ChatGPT

ADRscan Requirements and Implementation Roadmap

Functional Requirements

- **Multi-Language, Multi-Repository Support:** The tool must operate across diverse programming languages and project structures. It should scan mono-repos (many components in one repo) or multiple separate repos with equal ease. This includes analyzing code and configuration files in different languages (e.g. Python, Java, Rust, etc.) and **Infrastructure-as-Code** definitions (Terraform, CloudFormation, Kubernetes manifests, etc.) to capture architectural elements. The design will allow pointing the tool at different source locations (via config or CLI args) so that even if ADRs reside in one repository and code in another, `adrscan` can still function by being configured with those paths. This ensures broad **portability** and does not assume a single specific tech stack or repository layout.

- **Rust CLI with Key Commands:** Provide a standalone Rust-based command-line interface (`adrscan`) that works offline (no network calls needed). The CLI will support the following subcommands, each addressing a core feature:

- `init`**:** Initialize an ADR directory and configuration. Running `adrscan init` in a project will create a standard folder (e.g. `docs/adr/`) with a template first ADR (like a conventions or ADR process record) and a sample config file (e.g. `.adrscan.yml`). This bootstraps the project to use ADRscan by establishing where ADRs will live and any default settings.
- `inventory`**:** Scan and inventory all existing ADRs and relevant project state. This command will parse all ADR markdown files in the repository (or specified directory) to collect their metadata and content. It will read YAML frontmatter of each ADR to extract fields like **Status** (e.g. proposed, accepted, superseded, deprecated) [1] , dates, tags, and any cross-links (like "Supersedes" or references to other ADRs). Only ADRs marked as "accepted" (and not superseded) are considered active decisions. The inventory may also gather *current state indicators* from the codebase – for example, discovering technologies in use (language versions, frameworks, DB types, cloud providers) by analyzing files and configs. The output of `inventory` can be a report (console or file) listing all ADRs with their status and key metadata, plus a summary of detected system architecture elements. This command essentially provides a **snapshot** of the documented decisions and possibly the actual state.
- `diff`**:** Perform drift detection by diffing current state against the last known snapshot of accepted ADRs. **Drift detection** means finding where the implemented architecture deviates from the architecture decisions that have been recorded. For example, if an ADR declares a certain database or cloud provider as the standard, but the code or IaC now includes a different one, that's a drift. The `diff` command will compare the latest `inventory` (current ADRs + detected architecture state) to a saved baseline (e.g. a previous inventory snapshot file). New architectural elements in code that lack a corresponding ADR, or changes in code that conflict with an ADR's content, will be flagged. This includes scenarios like **introducing a new library or service** not covered by any ADR, or code

still using something that an ADR marked as deprecated. The `diff` output should clearly list each detected divergence (e.g. "Found usage of MongoDB in code, but no ADR covers this – potential new decision needed" or "ADR X says Y, but code shows Z"). This can be output in a human-readable diff format and/or a machine-readable report (JSON) for CI consumption. By running offline and locally, it ensures sensitive code is not sent elsewhere – analysis is all on the user's machine.

- `propose`**:** Auto-generate draft ADRs for detected drift or new decisions. Based on the findings from the `diff` step, the `propose` command will create new ADR markdown files (in the ADR directory) that describe the divergence and propose to record it as a formal decision. For each drift item, `adrscan propose` might generate a skeleton ADR with a unique number/id, a title (e.g. "Adopt MongoDB for X, replacing Y"), and fill in known metadata (status set to "proposed", current date, and perhaps an autogenerated context/decision description). The content might include a brief context of what changed (e.g. "During drift detection, we found that the system is using MongoDB contrary to ADR-0005 which approved PostgreSQL. This ADR proposes updating our storage decision."). These draft ADRs serve as starting points for architects to refine. The tool should ensure it **does not overwrite existing ADRs** – new ADRs get the next available number. It may also link superseded ADRs if applicable (e.g. mark an older ADR as superseded by this proposal if a decision is changing). This feature accelerates the documentation of architectural changes by pre-populating the records for review.

- `index`**:** Generate or update an index of ADRs. This command will create an **index file** (e.g. `docs/adr/index.md` or similar) listing all ADRs with their numbers, titles, and status. It ensures the table of contents of decisions is up-to-date. Each entry in the index can include a link to the ADR file and perhaps the status badge. For example, after running `index`, users get a markdown table or list of all ADRs (sorted by number or date) with "(Accepted)" or "(Superseded by ADR-xyz)" annotations. This helps stakeholders navigate the decision records easily. The index generation will parse ADR frontmatter and content to gather the needed info. (If an index file already exists, the tool will update it; otherwise, it creates one.) This keeps documentation tidy and synchronized with the actual ADR files.

