Research Journal of Engineering Technology and Medical Sciences (ISSN: 2582-6212), Volume 09, Issue 01, March-2026
Available at www.rjetm.in/

FINOPS DASHBOARD: A Cloud Cost Intelligence and Monitoring System using AWS Cost Explorer, Lambda, MySQL, and Grafana

Prashant Choudhary
Dept. of Information Technology
Barkatullah University Institute of Technology
Bhopal (M.P.), India
Prashantagchoudhary2004@gmail.com

Ankit Nagdiya
Dept. of Information Technology
Barkatullah University Institute of Technology
Bhopal (M.P.), India
ankitnagdiya917@gmail.com

Ankush Kumar
Dept. of Information Technology
Barkatullah University Institute of Technology
Bhopal (M.P.), India
ankushkumar981130@gmail.com

Shivanshu Bunkar
Dept. of Information Technology
Barkatullah University Institute of Technology
Bhopal (M.P.), India
sibbu.bunkar26@gmail.com

Mrs. Deepanjali Tarey
Asst. Prof. at Dept. of Information Technology
Barkatullah University Institute of Technology
Bhopal (M.P.), India

Mr. Pawan Singh Rajput
Asst. Prof. at Dept. of Information Technology
Barkatullah University Institute of Technology
Bhopal (M.P.), India

Dr. Rachna Kulhare
Asst. Prof. at Dept. of Information Technology
Barkatullah University Institute of Technology
Bhopal (M.P.), India

***Abstract*—This paper presents *FinOps Dashboard*, an automated cloud cost intelligence and monitoring system designed to provide real-time visibility into AWS cloud expenditure. Traditional cloud cost management relies on manual inspection of billing consoles, resulting in delayed insights, budget overruns, and limited accountability across teams. The FinOps Dashboard leverages AWS Cost Explorer for cost data retrieval, AWS Lambda for serverless data ingestion, MySQL for structured storage, and Grafana for interactive visualization. The system automatically collects daily cost and usage data, groups it by service and environment tag, persists records into a relational database, and presents them through a unified dashboard consisting of cost trend graphs, service-level pie charts, and total expenditure statistics. The proposed solution demonstrates how modern cloud-native and open-source technologies can replace reactive cost management with a proactive, data-driven FinOps practice.**

***Index Terms*—FinOps, AWS Cost Explorer, AWS Lambda, Cloud Cost Optimization, Grafana, MySQL, Serverless, Cloud Monitoring, Cost Intelligence**

I. Introduction

The rapid adoption of cloud infrastructure has introduced a significant challenge for organizations: managing and optimizing cloud expenditure effectively. While cloud platforms like Amazon Web Services (AWS) offer flexible, on-demand pricing, the complexity of cost attribution across services, teams, and environments often leads to unexpected billing, poor visibility, and budget overruns.

Problem Statement: Organizations using AWS lack a centralized, automated, and developer-friendly mechanism to continuously monitor, aggregate, and visualize cloud cost data across services and environments. Native AWS billing tools provide raw data but offer limited historical analysis, custom tagging visibility, and integration with third-party monitoring systems.

Objective: To design and implement an automated cloud cost intelligence pipeline that periodically fetches AWS cost and usage data through the Cost Explorer API, stores it in a structured relational database, and renders it as actionable visual dashboards using Grafana. The FinOps Dashboard addresses the need for real-time cost accountability, service-level expenditure tracking, and environment-based cost attribution — all without any manual intervention.

Cloud financial management, commonly referred to as FinOps, is a cultural and operational practice that enables engineering, finance, and business teams to make data-driven cloud spending decisions [1]. The system presented in this paper operationalizes the core principles of FinOps by automating cost visibility and enabling granular analysis at the service

and tag level.

II. Literature Review

Cloud cost management has emerged as a critical discipline as enterprise cloud adoption continues to grow. Research by Gartner indicates that organizations waste approximately 30% of their cloud spend due to lack of visibility and poor resource allocation [2]. Several tools and approaches have been proposed to address this problem.

