorthand with plain phrases.
  - Example: **IDL** → **"program description file produced by Anchor"**.
  - Example: **RPC / devnet** → **"the Solana test network (devnet) via a network endpoint"** only where needed; otherwise described as "test network".
- Each story is atomic: one action → one outcome.

Final user stories (atomic, plain-language)

Anchor Developer (primary)

1. **Run generator to create a local UI scaffold**
   - **Action:** Developer runs a single command in their Anchor project folder.
   - **Outcome:** A new folder (`./anchor-ui`) appears with a runnable web app scaffold.
   - **Acceptance:** The folder contains `package.json`, an app page, a program description file copy, and a short README.
2. **Start the generated app locally**
   - **Action:** Developer installs dependencies and starts the app on their machine.
   - **Outcome:** A browser page opens showing the generated interface for the program.
   - **Acceptance:** The page loads without compile errors.
3. **Connect wallet to the generated UI**
   - **Action:** Developer connects a Solana wallet (e.g., Phantom) in the browser UI.
   - **Outcome:** The app recognizes the connected wallet and displays the wallet address.
   - **Acceptance:** Wallet address visible and the app shows a “Connected” state.
4. **Call a program instruction from the UI**
   - **Action:** Developer selects an instruction, fills its simple input fields, and clicks Submit.
   - **Outcome:** The instruction runs on the Solana test network and a transaction ID is shown.
   - **Acceptance:** Transaction ID shown and it links to the test-network explorer.

5. **Provide missing account keys when needed**
    - **Action:** If the app cannot determine a required account automatically, the developer pastes a public key into a field.
    - **Outcome:** The app uses that key so the instruction can run.
    - **Acceptance:** Input validates as a proper key format and Submit becomes available.
6. **Regenerate safely after program changes**
    - **Action:** Developer re-runs the generator after changing the program description.
    - **Outcome:** New scaffold is created without accidentally overwriting local edits (default: generate to a new folder or warn).
    - **Acceptance:** Generator warns before overwrite and provides an alternate output folder option.

Frontend Engineer (integrator)

1. **Inspect generated code structure**
    - **Action:** Engineer opens the generated files to review their organization.
    - **Outcome:** Engineer quickly finds the main app page, the program-connection file, and form components.
    - **Acceptance:** Engineer locates these files within 10 minutes.
2. **Import one generated component into a host app**
    - **Action:** Engineer copies one form component and the connection helper into an existing Next.js app.
    - **Outcome:** The host app builds and the form still calls the program on the test network.
    - **Acceptance:** Host app builds and a test form submission returns a transaction ID.
3. **Swap a basic input for a design-system input**
    - **Action:** Engineer replaces a generated primitive input with a design-system input (styling only).
    - **Outcome:** UI appearance changes but function and form submission remain correct.
    - **Acceptance:** After the swap, submissions still produce the same transaction behavior.
4. **Add simple client-side validation**
    - **Action:** Engineer adds validation for a public key field and a numeric field.
    - **Outcome:** The form blocks bad input before sending it to the network.
    - **Acceptance:** Invalid entries show inline errors and prevent submission.

Hackathon Team Member (rapid prototyper)

1. **Generate a demo app from example code quickly**
    - **Action:** Team runs the generator using a provided example program and starts the app.
    - **Outcome:** Team has a working demo within two hours.

- **Acceptance:** Demo shows at least one interactive instruction and an account read.

2. **Share a live demo for non-developers**
   - **Action:** Team deploys the generated app to a free host or shares the local screen.
   - **Outcome:** Non-developer judges or teammates can click and see the flow happen.
   - **Acceptance:** At least one non-developer can complete the demo steps successfully.
3. **Make a quick tweak for presentation**
   - **Action:** Team edits a field label or default value in the generated code.
   - **Outcome:** The change appears in the UI immediately for the demo.
   - **Acceptance:** The updated label appears after restarting the app or hot reload.

