**Eliminating Service Account Keys: Using Workload Identity Federation (WIF) to Call Cloud Run Services Securely**

**1. Introduction**

Service Account JSON keys (keyfile.json) have long been used to authenticate applications calling Google Cloud services. While functional, they introduce **security risks**:

* Keys are **long-lived** and vulnerable to leakage.
* Keys must be **distributed** to client environments, increasing attack surface.
* Rotating keys is **manual** and error-prone.

**Workload Identity Federation (WIF)** solves this problem by enabling **short-lived, automatically rotated credentials** without persisting a key file. This paper explains how to use **WIF** to call **Cloud Run** services securely, eliminating the need for keyfile.json.

**2. Cloud Run Authentication Models**

Cloud Run supports **two** primary authentication models:

| **Authentication Type** | **Token Type Required** | **When to Use** |
| --- | --- | --- |
| **IAM-based auth** | OAuth **Access Token** (ya29.) | If service is restricted to IAM principals |
| **Audience-based OIDC auth** | **ID Token** (JWT with aud claim set to Cloud Run URL) | Default for service-to-service communication |

Most **private Cloud Run** deployments use **OIDC audience-based authentication**. In this mode:

* The caller must present an **ID Token** signed by a trusted Google issuer.
* The token’s **audience (aud) claim** must match the Cloud Run service URL.
* The calling identity must have the **roles/run.invoker** permission.

**3. Why Keyfile-Based Auth Is Problematic**

Typical Python code to call Cloud Run looks like this:

python

CopyEdit

import google.auth.transport.requests

import google.oauth2.id\_token

request = google.auth.transport.requests.Request()

audience = "https://my-service-abc.a.run.app/"

id\_token = google.oauth2.id\_token.fetch\_id\_token(request, audience)

This works **only** if:

* The application is running on GCP **or**
* GOOGLE\_APPLICATION\_CREDENTIALS points to a **service account key file**.

Problems with this approach:

* **Security risk**: key files can be leaked.
* **Compliance issue**: many enterprises forbid storing service account keys.
* **Operational burden**: rotating keys manually.

**4. Workload Identity Federation (WIF) to the Rescue**

**WIF** allows workloads running outside GCP (on-prem, other clouds, CI/CD) to:

1. **Authenticate using an external identity provider** (Azure AD, AWS IAM, GitHub OIDC, etc.).
2. **Exchange** that external token for a **short-lived Google STS token**.
3. **Impersonate** a Google service account to obtain the required credentials.
4. Use those credentials to **call Cloud Run** with an **ID Token**.

This **removes the need for a static service account key**.

**5. High-Level Architecture**

pgsql

CopyEdit

+------------------------+ +------------------------+ +------------------------+

| External Workload | | Workload Identity Pool | | Google Cloud IAM |

| (Azure VM, GitHub CI) | ---> | (Trust Configuration) | ---> | Service Account |

+------------------------+ +------------------------+ +------------------------+

External Token Token Exchange (STS) Short-lived Access Token

↓

Impersonate Target Service Account

↓

Generate Cloud Run ID Token (OIDC)

↓

Call Cloud Run HTTPS Endpoint

**6. Terraform Setup**

**6.1 Create Workload Identity Pool**

hcl

CopyEdit

resource "google\_iam\_workload\_identity\_pool" "my\_pool" {

workload\_identity\_pool\_id = "azure-wif-pool"

display\_name = "Azure WIF Pool"

}

**6.2 Create Provider (Azure AD Example)**

hcl

CopyEdit

resource "google\_iam\_workload\_identity\_pool\_provider" "azure\_provider" {

workload\_identity\_pool\_id = google\_iam\_workload\_identity\_pool.my\_pool.workload\_identity\_pool\_id

workload\_identity\_pool\_provider\_id = "azure-provider"

display\_name = "Azure Provider"

attribute\_mapping = {

"google.subject" = "assertion.sub"

}

oidc {

issuer\_uri = "https://sts.windows.net/<tenant-id>/"

