ChatGPT

REPO-INTEL Setup and Execution Guide (Prime Technical Services)

This guide provides a beginner-friendly, step-by-step walkthrough to set up and use the **REPO-INTEL** business intelligence engine from scratch. We will go from creating the project directory and configuration files, through running the harvesting scripts, to integrating the results with ChatGPT-5 (Agent Mode) and Sourcegraph for deep analysis. By the end, you'll have a fully updated REPO-INTEL system that indexes relevant GitHub repositories (including recent configuration updates for keywords, allowlists, and sources) and know how to leverage it for business development (BD) insights.

Prerequisites: You should have Python 3.x installed, Docker Desktop running, and access to a GitHub account (for source code and tokens). Basic GitHub and command-line familiarity is assumed.

Step 1: Set Up the Project Directory Structure

First, create a new directory for the REPO-INTEL project. In this guide, we'll use `repo-intel` as the folder name (you can create it via Windows Explorer or with `mkdir repo-intel` in a terminal). Inside this directory, create the subfolders as shown below:

```
repo-intel/
├── config/              # Configuration files (keywords, allowlists,
sources)
├── scripts/             # Harvesting scripts (to collect and process data)
├── pipeline/            # Query files for advanced searches (GraphQL,
Sourcegraph)
├── outputs/             # Harvested outputs (catalog, repo cards,
embeddings)
└── .github/workflows/   # (Optional) CI workflows for automation (e.g.,
nightly updates)
```

Within `repo-intel/config/`, we will place three YAML files: `keywords.yml`, `allowlists.yml`, and `sourcelists.yml`. These contain the latest configuration for our BD Intelligence Engine harvest. The `scripts/` folder will house Python scripts to fetch data and build the catalog (we'll add these shortly). The structure above follows the recommended template [1] [2].

> **Note:** If you plan to use Git for version control (highly recommended), you can `git init` in the `repo-intel` directory now, or you can initialize a GitHub repo and clone it. We will cover pushing to GitHub in a later step.

Step 2: Create a Bootstrapping Script

Next, create a simple bootstrap script to set up your Python environment and dependencies. This will help ensure all required packages are installed in one go. Open a text editor and save the following as `bootstrap.sh` inside the `repo-intel` directory:

```
#!/bin/bash
# bootstrap.sh: Setup Python virtual environment and install required packages

# 1. Create a Python virtual environment in the .venv folder
python3 -m venv .venv

# 2. Activate the virtual environment
source .venv/bin/activate

# 3. Upgrade pip (optional, to get the latest package installer)
python -m pip install --upgrade pip

# 4. Install the required Python packages
pip install -r requirements.txt

echo "Environment setup complete. Virtual env is active. Ready to run harvest
scripts."
```

This script will create a virtual environment (to isolate our project's Python packages), activate it, and install all necessary libraries. We will define our `requirements.txt` in a moment (it lists the Python packages needed for REPO-INTEL). The use of a virtual environment ensures the project's dependencies do not conflict with other Python projects on your machine.

After saving `bootstrap.sh`, you might need to give it execute permission (on macOS/Linux) by running `chmod +x bootstrap.sh` in the terminal.

Step 3: Define Project Requirements

Create a `requirements.txt` file in the `repo-intel` directory with the Python libraries needed for the harvest scripts. You can include the following (one per line):

```
PyYAML       # for reading YAML config files
requests     # for making HTTP requests (GitHub API calls)
PyGithub     # optional: GitHub API client (for GraphQL queries)
pandas       # for CSV generation and data handling
openai       # for embeddings using OpenAI API (if using OpenAI for vector
embeddings)
```

```
chromadb     # for vector database storage of embeddings (if using Chroma DB)
sentence-transformers   # for local embedding model (if not using OpenAI API)
```

Feel free to adjust based on your preferences (for example, you might use only one of `openai` or `sentence-transformers` depending on your embedding approach). With the requirements file in place, the bootstrap script will know what to install.

Step 4: Run the Bootstrapping Script to Set Up the Environment

Now run the bootstrap script to set everything up. In a terminal, navigate to the `repo-intel` directory and execute:

```
./bootstrap.sh
```

This will create and activate the virtual environment (`.venv`) and install all required packages. After this completes, you should see a message like "Environment setup complete..." and your shell prompt might be prefixed with `(.venv)` indicating the virtual environment is active.

> **Windows Users:** On Windows, you might create a `bootstrap.bat` or run the commands manually, since the above `source` command is for UNIX-like shells. To activate the venv on Windows, use `repo-intel\.venv\Scripts\activate.bat` in Command Prompt or `activate.ps1` in PowerShell.

Keep the virtual environment activated for the coming steps (you'll see `(.venv)` in your prompt). From now on, all python commands refer to the Python inside this environment.

Step 5: Update Configuration Files (keywords, allowlists, sourcelists)

With the structure ready, we will configure what our intelligence engine will harvest. Prime Technical Services has refined a set of **keywords**, an **allowlist** of GitHub sources, and a list of external **sources** to crawl, based on recent BD focus areas. We will create three files under `repo-intel/config/` with this updated configuration. Copy the contents below into the respective files:

`config/keywords.yml` – *Topics and keywords to search on GitHub.* This list covers DoD programs, compliance frameworks, DevSecOps tools, and other terms relevant to our business domain.

```yaml
# config/keywords.yml
# List of topical keywords and phrases for harvesting relevant GitHub repos
- "DoD"
- "DLA"
- "RMF"
- "STIG"
```

```
- "DISA"
- "Zero Trust"
- "DevSecOps"
- "CMMC"
- "SCAP"
- "FedRAMP"
- "ServiceNow"
- "JIRA"
- "MBSE"
- "Digital Engineering"
- "SysML"
- "ICBM"
- "Sentinel"
```

> These keywords reflect areas of interest like Department of Defense (DoD) projects, Risk Management Framework (RMF) and Security Technical Implementation Guides (STIG) compliance, Zero Trust security, DevSecOps pipelines, certification standards (CMMC, FedRAMP), specific tools (ServiceNow, JIRA), and program-specific terms (MBSE – Model-Based Systems Engineering, digital engineering, the Sentinel ICBM program, etc.). They were chosen to align with high-need skill areas and technologies in our target contracts [3]. You can modify or add to this list as your focus evolves.

