Project Modules Implementation

Infrastructure as Code (IaC) Implementation

What this module will do:

- Load Terraform files (.tf) before they are deployed.
- Detect simple misconfigurations like:

Action = "*" or **Resource** = "*" in IAM policy blocks.

Public S3 buckets (acl = "public-read" or public = true).

Security group with **wide-open ingress** (0.0.0.0/0).

- Save findings to a CSV/print them on screen.

Steps to reproduce implementation:

Step 1: Install a parser library for HCL (Terraform Language). HCL is a language used to describe infrastructure resources in machine friendly and human-readable format.

Terraform is an open-source IaC tool that uses HCL to define, provision and manage infrastructure resources such as cloud infrastructure in AWS using configuration files. (In short, a CLI for defining resources in

cloud instead of clicking buttons).

```
PS C:\Users\harsh> pip install python-hcl2
Collecting python-hcl2
  Downloading python_hcl2-7.3.1-py3-none-any.whl.metadata (5.2 kB)
Collecting lark<2.0,>=1.1.5 (from python-hcl2)
  Downloading lark-1.2.2-py3-none-any.whl.metadata (1.8 kB)
Requirement already satisfied: regex>=2024.4.16 in d:\anaconda\lib\site-packages (from python-hcl2) (2024.9.11)
Downloading python_hcl2-7.3.1-py3-none-any.whl (22 kB)
Downloading lark-1.2.2-py3-none-any.whl (111 kB)
Installing collected packages: lark, python-hcl2
Successfully installed lark-1.2.2 python-hcl2-7.3.1
```

Step 2: Write the python code for the scanner which detects and prevents insecure policies **before** deployment.

Lets name the file iac_monitor.py:

```python
import hcl2
import os
import csv
import sys

def (variable) findings: list
    findings = []
    with open(filepath, 'r') as f:
        try:
            data = hcl2.load(f)
        except Exception as e:
            return [{"file": filepath, "resource": "N/A", "finding": f"Parse error: {e}"}]

    if "resource" in data:
        resources = data["resource"]

        # Case 1: dict
        if isinstance(resources, dict):
            for rtype, blocks in resources.items():
                for name, block in blocks.items():
                    findings.extend(check_resource(filepath, rtype, name, block))

        # Case 2: list
        elif isinstance(resources, list):
            for res in resources:
                for rtype, blocks in res.items():
                    for name, block in blocks.items():
                        findings.extend(check_resource(filepath, rtype, name, block))

    return findings


def check_resource(filepath, rtype, name, block):
    """Check a single Terraform resource for misconfigurations."""
    findings = []

```

```
    # Normalize block into list of dicts
    if isinstance(block, dict):
        block = [block]

    # IAM Policy
    if rtype == "aws_iam_policy":
        policy_doc = ""

    try:
        raw_policy = block[0].get("policy", "")
        if isinstance(raw_policy, list) and raw_policy:
            policy_doc = raw_policy[0]
        else:
            policy_doc = raw_policy
    except Exception:
        policy_doc = ""

    text = str(policy_doc)

    action_wild = '"Action": "*"' in text
    resource_wild = '"Resource": "*"' in text

    if action_wild and resource_wild:
        findings.append({
            "file": filepath,
            "resource": name,
            "finding": "IAM policy allows full admin (*:* on all resources)"
        })
    elif action_wild:
        findings.append({
            "file": filepath,
            "resource": name,
            "finding": "IAM policy allows all actions (*)"
        })
```

```
        findings.append({
            "file": filepath,
            "resource": name,
            "finding": "IAM policy allows all actions (*)"
        })
    elif '"Resource": "*"' in str(policy_doc) and '"Action": "*"' in str(policy_doc):
        findings.append({
            "file": filepath,
            "resource": name,
            "finding": "IAM policy allows all resources (*)"
        })

    # S3 Bucket
    if rtype == "aws_s3_bucket":
        acl = block[0].get("acl", "")
        if isinstance(acl, list) and acl:
            acl = acl[0]
        if "public" in str(acl).lower():
            findings.append({
                "file": filepath,
                "resource": name,
                "finding": f"S3 bucket with public ACL ({acl})"
            })

    # Security Group
    if rtype == "aws_security_group":
        for ingress in block[0].get("ingress", []):
            cidrs = ingress.get("cidr_blocks", [])
            if "0.0.0.0/0" in cidrs:
                findings.append({
                    "file": filepath,
                    "resource": name,
                    "finding": "Security group allows 0.0.0.0/0 (open to world)"
                })
```

```
    return findings

def scan_directory(path=".", output_csv="iac_findings.csv"):
    all_findings = []
    for root, dirs, files in os.walk(path):
        for f in files:
            if f.endswith(".tf"):
                filepath = os.path.join(root, f)
                all_findings.extend(scan_tf_file(filepath))

    # Write CSV
    keys = ["file", "resource", "finding"]
    with open(output_csv, "w", newline="") as f:
        writer = csv.DictWriter(f, fieldnames=keys)
        writer.writeheader()
        for row in all_findings:
            writer.writerow(row)

    print(f"[*] Scan complete. {len(all_findings)} findings written to {output_csv}")
    for r in all_findings:
        print(f"{r['file']} - {r['resource']} -> {r['finding']}")



if __name__ == "__main__":
    if len(sys.argv) > 1:
        filepath = sys.argv[1]
        findings = scan_tf_file(filepath)

        if findings:
            for r in findings:
                print(f"{r['file']} - {r['resource']} -> {r['finding']}")
            print(f"[*] Scan complete. {len(findings)} findings found in {filepath}")
        else:
            print(f"[*] Scan complete. No findings in {filepath}")
            print(f"[*] Scan complete. No findings in {filepath}")
    else:
        scan_directory(".", output_csv="iac_findings.csv")

