Serverless FinOps AutoStop Prototype Development

Thesis Phase 4: Analysis of the results and Prototype Development

1. Introduction

The objective of this phase is to develop a code-based functional prototype designed to demonstrate the automated shutdown of unused cloud resources. This practical implementation addresses the critical issue of resource wastage identified in the previous phases of the research.

The developed solution is a **Serverless FinOps AutoStop Automation** tool that proactively identifies and stops idle Amazon EC2 instances. By leveraging an Event-Driven architecture, the system ensures that compute costs are incurred only when resources are actively delivering value, strictly adhering to FinOps principles.

2. Solution Architecture

The prototype is built on a **Serverless** architecture to minimize the operational overhead and cost of the tool itself.

2.1 Core Components

- **AWS Lambda (Compute):** The core logic is encapsulated in a Python function. Being stateless and ephemeral, it eliminates the need for a dedicated management server, which would itself incur costs ("FinOps for FinOps").
- **Infrastructure as Code:** The entire solution is defined declaratively using **AWS SAM** (Serverless Application Model), enabling repeatable deployments and version control of the infrastructure configuration.
- **Amazon EventBridge (Scheduler):** Acts as the reliable trigger, invoking the Lambda function on a configurable schedule (default: hourly).
- **Tagging Strategy (Source of Truth):**
    - The system avoids external state databases (like *DynamoDB*) to reduce complexity and cost.
    - **Logic:** It strictly targets only instances tagged with AutoStop: true. This "Opt-In" model prevents accidental disruptions to critical legacy workloads.

2.2 Notification System

The solution integrates with Amazon SNS (Simple Notification Service) to provide real-time visibility into automated actions.

- **SNS Topic:** A dedicated topic is provisioned for all AutoStop events.
- **Email Subscription:** Configurable email alert notify stakeholders when instances are stopped.
- **Structured Payloads:** Notifications include machine-readable JSON with instance metadata (e.g. *instance_id*, *instance_name*, *owner*, *reason*, *timestamp*), enabling integration with external systems.

3. Implementation Details

The implementation consists of two primary artifacts: the application logic (autostop.py), and the infrastructure definition (template.yaml). The following sections detail the key implementation decisions.

3.1 Idle Detection Logic

A resource is considered idle only if it meets **all** the following configurable conditions over a lookback window (default: 60 minutes):

- **CPU Utilization:** Below a specific threshold (Default: < 2%).
- **Network Activity:** Total In/Out traffic below a threshold (Default: < 5 MiB).
- **Disk I/O:** Total Read/Write data volume below a threshold (Default: < 150 MiB). The system captures both instance-store operations (*DiskReadBytes, DiskWriteBytes*) and EBS volume operations (*EBSReadBytes, EBSWriteBytes*) to ensure accurate detection across all EC2 instance families. Legacy instance-store metrics are treated as optional, since modern EBS-only instances (e.g. T3, M5) do not report them.

This multi-dimensional approach prevents false positives where an instance might be processing a memory-intensive job (low CPU) or waiting for a long database query (low Network).

3.2 Concurrency & Scalability

AWS Lambda functions have a maximum execution timeout of **15 minutes**. The function is configured to use this maximum value (Timeout: 900 seconds) to ensure sufficient time for processing large instance fleets, and implements a **multi-threaded** approach using ThreadPoolExecutor.

- **Optimization:** Since CloudWatch API calls are I/O bound, the code parallelizes these requests (*MAX_WORKERS* default: 10).
- **Network Tuning:** The botocore configuration was tuned (max_pool_connections) to ensure thread safety and prevent connection pool blocking.

3.3 Guardrails (Safety Mechanisms)

To ensure production safety, several guardrails are hardcoded:

- **Spot Instances:** Automatically skipped, as they cannot be stopped (only terminated).
- **Auto Scaling Groups (ASG):** Skipped to prevent the ASG from detecting an "unhealthy" stopped instance and launching a replacement.
- **Warm-up Period:** Instances launched within the last 5 minutes are ignored to allow for initialization scripts (UserData) to complete.
- **Manual KeepAlive Override:** Instances tagged with *KeepAlive: true* are unconditionally skipped, providing operations teams with a mechanism to temporarily protect instances during critical maintenance windows without removing them from the AutoStop program.

4. Security & Compliance

Security is a foundational element of this thesis, demonstrating how automation can be both powerful and secure.

4.1 Attribute-Based Access Control (ABAC)

Instead of granting the Lambda blanket permission (ec2:StopInstances on Resource: *), the IAM Policy enforces a strict condition:

"Condition": { "StringEquals": { "ec2:ResourceTag/AutoStop": "true" } }

Impact: Even if the Lambda credentials are compromised, an attacker cannot stop critical Production Databases unless they are explicitly tagged with AutoStop. This implements the **Principle of Least Privilege**.

4.2 Privilege Escalation Prevention

The function is also restricted from tagging arbitrary resources. The ec2:CreateTags permission is similarly scoped, preventing a compromised function from "tagging to target" (i.e., adding the AutoStop tag to a critical server just to be able to stop it).

5. Reliability Patterns

Reliability is prioritized over cost savings during failure modes.

5.1 Fail-Open Strategy

The system implements a "Fail-Open" design. If the monitoring data (CloudWatch metrics) is unavailable due to an API error or network issue, the system **defaults to assuming the**

instance is BUSY. Rationale: It is better to waste a few cents keeping an idle server running than to incur a business outage by erroneously stopping a server we cannot monitor.

5.2 Dry-Run Mode

The solution supports a DRY_RUN environment variable. When enabled, the system performs all checks and logs all intended actions ("SIMULATION_STOP") without actually calling the stop API. This allows for safe auditing and testing in sensitive environments.

6. Observability

To move beyond simple logging, the prototype implements **Structured JSON Logging**.

- **Implementation:** A custom JsonFormatter captures all context (Instance ID, Request ID, Metric Values).
- **Benefit:** This allows logs to be machine-parsed by tools like CloudWatch Logs Insights. We verify not just that the code ran, but we record the *exact* metric values even for active instances, allowing for data-driven tuning of thresholds (e.g., proving that 105MB disk usage is normal background noise).

7. Future Work

For future iterations of this project, the following improvements are identified:

- **Cross-Instance Metric Batching:** The current implementation issues one *GetMetricData* call per instance. For fleets exceeding 10,000 instances, queries could be consolidated to include multiple instances per API call, significantly reducing API costs and execution time.
- **Hibernation Support:** Extending the logic to support EC2 Hibernation, preserving memory state for faster resume times.
- **Owner Notification:** Integrating with chat platforms (Slack/Teams) to dynamically notify the specific owner defined in the instance tags.