Auditor / Security Researcher

1. **Exercise each program instruction via UI**
   - **Action:** Auditor uses the generated UI to run every instruction once with valid inputs.
   - **Outcome:** Auditor collects transaction IDs and sees account states after each run.
   - **Acceptance:** Auditor obtains a transaction ID for each instruction attempted.
2. **Reproduce a reported issue using the UI**
   - **Action:** Auditor inputs a known problematic sequence into the UI and runs it.
   - **Outcome:** Auditor reproduces the issue and captures evidence (tx IDs, account snapshots).
   - **Acceptance:** Reproduction includes at least one tx ID and one account snapshot.
3. **Export a test input set for automation**
   - **Action:** Auditor saves a set of form inputs as JSON from the UI.
   - **Outcome:** That JSON can be used later in a small script to replay the same calls.
   - **Acceptance:** Replay script using the saved JSON produces the same or comparable transactions.

Refinement Log (appendix)

For each change I made, I show the **Before** story (short paraphrase from Part B) and the **After** refined story, plus a short rationale.

---

1. Generate UI from IDL

- **Before:** "Run CLI to locate IDL and write ./anchor-ui; CLI exits 0."
- **After:** "Run generator to create a local UI scaffold" (see story).
- **Rationale:** Simplified language (no CLI jargon), made the action and outcome explicit for non-technical stakeholders.

2. Run the generated app locally and call a program

- **Before:** "Start the generated app, connect Phantom, select instruction, fill args, submit, tx signed and confirmed."
- **After:** Split into three stories:
  - "Start the generated app locally"
  - "Connect wallet to the generated UI"
  - "Call a program instruction from the UI"
- **Rationale:** Split into atomic steps so each is a single action with a clear acceptance test; removed jargon like "sign tx" replaced by "transaction ID is shown".

3. Manual account fallback

- **Before:** "Provide manual pubkey inputs when automated PDA/account derivation fails."
- **After:** "Provide missing account keys when needed" (story text).
- **Rationale:** Rephrased to remove PDA jargon and focus on simple action: paste key; clarified validation expected.

4. Regenerate safely after IDL changes

- **Before:** "Iterate on IDL and regenerate safely — preserve edits or flag changes."
- **After:** "Regenerate safely after program changes" (story).
- **Rationale:** Made behavior explicit (warn before overwrite, alternate folder) and broke into single action.

5. Frontend integration stories

- **Before:** Several combined items like "Import generated components, replace UI, run build, validate transactions."
- **After:** Split into separate stories: inspect structure; import one component; swap input for design input; add validation.
- **Rationale:** Each action is a distinct engineering task; easier to assign and test.

6. Hackathon demo stories

- **Before:** "Rapid prototype & demo; non-dev teammate testing; generate screenshots."
- **After:** Three atomic stories: generate demo app quickly; share live demo; make quick tweak for presentation.
- **Rationale:** Clear, time-bounded expectations (e.g., two hours) and non-technical phrasing for presentation tasks.

7. Auditor stories

- **Before:** "Exercise all instruction paths; reproduce bug; generate test harness."
- **After:** Three stories: exercise each instruction; reproduce reported issue; export test input set.
- **Rationale:** Split into concrete, testable actions and remove developer-only language like "harness" replaced by "save inputs as JSON".

8. Language and acceptance criteria

- **Before:** Many stories included technical terms like "IDL", "RPC", "BN", "program.methods().rpc()".
- **After:** Rewrote using plain language and added simple acceptance criteria (e.g., "transaction ID shown").
- **Rationale:** Stakeholder-friendliness and clearer pass/fail tests for the POC.

Part D: Defining Potential On-Chain Requirements

the tool itself doesn't *require* writing new on-chain programs.

However, many user actions *depend on* existing on-chain state and program behavior (a deployed program, accounts, payer funds, correct account permissions, etc.). Below I map each refined user story to the **on-chain prerequisites or requirements** that must be present or satisfied for that story to succeed. Treat this as a checklist of what must exist on the blockchain (or be true of the deployed program) for the generated UI to work end-to-end.

Anchor Developer (primary)

1. Run generator to create a local UI scaffold

On-chain requirements (brainstorm):

- *None required.* This story only produces local files.
- *Optional:* If generator validates a programId from the IDL, that programId should point to a deployed program (or generator should allow a placeholder).

---

2. Start the generated app locally

On-chain requirements:

- *None strictly required* to start the app.
- *Optional / recommended:* default network should be a known test node (devnet RPC) so subsequent actions can use it.