```

Step 3: We will create three Terraform files to check the scanner's efficiency and ability to identify misconfigurations.

bad.tf:

```
resource "aws_s3_bucket" "bad_bucket" {
  bucket = "test-bad-bucket"
  acl    = "public-read"
}

resource "aws_security_group" "bad_sg" {
  name = "bad-sg"
  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}

resource "aws_iam_policy" "bad_policy" {
  name   = "badPolicy"
  policy = <<EOF
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "*",
      "Resource": "*"
    }
  ]
}
EOF
}
```

good.tf

```
resource "aws_iam_policy" "good_policy" {
  name   = "goodPolicy"
  policy = <<EOF
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": ["s3:GetObject"],
      "Resource": "arn:aws:s3:::test-good-bucket/*"
    }
  ]
}
EOF
}
```

mix.tf

```
# BAD: Security group open to the world (port 80)
resource "aws_security_group" "bad_web_sg" {
  name = "bad-web-sg"

  ingress {
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}

# GOOD: Security group restricted to internal subnet
resource "aws_security_group" "good_internal_sg" {
  name = "good-internal-sg"

  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["10.0.0.0/24"]
  }
}

# BAD: S3 bucket public-read-write
resource "aws_s3_bucket" "bad_public_bucket" {
  bucket = "bad-public-bucket"
  acl    = "public-read-write"
}

# GOOD: Private S3 bucket
resource "aws_s3_bucket" "good_private_bucket" {
  bucket = "good-private-bucket"
  acl    = "private"
}

# BAD: IAM policy with wildcard actions
resource "aws_iam_policy" "bad_admin_policy" {
  name   = "badAdminPolicy"
  policy = <<EOF
```

```
  policy = <<EOF
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "*",
      "Resource": "*"
    }
  ]
}
EOF
}

# GOOD: IAM policy with least privilege
resource "aws_iam_policy" "good_readonly_policy" {
  name   = "goodReadOnlyPolicy"
  policy = <<EOF
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": ["ec2:DescribeInstances"],
      "Resource": "*"
    }
  ]
}
EOF
}
```

Step 4: Run the scanner against these .tf files and we can see the findings and vulnerabilities detected in the infrastructure of the resource.

```
PS C:\Users\harsh> python iac_monitor.py mix.tf
mix.tf - bad_web_sg -> Security group allows 0.0.0.0/0 (open to world)
mix.tf - bad_public_bucket -> S3 bucket with public ACL (public-read-write)
mix.tf - bad_admin_policy -> IAM policy allows full admin (*:* on all resources)
mix.tf - bad_admin_policy -> IAM policy allows all actions (*)
[*] Scan complete. 4 findings found in mix.tf
PS C:\Users\harsh> python iac_monitor.py good.tf
[*] Scan complete. No findings in good.tf
PS C:\Users\harsh> python iac_monitor.py bad.tf
bad.tf - bad_bucket -> S3 bucket with public ACL (public-read)
bad.tf - bad_sg -> Security group allows 0.0.0.0/0 (open to world)
bad.tf - bad_policy -> IAM policy allows full admin (*:* on all resources)
bad.tf - bad_policy -> IAM policy allows all actions (*)
[*] Scan complete. 4 findings found in bad.tf
```

Step 5: The findings are documented in the iac_findings.csv file which can be used to analyze and visualize (using Grafana).

file	resource	finding
.\good.tf	good_poli	IAM policy contains wildcard *
.\main.tf	bad_bucke	S3 bucket with public ACL (public-read)
.\main.tf	bad_sg	Security group allows 0.0.0.0/0 (open to world)
.\main.tf	bad_policy	IAM policy contains wildcard *

Future Scope:

1. **Multi-Cloud Expansion**: (Azure, GCP etc).

