ChatGPT

SOC 2 Compliance Automation: Requirements and 30-Day Implementation Roadmap

Automating SOC 2 compliance evidence collection and readiness checks via GitHub is a multi-faceted project. This report consolidates functional and non-functional requirements (drawn from prior discussions and the repository), evaluates the current state of the **Soc-2-compliance** repository, and provides a phased implementation plan (over ~30 days) with Gherkin-style scenarios as guidance. The focus is on **SOC 2 Type II** compliance (continuous control effectiveness), as opposed to Type I (point-in-time design); we aim to go straight for Type II unless a preliminary Type I-style readiness step is absolutely necessary [1] [2].

Functional Requirements

- **Automated Compliance Dashboard:** The solution must automatically generate a **SOC 2 compliance readiness dashboard** that tracks controls and their implementation status. This dashboard should consolidate evidence from various scans and checks into an easy-to-read format (e.g. a markdown table or HTML report). It should map to the SOC 2 Trust Services Criteria, showing each control, expected evidence, implementation notes, and a status (e.g. *Planned*, *In Progress*, *Implemented*) [3] [4]. The dashboard provides at-a-glance insight into compliance gaps and progress for auditors and the team.

- **Integrated Security Scanning:** Leverage multiple scanning tools to gather evidence and detect issues:

- *Infrastructure-as-Code (IaC) Misconfigurations:* Use **Checkov** and **tfsec** to scan Terraform/Cloud config for security compliance. The tool should utilize Checkov's built-in rules for SOC 2 and other standards to ensure infrastructure meets required controls (encryption, network isolation, etc.) [5]. *Scenario:* If an S3 bucket is defined without encryption, the Checkov scan flags it as a violation of SOC 2 security controls before deployment [5] [6].
- *Dependency Vulnerabilities:* Use **Trivy** to scan application dependencies (open-source libraries, Docker images) for known vulnerabilities. Trivy is an all-in-one open source security scanner that can find CVEs and misconfigurations across code, binaries, container images, etc [7]. This ensures **Secure SDLC** practices by identifying libraries that could violate SOC 2's security criteria (e.g. high-severity CVEs must be remediated) [8].
- *Secret Exposure:* Use **TruffleHog** to detect hard-coded secrets and credentials in the repository. TruffleHog can scan the Git history and code for patterns or high-entropy strings indicating API keys, passwords, or tokens [9]. (This supports SOC 2 controls around logical access and data protection by ensuring no secrets are inadvertently leaked in code).
- *Repo Security Posture:* Use the **GitHub CLI (`gh`)** to capture repository metadata relevant to SOC 2 controls. For example, verify that 2FA is enabled for all collaborators, branch protection rules (requiring PR reviews & CI checks) are in place, and dependency review is enabled [10] [11]. This automates evidence collection for controls like access management and change management.

- *Policy Documentation Generation:* From a structured **controls.yml**, generate human-readable policy documents or tables. Each control in SOC 2 (e.g. CC1.1 "Control Environment") should be mapped to the organization's implementation and evidence. The tool should output or update files (Markdown or PDF) that an auditor can review, derived from the controls.yml data. For example, a script can convert `controls.yml` into a Markdown table of controls vs. evidence (similar to the existing `soc2-table.md`) for the dashboard [3] [4].

- **CI/CD Integration (GitHub Actions):** The compliance checks must be executable as a GitHub Action in CI, triggered on demand or on schedule. The preferred approach is a GitHub Action workflow that runs the scans and produces artifacts on every push to main or periodically (e.g. nightly). This ensures continuous compliance monitoring in line with SOC 2 Type II (which evaluates controls *over time*). For initial development or standalone use, running the tool manually (via a script) should also be possible – but the end goal is seamless CI automation. The GitHub Action should run on a **self-hosted runner** if needed (to have all tools available), and the action can be included in any repo by adding it to the `.github/actions` folder and referencing it locally [12] [13]. *(This local-action approach was used in the repository's example usage.)*

