# Google Gemini

**Clip source:** Google Gemini

# Google Gemini

You are an expert AWS engineer.

Your goal is to rightsize AWS RDS Postgres Instance, AWS RDS Postgres Multi AZ Instance and AWS Aurora Postgres.

What would be the criteria to assess the right size.

We have instance size, IOPS and cloudwatch metrics of usage.

Also recommend if Aurora serverless would be a choice.
This is a multidimensional optimization problem involving **Performance**, **Availability**, and **Cost**. Rightsizing is not just about cutting costs; it is about matching provisioned capacity to actual workload demand with a safety buffer.

Here is the expert criteria for assessing rightsizing for AWS RDS and Aurora Postgres, followed by the assessment of Aurora Serverless v2.

# 1. The Rightsizing Framework

You must evaluate three primary dimensions: **Compute (CPU)**, **Memory (RAM)**, and **Storage I/O**.

## A. Compute (CPU) Optimization

Postgres is often memory-bound rather than CPU-bound, but CPU sizing determines your network throughput and EBS bandwidth limits.

- **Key CloudWatch Metric:** `CPUUtilization`
- **Assessment Criteria:**

- **Over-provisioned:** If peak CPU < 40% and average CPU < 20% over a 30-day period.

  *Action:* Downsize by one instance class (e.g., `db.m5.2xlarge` to `db.m5.xlarge`).
- **Under-provisioned:** If average CPU > 70% or you see frequent spikes hitting 100% causing latency.

  *Action:* Upsize or switch to a Compute Optimized (c6g/c7g) family if memory is sufficient.
- **Burstable Consideration:** If using T-series (e.g., `db.t3`), check `CPUCreditBalance`. If it consistently drops to zero, you are being throttled; move to M-series or R-series.

## B. Memory (RAM) Optimization

This is the most critical metric for Postgres. Postgres relies heavily on the filesystem cache (OS page cache) for performance.

- **Key CloudWatch Metrics:** `FreeableMemory`, `SwapUsage`, `ReadIOPS`
- **Assessment Criteria:**
  - **Over-provisioned:** If `FreeableMemory` is consistently > 60% of total allocated RAM and `ReadIOPS` are near zero (indicating the working set fits entirely in RAM with room to spare).
  - **Under-provisioned:**

    **Swap usage is > 0:** This is a red flag. Postgres performance degrades massive if it swaps. Upsize immediately.
    **High ReadIOPS with Low FreeableMemory:** This indicates "Thrashing." The working set is larger than the buffer pool, forcing Postgres to fetch pages from disk constantly. Switch to Memory Optimized (r6g/r7g) classes.

## C. Storage I/O Optimization

Performance issues here often look like CPU issues.

- **Key CloudWatch Metrics:** `ReadIOPS`, `WriteIOPS`, `DiskQueueDepth`, `ReadThroughput`, `WriteThroughput`
- **Assessment Criteria:**

- **Throughput Bound:** If your throughput hits the limit of the *Instance Class*. (Note: Smaller instances have lower throughput caps, regardless of how fast your EBS volume is).
- **Latency:** Check `ReadLatency` and `WriteLatency`. If `DiskQueueDepth` > 5 consistently, your storage is the bottleneck.

# 2. Service-Specific Nuances

## AWS RDS Postgres (Single AZ)

- **Focus:** Purely match the workload to the instance.
- **Metric to Watch:** `EBSByteBalance%` (for burstable instances). If this drops, your disk performance will tank.

## AWS RDS Postgres (Multi-AZ)

- **The "Double Write" Penalty:** In Multi-AZ, every write is synchronously replicated to the standby.
- **Criteria:** If `WriteLatency` is high but CPU is low, the bottleneck might be the synchronous replication lag, often caused by network distance or instance size limits.
- **Rightsizing Tip:** You cannot size the Primary and Standby differently. If you downsize, you downsize both. Ensure the new smaller instance supports the network bandwidth required for synchronous replication.

## AWS Aurora Postgres

Aurora handles I/O differently. It uses a log-structured storage system and does not use standard EBS volumes.