`config/allowlists.yml` – *Which GitHub orgs/repos to include.* This file restricts the harvest to certain known organizations or repositories, ensuring we focus on quality sources. (It can also be left broad, but an allowlist ensures relevance.)

```
# config/allowlists.yml
# Organizations and specific repositories to prioritize/include in search
orgs:
  - "DISA"             # Defense Information Systems Agency (STIG releases,
tools)
  - "ansible-lockdown" # Community that provides Ansible STIG role
implementations
  - "ComplianceAsCode" # RedHat's SCAP Security Guide content (STIG profiles
etc.)
  - "OpenRMF"          # (If a specific org exists for OpenRMF project; if not,
see repo below)
  - "mitre"            # MITRE (for any open-source cyber tools, e.g., STIG
viewer, Heimdall)
  - "dod-ccpo"         # DoD Central Cloud (if exists, e.g., Platform One
repositories)
repos:
  - "Cingulara/openrmf-docs"    # OpenRMF (Risk Management Framework)
documentation and tools
  - "vmware/dod-compliance-and-automation"  # DoD STIG automation scripts
(example by VMware)
```

```
  - "ansible-lockdown/RHEL8-STIG"  # Example STIG compliance automation repo
  - "OpenSCAP/openscap"          # OpenSCAP security automation toolkit
```

> In the allowlist, we included organizations known for DoD or compliance-related open source work (DISA, ansible-lockdown, etc.) [4]. We also list specific repositories like **OpenRMF** (an open-source RMF tool) and **VMware's DoD automation** scripts as must-include sources. This ensures our harvest captures these even if they don't surface via keyword search alone. You can add or remove entries here depending on what sources you trust or care about. If you leave the allowlist empty, the harvest will consider all repos returned by the keyword search (which could include irrelevant ones).

`config/sourcelists.yml` – *External URLs to crawl for leads.* Here we list web pages (like "Awesome" lists or documentation pages) that the harvest pipeline might crawl to find additional repository links or context.

```
# config/sourcelists.yml
# Web pages (Awesome lists, documentation, etc.) to crawl for repo links or info
- "https://github.com/DevSecOps/awesome-devsecops"
# Curated list of DevSecOps tools and resources
- "https://public.cyber.mil/stigs/"                # DISA STIG official
downloads (for context, non-GitHub)
- "https://github.com/jakelikescloud/awesome-nist"   # (Example) Awesome list
for NIST compliance and tools
```

> The sourcelist can include any page that contains links to projects or info relevant to your search. In our example, we added a popular **Awesome DevSecOps** list and the **DISA STIG** site. The harvesting scripts can be extended to parse these pages for additional repository URLs beyond what the GitHub API returns. This is optional but can enrich your catalog with important projects that are referenced in such lists.

Once these three files are created and saved, double-check that the formatting is correct (YAML is sensitive to indentation and dashes). They should be easy to copy-paste as provided above.

Step 6: Securely Set Your GitHub Token (GH_TOKEN)

To allow our scripts to query the GitHub API (in particular, the GraphQL API for searching repositories), we need a GitHub Personal Access Token. This token will be used by the scripts to authenticate and must be kept secure.

Creating a GH_TOKEN: Go to your GitHub account settings -> **Developer Settings** -> **Personal access tokens** (classic) -> **Generate new token**. Create a token with **repo** scope and **read:org** scope (these scopes allow reading public repo info and organization info, which is sufficient for our needs) [5]. Name it something like "Repo-Intel Token". Once generated, copy the token string (a long string of letters/numbers).

Storing the token: We will supply this token to the scripts via an environment variable named `GH_TOKEN`. **Do not hardcode your token in any file** (to avoid accidental leaks). Instead, set it in your shell before running the harvest. For example, in the terminal run:

```
export GH_TOKEN=<your_token_value_here>
```

Replace `<your_token_value_here>` with the token string you copied. This will set `GH_TOKEN` for the duration of your session. On Windows, use `set GH_TOKEN=...` in Command Prompt or `$Env:GH_TOKEN="..."` in PowerShell.

> **Security Tip:** Treat this token like a password. Do not commit it to Git. If using source control, add any file that might contain secrets to `.gitignore`. In our case, we are only using an environment variable, which won't be saved in the repo. If you share this project, others should use their own token.

Now we're all set to start harvesting data from GitHub.

Step 7: Run Harvest Script 1 – `harvest_github.py`

The first script will query GitHub for repositories relevant to our topics. Let's create the script file first and then run it.

Create `scripts/harvest_github.py`: This script uses GitHub's GraphQL API to search for repositories matching our keywords and optional allowlists. Here's a simplified content (the actual implementation may vary, but key points are illustrated):

```
# scripts/harvest_github.py
import os, sys, requests, yaml, math

# Load config files
keywords = yaml.safe_load(open(sys.argv[1]))  # config/keywords.yml
allowlists = yaml.safe_load(open("config/allowlists.yml"))
GH_TOKEN = os.environ.get("GH_TOKEN")

if not GH_TOKEN:
    sys.exit("GH_TOKEN environment variable not set. Exiting.")

# Construct GraphQL query
search_terms = keywords  # expecting a list of keywords
# Example: combine keywords into a single search query
query_string = " ".join([f"\"{term}\"" for term in search_terms])
graphql_query = {
    "query": f"""
    query {{
```

```
      search(type: REPOSITORY, query: "{query_string} stars:>5", first: 100) {{
        nodes {{
          ... on Repository {{
            nameWithOwner
            description
            url
            stargazerCount
            updatedAt
            primaryLanguage {{ name }}
            repositoryTopics(first:10) {{ nodes {{ topic {{ name }} }} }}
            licenseInfo {{ spdxId }}
          }}
        }}
      }}
    }}
    """
}

# Call GitHub GraphQL API
headers = {"Authorization": f"Bearer {GH_TOKEN}"}
response = requests.post("https://api.github.com/graphql", json=graphql_query,
headers=headers)
data = response.json()

# TODO: Filter results based on allowlists (if provided)
# ...

# Save output to JSON file
out_file = sys.argv[2] if len(sys.argv) > 2 else "outputs/raw.json"
with open(out_file, "w") as f:
    yaml.safe_dump(data, f)
print(f"Saved raw repo data to {out_file}")
```

(The above is a conceptual snippet for illustration. The actual script might handle pagination, allowlist filtering, etc.)

After creating the script, run it as follows (from the `repo-intel` directory):

```
python scripts/harvest_github.py config/keywords.yml outputs/raw.json
```

This will execute the GraphQL search for our keywords. We've limited it to repositories with >5 stars in the example query for quality, and the search is broad across repository name, description, and topics. The script will save the results to `outputs/raw.json`.

What this script does: It harvests repository metadata from GitHub – essentially collecting a list of repos that match our DoD and tech topics [6]. Specifically, it gathers for each repo: the name (with owner),

description, URL, star count, last update timestamp, primary language, top topics, and license info. We now have an initial *unranked* catalog of candidate repositories in the `raw.json` file.