Native cloud billing dashboards such as the AWS Cost Management Console provide basic cost breakdowns but require manual navigation and lack persistent historical storage or custom dashboard capabilities. Third-party platforms such as CloudHealth by VMware and Apptio Cloudability offer comprehensive FinOps features but are expensive and proprietary, making them inaccessible to small and medium-sized teams. Academic research on cloud cost optimization highlights the importance of tag-based cost attribution and granular daily monitoring for identifying cost anomalies [3]. Studies also demonstrate that serverless architectures such as AWS Lambda are highly effective for periodic data pipeline tasks due to their event-driven nature and pay-per-execution pricing model [4]. Grafana has been widely adopted in industry for monitoring and observability dashboards, with MySQL as a supported data source for structured metric storage [5].

The FinOps Dashboard builds upon these findings by integrating AWS Cost Explorer, Lambda, MySQL, and Grafana into a cohesive, open-source cost intelligence pipeline accessible to development teams without significant infrastructure overhead.

III. Methodology

Tech Stack:

- AWS Cost Explorer API (Cost Data Source)
- AWS Lambda (Serverless Ingestion Function)
- Boto3 SDK (AWS Python SDK for API Integration)
- MySQL (Relational Database for Cost Storage)
- MySQL Connector for Python (Database Adapter)
- Grafana (Dashboard and Visualization Layer)
- Python 3.10 (Primary Programming Language)
- AWS CloudWatch Events (Lambda Scheduling Trigger)

Architecture Overview:

- Event-driven serverless pipeline triggered daily via CloudWatch
- AWS Cost Explorer queried with daily granularity and multi-dimensional grouping
- Normalized relational schema for structured cost and usage persistence
- Grafana connected to MySQL for real-time dashboard rendering
- Tag-based attribution using the `Environment` tag key

Development Workflow:

1) Database schema design for the `aws_cost_usage` table
2) Lambda function implementation for cost data ingestion

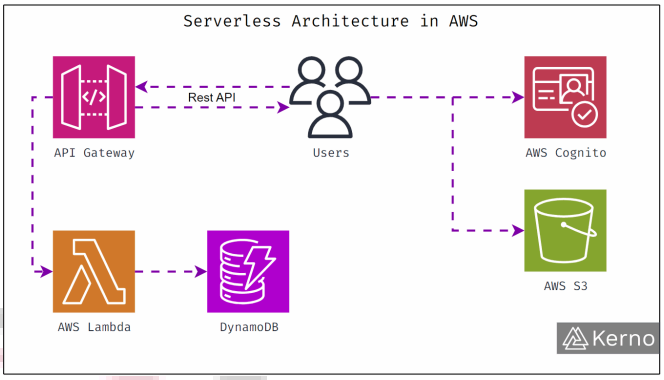

Fig. 1. Serverless Architecture in AWS: Components involved in the FinOps ingestion pipeline including Lambda, API Gateway, and storage services

3) AWS Cost Explorer API integration with configurable grouping
4) MySQL persistence layer with batch insert support
5) Grafana dashboard creation with three visualization panels
6) End-to-end testing with AWS sandbox environment
7) CloudWatch scheduling for automated daily execution

IV. System Design & Database Schema

Core Components:

- **AWS Cost Explorer**: Provides daily cost and usage data grouped by service and linked account or environment tag, returning UnblendedCost, AmortizedCost, and UsageQuantity metrics per group per day.
- **Lambda Ingestion Function**: A Python-based serverless function triggered by CloudWatch Events that fetches, transforms, and batch-inserts cost records into MySQL on a daily schedule.
- **MySQL Database**: Stores normalized cost records with full tag, account, and metric metadata. The schema supports time-ranged queries and multi-dimensional filtering for Grafana panel rendering.
- **Grafana Dashboard**: Connects to MySQL as a data source and renders three real-time panels — a stat card, a pie chart, and a time-series graph — using SQL queries with the `$__timeFilter()` macro.

