tamnoon Masterclass

Ghost Database Detection & Remediation

Modern cloud environments are some of the most complex pieces of infrastructure ever built, with data spread across multiple services, data stores, and external services. In all this complexity, it's surprisingly easy for database instances to slip through the cracks.
These forgotten or inactive databases—often referred to as "ghost databases"—can linger for months or years without being patched, monitored, or even noticed.
This guide is aimed at engineers and cloud security professionals who might not be aware of these risky parts of their infrastructure, and was written by Tamnoon's team of cloud security professionals to help the industry better address these challenges.

Table of Contents

Understanding Ghost Databases

A **ghost database** is any cloud-hosted DB instance (like AWS RDS, Azure SQL Database, GCP Cloud SQL) that has effectively fallen out of use but remains active and incurring costs. Typical signs of a ghost database include:

- No Recent Connections: No observed traffic (reads/writes) over a set threshold (e.g., 30 days).
- Missing Ownership Tags: No identifiable team or application is in charge.
- Stale Application References: The code or service that was once accessed is deprecated or replaced, but the DB instance itself was never retired.

From a distance, an idle database might appear harmless, but in reality, it can hold significant risk. Think of it as an abandoned warehouse on the edge of town: largely invisible day-to-day, yet still there, unlocked, and full of valuables.

Why Security Engineers Should Care

1. **Security Exposure**

- **Unpatched Vulnerabilities:** Ghost databases often escape routine security checks and patches. Attackers who discover such instances may find exploitable legacy versions of MySQL, PostgreSQL, or SQL Server.
- **Public Endpoints:** Some databases might have been accidentally left open to the internet or have weak security group rules, greatly increasing the chance of a breach.

2. Compliance & Data Protection

- **Sensitive Data:** Many ghost databases contain personally identifiable information (PII). Even if they're inactive, the data inside can violate retention policies or data privacy regulations such as GDPR or HIPAA.
- **Audit Failures:** Auditors or regulators can penalize organizations for losing track of where regulated data resides.

A Real-World Example of Ghost Databases

A **mid-sized healthcare provider** our team worked with once discovered over forty "test" Amazon RDS instances created during a series of proof-of-concept projects. Twelve of these instances contained actual patient records copied from production.

Since the proof-of-concept was abandoned, nobody realized that:

- These databases were still online (some even publicly accessible).
- They had not received security patches for over half a year.
- The monthly bill for all forty instances had climbed into the thousands (not strictly relevant, but important to the organization nonetheless.

When a routine cost analysis revealed the spike, the team tried to remediate the situation quickly.

In the end, they had to migrate or delete dozens of orphaned databases—a task made more complicated by compliance regulations around patient data (since, as soon as you discover you "hold" this data, there are relatively convoluted instructions around how to get rid of it properly).

This case, which fortunately ended without any serious damage, shows the kind of hidden, unmanaged risk lurking in ghost databases.

How to Hunt for Ghost Databases

Once you understand ghost databases and why they matter, the next challenge is finding them.

In AWS, ghost database detection hinges on **observing patterns of inactivity** and **confirming they're truly abandoned.** Below are the main AWS-native detection methods:

1. **Cloud Provider Methods**
    i. AWS CloudWatch Metrics
    ii. Amazon RDS Performance Insights
    iii. CLI-based Tag Audits and the Resource Tag Editor
    iv. AWS Cost Explorer & Budgets
2. **Open Source Approach**
    a. Querying - Steampipe
    b. Detection + Automation - CloudCustodian

Cloud Provider Methods

A. AWS CloudWatch Metrics

A CloudWatch offers a straightforward approach:

- `DatabaseConnections` monitors the average number of active connections. If it remains near zero for 14–30 days, the DB is likely idle.
- `CPUUtilization, FreeStorageSpace, FreeableMemory` that are unchanging or flat often indicate zero workload.

