ChatGPT

30-Day Implementation Roadmap for Refactored ADRScan CLI

Week 1 – Sprint Kickoff & Core Framework Setup

- **Project Setup & Planning** – *Project Lead*:
- Initialize the GitHub repository and project board. Set up standard **issue templates** (e.g. Bug Report, Feature Request) via GitHub's template builder [1].
- Define issue **labels** for easy categorization: e.g. `bug`, `enhancement`, `wasm`, `cli`, `documentation`. Establish milestones for each sprint (Week 1–4) or major release (e.g. *MVP v1.0*).
- Configure continuous integration (CI) workflows. For example, add a GitHub Actions workflow to build and test on every push/PR. Include **matrix testing** on Linux, macOS, and Windows to ensure multi-platform support [2].

- Adopt a **code review and branching strategy** (feature branches -> PRs -> main). All developers agree on code style (enforce with rustfmt/clippy in CI) and semantic versioning for the project.

- `init.rs` **(CLI Initialization)** – *CLI Engineer*:

- Scaffold the command-line interface structure using a Rust CLI framework (e.g. Clap or StructOpt). Implement the top-level `adrscan` command with subcommands: `init`, `inventory`, `diff`, `new` (for ADR proposal), etc.
- Develop the `adrscan init` functionality to set up a new ADR directory in the project. This may create a standard folder (e.g. `docs/adr/` or as configured) and an optional first ADR markdown template (ADR 0001 record).
- Ensure the CLI has proper help messaging and version info. Parse global flags (like `--verbose` or `--output-format`) that will be needed for formatting outputs. Stub out subcommands so other modules can be integrated in later sprints.

- **Testing:** Write basic unit tests for CLI argument parsing and the `init` command (using temp dirs for file creation). Verify that running `adrscan init` twice does not overwrite existing data unintentionally.

- `inventory.rs` **(ADR Inventory & Hashing)** – *CLI Engineer*:

- Implement functionality to scan the ADR directory and build an **inventory list** of ADR files (e.g. reading all `.md` files). For each ADR, compute a content **hash** (e.g. SHA-256 of the file contents) to detect changes. Use a deterministic method for hashing and listing (e.g. sort files by name or date consistently).
- Store the inventory (ADR filename -> hash) in memory and (optionally) output it as a JSON or a lockfile (for later diffing). Ensure the hashing ignores irrelevant differences (like whitespace or line endings) if required by design.

- Address deterministic output early: avoid non-deterministic iteration over data structures. For example, use `BTreeMap` instead of `HashMap` or otherwise fix iteration order, since default `HashMap` iteration is randomized [3]. This guarantees the same inventory order on each run for identical inputs.

- **Testing:** Create unit tests feeding in sample ADR files and verifying the inventory list and hashes. Compute a known hash for a sample ADR content to verify correctness. Include tests for edge cases (no ADR directory, empty file, etc.).

- **Team Coordination:**

- The *CLI engineer* focuses on command parsing and core features (`init` and inventory). In parallel, a *Tester/QA* begins drafting a test plan for upcoming features (diffing, validation, etc.), identifying needed test cases.
- Set up a kickoff meeting to review functional requirements and design decisions for modules (e.g. how diffing will work, format of ADR proposals). Ensure everyone understands the ADR format and goals of the refactor.
- Document any architectural decisions (like choosing to separate library logic from CLI) in an ADR if applicable, eating our own dog food.

Week 2 – Core Features Development

- `inventory.rs` **(Diffing Capabilities)** – *CLI Engineer*:
- Extend the inventory module to support **diffing** between two states of the ADR inventory. Implement an `adrscan diff` command that compares the current ADR directory's hash list against a baseline (possibly a saved inventory from a previous run or the main branch).
- Highlight changes: identify newly added ADRs, removed ADRs, and modified ADR files (content hash changed). Provide a clear textual output of differences (e.g. "ADR 0005 – *Status changed*" or "ADR 0007 – *New*").
- Ensure diff output is **deterministic** in ordering. For example, always list changes sorted by ADR ID or filename to avoid random ordering differences across runs.

- **Testing:** Simulate a scenario with an initial inventory vs an updated inventory. Write tests that feed two inventory states and verify that `diff` outputs the correct added/removed/changed items. Include a test where nothing changed to ensure it reports "no changes" appropriately.

