PC Metrics

Project Title: PC Metrics
Name: Elad Lavi
ID: 205576440
Course: Internet of Things
Lecturer: Guy Tel-Zur

Abstract

The PC Metrics project aims to monitor the key thermal metrics of a gaming PC in real-time and leverage cloud services and AI to improve cooling performance. In this project, I developed a Python-based IoT system that simulates a gaming PC's temperature sensors and fan speeds, publishes the telemetry to AWS IoT Core, and stores it in Amazon Timestream. A real-time dashboard was created using Amazon Managed Grafana to visualize CPU/GPU temperatures and fan RPMs, with alert thresholds indicated. AWS CloudWatch alarms and SNS notifications were configured to proactively warn when component temperatures exceed safe limits. Additionally, an AI component (Anthropic's Claude model) was integrated to analyze temperature patterns and suggest optimized fan curves. The results were continuous live monitoring, successful alert triggering for high temperatures, and a proof-of-concept for AI-driven fan tuning. Key challenges addressed included ensuring secure MQTT connectivity with IoT certificates, and fine-tuning alert conditions. This report details the project implementation, outcomes, challenges, and potential future enhancements such as adding more sensors, user-facing portals, mobile alerts, and advanced machine learning for predictive analytics.

Project Objective

Modern high-end gaming PCs often struggle with **thermal management** during heavy workloads. Components like CPUs and GPUs can reach **dangerously high temperatures (80°C or more)** under intense gaming sessions, which can **degrade hardware over time** If while gaming those temps are sustained over long periods of time.
Gamers typically rely on manual fan curve adjustments to balance cooling and noise, but this is **tedious and inefficient**, and often results in suboptimal cooling profiles that either **run too hot or produce excessive fan noise**. Furthermore users lack historical data to understand trends or catch gradual thermal degradation.

Project Goal:

The goal of PC Metrics is to create an IoT-based solution for **real-time monitoring** of a gaming PC's temperatures and fan speeds via the cloud, and to provide **smart cooling recommendations** automatically. By streaming sensor data to the cloud, I can aggregate and analyze thermal telemetry, trigger alerts before damage occurs, and ultimately use AI to generate optimal fan curves tailored to the system's behavior.

Personally, I wanted to ensure my gaming rig (and others in the community) can be monitored proactively, receiving warnings when it overheats and even getting AI-driven advice to improve cooling performance.

In summary, the objectives of the project are:

Real-Time Monitoring: Continuously collect CPU, GPU, SSD, and motherboard temperatures, as well as fan speeds, and view them live on a dashboard.

Cloud Data Aggregation: Send the telemetry to a centralized cloud database, enabling historical data tracking and multi-device analysis (for example, comparing metrics across multiple PCs).

Preventative Alerts: Implement automated alerts (SMS/email) via cloud services when temperatures exceed safe thresholds, so action can be taken before thermal throttling or damage occurs.

AI-Driven Cooling Optimization: Utilize an AI model in the cloud to analyze the patterns of temperature and fan data and recommend improved fan speed curves (balancing cooling vs. noise) for optimal performance.

Implementation

To fulfill the above objectives, I designed a system composed of a **local simulator** and various **AWS cloud services** working in concert. The architecture follows a typical IoT pipeline: data generation at the edge, ingestion to the cloud, storage and visualization, alerting, and higher-level analysis.

Key implementation components:

PC Telemetry Simulator (Edge Device): I developed a Python script (`simulate_pc_metrics.py`) that mimics a gaming PC's sensor outputs in real time. This script uses a `PCComponentSimulator` class to model the behavior of components:

It simulates **gaming sessions** by randomly starting/stopping a "game" and gradually ramping the `gaming_intensity` variable from idle to full load and back. This affects heat generation : higher intensity leads to higher CPU/GPU temps, then a cooldown phase is simulated once the session ends.

Component temperatures (CPU, GPU, SSD, motherboard) are updated each cycle based on the current load using realistic rules with a certain rate of change and random noise to emulate real fluctuations. The code ensures temperatures never exceed defined max limits (CPU max 88°C) by capping the values.

Fan speeds (CPU fan, GPU fan, case fan) are calculated via preset fan curves relative to temperature. In the simulation, I defined baseline RPMs for idle and scaled them up as temperatures rise. For example, the GPU fan might be ~1000 RPM at idle and ramp to 4000+ RPM near 85°C. This provides a realistic relationship where higher temps cause higher fan speeds.