> You can open `outputs/raw.json` in a text editor or JSON viewer to inspect the raw data. It will contain a JSON structure with an array of repository nodes. This raw list might contain some repositories that are not relevant; that's okay, as subsequent steps will refine and rank them. If your allowlist is configured, you could also filter the `data` to only include repos from those orgs or specifically listed repos.

Step 8: Run Harvest Script 2 – `pull_readmes.py`

Now that we have a list of repositories, the next step is to pull down some content from each repo (like the README and any docs) for analysis. Create the `scripts/pull_readmes.py` script as below:

```
# scripts/pull_readmes.py
import os, sys, requests, json

GH_TOKEN = os.environ.get("GH_TOKEN")
if not GH_TOKEN:
    sys.exit("GH_TOKEN not set. Exiting.")
headers = {"Authorization": f"Bearer {GH_TOKEN}"}

# Load the raw repo list from previous step
raw_data = json.load(open(sys.argv[1]))
repos = raw_data["data"]["search"]["nodes"]

output_dir = sys.argv[2] if len(sys.argv) > 2 else "outputs/readmes"
os.makedirs(output_dir, exist_ok=True)

for repo in repos:
    full_name = repo["nameWithOwner"]  # e.g., "owner/name"
    # Make an API request to get the README content via GitHub REST API
    readme_url = f"https://api.github.com/repos/{full_name}/readme"
    resp = requests.get(readme_url, headers=headers)
    if resp.status_code == 200:
        content = resp.json().get("content", "")
        # The content is base64 encoded per GitHub API; decode it
        import base64
        readme_text = base64.b64decode(content).decode('utf-8', errors='ignore')
    else:
        readme_text = "# README not found or inaccessible\n"
    # Save to file named after the repo
    repo_dir = os.path.join(output_dir, full_name.replace("/", "__"))
    os.makedirs(repo_dir, exist_ok=True)
    with open(os.path.join(repo_dir, "README.md"), "w", encoding="utf-8") as f:
```

```
        f.write(readme_text)
    print(f"Saved README for {full_name}")
```

(This script fetches each repository's README via the GitHub REST API. It stores each README in a subfolder under `outputs/readmes/`*, with folder names like* `owner__repo` *to avoid slashes in file names. In a more advanced setup, you might clone the repo shallowly or fetch additional files (like* `LICENSE` *or anything in a* `docs/` *directory).)*

Run the script:

```
python scripts/pull_readmes.py outputs/raw.json outputs/readmes/
```

This will iterate through the list of repos from `raw.json`, retrieve each README, and save them under `outputs/readmes/`. Now we have the actual content from each repository (or at least the main documentation) stored locally.

What this script does: It collects textual content (documentation) from each repository in our list. This is important because the next step will analyze the content for certain keywords and attributes (like presence of security terms or the length of docs) to help score the repositories. Essentially, we're enriching our dataset with details from inside the repos (READMEs, etc.), not just the metadata.

If a repository had extensive documentation (like multiple markdown files), you could extend the script to fetch those as well (e.g., via the GitHub API for contents or by doing a shallow `git clone`). For our use-case, grabbing the README and perhaps a license is sufficient.

Step 9: Run Harvest Script 3 – `rank_repos.py`

Now we have raw data and content for each repo. The ranking script will assign a score to each repository based on various factors to determine which ones are most relevant and valuable for us. Create `scripts/rank_repos.py`:

```
# scripts/rank_repos.py
import sys, json, math

# Load raw data from earlier steps
raw_data = json.load(open(sys.argv[1]))
repos = raw_data["data"]["search"]["nodes"]

# Define scoring weights (you can tweak these)
W_STARS = 1.0
W_FRESHNESS = 1.0
W_DOCS = 1.0
W_LICENSE = 0.5
W_SECURITY = 1.5
```

```
scores = {}
for repo in repos:
    name = repo["nameWithOwner"]
    stars = repo.get("stargazerCount", 0)
    # Freshness: let's use months since last update as a penalty
    import datetime
    updated = datetime.datetime.fromisoformat(repo["updatedAt"].replace("Z",""))
    months_since_update = (datetime.datetime.utcnow() - updated).days / 30
    freshness_score = max(0, 1 - (months_since_update / 12))  # 1 if updated
within last month, down to ~0 after a year

    # Documentation depth: e.g., length of README (we saved it earlier)
    readme_path = f"outputs/readmes/{name.replace('/', '__')}/README.md"
    try:
        text = open(readme_path, encoding="utf-8").read()
        docs_score = min(1.0, len(text) / 5000.0)  # crude: 1.0 if README >5000
chars, smaller if shorter
    except FileNotFoundError:
        docs_score = 0.0

    # License friendliness: prefer MIT or Apache (permissive licenses)
    license = repo.get("licenseInfo", {}).get("spdxId") or ""
    if license in ["MIT", "Apache-2.0"]:
        license_score = 1.0
    elif license:
        license_score = 0.5  # some other license
    else:
        license_score = 0.0  # no license info

    # Security posture keywords: check if README contains certain terms
    readme_text = text.lower() if 'text' in locals() else ""
    security_keywords = ["stig", "rmf", "scap", "800-53", "zero trust", "nist"]
    security_score = 0.0
    for term in security_keywords:
        if term in readme_text:
            security_score += 1
    security_score = min(1.0, security_score / len(security_keywords))  #
normalize 0 to 1

    # Calculate weighted score
    score = (W_STARS * math.log1p(stars)) \
          + (W_FRESHNESS * freshness_score * 5) \
          + (W_DOCS * docs_score * 5) \
          + (W_LICENSE * license_score * 5) \
          + (W_SECURITY * security_score * 5)
    scores[name] = round(score, 2)
```

```
# Save scores to outputs/scores.json
out_file = sys.argv[2] if len(sys.argv) > 2 else "outputs/scores.json"
json.dump(scores, open(out_file, "w"), indent=2)
print(f"Saved scores for {len(scores)} repos to {out_file}")
```

Run the ranking:

```
python scripts/rank_repos.py outputs/raw.json outputs/scores.json
```

What this script does: It reads in our list of repositories (`raw.json`) and computes a composite score for each repository. The example scoring logic above considers:

- **Stars** (more stars -> higher score, using a log scale for reasonable weight),
- **Recency** (recently updated repos score higher, with a decay if a repo hasn't been updated in a long time),
- **Documentation depth** (repos with longer or more thorough README/docs score higher),
- **License** (repos with a permissive license like MIT/Apache get a boost, since they can be more easily used in our projects),
- **Security keywords in content** (if the README contains terms like STIG, RMF, NIST 800-53, etc., it indicates the project is relevant to cybersecurity compliance, so boost those).

These criteria reflect the desired attributes: popular, active projects with good docs, friendly licensing, and relevant security content [7]. The weighting (W_SECURITY, etc.) can be tuned. In our weights, we gave an extra boost to "security posture keywords" because finding projects that explicitly mention STIG, RMF, etc., is particularly valuable for our use case.