- **ADR Proposal Generation** – *CLI Engineer*:

- Implement the `adrscan new` (or `adrscan propose`) subcommand to streamline creating a new ADR markdown file. This should generate a new ADR file with the next sequence number in the ADR directory (e.g. if last ADR was 0005, create `0006-my-decision.md`).
- Auto-populate the new ADR file with a **template**: include headers for *Status: Proposed*, *Context*, *Decision*, *Consequences*, etc., and maybe a timestamp or date. If there's a convention of linking to superseded ADRs, include placeholders for that.
- Ensure the proposal generation logic validates that the new ADR number is unique and sequential. It might read existing ADR filenames to find the max number. Also, handle custom title input if provided (e.g. `--title "Use PostgreSQL"` to set the ADR title in the file).

- **Testing:** Use a temporary directory with some ADR files, run `adrscan new`, then verify that a new file is created with the correct name and template content. Test edge cases like when no ADRs exist (should start at 0001) or when ADR directory is missing (should error or create it).

- **Validation & Formatting Module** – *CLI Engineer*:

- Develop an ADR **validation** feature (`adrscan validate`) that checks all ADR files for adherence to expected structure and conventions. For each ADR in the inventory, verify required sections (e.g., each has a Status, Context, Decision, Consequences section) and that the Status is one of allowed values (Proposed, Accepted, Deprecated, etc.).
- Implement formatting rules or suggestions as needed. For example, ensure each ADR's filename matches the "NNNN-title.md" format or that links between ADRs (if any) are valid. If any ADR deviates, print warnings or errors with line numbers if possible.
- Integrate an output formatting system for CLI feedback. Support `--format=json` (machine-readable output) vs default human-readable text, if required by GitHub Action integration. This ties into deterministic output: the JSON or text output for validations and diffs should be consistent across runs given the same state.

- **Testing:** Create some sample ADR files with intentional errors (missing sections, incorrect statuses, broken links) and confirm that `adrscan validate` catches these issues and prints the expected messages. Also test well-formed ADRs to ensure no false positives.

- **GitHub Action Workflow Integration** – *DevOps Engineer*:

- Design a strategy for **GitHub Action integration** so that the tool can run in CI to enforce ADR standards. This could be a simple usage of the CLI in a workflow file. For example, a workflow that runs on pull requests to execute `adrscan validate && adrscan diff` and fails if any ADR is invalid or if there are unexpected changes.
- If needed, create a dedicated GitHub Action (composite or Docker-based) in a separate repository or within `.github/actions`. However, since `adrscan` will be a CLI tool published on crates.io and npm, using it via a workflow script (install with Cargo or `npm install` for WASM) might be sufficient.
- Add documentation for integration: e.g., how to use `adrscan` in a CI environment. Provide a YAML snippet in the README or docs showing a job that installs Rust (or uses the NPM package) and runs the tool. Make sure to mention any required environment setup (like if the repo has ADRs in a specific path).

- **Collaboration:** Coordinate with the *Tester* to simulate the GitHub Action locally (using `act` or a dry run) to ensure it behaves as expected. For now, stub this out; final testing will be in Week 3–4 once the tool is more complete.

- **Progress Review & Adjustments**:

- Mid-sprint review at end of Week 2: The team (CLI engineer, others) meets to demo the `inventory` and `diff` outputs, `new` ADR generation, and validation results. Verify all functional requirements for these features are on track.

- Collect feedback on CLI UX (are commands and messages clear?). Adjust plans for Week 3 if certain features need more attention or if any functional requirement is not yet met. Ensure that inventory hashing, diffing, proposal generation, validation, and formatting features are all at least basically implemented by now.

Week 3 – Hardening, WASM Integration, and Testing

- **Comprehensive Testing & Coverage** – *Tester/QA*:
- Expand the test suite to increase **code coverage** for all modules. Write integration tests that simulate real-world usage: e.g., create a fake repo with ADR files, run the sequence `init -> new -> validate -> inventory -> diff` and verify the outcomes at each step.
- Use code coverage tools (like `cargo tarpaulin` or `grcov`) to measure coverage. Aim for a high percentage (80-90%+) to ensure confidence. Focus on tricky edge cases (e.g., diff when an ADR's content changes only in whitespace, or validation of an ADR with extra sections).
- Ensure tests cover **multi-platform concerns**: e.g., on Windows, file paths might use `\`. Incorporate tests for path handling by leveraging Rust's temp dirs (which use correct OS-specific separators) and ensure no hard-coded "/" in code.

- **Bug fixing:** As tests surface issues, the team fixes them promptly. For example, if any nondeterministic behavior is observed (a test fails intermittently), track it down (could be a HashMap iteration order issue, etc.) and resolve by using deterministic data structures or seeding as needed [3].