}

}

**6.3 Service Account for Cloud Run**

hcl

CopyEdit

resource "google\_service\_account" "wif\_sa" {

account\_id = "traj-wif-sae-cd-dev-tlt-fr-a00-1"

display\_name = "WIF SA"

}

**6.4 Bind WIF Principal to Service Account**

hcl

CopyEdit

resource "google\_service\_account\_iam\_member" "allow\_wif\_to\_impersonate" {

service\_account\_id = google\_service\_account.wif\_sa.name

role = "roles/iam.serviceAccountTokenCreator"

member = "principal://iam.googleapis.com/projects/<PROJECT\_NUMBER>/locations/global/workloadIdentityPools/azure-wif-pool/subject/<subject-claim>"

}

**6.5 Allow Service Account to Call Cloud Run**

h

CopyEdit

resource "google\_cloud\_run\_service\_iam\_member" "allow\_invoker" {

service = "<cloud-run-service-name>"

location = "europe-west3"

role = "roles/run.invoker"

member = "serviceAccount:${google\_service\_account.wif\_sa.email}"

}

**7. Python Implementation**

**7.1 Generate WIF Access Token & Impersonate**

python

CopyEdit

import os

import json

import requests

from google.auth.transport.requests import Request

from google.oauth2 import impersonated\_credentials

from google.auth.credentials import Credentials

WIF\_HOME = os.environ.get("WIF\_HOME", ".")

WIF\_TOKEN\_FILENAME = os.path.join(WIF\_HOME, "wif\_token.txt")

def get\_impersonated\_id\_token(audience: str) -> str:

"""Generate an ID token for Cloud Run using WIF impersonation."""

# Read short-lived access token generated via WIF STS exchange

with open(WIF\_TOKEN\_FILENAME, "r") as f:

access\_token = f.read().strip()

target\_service\_account = "traj-wif-sae-cd-dev-tlt-fr-a00-1@<PROJECT\_ID>.iam.gserviceaccount.com"

# Call IAMCredentials API to generate ID token

url = f"https://iamcredentials.googleapis.com/v1/projects/-/serviceAccounts/{target\_service\_account}:generateIdToken"

headers = {"Authorization": f"Bearer {access\_token}", "Content-Type": "application/json"}

body = {"audience": audience, "includeEmail": True}

resp = requests.post(url, headers=headers, json=body)

if resp.status\_code != 200:

raise RuntimeError(f"Failed to get ID token: {resp.status\_code} - {resp.text}")

return resp.json()["token"]

**7.2 Call Cloud Run**

python

CopyEdit

cloud\_run\_url = "https://notes-enrich-service-182464272093.europe-west3.run.app/refine"

audience = "https://notes-enrich-service-182464272093.europe-west3.run.app/"

id\_token = get\_impersonated\_id\_token(audience)

payload = {"alertID": "INCI23455", "text": "Sample text", "model": "gemini", "prompt": "prompt1", "temp": 0.1}

headers = {"Authorization": f"Bearer {id\_token}", "Content-Type": "application/json"}

r = requests.post(cloud\_run\_url, json=payload, headers=headers)

print(r.status\_code, r.text)

**8. Security Benefits**

* **No static keys** — nothing to leak or rotate manually.
* **Short-lived tokens** — automatically expire in ~1 hour.
* **Fine-grained IAM** — grant only the permissions needed (run.invoker, serviceAccountTokenCreator).
* **Auditable** — every impersonation and invocation is logged in Cloud Audit Logs.

**9. Best Practices**

* Store **WIF token** only in memory or a secure temp file.
* Use **least privilege** — only roles/run.invoker and roles/iam.serviceAccountTokenCreator where needed.
* Rotate **trust configurations** periodically.
* Monitor **Cloud Audit Logs** for unusual impersonation patterns.

**10. Conclusion**

By replacing **service account key files** with **Workload Identity Federation**:

* You **remove a major security risk**.
* You **align with Google’s best practices** for credential management.
* You **enable secure cross-environment Cloud Run calls** without compromising security.

This pattern is applicable not just for Cloud Run, but **any Google Cloud service** requiring authenticated API calls.