After running, you'll have `outputs/scores.json` containing each repo and its score. For example:

```
{
  "ansible-lockdown/RHEL8-STIG": 8.7,
  "ComplianceAsCode/content": 9.5,
  "Cingulara/openrmf-docs": 7.2,
  "...": ...
}
```

You can inspect this file to see how repos ranked. Higher is better.

Step 10: Run Harvest Script 4 – `build_catalog.py`

Now we'll generate the final catalog of repositories in a human-readable format. This script will combine the raw data and scores to produce a **catalog markdown file** (and a CSV file) summarizing each repo. Create `scripts/build_catalog.py`:

```
# scripts/build_catalog.py
import sys, json, csv
from pathlib import Path

scores = json.load(open(sys.argv[1]))
raw_data = json.load(open("outputs/raw.json"))  # you could pass this as arg too
repos = raw_data["data"]["search"]["nodes"]

# Open catalog.md for writing
catalog_md = open(sys.argv[2], "w", encoding="utf-8")
# Write header for markdown
catalog_md.write("# REPO-INTEL Catalog\n\n")
catalog_md.write("| Repository (Name/Owner) | Stars | Last Update | Primary
Language | License | Score | Description |\n")
catalog_md.write("|---|---:|---:|---|---|---:|---|\n")

# Prepare CSV output
csv_file = open(sys.argv[3], "w", newline='', encoding="utf-8")
csv_writer = csv.writer(csv_file)
csv_writer.writerow(["Repository", "Stars", "UpdatedAt", "PrimaryLanguage",
"License", "Score", "Description"])

for repo in repos:
    name = repo["nameWithOwner"]
    stars = repo.get("stargazerCount", 0)
    updated = repo["updatedAt"][:10]  # date portion
    lang = repo.get("primaryLanguage", {}).get("name", "N/A")
    license = repo.get("licenseInfo", {}).get("spdxId", "None")
    score = scores.get(name, 0)
    desc = repo.get("description", "") or ""
    # Write to markdown
    catalog_md.write(f"| [{name}]({repo['url']}) | {stars} | {updated} | {lang}
| {license} | **{score}** | {desc} |\n")
    # Write to CSV
    csv_writer.writerow([name, stars, updated, lang, license, score, desc])

catalog_md.close()
csv_file.close()
print(f"Catalog generated: {sys.argv[2]} and {sys.argv[3]}")
```

Run the build step:

```
python scripts/build_catalog.py outputs/scores.json outputs/catalog.md outputs/
catalog.csv
```

After this, open the file `outputs/catalog.md` in a Markdown viewer or text editor. You should see a nicely formatted table listing each repository, with columns for stars, last update, language, license, score, and description, and with the repository name linking to its GitHub page. The CSV (`catalog.csv`) contains the same information which you can open in Excel or any spreadsheet tool for sorting/filtering.

What this script does: It compiles our findings into a **catalog** – a one-stop reference of the top repositories discovered. The catalog is essentially a "living library" of open-source projects relevant to our BD and engineering work [6] [8]. Each repo's score is included (so you can sort by it or quickly see which are top-ranked). The Markdown format allows easy browsing (and can be opened by ChatGPT's agent as well), and the CSV format allows analysts to do further sorting or integration into other systems.

You have now successfully harvested and cataloged the repositories!

At this stage, the `outputs/` directory should contain: - `catalog.md` – the human-readable catalog of repos. - `catalog.csv` – same data in CSV form. - `repo_cards/` – (Optional) if the script was extended to produce individual markdown files per repo (one-pagers with more detail). Our simple script did not generate these explicitly, but you can create them if needed by writing one markdown file per repository with extended info (e.g., include a snippet of the README or highlights). - `embeddings/` – (Optional, next step) any vector embeddings for semantic search.

Before we move on to using this data, let's finalize the optional embedding step.

Step 11: (Optional) Run Harvest Script 5 – `index_embeddings.py`

This step is optional but powerful. It involves converting the text we gathered (READMEs, etc.) into vector embeddings and storing them, enabling semantic search and Q&A over the repository content. If you want to ask natural language questions like "Which repos automate STIG checks with Ansible?" and get relevant answers, embeddings are the way to go.

Create `scripts/index_embeddings.py` (or you could integrate this into the build script). A simple approach is to use OpenAI's text-embedding API or a local model to embed each repo's README content.

For example, using OpenAI API (requires an OpenAI API key as environment variable `OPENAI_API_KEY`):

```
# scripts/index_embeddings.py
import os, sys, json
import tiktoken, openai

openai.api_key = os.environ.get("OPENAI_API_KEY")
if not openai.api_key:
    sys.exit("OPENAI_API_KEY env var required for embeddings.")

input_dir = sys.argv[1] if len(sys.argv) > 1 else "outputs/readmes"
output_dir = sys.argv[2] if len(sys.argv) > 2 else "outputs/embeddings"
os.makedirs(output_dir, exist_ok=True)
```

```
embeddings_index = {}

for root, dirs, files in os.walk(input_dir):
    for file in files:
        if file.lower().endswith(".md"):
            filepath = os.path.join(root, file)
            repo_id = filepath.split("outputs/readmes/")[-1]  # e.g.,
owner__repo/README.md
            text = open(filepath, encoding="utf-8").read()
            # optional: truncate text to max tokens for embedding model
            text = text[:8000]
            # call OpenAI embedding API (using text-embedding-ada-002)
            response = openai.Embedding.create(input=text, model="text-
embedding-ada-002")
            embedding = response['data'][0]['embedding']
            embeddings_index[repo_id] = embedding

# Save embeddings to JSON (could be large, consider binary storage in practice)
with open(os.path.join(output_dir, "embeddings.json"), "w") as f:
    json.dump(embeddings_index, f)
print(f"Saved embeddings for {len(embeddings_index)} docs to {output_dir}")
```

Alternatively, you can use a local embedding model via the `sentence-transformers` library to avoid external API calls (this might be slower and requires downloading a model, but avoids needing an API key).

Run the embeddings script (if using it):

```
export OPENAI_API_KEY=<your_openai_key_here>   # if using OpenAI API
python scripts/index_embeddings.py
```

This will generate an `outputs/embeddings/embeddings.json` file (or a collection of files if using a vector DB like Chroma which stores in its own format). Each entry maps a repo (or document) to an embedding vector. With this in place, you can perform semantic similarity searches: for example, given a query embedding, find the nearest repo embeddings.