- **WASM Compatibility (`wasm.rs`)** – *WASM Engineer*:

- Refactor the codebase to support compiling to WebAssembly. Extract core logic into a library module that can be shared between the CLI binary and a WASM interface. For instance, the inventory and diff functions should reside in lib code that can be called from both main (CLI) and from `wasm_bindgen` exports.
- Use **conditional compilation** to handle differences between native and WASM. For example, gate file system access behind `#[cfg(not(target_arch = "wasm32"))]` and provide alternative interfaces for WASM (where the caller might supply file contents or a list of ADRs). The StackOverflow example shows using `#[cfg(target_arch = "wasm32")]` to segregate code for the WASM target [4].
- Implement `wasm.rs` to expose key functionalities via `#[wasm_bindgen]`. For example, export a function to validate ADR text (so it can be used in a web app), a function to produce an inventory diff given two lists of ADR metadata, etc. These should return results in JS-friendly structures (strings, arrays, object via serde).

- **Build & Test WASM:** Set up `wasm-pack` to build the project as a WASM package. Run `wasm-pack test --node` (for headless testing in Node.js) to execute any tests that can run in the WASM environment. Verify that all exported functions work as expected in JS (possibly write a small JS snippet to call them). Address any panics or issues (e.g., if certain Rust crates aren't compatible with `no_std` or WASM).

- **Multi-Platform Verification** – *CLI Engineer & Tester*:

- By mid-Week 3, ensure the tool works on **all target platforms**. Utilize the CI matrix from Week 1 to run tests on Windows and macOS on every push. Fix platform-specific bugs: e.g., line ending issues (use `\r\n` handling on Windows if needed) or case-sensitive file system quirks on macOS/Linux.
- Perform manual smoke tests on each platform (if accessible to developers): run `adrscan` commands in a sample repo on Windows, Linux, and Mac, verifying they produce the same results and no crashes. This ensures real environment issues (like Windows path permission or Unicode handling in filenames) are caught.

- Double-check that outputs are identical across platforms for the same input (particularly relevant for hashing and diff output). This is part of **deterministic output** verification – the team ensures that the refactoring hasn't introduced any OS-dependent variation.

- **Finalize GitHub Action Integration** – *DevOps Engineer*:

- With a stable CLI now available, finalize the **GitHub Action** usage. If creating a custom action, package the CLI (or the WASM via npm) in an Action. Alternatively, use the Action to install the Rust crate via `cargo install adrscan` or the npm package and run it.
- Provide a tested workflow example: e.g., a `adrscan.yml` workflow that checks out code, installs adrscan, and runs `adrscan validate`. Ensure that if validation or diff finds issues, the action exits with non-zero code to fail the build (so ADR policy violations stop a PR).
- Add a badge to the README for the CI status (and one for crates.io version once published). This is also a good time to set up **coverage reporting** in CI (using Coveralls or Codecov) if desired, to track that test coverage requirement.

- **Documentation:** Start drafting documentation for how to use the CLI in CI pipelines. Possibly include a section in README or docs/ on *"Using ADRScan in GitHub Actions"* with copy-paste YAML snippets for users.

- **Inter-team Review:**

- At the end of Week 3, conduct a thorough review of non-functional requirements. Verify test coverage meets the target, the tool runs on all platforms, outputs are deterministic, and the WASM build is functional (even if not fully utilized yet).
- The *WASM engineer* demonstrates the NPM package in a sample project (if possible) – for example, by running a small Node script or web demo that uses the adrscan WASM to parse an ADR.
- List any remaining tasks for final week (e.g., any documentation pages incomplete, any minor feature not done, any failing tests to fix). Prioritize these in the Week 4 plan.

Week 4 – Final Polishing, Documentation, and Release

- **Code Polish & Determinism Audit** – *CLI Engineer*:
- Do a final pass through the codebase for cleanup. Remove or resolve any TODOs and ensure code readability. Check that all modules (`init.rs`, `inventory.rs`, `diff.rs`, `validate.rs`, `wasm.rs`, etc.) adhere to consistent style and error handling patterns (using `Result` and thiserror or anyhow for error management).
- Confirm **deterministic output** in all features. For example, verify that hashing uses a stable algorithm and outputs are sorted. If any randomness is used (like the default HashMap seed),

ensure it's overridden or not affecting output order [3]. Where applicable, consider using a fixed seed for hashes if cryptographic hash is used (though not usually necessary with SHA-256).
- Optimize performance if needed (30 days is short, so only obvious low-hanging fruit). For example, if scanning inventory is slow for large ADR counts, ensure it's using efficient I/O (maybe use rayon for parallel hashing if time permits, but only if safe and deterministic).

- **Final regression test:** run the full test suite and CLI manual tests one more time. Everything should be green before proceeding to release steps.