---

3. Connect wallet to the generated UI

On-chain requirements:

- No smart contract changes required.
- Wallet must hold a valid Solana account on the chosen network.
- If the wallet needs funds to send transactions, a developer must have devnet/test SOL in that wallet.

---

4. Call a program instruction from the UI

On-chain requirements:

- **Program deployed**: The Anchor program (referenced by the IDL/programId) must be deployed on the network (devnet/mainnet as targeted).

- **Instruction available**: The instruction name/argument types in the IDL must match the deployed program's interface.
- **Required accounts exist**: Any accounts the instruction expects (state accounts, token accounts, PDAs, system accounts) must either exist or be creatable by the instruction.
- **Permissions/ownership**: Caller wallet must have required signer or authority permissions if instruction requires signer or owner.
- **Rent/exemption & balance**: If instruction creates accounts, there must be enough lamports to make them rent-exempt (payer must fund).
- **Correct serialization**: Client must serialize args exactly as on-chain expects (Borsh layout).
- **Network availability**: RPC endpoints must accept and propagate transactions; program must be on the same cluster (devnet).

---

5. Provide missing account keys when needed

On-chain requirements:

- Provided pubkey must correspond to an existing account (if instruction expects it exists) or be acceptable as new account input.
- If the instruction expects an owner or specific account state, the provided key must match those expectations (ownership, program-derived addresses, etc.).
- If account must be rent-exempt, payer must fund creation if instruction creates it.

---

6. Regenerate safely after program changes

On-chain requirements:

- *None for generation itself.*
- If verifying a programId or testing changes immediately, the developer must deploy the updated program to the same network and ensure the IDL matches the deployed program.

Frontend Engineer (integrator)

1. Inspect generated code structure

On-chain requirements:

- *None.* This is local inspection.

2. Import one generated component into a host app

On-chain requirements:

- Host environment must be able to talk to the same network where the program is deployed.
- ProgramId used by the component must reference a deployed program accessible from that host's RPC.

3. Swap a basic input for a design-system input

On-chain requirements:

- *None.* But unchanged serialization must be preserved so on-chain calls remain valid.

4. Add simple client-side validation

On-chain requirements:

- Validation must match on-chain constraints (e.g., max/min values enforced by program). If front-end validation rejects valid on-chain values or accepts invalid ones, transactions will fail.

Hackathon Team Member (rapid prototyper)

1. Generate a demo app from example code quickly

On-chain requirements:

- Example programId must point to an existing deployed demo program (or include scripts to deploy to devnet).
- Wallets used in the demo must have devnet test SOL (faucet).
- Demo accounts (state accounts, sample token accounts) must be present or creatable by the demo flows.

2. Share a live demo for non-developers

On-chain requirements:

- The demo's program must be stable on the chosen network (devnet), not ephemeral.
- If demo uses public testnet accounts, those accounts must be funded and have correct data; otherwise the demo should create accounts (and payer must pay).
- RPC availability and rate limits must be considered so judges can access the program.

3. Make a quick tweak for presentation

On-chain requirements:

- If tweak changes default inputs or creates new transactions, ensure those flows still hit valid on-chain state (e.g., default pubkeys exist) and payer has enough SOL to cover operations.

Auditor / Security Researcher

1. Exercise each program instruction via UI

On-chain requirements:

- All instructions must be deployed and callable.
- Auditor's wallet must have signer privileges where required (or the program must permit public calls).
- Account fetch endpoints must expose account state (account data must be readable by RPC).
- For state-altering instructions, ensure test environment is safe (devnet) and test data exists to avoid accidental mainnet test.

2. Reproduce a reported issue using the UI

On-chain requirements:

- The program must be deployed with the same state/context as when the issue occurred (or a replicated test state).
- Auditor must have access to account keys or test accounts that reproduce the condition (some issues require particular account states or balances).
- Transactions must be reproducible (same instruction layout and serialization).

3. Export a test input set for automation

On-chain requirements:

- The saved inputs must reference on-chain accounts that are stable or reproducible in the test environment (e.g., fixtures or seeded accounts).
- If replay will modify state, tests should run on devnet or a local validator to avoid polluting shared resources.