What this enables: The ability to ask questions in natural language and retrieve relevant pieces of repository content. For instance, we could ask "Show me STIG automation toolchains with Ansible playbooks and DISA content" – an embeddings-based search could find the README that mentions those terms even if phrased differently [9]. Essentially, it turns our static catalog into a mini knowledge base that can be queried intelligently.

If you do not need this capability initially, you can skip embedding for now. You can always add it later when you want to integrate Q&A or agent-based querying.

Step 12: (Optional) Launch a Q&A Retrieval API Service

With embeddings created, the next optional step is to run a service that can serve these embeddings for queries. One straightforward way is to use OpenAI's **Retrieval Plugin** (from the chatGPT plugins repo) or a simple Flask app that loads the embeddings and on each query finds the top relevant documents.

Using OpenAI Retrieval Plugin: OpenAI has an open-sourced retrieval plugin that can be self-hosted. It essentially provides a REST API where you can upsert documents and then query them with natural language, returning relevant chunks. While the "plugin" aspect was for ChatGPT, it can function as a standalone Q&A API. You can feed it our docs (READMEs) along with the embeddings or let it generate its own.

Given that we already have embeddings, we might implement our own minimal API. But to keep things simple, you could use the approach from OpenAI's retrieval plugin (which supports various vector DBs like Pinecone, etc.). Another lightweight approach: use Chroma's REST interface or LanceDB's server mode.

For illustration, let's say we choose to quickly stand up a simple Flask app:

```
# qna_server.py (Example of a minimal retrieval API)
from flask import Flask, request, jsonify
import numpy as np, json

# Load embeddings index
embeddings = json.load(open("outputs/embeddings/embeddings.json"))
# Precompute normalized vectors for cosine similarity
embeddings = {k: np.array(v, dtype='float32')/np.linalg.norm(v) for k,v in
embeddings.items()}

app = Flask(__name__)

@app.route("/query", methods=["POST"])
def query():
    data = request.get_json()
    query_text = data.get("query")
    if not query_text:
        return jsonify({"error": "No query provided"}), 400
    # Embed the query using same method as our index (here OpenAI API)
    query_vec = openai.Embedding.create(input=query_text, model="text-embedding-
ada-002")['data'][0]['embedding']
    q = np.array(query_vec, dtype='float32')/np.linalg.norm(query_vec)
    # Compute similarity with each doc
    sims = {}
    for repo_id, emb in embeddings.items():
        sims[repo_id] = float(np.dot(q, emb))
    # Get top 3
    top3 = sorted(sims.items(), key=lambda x: x[1], reverse=True)[:3]
```

```
    results = []
    for repo_id, score in top3:
        owner_repo = repo_id.replace("__", "/").replace("/README.md","")
        results.append({
            "repo": owner_repo,
            "score": round(score,3),
            "url": f"https://github.com/{owner_repo}"
        })
    return jsonify({"query": query_text, "results": results})

if __name__ == "__main__":
    app.run(port=8000)
```

Run this server with `python qna_server.py`. Now you have a simple API that, given a query, returns the top matching repositories. In a real scenario, you'd probably also return a snippet of the README or more context, but this shows the concept.

Why launch a Q&A API? This allows integration with ChatGPT or other tools in a retrieval-augmented generation scenario. For example, ChatGPT Agent Mode could call this API (if configured as a tool) to ask questions and get relevant repo info. Or internally, your team could use this API to query the knowledge base programmatically.

If not using a custom API, you can also query the embeddings directly in a notebook or script. The key point is that we have structured data (catalog) and unstructured data (repo content) both accessible now.

(If this feels complex, remember this step is optional. You can skip directly to using the catalog with ChatGPT, which we'll cover next.)

Step 13: Push the Results to GitHub (Monorepo vs Standalone Repo)

Now that you have built everything locally, it's wise to version control and back up your work by pushing it to GitHub. There are two ways you might integrate this into GitHub, depending on your preference or organizational policy:

- **A dedicated repository (standalone)** for REPO-INTEL (e.g., `prime-tech/repo-intel` on GitHub).
- **As part of a monorepo** – if your company maintains a single repository for all BD tools and data, you could add this project under a subdirectory of that monorepo.

We'll briefly cover both scenarios:

☞ **Standalone Repo Approach:**

1. Create a new repository on GitHub (via the GitHub UI) called, for example, `repo-intel`. Do not initialize it with a README (since we already have files).
2. On your local machine, if you haven't yet, do `git init` inside the `repo-intel` directory. Then connect it to the GitHub repo:

```
git remote add origin https://github.com/<your-org-or-username>/repo-intel.git
```

3. Add and commit files:

```
git add .
git commit -m "Initial commit: add REPO-INTEL config and scripts"
git push -u origin main   # push the main (or master) branch to origin
```

4. If everything is set up, you should now see your project on GitHub.
5. (Optional) Add the output files to the repo as well. The `catalog.md` and any generated repo cards can be committed so that the latest catalog is easily viewable on GitHub. For example, commit `outputs/catalog.md` and the `outputs/repo_cards/` folder. You might want to **exclude** large files like the embeddings JSON from the repo (perhaps add `outputs/embeddings/` to a `.gitignore`).

☞ **Monorepo Approach:**

1. If your company has a monorepo (let's call it `prime-tech/all-in-one`), you might want this project to live under a subfolder there (say `bd-tools/repo-intel/`).
2. In that case, instead of creating a new repo, navigate to the monorepo's folder and create the directory `bd-tools/repo-intel/` (or whatever path is appropriate).
3. Move or copy your files into that folder. Then use the regular Git workflow within the monorepo:

```
git add bd-tools/repo-intel
git commit -m "Add REPO-INTEL tool with configs and scripts"
git push   # push to the remote (maybe a feature branch if required)
```

4. Depending on your dev process, open a Pull Request to merge this addition into the main branch of the monorepo, etc.
5. Once merged, the files are in the monorepo. The usage doesn't change, except that now your path context is different (`bd-tools/repo-intel/...`), and the repo is larger.

Keeping the Catalog Updated: You may want to update the catalog over time (e.g., nightly or weekly). One way is to rerun the harvest scripts periodically and commit the new `catalog.md`. You can automate this with a GitHub Action workflow. For instance, a **nightly workflow** (as in `.github/workflows/nightly_catalog.yml`) could run the Python scripts and push changes [10] [11]. An example snippet from our template:

```
# .github/workflows/nightly_catalog.yml
name: Nightly repo-intel refresh
on:
  schedule: [{ cron: "17 3 * * *" }]  # runs every day at 03:17 UTC
  workflow_dispatch: {}
```

```
jobs:
  build-catalog:
    runs-on: ubuntu-latest
    permissions:
      contents: write
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with:
          python-version: "3.11"
      - run: pip install -r requirements.txt
      - env:
          GH_TOKEN: ${{ secrets.GH_TOKEN }}
        run: |
          python scripts/harvest_github.py config/keywords.yml outputs/raw.json
          python scripts/pull_readmes.py outputs/raw.json outputs/readmes/
          python scripts/rank_repos.py outputs/raw.json outputs/scores.json

python scripts/build_catalog.py outputs/scores.json outputs/catalog.md outputs/
catalog.csv
      - name: Commit catalog
        run: |
          git config user.name "repo-bot"
          git config user.email "[email protected]"
          git add outputs/catalog.md outputs/catalog.csv
          git commit -m "nightly catalog refresh" || echo "No changes"
          git push
```

You would need to add your `GH_TOKEN` as a secret in the GitHub repo settings so the Action can use it. This way, every day the catalog updates itself with the latest repo info (new stars, new repos, etc.). The above workflow is just an example; adjust the schedule and details to your needs.