- **The "IO-Optimized" Choice:** If your bill shows I/O costs > 25% of your total database spend, switch the cluster configuration to **Aurora I/O-Optimized**. This removes I/O charges in exchange for a higher instance hourly rate.
- **Metric Change:** Ignore `DiskQueueDepth`. Focus on `BufferCacheHitRatio`. If this drops below 99% (for OLTP), you need more RAM (larger instance), not more I/O.

# 3. Recommendation: Is Aurora Serverless v2 a choice?

Aurora Serverless v2 is a powerful tool, but it is **not** always cheaper. It scales capacity in "Aurora Capacity Units" (ACUs).

| Feature | **Choose Serverless v2 IF...** | **Stick to Provisioned IF...** |
| --- | --- | --- |
| **Workload Pattern** | Highly unpredictable, spiky, or intermittent (e.g., Dev/Test, seasonal apps). | Steady-state, predictable production load (e.g., 24/7 SaaS). |
| **Cost Efficiency** | The database is idle for long periods (scales down to 0.5 ACU). | You run at constant high utilization. (Serverless ACUs cost ~2x more per GB of RAM than provisioned instances). |
| **Scaling Speed** | You need instant vertical scaling (milliseconds) to handle "flash crowds." | You can tolerate the ~15 min downtime or failover time to resize an instance manually. |

**The Verdict on Serverless:**

- **Do not use Serverless v2** if your application has a consistent baseline load (e.g., always uses at least 16GB RAM). You will pay a "convenience premium" significantly higher than a reserved provisioned instance.
- **Use Serverless v2** for non-production environments or workloads where the database is idle >60% of the time.

# 4. Summary Decision Matrix

| Metric | Condition | Recommendation |
|---|---|---|
| **CPU** | Max < 40% (30 days) | **Downsize** Instance Class (e.g., 4xl -> 2xl) |
| **RAM** | Swap > 0 OR Cache Hit < 95% | **Upsize** to Memory Optimized (Family `r6g`) |
| **IOPS** | DiskQueueDepth > 5 | **Modify Storage** to gp3 or increase IOPS |
| **Pattern** | Idle > 60% of time | **Migrate** to Aurora Serverless v2 |

**Would you like me to analyze a specific set of these metrics if you paste them here (e.g., "CPU 20%, Freeable Memory 50GB") to give a precise resizing recommendation?**
using boto3 client to generate metrics for storage I/O optimization .

Apply assessment criteria to make decisions.

Analysis could be for the last 15 days.

Authentication uses SSO login. Profile and Region will be provided as input arguments.

Use sparklines for charts.
This is a Python script using `boto3` to perform the Storage I/O optimization analysis. It fetches metadata from the RDS API to determine provisioned limits and compares them against 15 days of CloudWatch metrics.

# Prerequisites

You will need to install `boto3` and `sparklines`:

Bash

```
pip install boto3 sparklines
```

# The Analysis Script ( `rds_io_analyzer.py` )

This script authenticates via SSO profile, detects if the instance is RDS or Aurora, identifies the storage limits (IOPS/Throughput) based on the configuration, and visualizes usage with sparklines.