Summary & practical guidance

- **Do you need to write on-chain code for the tool itself?** No — the generator tool is a client-side developer utility. You do **not** need to deploy new smart contracts as part of the generator POC.
- **What *is* needed on-chain?** For the generated UI to actually perform its key flows, **the target Anchor program must be deployed** to the target network (devnet for POC), and the deployed program must match the IDL the generator uses. Additionally, the user wallets must have test SOL and any required accounts (state accounts, token accounts) must exist or be creatable by the instruction.
- **Edge cases to plan for in the UI/CLI**:
  - If IDL references PDAs but provides no seeds, generator should ask the developer for pubkeys or provide a JSON input box.
  - For instructions that create accounts, the UI should display an estimated fee or flag when the connected wallet has insufficient funds.
  - For auditing/replaying, provide a recommended workflow: use devnet or a local validator; seed accounts with fixtures; export and store test input JSON.

Part B: Process Appendix

Section 0 — Input & high-level constraints

Key constraints and decisions I provided:

- Stack: **Next.js + TypeScript + Node.js**.
- Must use **Anchor** and Anchor-generated IDLs only.
- MVP targeted for **devnet**, mainnet optional.
- Two-week (then later I said 2-week) MVP goal; minimal modular design to plug in LLM later.
- Wanted CLI-first flow that generates raw code into a folder and runs locally at `localhost:3000`.

- LLM/AI to be integrated **later**; POC must be deterministic and local-first.

Section 1 — The prompts / user interactions used in this assignment (chronological sampling)

1. “I’m a full-stack Web3 developer… propose and build a capstone project on Solana, using Anchor… three categories: Developers’ UX, Users’ UX, Anchor tools… focus on Anchor.”
2. I confirmed stack (Next.js, TypeScript, Node.js), open to both dev- and user-facing, devnet required.
3. I selected **Anchor IDL-to-UI Generator** and asked to dive deeper (tech stack, deployables, pain points, workflow, existing tools, final project proposal).
4. I specified: CLI that generates raw code from IDL, serves at `localhost:3000`, supports both raw code generation and live interaction, strictly Anchor IDLs.
5. I set a 2-week MVP horizon and asked for minimal modular base and LLM later.
6. I asked to rewrite concise deliverables (naming, overview, value prop, PMF, target markets, competitor landscape, FMF, critique/refinements).
7. I requested persona lists and prioritized personas for the POC.
8. I asked to map core functions for prioritized users and derive top POC stories and technical requirements.
9. I asked for adversarial analysis and granularity checks, plus Part C refinements (de-jargon, atomic stories, refinement log).
10. I asked whether on-chain requirements are applicable and for a mapping of on-chain prerequisites for each story.
11. Finally I asked for this **Process Appendix** documenting the whole process.

Section 2 — Key AI outputs (summary by part)

Part A — Project definition, MVP & modular plan

- **Project Name idea list** (single-word options): Forge, Loom, Stitch, Bolt, Scaffold, Frame, Autogen, Weave, Mosaic, Cast.
- **Concise project overview** (2–5 sentences) outlining the CLI → IDL → Next.js scaffold flow, devnet targeting, and optional LLM.
- **Tech stack**: Next.js, TypeScript, Node.js CLI, `@coral-xyz/anchor`, `@solana/web3.js`, `@solana/wallet-adapter-react`, Tailwind (optional) — satisfies Anchor usage.
- **Barebones MVP spec** (2-week): CLI `anchor-ui generate` → parse IDL → generate minimal Next.js app with wallet, forms for scalar args, manual pubkey fallback, submit wiring, logs, account fetch display.
- **Minimal modular architecture**: layers: CLI, IDL parser, Template/Codegen module, static bootstrap, dev server runner, extensibility hooks (`templates/v1/`, `llm_adapter/`).
- **CLI UX & folder layout** example and explanation of what is in generated app.

- **Two-week roadmap** broken down day-by-day.