Step 14: Connect REPO-INTEL to ChatGPT-5 (Agent Mode)

One of the major goals of building this REPO-INTEL system is to leverage it with advanced AI (like ChatGPT-5 with Agent Mode) to accelerate research and decision-making. Here's how to connect and use the data we've prepared:

1. Enable the GitHub Connector in ChatGPT (Agent Mode): In the ChatGPT interface, go to **Settings -> Beta Features (or Connectors)** and ensure the **GitHub connector** is enabled. Authorize it if prompted and select the repository (or organization) that contains your REPO-INTEL project [12]. If you used a standalone repo, choose that; if it's in a monorepo, you might select the whole organization or specific repo.

This authorization lets the ChatGPT agent read content from the repo. The agent will be able to open files like `catalog.md` or any of the repo markdowns.

2. (If used) Enable the Google Drive connector: If you want the agent to also have access to documents like the catalog exported to PDF or other BD documents, you can connect Google Drive and place those files there [13]. This step is optional but can enrich the agent's context (for example, you might put the **Competitive Capture Strategy PDF** or other reference docs in Drive for the agent to cite if needed).

3. Using Agent Mode to Query the Catalog: Now, open a ChatGPT session in Agent Mode. You can instruct it to use the GitHub repo as a knowledge source. For example, try prompts like:

- *"Search our repo-intel catalog.md for DISA STIG automation stacks and list the top 10 repositories. For each, open its README and summarize the deployment steps."* – This prompt will cause the agent to load `outputs/catalog.md`, find entries related to "DISA STIG automation", then for each repo it deems top, use the GitHub connector to open that repo's README.md, and give you a summary [14].

- *"Using the Sourcegraph instance, search across public code for* `rmf OR "nist 800-53"` *combined with* `ansible OR terraform`*, and show highly-starred repos updated in the last 6 months that match."* – This prompt directs the agent to perform a cross-repo code search using Sourcegraph (if you set it up, more on that soon) [15].

- *"From our catalog, build a shortlist of repos we should consider for a upcoming proposal requiring STIG compliance automation. Provide a table with repository name, its purpose, maturity (stars/last update), license, and what next steps would be to integrate it."* – The agent can read `catalog.md` and compile such a table [16].

The agent will utilize the data we've created. It can follow the links in `catalog.md` to directly open those GitHub repos and even read code if needed. Essentially, we've given the agent a map of relevant projects, so it doesn't have to search the entire internet blindly – it has a curated library to draw from.

4. Question Answering with Embeddings (Agent Mode Advanced): If you set up the Q&A API in Step 12 or loaded embeddings into a retrieval plugin, you can enable that as well. For instance, if running a local retrieval service on port 8000, you might configure the agent with a custom tool (this usually requires the enterprise version where you can add tools, or you'd integrate via the plugin system). The simpler approach: have the agent rely on the catalog and Sourcegraph, which often suffices.

5. Direct Deep Research Mode: Even outside of Agent Mode, you (as a user) can open the `catalog.md` file in Deep Research (ChatGPT's browsing mode) to manually explore the list. But Agent Mode's advantage is it can automate the lookup and summarization process.

In summary, connecting to ChatGPT-5 Agent Mode transforms REPO-INTEL from a static list into an interactive assistant that **understands your curated repos**. You can ask high-level questions and the agent will use the repo intel to answer or create plans.

For example, if you ask: *"What open-source tools can help implement RMF or STIG controls in a DevSecOps pipeline?"*, the agent will likely consult the `catalog.md` for anything with RMF or STIG (finding perhaps OpenRMF, ComplianceAsCode content, Ansible roles, etc.), and then open those entries to give specifics, citing the repos. This is extremely powerful for quickly generating insights or proposal content that is grounded in real technology options.

(Remember: ensure your agent has access by verifying the connectors. Also, large files like the entire repository code aren't indexed by default – but the agent can open specific files as needed. The `catalog.md` *and any* `repo_cards` *we generated are key entry points for it.)*

Step 15: Layer B – Integrating Sourcegraph for Scalable Code Search

While the catalog and ChatGPT can cover a curated set of repositories (Layer A, your private library [17]), you might want to cast a wider net across open source code (Layer B [18]). **Sourcegraph** is a self-hosted code search engine that can index thousands of repositories and provide instant regex and structural search across them. Here's how to integrate it:

1. Deploy Sourcegraph CE via Docker: The quickest way is using the single-container Docker image. In a terminal on your machine (with Docker running):

```
docker run -d -p 7080:7080 -p 3370:3370 \
    -v ~/.sourcegraph/config:/etc/sourcegraph \
    -v ~/.sourcegraph/data:/var/opt/sourcegraph \
    sourcegraph/server:latest
```

This will start Sourcegraph listening on port 7080 (and 3370 for search API). After a minute, navigate to `http://localhost:7080` in your browser. You'll be prompted to set an admin username/password. Do that, and you'll land on the Sourcegraph web UI [19].

2. Connect Sourcegraph to GitHub: In Sourcegraph, go to **Site Admin > Manage Code Hosts** (or **Add External Service**). Add a **GitHub** connection. Provide your GitHub token (a token with at least repo read access; you can reuse the GH_TOKEN from earlier or generate a new one specifically for Sourcegraph). Then, in the configuration, specify what repositories to sync. You can use **repositoryQuery** entries to automatically search and include repos by criteria [20]. For example, a config might look like:

```
{
  "url": "https://github.com",
  "token": "YOUR_GH_PAT_HERE",
  "repositoryQuery": [
    "org:ansible-lockdown",
    "org:ComplianceAsCode",
    "topic:RMF",
    "topic:STIG",
    "topic:DevSecOps",
    "prime-tech/repo-intel"
  ]
}
```

This tells Sourcegraph to index all repos under `ansible-lockdown`, `ComplianceAsCode`, any repo with a GitHub topic "RMF", "STIG", or "DevSecOps", as well as our own repo-intel (for completeness). You can add

specific repos or orgs, or use topics/keywords as shown. Save this configuration. Sourcegraph will start cloning those repositories in the background. (It may take a while if it's a lot of code – you can check progress under **Site Admin > Repository statuses**.)