- **Configuration Flexibility:** The tool should use a configuration file (e.g. `.adrscan.yaml` or `adrscan.toml`) to allow customization without command-line flags every time. Config options include specifying the ADR directory path (if not the default), file naming conventions, file extension patterns to consider for code scanning, and any ignore patterns (e.g. directories to skip). Configuration can also allow organizations to define *key decision keywords* or mapping rules for drift detection (for example, mapping "database" to certain config files or listing approved tech stack components so the tool knows what to expect). By making this configurable, `adrscan` can be adapted to different project conventions. The config might also let users toggle features (like whether to auto-generate ADR content or just flag drift) and set thresholds (e.g. only flag drift for certain critical categories). **Snapshotting** behavior can be configured here too – for instance, the location to store inventory snapshots (default could be an `.adrscan_snapshot.json` in the ADR directory or similar). The tool will read this config at runtime to adjust its behavior, ensuring **extensibility** and ease of adoption in various environments.

- **Snapshotting and State Tracking:** ADRscan will utilize **snapshot files** to record the state of ADRs and system architecture at a given point in time. When `adrscan inventory` runs, it can output a snapshot (JSON or YAML) containing the list of current accepted ADRs (IDs, titles, key decisions) as well as a summary of detected architecture elements (e.g. "languages: [Python 3.10], databases:

[PostgreSQL], cloud: [AWS]”). This snapshot can be saved in the repo (committed to version control) so that it represents the “approved architecture state” at that commit. Later, running `adrscan diff` will load the previous snapshot and compare it to the new inventory. This approach allows detecting drift over time (e.g. between two releases) and provides traceability. The tool’s snapshot mechanism should be robust to changes in ADR files (for example, if an ADR was edited or its status changed, that’s captured as a diff) and to new/removed ADRs. If no snapshot file is present, `diff` could compare against the last Git commit of the snapshot or simply report all current state as “new” (initial run scenario). Snapshotting ensures that even if the code changes gradually, the tool can identify what changed since the last known alignment with ADRs.

- **Frontmatter Parsing and Metadata Handling:** Each ADR is expected to include a YAML frontmatter section at the top for metadata (following MADR or similar templates). ADRscan must parse this frontmatter to extract important fields for its logic. Key fields include **Status** (e.g. "proposed", "accepted", "rejected", "deprecated", "superseded" as common values [1]), **Date**, **Deciders/Authors**, and any custom tags or links (like "Supersedes: ADR-0001"). The tool will interpret these:

- **Status:** Only ADRs marked "accepted" (and not superseded) represent active decisions to enforce. ADRs that are "proposed" or "draft" are not yet binding (the tool might list them but not flag drift on them). If an ADR is "superseded", the tool should note which ADR superseded it (chain of history) and consider the superseded one as historical (ignored for current rules). For example, if ADR-0003 is superseded by ADR-0008, only ADR-0008's content is relevant to drift checks. This status awareness is crucial so the tool doesn’t mistakenly enforce outdated decisions.
- **Links Between ADRs:** If frontmatter or content indicates one ADR supersedes another, or relates to another (e.g. "Supersedes: 0003", "Related ADR: 0010 on a similar topic"), the inventory should capture these relationships. This can be used to present context in reports (like showing an ADR family history) and possibly to update statuses (marking older ones as superseded).

- **Structured Fields for Decisions:** We encourage including structured data in ADRs when possible (for instance, a section or frontmatter listing specific decisions or technologies). ADRscan will support parsing known patterns – for example, if an ADR frontmatter has a field `decisions:` listing key-value pairs (like `database: PostgreSQL` or `auth_method: JWT`), the tool can directly use those in drift detection. Even if not explicitly structured, ADRscan might use simple text analysis to identify certain technologies mentioned in the ADR’s Decision section. The more structure in ADRs, the more automatically the tool can map them to implementation. This parsing ability means ADRscan acts as a lightweight ADR **indexing engine**, making metadata easily queryable.

- **Cross-Referencing Code and ADRs:** A functional requirement is the ability to map implemented code/config back to the ADR that governs it. ADRscan will support optional markers or annotations to help this mapping. For instance, developers could include an ADR reference in code comments or commit messages (like `// ADR-0005` or in IaC scripts a comment referencing the ADR). The tool’s inventory step can scan for these references to understand where in the codebase a given ADR is relevant. This is not mandatory but if present, it greatly enhances drift detection accuracy (the tool can check if code sections aligned with ADRs have changed unexpectedly). Even without explicit markers, ADRscan will attempt heuristic mapping – e.g., if an ADR says "use Library X for Y", the tool might search the code for usage of Library X (to confirm it’s being used) or detect usage of an alternative library (which could mean drift). Ensuring such cross-referencing logic is in place will make the tool effective across large codebases where manually tracking compliance is hard.

