THOMAS MORE

Lab 5.6

Lightweight Anomaly Detection with KQL Queries

Toepasbare Cloud Security voor IT-Professionals

2025 - 2026

Contents

1. Lightweight Anomaly Detection with KQL Queries

1.1. Context

In Labs 5.1 through 5.4, you audited compliance, fixed violations manually, enforced guardrails with Azure Policy, and enabled remediation patterns.

In this lab, you will use KQL queries in Log Analytics to detect suspicious or unusual behavior in application and platform logs.

This is not machine learning in the strict sense. It is a lightweight detection lab: you will create known traffic patterns, inspect the resulting logs, and write queries that highlight behavior that stands out from the normal baseline.

You will generate and inspect several types of demo signals:

- Normal health checks.
- Repeated failed requests.
- Probes for internal or sensitive-looking endpoints.
- Secret-demo requests that may trigger Key Vault access.
- A short burst of concurrent requests.
- Harmless management-plane changes in Azure.

> **Note**
>
> The Day 5 infrastructure should have Application Insights connected to the App Services and a Log Analytics workspace available. Before starting, make sure the apps are deployed and that diagnostic settings are enabled for the resources you want to query.

> **Note**
>
> Important: deploy the apps before generating demo traffic. From the Day 5 folder, run `./deploy-apps.ps1 -StudentPrefix <your_prefix>`. Also make sure Key Vault diagnostic settings were enabled in an earlier lab if you want to query Key Vault audit logs.

1.2. Learning goals

By the end of this lab, you should be able to explain:

- How to use KQL to detect suspicious patterns in log data.
- How application traffic anomalies can appear as error spikes, endpoint probing, or request bursts.
- How to use `where`, `summarize`, `top`, `bin`, `join`, and `render` in detection queries.
- Why logs and metrics are both useful, but answer different questions.
- How to export Azure Activity Logs to Log Analytics.
- How to correlate management-plane activity with application telemetry.
- Which prerequisites are needed before Key Vault and Microsoft Entra sign-in logs appear in Log Analytics.
- How to turn individual KQL queries into a simple Azure Monitor Workbook.
- How to create a query-based alert for a sudden increase in application errors.

1.3. Estimated time

85 minutes

Suggested timing:

Time	Activity
10 minutes	Deploy apps to App Services
10 minutes	Wait for deployment and telemetry ingestion
10 minutes	Enable or verify Activity Log and resource diagnostic settings
10 minutes	Create observable demo events
10 minutes	Query application errors and request bursts
10 minutes	Correlate management-plane activity with app errors
10 minutes	Inspect Key Vault or sign-in logs if available
10 minutes	Build a lightweight Azure Monitor Workbook
10 minutes	Create a query-based alert for a sudden error increase
5 minutes	Reflection and discussion

1.4. Before you start

Make sure you have:

- [ ] Day 5 infrastructure deployed.
- [ ] Apps deployed to App Services.
- [ ] Access to the Log Analytics workspace.
- [ ] Application Insights connected to the App Services.
- [ ] Subscription Activity Log export enabled for `AzureActivity` queries.
- [ ] App Service diagnostic settings enabled if you want App Service resource logs.
- [ ] Key Vault diagnostic settings enabled if you want to query Key Vault audit logs.
- [ ] Microsoft Entra diagnostic settings enabled if you want to query `SigninLogs`.

> **Note**
>
> Not every table in this lab is guaranteed to contain data in every tenant. `AppRequests` and `AppExceptions` depend on Application Insights telemetry. Key Vault audit logs depend on Key Vault diagnostic settings. `SigninLogs` depend on Microsoft Entra diagnostic settings. `AzureActivity` depends on exporting the subscription Activity Log to Log Analytics.

2. Deploy the Apps

The Day 5 infrastructure creates the App Services, but the applications must still be deployed.

From your repository root, run:

```
cd day-5
```

Then deploy the apps:

```
./deploy-apps.ps1 -StudentPrefix <your_prefix>
```

Replace `<your_prefix>` with the same prefix you used when deploying the infrastructure.

The script should read the App Service names from Terraform outputs, package the backend applications, and deploy them to Azure App Service. Azure App Service will perform the remote build, so you do not need to run `npm install` locally for the backends.

After deployment, verify that the public backend is running:

```
az rest --method get --url "https://app-cloudsec-<your_prefix>-dev-api-
d5.azurewebsites.net/api/health"
```

If the health check returns a `200` response and a small JSON body such as `{"status":"ok"}`, the app is running.

> **Note**
>
> Remote builds and telemetry ingestion take time. If the health check fails immediately after deployment, wait a few minutes and try again. After generating traffic, wait another 5-10 minutes before assuming the logs are missing.

3. Enable Management and Resource Logs

This lab uses several different log sources. They do not all arrive in the same table.