Python

```
import boto3
import argparse
import sys
from datetime import datetime, timedelta
from statistics import mean
import sparklines # pip install sparklines

def get_args():
parser = argparse.ArgumentParser(description="RDS/Aurora Storage I/O
Rightsizing Analyzer")
parser.add_argument("--profile", required=True, help="AWS SSO Profile
Name")
parser.add_argument("--region", required=True, help="AWS Region")
parser.add_argument("--db-identifier", required=True, help="RDS Instance
or Aurora Cluster Identifier")
return parser.parse_args()

def get_cw_metric(cw_client, namespace, metric, db_id, start, end):
"""Fetches Max and Avg metrics for the last 15 days with 1-hour
granularity"""
try:
response = cw_client.get_metric_statistics(
Namespace=namespace,
MetricName=metric,
Dimensions=[{'Name': 'DBInstanceIdentifier', 'Value': db_id}],
StartTime=start,
EndTime=end,
Period=3600, # 1 Hour granularity for 15 day overview
Statistics=['Average', 'Maximum']
)
# Sort by timestamp to ensure sparkline order
return sorted(response['Datapoints'], key=lambda x: x['Timestamp'])
```

```
except Exception as e:
print(f"Error fetching {metric}: {e}")
return []

def print_spark(label, data_points, key='Maximum', unit='Count'):
if not data_points:
print(f"{label}: No Data")
return 0, 0

values = [dp[key] for dp in data_points]
if not values:
return 0, 0
avg_val = mean(values)
max_val = max(values)
# Create sparkline
spark = sparklines.sparklines(values)[0]
# Format string
fmt_str = f"{label:<20} |{spark}| Max: {max_val:.2f} {unit}, Avg:
{avg_val:.2f} {unit}"
print(fmt_str)
return max_val, avg_val

def assess_io(db_info, read_iops, write_iops, read_thru, write_thru,
queue_depth):
print("n--- Rightsizing Assessment ---")
# 1. Calculate Total IOPS (Max Read + Max Write is a rough worst-case
estimate)
# In reality, peaks might not align, but this is safe conservative
sizing.
max_read_iops = max([x['Maximum'] for x in read_iops]) if read_iops else
0
max_write_iops = max([x['Maximum'] for x in write_iops]) if write_iops
else 0
total_max_iops = max_read_iops + max_write_iops
storage_type = db_info.get('StorageType', 'aurora')
allocated_storage = db_info.get('AllocatedStorage', 0)
provisioned_iops = db_info.get('Iops', 0) # For io1, io2, gp3
# --- IOPS Assessment ---
print(f"Detected Storage: {storage_type.upper()} | Provisioned IOPS:
{provisioned_iops if provisioned_iops else 'Baseline'}")
```

```
# gp2 baseline is 3 IOPS per GB, min 100, max 16,000 (burstable to 3000)
baseline_iops = max(100, min(allocated_storage * 3, 16000))
if total_max_iops > baseline_iops:
print(f"⚠️ [CRITICAL] IOPS Bound: Max IOPS ({total_max_iops:.0f})
exceeds gp2 baseline ({baseline_iops}). Latency likely.")
print(f" >>> Recommendation: Switch to gp3 or Provisioned IOPS (io1).")
else:
print(f"✅ IOPS within gp2 baseline limits.")
elif storage_type in ['io1', 'io2', 'gp3'] and provisioned_iops:
utilization = (total_max_iops / provisioned_iops) * 100
if utilization > 90:
print(f"⚠️ [CRITICAL] IOPS Saturation: Reaching {utilization:.1f}% of
provisioned limit.")
print(f" >>> Recommendation: Increase Provisioned IOPS.")
elif utilization < 40:
print(f" [SAVINGS] Over-provisioned IOPS. Peak usage is only
{utilization:.1f}%.")
print(f" >>> Recommendation: Decrease Provisioned IOPS to save costs.")
else:
print(f"✅ IOPS sizing is healthy.")

# --- Throughput Assessment ---
# Simplified check for general limits (approx 500-1000 MB/s for many
instances, varies by class)
max_total_throughput_mb = (max([x['Maximum'] for x in read_thru],
default=0) +
max([x['Maximum'] for x in write_thru], default=0)) / 1024 / 1024
print(f"Peak Throughput: {max_total_throughput_mb:.2f} MB/s")
# --- Latency/Queue Assessment ---
# Queue Depth > 5 usually implies disk cannot keep up with requests
max_qd, avg_qd = 0, 0
if queue_depth:
vals = [x['Average'] for x in queue_depth]
max_qd = max(vals)
avg_qd = mean(vals)
if avg_qd > 5:
print(f"⚠️ [CRITICAL] High Queue Depth (Avg: {avg_qd:.2f}). Disk
subsystem is a bottleneck.")
elif max_qd > 20:
```

```
else:
print(f"✅ Queue Depth is healthy (Avg < 5).")

def main():
args = get_args()
session = boto3.Session(profile_name=args.profile,
region_name=args.region)
rds = session.client('rds')
cw = session.client('cloudwatch')
# 1. Get Instance Details
print(f"Fetching details for: {args.db_identifier}...")
try:
db_resp =
rds.describe_db_instances(DBInstanceIdentifier=args.db_identifier)
db_info = db_resp['DBInstances'][0]
except Exception as e:
print(f"Error finding DB: {e}")
sys.exit(1)

# 2. Time Window (Last 15 days)
end_time = datetime.utcnow()
start_time = end_time - timedelta(days=15)
print(f"Analyzing Metrics: {start_time.strftime('%Y-%m-%d')} to
{end_time.strftime('%Y-%m-%d')}")
print("-" * 60)

# 3. Fetch Metrics
# Note: Aurora uses 'VolumeReadIOPS' usually at cluster level, but
'ReadIOPS' at instance level works for standard RDS.
# We assume Standard RDS metric names here.
namespace = 'AWS/RDS'
r_iops = get_cw_metric(cw, namespace, 'ReadIOPS', args.db_identifier,
start_time, end_time)
w_iops = get_cw_metric(cw, namespace, 'WriteIOPS', args.db_identifier,
start_time, end_time)
r_thru = get_cw_metric(cw, namespace, 'ReadThroughput',
args.db_identifier, start_time, end_time)
w_thru = get_cw_metric(cw, namespace, 'WriteThroughput',
args.db_identifier, start_time, end_time)
```

```
# 4. Visualize
print_spark("Read IOPS", r_iops)
print_spark("Write IOPS", w_iops)
print_spark("Read Throughput", r_thru, unit='B/s')
print_spark("Write Throughput", w_thru, unit='B/s')
print_spark("Queue Depth", q_depth, key='Average', unit='Count') # QD is usually best viewed as Average
# 5. Assess
assess_io(db_info, r_iops, w_iops, r_thru, w_thru, q_depth)

if __name__ == "__main__":
main()
```