Below is a simple AWS CLI command that uses `DatabaseConnections` to find databases with no connections in the last 30 days:

```
START_TIME=$(date -d '30 days ago' +%Y-%m-%dT%H:%M:%S)
END_TIME=$(date +%Y-%m-%dT%H:%M:%S)
DB_ID="my-ghost-db"

aws cloudwatch get-metric-statistics \
  --namespace AWS/RDS \
  --metric-name DatabaseConnections \
  --start-time $START_TIME \
  --end-time $END_TIME \
  --period 86400 \
  --statistics Average \
  --dimensions Name=DBInstanceIdentifier,Value=$DB_ID
```

Which might yield something like this

```
{
    "Label": "DatabaseConnections",
    "Datapoints": [
        {
            "Timestamp": "2023-05-01T00:00:00Z",
            "Average": 0.0,
            "Unit": "Count"
        },
        {
            "Timestamp": "2023-05-02T00:00:00Z",
            "Average": 0.0,
            "Unit": "Count"
        },
        {
            "Timestamp": "2023-05-03T00:00:00Z",
            "Average": 0.0,
            "Unit": "Count"
        }
        // ... additional days omitted ...
    ],
    "ResponseMetadata": {
        "RequestId": "f1234567-8abc-90de-1112-13f56fgh7890",
        "HTTPStatusCode": 200,
        "RetryAttempts": 0
    }
}
```

You might note that the `average` attribute for `DatabaseConnections` remains 0.0 for multiple consecutive days, indicating that this is a ghost database with no traffic.

Tip: *Automate this via a scheduled script or Lambda to scan all RDS instances.*

B. Amazon RDS Performance Insights

If you enable **Amazon RDS Performance Insights**, you can detect usage (or lack thereof) more precisely.

More specifically, DBLoad is a time series metric that has dimensions, which are a way to "slice and dice" the metric according to its different characteristics.

Let's see a live example to better illustrate this. This lambda function does the following:

1. Queries `db.load.avg` over a 30-day window.
2. If the average load is negligible, the DB might be a ghost.
3. Logs the result, but you can also send it to Slack or SNS for notifications.

```
import boto3
import datetime

def lambda_handler(event, context):
    region = "us-east-1"
    db_arn = "arn:aws:rds:us-east-1:123456789012:db:my-ghost-db"
 # Replace with your DB ARN

    client = boto3.client("pi", region_name=region)

    end_time = datetime.datetime.utcnow()
    start_time = end_time - datetime.timedelta(days=30)

    try:
        # Query the "db.load.avg" metric from Performance Insights
        response = client.get_resource_metrics(
            ServiceType='RDS',
            Identifier=db_arn,
            MetricQueries=[
                {
                    'Metric': 'db.load.avg',
                    'GroupBy': {
                        'Group': 'db.sql_tokenized'
                    }
                }
            ],
```

```
            StartTime=start_time,
            EndTime=end_time,
            PeriodInSeconds=3600
        )

        if not response.get('MetricList'):
            print(f"[Ghost Check] DB '{db_arn}' returned no Performance Insights data.")
        else:
            metric_data_points = response['MetricList'][0]['DataPoints']
            total_load = sum(dp['Value'] for dp in metric_data_points)
            avg_load = total_load / len(metric_data_points) if metric_data_points else 0

            if avg_load < 0.01:
                print(f"[Ghost Check] DB '{db_arn}' is basically idle (avg load: {avg_load:.4f}).")
            else:
                print(f"[Ghost Check] DB '{db_arn}' has some activity (avg load:
{avg_load:.4f}).")

    except Exception as e:
        print(f"Error fetching PI data: {e}")
```

When it finishes running, it will look something like this:

```
START RequestId: 41f90045-deab-11ed-babc-1234567abcd
Version: $LATEST
[Ghost Check] DB 'arn:aws:rds:us-east-1:123456789012:db:my-
ghost-db' is basically idle (avg load: 0.0007).
END RequestId: 41f90045-deab-11ed-babc-1234567abcd
REPORT RequestId: 41f90045-deab-11ed-babc-1234567abcd
Duration: 142.53 ms  Billed Duration: 143 ms  Memory Size:
128 MB  Max Memory Used: 67 MB
```

Note that the Lambda found an average load of 0.0007 over the last 30 days, which is practically no queries or activity. This strongly indicates an abandoned (ghost) database.

C. CLI-based Tag Audits and the Resource Tag Editor

Databases lacking proper tags can easily slip through the cracks. When they do, you should first make sure they are actually ghost databases, and only then proceed to take action on them.

To identify tagless databases, you have two main approaches

1. **AWS Resource Groups Tag Editor**

- Using the Tag Editor Console, filter for RDS instances missing required tags like Owner or Environment.

