

Application Performance Monitoring

Assignment 3 Report

ECE1779 – Introduction to Cloud Computing

Gilkara Pranav Naidu - 1005055146

Jian Shi - 1001841536

Raghavendran Thiruvengadam - 1004919993

Shi Hu – 1001416498

Introduction

An Application Performance Monitoring (APM) tool is one of the most important tools in an arsenal of any software company. Licenses to leading APM tools usually cost software companies a huge fortune, however, these tools promise a great return in investment by providing valuable insight on the Performance, Failures and root causes of the failures for all their Software Product. We aim to design and analysis the cost involved in developing an APM tool. Some of the key features our APM tool provides are as under:

- **Privilege based access to features:** Company's administrators provide access level to all employees based on which they can use certain features of the application.
- **Metric Browser:** Users can monitor the metric values of all the components (nodes) of their application.
- **Dashboarding:** Widgets seen on the metric browser can be pinned to company-wide common dashboards.
- **Health Rules & Alerting:** Health Rule for various metrics of a node can be set up and can be associated to custom actions or notifications.
- **REST API:** A company can add an extra layer of analytic solutions by using the REST API to fetch the metric data or the Event data of their application.

Section 1: Using the Application

We first deploy the application using Zappa with the command *zappa deploy dev*. A zappa settings file has already been created and can be reused. If any changes need to be made after deployment, run the command *zappa update dev*. Once the application is deployed and can be accessed through the link generated by zappa, we perform the following steps for a new company:

1. Register a new account. This automatically creates all the resources necessary and creates a root user with access level 3 (administrator level access) – who can give other users permissions.
2. The root user needs to register new users by adding them using the Users page in the web application. The root user can also specify what access level the new user can have.
3. Once an account is created, and the users have been given access by the root user, they can now login anytime. Based on their access level (1, 2, 3), their access to the web application will be limited. Access level 1 users can only monitor the agents (node PCs in our case). Access level 2 users can monitor the agents and also add health rules and create policies if required. Access level 3 users the ability to create new users in addition to all these permissions.

4. Users can create widgets that store specific agent information for quicker access. These widgets can be pinned to a dashboard, which is common within a company. All users see the same dashboard and have the ability to add widgets to the dashboard.
5. Alerts can be monitored using the Alerts page, where a log of all the alerts (including real time ones if applicable) are visible. To prevent clogging, only past 100 alerts are available. The older ones are permanently deleted.

Section 2: Architecture

This application primarily has two lambda functions – one of which hosts the main web application, and the other collects the CPU utilization, disk utilization and memory usage metrics from individual agents (computers and servers in our use-case). The lambda hosting the main web application is deployed with Zappa, and the secondary background lambda function is invoked manually by each agent at regular intervals.

- **Section 2.1: Primary Web Application**
 - **Trigger: HTTP requests (Zappa's automatic API gateway configuration)**
 - The primary web application allows account creation, logging in, and monitoring metrics. A user can access only certain pages based on their access level as defined by the root user (admin).
 - For registration, logging in and checking access levels, the application directly interacts with the DynamoDB to perform queries if required, and makes updates to the tables defined in Section 3.1.
 - The lambda also interacts with RDS whenever the monitoring page is accessed to display the metrics. Metrics from the last 30 minutes for the particular node and application is fetched from a MySQL database in the RDS.
 - The primary lambda does *not* directly interact with the secondary lambda function – which is supposed to be run on its own by each agent being monitored.
- **Section 2.2: Background Process**
 - **Trigger: Invoked by agent monitoring scripts**
 - A script runs on each of the agents which collects the three metrics required every minute and then *invokes the secondary lambda function*.
 - The data sent to the lambda includes information such as: timestamp, average CPU utilization, disk utilization, memory usage, and agent name.
 - This lambda gathers the metrics, and *pushes them into a MySQL database* in RDS, using the appropriate agent name. The details of this database are given in **Section X**.
 - Once these metrics are stored in the RDS, the lambda then *accesses the 'policies' table in DynamoDB* for that particular agent, and checks if any of the defined health rules for that agent are violated.
 - If any health rule is violated, the lambda function then stores this violation (also called an event in our case) into the *'Events' table in the RDS* – which is then used

by the primary lambda function to display the list of violated policies along with their timestamps and information about which agent was affected.