- **Artifact Collection and Reporting:** All outputs (scan results, generated policies, logs) should be gathered as **artifacts** for review. The workflow should upload these artifacts, and crucially, integrate with GitHub's Code Scanning feature by uploading results in SARIF format. By converting scanner outputs (Checkov, tfsec, Trivy) to SARIF and using the `github/codeql-action/upload-sarif` action, we can display third-party scan alerts in the repository's **Security** tab [14] [15]. This satisfies SOC 2's requirement for code vulnerability scanning to be surfaced and tracked in the SDLC [16]. *Example:* After the action runs, any misconfiguration or critical vulnerability is visible as a Code Scanning alert on GitHub, and the compliance dashboard marks the related control as "Not Implemented" until resolved.

- **Focus on SOC 2 Type II Controls:** Ensure the tool covers the core **Trust Services Criteria (TSC)** relevant to SOC 2. Initially, concentrate on **Security (CC series)** and **Change Management (CC4.x)** controls via the checks above, since Type II audits will test that these controls operate effectively over a period [17] [18]. If obtaining a Type I report is deemed a necessary milestone (as a readiness check), the tool's output should also support that (essentially a point-in-time snapshot), but priority is given to capabilities that enable ongoing monitoring for a Type II audit [2].

- **Ease of Use and Configuration:** The solution should require minimal effort to adopt. Aside from installing the necessary scanner tools (see non-functional reqs), a user should just include the action and perhaps adjust a config file (like `controls.yml`) to match their specific controls. The `controls.yml` should be easily extensible, so organizations can add their own controls or map to additional frameworks (future scope). Default **policy templates** (e.g., for common SOC 2 policies) could be provided to jump-start documentation generation [19]. Additionally, results should be presented in a clear manner (e.g., the statuses in the dashboard should be color-coded or use emojis to quickly denote pass/fail).

- **GitHub Marketplace Readiness:** The project should be structured so it can be published as a reusable Action on the GitHub Marketplace. This includes having a single `action.yml` metadata at the repository root (defining inputs, outputs, branding, etc.), a clear README with usage instructions (already partially done), and version tags for releases [20]. No extraneous workflows should reside in the repo (to comply with marketplace rules) [21]. The end result is a tool that others can easily find and use via Marketplace (installable with `uses: <owner>/soc-2-compliance@v1`).

Non-Functional Requirements

- **Environment & Dependencies:** Because multiple scanners are involved, the runtime environment must have these tools available or packaged. The current action expects the host runner to provide `gh` (GitHub CLI), `docker`, `trufflehog`, `tfsec`, `checkov`, `python3`, and `npx` [22]. This is acceptable on a self-hosted runner where one can pre-install tools. However, to improve portability, consider a Docker container for the action that includes all dependencies, or an installation step for each (e.g., use a composite Action to install missing tools). Ensuring all scanners run with compatible versions is important for reliability.

- **Performance:** Scans should complete in a reasonable time to be usable in CI pipelines. Each tool should be configured to scope to the repository's content (e.g., if no Terraform, skip tfsec) to avoid unnecessary work. Running tools in parallel jobs where possible can speed up execution. The 30-day roadmap should include optimizing the workflow (for instance, using artifact caching for dependency scans or limiting scans to changed components on each run) so that the CI job stays efficient.

- **Security & Privacy:** Since the tool handles potentially sensitive data (e.g. scanning for secrets, handling policy docs with internal info), it must safeguard this data. Any discovered secret should **not** be publicly exposed in logs – the action should mask secrets or only report a high-level finding (so developers can rotate the secret). Collected evidence and reports should be stored as artifacts or in a secure location, not posted to public channels. Additionally, the action itself should follow security best practices (pinning action versions, least privilege for any tokens used). Running the compliance action should not itself introduce vulnerabilities or require overly broad permissions (it typically just needs repository read access, and perhaps security events write access for uploading SARIF).

