# Experiment-Based Learning Generator

> Topic: **Migrating workloads from node service account to Workload Identity** · saved 2026-05-06T12:04:42

Download migrating-workloads-from-node-service-account-to-workload-id-20260506.md

# Migrating Workloads from Node Service Account to Workload Identity

## Introduction

In Google Kubernetes Engine (GKE), pods historically inherited the IAM identity of the Compute Engine VM they ran on — the **node service account**. This means every pod on a node shared the same Google Cloud permissions, often the broad `default` Compute Engine SA with `Editor` rights. That's a blast radius problem: a compromise of one pod equals a compromise of every pod on that node.

**Workload Identity** fixes this by binding a Kubernetes Service Account (KSA) to a Google Service Account (GSA) so that each pod authenticates to Google Cloud APIs as its own least-privileged identity. Under the hood, GKE runs a metadata server (`gke-metadata-server`) on each node that intercepts requests to the GCE metadata endpoint and returns tokens scoped to the pod's KSA, using the cluster as an OIDC identity provider trusted by IAM.

Key concepts to hold in mind: (1) the **identity binding** is `KSA ↔ GSA` via the `roles/iam.workloadIdentityUser` role; (2) a **workload identity pool** named `PROJECT_ID.svc.id.goog` represents your cluster's identities to IAM; (3) pods opt in via the KSA annotation `iam.gke.io/gcp-service-account`; (4) on the newer **Workload Identity Federation for GKE**, the KSA itself can be granted IAM roles directly without a GSA; (5) the migration is gradual — you can run it node-pool by node-pool and pod by pod.

This sheet walks you through the migration hands-on. You'll observe what the node SA actually exposes, set up Workload Identity, prove a pod gets a different identity, then migrate a real workload and lock down the node SA.

## Experiments

### 1. Observe the Node Service Account from Inside a Pod

**Goal**: See concretely what identity your pods currently run as and why that's risky.

**Materials**: - A GKE cluster (Standard mode, Workload Identity *not* enabled yet) - `gcloud` and `kubectl` configured - A test namespace: `kubectl create ns wi-lab`

**Procedure**: 1. Inspect your node pool's service account: `gcloud container node-pools describe NODE_POOL --cluster CLUSTER --zone ZONE --format='value(config.serviceAccount)'` 2. List the IAM roles bound to that SA: `gcloud projects get-iam-policy PROJECT_ID --flatten="bindings[].members" --filter="bindings.members:SA_EMAIL"` 3. Run an interactive pod: `kubectl -n wi-lab run probe --rm -it --image=google/cloud-sdk:slim -- bash` 4. From inside the pod, query the metadata server: `curl -H "Metadata-Flavor: Google" \ http://metadata.google.internal/computeMetadata/v1/instance/service-accounts/default/email gcloud auth list gsutil ls # try listing buckets`

**Observation**: The pod reports the node's SA email and can use any permission that SA holds. Note especially that `gsutil ls` may succeed against buckets you didn't intend any pod to touch. This is the "shared identity" problem you're about to fix.

### 2. Enable Workload Identity and Inspect the Metadata Server

**Goal**: Turn on Workload Identity and watch the metadata server change behavior.

**Materials**: Same cluster as Experiment 1.

**Procedure**: 1. Enable on the cluster: `gcloud container clusters update CLUSTER --zone ZONE \ --workload-pool=PROJECT_ID.svc.id.goog` 2. Enable on a node pool (or create a new one). Updating in place will recreate nodes: `gcloud container node-pools update NODE_POOL --cluster CLUSTER --zone ZONE \ --workload-metadata=GKE_METADATA` 3. Confirm the `gke-metadata-server` DaemonSet is running: `kubectl -n kube-system get ds gke-metadata-server -o wide` 4. Re-run the probe pod from Experiment 1 and call the metadata server again: `curl -H "Metadata-Flavor: Google" \ http://metadata.google.internal/computeMetadata/v1/instance/service-accounts/default/email`

**Observation**: The email returned is no longer the node SA — it's something like `PROJECT_ID.svc.id.goog[wi-lab/default]`, the KSA's federated identity. `gcloud auth list` shows no usable credentials yet because you haven't bound the KSA to anything. The "ambient" node identity is gone — that's the security win.