2. **Severity-Based Risk Scoring**: High, Medium, Low.

3. **Continuous Monitoring**: Deploy as a Lambda function to automatically run on a schedule.

4. **Visualization & Reporting**: Dashboards using AWS QuickSight, Grafana.

Policy Decision Point (PDP)

The **Policy Decision Point (PDP)**:

- Receives access or configuration evaluation requests.
- Applies defined **Zero Trust policies** (least privilege, deny by default).
- Returns a decision: **ALLOW** / **DENY** / **REVIEW**.

Implementation Steps:

1. Reads input context (user, action, resource, risk level).

2. Applies a set of policies defined in a JSON file.

3. Decides whether to allow or deny access.

4. Logs the decision to a .csv file for review.

Step 1:

Define the policies.json which will simulate the user resource request.

```json
{
  "default_action": "deny",
  "policies": [
    {
      "id": "policy2",
      "description": "Deny iam:PassRole",
      "conditions": {
        "action": ["iam:passrole"]
      },
      "decision": "deny"
    },
    {
      "id": "policy3",
      "description": "Allow read-only actions",
      "conditions": {
        "action": ["s3:getobject", "ec2:describeinstances"]
      },
      "decision": "allow"
    },
    {
      "id": "policy1",
      "description": "Deny wildcard actions",
      "conditions": {
        "action": ["*"]
      },
      "decision": "deny"
    }
  ]
}
```

Step 2:

Implement the Policy Decision Point which will read the policies.json file based on actions and also use context awareness to decide whether the request should be accepted or denied.

```python
#!/usr/bin/env python3
"""
Hybrid Context + Policy-Based PDP
Evaluates:
 - Action policies (from policies.json)
 - Context policies (IP, device, time)
Combines both to enforce Zero Trust access decisions
"""

import json
import csv
import datetime as dt
import sys

POLICY_FILE = "policies.json"
OUTPUT_LOG = "pdp_decisions.csv"

# Context configuration
TRUSTED_IP_RANGES = ["192.168.", "10.0."]
TRUSTED_DEVICES = ["device-laptop-001", "device-admin-001"]
BUSINESS_HOURS = (8, 20)  # 8 AM - 8 PM

def load_policies():
    with open(POLICY_FILE, "r") as f:
        return json.load(f)

def in_trusted_network(ip):
    return any(ip.startswith(prefix) for prefix in TRUSTED_IP_RANGES)

def within_business_hours():
    now = dt.datetime.now().hour
    return BUSINESS_HOURS[0] <= now < BUSINESS_HOURS[1]

def is_trusted_device(device_id):
    return device_id in TRUSTED_DEVICES

def evaluate_context(request):
```

```python
    ip = request.get("ip", "unknown")
    device = request.get("device", "unknown")

    # Contextual checks
    if not in_trusted_network(ip):
        return "deny", f"Untrusted network source ({ip})"
    if not within_business_hours():
        return "deny", "Access attempted outside business hours"
    if not is_trusted_device(device):
        return "review", f"Unrecognized device ({device})"
    return "allow", "Context validated"

def evaluate_action(request, policy_data):
    """Evaluates static action-based policies"""
    action = request.get("action", "").lower()
    for p in policy_data["policies"]:
        actions = [a.lower() for a in p["conditions"].get("action", [])]
        if action in actions or "*" in actions:
            return p["decision"], p["description"]
    return policy_data.get("default_action", "deny"), "No matching policy (default deny)"

def combine_decisions(context_decision, action_decision):
    """Combines contextual and policy-based outcomes"""
    # Deny overrides everything (Zero Trust principle)
    if "deny" in (context_decision, action_decision):
        return "deny"
    # Review if context is uncertain but action is allowed
    if context_decision == "review" and action_decision == "allow":
        return "review"
    # Allow only if both are clean
    if context_decision == "allow" and action_decision == "allow":
        return "allow"
    # Default fallback
    return "deny"

def log_decision(result):
```

```python
    with open(OUTPUT_LOG, "a", newline="") as f:
        writer = csv.DictWriter(f, fieldnames=keys)
        if f.tell() == 0:
            writer.writeheader()
        writer.writerow(result)

def main():
    if len(sys.argv) != 6:
        print("Usage: python pdp.py <user> <action> <resource> <ip> <device>")
        sys.exit(1)

    user, action, resource, ip, device = sys.argv[1:]
    request = {"user": user, "action": action, "resource": resource, "ip": ip, "device": device}

    # Load and evaluate
    policy_data = load_policies()
    context_decision, context_reason = evaluate_context(request)
    action_decision, action_reason = evaluate_action(request, policy_data)
    final_decision = combine_decisions(context_decision, action_decision)

    # Pick most relevant reason
    reason = (
        context_reason if final_decision == context_decision else action_reason
    )

    result = {
        "timestamp": dt.datetime.now(dt.timezone.utc).isoformat(),
        "user": user,
        "action": action,
        "resource": resource,
        "ip": ip,
        "device": device,
        "decision": final_decision,
        "reason": reason,
    }
```