- **Usability & UX:** The output (dashboard, logs, alerts) should be **clear and actionable**. Developers should easily understand what needs to be fixed when a control is non-compliant. For example, if a dependency with a CVE is found, the dashboard could mark the relevant control (e.g. CC3.1 - Logical Access or CC7.1 – System Security) as "⚠ Not Compliant" with a note to update that dependency. Where possible, link to additional context or remediation steps (maybe in the markdown table, the "Implementation Notes" column can provide remediation tips or links). The solution should also integrate with existing developer workflow: e.g., failing the CI build on critical issues to enforce prompt fixes (aligned with SOC 2's requirement that high-severity issues are remediated timely [8]), and perhaps opening GitHub issues automatically for outstanding compliance tasks (as a future enhancement).

- **Maintainability & Extensibility:** The framework should be structured to allow new checks or integrations. If, for instance, the organization later wants to add **Privacy** criteria checks or integrate an **Audit logging** tool, the codebase (scripts and config) should accommodate that without major rewrites. Using config files (YAML/JSON) for control definitions and mapping to evidence makes it easier to extend. Clear documentation and a contribution guide (the repository already includes a `CONTRIBUTING.md`) will help maintain and grow the project.

- **Alignment with SOC 2 Criteria:** The tool's scope should cover at least the **Common Criteria (CC)** across the Trust Services Principles of Security, and any additional criteria the organization chooses (Availability, Confidentiality, Processing Integrity, Privacy, as applicable). Non-functional quality here means ensuring the tool's checks truly address the stated controls. For example, if a control expects "Customer data is encrypted at rest," the tool should verify encryption settings (via IaC scan or cloud config). We should avoid false sense of security: each automated check must accurately reflect compliance status for that control (or we clearly document what is not covered by automation).

Current Repository Evaluation

The current repository **tbowman01/Soc-2-compliance** already implements a foundation of the above functionality. It defines a GitHub Action (located in `.github/actions/soc2-compliance`) that runs a series of scripts for SOC 2 evidence collection. According to the README, this action **"automates the generation of SOC 2 compliance artifacts during your CI pipeline"** by integrating multiple tools: **Checkov and tfsec for IaC misconfiguration scanning, Trivy for dependency scanning, TruffleHog for secrets scanning, plus capturing GitHub repo metadata and generating policy documents from structured controls** [23] . This design aligns well with the functional requirements identified. The usage snippet shows inclusion as a local action, which is suitable for testing and internal use [12] .

Key strengths of the current implementation: - It covers all major scanning areas (infrastructure, dependencies, secrets) with popular open-source tools, which likely map to various SOC 2 controls (for example, Checkov/tfsec help with CC6.1 Change Management and CC7.1 system configuration hardening, Trivy with CC7.2 Vulnerability Management, etc.). - It uses a **structured controls file (`controls.yml`)** to drive policy generation. This indicates an attempt to map the Trust Services Criteria to specific evidence. In fact, the repository contains a `soc2-table.md` (likely generated or to be generated) listing controls like CC1.1, CC1.2, etc., with expected evidence and status [3] [4] . This is effectively the compliance dashboard in markdown form. - The action is set up to run in a CI job and requires certain tools in the environment, as documented in the README's Requirements [22] , so the author has enumerated the pre-requisites clearly.