- **Standalone Offline Operation:** The CLI must function entirely offline, analyzing local repository files. All intelligence (parsing, diff logic) resides in the Rust code – **no external API calls** or internet access is required for core features. This is important for use in closed-source projects or restricted environments. It also means the tool should bundle any reference data it needs (for example, if it has a list of known frameworks to detect, it should have that internally or allow the config to provide it). Users can run `adrscan inventory/diff` on their machines or in CI without network connectivity. This also improves security, since source code or ADR content isn't sent out. (The only time internet might be used is for optional update checks or if the user explicitly connects to something, but by default it's self-contained.)

- **WebAssembly Module (WASM) for CI/CD:** The core scanning logic of ADRscan should be compiled to WebAssembly so it can run in Node.js environments and browser-like environments. Using `wasm-pack`, the project will produce a package for npm that includes the `.wasm` binary and JavaScript bindings [2]. This NPM package (e.g. `adrscan-wasm`) will expose functions corresponding to `inventory`, `diff`, and `propose` (and possibly others as needed). CI/CD pipelines or other tools can then import this module to run ADR checks as part of automated workflows. For example, a GitHub Action or a Node script can call an `adrscan.diff()` function to get a JSON report of drifts. The WASM module must be efficient and secure: it should sandbox the file operations (the host environment will have to grant file access) and not use any non-WASM-compatible Rust libraries. Ensuring parity between CLI and WASM outputs is critical – the same logic runs in both, just packaged differently. The functional requirement is that any drift detection or proposal generation that can be done with the CLI can also be done via the WASM interface (for instance, the Action might prefer to use the WASM library directly in a JavaScript runtime for convenience). Publishing to npm with proper versioning means teams can easily pull it into their build processes.

- **GitHub Action Integration (`adrscan-action`):** To facilitate easy adoption, a GitHub Action will be provided. This Action can be implemented in two ways: either wrapping the CLI (e.g. using a Docker container with the CLI installed) or by using the WASM module under the hood in a JavaScript action. The Action's purpose is to automate drift detection on repository changes. Key features of the Action:

- It should run `adrscan inventory`/`diff` on a repository whenever there's a code or ADR change (or on a schedule, e.g. daily checks).
- If drift is detected, the Action will create a GitHub Issue (or comment on a PR) summarizing the findings. For example, "⚠ Architecture drift detected: ADR-0007 (Use AWS S3) might be outdated as GCP Storage is now referenced. ADRscan suggests a new ADR." The issue can list the divergent items and perhaps tag the relevant team to review.
- Optionally, the Action can automatically commit or PR the draft ADRs generated by `adrscan propose`. For instance, it could push a new branch with the ADR files (or attach them to the issue for manual pick-up). This speeds up the review process by giving a tangible starting point for discussing the change.
- The Action should be configurable via YAML (like any GH Action) – e.g. allowing users to enable/disable auto-propose or issue creation, set the branch name for ADR proposals, etc. Security is important: the Action must use GitHub tokens properly to create issues/PRs and should not expose any sensitive info. It operates only on the repo's content.

- The Action effectively brings **CI enforcement**: if someone makes a change that violates an ADR, the CI can catch it and alert the team, thus preventing uncontrolled architecture drift.

- **Self-Serve Training and Documentation:** A comprehensive set of documentation and training material will accompany the tool. This includes a Markdown-based **user guide** with step-by-step examples that are runnable. For instance, the guide might include a small demo project (with some sample ADRs and code) and show how to run `adrscan init`, add an ADR, then simulate a drift and run `adrscan diff` to catch it. Code blocks in the documentation will show commands and expected outputs to help users follow along. The examples should be crafted so that users can copy-paste them or run in an environment (perhaps a provided Docker image or using `cargo install adrscan` and the demo files) to see real results. Topics in documentation will include configuration setup, writing ADRs in the expected format, interpreting the diff output, customizing rules, integrating with CI, and how to publish the WASM module usage in a Node project. By making the training self-serve, teams can onboard themselves to ADRscan easily. In addition, the project will likely provide a "reusable project bootstrap" – essentially a template repository or setup scripts that new projects can use to quickly adopt ADRscan (containing a ready `docs/adr` folder, an initial ADR example, the GitHub Action workflow file, etc.). This way, implementing the tool in a new repo is straightforward.