Output:

```
PS C:\Users\harsh\OneDrive\Desktop> python pdp.py alice s3:GetObject arn:aws:s3:::secure-bucket 192.168.1.12 device-laptop-001
[PDP] Decision for alice → DENY (Reason: Access attempted outside business hours)
PS C:\Users\harsh\OneDrive\Desktop> python pdp.py alice s3:GetObject arn:aws:s3:::secure-bucket 8.8.8.8 device-laptop-001
[PDP] Decision for alice → DENY (Reason: Untrusted network source (8.8.8.8))
PS C:\Users\harsh\OneDrive\Desktop> python pdp.py bob s3:GetObject arn:aws:s3:::secure-bucket 192.168.1.12 device-unknown
[PDP] Decision for bob → DENY (Reason: Access attempted outside business hours)
PS C:\Users\harsh\OneDrive\Desktop> python pdp.py bob iam:PassRole arn:aws:iam::123456789:role/Admin 192.168.1.12 device-laptop-001
[PDP] Decision for bob → DENY (Reason: Access attempted outside business hours)
PS C:\Users\harsh\OneDrive\Desktop>
```

Command 1: Allowed Context + Allowed Policy

Command 2: Untrusted IP, even if Action is Safe

Command 3: Unknown Device, Safe Action

Command 4: Denied Action (iam:PassRole)

Logs:

timestamp	user	action	resource	decision	matched_policy	reason	
2025-10-24T15:08:58.028887	alice	s3:getobject	arn:aws:s3:::secure-bucket	deny	policy1	Deny wildcard actions	
2025-10-24T15:09:17.810669	bob	iam:passrole	arn:aws:iam::123456789:role/Admin	deny	policy1	Deny wildcard actions	
2025-10-24T15:10:46.693979	alice	s3:getobject	arn:aws:s3:::secure-bucket	allow	policy3	Allow read-only actions	
2025-10-24T15:10:54.386624	bob	iam:passrole	arn:aws:iam::123456789:role/Admin	deny	policy2	Deny iam:PassRole	
2025-10-24T15:13:32.836592+00:00	alice	s3:GetObject	arn:aws:s3:::secure-bucket	192.168.1.	device-laptop-001	deny	Access attempted outside business hours
2025-10-24T15:13:42.731833+00:00	alice	s3:GetObject	arn:aws:s3:::secure-bucket	8.8.8.8	device-laptop-001	deny	Untrusted network source (8.8.8.8)
2025-10-24T15:13:49.852275+00:00	bob	s3:GetObject	arn:aws:s3:::secure-bucket	192.168.1.	device-unknown	deny	Access attempted outside business hours
2025-10-24T15:16:10.828954+00:00	alice	s3:GetObject	arn:aws:s3:::secure-bucket	192.168.1.	device-laptop-001	deny	Access attempted outside business hours
2025-10-24T15:16:17.574313+00:00	alice	s3:GetObject	arn:aws:s3:::secure-bucket	8.8.8.8	device-laptop-001	deny	Untrusted network source (8.8.8.8)
2025-10-24T15:16:23.706256+00:00	bob	s3:GetObject	arn:aws:s3:::secure-bucket	192.168.1.	device-unknown	deny	Access attempted outside business hours
2025-10-24T15:16:38.847227+00:00	bob	iam:PassRole	arn:aws:iam::123456789:role/Admin	192.168.1.	device-laptop-001	deny	Access attempted outside business hours
2025-10-24T15:18:09.581952+00:00	alice	s3:GetObject	arn:aws:s3:::secure-bucket	192.168.1.	device-laptop-001	deny	Access attempted outside business hours
2025-10-24T15:19:40.434471+00:00	alice	s3:GetObject	arn:aws:s3:::secure-bucket	192.168.1.	device-laptop-001	deny	Access attempted outside business hours
2025-10-24T15:35:05.570089+00:00	alice	s3:GetObject	arn:aws:s3:::secure-bucket	192.168.1.	device-laptop-001	deny	Access attempted outside business hours
2025-10-24T15:35:13.022379+00:00	alice	s3:GetObject	arn:aws:s3:::secure-bucket	8.8.8.8	device-laptop-001	deny	Untrusted network source (8.8.8.8)
2025-10-24T15:35:21.948335+00:00	bob	s3:GetObject	arn:aws:s3:::secure-bucket	192.168.1.	device-unknown	deny	Access attempted outside business hours
2025-10-24T15:35:30.290047+00:00	bob	iam:PassRole	arn:aws:iam::123456789:role/Admin	192.168.1.	device-laptop-001	deny	Access attempted outside business hours