# How to run it

Bash

```
python rds_io_analyzer.py --profile my-sso-profile --region us-east-1 --db-identifier my-production-db
```

# Interpretation of Results

1. **Sparklines**:
   - The visual line `▁▂▃▅` shows the trend over the last 15 days.
   - Look for "Flat tops" (████). This indicates you are hitting a hard limit (Clipping) and need to upsize.
2. **Queue Depth**:
   - This is the most honest metric. Even if IOPS looks low, if Queue Depth is high (>5), the disk cannot handle the specific I/O pattern (e.g., random access vs sequential).
3. **IOPS vs. Savings**:
   - If your Provisioned IOPS utilization is `< 40%`, the script will recommend saving money by lowering provisioned IOPS or switching to `gp3` (which is cheaper than `io1`).

explain Queue Depth:

The easiest way to understand **Queue Depth** is to think of it like a checkout line at a grocery store.

- **The Disk (Storage)** is the Cashier.
- **The I/O Requests (Reads/Writes)** are the Customers.
- **Throughput/IOPS** is how fast the cashier scans items.
- **Queue Depth** is how many people are standing in line waiting to be served *including* the person currently being served.

# Why is it the "Honest Metric"?

You might look at your IOPS and see you are doing 1,000 IOPS on a disk capable of 3,000 IOPS. You think, "Great, I'm only at 33% capacity! I'm safe."

But then you check **Queue Depth** and it is sitting at **20**.

**Why the disconnect?** If the "Customers" (I/O requests) are difficult—for example, random read operations that require the disk head to jump around physically (on HDD) or complex scatter-gather operations—the Cashier slows down. Even though they are only scanning 1 item per second (low IOPS), the line is building up out the door.

# The Rule of Thumb: > 5

- **Queue Depth 0-1:** Ideally, the disk is waiting for work or handling it instantly.
- **Queue Depth < Instance vCPU Count:** Generally acceptable. If you have 4 vCPUs, a queue of 4 means every CPU has one storage request pending. This is efficient.
- **Queue Depth > 5 (consistently):** This is the danger zone. It means requests are arriving faster than the disk can service them.
  - **Result:** The application experiences "Latency." The database halts query execution while it waits for the disk to return data. Your users see a spinning wheel.