Non-Functional Requirements

- **Performance and Efficiency:** ADRscan should efficiently handle repositories of substantial size. Scanning thousands of files for drift should be reasonably fast. This might require using multithreading (Rust async or rayon, for example) to parallelize file I/O and pattern searches across the codebase. The tool should avoid excessive CPU or memory usage; for instance, using streaming to read files rather than loading very large files entirely into memory. A typical run of `inventory` or `diff` on a medium project (say dozens of ADR files and a few hundred code files) should complete in seconds, not minutes. Performance tuning (caching results, ignoring binary or irrelevant files, etc.) is expected, and users can configure exclude patterns to skip directories like `node_modules` to speed up scanning.

- **Accuracy and Robustness:** The drift detection logic must be reliable. False positives should be minimized – for example, if code mentions something in a context not related to architecture, it shouldn't flag it incorrectly. The parsing of ADR files needs to handle minor format differences gracefully (e.g. extra spaces in frontmatter, or ADRs written in slightly different templates). The tool should be robust against malformed ADR files or unusual text; it might warn the user if an ADR cannot be parsed rather than crashing. Similarly, if certain code files can't be read (permissions issues, etc.), the tool should skip with a warning but still complete the run. **Testing** will be key (as outlined in the implementation plan) to ensure the tool handles edge cases.

- **Compatibility and Portability:** The Rust CLI should compile and run on all major operating systems (Linux, Windows, macOS). Binaries will be provided for common platforms. The WASM module must be compatible with modern JavaScript runtimes (Node.js, browsers if needed) and follow best practices for packaging (using `wasm-pack` ensures this). The project should maintain parity between the CLI and WASM outputs so that users see no difference in results. Additionally, the tool should not assume internet access (offline mode by default) and should function in sandboxed CI runners. Portability also extends to working with various repository structures – whether ADR files are in `docs/adr` or another path, whether code is in a single repo or spread out – minimal hard-coded assumptions.

- **Extensibility and Maintainability:** The codebase will be modular. For instance, the functionality for scanning code for drift will be abstracted so new rules or patterns can be added. We might design trait-based scanners for different languages or file types (e.g. a trait `DriftChecker` that can be implemented for, say, checking package manifests vs ADR-approved dependencies). Adding support for a new language or tech should be feasible without rewriting core logic. The configuration file will allow plugin-like extensibility (not actual code plugins initially, but at least the ability to define regex patterns or file glob patterns for detecting certain architecture signals). Over time, we can consider a plugin system where external crates could be loaded for custom checks, but initially maintainability through clear modular code is enough. The project should follow clean code standards, with documentation of public APIs, to encourage community contributions. We plan to open-source `adrscan`, so clear contribution guidelines and code structure will be non-functional requirements.

- **Security Considerations:** As a tool that parses potentially untrusted project files (in CI, the repository code might be from untrusted contributors), `adrscan` must be secure. Written in Rust, it benefits from memory safety (preventing buffer overflow, etc.). It will avoid executing any code from the repository – it purely reads files as data. This prevents issues like remote code execution. We must ensure that parsing does not, for example, accidentally evaluate any embedded scripts in files. The output (especially in GH Action context) should be sanitized – e.g. if an ADR file had malicious content or extremely long lines, the tool should handle it gracefully (perhaps truncating overly long outputs in issue comments to prevent spam). The Action uses GitHub's token to create issues/PRs; we will follow the principle of least privilege (the token is provided by GH runner and scoped to the repo). No sensitive data (like secrets) should be read or transmitted by the tool. Also, when publishing to npm and crates.io, we will ensure the supply chain is secure: use CI to build reproducibly, sign releases if possible, and clearly version the releases so users can pin to known-good versions.

- **Reliability and Error Handling:** The tool should provide clear error messages when something goes wrong – for example, if the configuration file has an invalid format, or if the ADR directory is not found. It should exit with non-zero status on critical errors (so CI can detect a failure), but for minor issues (like one ADR file parse failed), it might continue processing others and then report the single failure at the end. Logging can be implemented (with verbosity levels) so users can get more insight if needed (e.g. a `-v` flag for debug info about which files are being scanned or which rule triggered a drift finding). Ensuring that the CLI doesn't panic and crash is important; all foreseeable errors should be caught and handled. We will include unit and integration tests to cover scenarios and ensure **reliability** of outputs.

- **Usability and UX:** Despite being a developer tool, emphasis on usability is a must. The CLI interface and the outputs should be user-friendly. Commands and flags should follow conventions (using Clap or similar for consistent UX). The text output for `diff` should be easy to read: e.g. using colors or clear symbols (if terminal supports) to indicate additions/removals in architecture. The content of proposed ADR drafts should be clean and formatted according to the established ADR template (MADR or other configured template), requiring minimal editing. The documentation (non-functional aspect) should be thorough – which we plan in Phase 4 – because a tool is only as good as its adoption, and adoption comes with clear docs and examples. Also consider localization/internationalization in the future (for now, probably English-only, but at least not hard-coding strings that make it impossible to adapt).