- By making the secondary lambda access the RDS and Dynamo, we are effectively increasing security by not giving the agent access to the databases.

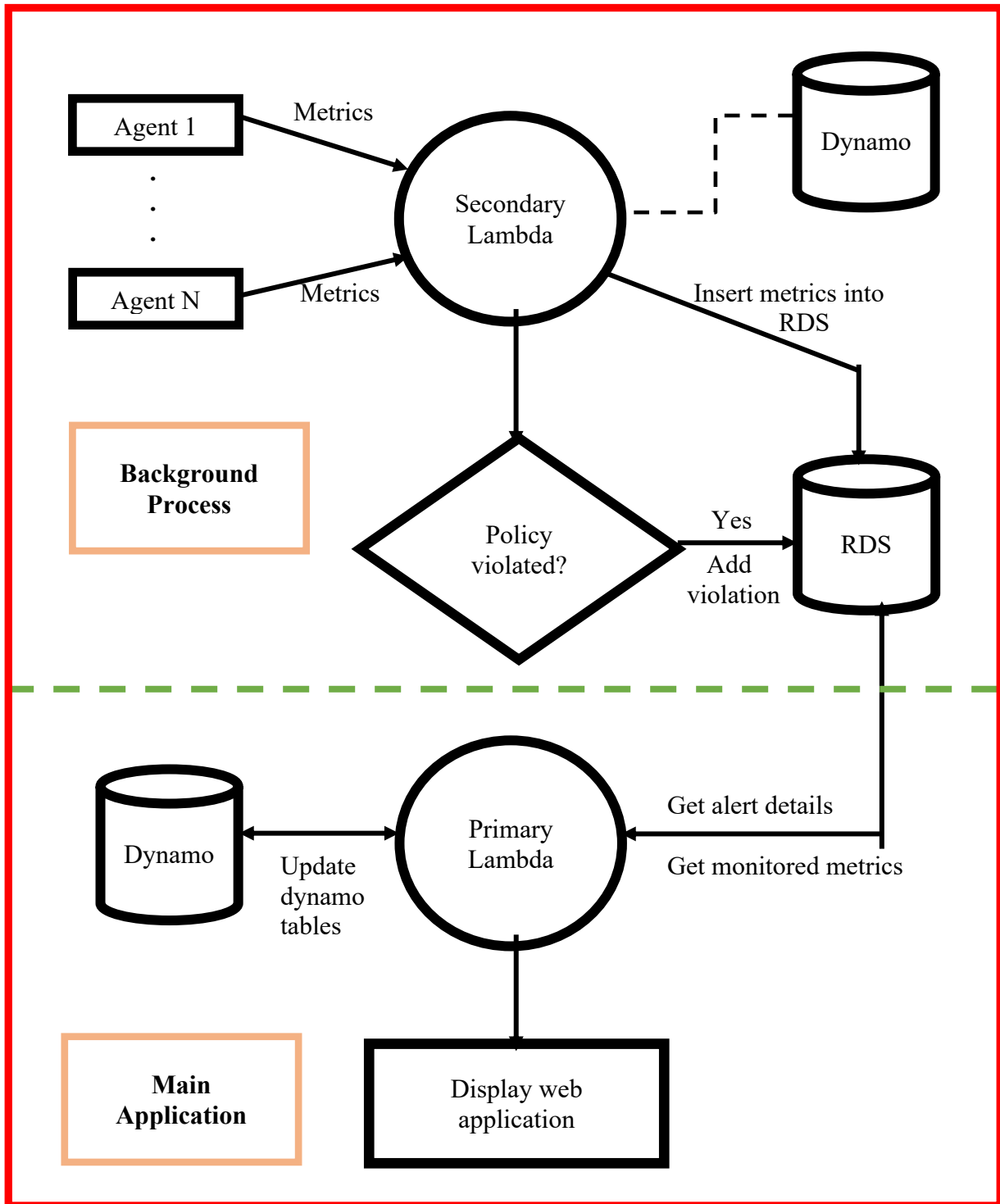


Fig1. Application Architecture

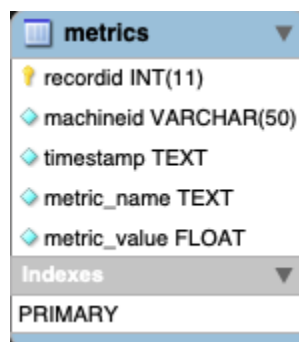
Section 3: Database

- **Section 3.1: DynamoDB**

- DynamoDB is used for storing new accounts, and the tables they require. For example, a company with name 'XYZ' will have the following tables created when the account is first registered:
 - XYZ_users: Stores user information and access level
 - XYZ_applications: Stores application details
 - XYZ_nodes: Stores node info
 - XYZ_alerts: Stores all the health rules
 - XYZ_action_group: Stores the corresponding action group for an alert
 - XYZ_policies: Stores policies created by users
 - XYZ_widgets: Stores widget ID for individual users
 - XYZ_dashboards: Stores dashboard ID for company
 - XYZ_events: Stores a log of past alerts and events
- Both the primary and secondary lambda make use of DynamoDB. The primary application registers new users, creates new alerts, health rules and policies, and reads the events log constantly to display past and present alerts.
- The secondary lambda uses DynamoDB to retrieve the alerts related to that particular node, and check if the health rules have been violated. If they have, then it updates the events table with a new entry that describes what was alerted and which node triggered the alert.