# Summary

If **IOPS are low** but **Queue Depth is high**, throwing more IOPS at the problem might not help if the latency is caused by the *nature* of the request (e.g., small random writes). You might need to change the storage type (e.g., to one with lower latency like io2 Block Express) or optimize the queries to be more sequential.

Would you like me to explain how to verify if your specific queue depth is caused by **Latency** versus **Volume**?

does queue depth comes in during writes only or also during reads

Queue Depth applies to **both writes AND reads**.

It is an aggregate metric. The `DiskQueueDepth` metric in CloudWatch sums up the total number of outstanding requests waiting to be served by the storage volume, regardless of whether they are asking to retrieve data (Read) or save data (Write).

Here is how they differ in impact:

# 1. Read Queue Depth (The "Blocking" Queue)

- **When it happens:** Your database tries to fetch data that isn't in RAM (Buffer Pool). It has to go to the disk.
- **Why it's painful:** Reads are usually **synchronous**. The application (and the user) stops and waits until the data returns.
- **Symptoms:** If Queue Depth is high due to reads, your application feels "slow" or "frozen."
- **Common Cause:** Cold cache (fresh restart), doing a full table scan on a large table, or insufficient RAM.

# 2. Write Queue Depth (The "Buffered" Queue)

- **When it happens:** You are inserting, updating, or deleting data.
- **Why it's (sometimes) less painful:** Writes in Postgres are often **asynchronous** regarding the data files. The database writes to the WAL (Write Ahead Log) first (which is fast and sequential) and "checkpoints" the dirty pages to the main data files later in the background.
- **Symptoms:** If Queue Depth is high due to writes, you might not notice it immediately in user latency *unless* the queue gets so deep that the WAL fills up or the checkpoint process creates a backlog, forcing the database to pause writes to catch up.
- **Common Cause:** Bulk data loading, heavy batch jobs, or an instance with low write throughput limits.

# How to distinguish them?

CloudWatch doesn't give you a separate "ReadQueueDepth" and "WriteQueueDepth" metric directly. You have to infer it by correlating `DiskQueueDepth` with your IOPS metrics:

- **High Queue Depth + High `ReadIOPS`:** You have a Read bottleneck. (Solution: More RAM or faster disk).
- **High Queue Depth + High `WriteIOPS`:** You have a Write bottleneck. (Solution: Higher IOPS/Throughput storage).

**Would you like me to explain how "Burst Balance" affects this on gp2 volumes?**
explain how "Burst Balance" affects this on gp2 volumes?

"Burst Balance" is the fuel gauge for your `gp2` volume's performance. When it hits empty, your "sports car" (the disk) suddenly turns into a bicycle, but the traffic (your database requests) doesn't stop.

This is the most common cause of sudden, unexplained spikes in Queue Depth on smaller `gp2` volumes.

# 1. The Mechanic: The Credit Bucket

On `gp2`, performance is tied to size. You earn I/O credits at a rate of **3 IOPS per GB**.

- **The Baseline:** A 100 GB drive earns 300 IOPS constantly.
- **The Burst:** Regardless of size, AWS lets you burst up to **3,000 IOPS** for short periods.
- **The Cost:** Every time you burst above your baseline, you spend credits from your bucket (Burst Balance).

# 2. The "Cliff" Scenario

Imagine you have a **100 GB volume**.

- **Baseline:** 300 IOPS.
- **Burst:** 3,000 IOPS.

**Scenario:** Your application runs a heavy report needing 1,500 IOPS.

1. **10:00 AM:** You have 100% Burst Balance. The disk happily delivers 1,500 IOPS. The Queue Depth is low (0-1).
2. **10:15 AM:** You are burning credits faster than you earn them. The balance drops to 0%.
3. **10:16 AM (The Cliff):** AWS strictly enforces the baseline. Your disk capacity instantly drops from 3,000 IOPS capability down to **300 IOPS**.

This is where the disaster happens. Your application is still sending 1,500 requests per second. But the disk can now only process 300.