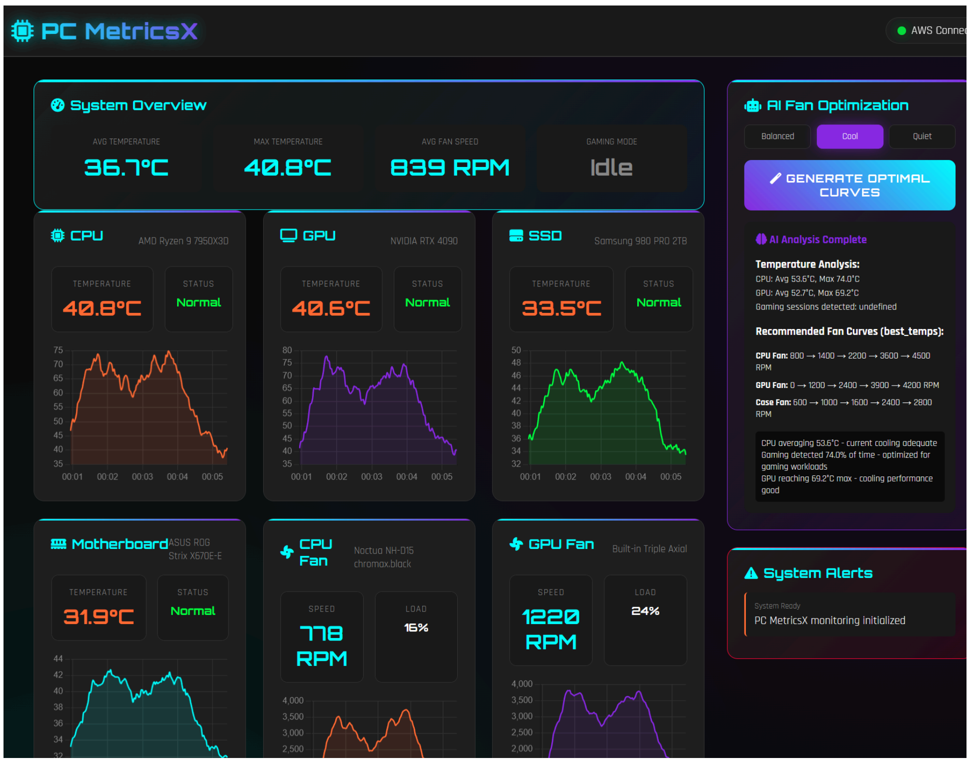
The simulator runs in an infinite loop (with 1-second intervals) where it updates the game state, recalculates temps and fan speeds, and then collects all sensor readings into a dictionary:

```
 while True:
    pc_simulator.update_gaming_session()
    pc_simulator.simulate_temperature_changes()
    pc_simulator.calculate_fan_speeds()
    sensor_data = pc_simulator.get_sensor_data()
    if aws_connected:
        aws_publisher.publish_data(sensor_data)
    time.sleep(config.SENSOR_UPDATE_INTERVAL)
```

This code runs continuously, and if the AWS IoT connection is active, each `sensor_data` JSON is immediately published to the cloud. The `get_sensor_data()` method packages readings like `cpu_temp`, `gpu_temp`, etc., along with a timestamp and device ID.
For example, a single JSON looks like:

```
 {
  "timestamp": 1750277013,
  "device_id": "GamingPC4",
  "cpu_temp": 41.2,
  "gpu_temp": 36.7,
  "ssd_temp": 35.3,
  "motherboard_temp": 32.9,
  "cpu_fan_rpm": 1165,
  "gpu_fan_rpm": 958,
  "case_fan_rpm": 747,
  "gaming_session": false,
  "gaming_intensity": 0.0
}
```

Local Dashboard (Flask Web App): For testing and demonstration, I also created a simple Flask web application (`app.py`) that serves a local dashboard. This dashboard uses Socket.IO to live-update charts in the browser with the latest sensor data and indicates whether the AWS cloud connection is active. It displays line graphs of the temperature and fan metrics and reads the same data stream (from a local JSON log) and even has an API endpoint to fetch current data as JSON. This component validated that the data generation was correct before sending to AWS.



System Overview
AVG TEMPERATURE 36.7°C
MAX TEMPERATURE 40.8°C
AVG FAN SPEED 839 RPM
GAMING MODE Idle

CPU AMD Ryzen 9 7950X3D
TEMPERATURE 40.8°C STATUS Normal

GPU NVIDIA RTX 4090
TEMPERATURE 40.6°C STATUS Normal

SSD Samsung 980 PRO 2TB
TEMPERATURE 33.5°C STATUS Normal

Motherboard ASUS ROG Strix X670E-E
TEMPERATURE 31.9°C STATUS Normal

CPU Fan Noctua NH-D15 chromax.black
SPEED 778 RPM LOAD 16%

GPU Fan Built-in Triple Axial
SPEED 1220 RPM LOAD 24%

AI Analysis Complete
Temperature Analysis:
CPU: Avg 53.6°C, Max 74.0°C
GPU: Avg 52.7°C, Max 69.2°C
Gaming sessions detected: undefined

Recommended Fan Curves (best_temps):
CPU Fan: 800 → 1400 → 2200 → 3600 → 4500 RPM
GPU Fan: 0 → 1200 → 2400 → 3900 → 4200 RPM
Case Fan: 600 → 1000 → 1600 → 2400 → 2800 RPM

CPU averaging 53.6°C - current cooling adequate
Gaming detected 74.0% of time - optimized for gaming workloads
GPU reaching 69.2°C max - cooling performance good