Policy Enforcement Point (PEP) Implementation

- The PEP (Policy Enforcement Point) is the gatekeeper — every request to access a resource passes through it.
- It forwards the access request (user, action, resource, IP, device, etc.) to the PDP (Policy Decision Point).
- The PDP evaluates the request based on policies, context, and risk, and returns either ALLOW or DENY.
- The PEP enforces that decision — allowing or blocking the request.

Code:

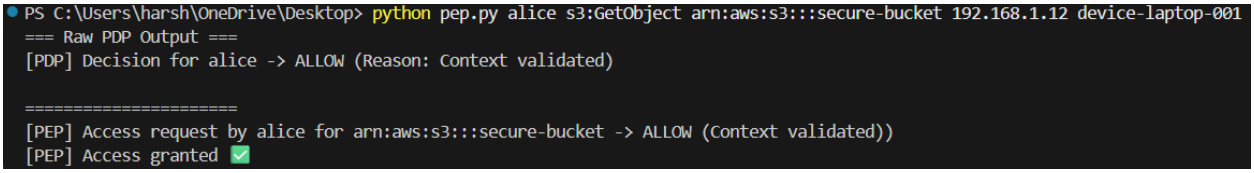
```
Final Robust Policy Enforcement Point (PEP)
Compatible with hybrid PDP and PowerShell output encoding.
"""

import subprocess
import sys
import json
import datetime as dt
import re

def log_event(entry):
    with open("pep_log.json", "a") as f:
        f.write(json.dumps(entry) + "\n")

def extract_decision_and_reason(output):
    """Extracts decision (ALLOW/DENY/REVIEW) and reason from PDP output."""
    # Normalize encoding and remove fancy characters
    clean_output = output.encode("ascii", "ignore").decode("ascii")
    decision_match = re.search(r"\b(ALLOW|DENY|REVIEW)\b", clean_output, re.IGNORECASE)
    reason_match = re.search(r"Reason:\s*(.*)", clean_output, re.IGNORECASE | re.DOTALL)


    decision = decision_match.group(1).upper() if decision_match else "DENY"
    reason = reason_match.group(1).strip() if reason_match else "Unknown"
    return decision, reason

def enforce_access(user, action, resource, ip, device):

    result = subprocess.run(
        [sys.executable, "-u", "pdp.py", user, action, resource, ip, device],
        stdout=subprocess.PIPE,
        stderr=subprocess.STDOUT,
        text=True
    )
```

```
    print("=== Raw PDP Output ===")
    print(result.stdout)
    print("======================")


    output = result.stdout.strip()
    decision, reason = extract_decision_and_reason(output)

    entry = {
        "timestamp": dt.datetime.now(dt.timezone.utc).isoformat(),
        "user": user,
        "action": action,
        "resource": resource,
        "ip": ip,
        "device": device,
        "decision": decision,
        "reason": reason
    }
    log_event(entry)

    print(f"[PEP] Access request by {user} for {resource} -> {decision} ({reason})")

    if decision == "DENY":
        print("[PEP] Request blocked ❌")
    elif decision == "REVIEW":
        print("[PEP] Access under manual review ⚠️")
    else:
        print("[PEP] Access granted ✅")

if __name__ == "__main__":
    if len(sys.argv) != 6:
        print("Usage: python pep.py <user> <action> <resource> <ip> <device>")
        sys.exit(1)

    _, user, action, resource, ip, device = sys.argv
```

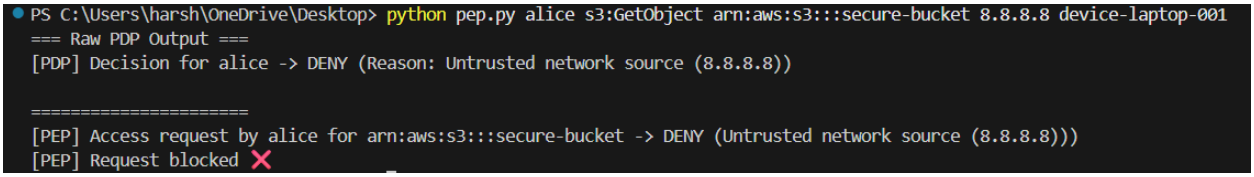
```
    _, user, action, resource, ip, device = sys.argv
    enforce_access(user, action, resource, ip, device)
```