### 3. Bind a Kubernetes Service Account to a Google Service Account

**Goal**: Give one specific KSA the ability to act as one specific GSA.

**Materials**: A target Cloud Storage bucket: `gsutil mb gs://wi-lab-PROJECT_ID`.

**Procedure**: 1. Create a least-privileged GSA: `gcloud iam service-accounts create wi-reader --display-name="WI lab reader" gcloud projects add-iam-policy-binding PROJECT_ID \ --member="serviceAccount:wi-reader@PROJECT_ID.iam.gserviceaccount.com" \ --role="roles/storage.objectViewer"` 2. Create a KSA: `kubectl -n wi-lab create serviceaccount reader-ksa` 3. Allow the KSA to impersonate the GSA: `gcloud iam service-accounts add-iam-policy-binding \ wi-reader@PROJECT_ID.iam.gserviceaccount.com \ --role="roles/iam.workloadIdentityUser" \ --member="serviceAccount:PROJECT_ID.svc.id.goog[wi-lab/reader-ksa]"` 4. Annotate the KSA: `kubectl -n wi-lab annotate serviceaccount reader-ksa \ iam.gke.io/gcp-service-account=wi-reader@PROJECT_ID.iam.gserviceaccount.com` 5. Run a pod using that KSA: `kubectl -n wi-lab run reader --rm -it \ --image=google/cloud-sdk:slim \ --overrides='{"spec":{"serviceAccountName":"reader-ksa"}}' -- bash` 6. Inside: `gcloud auth list` and `gsutil ls gs://wi-lab-PROJECT_ID`.

**Observation**: `gcloud auth list` now shows `wi-reader@...` as the active identity. The pod can read the bucket. A pod in the same namespace using the *default* KSA (try it!) cannot — proving identity is now per-KSA, not per-node.

### 4. Migrate a Real Workload and Verify No Regression

**Goal**: Practice the migration pattern on an existing Deployment without downtime.

**Materials**: An existing app that calls a Google Cloud API (use a small one like a Pub/Sub publisher, or write a 10-line Python publisher).

**Procedure**: 1. Identify which Google APIs the app uses and the *minimum* roles it needs (e.g., `roles/pubsub.publisher` on one topic). Document them. 2. Create a dedicated GSA for this app and grant only those roles on only those resources. 3. Create a KSA in the app's namespace, bind it to the GSA (Experiment 3 pattern), and annotate it. 4. Edit the Deployment to add `spec.template.spec.serviceAccountName: <ksa>`. Apply with `kubectl apply` (rolling update). 5. Tail logs during rollout: `kubectl logs -f deploy/APP`. Hit the app with a smoke test. 6. Compare token source inside a new pod: `kubectl exec deploy/APP -- curl -sH "Metadata-Flavor: Google" \ http://metadata.google.internal/computeMetadata/v1/instance/service-accounts/default/email`

**Observation**: The new pods report the GSA email; old pods (during rollout) still report the node SA. If the smoke test fails, the error message will name the missing role — fix it on the GSA, not by adding roles to the node SA. This is the typical migration loop: tighten, test, tighten.

### 5. Remove Privileges from the Node Service Account

**Goal**: Confirm the migration is complete by removing the safety net.

**Materials**: A list of every workload in the cluster and the KSA it now uses.

**Procedure**: 1. Audit pods still using `default` KSAs without the WI annotation: `kubectl get pods -A -o jsonpath='{range .items[*]}{.metadata.namespace}{"/"}{.metadata.name}{"\t"}{.spec.serviceAccountName}{"\n"}{end}'` 2. For any remaining pod that calls Google APIs, repeat Experiment 4. 3. Switch all node pools to `--workload-metadata=GKE_METADATA` if not already. 4. In a *non-production* environment first, remove broad roles from the node SA: `gcloud projects remove-iam-policy-binding PROJECT_ID \ --member="serviceAccount:NODE_SA_EMAIL" --role="roles/editor"` Leave only what nodes themselves need (logging, monitoring, Artifact Registry reader). 5. Watch `kubectl get events -A` and Cloud Logging for `PERMISSION_DENIED` errors over the next hour.