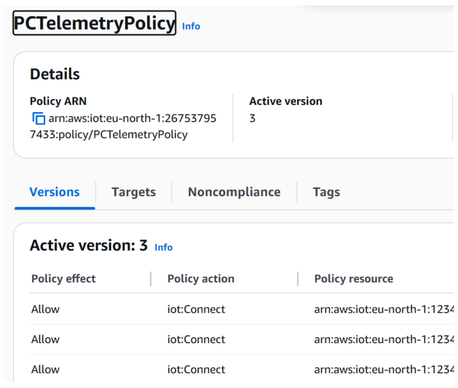
System Alerts
System Ready
PC MetricsX monitoring initialized

AWS IoT Core (MQTT Broker): AWS IoT Core is the cloud gateway for my IoT device (the PC simulator). I configured the simulator as an IoT "Thing" and set up mutual authentication using X.509 certificates. In the code, an `AWSIoTPublisher` class wraps the AWS IoT Device SDK, loading my AWS IoT Core **endpoint URL** and the certificate file paths from a config file. On startup, the simulator attempts to connect to IoT Core:

```
aws_publisher = AWSIoTPublisher()
aws_connected = aws_publisher.connect()
if aws_connected:
    print("SUCCESS: AWS IoT integration active")
```

The `connect()` method configures the MQTT client with the endpoint, port 8883, root CA, client certificate, and private key, then calls `connect()` to establish a TLS connection. If successful, the device is online on AWS IoT.

I had to ensure the **IoT policy** attached to the certificate allowed MQTT publish/subscribe on the topic.


PCTelemetryPolicy Info

Details

Policy ARN: arn:aws:iot:eu-north-1:267537957433:policy/PCTelemetryPolicy

Active version: 3

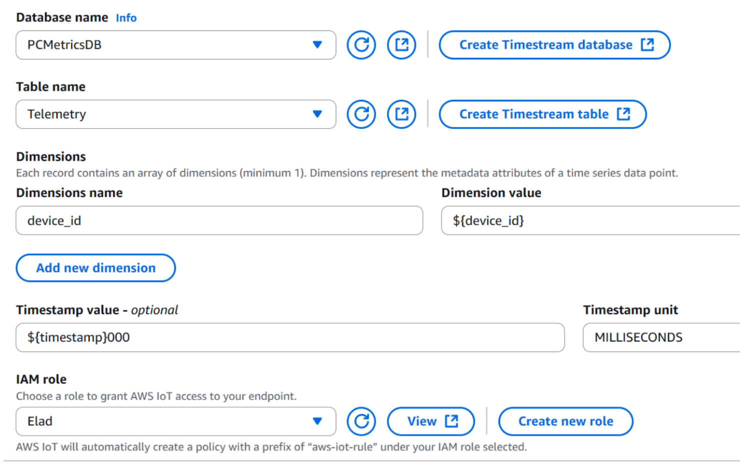
Versions | Targets | Noncompliance | Tags

Active version: 3 Info

Policy effect	Policy action	Policy resource
Allow	iot:Connect	arn:aws:iot:eu-north-1:1234
Allow	iot:Connect	arn:aws:iot:eu-north-1:1234
Allow	iot:Connect	arn:aws:iot:eu-north-1:1234

MQTT Topic: I defined the topic as `gamingPC/telemetry`. The simulator publishes each JSON payload to this topic using QoS 1 for reliable delivery. For example, it would publish to `gamingPC/telemetry` with the JSON shown above. On AWS IoT Core, I created a rule to route this incoming data to other services.

Amazon Timestream (Time-Series Database): To store and query the telemetry data, I used Amazon Timestream. I created a database and table to tag each record. The AWS IoT Rule was configured with an SQL statement to select all fields from the MQTT topic and insert them into the Timestream table. I used `SELECT * FROM 'gamingPC/telemetry'` as the rule query, which means every attribute of the JSON is stored. The rule's Timestream action specifies the database/table and uses the JSON `device_id` as a dimension (so data can be filtered by device). As a result, each telemetry JSON from the simulator is ingested as a new time-series data point in Timestream, with attributes like cpu_temp, gpu_temp, etc. recorded along with a timestamp.


Database name Info: PCMetricsDB

Create Timestream database

Table name: Telemetry

Create Timestream table

Dimensions

Each record contains an array of dimensions (minimum 1). Dimensions represent the metadata attributes of a time series data point.

Dimensions name	Dimension value
device_id	${device_id}

Add new dimension

Timestamp value - *optional*	Timestamp unit
${timestamp}000	MILLISECONDS

IAM role

Choose a role to grant AWS IoT access to your endpoint.