- **Cross-Repo and Team Workflow Considerations:** On a higher level, the tool's design should accommodate organizations that maintain a central repository of ADRs separate from code. For example, a company might have an `architecture-decisions` repository containing ADRs for all projects. ADRscan should allow pointing to that repo's ADR directory while scanning multiple other repos for drift. While a full multi-repo scan in one go might be complex, the tool could be invoked separately for each code repo but referencing the same ADR source. This implies that the inventory and diff logic can accept multiple input paths (one for ADRs, one for code) or that the config can specify external ADR sources. We won't implement full distributed scanning in the MVP, but the architecture should not preclude it. At minimum, it should be easy to run ADRscan on each repo and aggregate results. This also means the **output formats** (like the JSON report from `diff`) should be machine-mergeable if needed (so an organization could combine results from several runs). Essentially, the tool should remain flexible to fit into various workflow automations.

- **Versioning and Stability:** As development progresses, we will follow semantic versioning for releases. Early phases (MVP) might be 0.x versions, where we can still make breaking changes based on feedback. By Phase 3 (when the GitHub Action and WASM are released to a broader audience), we aim for a 1.0 release, implying a stable CLI interface and stable JSON output schema for the library. Non-functional requirement is to clearly communicate deprecations or changes to users (via CHANGELOG, etc.). The CLI's **backward compatibility** should be respected once at 1.0 – scripts or CI that rely on it should not break unexpectedly. The npm package API (the functions exposed) similarly should remain stable or follow semver for breaking changes. This stability guarantee is crucial for trust, especially as teams integrate the tool into critical CI pipelines.

Implementation Plan (Phased Rollout)

To deliver ADRscan effectively, we will implement it in four planned phases. Each phase builds on the previous, adding functionality and hardening the system. Below is the roadmap with details on deliverables, testing, stability, and effort for each phase.

Phase 1: Rust CLI MVP

- **Deliverables:** A minimum viable product of the `adrscan` Rust CLI covering basic features. In Phase 1, the focus is on the core ADR management functions. We will implement the `init`, `inventory`, and `index` commands first, as these lay the groundwork. Specifically:
- `init` will set up the ADR directory and create a sample ADR (template) and config file.
- `inventory` will parse ADR files (with frontmatter) and list them, but initially the drift detection integration will be minimal – it might just list technologies found without deep comparisons.

- `index` will generate a simple index.md of ADRs.
  The `diff` and `propose` commands may be stubbed or very basic in MVP (for example, `diff` might just compare counts of ADRs or do nothing substantial yet). The goal is to release a working CLI that can manage ADR files (creation, listing) and demonstrate parsing of metadata. All this will be

done in Rust with a well-defined project structure (e.g. modules for parsing, commands, etc.). Documentation will start with a basic README explaining usage of these commands.

- **Testing Strategy:** Phase 1 will include unit tests for critical functionality like frontmatter parsing (e.g. ensuring we correctly read the status, title, etc. from ADR markdown files) and for file operations (like `init` creates the right files). We'll also write integration tests simulating a user workflow: e.g. call `adrscan init` in a temp directory, then create a couple of ADR files manually, run `adrscan inventory` and verify the output lists those ADRs. These tests ensure the CLI behaves as expected. Given it's an MVP, we will test on at least one platform (Linux) thoroughly, and do smoke tests on Windows and Mac to catch obvious path issues. The testing focus is on correctness of basic operations and preventing regressions as we add more features later.

- **CLI/API Stability:** At this MVP stage, the tool is pre-1.0 (likely version 0.1.x). We **do not guarantee full stability** of the CLI interface or output yet – we will gather feedback and potentially adjust commands and options. However, we will aim to keep the basic concepts (init, inventory, etc.) consistent. Since the WASM module is not delivered yet in this phase, API stability for that is not a concern here. Users of the Phase 1 CLI should be aware it's an early release; stability guarantees are low, and breaking changes might occur in Phase 2 as we refine the design.

- **Estimated Effort:** Roughly **4-6 weeks** of development for a small team (1-2 developers). This accounts for setting up the project (project scaffolding, CI pipeline for building releases), implementing the core commands, and writing tests and initial docs. Since Rust is being used, some time will be spent choosing crates (e.g. for CLI arg parsing like Clap, for YAML frontmatter parsing, etc.) and ensuring cross-platform builds. Given the limited scope (no complex diff logic yet), this timeline is feasible. Additional buffer is included for refining the ADR parsing (which might involve tweaking the markdown/YAML handling if ADR files vary).