**Observation**: A clean cluster shows no permission errors after node SA scoping — meaning no pod was secretly relying on inherited privilege. Any errors that appear point precisely to a workload you missed; fix it the right way (its KSA), not by re-granting the node SA.

## Keystone Project

**Motivation**: Tie everything together by migrating a small but realistic multi-service application end-to-end and producing artifacts a team could actually adopt.

**What to build**: Deploy a 3-service demo app to a fresh GKE cluster: - `frontend`: reads config from Cloud Storage - `worker`: publishes to Pub/Sub - `archiver`: writes to BigQuery

Then: 1. Stand up the cluster *without* Workload Identity initially, deploy with the default node SA holding `Editor`. Confirm everything works. 2. Enable Workload Identity on the cluster and node pool. 3. For each service, create a dedicated GSA with the **minimum** roles on the **specific** resources it touches. Create a KSA per service, bind, annotate, and roll out. 4. Strip the node SA down to `roles/logging.logWriter`, `roles/monitoring.metricWriter`, `roles/stackdriver.resourceMetadata.writer`, and `roles/artifactregistry.reader`. 5. Write a `MIGRATION.md` documenting: the per-service IAM table (KSA → GSA → roles → resources), how you verified each cutover, and a rollback plan.

**Success criteria**: - All three services function with the locked-down node SA. - `gcloud asset search-all-iam-policies` shows no `Editor` or `Owner` on any GSA used by pods. - Killing the metadata DaemonSet causes pods to fail authentication (proving they're not falling back to node identity). - Your `MIGRATION.md` is concrete enough that a peer can follow it on another cluster.

**Stretch variations**: - Replace GSA-impersonation with **direct KSA IAM bindings** (newer Workload Identity Federation for GKE) and compare the IAM policy size. - Add an **OPA/Gatekeeper or Kyverno** policy that rejects any pod using the `default` KSA in application namespaces. - Use **Terraform** to codify the GSA/KSA/IAM bindings so the whole topology is reproducible. - Enable **Cloud Audit Logs** for IAM and graph which GSAs were used by which pods over a week.

## Follow-up Ideas

1. **Workload Identity Federation beyond GKE**
   The same OIDC-trust mechanism lets GitHub Actions, AWS workloads, or on-prem Kubernetes authenticate to Google Cloud without service account keys. Learning this generalizes the mental model from "GKE feature" to "modern keyless auth."

2. **SPIFFE/SPIRE and workload identity standards**
   GKE Workload Identity is one implementation of a broader idea: cryptographically attested workload identity. SPIFFE/SPIRE shows how to do this in heterogeneous environments and underpins service meshes like Istio.

3. **Policy-as-code for IAM least privilege**
   Tools like IAM Recommender, Policy Analyzer, and `gcloud policy-troubleshoot` can mine logs to suggest minimum roles. Pair them with Terraform and OPA to keep your KSA→GSA mappings honest as the app evolves.

4. **mTLS service-to-service auth with Istio/Anthos Service Mesh**
   Once each pod has a real identity, you can authorize *intra-cluster* calls by identity too, not just network position. This is the natural next layer of zero-trust beyond authenticating to Google APIs.

5. **Secret-less databases via IAM auth**
   Cloud SQL, AlloyDB, and Spanner all support IAM database authentication. With Workload Identity in place, you can delete database passwords from your cluster entirely — a surprisingly large reduction in secret-management surface area.