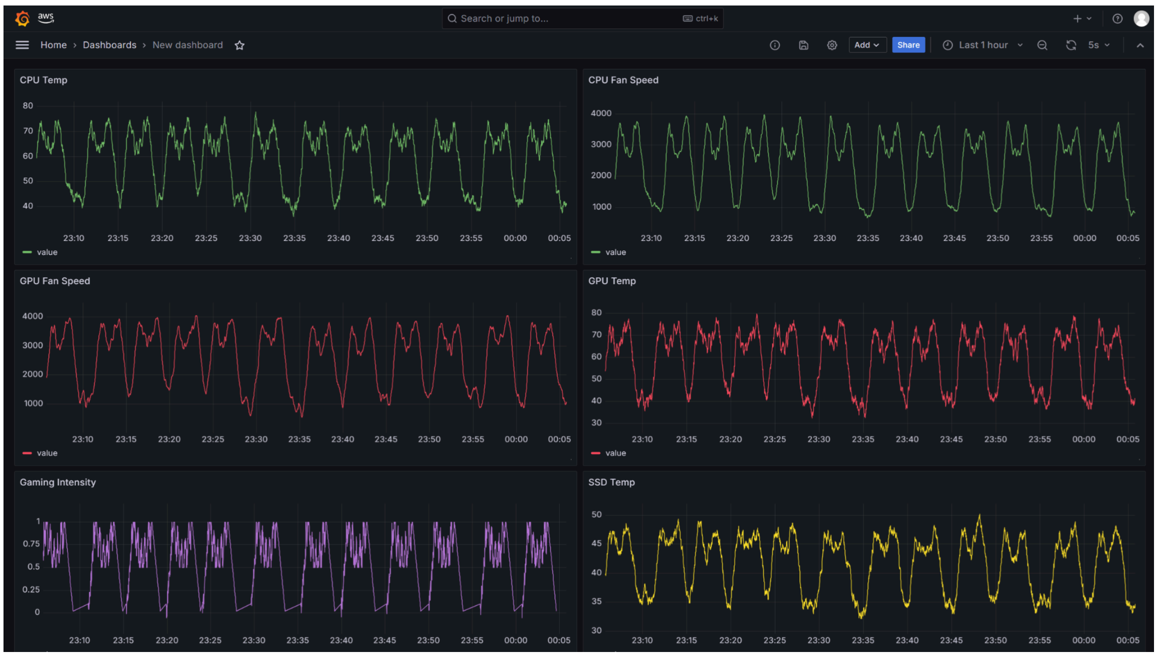
Elad

View | Create new role

AWS IoT will automatically create a policy with a prefix of "aws-iot-rule" under your IAM role selected.

Amazon S3 (Data Lake Backup): In addition to Timestream, the IoT rule also sends data to Amazon S3 for long-term storage. I set up an S3 bucket where the raw JSON payloads are dumped. Did this in order to have a durable archive of all telemetry in case I want to do offline analysis or if Timestream retention is limited. This was implemented by adding a second action in the IoT rule: an S3 put object action, which automatically deposits the message into a specified bucket/prefix. This is ensuring no data is lost.

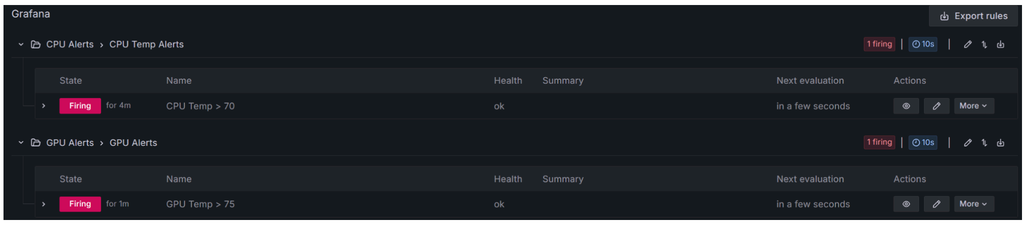
Amazon Managed Grafana (Dashboard & Visualization): To visualize the incoming data in real time, I used Amazon Managed Grafana with the Timestream plugin. I created a Grafana workspace in the same AWS region and added the Timestream data source, pointing it to the database/table where telemetry was stored. I then built a dashboard with multiple panels:



CPU,GPU,SSD,Motherboard Temperature&Fan graphs showing how the cooling reacts to the temperature changes.

A text or gauge panel for **Gaming Intensity**, which reflects the current simulated load (0 to 1 scale). This was more for interest, to see correlation of intensity to temps.

An **“Alert Panel”** that lists any active alerts (for example, “CPU Temp High” when triggered).