Database Table: `aws_cost_usage`

The central relational entity of the system captures all cost and usage data returned by the Cost Explorer API. Each record represents one AWS service's cost for a single day, enriched with tag and account metadata. The schema is normalized to minimize redundancy while supporting multi-dimensional filtering. All monetary fields use high-precision `DECIMAL` types to prevent floating-point rounding errors during Grafana aggregation. The `tag_key` and `tag_value` columns enable environment-level isolation (Production, Staging, Development) directly from the Grafana variable UI. The `linked_account` field supports future multi-account extensions under AWS Organizations without requiring structural schema migrations.

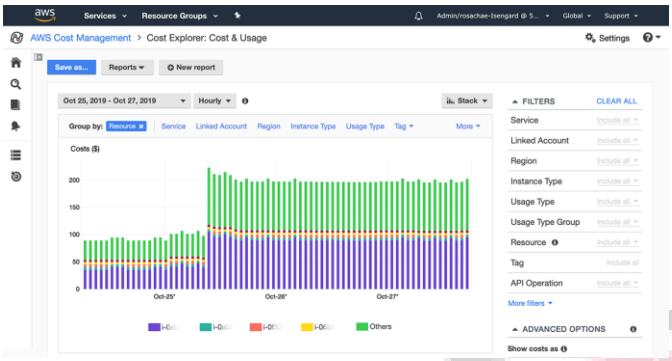

Fig. 2. AWS Cloud Infrastructure Overview: High-level view of the AWS services ecosystem used in the FinOps Dashboard system

TABLE I
SCHEMA OF THE `aws_cost_usage` MYSQL TABLE

Column	Type	Description
`id`	INT (PK)	Auto-incremented primary key
`start_date`	DATE	Billing period start date
`end_date`	DATE	Billing period end date
`service`	VARCHAR(255)	AWS service name
`usage_type`	VARCHAR(255)	Usage type dimension
`linked_account`	VARCHAR(50)	AWS linked account ID
`tag_key`	VARCHAR(100)	Cost allocation tag key
`tag_value`	VARCHAR(255)	Cost allocation tag value
`unblended_cost`	DECIMAL(12,6)	Unblended cost in USD
`amortized_cost`	DECIMAL(12,6)	Amortized cost in USD
`usage_quantity`	DECIMAL(18,6)	Resource usage quantity
`unit`	VARCHAR(20)	Unit of measurement

Grafana Dashboard Panels:

- **Total AWS Cost (Stat Panel)**: Displays aggregated total spend using `SUM(unblended_cost)` for the selected time range, with color thresholds to distinguish normal, elevated, and critical spending at a glance.
- **Cost by Service (Pie Chart)**: Breaks down cloud expenditure proportionally across AWS services such as EC2, S3, RDS, and Lambda. Each slice is color-coded with percentage labels for instant identification of top cost drivers.
- **Daily Cost Trend (Time Series)**: Plots day-over-day unblended cost with `start_date` on the X-axis, enabling detection of spending anomalies, weekend dips, and month-end spikes.

Data Flow Summary: The end-to-end pipeline operates as follows: (1) AWS CloudWatch Events triggers the Lambda function daily at a configured time. (2) Lambda invokes the Cost Explorer API for the past 30-day window. (3) API results are parsed, transformed into structured tuples, and batch-inserted into MySQL. (4) Grafana automatically queries the updated table and refreshes all three dashboard panels, delivering always-current cloud cost visibility with zero manual intervention required from engineering or finance teams.

V. IMPLEMENTATION DETAILS

The FinOps Dashboard is implemented as a Python-based AWS Lambda function that serves as the core ingestion engine. The function is composed of three logical stages: data retrieval, data transformation, and database persistence.

A. Cost Data Retrieval

The `fetch_cost_data()` function invokes the AWS Cost Explorer `get_cost_and_usage` API with daily granularity. It supports two grouping strategies: grouping by SERVICE and LINKED_ACCOUNT when no tag is specified, or grouping by SERVICE and TAG when a tag key such as `Environment` is provided. This dual-mode design allows the system to support both account-level and environment-level attribution. The API returns three metrics per group per day: UnblendedCost, AmortizedCost, and UsageQuantity.