Areas for improvement or gaps observed: - **GitHub Marketplace prep:** Currently, the action is invoked via the local path (`uses: ./.github/actions/soc2-compliance`). For Marketplace, the repository would need a top-level `action.yml`. It appears the code is structured as a composite action (shell scripts in `scripts/` folder) rather than a container action. That's fine, but to publish it, the repository should contain *only one* action metadata at root and no workflow files [20] . We might need to move `soc2-compliance` action definition to root or ensure it's the primary one. Also, version tagging and adding a proper `name`, `description`, and branding in `action.yml` would be needed. - **SARIF upload integration:** It's not clear if the current repository already uploads results to CodeQL/Code Scanning. The listed scripts (`collect-artifacts.sh`, etc.) likely output some results to files under `audit-results/` or `compliance-artifacts/`. We did not see an explicit step for SARIF upload in the README's usage. Implementing the SARIF upload (using GitHub's `upload-sarif` action) is mentioned in the requirements and will likely be a new addition. This will allow security alerts to show up natively on GitHub [14] [24] . - **Dependency on external tools:** As noted, the action expects a lot of binaries to be pre-installed. This can hinder ease of use for new users (especially if using GitHub-hosted runners which would require installing these on the fly). Containerizing the action or providing an installation script could improve this. This is something to address in the roadmap. - **Evidence depth and coverage:** The `controls.yml` and `soc2-table.md` suggest a manual or partially automated entry of control statuses. For example, an entry like CC4.1 (Change Management) with “Protected branches, peer review enforcement, Trunk-based CI/CD” marked *Implemented* [25] implies that the repository’s settings were configured accordingly. Ideally, the tool would **automatically verify** those (e.g., query branch protection via `gh api` to confirm PR reviews are required). It’s not clear if the current scripts do such checks. Enhancing this automation (so that the Status in the table is derived from actual scanning/queries rather than manual updates) is a potential improvement. - **Testing and Reliability:** There’s no mention of tests in the repo (no CI badge, and “0 forks/stars” indicates it’s early stage). Before publishing widely, adding some automated tests (even just sample data tests for the scripts) would be valuable to ensure reliability. For instance, a test could run `validate-dependencies.sh` on a sample `package-lock.json` with a known vulnerable library and assert that it catches it. - **Documentation and Templates:** The README, while concise, could be expanded to include an example of the dashboard output, instructions for setting up required secrets (if any, e.g., if GitHub API calls need a token beyond GITHUB_TOKEN), and how to interpret results. Also, providing **issue/PR templates** could improve UX (point #9 in the request): e.g., an issue template for “New Control Onboarding” or a PR template that includes a checklist for compliance (did you update documentation? did scans pass?). These templates help keep compliance in mind during development.

Overall, the current repository establishes a good foundation, and the next steps involve polishing and extending it to fully meet the requirements and be usable by others.

Phased Implementation Plan (30 Days)

Below is a phased plan to implement the remaining features and improvements in priority order, using **Gherkin-style scenarios** to describe the desired outcomes for each phase. Each phase is roughly 10 days, but can be adjusted. The focus is on delivering incremental value: **Phase 1** sets up the core scanning and dashboard functionality, **Phase 2** adds integration and security improvements (CodeQL, etc.), and **Phase 3** preps the project for public use and future growth.

Phase 1 (Days 1–10): Core Compliance Automation and Dashboard

Goals: Establish a working CI pipeline that runs all required scans and generates a compliance dashboard artifact. Ensure that for each Trust Criterion in scope, we collect at least one piece of evidence or status. This phase delivers the essential functionality for internal use in a single repo.

- **Setup & Configuration**
  Scenario: Initializing the SOC2 compliance action in a repository
  Given the repository contains the necessary config files (`controls.yml` with defined SOC 2 controls) and the `soc2-compliance` action (or it’s installed via Marketplace in future),
  When I add the workflow YAML to run the `soc2-compliance` action (triggered on pushes or manually),
  Then the workflow should execute all steps of the compliance scan (IaC check, dependency scan, secret scan, metadata capture) without errors,
  And it produces output directories (e.g. an `audit-results/` or artifact files) containing the results from each tool (e.g. `checkov_report.json`, `trivy_results.txt`, etc.),

And it generates or updates the `soc2-table.md` (compliance dashboard) reflecting the latest status of each control.

- **Scanning Functions Execution**
  Scenario: Running integrated scans for code, dependencies, and secrets
  Given the action has access to codebase and required tools (Checkov, tfsec, Trivy, TruffleHog installed),
  When the action's script for IaC scanning runs,
  Then it scans all Terraform/cloud config files and logs any SOC2-related misconfigurations found [5] (e.g., open security groups, missing encryption),
  And when the dependency scanning step runs, it identifies any vulnerable library (e.g., a known critical CVE in a package) and logs it,
  And when the secret scanning runs, any detected secret is reported (with appropriate masking).
  Implementation detail: We will configure each tool: e.g., `checkov --framework terraform --quiet --output json` etc., `trivy fs --exit-code 0 --security-checks vuln,config .` for dependencies, `trufflehog filesystem . --json`. These outputs will be saved for later phases.