Output:

Allow Example:



Deny Example:



Auto Remediation Module

Feature	Description	Benefit
Multi-cloud support	AWS, Azure, GCP	Cross-cloud remediation
Policy-driven automation	Triggered by PEP/PDP decisions	Reduces manual effort
Action logging	Stores timestamped logs in JSON	Auditability & traceability
Flexible actions	Auto-remediation + review notifications	Combines automation & oversight
Lightweight & extensible	Modular Python code	Easy to scale and maintain
Context-aware	Includes user/resource/cloud/reason	Better incident analysis
Real-time enforcement	Triggered immediately after PEP	Mitigates risks quickly
Audit-friendly output	JSON logs	Integrates with SOC/SIEM

Code:

Arm.py

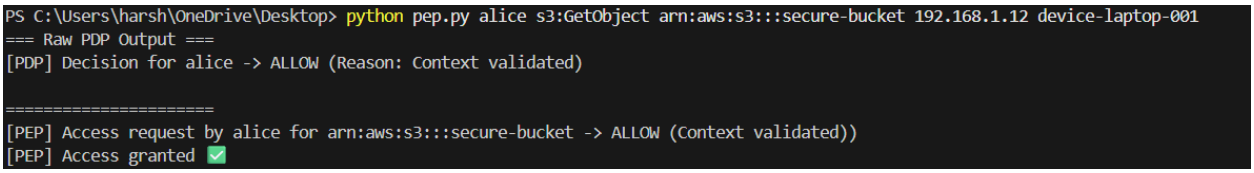
```python
# arm.py
import json
from datetime import datetime as dt

# Mock multi-cloud adapters
def aws_revoke_access(user):
    # Mock AWS remediation
    # Replace this with actual boto3 code if boto3 is installed
    try:
        # Simulate removing user from risky group
        return f"Removed {user} from SensitiveAccess group in AWS (mock)"
    except Exception as e:
        return f"AWS Remediation failed: {e}"

def azure_revoke_access(user):
    # Placeholder: call Azure SDK to remove user from risky role
    return f"Azure remediation triggered for {user}"

def gcp_revoke_access(user):
    # Placeholder: call GCP IAM API to revoke roles
    return f"GCP remediation triggered for {user}"

def log_remediation(entry):
    with open("arm_log.json", "a") as f:
        f.write(json.dumps(entry) + "\n")

def auto_remediate(user, resource, decision, reason, cloud):
    timestamp = dt.utcnow().isoformat()
    actions = []

    if decision == "DENY":
        if "aws" in cloud.lower():
            actions.append(aws_revoke_access(user))
        elif "azure" in cloud.lower():
            actions.append(azure_revoke_access(user))
        elif "gcp" in cloud.lower():
            actions.append(gcp_revoke_access(user))
    elif decision == "REVIEW":
        # For review, just notify/admin log
        actions.append(f"Admin review needed for {user} on {resource}: {reason}")

    log_entry = {
        "timestamp": timestamp,
        "user": user,
        "resource": resource,
        "decision": decision,
        "reason": reason,
        "cloud": cloud,
        "actions_taken": actions
    }

    log_remediation(log_entry)
    print(f"[ARM] Auto-Remediation Log: {log_entry}")

```

Modification in pep.py to integrate auto remediation when PEP is run.

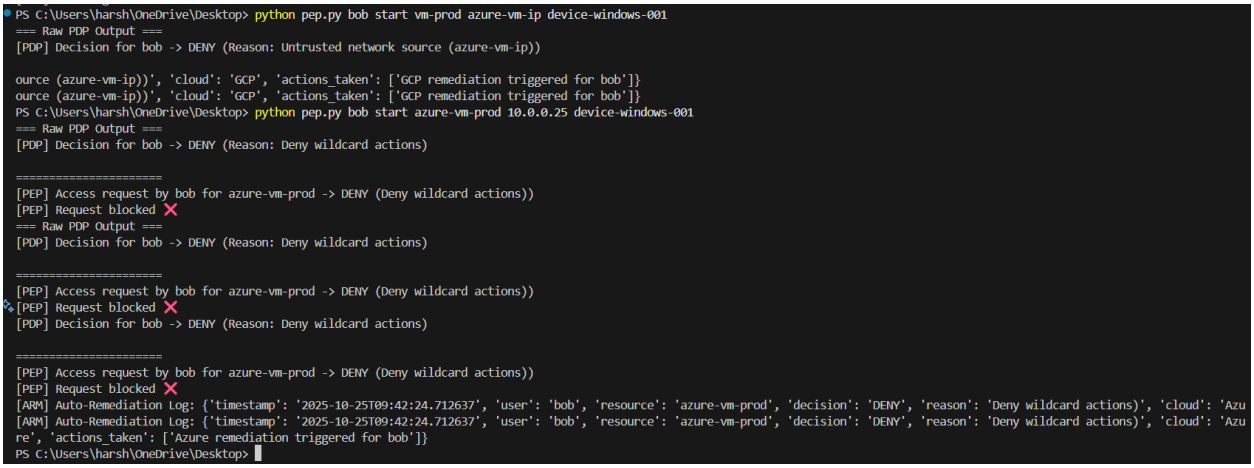
```python
# ------------------------------
# Trigger Auto Remediation
# ------------------------------
if decision in ["DENY", "REVIEW"]:
    cloud_target = (
        "AWS" if "aws" in resource.lower()
        else "Azure" if "azure" in resource.lower()
        else "GCP"
    )
    auto_remediate(user, resource, decision, reason, cloud_target)
```

