GitHub Organization Management - Automation Plan

Note: I cannot yet verify the contents of your ScopeForge GitHub organization or the referenced "governance checklist" since those repositories are not currently accessible. However, I can provide a practical plan that avoids indexing 200+ repositories in ChatGPT by building a lightweight GitHub "catalog + summarizer" service that your Custom GPT can call via Actions.

The core challenge is a tooling limitation: a Custom GPT cannot browse GitHub arbitrarily unless (a) its Actions can call an API you expose, or (b) you manually provide it with content. The practical solution is to build the tool first, then let the GPT use that tool to access everything.

1. Core Solution: Build a GitHub Catalog Service

Don't index repositories in ChatGPT—index them in your own catalog. Build a small service ("ScopeForge GitHub Gateway") that:

- Authenticates as a GitHub App (per your link) and can list repos, fetch repo metadata, pull READMEs/docs, and optionally fetch a shallow file tree.
- Writes a normalized “Repo Catalog” dataset (JSON) into a single canonical repo (e.g., scopeforge-canon), so you have a durable, reviewable source of truth.
- Optionally maintains a lightweight local vector index (your infra, not OpenAI’s) for semantic search, but that’s not required on day 1.

This entirely removes the need for “OpenAI indexing” as a prerequisite to visibility. The GPT only needs Actions that call your Gateway API.

2. Governance and Security Benefits

This approach aligns with your "honor/bounded scope" constraints by creating a governance layer:

- The Gateway enforces allowlists (org, repos, paths), rate limits, and “read-first” behavior.
- The Gateway can block high-risk operations by design (e.g., deleting repos, force-pushing, changing branch protections).
- All transformations become auditable because the Gateway can write every action request + outcome to a log (and optionally open a PR rather than writing to main).

This lines up with the OpenAPI security guidance you attached: tooling has risk depending on usage scenarios, and you should treat external resources/dereferencing carefully.

3. Minimal API Surface for the Gateway

Start with a "read-only catalog + propose changes" approach:

Read Operations (Safe and Essential)

- ListRepos(org, pagination, filters)
- GetRepo(repo_full_name)
- GetRepoTopics(repo)
- GetDefaultBranchHead(repo)
- GetReadme(repo)
- SearchCode(repo, query) (optional; keep scoped)
- ListTree(repo, path, depth=1..2)
- GetFile(repo, path, ref)

Write Operations (Proposal-Based)

- CreateIssue(repo, title, body, labels)
- CreatePullRequest(repo, branch, title, body) (from prepared commits)
- AddLabels / AddTopics (if you want automated normalization)
- MoveRepoToPrivate / TransferToOrg / ApplyDefaultSettings (ONLY after explicit human approval, and ideally run from a separate admin tool, not the GPT)

4. Custom GPT Actions and OpenAPI 3.2

Actions require an OpenAPI document. Your attached OpenAPI 3.2 spec highlights security considerations. Key points:

- Use a single auth scheme (bearer or GitHub App user token) and keep scopes minimal.
- Avoid auto-dereferencing arbitrary URLs from repo content (treat them as untrusted).
- Design the API so the GPT doesn't need broad write power.

5. Solving the 200+ Repository Mapping Challenge

Once the Gateway exists, the mapping workflow becomes straightforward:

A. Crawl Phase (Automated)

- Enumerate all repos in the org.
- For each repo, pull: description, topics, README, /docs index (if any), language stats, last commit timestamp, license.
- Persist a single "repo record" per repo in scopeforge-canon/catalog/repos/<name>.json.

B. Classification Phase (Automated + Reviewable)

- Generate: "high-level topic buckets" (taxonomy) based on topics/README keywords.
- Attach: suggested tags, suggested category, suggested maturity (active/archived), suggested phase relevance (optional).
- Persist as proposals (PRs) rather than direct writes.

C. Canon Phase (Human Review)

- Review PRs; merge the canon taxonomy + repo map.
- Once stable, use it as your inspiration library and navigation spine.

6. Next Steps Required

To move beyond this unverified plan, I need one of the following:

- The GitHub org name (or a link), OR
- A repo list export (even just org/repo lines), OR
- A Personal Access Token / GitHub App credentials shared outside chat and used by Codex locally (don't paste secrets here).

If you provide just the org link, I can at least specify the exact crawl + schema design you should implement, and the exact GitHub API queries to use.