2. **CLI Tag Check**

Running rds describe-db-instances and filtering on the tags could lead to a great visual representation of all missing tags:

```
aws rds describe-db-instances \
 --query 'DBInstances[].{DB:DBInstanceIdentifier,Tags:TagList}'
\
 --output table
```

This might yield something like the following:

```
-------------------------------------------------------------
|                        DescribeDbInstances                |
+----------------+------------------------------------------+
|      DB        |                Tags                      |
+----------------+------------------------------------------+
|  my-ghost-db   |  []                                      |
|  dev-database  |  [                                       |
|                |     {                                    |
|                |         "Key": "Owner",                  |
|                |         "Value": "dev-team"              |
|                |     }                                    |
|                |  ]                                       |
+----------------+------------------------------------------+
```

While **my-ghost-db** has no tags (in comparison to **dev-database** that has an Owner tag, this might trigger suspicion (no owner, no environment context), and when combined with zero connections, it's almost certainly a ghost DB.

Tip: Use **RCPs** (see our dedicated guide) or **IaC policies** to block RDS creation if the Owner tag is missing.

D. AWS Cost Explorer & Budgets

Ghost databases still rack up costs:

Cost Explorer

- Filter by "Amazon RDS" to spot ongoing monthly spend with near-zero usage.

AWS Budgets & Anomaly Detection

- If you see steady RDS costs and no reported usage, investigate further.

Open Source Approach

A. Steampipe for Cross-Service Analysis

Steampipe is an open-source tool for querying AWS resources with SQL. This avoids writing multiple scripts to call different AWS services.

The main reason for using Steampipe for this type of analysis is that it enables you to create really robust queries, ones that look at multiple types of resource metrics and compare them, and perform the computation “in-place,” getting just the results you want without resorting to multiple different calls to different APIs.

It also allows you to easily export and integrate the results with something like Slack or JIRA for continued follow-up.

The example query below identifies potentially unused RDS database instances in AWS by locating those with zero average connections over the past 30 days. It joins data from the AWS RDS database instance table with CloudWatch metrics, specifically looking at the DatabaseConnections metric averaged over one-day periods (86400 seconds).

You’ll note that it returns quite a lot of valuable data, including the database identifiers, engine types, instance classes, and allocated storage, which can all be used for further investigation.

```
SELECT
  db.db_instance_identifier,
  metrics.average AS avg_connections,
  db.engine,
  db.db_instance_class,
  db.allocated_storage
FROM
  aws_rds_db_instance AS db
JOIN
  aws_cloudwatch_metric_statistics AS metrics
ON
  db.db_instance_identifier = metrics.dimensions ->> 'DBInstanceIdentifier'
WHERE
  metrics.metric_name = 'DatabaseConnections'
  AND metrics.period = 86400        -- 1-day period
  AND metrics.start_time >= now() - interval '30 days'
  AND metrics.average = 0
ORDER BY
  db.db_instance_identifier;
```

Which might yield something like this:

```
+------------------------+---------------+---------+------------------
+-------------------+
| db_instance_identifier  | avg_connections| engine   | db_instance_class |
allocated_storage |
+------------------------+---------------+---------+------------------
+-------------------+
| my-ghost-db            | 0              | postgres | db.t3.medium      |
50                |
| old-test-db            | 0              | mysql    | db.t2.micro       |
20                |
+------------------------+---------------+---------+------------------
+-------------------+
```

Note that both `my-ghost-db` and `old-test-db` show **avg_connections** = 0 over the last 30 days, which makes them strong ghost candidates.

B. Policy-Based Detection/Automation with Cloud Custodian

Cloud Custodian lets you define YAML policies for automated detection and tagging.

The specific Cloud Custodian policy below scans all RDS resources and identifies instances with zero average database connections over a 30-day period. When such "ghost databases" are found, the policy automatically tags them with "ghost-db" and schedules them for deletion after a 7-day grace period, including a notification message.

Note that this slips a little bit into remediation. It does not only scan for these ghost databases; it also automatically “queues them up” for remediation and adds a notification.

```yaml
policies:
  - name: find-ghost-rds
    resource: rds
    description: "Detect RDS DB instances with zero connections over 30 days"
    filters:
      - type: metrics
        name: DatabaseConnections
        statistic: Average
        period: 86400
        days: 30
        value: 0
    actions:
      - type: mark-for-op
        tag: ghost-db
        msg: "No connections for 30 days. Scheduled for removal."
        op: delete
        days: 7
```

A dry-run of this policy (i.e. `custodian run --dryrun`) will yield something like this:

```
2025-03-15 12:10:03,784: custodian.policy:INFO policy:find-ghost-rds
resource:rds region:us-east-1 count:1 time:1.54
2025-03-15 12:10:03,784: custodian.actions:INFO Marked 1 resources for
op:delete ghost-db
Resource IDs: [ "my-ghost-db" ]
```

Ghost Database Remediation

Table of Contents

Identifying ghost databases is only half the battle. The next step involves systematically locking them down, backing them up if needed, and eventually removing them if no one claims ownership

Below is a tiered approach that balances the problem's security and operational aspects. In our experience, the technical portion is just the tool; the actual work here is around processes and organizational buy-in.