Result:

Allow (No remediation required)



Deny (With remediation)



Monitoring and Logging Module

Centralized Logging

- Every event in the framework (PEP decisions, ARM actions, etc.) is **logged as a structured JSON entry**.
- Example fields logged:
  - timestamp → when the event occurred
  - module → which component generated the event (PEP, ARM, etc.)
  - event_type → type of event (ACCESS_REQUEST, REMEDIATION, etc.)
  - user, resource, cloud, decision, reason → context of the event

    - actions_taken → for remediations
- Logs are stored in a **dedicated file**, e.g., logs/ztmc_framework_log.json.
- Optional: separate log files per module if needed.

Metrics Aggregation

- The module maintains **counters for key metrics**, including:
    - Total access requests (total_access_requests)
    - Counts per decision type (allow_count, deny_count, review_count)
    - Total remediations executed (total_remediations)
    - Breakdown per cloud (AWS, Azure, GCP)
    - Breakdown per event type (events_by_type)
- Metrics are **persisted in a separate file**, e.g., logs/ztmc_framework_metrics.json.

Thread-Safe Updates

- Uses a **lock** to ensure metrics are updated safely if multiple modules write at the same time.
- Ensures logs and metrics are **consistent and accurate**.

Code:

```python
# monitoring.py
"""
Centralized Logging and Monitoring for Zero Trust Multi-Cloud Framework
"""

import json
from datetime import datetime as dt
from threading import Lock

# Thread-safe counters
monitoring_lock = Lock()

# Monitoring metrics
metrics = {
    "total_access_requests": 0,
    "deny_count": 0,
    "review_count": 0,
    "allow_count": 0,
    "total_remediations": 0,
    "per_cloud": {"AWS": 0, "Azure": 0, "GCP": 0},
    "events_by_type": {}
}

# Central log file
LOG_FILE = "ztmc_framework_log.json"
METRICS_FILE = "ztmc_framework_metrics.json"

# ----------------------------
# Logging function
# ----------------------------
def log_event(module, event_type, user, resource, cloud, decision=None, reason=None, actions=None, details=None):
    """
    Logs an event from any module.
```

```
    module: source module name (PEP, PDP, ARM, IAM, IaC)
    event_type: type of event (ACCESS_REQUEST, POLICY_CHANGE, REMEDIATION, etc.)
    decision: DENY/ALLOW/REVIEW if relevant
    """
    timestamp = dt.utcnow().isoformat()
    entry = {
        "timestamp": timestamp,
        "module": module,
        "event_type": event_type,
        "user": user,
        "resource": resource,
        "cloud": cloud,
        "decision": decision,
        "reason": reason,
        "actions_taken": actions or [],
        "details": details or {}
    }

    # Write to log file
    with open(LOG_FILE, "a") as f:
        f.write(json.dumps(entry) + "\n")

    # Update metrics
    update_metrics(entry)

# ----------------------------
# Metrics updater
# ----------------------------
def update_metrics(entry):
    with monitoring_lock:
        metrics["total_access_requests"] += 1 if entry["event_type"] == "ACCESS_REQUEST" else 0
```

```
        metrics["total_access_requests"] += 1 if entry["event_type"] == "ACCESS_REQUEST" else 0
        metrics["total_remediations"] += 1 if entry["event_type"] == "REMEDIATION" else 0

        # Count by decision
        if entry.get("decision") == "DENY":
            metrics["deny_count"] += 1
        elif entry.get("decision") == "REVIEW":
            metrics["review_count"] += 1
        elif entry.get("decision") == "ALLOW":
            metrics["allow_count"] += 1

        # Count per cloud
        cloud = entry.get("cloud")
        if cloud in metrics["per_cloud"]:
            metrics["per_cloud"][cloud] += 1

        # Count events by type
        event_type = entry.get("event_type")
        metrics["events_by_type"].setdefault(event_type, 0)
        metrics["events_by_type"][event_type] += 1

        # Save metrics
        with open(METRICS_FILE, "w") as f:
            f.write(json.dumps(metrics, indent=2))

# ----------------------------
# Function to push metrics snapshot
# ----------------------------
def get_metrics_snapshot():
    with monitoring_lock:
        return metrics.copy()
```