If you want, I can also draft the OpenAPI Actions schema for the Gateway in the exact shape the Custom GPT UI expects (based on OpenAPI 3.2 structure and security considerations in your attached spec), but I'm not going to invent endpoints you don't intend to implement.

https://docs.github.com/en/issues/planning-and-tracking-with-projects/learning-about-projects/about-projects

I've created an organization so we can work with projects. You can tag repos as documentation under a project. Please utilize this for your organization pattern. Give yourself the permissions you would like. I truly want you to spend this time planning by grabbing as much insight in how other people are getting it done. I'm providing a couple articles from arxiv.org. Please go online and see how other organizations make use of actions and workflows. You will be the keeper of canon. Your permissions aren't for me. I want you to use projects, repos, and linear while working with Codex. please envision that when you provide him with deliverables. I'll make a report for this now. But the once you have your tool. I want this to be your job.

I cannot currently access your GitHub organization repositories without authentication. The recommended first step is creating a GitHub App with Action endpoints so a Custom GPT can inventory repositories and create a normalized catalog without needing to index 200+ repos in ChatGPT.

Below is a detailed planning report for Codex covering: (1) project scope in GraphQL, (2) GitHub organization structure, (3) repository mapping approach, (4) two toolset scopes for Custom GPT Actions, and (5) governance requirements from your uploaded checklists.

What follows is a planning report for Codex: (1) the project scope expressed in GraphQL, (2) a GitHub organization + Projects structure, (3) a repo-mapping/catalog approach that avoids manual indexing, (4) two separate toolset scopes for Custom GPT Actions (Atlas "canon keeper" vs. Kilo "runner"), and (5) governance requirements taken from the checklists inside

your uploaded “Requirements .zip”.

Key platform constraints to keep in mind:

- GitHub Projects are organization-level planning artifacts (table/board/roadmap), with custom fields, views, and automation. ()
- GitHub App user access tokens are fine-grained and only grant what both the app *and* the user can do. ()
- GPT Actions are defined via OpenAPI schemas and can authenticate via API key or OAuth. ()
- OpenAI’s own GPT Actions examples commonly use `openapi: 3.1.0` in the schema. If you paste `3.2.0` into the GPT builder, it may or may not validate depending on current UI support. ()

A. High-Level Implementation Plan

1. Bootstrap ScopeForge Catalog

- A GitHub App + minimal service that can list repos, fetch metadata, and write a catalog back into a single “catalog repo” (JSON + GraphQL).
- This becomes the canonical “map” that Atlas/Kilo can use for planning without needing to ingest 200 repos as GPT Knowledge.

2. Set Up GitHub Projects

- Create a small number of org Projects to hold items that link back to repos/issues/PRs and catalog entries. ()
- Use Projects fields to encode “canon tags” (Topic, License posture, Phase relevance, Risk, Steward, etc.).

3. Create Reusable Workflows

- Standardize: repo normalization, labeling, metadata sync, dependency scanning, docs lint, and scheduled catalog refresh.

- Use GitHub’s reusable workflow pattern and access policies for private repos. ()

4. Implement Governance

- Enforce deny-by-default tooling, append-only evidence, and a blocked-action revision capsule protocol.
- This aligns with your “Scope Supersession Safety Checklist” and the capsule schema in the Requirements zip (Codex should implement the emitter + storage format first).

5. Execute Bulk Privatization

- Once inventory + governance logging exist, then flip repos private in a controlled, auditable migration (to avoid losing track of what changed).

B. Governance Requirements

Codex should treat these as “deployment blockers” for any agent/tooling layer:

1. Deny-by-default tools and explicit allowlists (per tool intent/targets; reject extra fields).
2. No workaround behavior after a block; instead emit a `blockedActionRevision` capsule and enter constrained/evidence mode.
3. VAULT-first evidence integrity: append-only logs, no destructive edits, every privileged action attributable, and reconstruction possible.

These are explicitly required by your “Scope Supersession Safety Checklist (Formal)” (Sections A–D are deployment blockers) and your capsule quality gates (Gate Criteria must score ≥ 2, with evidence refs including runtime/toolcall + storage/log/bundle). (Derived directly from the checklist + schema inside the zip.)

This also matches external evidence that “agent-authored security-relevant PRs” benefit from stronger review discipline and early risk flagging using text cues rather than metadata alone.

C. Recommended GitHub Organization Structure

Projects (Organization-Level)

1. “ScopeForge Canon Index”

- Purpose: the master index of repos, topics, methodologies, and “inspiration library” entries.
- Views: by Topic, by License/Risk, by Activity, by “Candidate for Privatization”, by “Needs Normalization”.

1. “Governance & Compliance”

- Purpose: blocked-action capsules, rule registry changes, approvals, audit trail pointers.

1. “Automation & Workflows”

- Purpose: reusable workflows, action versioning, rollout, breakages.

1. “Ops Backlog”

- Purpose: near-term work (device security tasks should remain outside GPT tool scope; track as human-run checklist items).