---

▾ **View raw markdown**

```
---
title: "Migrating Workloads from Node Service Account to Workload Identity"
topic: "GKE Workload Identity Migration"
---

# Migrating Workloads from Node Service Account to Workload Identity
```

## Introduction

In Google Kubernetes Engine (GKE), pods historically inherited the IAM identity of the Compute Engine VM they ran on — the **node service account**. This means every pod on a node shared the same Google Cloud permissions, often the broad `default` Compute Engine SA with `Editor` rights. That's a blast radius problem: a compromise of one pod equals a compromise of every pod on that node.

**Workload Identity** fixes this by binding a Kubernetes Service Account (KSA) to a Google Service Account (GSA) so that each pod authenticates to Google Cloud APIs as its own least-privileged identity. Under the hood, GKE runs a metadata server (`gke-metadata-server`) on each node that intercepts requests to the GCE metadata endpoint and returns tokens scoped to the pod's KSA, using the cluster as an OIDC identity provider trusted by IAM.

Key concepts to hold in mind: (1) the **identity binding** is `KSA ↔ GSA` via the `roles/iam.workloadIdentityUser` role; (2) a **workload identity pool** named `PROJECT_ID.svc.id.goog` represents your cluster's identities to IAM; (3) pods opt in via the KSA annotation `iam.gke.io/gcp-service-account`; (4) on the newer **Workload Identity Federation for GKE**, the KSA itself can be granted IAM roles directly without a GSA; (5) the migration is gradual — you can run it node-pool by node-pool and pod by pod.

This sheet walks you through the migration hands-on. You'll observe what the node SA actually exposes, set up Workload Identity, prove a pod gets a different identity, then migrate a real workload and lock down the node SA.

## Experiments

### 1. Observe the Node Service Account from Inside a Pod

**Goal**: See concretely what identity your pods currently run as and why that's risky.

**Materials**:
- A GKE cluster (Standard mode, Workload Identity *not* enabled yet)
- `gcloud` and `kubectl` configured
- A test namespace: `kubectl create ns wi-lab`

**Procedure**:
1. Inspect your node pool's service account:
   ```
   gcloud container node-pools describe NODE_POOL --cluster CLUSTER --zone ZONE --format='value(config.serviceAccount)'
   ```
2. List the IAM roles bound to that SA:
   ```
   gcloud projects get-iam-policy PROJECT_ID --flatten="bindings[].members" --filter="bindings.members:SA_EMAIL"
   ```
3. Run an interactive pod:
   ```
   kubectl -n wi-lab run probe --rm -it --image=google/cloud-sdk:slim -- bash
   ```
4. From inside the pod, query the metadata server:
   ```
   curl -H "Metadata-Flavor: Google" \
     http://metadata.google.internal/computeMetadata/v1/instance/service-accounts/default/email
   gcloud auth list
   gsutil ls   # try listing buckets
   ```

**Observation**: The pod reports the node's SA email and can use any permission that SA holds. Note especially that `gsutil ls` may succeed against buckets you didn't intend any pod to touch. This is the "shared identity" problem you're about to fix.

### 2. Enable Workload Identity and Inspect the Metadata Server

**Goal**: Turn on Workload Identity and watch the metadata server change behavior.

**Materials**: Same cluster as Experiment 1.

**Procedure**:

1. Enable on the cluster:
   ```
   gcloud container clusters update CLUSTER --zone ZONE \
     --workload-pool=PROJECT_ID.svc.id.goog
   ```
2. Enable on a node pool (or create a new one). Updating in place will recreate nodes:
   ```
   gcloud container node-pools update NODE_POOL --cluster CLUSTER --zone ZONE \
     --workload-metadata=GKE_METADATA
   ```
3. Confirm the `gke-metadata-server` DaemonSet is running:
   ```
   kubectl -n kube-system get ds gke-metadata-server -o wide
   ```
4. Re-run the probe pod from Experiment 1 and call the metadata server again:
   ```
   curl -H "Metadata-Flavor: Google" \
     http://metadata.google.internal/computeMetadata/v1/instance/service-accounts/default/email
   ```

**Observation**: The email returned is no longer the node SA — it's something like `PROJECT_ID.svc.id.goog[wi-lab/default]`, the KSA's federated identity. `gcloud auth list` shows no usable credentials yet because you haven't bound the KSA to anything. The "ambient" node identity is gone — that's the security win.