Signal	How to enable it	Typical table
Application requests	Application Insights connected to the App Service	`AppRequests`
Application exceptions	Application Insights connected to the App Service	`AppExceptions`
Azure management-plane events	Export the subscription Activity Log to Log Analytics	`AzureActivity`
App Service resource logs	Diagnostic settings on each App Service	`AzureDiagnostics` or App Service-specific tables
Key Vault audit events	Diagnostic settings on the Key Vault	`AZKVAuditLogs` or `AzureDiagnostics`
Microsoft Entra sign-ins	Microsoft Entra diagnostic settings	`SigninLogs`

3.1. Enable Subscription Activity Log Export

`AzureActivity` contains Azure Activity Log records. These are management-plane events such as creating, updating, starting, stopping, or changing Azure resources.

Activity Log data does not automatically appear in every Log Analytics workspace. You must export the subscription Activity Log to the workspace.

In the Azure portal:

- Go to **Monitor**.
- Open **Activity log**.
- Select **Export Activity Logs**.
- Select your lab subscription.
- Choose **Add diagnostic setting**.
- Select at least **Administrative**.
- Send the logs to the lab **Log Analytics workspace**.
- Save the setting.

Useful reference:

https://learn.microsoft.com/en-us/azure/azure-monitor/platform/activity-log

> **Note**
>
> `AzureActivity` is subscription Activity Log data. Resource-level diagnostic settings on an App Service, Key Vault, or resource group are useful too, but they do not replace subscription Activity Log export.

3.2. Enable App Service Resource Logs

You should also enable diagnostic settings on each App Service individually if you want to inspect App Service resource logs.

In the Azure portal:

- Go to the App Service.
- Open **Diagnostic settings**.
- Select **Add diagnostic setting**.
- Select useful categories such as HTTP logs, console logs, application logs, audit logs, and platform logs.
- Send the logs to the lab Log Analytics workspace.
- Save the setting.

This captures resource-level signals that are different from application telemetry and subscription Activity Log records.

> **Note**
>
> Existing portal configuration alone is not enough for the later queries to show interesting results. Create a fresh management-plane event, generate application traffic, and optionally trigger a Key Vault read.

4. Create Observable Events for the Queries

The queries in this lab only make sense if there is recent data to investigate. Before running the detection queries, create a few harmless events that should appear in Log Analytics.

This section creates three types of signals:

- Application requests in `AppRequests`.
- Management-plane changes in `AzureActivity`.
- Optional Key Vault access events in `AZKVAuditLogs` or `AzureDiagnostics`.

4.1. 1. Confirm the Log Analytics Workspace

First, identify the workspace used by this lab.

In the Azure portal:

- Go to your lab resource group.
- Open the **Log Analytics workspace**.
- Keep this workspace open in a separate tab.
- Use this workspace for all KQL queries in this lab.

Run this quick discovery query:

```
search *
| where TimeGenerated > ago(2h)
| summarize Count = count(), LastSeen = max(TimeGenerated) by $table
| sort by LastSeen desc
```

This query shows which tables currently contain data. It is useful when a table name in the lab is empty because it helps you find where the data is actually arriving.

4.2. 2. Create a Management-Plane Event

After enabling Activity Log export, create a harmless resource change. This gives the `AzureActivity` queries something concrete to find.

Replace the resource group name with your own lab resource group:

```
az group update \
  --name <your-resource-group> \
  --set tags.Lab56ActivityTest=enabled
```

Then wait a few minutes and run:

```
AzureActivity
| where TimeGenerated > ago(1h)
| project TimeGenerated, OperationNameValue, ResourceGroup, ResourceProviderValue,
Resource, Caller, ActivityStatusValue
| sort by TimeGenerated desc
| take 20
```

If this returns results, the Activity Log export is working.

If it does not return results, check the Activity Log directly in the portal:

- Go to **Monitor → Activity log**.
- Filter to the last hour.
- Filter to your lab resource group.
- Look for the resource group tag update.

If the event is visible in **Monitor → Activity log** but not in `AzureActivity`, the most likely causes are:

- The Activity Log export points to a different Log Analytics workspace.
- The diagnostic setting was created at the wrong scope.
- The event has not been ingested yet.

4.3. 3. Generate Application Traffic

Before writing detection queries, generate predictable traffic that gives you something to investigate.

Set your backend URL:

```
$BACKEND_URL = "https://app-cloudsec-<your-prefix>-dev-api-d5.azurewebsites.net"
```

Run the traffic generator:

```
./scripts/traffic-gen.ps1
```

> **Note**
>
> The script does not prove that an attack happened. It creates controlled patterns that resemble things defenders often investigate: repeated errors, endpoint probing, sensitive endpoint access, and request bursts.

Then confirm that request telemetry is arriving:

```
AppRequests
| where TimeGenerated > ago(1h)
| summarize RequestCount = count() by bin(TimeGenerated, 5m)
| render timechart
```

Inspect the latest requests:

```
AppRequests
| where TimeGenerated > ago(1h)
| project TimeGenerated, Name, Url, ResultCode, Success, OperationId
| sort by TimeGenerated desc
| take 20
```

4.4. 4. Optional: Create a Key Vault Event

This step only works if Key Vault diagnostic settings are enabled and send audit logs to the same Log Analytics workspace.