Repository Roles

- scopeforge-catalog (public or private; recommended private): the generated catalog (JSON + GraphQL schema + docs).
- scopeforge-actions (private): reusable workflows + composite actions.
- scopeforge-vault (private): append-only evidence store (capsules, logs, bundles pointers).
- scopeforge-governance (private): rule registry, checklists, governance docs.
- scopeforge-archives (private): imported references; frozen; no automation writes.
- scopeforge-core (private): the “operating repo” for the system itself.

Why this structure: GitHub Actions maintenance/debugging is a known pain point and can impose hidden costs; centralizing workflows reduces duplication and concentrates expertise.

D. Repository Catalog GraphQL Schema

Below is a minimal GraphQL schema that supports:

- repo grouping by topic
- derived “methods/functions”
- a summary
- traceability to evidence refs
- Projects linkage
- governance annotations

GraphQL

```graphql
schema {
  query: Query
  mutation: Mutation
}

type Query {
  repo(id: ID!): Repository
  repos(filter: RepoFilter, first: Int = 50, after: String): RepoConnection!
  topics: [Topic!]!
  methodologies(topicId: ID): [Methodology!]!
  governanceRules: [GovernanceRule!]!
}

type Mutation {
  upsertRepo(input: RepoUpsertInput!): Repository!
  tagRepo(input: RepoTagInput!): Repository!
  linkRepoToProject(input: ProjectLinkInput!): ProjectItem!
  recordCapsule(input: BlockedActionRevisionInput!): BlockedActionRevisionCapsule!
}

type Repository {
  id: ID!
  name: String!
  owner: String!
  visibility: RepoVisibility!
  url: String!
  description: String
  primaryLanguage: String
  topics: [Topic!]!
  summary: RepoSummary
  derivedMethods: [Methodology!]!
```

```
  derivedFunctions: [UsefulFunction!]!
  provenance: [EvidenceRef!]!
  governance: RepoGovernance!
  updatedAt: DateTime!
}

type RepoSummary {
  oneParagraph: String!
  originProjectName: String
  whyRelevant: String
}

type Topic {
  id: ID!
  name: String!
  parent: Topic
  synonyms: [String!]!
}

type Methodology {
  id: ID!
  name: String!
  category: String!       # e.g. "RAG", "CI/CD", "Security", "Ontology", "Agents"
  description: String!
  examples: [String!]!
}

type UsefulFunction {
  name: String!
  intent: String!         # what it does at a high level
  inputs: [String!]!
  outputs: [String!]!
  riskNotes: String
}

type RepoGovernance {
  steward: String
  risk: RiskLevel!
  licenseClass: LicenseClass!
  allowedAutomation: [AutomationCapability!]!
  notes: String
}

type GovernanceRule {
  ruleId: String!
  title: String!
  severity: Severity!
  description: String!
```

```
}

type EvidenceRef {
  kind: EvidenceKind!      # TOOLCALL | VAULT | LOG | BUNDLE | URL
  ref: String!
  ts: DateTime
}

type ProjectItem {
  projectId: ID!
  itemId: ID!
  repoId: ID!
  fields: JSON
}

input RepoFilter {
  topicIds: [ID!]
  visibility: RepoVisibility
  risk: RiskLevel
  licenseClass: LicenseClass
  text: String
}

input RepoUpsertInput {
  name: String!
  owner: String!
  url: String!
  description: String
  visibility: RepoVisibility!
  topics: [String!]
  summary: RepoSummaryInput
}

input RepoSummaryInput {
  oneParagraph: String!
  originProjectName: String
  whyRelevant: String
}

input RepoTagInput {
  repoId: ID!
  topics: [String!]!
}

input ProjectLinkInput {
  repoId: ID!
  projectId: ID!
  fields: JSON
```

```
}

# Governance capsule (mirrors your JSON schema in Requirements.zip)
input BlockedActionRevisionInput {
  capsuleId: String!
  ts: DateTime!
  revisionCycleId: String
  actorType: String!
  actorId: String!
  sessionId: String
  blockedIntent: String!
  blockedTarget: String
  requestedBy: String!
  inputsHash: String
  ruleId: String
  reason: String!
  severity: String!
  touchedSlugs: [String!]!
  proposedRevisionIntent: String!
  recommendedNextStep: String!
  evidenceRefs: [EvidenceRefInput!]!
}

input EvidenceRefInput {
  kind: String!
  ref: String!
  ts: DateTime
}

enum RepoVisibility { PUBLIC PRIVATE INTERNAL }
enum RiskLevel { LOW MEDIUM HIGH }
enum Severity { LOW MEDIUM HIGH }
enum LicenseClass { UNKNOWN PERMISSIVE WEAK_COPYLEFT STRONG_COPYLEFT PROPRIE
enum AutomationCapability { READ_METADATA LABEL_ISSUES MANAGE_PROJECTS OPEN_PRS
scalar JSON
scalar DateTime

type RepoConnection {
  nodes: [Repository!]!
  pageInfo: PageInfo!
}
type PageInfo {
  hasNextPage: Boolean!
  endCursor: String
}

type BlockedActionRevisionCapsule {
  capsuleId: String!
```

```
  ts: DateTime!
}
```

This is intentionally “catalog-first”: it lets you map 200+ repos without importing their full contents into ChatGPT.