- **Compliance Dashboard Generation**
  Scenario: Automated assembly of the compliance status table
  Given we have a structured list of controls and corresponding evidence results from scans,
  When the policy generation script runs (e.g., `generate-policies.sh`),
  Then it reads `compliance-config/controls.yml` which contains entries like "CC1.1 – Control Environment expects Code of Conduct, etc.",
  And it populates a Markdown table (`soc2-table.md`) with each control's **Title**, **Expected Evidence**, **Implementation Notes**, and **Status**.
  And for controls that can be tied to automated checks, it sets the Status based on scan results.
  And for controls not automatically verifiable (e.g., "Board Oversight" with meeting minutes), it leaves them as *Planned* or *Not Started* for manual input.
  Example: After Phase 1, the `soc2-table.md` might show CC4.1 "Change Management" as *Implemented* with notes "Protected branches, peer review enforced" if the repository's branch protection is detected, or *Not Implemented* if not. (Initially, we might require manually updating the Status until Phase 2 where we automate that detection.)

- **Basic Reporting & Artifact Upload**
  Scenario: Uploading compliance artifacts for review
  Given the scans produce output files (reports, logs, table markdown),
  When the job completes,
  Then all key artifacts are available in the workflow run (e.g., as zipped artifacts or as part of the repository in a specific folder),
  And the team can download these to examine detailed findings.
  And the README is updated to show a snippet of the dashboard and explain how to interpret it.

Phase 1 Deliverables: A functioning GitHub Action workflow in the repository that runs on a schedule or push. It should end with a generated **SOC2 compliance table** (and other artifacts) that reflect the current repo's compliance stance. This phase might mostly use local action invocation; we will verify that all tools run correctly on the chosen runner. We will also validate a couple of controls end-to-end (e.g., intentionally introduce a misconfiguration to see it flagged in the table as non-compliant).

Phase 2 (Days 11–20): Integration with GitHub Security Features & Enhanced Automation

Goals: Improve the integration with GitHub's native security and compliance features. This includes uploading scanner results to GitHub code scanning for a richer dashboard, automating the update of control statuses based on real checks, and ensuring the CI fails or alerts on important issues. We also aim to incorporate additional GitHub automation (issues, notifications) to streamline the user experience.

- **Code Scanning Alerts Integration**
  Scenario: Converting scan results to Code Scanning alerts
  Given the JSON reports from Checkov/tfsec and Trivy are generated,
  When the workflow runs an **Upload SARIF** step after scanning,
  Then the results are converted to SARIF (either by using the tools' SARIF output if available, or via a converter script),
  And the `github/codeql-action/upload-sarif@v3` action is invoked to upload the SARIF file [15],
  Then the repository's "Security > Code Scanning Alerts" page shows the findings categorized by the tool (e.g., "Checkov (categories: Infrastructure)" with list of misconfigurations, "Trivy (Dependency)" with CVE findings).
  And each alert can be drilled down to see details and code references. This integration provides a single pane for developers to see and manage compliance-related issues on GitHub [16].
  Note: We'll ensure that this step only runs on the default branch or scheduled runs (to avoid flooding PRs with too many alerts).

- **Automated Control Status Evaluation**
  Scenario: Dynamically updating the compliance dashboard based on checks
  Given certain controls map directly to repository settings or scan outcomes (e.g., "MFA enabled for GitHub" or "All code changes reviewed via PR"),
  When the policy generation script runs in this phase,
  Then it calls GitHub's API (via `gh` CLI or `curl`) to check the actual setting (for example, `gh api repos/:owner/:repo/branches/main/protection` for branch protection rules),
  And it parses scan outputs to see if vulnerabilities were found,
  So it can automatically mark a control's Status as **Implemented/OK** or **Non-compliant**.
  For example: If the "Code Review" control expects that all merges to main have an approving review, the script checks if branch protection requires at least 1 review. If true, mark "Code Review" control as *Implemented*, else *Not Implemented* [26] [11]. Another example: a control "Vulnerability Management" could be tied to "no critical vulns open"; the script can see if Trivy found any critical CVEs and mark status accordingly.
  Then the updated `soc2-table.md` truly reflects live data rather than static inputs, making the dashboard a living document of compliance readiness.