Grafana

Export rules

CPU Alerts › CPU Temp Alerts — 1 firing | 10s

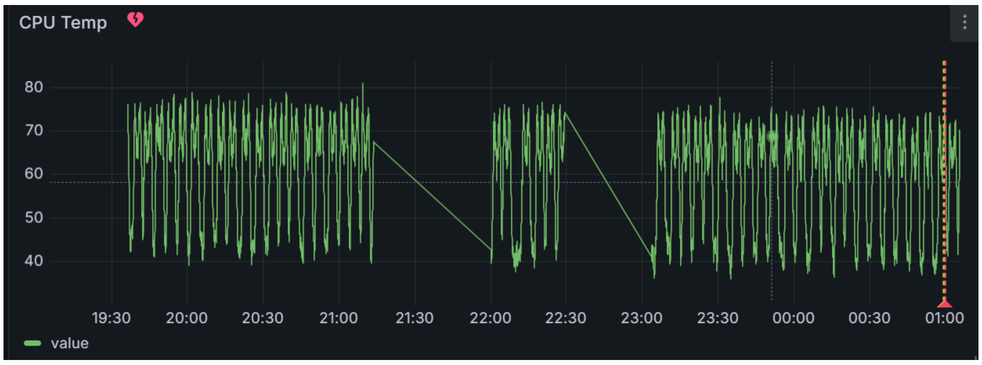
State	Name	Health	Summary	Next evaluation	Actions
Firing for 4m	CPU Temp > 70	ok		in a few seconds	More

GPU Alerts › GPU Alerts — 1 firing | 10s

State	Name	Health	Summary	Next evaluation	Actions
Firing for 1m	GPU Temp > 75	ok		in a few seconds	More

Grafana is well-suited for real-time IoT dashboards : it supports auto-refresh and can pull new data from Timestream as it arrives. In the dashboard, I set the refresh interval to 5 seconds and the time range to last 5 minutes by default, so effectively get a live feed.

In terms of integration, I had to set up AWS IAM roles to allow Grafana to read Timestream data. AWS Managed Grafana uses a service role that I configured with Timestream read permissions. Also, I ensured all resources (IoT, Timestream, Grafana) were in **us-east-1** region to avoid cross-region data issues and because TimeStream was accessible in this region.



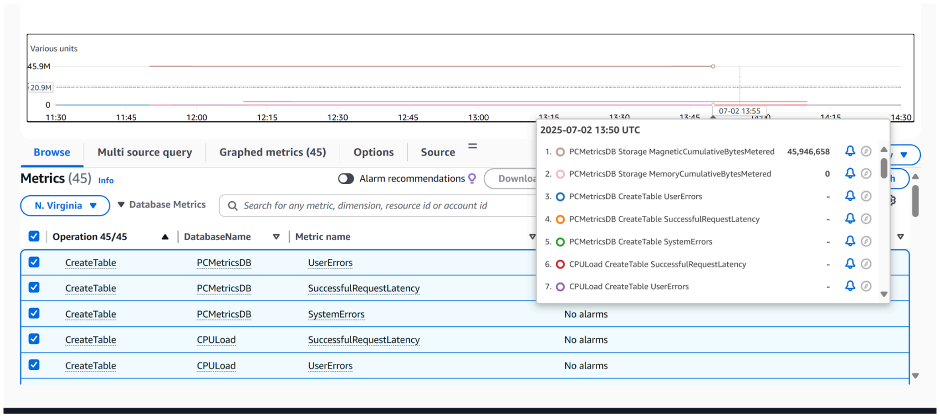
AWS CloudWatch Alarms-Monitoring & Alert (Lambda & SNS): For the **alerting system**, I leveraged Amazon CloudWatch to monitor the telemetry and trigger notifications.

I Used an AWS IoT rule (or a small Lambda function subscribed to the IoT topic) to publish custom CloudWatch metrics for each temperature. Whenever a new message arrives, it could push the `cpu_temp` value to a CloudWatch metric (e.g., Metric name "CPU_Temperature" with dimension device=GamingPC4).

Then, in CloudWatch, create **alarms** on those metrics. I set threshold conditions such as *CPU_Temperature > 75 for 2 consecutive minutes -> ALARM*. Similar alarms for GPU_Temperature > 75, SSD_Temperature > 80, and Motherboard_Temp > 80.

CloudWatch alarms were configured to perform an action when triggered: specifically, to **send a notification to an Amazon SNS topic**. AWS CloudWatch can easily hook into SNS for alarm actions. I created an SNS topic (e.g., "PCMetricsAlerts") and subscribed my email to it for notifications. Thus, when an alarm goes to ALARM state, SNS sends out an email with the alert details. For example, if during a test the CPU stayed above 75°C for over 120 seconds, the "CPU_High_Temp" alarm would trigger and I'd receive an email saying *"ALARM: CPU_High_Temp in state ALARM (CPU_Temperature > 75°C)"*.

The use of CloudWatch/SNS ensures proactive alerting, as required, essentially functioning as the project's "smoke alarm" for the PC.