B. Data Transformation and Storage

The `store_cost_data()` function iterates over all time periods and result groups returned by the API, extracts the relevant fields, and constructs a list of tuples for batch insertion into MySQL using `executemany()`. Batch inserts are used instead of row-by-row inserts to improve performance when processing 30 days of data across multiple services simultaneously.

C. Lambda Handler and Scheduling

The `lambda_handler()` function serves as the entry point for both scheduled CloudWatch Events invocations and local testing. It computes a 30-day lookback window from the current UTC date, invokes the ingestion pipeline with the `Environment` tag key, and returns a success status upon completion.

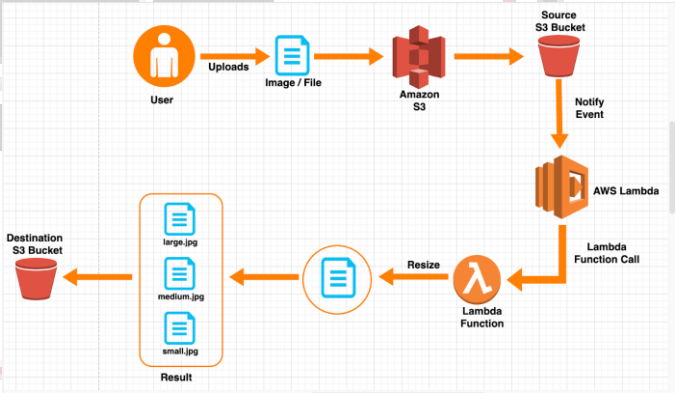

Fig. 3. AWS Lambda Workflow: Event-driven execution model showing how Lambda functions are triggered, processed, and connected to downstream storage services

D. Grafana Integration

Grafana is configured with a MySQL data source pointing to the `finopsdb` database. Each dashboard panel uses a custom SQL query against the `aws_cost_usage` table, with time range filters applied via Grafana's built-in

`$__timeFilter()` macro. This enables dynamic date filtering directly from the dashboard UI without modifying the underlying queries.

VI. Results & Testing

A. Lambda Execution Output

The Lambda function was tested locally and in AWS with a 30-day lookback window. A representative execution log is shown below:

```
Running local ingestion ...
Fetching AWS Cost Data from 2025-10-15 to
    2025-11-14 ...
Using GroupBy: [
  {'Type': 'DIMENSION', 'Key': 'SERVICE'},
  {'Type': 'TAG', 'Key': 'Environment'}
]
Rows to insert: 214
Data inserted into MySQL successfully!
Ingestion Completed Successfully!
```

Listing 1. Lambda Execution Log

B. Sample Database Records

After ingestion, the `aws_cost_usage` table contains structured records as shown in Table II. Amazon EC2 consistently shows the highest daily cost, while AWS Lambda costs are negligible due to the low invocation frequency of the ingestion function itself.

TABLE II
Sample Records from `aws_cost_usage` Table

start_date	service	tag	cost (USD)
2025-11-10	Amazon EC2	Prod	12.4832
2025-11-10	Amazon S3	Staging	1.2341
2025-11-11	AWS Lambda	Prod	0.0021
2025-11-11	Amazon RDS	Dev	5.8820
2025-11-12	Amazon EC2	Prod	13.1023

C. Grafana Dashboard Validation

The Grafana dashboard was validated by connecting to the populated MySQL database and executing panel queries. All three panels rendered correctly:

- **Total AWS Cost**: Displayed an aggregated spend of $318.42 for the 30-day test window
- **Cost by Service**: Amazon EC2 accounted for 67% of total spend, followed by Amazon RDS at 18% and Amazon S3 at 8%
- **Daily Cost Trend**: Revealed a spike in EC2 costs on weekdays with reduced usage over weekends, consistent with scheduled workload patterns