- **CI Failure and Notifications**
  Scenario: Enforcing and alerting on critical compliance failures
  Given we want to ensure critical issues are addressed promptly (per SOC2's requirements on timely

remediation [8]),
When a scan finds a **High/Critical** severity issue (e.g., an open S3 bucket or a leaked secret or a critical CVE),
Then the GitHub Action should mark the job as failed (optionally only for production branch scans to not block dev branches unnecessarily),
And it should post a clear **failure summary** in the workflow logs or as a PR comment. For instance, " SOC2 Compliance Check Failed: 1 Critical issue found (Trivy detected log4j CVE-2021-44228). See Security tab or artifacts for details."
And (optionally) an issue can be opened automatically using GitHub CLI to track the remediation of the finding, tagging it as "security" or "compliance".
And the responsible team is notified (could integrate with Slack or email via webhook in future).
This ensures that serious compliance deviations get immediate attention. Minor issues could be logged without failing the pipeline, perhaps configurable by severity threshold.

- **Enhancing User Experience with Templates & Docs**
  Scenario: Improving GitHub repository UX for compliance
  Given the project repository itself should exemplify good practice,
  When we add Issue/PR templates and a contributing guide,
  Then users are guided to provide necessary info. e.g., a **SOC2 Exception** issue template could help record when a control is intentionally not implemented with justification (useful for audits).
  And a **Pull Request template** could include a checklist: "- [ ] I have run the SOC2 compliance action locally and updated documentation if needed" to remind contributors of compliance steps.
  And in the README, include a section for "Roadmap" and "Common Issues" to set user expectations (pointing out, for example, that they need certain tools or how to interpret the Code Scanning alerts).
  These additions don't directly affect functionality but greatly improve adoption and **user satisfaction** by reducing friction (addressing point #9 about user experience).

- **Security Hardenings:**
  In this phase, also review the action's security. For instance, restrict the GitHub token permissions in the workflow (only `contents:read` and `security-events:write` for SARIF upload). Ensure that secrets (if any API keys for third-party tools) are handled via encrypted secrets and not logged. This is in line with SOC2's confidentiality criteria.

By the end of Phase 2, the project will have a robust integration with GitHub's compliance tooling and a more **intelligent dashboard**. A developer should be able to see compliance status in multiple ways: the markdown table in the repo, the Security alerts, and possibly open issues for any gaps. The action will not just collect data but also enforce policies (to an extent) by failing pipelines on critical issues.

Example outcome of Phase 2: A scheduled run of the workflow triggers on Monday morning. It runs all scans, updates the `soc2-table.md` where now CC1.2 "Board Oversight" might still say *Planned* (as it's manual), but CC3.1 "Logical Access Controls" is marked *Implemented* because the script confirmed SSO/MFA is on and no leaked creds were found. One control "CC7.1 – Vulnerability Management" is marked *Non-compliant (⚠Critical vulns detected)* because Trivy found a critical issue. The action fails, and in the Security tab an alert is present for that CVE. The team gets notified and an issue is logged to upgrade the library. Once resolved and scans pass, CC7.1 will flip to *Implemented* and pipeline will go green.

Phase 3 (Days 21–30): Marketplace Release and Future Roadmap Enhancements

Goals: Finalize the project for public use by addressing packaging and compliance with Marketplace requirements. Also, outline additional improvements for beyond 30 days (roadmap) such as support for other frameworks or advanced features. This phase is about polish, compliance with standards, and setting the stage for continuous improvement.