Part B — Personas and Core Function Mapping

- **Comprehensive persona list** (direct users, indirect beneficiaries, admins, stakeholders, security roles, LLM roles, CI/CD, docs, buyers, partners).
- **Selected top 4 prioritized personas for POC**:
  1. Anchor Developer (primary)
  2. Frontend Engineer (Integrator)
  3. Hackathon Team Member (Rapid prototyper)
  4. Auditor / Security Researcher
- **Rationale** for each persona and validation metrics.
- **Function mapping** listing the key interactions each persona must perform (e.g., run CLI, connect wallet, submit instruction, manual pubkey fallback).
- **Top two critical interaction paths** (Anchor Developer generate & call; Frontend Engineer integrate).
- **Derived key technical requirements** from those interactions: CLI, IDL parser (ProgramSpec), generator templates, wallet integration, scalar field serialization (BN handling), manual pubkey fallback, tx logs, account fetch, sample Anchor program for demo, non-destructive regen behavior, smoke scripts.

Part C — Granularity & Clarity Refinement

- **Refined user stories** converted into atomic, plain-language user stories per persona (Anchor Developer stories split into run generator, start app, connect wallet, call instruction, provide missing keys, regenerate safely; Frontend Engineer stories split into inspect, import component, swap input, add validation; Hackathon and Auditor stories likewise).
- **Part C Refinement Log** describing each change (Before → After + rationale), focusing on de-jargonization, atomicity, clarity, no overlap.

Part D — On-chain requirements analysis

- Clarified that the tool **does not itself require on-chain programs**, but generated UI depends on a deployed Anchor program on the chosen network.
- For each refined user story, produced a bulleted list of **on-chain prerequisites** (e.g., program deployed, required accounts exist or can be created, wallet has funds, correct serialization).
- Concluded that no new on-chain code is necessary for the tool, but documentation and optional demo deployment scripts are strongly recommended to allow non-technical users to run demos.

Section 3 — Detailed analyses & rationale for refinements

A. Why CLI-first, local-first, commit-ready code?

- **Problem:** Hosted preview products and AI-assisted builders are attractive but many teams need code they can commit, version, and customize. Hosted solutions create lock-in.

- **Choice:** Make the POC local-first CLI that creates a repo-ready folder, ensuring developers can own the code.
- **Rationale:** In developer tooling, adoption is driven by trust — teams must be able to inspect and modify generated code. That trust is essential for Anchor developers dealing with wallets and on-chain effects.

B. Why prioritize Anchor Developer & Frontend Engineer personas?

- **Problem to prove:** The tool's core technical claim is “IDL → runnable UI”. If it fails for an Anchor Developer (can't generate or run), the product is moot.
- **Choice:** Anchor Developer is the core validator of correctness; Frontend Engineer validates that generated code is usable and integratable.
- **Rationale:** Combining both ensures both correctness *and* adoptability — the two most critical signals for developer tooling.

C. Why defer LLM/AI to V2 and declare hook points?

- **Problem:** LLMs are powerful but can hallucinate and produce code that looks good but breaks serialization or safety constraints — risky for financial flows.
- **Choice:** Make LLM optional and plug-in based. Design generator as `parse → (optional llmAdapter) → render`.
- **Rationale:** This keeps POC deterministic and safe; LLMs only provide *suggestions* that developers review and accept.

D. Why require ProgramSpec schema & explicit serialization rules?

- **Problem:** Mismatch between client serialization and on-chain Borsh layout causes runtime failures — the most common source of "it compiles but fails".
- **Choice:** Define canonical `ProgramSpec` with schema, and implement `lib/serialization.ts` helper to parse numeric types to `BN`, validate pubkeys, etc.
- **Rationale:** Ensures generated forms encode values exactly as the Anchor program expects.

E. Why explicit manual pubkey fallback + conservative regeneration?

- **Problem:** PDAs and missing seed metadata cannot be reliably derived from IDL alone (IDL often lacks seed metadata).
- **Choice:** Show manual input for accounts generator can't resolve; generator doesn't overwrite existing files unless `-force`.
- **Rationale:** Minimizes surprises and prevents data loss for developers.

F. Why include smoke scripts & sample programs?

- **Problem:** Many reviewers or non-technical judges need reproducible demos without setting up Anchor or deploying code themselves.
- **Choice:** Include a sample Anchor program and a smoke script to demonstrate end-to-end flows on devnet quickly.
- **Rationale:** Smoothes on-ramp for hackathon teams and reviewers.