- **Deficit:** 1,200 requests per second are arriving but not being served.
- **Result:** They pile up in the queue. Your **Queue Depth** goes from 1 to 100+ in seconds.
- **Symptom:** The database locks up. Queries time out. The CPU might actually *drop* because it's just waiting on I/O (iowait).

# 4. How to Spot it & Fix it

- **The Metric:** Check `BurstBalance` in CloudWatch. It is a percentage (0-100%).
- **The Danger Zone:** If it is trending down effectively (e.g., < 40%), you are living on borrowed time.

**The Fixes:**

1. **The "Modern" Fix:** Switch to **gp3**.
   - `gp3` does not use burst buckets. It gives a flat, consistent 3,000 IOPS baseline regardless of volume size. It effectively eliminates the "Cliff" risk.
2. **The "Old" Fix (if stuck on gp2):** Increase the storage size.
   - Since baseline = 3 IOPS * Size, a 1 TB drive gets a baseline of 3,000 IOPS. It never needs to burst because its baseline matches the burst limit.

**Recommendation:** If you see `BurstBalance` dropping, convert the volume to `gp3` immediately. It is usually cheaper and safer.
using boto3 client to generate metrics for cpu optimization .

Apply assessment criteria to make decisions.

Analysis could be for the last 15 days.

Authentication uses SSO login. Profile and Region.

Also include Configuration for DB instance - Postgres engine version, vCPU and vRAM.
Here is the Python script to analyze CPU optimization. It includes a specific function to resolve the hardware specifications (vCPU and RAM) by mapping the RDS instance class to its underlying EC2 definition.