Tiered Cleanup Strategy

General overview

Below is a high-level sequence that we're going to dive into in the steps below:

1. **Day 0**: Detection flags `my-ghost-db` as idle.
2. **Day 1**: Triage → Tag with `ghost-db`, notify Slack channel or ticketing system.
3. **Day 8**: No response → Quarantine step using `aws rds modify-db-instance` (remove public access, attach `ghost-quarantine-sg`).
4. **Day 22**: Still unclaimed → Perform final backup (`aws rds create-db-snapshot`).
5. **Day 30**: Terminate (`aws rds delete-db-instance`).
6. **Day 31**: Profit! 🎉

Phase 1: Triage (Days 0–7)

1. Auto-Tagging

First, you should apply a special tag (e.g., `ghost-db` or `zombie-{date}`) to mark the instance as a candidate for removal using one or more of the methods we described for detection above.

Once that's done, remember that you're doing a "handoff"/approval process with other teams from the organization, so provide context in the tag or a related field (e.g., `reason: "no connections for 30 days"`).

2. Owner Notification

Integrate with your preferred channel (Slack, Microsoft Teams, or email via AWS SNS) to send notifications to the relevant person in the organization.

A typical notification includes a DB identifier, inactivity duration, and a link to relevant runbooks or instructions.

Make sure to add a grace period. It gives application owners time to speak up if they still need the database (e.g., for monthly reporting or a rarely used backup).

The script below is a very simplified version of tagging that adds a status tag to a database, noting that you have marked it as a ghost database at a specific date.

```
aws rds add-tags-to-resource \

--resource-name arn:aws:rds:us-east-1:123456789012:db:my-ghost-db \

--tags Key="status",Value="ghost-db" Key="detectedOn",Value="$(date +%Y-%m-
%d)"
```

Phase 2: Quarantine (Days 8–21)

If there's no response from a responsible team, or if the owner confirms the DB is no longer needed but still wants a final review, you should "quarantine" the database for a **set period of time** to allow for all logistics to take place before we actually delete the database.

We suggest following a relatively robust quarantine process by performing three distinct actions on three different layers: network, security groups, and IAM roles.

1. Network Isolation

Start by removing any public accessibility to prevent inbound connections from the internet. This is a good and simple example of how to do that using the AWS CLI.

```
aws rds modify-db-instance \

  --db-instance-identifier my-ghost-db \

  --no-publicly-accessible \

  --apply-immediately
```

2. Restrictive Security Group

Create a security group (e.g., `ghost-quarantine-sg`) with inbound rules allowing access **only** from trusted admin IPs or a bastion host, then attach it to the DB instance, removing the old security groups.

Note that this will still allow the database owner (who we assume comes from `203.0.113.5/32`) to access the database and perform a final review.

```
# 1) Create a quarantine SG with no inbound rules
aws ec2 create-security-group \
  --group-name ghost-quarantine-sg \
  --description "Quarantine security group for ghost DBs" \
  --vpc-id vpc-123abc

# 2) Restrict inbound traffic to admin IP (e.g., 203.0.113.5/32)
aws ec2 authorize-security-group-ingress \
  --group-id sg-987xyz \
  --protocol tcp \
  --port 3306 \
  --cidr 203.0.113.5/32
```

3. IAM Lockdown

Remove all IAM roles except those for admin or backup processes and rotate or remove DB credentials from AWS Secrets Manager if they're no longer needed.