- **Documentation Completion** – *All Devs / Technical Writer*:

- Finalize the **README** and documentation. Include usage examples for each command (`init`, `new`, `diff`, `validate`), so users can quickly understand how to use the tool. Add any design rationale or links to ADR concept resources if helpful for context.
- Create a **GitHub Project Wiki** or `docs/` directory if more extensive documentation is needed (e.g., an ADRscan user guide). At minimum, ensure the crate's Rustdoc is thorough: document public functions and structures, especially those exposed via WASM (so the generated TypeScript definitions, if any, are clear).
- Add **badges** to the README: CI build status, crates.io version, npm version, and docs.rs documentation if applicable. This provides a quick at-a-glance view of project health (for example, a crates.io badge shows the published version).

- If any ADR was written to capture decisions in this refactor (for example, an ADR about "Adopting adrscan tool" or internal design decisions), ensure those ADR files are finalized and stored in the repository for posterity.

- **Release Preparation & Checklist** – *DevOps Engineer*:

- Bump the version in `Cargo.toml` to the new release number (e.g. 1.0.0 if this is the first major release after refactor). Ensure `Cargo.toml` has proper metadata: description, homepage, repository, license, keywords.
- Compile in release mode and run a `cargo publish --dry-run` to verify that all files to be published are included and the package is properly configured (no missing README, etc.) [5]. Address any warnings from `cargo publish` (like licensing or metadata issues).
- Once dry run is successful, tag the release in Git (`git tag -a v1.0.0 -m "Release v1.0.0"`). Push the tag to GitHub and observe CI. After CI passes on the tagged commit, proceed to **publish the crate to crates.io** (without the dry-run flag) [6]. Monitor the crates.io page to ensure the new version appears.
- For the WASM/NPM package: run `wasm-pack build --release` to generate the `pkg/` with the compiled WebAssembly and JS bindings. Verify the `package.json` in `pkg/` has the correct version and name. Then use `wasm-pack publish` to publish to npm (this internally uses `npm publish`) [7]. Ensure you have npm credentials set up beforehand.

- After publishing, test the installation of both artifacts: do a `cargo install adrscan` to verify the binary installs correctly, and an `npm install adrscan-wasm` (whatever the package name is) in a demo project to ensure it pulls the package. Fix any issues post-publish if discovered (e.g., missing files in package).

- **Post-release Wrap Up** – *Project Lead*:

  - Create a **GitHub Release** entry using the tag, with release notes highlighting new features (inventory hashing, diffing, validation, etc.), and link to any relevant documentation. This helps users (and team members) see what was accomplished in the 30-day sprint.
  - Update the GitHub project board: move all done tasks to "Done" column, and reflect on any incomplete items. If some non-critical improvements were left out, log them as issues for future development and possibly assign to a new milestone (e.g., v1.1 enhancements).
  - Ensure the **issue tracker** is up to date: close issues that were resolved by the refactor (linking the closing commit/PR). For any bugs discovered during the sprint that were deferred, keep them open and label accordingly (maybe label `future` or `backlog`).
  - Finally, announce the release internally or on relevant channels. Encourage the team to take pride in the accomplishment – and maybe solicit user feedback. The team should monitor for any bug reports from early adopters and be ready to issue patch releases if needed.

Throughout this 30-day sprint, the team covers all functional requirements (inventory hashing, diffing, ADR proposal generation, validation, formatting, and CI integration) and meets non-functional goals (high test coverage, multi-platform compatibility, deterministic output, and WASM support). By the end of Week 4, **ADRScan** is refactored, well-tested, documented, and published on both crates.io and npm, with a clear path for maintenance and future enhancements. [3] [2] [5] [1] [4]

---

[1] Configuring issue templates for your repository - GitHub Docs
https://docs.github.com/en/communities/using-templates-to-encourage-useful-issues-and-pull-requests/configuring-issue-templates-for-your-repository

[2] A Few Github Action "Recipes" for Rust - shift.click
https://shift.click/blog/github-actions-rust/

[3] Same deterministic code on same platform produces different outputs on different runs - help - Rust
https://users.rust-lang.org/t/same-deterministic-code-on-same-platform-produces-different-outputs-on-different-runs/119648

[4] How do I conditionally compile for WebAssembly in Rust? - Stack Overflow
https://stackoverflow.com/questions/48350087/how-do-i-conditionally-compile-for-webassembly-in-rust

[5] [6] Rust Crate Release Checklist · GitHub
https://gist.github.com/BartMassey/a8bf0d5fee366f55b6ed90c3c55ef20d

[7] pack and publish - Hello wasm-pack!
https://rustwasm.github.io/wasm-pack/book/commands/pack-and-publish.html