Phase 2: Drift Detection & ADR Proposal Engine

- **Deliverables:** Phase 2 introduces the heart of ADRscan's unique value: full **drift detection** (`diff` command) and the **ADR proposal generation** (`propose` command). We will implement:
- A robust `diff` command that compares the current code/infrastructure state to the accepted ADRs. This involves building the logic to scan the repository for key changes. Deliverables include writing detectors for common drift types (e.g. new dependency or tech usage, or something conflicting with an ADR). For example, if an ADR says "Cloud = AWS", we'll include a detector that scans IaC for mentions of other cloud providers and flags them. We'll also implement the snapshot loading/storing so that `diff` can remember past state.
- The `propose` command fully implemented to generate markdown files. We will likely create a template for proposed ADR drafts. The deliverable is that running `adrscan propose` after a drift is detected will result in new file(s) like `adr/YYYY-MM-DD-X.md` (with X being next number) populated with a title and sections (Context, Decision, Consequences) partly filled out. This might use a template engine or hard-coded template strings.
- Possibly enhancements to `inventory` in this phase: for drift detection to work well, `inventory` might also gather the "current state" snapshot (list of tech detected). So we deliver an updated inventory that not only lists ADRs but also outputs the snapshot data file.
- We also plan to improve the `init` if needed (for example, include in the template config some default drift rules).

- By end of Phase 2, the CLI should essentially have all major functionality implemented (manage ADRs, detect drifts, propose new ADRs).

- **Testing Strategy:** Testing in this phase will be more extensive, because drift detection has many scenarios. We will create multiple **sample projects** with known conditions to test:

- A project where nothing drifted (all code matches ADRs) to ensure `diff` correctly reports no issues.
- A scenario where a new tech is introduced (like an ADR says "Database: MySQL" but we add a Postgres config) and verify `diff` catches it.
- A scenario where an ADR is superseded by another and code reflects only the new one (should be no drift).
- Cases where code still contains something that an ADR deprecated.

For each, we'll assert that the `adrscan diff` output flags the correct items. We will also test the content of generated ADR proposals: ensure the markdown is valid, the frontmatter is present with correct status ("proposed") and that it references the drift (maybe including the old ADR link or code evidence). Unit tests will cover the lower-level detectors (e.g. a function that checks a list of dependencies against ADR rules). We will incorporate user feedback (if any early adopters) to add tests for any missed drift patterns. Performance tests might be done in this phase to ensure that adding drift detection didn't degrade speed unacceptably (for instance, test `diff` on a large dummy repository).

- **CLI/API Stability:** By the end of Phase 2, we anticipate the CLI commands and options will mature. We will likely iterate based on what we learn implementing drift detection (maybe adding flags like `--snapshot` to specify a snapshot file, or options to limit scope of diff). Once finalized in this phase, the CLI interface should approach stability. We might label this release as 0.9.x – feature-complete in terms of functionality but still pre-1.0 until we test integration in Phase 3. API stability: at this point, we will refactor the code to clearly separate logic that will be exposed as a library (for WASM). The internal functions for inventory, diff, propose will have stable signatures that can be called programmatically. We intend to use this as the basis for the WASM binding in the next phase. So, Phase 2 is about stabilizing the **core logic** if not the official 1.0 release yet. We will document any breaking changes from Phase 1 (e.g. if we changed output formats or config keys) and try to minimize further breaking changes moving forward.

- **Estimated Effort:** Approximately **6-8 weeks** for this phase. Drift detection is the most complex part, and will involve research and tweaking. We need to allocate time for implementing several detectors (language detection, dependency analysis for at least a few ecosystems, cloud/IaC scanning, etc.). Also generating ADR proposals requires careful templating and possibly iterative refinement to produce useful drafts. Writing and running the expanded test suites will take a significant portion of the effort as well. Considering potential complexity, we assume closer to 8 weeks if thorough testing and some refactoring is needed. This phase likely requires collaboration with architects or potential users to validate that the drift checks align with real-world expectations, which we will account for in the timeline.