2025-07-02 13:50 UTC

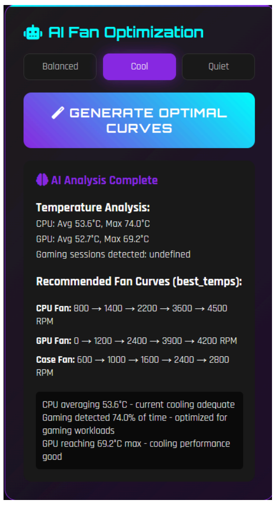
1. PCMetricsDB Storage MagneticCumulativeBytesMetered 45,946,658
2. PCMetricsDB Storage MemoryCumulativeBytesMetered 0
3. PCMetricsDB CreateTable UserErrors -
4. PCMetricsDB CreateTable SuccessfulRequestLatency -
5. PCMetricsDB CreateTable SystemErrors -
6. CPULoad CreateTable SuccessfulRequestLatency -
7. CPULoad CreateTable UserErrors -

Operation 45/45	DatabaseName	Metric name	
CreateTable	PCMetricsDB	UserErrors	
CreateTable	PCMetricsDB	SuccessfulRequestLatency	
CreateTable	PCMetricsDB	SystemErrors	No alarms
CreateTable	CPULoad	SuccessfulRequestLatency	No alarms
CreateTable	CPULoad	UserErrors	No alarms

AI Integration : Claude Model for Fan Curve Recommendations: A Unique feature of PC Metrics is the use of AI to optimize fan curves. I integrated Anthropic's Claude (a large language model) via an API key I had to analyze the telemetry data and suggest better cooling strategies. In the project's code (`claude_ai.py`), I created a class `ClaudeAIOptimizer` that can take historical sensor data and produce recommendations. The idea is to feed the model summary statistics (max, avg temps, etc.) **and have it generate an "optimal fan profile" preferring thermals, acoustics or balancing both.**

Claude can analyze the time-series for patterns (detect that GPU hits 85°C frequently when gaming) and output a more nuanced fan curve or cooling strategy (perhaps recommending additional cooling hardware or different fan control parameters). The long-term vision is an AI-driven feedback loop: the system would periodically run analysis (maybe daily or when enough data is collected) and then suggest to the user or change by itself the fan curves according to the user's preferences.

This would offload the tedious task of fan tuning to an AI that learns from data. I believe that using an LLM for this is an interesting application of AI in the IoT space. Treating the AI as an expert that can transform telemetry data into actual advice.



Temperature Analysis:
CPU: Avg 53.6°C, Max 74.0°C
GPU: Avg 52.7°C, Max 69.2°C
Gaming sessions detected: undefined

Recommended Fan Curves (best_temps):

CPU Fan: 800 → 1400 → 2200 → 3600 → 4500 RPM

GPU Fan: 0 → 1200 → 2400 → 3900 → 4200 RPM

Case Fan: 600 → 1000 → 1600 → 2400 → 2800 RPM

```
CPU averaging 53.6°C - current cooling adequate
Gaming detected 74.0% of time - optimized for gaming workloads
GPU reaching 69.2°C max - cooling performance good
```

Results

Real-Time Cloud Dashboard: The Grafana dashboard successfully displays live data from the simulator. As I run the simulator on my PC, I can watch the Grafana panels update every second with new points. The **temperature graphs clearly reflect the gaming cycles** : for instance, during a simulated intense gaming period, CPU temperature climbed from an idle ~42°C to about 78°C over the course of a minute, then leveled off. Additionally, the GPU temperature rose into the 70s°C. When the gaming session ended the temperatures gradually fell back to baseline. I also see the **fan RPM responses**: the CPU fan, which idles ~1200 RPM, ramped up to ~4500 RPM at peak CPU temp, and reduced its speed as the temp changed.

Data Storage and Querying: In Amazon Timestream, I verified that data points were being recorded. Running a query via AWS Console `SELECT MAX(cpu_temp), MAX(gpu_temp) FROM "PCMetricsDB"."Telemetry"  AND time BETWEEN ago(1h) AND now()` successfully returned the expected values. Over a few hours of runtime, lots of records were accumulated in Timestream. I also checked the S3 bucket : it contained the raw JSON files from the IoT rule. The files were appropriately timestamped. This means if needed, I have a full backup of all telemetry data outside the time-series database, **fulfilling the data retention goal.**

Alerts and Notifications: The alert mechanism was demonstrated by intentionally pushing the system into an alarm state. In one test, I set the simulator's CPU max temperature slightly higher and allowed it to run hot for a prolonged period. The CloudWatch alarm for CPU > 75°C (with a 1 minute evaluation period) went into **ALARM state**, and moments later I received an **SNS email alert** in my inbox stating the alarm name and that it had been triggered. This proved the pipeline is functional : from IoT data to CloudWatch metric to SNS notification. Additionally, on the Grafana dashboard, the top of the CPU temperature panel showed a red indicator and the text "alerting", which is Grafana's way of highlighting that an alert condition is active. In a real deployment, the combination of visual alerts and SNS (email/SMS) alerts means the user would be quickly informed of any overheating.

AI Recommendations Output: With the simulator data collected, I triggered the AI analysis function to see what it would recommend. The (simulated) Claude output was along these lines: *"CPU averaging 66.3°C (max 79.5°C) : current cooling adequate, but nearing limit under load. GPU reaching 82.1°C max with* ***enhanced cooling suggested*** *(fan curve could be more aggressive). Gaming was detected 35% of the time : optimize for gaming workloads."* It then provided a suggested fan RPM curve for each component favoring extra cooling for the GPU. In the future I would expect even more insightful commentary (For example: identifying that perhaps the case fan is the bottleneck or recommending additional case fans if temps remain high).