- **Section 3.2: RDS**

- Besides DynamoDB, we still use RDS to store some data since it allows us to run powerful SQL queries against columns that are not primary keys, which is not feasible in DynamoDB. Therefore, using RDS could provide the application more flexibility and allow more features to be easily implemented.



The screenshot shows the structure of an RDS database table named 'metrics'. It lists five columns: 'recordid' (INT(11)), 'machineid' (VARCHAR(50)), 'timestamp' (TEXT), 'metric_name' (TEXT), and 'metric_value' (FLOAT). Below the columns, there is an 'Indexes' section showing a 'PRIMARY' index.

Column	Data Type
recordid	INT(11)
machineid	VARCHAR(50)
timestamp	TEXT
metric_name	TEXT
metric_value	FLOAT

Index
PRIMARY

Figure 2. RDS structure

recordid	machineid	timestamp	metric_name	metric_value
3534	shi_pc	26265575	disk_util	49.4
3535	raghav_pc	26265637	cpu_util	24.6
3536	raghav_pc	26265637	mem_util	75.9
3537	raghav_pc	26265637	disk_util	8.1
3538	raghav_pc	26265668	cpu_util	24
3539	raghav_pc	26265668	mem_util	76.1
3540	raghav_pc	26265668	disk_util	8.1
3541	shi_ec2_1	26265737	cpu_util	100
3542	shi_ec2_1	26265737	mem_util	56.1
3543	shi_ec2_1	26265737	disk_util	87

Figure 3. A snapshot of table “metrics”