Phase 3: WASM Module & GitHub Action Integration

- **Deliverables:** In Phase 3, we take the now robust CLI logic and make it available in environments like CI and npm:

- **WASM Module:** Using `wasm-pack`, compile the core library to WebAssembly and produce an npm package (e.g. `@myorg/adrscan`) that can be imported in Node.js. Deliverables include writing JavaScript/TypeScript bindings or wrapper functions for the core features. For example, the npm package might expose `async function inventory(path, config) -> returns JSON`, `diff(oldSnapshot, newSnapshot) -> returns JSON diff`, etc., making it easy to call from a Node script or action. We will include TypeScript definitions for these for better developer experience. We will ensure the package is published to npm with proper README on how to use it. This also involves verifying that all needed functionality is compatible with WASM (file access will be through the WASM FS or an interface we define, possibly requiring the caller to supply file contents or run in Node with access to fs).
- **GitHub Action (`adrscan-action`):** Develop a GitHub Action that uses the above WASM module or the CLI. The deliverable is a ready-to-use Action definition (YAML and any supporting code, likely a JavaScript action if using the WASM). This Action will handle inputs (e.g. user can configure what commands to run: just diff, or diff+propose) and perform the logic to create issues or PRs. We will implement the automation to open an issue when drift is found, including relevant details (probably using the GitHub REST API via octokit in the JS action). If auto-propose is enabled, the Action will commit the new ADR file on a branch and either open a PR or attach to the issue. We will provide a default issue/PR title and body format, which can be refined by the user through action inputs or templates. Another deliverable is documentation for the Action usage (how to add it to a workflow file).

- **Additional CLI Polish:** In parallel, we may release a CLI update if needed to support the Action better (for example, a `--ci` flag or output modes like `--format=json` for easier consumption by the Action if we decide to use CLI via Docker). This phase ensures that the CLI and Action play well together. We might also address any cross-platform build issues now, ensuring the CLI can be installed easily (Homebrew tap for Mac, etc., although the Action will mostly use the WASM to avoid platform dependence).

- **Testing Strategy:** Phase 3 testing has two main areas: the WASM library and the GitHub Action.

- For the **WASM module**, we'll write tests in JavaScript (or TypeScript) that import the packaged module and call the exposed functions. We'll use a sample repository (perhaps as a fixture) to simulate running `inventory` and `diff` via the WASM. We need to verify that results are identical to running the CLI on the same input. This ensures no regression or discrepancy introduced by the compilation to WASM. We'll test on Node.js (different versions if possible) and also consider testing in a browser environment (just to see if it could run, though primary target is Node).
- For the **GitHub Action**, we will use GitHub's recommended approach of testing actions. This may include running the action in a simulated environment using `act` (a local runner) or creating a test repo where we install the action and trigger it. We need to confirm that when drift is present, the action indeed creates an issue with the expected content, and when no drift, it perhaps posts a success message or no issue. If the action is set to create a PR for new ADRs, test that the PR is correctly formed (branch contains the new ADR file, PR description mentions linking to the issue or context). We will also perform security testing on the action (making sure it doesn't expose secrets in logs, for example).

- Additionally, integration tests could chain everything: e.g. push a commit to a test repo that introduces drift, then see if the action (running via CI) catches it and produces the outputs. While this is a bit beyond unit testing, it's a validation of the end-to-end functionality.

- **CLI/API Stability:** At this stage, we plan to reach a **1.0 release** for both the CLI and the WASM API. That means we consider the feature set complete and have ironed out major issues from phases 1 and 2. The CLI commands and flags should now remain stable (any future changes would be additive or in a major version bump). We will lock in the output formats (for example, if `adrscan diff --format=json` outputs a certain schema, we document it and won't change it without version increment). The WASM API (which is essentially a set of functions or methods available to consumers) will also be versioned; once published, changing function signatures or behavior would require a new major version. We will provide documentation of this API. Essentially, Phase 3's end marks the tool being production-ready for wider use, so we commit to backward compatibility from this point onwards in the 1.x series. The GitHub Action v1.0 will correspond to the CLI/WASM 1.0 and will also maintain its interface (action inputs, outputs, etc.) stable so that projects can rely on it long-term.

- **Estimated Effort:** Approximately **4-5 weeks**. A chunk of this time is for the technical work of configuring the build to produce WASM and writing the bindings, which can have tricky parts (like bundling the WASM, ensuring small size, etc.). Another portion is developing the GitHub Action – writing the action code (likely in JavaScript/TypeScript), testing it, and publishing it to the GitHub Marketplace. Documentation updates are also part of this phase (writing usage guides for the npm library and action). Since by now the core logic exists, the effort is more on integration and distribution, which is why the timeline is a bit shorter than Phase 2. However, if we encounter issues (for example, performance in WASM or limitations like file system access), we may need some extra time to devise solutions (such as Node-specific hooks to read files). 5 weeks is a reasonable target assuming 1-2 developers focusing on this, given a lot of it is build/packaging and not creating new algorithms from scratch.