3. Running Canned Queries: We included a file `pipeline/sourcegraph_queries.txt` in the repo for interesting search queries. You can open Sourcegraph's search bar and try queries from there. Some examples:

- **Find references to RMF and Terraform:** In Sourcegraph's search, enter a query like:

  ```
  "RMF" "800-53" file:README lang:Markdown
  ```

  This would find any README files mentioning RMF or 800-53. Or use the OR syntax:

  ```
  /RMF|800-53/ (ansible OR terraform)
  ```

  to find any code or text mentioning RMF or 800-53 along with either ansible or terraform (which might surface infrastructure-as-code tools for compliance).

- **Find STIG automation code:** For example:

  ```
  repo:ansible-lockdown/ STIG
  ```

  This will search within all repos under the ansible-lockdown org for the term "STIG". Or more broadly:

  ```
  STIG AND Ansible
  ```

  to find anywhere those two words co-occur.

- **Search by file names or paths:** Sourcegraph can do advanced queries. For instance:

  ```
  file:Dockerfile "STIG"
  ```

  to find if any Dockerfile mentions STIG (perhaps container hardening).

- **Leverage Sourcegraph's regex and structuring:** You can also search by language or symbol. For example, if looking for code related to "Zero Trust" in Python:

  ```
  "Zero Trust" lang:Python
  ```

If interested in MBSE related content, maybe search for "SysML" or specific tool names that might appear in code.

The `pipeline/sourcegraph_queries.txt` can contain a list of such search strings. You can run them one by one in the UI, or use Sourcegraph's CLI (`src`) to run them programmatically (if you installed it).

4. Using Sourcegraph with ChatGPT Agent: If your Sourcegraph instance is accessible to the ChatGPT browsing (note: if you run it locally, ChatGPT's server likely cannot reach it unless you port-forward it to a public URL or run Sourcegraph on a cloud VM), you could have the agent browse the Sourcegraph web UI. A simpler route: use Sourcegraph's public cloud (sourcegraph.com) for open-source searches. The agent, as shown in earlier examples, can directly query sourcegraph.com results [15]. Even without an account, it can get some results. For instance, the agent might do: - `browse:"https://sourcegraph.com/search?q=rmf+OR+%22nist+800-53%22+ansible+terraform+count:10&patternType=literal"` to get top results (this is essentially what the example in the document suggests). If you find it not straightforward, you can always run searches yourself in the Sourcegraph UI and feed specific findings to the agent or into your analysis.

Why Sourcegraph Layer B? It complements our curated list with the ability to **discover new repositories or code snippets** that our initial harvest might miss. Think of it this way: REPO-INTEL's catalog is your high-quality library (depth over a known set), whereas Sourcegraph is your wide area scanner (breadth across many codebases). For example, if a new open-source project related to "ICBM telemetry" appears on GitHub (perhaps not popular enough to be on our initial keyword search), a Sourcegraph query for "telemetry Minuteman" might surface it. It's an on-demand research tool.

Now you have both layers: a focused catalog (Layer A) and an internet-scale code search (Layer B) [18] [21]. In practice, use the catalog first for known relevant solutions, and use Sourcegraph when you need to search for something beyond those known repos.

Step 16: Leveraging the Catalog for BD Goals and Competitive Advantage

With the technical setup complete, let's talk about **how to use these outputs in a business development context**. The ultimate aim of REPO-INTEL is not just to gather data, but to inform strategy and give Prime Technical Services an edge in capturing contracts. Here are ways to leverage the catalog and insights, integrated with competitive strategy:

- **Identify Solutions for High-Need Labor Categories:** Earlier competitive analysis pinpointed areas like MBSE (Model-Based Systems Engineering), cybersecurity/cloud, cleared software development, IT support, and niche domains as under-served by competitors [22]. Use the catalog to find open-source tools or frameworks in these categories:
- *MBSE/Digital Engineering:* Search the catalog for MBSE or SysML – you might find projects like OpenMBEE (if it was pulled) or related modeling tools. Knowing these tools can help you propose faster solutions or training for Model-Based Engineering roles that competitors struggle to fill.
- *Cybersecurity & RMF:* Our keywords heavily targeted RMF, STIG, etc. The catalog likely includes repositories for automated STIG checking (e.g., Ansible roles, OpenSCAP content). These can be pitched as force-multipliers when proposing to take over a cybersecurity task from a competitor –

"we will use proven open-source automation to quickly harden and continuously monitor systems, something the incumbent might not be doing." This addresses a pain point and shows innovation.

- *Cleared Software Developers:* While open-source repos won't directly list "cleared" status, the presence of many DoD-related projects (e.g., code related to mil-standard, or by defense contractors) could indicate where talent is contributing. Also, by curating cutting-edge tools (like a new framework or library in AI/ML or DevSecOps), you position your team to attract talent interested in those tools – thereby indirectly aiding recruiting for those hard-to-fill cleared dev positions.
- *Enterprise IT & Support:* Maybe the catalog shows projects for automating helpdesk, or scripts for Active Directory, etc., which you can use to train your IT support staff or streamline their work, making your service more efficient than competitors who rely purely on manpower.

- *Niche Domain Projects:* If any specialized repositories showed up (for example, a simulation toolkit for telemetry, or a nuclear surety reference implementation), having that in your library means you can quickly demonstrate knowledge in niche areas. E.g., "We saw an open-source project on GitHub that simulates telemetry for missile systems – our engineers have studied it and can leverage similar techniques for Sentinel." Even if niche, it's a differentiator to show you're aware of those details.

- **Map Repositories to BD "Wedge" Tactics:** In the *Competitive Capture & Insertion Strategy* document, several **messaging hooks** and tactics were recommended, such as highlighting long-open requisitions, offering high-touch service, quick clearance transfers, specialized focus, and surge support [23] [24]. You can use the intelligence from the repos to support these messages:

- *"Fresh Candidates for Long-Open Reqs":* If a repo indicates a technology or skill that has been hard to find (say a repo for a specialized military standard that few know), you can train candidates on it proactively. For instance, if Insight Global can't fill an MS&A engineer role for 6 months, and our catalog had an MS&A simulation open-source tool, we could get an engineer versed in that tool – effectively grooming a "fresh candidate" with relevant skills to fill that gap [25].
- *"Specialized Focus vs. Body-Shop":* Our curated repos show depth in DoD tech (like references to MIL-STDs, DoD cloud artifacts, etc.). This demonstrates Prime's *technical literacy* in defense matters. Use that: "We're not just body-shop recruiters; we actively maintain an internal library of open-source defense tools (from STIG automation to modeling simulators). We understand the tech and can hit the ground running" [24].