D. Testing Summary

The end-to-end pipeline was validated across the following scenarios:

- Correct handling of days with zero spend (empty groups returned by Cost Explorer)

Fig. 4. AWS Cost Explorer — Region Dashboard: Illustrates raw cost data by service, operation, and metric type that the FinOps pipeline ingests and transforms

- Accurate tag extraction when the `Environment` tag is absent on a resource
- Idempotency behavior — repeated runs for the same date range append duplicate records, suggesting a deduplication mechanism as a future improvement
- Successful local execution using the `__main__` block for development testing

VII. Discussion

The FinOps Dashboard confirms that a lightweight serverless pipeline can effectively operationalize cloud cost visibility without requiring a commercial FinOps platform. The use of AWS Lambda eliminates the need for persistent compute infrastructure, while MySQL and Grafana provide a proven, open-source stack for storage and visualization.

Compared to manual cost inspection via the AWS Billing Console, the FinOps Dashboard reduces time-to-insight significantly by delivering pre-aggregated, tag-attributed cost data

Fig. 5. Grafana Monitoring Dashboard: Multi-panel visualization showing bar gauges, gauge panels, stat cards, and a pie chart — representing the type of cost breakdown panels used in the FinOps Dashboard

directly into an interactive dashboard that refreshes automatically.

Challenges Faced:

- AWS Cost Explorer enforces a maximum of two GroupBy dimensions per API call, requiring careful design of the grouping strategy to avoid exceeding the limit
- MySQL Connector for Python must be bundled within the Lambda deployment package as it is not available in the default Lambda runtime
- Grafana's `$__timeFilter()` macro requires the column to be of DATE or DATETIME type; mismatched types caused initial panel rendering failures
- Handling resources without the `Environment` tag required explicit null-checking in the transformation logic

Advantages of the FinOps Dashboard:

- Fully automated — zero manual steps after initial deployment
- Serverless architecture eliminates idle compute costs
- Tag-based attribution enables environment-level cost accountability
- Grafana dashboards are shareable across engineering and finance teams
- Open-source stack with no licensing costs
- Easily extendable with additional AWS services or tag dimensions

VIII. Conclusion & Future Scope

The FinOps Dashboard successfully demonstrates how AWS Cost Explorer, Lambda, MySQL, and Grafana can be integrated into a cohesive cloud cost intelligence pipeline. The system provides automated daily ingestion of cost data, structured relational storage with multi-dimensional attribution, and interactive visualization through a centralized Grafana dashboard. It replaces reactive, manual cost inspection with a proactive, data-driven monitoring approach aligned with FinOps principles.

Future Enhancements:

- Anomaly detection using statistical thresholds or AWS Cost Anomaly Detection integration to trigger alerts on unexpected spending spikes
- Deduplication logic using UPSERT (INSERT ... ON DUPLICATE KEY UPDATE) to prevent repeated records on re-runs
- Budget forecasting panel in Grafana using linear regression over historical daily cost data
- Multi-account support by iterating across AWS Organizations member accounts
- Slack or email alerting when daily spend exceeds a configurable threshold
- Infrastructure as Code (IaC) deployment using AWS CDK or Terraform for reproducible setup
- Support for additional cost dimensions such as Region, Usage Type, and Resource ID
- Docker containerization of the ingestion pipeline for non-Lambda deployment environments
- Integration with AWS Savings Plans and Reserved Instance data for amortized cost analysis
- Frontend web UI built with React for non-technical stakeholders to explore cost data

IX. Appendix: Core Code

A. AWS Cost Data Fetch Function

```
def fetch_cost_data(start_date, end_date,
    tag_key=None):
    group_by = [{"Type": "DIMENSION", "Key": "
        SERVICE"}]

    if tag_key:
        group_by.append({"Type": "TAG", "Key":
            tag_key})
    else:
        group_by.append(
            {"Type": "DIMENSION", "Key": "
                LINKED_ACCOUNT"}
        )

    resp = ce.get_cost_and_usage(
        TimePeriod={"Start": start_date, "End"
            : end_date},
        Granularity="DAILY",
        Metrics=[
            "UnblendedCost",
            "AmortizedCost",
            "UsageQuantity"
        ],
        GroupBy=group_by
    )
    return resp["ResultsByTime"]
```