In summary, the system met the core requirements: I can **monitor my PC's vitals in real-time from anywhere via the cloud**, I get **alerts via email** if something goes wrong, and I have a pathway to use **AI for optimizing performance**. All of this was achieved using IoT and multiple AWS cloud technologies.

Challenges & Solutions

CloudWatch Alarm Tuning: Getting the alarms to not be too sensitive (or too insensitive) required some tweaking. If the evaluation period was too short, brief temperature spikes would trigger alerts too frequently. If too long, it might delay important warnings. I settled on a moderate evaluation period (e.g., 2 minutes with 1-minute datapoints) for temperature alarms. Also, because the simulator can produce quick swings, I chose to use the **max** statistic on the metric within the period to ensure any spike is caught.

IoT Core Certificate Connection Issues: Getting the simulator to connect to AWS IoT Core over MQTT (TLS) proved tricky at first. Despite following the steps, I saw some connection failures. The logs and error messages were not very specific, but I suspected a certificate or policy misconfiguration.Eventually, I discovered two issues: first I had not attached the IoT policy to the Thing's certificate, and second I was using the wrong Certificate file. After fixing these, the connection succeeded. The code's debug messages (which I built in) guided me here – I had included a check to print suggestions like "Check your certificate file paths and IoT endpoint" if connection fails.The takeaway is that IoT Core requires precise setup: all must align. Once they did, I saw the "SUCCESS: Successfully connected to AWS IoT Core!" log message, and data started flowing.

Security and Permissions: As with any IoT/cloud project, security was a concern. I encountered IAM permission issues when Grafana was trying to read Timestream. The Managed Grafana service needs an IAM role with policies for Timestream access. I had to adjust the trust policy and permissions to allow Grafana to assume the role. Similarly, the IoT rule that writes to Timestream needed an IAM role with `timestream:WriteRecords` permission. Setting these up properly was a bit of a learning curve.

In the end, the security architecture includes TLS encryption for data in transit (MQTT TLS 1.2), fine-grained IAM roles for IoT and Timestream, and restricted SNS topics. These measures align with best practices (as highlighted in the project presentation). For instance, my IoT policy only allows publish/subscribe on the `gamingPC/telemetry` topic and nothing else.

In overcoming these challenges, I gained experience in **debugging IoT systems** and learned to use various tools. By the time of completion, the system was stable and secure, and I had a deeper understanding of the interplay between IoT devices and cloud services.

Future Work

While the PC Metrics project as implemented provides a strong foundation, there are several areas of expansion and improvement I have identified:

Additional Sensors and Components: Currently, the simulator monitors CPU, GPU, SSD, and motherboard temperatures, and corresponding fan speeds. In the future, its possible to incorporate **more sensors** such as RAM temperatures and PSU (power supply) temperature or fan speed. The code is designed to be extensible. The IoT data schema and cloud setup would accommodate new fields with minimal changes.

User Portal and Multi-User Support: A future improvement is to build a **web portal or mobile app** where users can register their gaming PC (each gets a device ID and certs), and then view their own dashboards online. They could compare their PC metrics with the community averages : for instance, see if their CPU runs hotter than other similar systems. This could foster a community-driven approach to identifying best cooling setups.Mobile push notifications via SNS could also be integrated for alert delivery.

Deeper Machine Learning Modeling: The current AI approach offloads the intelligence to a pre-trained LLM. Another route is to develop specialized **ML models** on the collected data. For example, a regression model could predict the optimal fan speed percentage needed given a certain temperature and workload level, effectively learning the ideal fan curve from data. Or a classification model could predict "overheating risk" based on current trends (e.g., if temps are rising quickly, predict that it will hit critical level in X minutes, and proactively increase fan speeds or send alerts). With sufficient data (especially from multiple PCs), training a model using AWS SageMaker could be an exciting extension. One could also apply anomaly detection on the time-series data: if a fan is failing (RPM doesn't increase as expected with temperature), the system could flag that as an anomaly. Timestream integrates with AWS IoT Analytics and QuickSight, which could be used for deeper analysis too, but a custom ML model might be more powerful for prediction tasks.

Edge Execution of AI Recommendations: Currently the AI recommendations are suggestions for the user. In the future, I envision closing the loop by allowing the system to **automatically adjust fan settings**. Many motherboards support controlling fan speeds via software (through APIs or utilities). If PC Metrics could interface with those (perhaps via a local agent or script using something like OpenRGB or motherboard SDKs), it could apply the Claude/ML-recommended fan curve in real-time. This would effectively create a self-optimizing PC cooling system.

Integration with Existing Platforms: Many gamers use tools like MSI Afterburner or Corsair iCUE for monitoring and control. A future improvement could be to integrate PC Metrics with such tools. For instance, PC Metrics could export data or alerts to those platforms, or ingest data from them.

In summary, there are many avenues to extend PC Metrics. Long-term, the integration of more advanced AI and automated control stands to make this not just a monitoring tool, but an **autonomous cooling assistant** for PCs.