- **GitHub Marketplace Preparation**
  Scenario: Refactoring repository for Marketplace listing
  Given the functionality is in place and tested,
  When we reorganize the repository to publish the action,
  Then the repository will contain a single `action.yml` at the root with proper metadata (name, description, icon, branding color, inputs/outputs). [20]
  And ensure no GitHub Actions workflow files are present in the repository (to meet Marketplace rules) [27] – any CI for this project's own tests should either be removed or moved to a different branch while publishing.
  And the README is updated with a **Marketplace-compatible usage example** (`uses: tbowman01/Soc-2-compliance@v1`) and badges for version, CI status, etc.
  And we create a **release tag** (v1.0.0) and publish the action. After this, the action should appear in GitHub Marketplace under the chosen category (likely "Security" or "Continuous Integration").
  Then users can install it without copying files, which satisfies the requirement of making it easy to consume.

- **Documentation & Example**
  Scenario: Providing examples and templates for new users
  Given a user discovers our action on Marketplace,
  When they read the documentation,
  Then they should find a clear **quick start guide** (perhaps an example repository or a snippet they can copy). For instance, an example `workflow.yml` showing how to run the action on a schedule, and a sample `controls.yml` pre-filled with common SOC2 controls (which they can customize).
  And provide guidance on obtaining the tools (if not using a container action). Possibly, as part of documentation, mention how to set up a self-hosted runner or include a one-line step to install missing binaries on ubuntu runners.
  And include a FAQ addressing likely questions (e.g., *"Do I need SOC2 Type I first?"* — answer that while Type I can be a quick win, going for Type II is recommended and the tool supports continuous monitoring needed for Type II [1]).

- **Roadmap Definition**
  Scenario: Outlining future improvements
  Given we want the community to engage and know the project's direction,
  When we create a **ROADMAP.md** or use GitHub Projects,
  Then we list planned features beyond this 30-day scope, such as:

- *Multi-framework support:* ability to toggle rules for PCI DSS, ISO 27001, etc., possibly by extending controls.yml or integration with other scanning rulesets (e.g., Checkov's profiles or Trivy's compliance templates).

- *Ticketing integration:* like strongDM's **Comply** tool, generate Jira tickets or GitHub issues for controls that need human action, automating the evidence collection throughout the year [28]. This helps with continuous compliance (maybe an integration with external GRC platforms or Slack notifications).
- *Auditor export:* generate a polished PDF report of the controls and status (using a tool like Pandoc to convert the Markdown to PDF, with company branding) – effectively an audit-ready package. (Comply already demonstrates generating static sites/PDFs as an example [29], which we can emulate by providing a command to produce a static site or PDF from the collected data).
- *Secret scanning enhancement:* possibly integrate with GitHub's own secret scanning or use an alternative like Gitleaks to compare results. The goal would be to reduce false positives and provide more context (e.g., automatic revocation of found secrets via API).
- *Performance tweaks:* e.g., caching dependency scan results, skipping unchanged parts of IaC on incremental runs, etc.
- *Community contributions:* encourage contributions for new controls or support for different languages (e.g., if someone wants to add SAST scanning for code, it could be an optional module).

Then mark some of these as future work, so users know the project is active and growing.

- **Polish and QA**
  Scenario: Final testing and security review before release
  Given all features are implemented,
  When we run the workflow on a test repository intentionally configured with known issues (to simulate a "non-compliant" state),
  Then we verify the dashboard and alerts correctly identify those issues.
  And we test on a repository that is "clean" to ensure it reports all green.
  And perform code review for any sensitive operations (ensuring no credentials are logged, etc.).
  And if possible, get feedback from an auditor or compliance expert on the produced evidence to ensure it would satisfy a SOC2 audit (e.g., are the collected artifacts and statuses aligned with what auditors expect to see). This may involve tweaking wording or adding additional context for certain controls.

At the end of Phase 3, the **Soc-2-compliance** action should be ready for public use: it will be listed on GitHub Marketplace, with thorough documentation, a proven set of features, and a clear roadmap. Internally, the team (or open-source community) will have confidence that the tool can greatly assist in SOC 2 Type II preparation by automating significant portions of evidence collection and testing of controls.

Example compliance dashboard (from an open-source tool StrongDM Comply) illustrating control statuses. This example shows a web-based dashboard where each SOC2 control is listed with its status (green check marks for compliant controls, warnings for those in progress or not implemented). Our GitHub Action will produce a similar summary in Markdown and in the GitHub Security tab, ensuring we have both human-readable and developer-focused views of compliance.