### 3. Bind a Kubernetes Service Account to a Google Service Account

**Goal**: Give one specific KSA the ability to act as one specific GSA.

**Materials**: A target Cloud Storage bucket: `gsutil mb gs://wi-lab-PROJECT_ID`.

**Procedure**:
1. Create a least-privileged GSA:
   ```
   gcloud iam service-accounts create wi-reader --display-name="WI lab reader"
   gcloud projects add-iam-policy-binding PROJECT_ID \
     --member="serviceAccount:wi-reader@PROJECT_ID.iam.gserviceaccount.com" \
     --role="roles/storage.objectViewer"
   ```
2. Create a KSA:
   ```
   kubectl -n wi-lab create serviceaccount reader-ksa
   ```
3. Allow the KSA to impersonate the GSA:
   ```
   gcloud iam service-accounts add-iam-policy-binding \
     wi-reader@PROJECT_ID.iam.gserviceaccount.com \
     --role="roles/iam.workloadIdentityUser" \
     --member="serviceAccount:PROJECT_ID.svc.id.goog[wi-lab/reader-ksa]"
   ```
4. Annotate the KSA:
   ```
   kubectl -n wi-lab annotate serviceaccount reader-ksa \
     iam.gke.io/gcp-service-account=wi-reader@PROJECT_ID.iam.gserviceaccount.com
   ```
5. Run a pod using that KSA:
   ```
   kubectl -n wi-lab run reader --rm -it \
     --image=google/cloud-sdk:slim \
     --overrides='{"spec":{"serviceAccountName":"reader-ksa"}}' -- bash
   ```
6. Inside: `gcloud auth list` and `gsutil ls gs://wi-lab-PROJECT_ID`.

**Observation**: `gcloud auth list` now shows `wi-reader@...` as the active identity. The pod can read the bucket. A pod in the same namespace using the *default* KSA (try it!) cannot — proving identity is now per-KSA, not per-node.

### 4. Migrate a Real Workload and Verify No Regression

**Goal**: Practice the migration pattern on an existing Deployment without downtime.

**Materials**: An existing app that calls a Google Cloud API (use a small one like

a Pub/Sub publisher, or write a 10-line Python publisher).

**Procedure**:
1. Identify which Google APIs the app uses and the *minimum* roles it needs (e.g., `roles/pubsub.publisher` on one topic). Document them.
2. Create a dedicated GSA for this app and grant only those roles on only those resources.
3. Create a KSA in the app's namespace, bind it to the GSA (Experiment 3 pattern), and annotate it.
4. Edit the Deployment to add `spec.template.spec.serviceAccountName: <ksa>`. Apply with `kubectl apply` (rolling update).
5. Tail logs during rollout: `kubectl logs -f deploy/APP`. Hit the app with a smoke test.
6. Compare token source inside a new pod:
   ```
   kubectl exec deploy/APP -- curl -sH "Metadata-Flavor: Google" \
     http://metadata.google.internal/computeMetadata/v1/instance/service-accounts/default/email
   ```

**Observation**: The new pods report the GSA email; old pods (during rollout) still report the node SA. If the smoke test fails, the error message will name the missing role — fix it on the GSA, not by adding roles to the node SA. This is the typical migration loop: tighten, test, tighten.