Phase 4: Documentation, Training Materials, and Project Bootstrap

- **Deliverables:** The final phase concentrates on **polish and adoption**:
- Complete and publish all documentation. This includes a comprehensive user guide (in Markdown, possibly as a GitHub Pages or docs site) covering installation, configuration, command usage, examples, and troubleshooting. We will also deliver the self-serve training tutorial as described in requirements – likely as a series of markdown files or an interactive Jupyter Notebook (if we integrate with something like GitHub Codespaces) that users can run. The deliverable is that a new user can follow the tutorial and learn how to use ADRscan step by step, with minimal assistance.
- A **reusable project template** (bootstrap) will be delivered. This could be in the form of a GitHub repository template that has ADRscan already configured. For example, a repo with a pre-made `docs/adr` folder with an ADR template and a sample ADR, the `.adrscan.yml` config filled with typical defaults, and a GitHub Actions workflow file using `adrscan-action`. This allows teams to click "Use this template" on GitHub to create a new repo that's already set up for ADR management and drift checks. Alternatively, it could be a script or subcommand (perhaps `adrscan init --scaffold`) that adds all these pieces to an existing repo. Either way, the deliverable is to simplify the onboarding for new projects.
- Another deliverable is finalizing any remaining features deferred from earlier phases. For instance, if during documentation we realize a need for a small feature (like a `adrscan new` command to create a new ADR from CLI, similar to other tools), we might implement that now since core is done. We also ensure that all example code and outputs in the documentation are up-to-date with the actual tool behavior (running the tool to generate those outputs for realism).

- We will also prepare outreach materials (like a blog post or release notes) announcing ADRscan 1.0, summarizing its capabilities. This is not a requirement per se, but part of phase 4 we consider "making sure people know how to use it and can trust it."

- **Testing Strategy:** In this phase, testing is about verification and user acceptance:

- We will do final cross-platform testing of installation and usage (install via Homebrew on Mac, via Cargo on Linux/Windows, use the npm package in a fresh project, etc.) to ensure the onboarding is smooth. We'll double-check that all links in documentation work (for example, if we host documentation, ensure the code examples are correct).
- If possible, we may run a small **pilot usage** with a friendly team or an example open-source project to see ADRscan in action and see if the documentation left anything unclear. Their feedback can drive last-minute tweaks or clarifications.
- We also verify that the GitHub Action and WASM package still work with the final version in a real scenario (a dry-run on a sample repo, making sure the issues created by the action are well-formatted and helpful).

- Since this phase involves minimal new code (mostly docs), the "testing" is more proofreading and validating the learning materials produce the intended results. We will incorporate any errata found into immediate fixes.

- **CLI/API Stability:** By phase 4, we are in the stable release track (1.x). Any changes in this phase should avoid breaking anything. If we did add a minor feature like `adrscan new` for convenience, it would be additive. We consider phase 4 the polish on a stable foundation – so the CLI, API, and Action are all stable. We will tag a 1.0.0 (if not already done in Phase 3) or a 1.1.0 if we added features. From here on, we would only release non-breaking improvements or critical fixes unless a future 2.0 is planned. We will explicitly state the support policy: for example, if any bugs are found, we'll patch them in point releases; any feature requests will be considered for future minor releases.

- **Estimated Effort:** Around **2-3 weeks**. Writing and reviewing documentation is time-consuming but we have a lot of material from prior phases (READMEs, test cases, etc., plus possibly Phase 3's action usage guide). Assuming one person dedicating effort, 2 weeks to compile and refine all docs is realistic, with an extra week for review and setting up the template repository. This phase also includes the time to respond to any initial user feedback as the tool might have been used by early adopters by now. Because no heavy coding is expected (aside from perhaps small enhancements), the effort is mainly in careful writing and enabling easy adoption.

Each phase will conclude with a review and a **go/no-go checkpoint** to ensure quality before moving to the next. By following this phased roadmap, we ensure that ADRscan evolves systematically – starting from a solid ADR management foundation, then adding the innovative drift detection capabilities, then making it broadly accessible via CI integration, and finally ensuring users can easily learn and bootstrap it. This plan will deliver a powerful tool that helps organizations keep their architecture decisions in sync with their code, addressing the risk of architecture drift which is often "difficult to detect" and risky in fast-changing environments [3]. With ADRscan, teams get automated insight into when their implementation strays from their documented decisions, and assistance in recording new decisions to close the gap, all in a portable, secure, and user-friendly package.

---

1 Template - Clan Documentation
https://docs.clan.lol/decisions/_template/

2 Packaging and publishing - Hello wasm-pack!
https://rustwasm.github.io/wasm-pack/book/tutorials/npm-browser-packages/packaging-and-publishing.html

3 Architecture Drift: What It Is and How It Leads to Breaches
https://www.crowdstrike.com/en-us/blog/architecture-drift/