# The Analysis Script (`rds_cpu_analyzer.py`)

Python

```
import boto3
import argparse
import sys
from datetime import datetime, timedelta
from statistics import mean
import sparklines # pip install sparklines

def get_args():
parser = argparse.ArgumentParser(description="RDS/Aurora CPU Rightsizing
Analyzer")
parser.add_argument("--profile", required=True, help="AWS SSO Profile
Name")
parser.add_argument("--region", required=True, help="AWS Region")
parser.add_argument("--db-identifier", required=True, help="RDS Instance
or Aurora Cluster Identifier")
return parser.parse_args()

def get_db_config(session, db_id):
"""Fetches DB details and maps to EC2 spec to find vCPU/RAM"""
rds = session.client('rds')
ec2 = session.client('ec2')
try:
# 1. Get RDS details
resp = rds.describe_db_instances(DBInstanceIdentifier=db_id)
db = resp['DBInstances'][0]
class_name = db['DBInstanceClass']
engine = db['Engine']
version = db['EngineVersion']
# 2. Map to EC2 to get vCPU/RAM (Remove 'db.' prefix, e.g., db.m5.large
-> m5.large)
ec2_type = class_name.replace('db.', '')
vcpu = "Unknown"
ram = "Unknown"
try:
# Skip for 'serverless' classes if encountered
if 'serverless' not in class_name:
ec2_resp = ec2.describe_instance_types(InstanceTypes=[ec2_type])
if ec2_resp['InstanceTypes']:
spec = ec2_resp['InstanceTypes'][0]
```

```
vcpu = spec['VCpuInfo']['DefaultVCpus']
ram_gb = spec['MemoryInfo']['SizeInMiB'] / 1024
ram = f"{ram_gb:.0f} GB"
except Exception:
pass # Fallback if EC2 lookup fails (e.g. some older/custom legacy
classes)

return {
'id': db_id,
'class': class_name,
'engine': f"{engine} {version}",
'vcpu': vcpu,
'ram': ram,
'is_burstable': class_name.startswith('db.t')
}
except Exception as e:
print(f"Error fetching DB Config: {e}")
sys.exit(1)

def get_cw_metric(cw_client, namespace, metric, db_id, start, end):
try:
response = cw_client.get_metric_statistics(
Namespace=namespace,
MetricName=metric,
Dimensions=[{'Name': 'DBInstanceIdentifier', 'Value': db_id}],
StartTime=start,
EndTime=end,
Period=3600, # 1 Hour granularity
Statistics=['Average', 'Maximum']
)
return sorted(response['Datapoints'], key=lambda x: x['Timestamp'])
except Exception:
return []

def print_spark(label, data_points, key='Maximum', unit='%'):
if not data_points:
print(f"{label:<25} | No Data")
return 0, 0

values = [dp[key] for dp in data_points]
```

```
max_val = max(values)
spark = sparklines.sparklines(values)[0]
print(f"{label:<25} |{spark}| Max: {max_val:.2f}{unit}, Avg:
{avg_val:.2f}{unit}")
return max_val, avg_val

def assess_cpu(config, max_cpu, avg_cpu, min_credits=None):
print("n--- Rightsizing Assessment ---")
# 1. Check Utilization
if max_cpu < 40 and avg_cpu < 20:
print(f" [OVER-PROVISIONED] Instance is underutilized.")
print(f" Criteria: Peak CPU ({max_cpu:.1f}%) < 40% and Avg
({avg_cpu:.1f}%) < 20%.")
print(f" Recommendation: Downsize instance (e.g., to next smaller size
in {config['class'].split('.')[1]} family).")
elif avg_cpu > 70 or max_cpu > 95:
print(f" [UNDER-PROVISIONED] Instance is heavily loaded.")
print(f" Criteria: Avg CPU > 70% or Frequent spikes > 95%.")
print(f" Recommendation: Upsize instance or switch to Compute Optimized
(c6g/c7g).")
else:
print(f"✅ [OPTIMIZED] CPU usage is within healthy bounds.")

# 2. Check Burstable Credits (T-series only)
if config['is_burstable']:
print(f"n--- Burstable (T-Series) Check ---")
if min_credits is not None and min_credits < 50:
print(f"⚠️ [CRITICAL] CPU Credits Exhausted (Min: {min_credits:.2f}).")
print(f" Performance is being throttled to baseline.")
print(f" Recommendation: Switch to Unlimited Mode or move to M/R
instance family.")
else:
print(f"✅ Credit balance is healthy.")

def main():
args = get_args()
session = boto3.Session(profile_name=args.profile,
region_name=args.region)
cw = session.client('cloudwatch')
# 1. Get Configuration
```

```
config = get_db_config(session, args.db_identifier)
print("n" + "="*60)
print(f"DB Identifier : {config['id']}")
print(f"Engine : {config['engine']}")
print(f"Instance Class : {config['class']}")
print(f"Hardware Spec : {config['vcpu']} vCPU | {config['ram']} RAM")
print("="*60 + "n")

# 2. Time Window
end = datetime.utcnow()
start = end - timedelta(days=15)
# 3. Fetch Metrics
cpu_data = get_cw_metric(cw, 'AWS/RDS', 'CPUUtilization',
args.db_identifier, start, end)
# Fetch credits only if T-series
credit_data = []
if config['is_burstable']:
credit_data = get_cw_metric(cw, 'AWS/RDS', 'CPUCreditBalance',
args.db_identifier, start, end)

# 4. Visualize
max_cpu, avg_cpu = print_spark("CPU Utilization", cpu_data,
key='Maximum', unit='%')
min_credits = None
if config['is_burstable'] and credit_data:
# For credits, we care about the Minimum balance
vals = [x['Average'] for x in credit_data]
min_credits = min(vals)
spark = sparklines.sparklines(vals)[0]
print(f"{'CPU Credit Balance':<25} |{spark}| Min: {min_credits:.2f}")

# 5. Make Decisions
assess_cpu(config, max_cpu, avg_cpu, min_credits)

if __name__ == "__main__":
main()
```

# How to Run

Bash

```
python rds_cpu_analyzer.py --profile my-sso-profile --region us-east-1 --db-identifier my-db-instance
```

# Assessment Criteria Logic Used

1. **Over-provisioned:** If the **Maximum** CPU never exceeded 40% and the **Average** stayed below 20% for the full 15 days, the instance is too large.
2. **Under-provisioned:** If the **Average** CPU is consistently > 70%, the database is CPU-bound and likely causing query latency.
3. **Burstable Check:** If it is a T-series (e.g., `db.t3.medium`) and `CPUCreditBalance` drops near zero, the script flags a critical throttling warning.

Gemini can make mistakes, so double-check it