- Specifically, we maintain only one table called “metrics” in RDS, which stores metrics collected from each node. As shown in **Figure 3**, it has five columns. Column “recordid” is the primary key and the number would be automatically incremented every time a new entry is inserted. Column “machineid” denotes from which node a metric entry is collected. Column “timestamp” stores a value to indicate when a metric record is collected.
- In order to make plotting graphs of metric values more straightforward, a timestamp is stored in a format of UNIX time rounded to minutes. Column “metric_name” denotes which metric is collected. Currently, we have CPU, memory and disk utilization. Moreover, the design of the table would allow other kinds of metrics to be added if needed in the future. The last column
- “metric_value” basically stores a value of a metric. A snapshot of how different metrics of different nodes are stored is shown in **Figure 3**.
- The RDS table is used in a few places. As mentioned in **Section 2**, RDS is connected to both our main and secondary lambda functions. In the main application, RDS is mainly used to retrieve data to plot graphs of metrics per node. As I mentioned above, by using RDS, we could simply run a SQL query to select a specific metric of a specific node in the past 30 minutes without making them the primary keys. This would make the plotting much more efficient. Otherwise, the application may need to scan a whole table in the DynamoDB and do more manipulations to fetch useful data.
- The secondary lambda function basically uses RDS in two ways. First, it simply takes metric values collected in each node and inserts them into the table. The consideration is that we only need to maintain one connection between the secondary lambda function and the RDS rather than many connections between nodes and RDS. Second, the lambda function reads a SQL query from the “alert” table in DynamoDB and run it against RDS to identify whether there is an alert. Again, by using RDS here, we could simply design SQL queries of different conditions for different alerts. By simply running them on RDS, the application would know whether alerts happen. Here, the convenience and versatility are what DynamoDB could not provide.

Section 4: Cost Model

- **Section 4.1:** Based on number of users per company
 - For the cost model, the resources used are:
 - DynamoDB with on-demand pricing
 - AWS RDS with a t2.micro database
 - S3 bucket (which will never exceed free tier limit)
 - AWS Lambda (2 functions, one primary and one secondary)
 - The assumptions made for the cost model based on number of users a company gives access to are:
 - A user makes at most 1000 requests per day with the web application
 - A user makes at most 1000 write requests to the DynamoDB
 - A user makes at most 5000 read requests to the DynamoDB
- **Section 4.2:** Based on number of agents per company
 - The assumptions made for the cost model based on number of agents a company monitors are:
 - Each agent invokes secondary lambda every minute
 - Secondary lambda has memory of 150KB allotted to it
 - Each agent makes 1 write to DynamoDB (one for events) every minute
 - Each agent makes at most 25 reads to DynamoDB every minute

Daily Cost						
1 company – user cost						
Service/Users	1	10	100	1000	1000000	Notes about pricing
RDS (\$)	0.408	0.408	0.408	0.408	0.408	Single database cost - includes agents
S3 (\$)	0.023	0.023	0.023	0.023	0.023	Buckets do not store much, read/write costs are negligible
Lambda (\$)	0.014	0.14	1.4	14	14000	Scaled linearly with number of users -agent independent
Dynamo (\$)	0.00137	0.0137	0.137	1.37	1370	Scaled linearly - independent of agents
Total cost (\$)	0.44637	0.5847	1.968	15.801	15370.431	NA

Table 1: Cost model for various number of users

6 Months Cost						
1 company – user cost						
Service/Users	1	10	100	1000	1000000	Notes about pricing
Users	5.3526	30.252	279.246	2769.186	2766602.59	All costs not considering free tier

Table 2: Cost model for 6 months for various number of users

Daily Cost						
1 company –agent cost						
Service/Agents	1	10	100	1000	1000000	Notes about pricing
Lambda	0.052	0.52	5.2	52	52000	Shares S3 and RDS with user costs
Dynamo	0.00936	0.0936	0.936	9.36	9360	Shares S3 and RDS with user costs
Total Cost	0.06136	0.6136	6.136	61.36	61360	NA

Table 3: Cost model for various number of agents

6 Months Cost						
1 company –agent cost						
Service/Agents	1	10	100	1000	1000000	Notes about pricing
Agents	11.0448	110.448	1104.48	11044.8	11044800	All costs not considering free tier

Table 4: Cost model for 6 months for various number of agents

Note: These numbers seem excessive because our use-case is slightly different. We do not expect a company to have more than 100 agents, or more than 1000 users in general.