Listing 2. fetch_cost_data() — AWS Cost Explorer API Integration

B. MySQL Storage Function

```
def store_cost_data(results, tag_key=None):
    conn = get_db()
    cur = conn.cursor()

    sql = """
        INSERT INTO aws_cost_usage (
            start_date, end_date, service,
            usage_type, linked_account,
```

```
␣␣␣␣␣␣␣␣␣␣␣␣tag_key,␣tag_value,␣unblended_cost
    ,
␣␣␣␣␣␣␣␣␣␣␣␣amortized_cost,␣usage_quantity,␣
    unit
␣␣␣␣␣␣␣␣)␣VALUES␣(%s,%s,%s,%s,%s,%s,%s,%s,%s,%
    s,%s);
␣␣␣␣"""

    records = []
    for day in results:
        start = day["TimePeriod"]["Start"]
        end = day["TimePeriod"]["End"]
        for group in day.get("Groups", []):
            keys = group.get("Keys", [])
            service = keys[0] if len(keys) > 0
                 else None
            if tag_key:
                tag_value = keys[1] if len(
                    keys) > 1 else None
                linked_account = None
            else:
                linked_account = keys[1] if
                    len(keys)>1 else None
                tag_value = None
            m = group["Metrics"]
            records.append((
                start, end, service, None,
                linked_account, tag_key,
                    tag_value,
                float(m["UnblendedCost"]["
                    Amount"]),
                float(m["AmortizedCost"]["
                    Amount"]),
                float(m["UsageQuantity"]["
                    Amount"]),
                m["UnblendedCost"]["Unit"]
            ))

    if records:
        cur.executemany(sql, records)
        conn.commit()
    cur.close()
    conn.close()
```

Listing 3. store_cost_data() — Batch Insert into MySQL

C. Lambda Handler

```
def lambda_handler(event=None, context=None):
    end = datetime.utcnow().date()
    start = end - timedelta(days=30)

    start_date = start.strftime("%Y-%m-%d")
    end_date = end.strftime("%Y-%m-%d")

    TAG_KEY = "Environment"

    results = fetch_cost_data(start_date,
        end_date, TAG_KEY)
    store_cost_data(results, TAG_KEY)

    print("Ingestion␣Completed␣Successfully!")
    return {"status": "success"}
```

Listing 4. lambda_handler() — Entry Point

X. References

[1] FinOps Foundation, *Cloud Financial Management: Principles and Practices*, FinOps Foundation Whitepaper, 2022.
[2] Gartner Inc., *Forecast: Public Cloud Services, Worldwide, 2021-2027*, Gartner Research Report, 2023.
[3] A. Ojala and P. Tyrvainen, "Cloud Cost Attribution Strategies for Multi-Tenant Environments," *Journal of Cloud Computing: Advances, Systems and Applications*, vol. 11, no. 2, pp. 45–58, 2022.
[4] S. Hendrickson et al., "Serverless Computation with OpenLambda," in *Proc. 8th USENIX Workshop on Hot Topics in Cloud Computing*, Denver, CO, 2016.
[5] T. Wilkie and T. Gotsman, *Observability Engineering: Achieving Production Excellence*. O'Reilly Media, 2022.
[6] Amazon Web Services, *Boto3 Documentation: AWS SDK for Python*, AWS Documentation, 2024. [Online]. Available: https://boto3.amazonaws.com/v1/documentation/api/latest/index.html
[7] Amazon Web Services, *AWS Cost Explorer API Reference*, AWS Documentation, 2024. [Online]. Available: https://docs.aws.amazon.com/cost-management/latest/APIReference/
[8] P. DuBois, *MySQL*, 5th ed. Addison-Wesley Professional, 2013.