- *"Surge Support & Flexibility":* Should a client need to staff a team quickly, our repo intel could expedite ramp-up. For example, if a new Agile software team is needed, we might have a ready list of DevSecOps toolchain repos (CI/CD templates, container hardening scripts, etc.) to immediately provide the team with, making our 5-person surge deliver like a 7-person team. This agility in tooling and knowledge can be pitched in addition to just having people ready [26].

- **Inform Proposal Content and Past Performance:** The catalog and repo cards can be used in writing proposals or responses:

- When describing your technical approach, you can reference proven open-source components. E.g., "Our approach to RMF automation leverages open-source compliance as code (such as the SCAP Security Guide content [27] and OpenRMF) to accelerate accreditation" – citing from the catalog gives credibility and shows awareness.
- For **past performance** or capability statements, you might not list open-source projects explicitly, but the knowledge gained can subtly improve how you describe your capabilities ("...our engineers

have contributed to and utilized community-driven Hardened Configuration scripts for RHEL 8 STIG compliance...").

- The repo cards could even be shared (if appropriate) with a technical client to demonstrate the research done – e.g., "Here's a one-pager of an open-source tool we plan to use to solve your problem X." It shows preparedness and thoroughness.

- **Continuous Alignment with BD Goals:** As BD goals shift (new target programs, different tech focus), update the `keywords.yml` accordingly and rerun the harvest. For example, if next quarter the focus is on **AI/ML on classified systems**, you might add keywords like "PyTorch" and "FedRAMP High AI" or specific program names. The pipeline will fetch relevant repos (maybe open-source frameworks for secure AI). This keeps your intel current with your goals.

- **Leveraging Layer B for Competitive Intel:** You can also use Sourcegraph searches to spy on what competitors might be open-sourcing or discussing:

- Search for competitor names or domains in code. For instance, `Belcan` or `InsightGlobal` likely won't appear in code, but maybe smaller ones like "IronEagle" or "Shine Systems" could have GitHub accounts. If they have public repos (some smaller contractors open-source their internal tools), our allowlist or Sourcegraph might catch those. If found, those repos can reveal what tech stacks they use or problems they solved.
- Even if competitors don't publish code, they might use common frameworks. For instance, if we know a competitor staffs a project using ServiceNow, we can search open repos for "ServiceNow REST API code" to quickly educate our team on that integration, beating the competitor in expertise during proposal or execution.

- The **Sourcegraph** layer allows complex queries like `repo:*/insightglobal/` or searching code for strings like `"Apex Systems"` to see if someone's code or documentation mentions them (perhaps from a fork or config). Unlikely but interesting if something pops up.

- **Automation Pipelines and AI Missions:** With the harvested data, you can integrate it into automation:

- For example, a CI pipeline could use the `catalog.csv` to automatically open Pull Requests in some of your internal projects to update references (imagine you maintain an internal "awesome tools" list – it could be updated from catalog data).
- Or use the embeddings in an internal chatbot: employees could ask "Do we know any open source tools for vulnerability scanning?" and your bot (powered by the embeddings or the Q&A API) answers with one of the repos from the catalog (like it might pull up something related to Nessus automation if it was in a README).
- Another idea: incorporate the repo intel into your **talent management system**. If you parse the catalog and find trending languages or frameworks (say many top repos use **Go** for cloud, or **Rust** for cybersecurity tools), you might encourage training in those among your staff – aligning your workforce skills with what's cutting-edge (and likely to appear in RFPs).

Ultimately, the REPO-INTEL platform you built is a **strategic asset**. It keeps Prime Technical Services informed on technology relevant to contracts, helps identify where competitors are weak (if they're not

leveraging these tools and you are), and aids in quickly assembling solutions and proposals with confidence. It bridges the gap between open-source innovation and business capture strategy.

Conclusion and Next Steps

Congratulations – you have set up a comprehensive REPO-INTEL system from scratch! You started with just Python, Docker, and Git, and now you have:

- A structured repository for intelligence gathering, with all configurations and scripts in place.
- A living catalog of relevant GitHub repositories (fully updated with recent configurations and keywords tailored to your domain).
- Automated ways to refresh and maintain this knowledge base (scripts and optional GitHub Actions).
- Integration with advanced analysis tools: ChatGPT Agent Mode for interactive research, and Sourcegraph for wide-scale code search.
- A clear understanding of how to turn this technical data into business development insights, aligning with competitive strategy and capture tactics.

Going forward, treat this guide and your REPO-INTEL repo as a **living document**. Update the `keywords.yml` as new buzzwords or technologies emerge in your target market. Keep an eye on the catalog – you might discover new projects that warrant a closer look or even partnership. Encourage your technical team to contribute to the repo-intel process: they can suggest keywords, or add to the allowlist when they come across useful open-source in their work.

You now have a repeatable process to **harvest, analyze, and act on** open-source intelligence. This not only saves time (no more ad-hoc Googling for tools at the last minute of proposal writing) but also **elevates Prime Technical Services' technical credibility**. Your clients and partners will notice the preparedness and savvy that comes from having this library at your fingertips.

Lastly, don't forget to evangelize this internally – show the account managers how to use the catalog, perhaps host a lunch-and-learn to demonstrate asking ChatGPT questions and getting insights from the repo-intel. The more people use it, the more value it will deliver.

Good luck, and happy hunting for that next big contract!

Sources:

- Prime Tech REPO-INTEL framework and template structure [1] [2]
- REPO-INTEL harvest and ranking methodology (GitHub GraphQL, scoring criteria) [6] [7]
- ChatGPT Agent integration examples (using catalog and Sourcegraph) [14] [15]
- Competitive Capture Strategy – high-need labor categories & gaps [22], and messaging tactics [24] [26].
- Ansible Lockdown STIG repo example (open source tool for compliance) [28].

---

[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [20] [21] Original Framwork.docx
file://file-PN1RwMFE9R5JuKhP63ZFT1

19 Docker Single Container Deployment - Sourcegraph docs
https://sourcegraph.com/docs/admin/deploy/docker-single-container

22 23 24 25 26 Competitive Capture & Insertion Strategy for Prime Technical Services.pdf
file://file-P6gPDBcDQ9dGRbqc4BPDFK

27 DISA STIG for Red Hat Enterprise Linux 9 - Ansible role ... - GitHub
https://github.com/RedHatOfficial/ansible-role-rhel9-stig

28 ansible-lockdown/RHEL8-STIG - GitHub
https://github.com/ansible-lockdown/RHEL8-STIG