Conclusion

Working on PC Metrics has been a fun journey into the intersection of Internet of Things, cloud computing, and artificial intelligence. The first-person experience of watching my simulated PC temps on a Grafana cloud dashboard in real time : and getting an email when the “virtual CPU” overheated truly demonstrated the power of IoT for remote monitoring and management.

Key takeaways:

End-to-End IoT Solution Development: I learned how to build an IoT solution from scratch, starting at the edge device through to cloud ingestion (AWS IoT Core), storage (Timestream, S3), processing (CloudWatch alarms), and presentation (Grafana dashboards). Each layer has its role, and ensuring they work together (with correct data formats, permissions, and connectivity) is crucial.

Importance of Real-Time Data and Alerts: Seeing how quickly an alert can be raised and delivered emphasizes the value of real-time monitoring. If this were a real gaming PC, such a system could potentially prevent hardware damage by catching overheating early.

Cloud Services Integration: Using managed services greatly accelerated development. AWS IoT Core simplified secure device connectivity, Timestream eliminated the need to manage a database server, Managed Grafana provided a professional-grade visualization with minimal setup. The project reinforced that leveraging cloud services allows an individual to implement complex systems that once might have taken teams to build relatively quickly. It also taught me about the considerations of using these services (Cost, region, service limits).

Security & Scalability: I gained practical experience in securing IoT communications (using certs and AWS IoT policies) and understanding how the system could scale. The architecture is serverless and can scale to many devices/publishers :

Role of AI in IoT: Incorporating the AI element showed me a glimpse of how IoT data can be turned into sort of actionable intelligence that can make the user’s life easy with expert advices and automatic control over tedious tasks.

Personal Learning: Writing this report allowed me to reflect on why I made certain decisions and how I solved problems. It feels like documenting not just what I built, but what I learned. Explaining it as “I did X, then Y, and solved it by Z” solidified my understanding and I hope also makes it clear how much was gained from this project.

In conclusion, PC Metrics achieved its aim as a proof-of-concept for smart PC monitoring using IoT. As it stands, the project demonstrates a working model of real-time, cloud-enabled PC telemetry with intelligent oversight. There is also potential for further development in both quality and scale. It has been rewarding, and it opened the door for me to continue learn about IoT and Cloud technologies **(Planning on getting an AWS Architect certificate)**

Appendices

Whole project is on GitHub at: github.com/eladlavi55/PC-Metrics

Appendix A: Demo Video: youtu.be/t5dDW7nQ_fE

Appendix B: Sample Code Snippet (Simulation Loop)

```
# Main loop excerpt from simulate_pc_metrics.py
while True:
   pc_simulator.update_gaming_session()       # Update gaming_session and intensity
   pc_simulator.simulate_temperature_changes() # Calculate new temperatures based on intensity
   pc_simulator.calculate_fan_speeds()        # Adjust fan RPMs according to temps

   sensor_data = pc_simulator.get_sensor_data()  # Collect readings into dict

   if aws_connected:
      aws_publisher.publish_data(sensor_data)   # Send data to AWS IoT Core

   # (Optional: emit to local dashboard via socketio, log to file)
   time.sleep(config.SENSOR_UPDATE_INTERVAL)  # Wait 1 second before next reading
```

Explanation: This loop runs in the background of the simulation. It continuously updates the state of the PC components and publishes the sensor readings to AWS if connected. The `update_gaming_session()` manages the transition from idle to gaming and back, `simulate_temperature_changes()` applies a model to increase or decrease temperatures towards a target (with some randomness), and `calculate_fan_speeds()` sets fan speeds based on the new temperatures (following a predefined fan curve logic). The data is then gathered and published over MQTT. The loop sleeps for the configured interval (1 second) to simulate real-time sensor polling.

Appendix C: AWS IoT Policy Example

Below is an example AWS IoT Core policy used for the device certificate in this project. It gives the necessary permissions for the device to connect, publish telemetry to the topic, and subscribe if needed:

```
{
 "Version": "2012-10-17",
 "Statement": [
  {
   "Effect": "Allow",
   "Action": "iot:Connect",
   "Resource": "arn:aws:iot:us-east-1:<AWS_ACCOUNT_ID>:client/${iot:ClientId}"
  },
  {
   "Effect": "Allow",
   "Action": [
    "iot:Publish",
    "iot:Subscribe",
    "iot:Receive"
   ],
    "Resource": "arn:aws:iot:us-east-1:<AWS_ACCOUNT_ID>:topic/gamingPC/telemetry*"
  }
 ]
}
```

Notes: `${iot:ClientId}` is a variable referring to the Thing's client ID (in our case "GamingPC4"). This policy allows my device to connect to AWS IoT and to publish/subscribe to any topic under `gamingPC/telemetry`. This policy was attached to the device's X.509 certificate in AWS IoT Core so that the simulator could authenticate and communicate. Security is kept tight by not allowing other actions or broader resources.