Section 4 — The refinement log (consolidated)

This repeats the Part C Refinement Log but in a compact list so we can trace each change at a glance:

- **Split composite stories into atomic stories**: e.g., “start app + connect wallet + submit tx” → three standalone stories. (Rationale: atomicity & testability.)
- **Removed technical jargon**: `IDL` expanded to “program description file produced by Anchor” where needed; `RPC/devnet` explained as “test network” for non-technical reviewers. (Rationale: stakeholder comprehension.)
- **Added explicit acceptance criteria** to each story: e.g., “transaction ID shown and links to explorer” or “file present in `./anchor-ui` folder”. (Rationale: pass/fail clarity.)
- **Specified ProgramSpec schema** and function contracts (e.g., `initProgram(rpc?)` signature) so templates and codegen align. (Rationale: engineering clarity.)
- **Defined generator safety behavior**: default must not overwrite existing folders; `-force` / `-merge` flags described. (Rationale: preserve developer code.)
- **Added minimal serialization rules** for common types (BN for u64, base58 validation for pubkeys). (Rationale: correctness with Anchor.)
- **Mapped on-chain prerequisites** for each story, acknowledging the generator itself doesn’t need on-chain code but needs a deployed program to call. (Rationale: operational clarity.)

Section 5 — Final technical artifact checklist

These are the discrete artifacts and implementation contracts that are now well-specified and ready for implementation or hand-off:

1. **CLI contract**
   - Command: `npx anchor-ui generate [--idl path] [--out dir] [--network devnet] [--open] [--force] [--merge]`.
   - Behavior: auto-detect IDL or use `-idl`, safe default output, optional `-open` to run `npm install` and `npm run dev` (prompt user).
   - Exit codes and messages defined superficially in requirements.
2. **ProgramSpec JSON schema**
   - Fields: program name, programId, instructions (name, args[], accounts[]), accounts (struct definitions), types[] — schema provided earlier and ready to formalize as JSON Schema.
3. **Renderer templates list**
   - `package.json`, `tsconfig.json`, `next.config.js`
   - `idl/<program>.json` copy
   - `lib/anchor.ts` with `initProgram(...)`
   - `components/InstructionForm.tsx` with a `submit` handler (calls Anchor client)
   - `pages/index.tsx` to list instruction forms
   - `components/TxLog.tsx` & `components/WalletProvider.tsx` (basic)
   - README with devnet instructions and preflight checklist

4. **Serialization helper**
   - `lib/serialization.ts` with `parseArg(value, typeSpec)` and `isValidPubkey(...)`.
5. **Account handling rules**
   - Default buyer/account heuristics for `payer` and `authority`; manual pubkey field for unresolved accounts.
6. **Regeneration safety**
   - Default: fail if output exists; `-force` to overwrite; `-merge` or output to new folder recommended.
7. **Smoke script & sample Anchor program**
   - `scripts/smokeCall.js` to call a simple instruction (optional sample keypair or faucet guidance for devnet).
8. **Part C refined user stories** — ready to convert into tickets.

Section 6 — Implementation & next steps AI recommended (and why)

Proposed two practical next actions (choose either):

- **Option A — Generate the actual minimal CLI skeleton + templates now**

  I can produce copy-and-paste TypeScript files for the CLI bootstrapping and the `basic-next` templates (`lib/anchor.ts`, `components/InstructionForm.tsx`, `pages/index.tsx`, `package.json`). This accelerates the two-week POC.

- **Option B — Produce the exact `ProgramSpec` JSON Schema and an example `ProgramSpec.json`**

  This gives engineers a precise parser target and reduces ambiguity for the codegen templates. Also helpful if we want to implement the parser first.

AI indicated earlier which of those you'd like; you can request either or both now and I will generate the files.

Section 7 — Closing notes about process integrity

- I did **not** run any external commands or deploy anything — all the above is a planning and design artifact.
- I avoided exposing any private chain-of-thought or internal reasoning beyond the summarized analyses and rationales. All recommendations are explicit and testable.
- The approach balances **speed to POC** (2-week deterministic generator) with **future extensibility** (LLM/AI hooks and template modularity). It prioritizes developer trust (commit-ready code, no surprise overwrites) and program-level correctness (serialization, manual account fallback).