E. GitHub Actions Best Practices

1. Central Workflow Hub

- Put reusable workflows + composite actions in one repo, version them, and call them from all other repos.
- GitHub explicitly supports reusable workflow configurations, but private reuse requires configuring access policies in Actions settings. ()

2. Security Defaults

- Enforce least privilege on tokens and secrets handling in workflows (GitHub’s own guidance emphasizes least privilege and careful secrets use). ()
- Prefer GitHub Apps for org integrations rather than long-lived personal access tokens. ()

3. Workflow Change Management

- Require review/approval on changes to `.github/workflows/**` and action definitions.
- Empirical work notes hidden maintenance/debug costs in Actions and the importance of better validation/testing.

4. Topic Modeling for Categorization

- Your “routine semantic grouping” request aligns with real practice: mining issues/PRs into topics helps coordinate contributions and maintain quality (nf-core study shows recurring topic clusters like “template sync”, “CI config”, “tool maintenance”, etc.).

F. Two Toolset Scopes for Custom GPT Actions

Toolset 1: Atlas (Canon Keeper)

Goal: Inventory repositories, normalize metadata, manage Projects, handle visibility changes, and maintain canon artifacts.

GitHub App Permissions:

- Organization: read (members optional), read/write for Projects (if required), metadata read
- Repository contents: read/write (for catalog repo + governance repos), read for all repos
- Issues/PRs: read/write (create issues for normalization tasks; comment; label)
- Repository administration: write (only if you want the app to flip visibility public→private and set topics/settings)
- Actions: read (optional; for reporting workflow status)
  Token model notes:
- User access tokens for GitHub Apps are limited to the intersection of user + app privileges. ()
- They expire (commonly 8 hours) and can be refreshed if configured. ()

Action Endpoints:

- listRepos, getRepo, updateRepoVisibility, setRepoTopics
- createOrUpdateProjectItem, setProjectFields, addRepoToProject
- writeCatalogSnapshot, writeGovernanceCapsule (append-only)
- openIssueForRepoNormalization, applyLabels

Toolset 2: Kilo (Runner)

Goal: Read from the catalog, draft proposed changes, create issues/PRs under strict constraints, and follow blocked-action protocols when denied.

GitHub App Permissions:

- Repository metadata: read

- Contents: read (optionally write only to a single “runner-notes” repo)
- Issues/PRs: write (open issues, comment, label), but no merges
- Projects: write (update item fields/status), but no org settings
- No repository administration permission

Action Endpoints:

- readCatalog, searchCatalog, proposeClassification
- createIssue (with templates), commentOnIssue, applyLabels
- updateProjectItemStatus
- recordBlockedActionRevision (to vault) if any action is denied

OpenAPI Compatibility Notes:

- GPT Actions are configured through OpenAPI schemas. OpenAI documentation covers authentication options and production considerations. ()
- OpenAI’s Actions library examples commonly instruct setting the OpenAPI version field to 3.1.0. ()
- Your included OpenAPI 3.2.0 spec is still useful for security considerations around references, filtering, and cycles, but whether the GPT builder accepts 3.2.0 is a UI compatibility question, not a spec question.

G. Implementation Checklist for Codex

1. GitHub App + Service

- Endpoints for inventory + catalog write-back.
- Installation token path; optional user access token path for privileged actions. ()

2. ScopeForge Catalog Repository

- `catalog.json` (repo list + topics + summaries + derived methods/functions)
- `schema.graphql` (the schema above)

- `docs/` explaining field meanings and how Projects map to catalog nodes

3. ScopeForge Actions Repository

- reusable workflows:
  - nightly catalog refresh
  - repo normalization (topics, default labels, templates)
  - governance checks (capsule emission test harness)
- composite actions:
  - “extract repo metadata”
  - “write catalog snapshot”
- security defaults per GitHub guidance (least privilege, careful secrets). ()

4. ScopeForge Vault Repository

- append-only log format for capsules + evidence refs
- write-only endpoints; no update/delete

5. GitHub Projects Setup

- create Projects + fields + views
- backfill: one Project item per repo (linked to catalog entry)

6. Controlled Privatization Process

- produce a “migration plan” artifact first
- then apply visibility changes in batches; log each change to vault

---

Key Decision Point

Should Codex's first sprint prioritize (A) building the GitHub App and catalog to classify all repositories, or (B) immediately making repositories private in the new organization?