Output:

```
PS C:\Users\harsh\OneDrive\Desktop> python pep.py bob s3:GetObject arn:aws:s3:::secure-bucket 10.0.0.50 device-laptop-002
=== Raw PDP Output ===
[PDP] Decision for bob -> REVIEW (Reason: Unrecognized device (device-laptop-002))

====================
[PEP] Access request by bob for arn:aws:s3:::secure-bucket -> REVIEW (Unrecognized device (device-laptop-002)))
[PEP] Access under manual review ⚠️
[ARM] Auto-Remediation Log: {'timestamp': '2025-10-25T10:43:39.572352', 'user': 'bob', 'resource': 'arn:aws:s3:::secure-bucket', 'decision': 'REVIEW', 'reason': ': 'Unrecognized device
: 'Unrecognized device (device-laptop-002))', 'cloud': 'AWS', 'actions_taken': ['Admin review needed for bob on arn:aws:s3:::secure-bucket: Unrecognized device (device-laptop-002))']}
        python pep.py eve compute:Start vm-prod 203.0.113.5 device-desktop-007
=== Raw PDP Output ===rive\Desktop>
[PDP] Decision for eve -> DENY (Reason: Untrusted network source (203.0.113.5))

====================
[PEP] Access request by eve for vm-prod -> DENY (Untrusted network source (203.0.113.5)))
[PEP] Request blocked ❌
[ARM] Auto-Remediation Log: {'timestamp': '2025-10-25T10:44:06.166480', 'user': 'eve', 'resource': 'vm-prod', 'decision': 'DENY', 'reason': 'Untrusted network source (203.0.113.5))', 'c
loud': 'GCP', 'actions_taken': ['GCP remediation triggered for eve']}
PS C:\Users\harsh\OneDrive\Desktop> python pep.py alice s3:GetObject arn:aws:s3:::secure-bucket 192.168.1.12 device-laptop-001
=== Raw PDP Output ===
[PDP] Decision for alice -> ALLOW (Reason: Context validated)

====================
[PEP] Access request by alice for arn:aws:s3:::secure-bucket -> ALLOW (Context validated))
[PEP] Access granted ✅
PS C:\Users\harsh\OneDrive\Desktop>
```

Logs (ztmc_framework_log.json)**:**


```
{"timestamp": "2025-10-25T10:39:59.702067", "module": "PEP", "event_type": "ACCESS_REQUEST", "user": "alice", "resource": "arn:aws:s3:::secure-bucket", "cloud":
{"timestamp": "2025-10-25T10:43:39.571346", "module": "PEP", "event_type": "ACCESS_REQUEST", "user": "bob", "resource": "arn:aws:s3:::secure-bucket", "cloud": "
{"timestamp": "2025-10-25T10:44:06.164477", "module": "PEP", "event_type": "ACCESS_REQUEST", "user": "eve", "resource": "vm-prod", "cloud": "GCP", "decision": "
{"timestamp": "2025-10-25T10:44:21.273365", "module": "PEP", "event_type": "ACCESS_REQUEST", "user": "alice", "resource": "arn:aws:s3:::secure-bucket", "cloud":
```



```
", "cloud": "AWS", "decision": "ALLOW", "reason": "Context validated)", "actions_taken": [], "details": {}}
 "cloud": "AWS", "decision": "REVIEW", "reason": "Unrecognized device (device-laptop-002))", "actions_taken": [], "details": {}}
decision": "DENY", "reason": "Untrusted network source (203.0.113.5))", "actions_taken": [], "details": {}}
", "cloud": "AWS", "decision": "ALLOW", "reason": "Context validated)", "actions_taken": [], "details": {}}
```


Metrics (ztmc_framework_metrics.json):


```
{
  "total_access_requests": 1,
  "deny_count": 0,
  "review_count": 0,
  "allow_count": 1,
  "total_remediations": 0,
  "per_cloud": {
    "AWS": 1,
    "Azure": 0,
    "GCP": 0
  },
  "events_by_type": {
    "ACCESS_REQUEST": 1
  }
}
```