This is a relatively involved process that should be handled with care; it constitutes "playing around" with the AWS secret manager and can cause irreversible damage. Be sure to **work cautiously and do not run commands you do not understand:**

```
# 1) List all IAM roles associated with the database (to identify what
needs to be removed)
aws rds describe-db-instances \
  --db-instance-identifier my-ghost-db \
  --query "DBInstances[0].AssociatedRoles[*].RoleArn" \
  --output text

# 2) Remove all associated roles except admin/backup roles
aws rds remove-role-from-db-instance \
  --db-instance-identifier my-ghost-db \
  --role-arn arn:aws:iam::123456789012:role/unnecessary-role-1

# 3) Repeat removal for each unnecessary role
aws rds remove-role-from-db-instance \
  --db-instance-identifier my-ghost-db \
  --role-arn arn:aws:iam::123456789012:role/unnecessary-role-2 # Add here
all roles, one by one
# 4) Find any secrets for this database in Secrets Manager
aws secretsmanager list-secrets \
  --filters Key=description,Values="Credentials for my-ghost-db" \
  --query "SecretList[*].ARN" \
  --output text

# 5) Remove unnecessary secrets or disable access - repeat for all secrets
aws secretsmanager update-secret \
  --secret-id arn:aws:secretsmanager:region:123456789012:secret:my-ghost-
db-credentials-abc123 \
  --description "QUARANTINED: Former credentials for my-ghost-db"

# 6) Apply resource policy restricting access to the secret
aws secretsmanager put-resource-policy \
  --secret-id arn:aws:secretsmanager:region:123456789012:secret:my-ghost-
db-credentials-abc123 \
  --resource-policy '{
    "Version": "2012-10-17",
    "Statement": [{
      "Effect": "Deny",
      "Principal": "*",
      "Action": "secretsmanager:GetSecretValue",
      "Resource": "*",
      "Condition": {
        "ArnNotEquals": {
          "aws:PrincipalArn": [
            "arn:aws:iam::123456789012:role/AdminRole",
            "arn:aws:iam::123456789012:role/BackupRole"
          ]
        }
      }
    }]
  }'
```

Once you've performed the three steps above for every ghost DB you've found, it's safe to say that these databases are quarantined. Next, you should wait 2-3 weeks to ensure no one is raising any flags that these databases are required and only proceed to the last stage.

Phase 3: Backup & Termination (Days 22–30)

If the database remains unclaimed or is confirmed to be unnecessary after quarantine:

1. Final Backup

Use **AWS Backup** or manual snapshot with encryption (KMS) while tagging the snapshot with a retention period or store it in a locked backup vault for legal hold if needed (see more below).

Taking a snapshot is easy:

```
aws rds create-db-snapshot \
  --db-instance-identifier my-ghost-db \
  --db-snapshot-identifier my-ghost-db-final-$(date +%Y%m%d)
```

2. Delete the DB Instance

HERE BE DRAGONS

Always double-check the snapshot or backup was successful, and document the action (e.g., in a Jira ticket or Change Management system).

Once you're done, delete the instance.

```
aws rds delete-db-instance \
  --db-instance-identifier my-ghost-db \
  --skip-final-snapshot
```

(Use `--skip-final-snapshot` only if you have already created a snapshot or do not need one.)

Handling Special Cases

You should be mindful of a few delicate cases when doing this type of work.

Remember that in most organizations, data is not simply used—it's also stored for various compliance and legal reasons. In addition, there are legacy applications that no one particularly owns anymore that gets accessed rarely. These are still viable applications and should become unavailable just because the database engine running them is old.

Below are a few common examples of special cases you should look out for and what you should do about them:

1. **Legal Hold Databases:** Some DBs might need indefinite storage for compliance reasons (e.g., eDiscovery). Use AWS Backup Vault Lock or store final snapshots in a separate account with restricted access.
2. **Legacy Databases with Partial Activity:** A DB that sees a single query per month can look "ghostly." Communicate with application owners to confirm whether monthly or quarterly usage is intentional or truly redundant.
3. **Cross-Region Replicas:** Make sure you track down any read replicas. Deleting the primary might leave behind a "ghost" replica in a different region or vice versa.

This phased approach ensures minimal disruption if a database is erroneously flagged while still providing a clear path to reclaim cost and reduce the attack surface.

Summary

Looking for a TLDR? Ask, and you shall receive. Here's a quick recap of the main takeaways from this masterclass:

1. Ghost databases are forgotten or inactive cloud-hosted databases that still consume resources and pose security risks.
2. They often go unnoticed because nobody monitors them, updates them, or pays attention to ownership tags.
3. Security risks include unpatched vulnerabilities, public endpoints left open, and unknown data exposure (like PII).
4. Detection methods rely on low or zero connection metrics, missing or incorrect tags, and cost spikes. Tools like CloudWatch, RDS Performance Insights, AWS Cost Explorer, Steampipe, and Cloud Custodian help locate ghost databases.
5. Remediation follows a tiered process:
   a. **Triage** by tagging and notifying possible owners.
   b. **Quarantine** by restricting network, security groups, and IAM roles.
   c. **Backup & Termination** after confirming no owners need the data.
6. Special considerations include compliance requirements, legal holds, and legitimate but infrequent usage.