GitHub Actions example

```yaml
permissions:
  id-token: write # enables OIDC
  contents: read # Permits repo clone

steps:
  - uses: actions/checkout@11bd71901bbe5b1630ceea73d27597364c9af683 # v4.2.2

  # This works for Vault & Akeyless
  - name: Fetch DB password
    id: db
    uses: akeyless-community/akeyless-github-action@be876a4550345a460fa256acfa996a6d94a4642e # v1.1.2
    with:
      static-secrets: |
        - name: "/db/prod"
          prefix-json-secrets: "PG"
          parse-json-secrets: true

  - run: psql -h host -U app
    env:
      PGPASSWORD: ${{ env.PG_PASSWORD }}
```

C# Application Using Managed Identities example

https://learn.microsoft.com/en-us/azure/azure-sql/database/azure-sql-passwordless-migration?view=azuresql&tabs=sign-in-azure-cli%2Cazure-portal-create%2Cazure-portal-assign%2Capp-service-identity%2Capp-service-connector

```
// Passwordless App (Azure VM, AKS, App Service, Container Apps) to Database (SQL Server) connection
// Using Microsoft.Data.SqlClient

string connectionString = app.Configuration.GetConnectionString("AZURE_SQL_CONNECTIONSTRING")!;

using var conn = new SqlConnection(connectionString);
conn.Open();

var command = new SqlCommand("SELECT * FROM Persons", conn);
using SqlDataReader reader = command.ExecuteReader();

// Connection String Example
// Server=tcp:<database-server-name>.database.windows.net,1433;Initial Catalog=<database-name>
// ;Encrypt=True;TrustServerCertificate=False;Connection Timeout=30;Authentication="Active Directory
Default";
```

Applications: load secrets at startup via Vault/Akeyless SDK; never commit .env with real values and the file to GitHub

IAC: Avoid embedding secrets in state files (use managed identities); never commit .tfstate files to GitHub

---

Handling PII in code

Handling Fake or Desensitised PII in Source Code
When test data must look like real PII, make it unmistakably safe for scanners and reviewers by using **one** of the two approved signals below:

1. **Entire file is safe** – rename it:
   - `*_unpii.*` → fully desensitised data (no identifying risk)
   - `*_synthpii.*` → hand-crafted, realistic-looking synthetic PII
     A correctly suffixed filename tells automation to skip the whole file.
2. **Only certain lines are safe** – annotate them with a single token:

`LNRSPII:<SCOPE>:<CLASSIFICATION>:<DATA_TYPES>:<USER_ID>`

Field	Values	Purpose
SCOPE	`LINE`, `LINES_<n>`, `FILE`	Size of the exemption
CLASSIFICATION	`SYNTHETIC` | `NON_IDENTIFYING` | `FALSE_POSITIVE`	Why the data is harmless
DATA_TYPES	`SSN,PHONE,EMAIL,DL,DOB,ADDRESS,PERSON`	What detectors to silence
USER_ID	your corporate GitHub username	Personal accountability

---

Example:

```
# synthetic SSNs for the next three lines
LNRSPII:LINES_3:SYNTHETIC:SSN,EMAIL:jburger
123-45-6789
456-23-9870
073-55-9123

# entire file is fake — suffix tells the scanner to ignore everything
test_accounts_synthpii.json
```

Untyped or malformed fake PII will fail both code review and automated scans.

Edge-cases – when “harmless” data becomes PII

Data point(s)	PII?	Notes
Single RELX land-line number OR `@RELX.com` e-mail alone	**No**	Corporate-only & not person-specific
Mobile (cell) number + name	**Yes**	Associates unique number to individual
Postal code + birth date	**Yes**	Quasi-identifiers—combo can re-identify (HIPAA 18 identifiers)

Data point(s)	PII?	Notes
Employee ID + org chart title	**Yes**	Allows mapping to named person via directory
IP address alone	**Borderline**	Treat as PII if logged with timestamp and user context
Git commit author e-mail (`name@users.noreply.github.com`)	**No**	Already anonymised by GitHub

Rule: If two or more non-PII fields can single out a person, treat the combination as PII and handle as above.

Putting Example URLs in Code

GitHub Advanced Security's Secrets in URLs scanner uses pattern recognition to distinguish between real secrets and placeholder values in URLs. To prevent false positives, the scanner treats certain values as examples: usernames like `user` or `username`, passwords such as `pass` or `password`, and hostnames like `localhost`, `127.0.0.1`, or any domain ending in `example.com`. When these placeholders are used in a URL (e.g., `https://user:pass@example.com`), they are recognized as templates or documentation examples and are **not** flagged as secrets. This allows teams to safely include illustrative URLs in documentation or config files without triggering alerts.

Detection & prevention stack

- **Pre-commit hook** – `detect-secrets` (run locally).
- **GHAS secret scanning** – flags historic secrets.
- **GHAS push protection** – blocks new secrets, shows diff & remediation link.
- Alerts post to Teams **#GitHub-Security** automatically.

🛑 If sensitive data leaks:

If a secret is ever committed to a GitHub repository—**even in a single commit or closed branch—it must be treated as compromised**. Follow these steps to properly contain, clean up, and document the incident:

1. Revoke and Rotate the Secret

- **Immediately rotate or revoke** the exposed secret (API key, token, credential, etc.) following your team's **change control process**.
- Document the rotation via a change ticket or standard operational procedure.

2. Remove the Secret from Source Code

- Replace the exposed secret with a **reference to a secure variable**, such as:
  - An environment variable
  - A GitHub Actions secret

    - A secret pulled securely from Vault or Akeyless
- **Do not leave the original secret commented out or obfuscated**—remove it entirely

3. Choose a Cleanup Path

✅ **If the alert is a false positive or secret was rotated:**

- Reach out to your **BISO** or security contact to mark the alert as #**remediated or #false positive along with your name and date**

⚠️ **If the secret must be fully removed from Git history:**

- Understand this will **invalidate part of your Git audit trail**.
- Follow GitHub's documented guide: **Removing sensitive data from a repository**.
- Insurance has an excellent document on this process written by Rob Morrow. Here is a link
- Before moving any tags or branches to rewritten history:
    - Create an **empty commit** using `git commit --allow-empty` with a message like: `"Removed secret per [JIRA-1234], rewritten history by [name], peer reviewed by [reviewer]."`
- Push the cleaned branch/tags carefully.

4. Finalize and Document

- Push your fix to the repository and reference the original **secret scan alert ID** in your commit or PR message.
- Log the incident in the **Retrospective page** or designated security tracking system.

Golden rules (remember these)

1. **Vault/Akeyless or GitHub Secrets—nothing else.**
2. Inject secrets via env vars or OIDC, never hard-coded.
3. Review PR diffs for config & test files.
4. Tag synthetic or hashed PII clearly.
5. Need help? Post in Teams **#Open Azure Office Hours**