Conclusion

By following this phased plan, in about 30 days we will transform the current project into a comprehensive **SOC 2 Type II compliance automation toolkit**. In summary, we will provide an automated GitHub Action that **continuously assesses security controls**, generates a living compliance dashboard, and integrates with GitHub's native security features to surface issues. The focus on SOC 2 Type II means continuous monitoring – our solution will enable organizations to not only prepare for an audit but to maintain compliance as code changes.

We prioritized implementing the core scanning and dashboard first (Phase 1) to get immediate value. Then we integrated with code scanning and tightened the feedback loops for developers (Phase 2), aligning with best practices that **SOC 2 compliance should be embedded in the SDLC** [30] [10]. Finally, we addressed packaging and forward-looking enhancements (Phase 3) to ensure the project's longevity and ease of adoption.

With these changes, the **SOC 2 Compliance Action** will help engineering teams save time on audit preparation and reduce the risk of missing a critical control. It automates what can be automated (infrastructure checks, code scans, evidence gathering) and highlights what remains manual, thus acting as a "compliance co-pilot." As experts note, using automation and continuous monitoring can significantly reduce the time and cost of SOC 2 audits [31] [32] – this project aims to bring those benefits to our organization and the broader community.

Sources:

- Project README and code – description of integrated tools and requirements [23] [22]
- Secureframe Blog – guidance on choosing SOC 2 Type II for long-term compliance [1] [2]
- Checkov Use Cases – using Checkov for compliance with standards like SOC2 [5]
- Trivy Documentation – capabilities of Trivy in scanning for vulns/misconfigs [7]
- Jit Comparison – TruffleHog for secret scanning in CI [9]
- Scytale (Tech Talk) – embedding security in SDLC for SOC2, key GitHub controls [10] [16]
- GitHub Docs – uploading SARIF to show third-party scan results in Security tab [14] [15]
- GitHub Docs – requirements for publishing an action on Marketplace [20]
- strongDM Comply – example of policy docs and compliance automation features [19] [33]

---

[1] [2] SOC 2 Type 1 vs Type 2: What's the Difference? | Secureframe
https://secureframe.com/hub/soc-2/type-1-vs-type-2

[3] [4] [25] raw.githubusercontent.com
https://raw.githubusercontent.com/tbowman01/Soc-2-compliance/main/soc2-table.md

[5] [6] 13 Checkov Use Cases for Devops Engineers | by Obafemi | Medium
https://medium.com/@obaff/13-checkov-use-cases-for-devops-engineers-b3b3bb3667af

[7] Using Trivy to Scan Software Artifacts - Chainguard Academy
https://edu.chainguard.dev/chainguard/chainguard-images/staying-secure/working-with-scanners/trivy-tutorial/

[8] [10] [11] [16] [26] [30] Setting Up GitHub for SOC 2 Compliance | Scytale
https://scytale.ai/resources/setting-up-github-for-soc-2-compliance/

[9] TruffleHog vs. Gitleaks: A Detailed Comparison of Secret Scanning ...
https://www.jit.io/resources/appsec-tools/trufflehog-vs-gitleaks-a-detailed-comparison-of-secret-scanning-tools

[12] [13] [22] [23] GitHub - tbowman01/Soc-2-compliance
https://github.com/tbowman01/Soc-2-compliance

14 15 24 Uploading a SARIF file to GitHub - GitHub Docs
https://docs.github.com/en/code-security/code-scanning/integrating-with-code-scanning/uploading-a-sarif-file-to-github

17 18 31 32 SOC 2 Type 1 vs. Type 2: Key differences | Vanta
https://www.vanta.com/collection/soc-2/soc-2-type-1-vs-type-2

19 28 29 33 GitHub - strongdm/comply: Compliance automation framework, focused on SOC2
https://github.com/strongdm/comply

20 21 27 Publishing actions in GitHub Marketplace - GitHub Docs
https://docs.github.com/en/actions/how-tos/create-and-publish-actions/publish-in-github-marketplace