### 5. Remove Privileges from the Node Service Account

**Goal**: Confirm the migration is complete by removing the safety net.

**Materials**: A list of every workload in the cluster and the KSA it now uses.

**Procedure**:
1. Audit pods still using `default` KSAs without the WI annotation:
   ```
   kubectl get pods -A -o jsonpath='{range .items[*]}{.metadata.namespace}{"/"}{.metadata.name}{"\t"}{.spec.serviceAccountName}{"\n"}{end}'
   ```
2. For any remaining pod that calls Google APIs, repeat Experiment 4.
3. Switch all node pools to `--workload-metadata=GKE_METADATA` if not already.
4. In a *non-production* environment first, remove broad roles from the node SA:
   ```
   gcloud projects remove-iam-policy-binding PROJECT_ID \
     --member="serviceAccount:NODE_SA_EMAIL" --role="roles/editor"
   ```
   Leave only what nodes themselves need (logging, monitoring, Artifact Registry reader).
5. Watch `kubectl get events -A` and Cloud Logging for `PERMISSION_DENIED` errors over the next hour.

**Observation**: A clean cluster shows no permission errors after node SA scoping — meaning no pod was secretly relying on inherited privilege. Any errors that appear point precisely to a workload you missed; fix it the right way (its KSA), not by re-granting the node SA.

## Keystone Project

**Motivation**: Tie everything together by migrating a small but realistic multi-service application end-to-end and producing artifacts a team could actually adopt.

**What to build**: Deploy a 3-service demo app to a fresh GKE cluster:
- `frontend`: reads config from Cloud Storage
- `worker`: publishes to Pub/Sub
- `archiver`: writes to BigQuery

Then:
1. Stand up the cluster *without* Workload Identity initially, deploy with the default node SA holding `Editor`. Confirm everything works.
2. Enable Workload Identity on the cluster and node pool.
3. For each service, create a dedicated GSA with the **minimum** roles on the **specific** resources it touches. Create a KSA per service, bind, annotate, and roll out.
4. Strip the node SA down to `roles/logging.logWriter`, `roles/monitoring.metricWriter`, `roles/stackdriver.resourceMetadata.writer`, and `roles/artifactregistry.reader`.
5. Write a `MIGRATION.md` documenting: the per-service IAM table (KSA → GSA → roles

→ resources), how you verified each cutover, and a rollback plan.

**Success criteria**:
- All three services function with the locked-down node SA.
- `gcloud asset search-all-iam-policies` shows no `Editor` or `Owner` on any GSA used by pods.
- Killing the metadata DaemonSet causes pods to fail authentication (proving they're not falling back to node identity).
- Your `MIGRATION.md` is concrete enough that a peer can follow it on another cluster.

**Stretch variations**:
- Replace GSA-impersonation with **direct KSA IAM bindings** (newer Workload Identity Federation for GKE) and compare the IAM policy size.
- Add an **OPA/Gatekeeper or Kyverno** policy that rejects any pod using the `default` KSA in application namespaces.
- Use **Terraform** to codify the GSA/KSA/IAM bindings so the whole topology is reproducible.
- Enable **Cloud Audit Logs** for IAM and graph which GSAs were used by which pods over a week.

## Follow-up Ideas

1. **Workload Identity Federation beyond GKE**
   The same OIDC-trust mechanism lets GitHub Actions, AWS workloads, or on-prem Kubernetes authenticate to Google Cloud without service account keys. Learning this generalizes the mental model from "GKE feature" to "modern keyless auth."

2. **SPIFFE/SPIRE and workload identity standards**
   GKE Workload Identity is one implementation of a broader idea: cryptographically attested workload identity. SPIFFE/SPIRE shows how to do this in heterogeneous environments and underpins service meshes like Istio.

3. **Policy-as-code for IAM least privilege**
   Tools like IAM Recommender, Policy Analyzer, and `gcloud policy-troubleshoot` can mine logs to suggest minimum roles. Pair them with Terraform and OPA to keep your KSA→GSA mappings honest as the app evolves.

4. **mTLS service-to-service auth with Istio/Anthos Service Mesh**
   Once each pod has a real identity, you can authorize *intra-cluster* calls by identity too, not just network position. This is the natural next layer of zero-trust beyond authenticating to Google APIs.

5. **Secret-less databases via IAM auth**
   Cloud SQL, AlloyDB, and Spanner all support IAM database authentication. With Workload Identity in place, you can delete database passwords from your cluster entirely — a surprisingly large reduction in secret-management surface area.