Trigger a harmless Key Vault read operation:

```
az keyvault secret show \
  --vault-name <your-key-vault-name> \
  --name <your-demo-secret-name>
```

Then try the resource-specific table first:

```
AZKVAuditLogs
| where TimeGenerated > ago(1h)
| project TimeGenerated, OperationName, ResultType, CallerIpAddress, Identity
| sort by TimeGenerated desc
| take 20
```

If `AZKVAuditLogs` is empty, your Key Vault diagnostic setting may be using the legacy Azure diagnostics destination mode. In that case, use:

```
AzureDiagnostics
| where TimeGenerated > ago(1h)
| where ResourceProvider == "MICROSOFT.KEYVAULT"
| where Category == "AuditEvent" or Category == "KeyVaultAuditLogs"
| project TimeGenerated, OperationName, ResultType, ResultSignature, CallerIPAddress,
ResourceId
| sort by TimeGenerated desc
| take 20
```

> **Note**
>
> Key Vault logs can appear in `AZKVAuditLogs` when resource-specific logging is used, or in `AzureDiagnostics` when Azure diagnostics mode is used. Both are valid, but the table name depends on the diagnostic setting.

5. Confirm Telemetry Is Arriving

Open the Log Analytics workspace connected to your Application Insights resource.

Run this query first:

```
AppRequests
| where TimeGenerated > ago(1h)
| summarize RequestCount = count() by bin(TimeGenerated, 5m)
| render timechart
```

If you see no results, wait a few more minutes and try again.

You can also inspect the most recent requests:

```
AppRequests
| where TimeGenerated > ago(1h)
| project TimeGenerated, Name, Url, ResultCode, Success, OperationId
| sort by TimeGenerated desc
| take 20
```

> **Note**
>
> `AppRequests` contains request telemetry from Application Insights. Useful columns include `TimeGenerated`, `Name`, `Url`, `ResultCode`, `Success`, and `OperationId`.

If a specific table is empty, first check whether the workspace is receiving any data at all:

```
search *
| where TimeGenerated > ago(2h)
| summarize Count = count(), LastSeen = max(TimeGenerated) by $table
| sort by LastSeen desc
```

Then decide which query to use:

If you see this table	Use it for
`AppRequests`	Application request traffic, errors, endpoints, and bursts
`AppExceptions`	Application exceptions
`AzureActivity`	Subscription Activity Log / management-plane changes
`AZKVAuditLogs`	Key Vault audit events using resource-specific logging
`AzureDiagnostics`	Legacy resource logs, including Key Vault logs in Azure diagnostics mode
`SigninLogs`	Microsoft Entra sign-in activity

> **Note**
>
> Do not assume that an empty table means logging is broken. It may mean that no matching event has happened yet, the data is in another table, or the diagnostic setting uses a different destination mode.

6. Query for Error Spikes

The first detection query looks for repeated failed requests.

Repeated `404`, `403`, or `500` responses can indicate scanning, broken clients, failed deployments, or real attack traffic. The query does not decide the cause by itself; it helps you find the pattern.

6.1. HTTP 4xx and 5xx Responses

```
AppRequests
| where TimeGenerated > ago(1h)
| where toint(ResultCode) >= 400
| summarize ErrorCount = count() by bin(TimeGenerated, 5m), ResultCode
| render timechart
```

6.2. Top Failing URLs

```
AppRequests
| where TimeGenerated > ago(1h)
| where toint(ResultCode) >= 400
| summarize ErrorCount = count() by Url, ResultCode
| top 20 by ErrorCount desc
```

6.3. Exceptions

Some failed requests produce HTTP error codes but not application exceptions. Query both.

```
AppExceptions
| where TimeGenerated > ago(1h)
| summarize ExceptionCount = count() by bin(TimeGenerated, 5m), ProblemId
| render timechart
```

Inspect recent exceptions:

```
AppExceptions
| where TimeGenerated > ago(1h)
| project TimeGenerated, ProblemId, ExceptionType, Message, OperationName,
OperationId
| sort by TimeGenerated desc
| take 20
```

> **Question**
>
> Which error pattern is easiest to spot: the repeated 404s, application exceptions, or both? Why might failed requests not always appear in `AppExceptions`?

7. Detect Unusual Endpoint Access Patterns

The next queries look for endpoints that were accessed unusually often or contain sensitive-looking path names.

7.1. Top Endpoints

```
AppRequests
| where TimeGenerated > ago(1h)
| extend Path = tostring(parse_url(Url).Path)
| summarize RequestCount = count(), FailedCount = countif(tobool(Success) == false)
by Path
| top 20 by RequestCount desc
```

7.2. Suspicious Path Names

```
AppRequests
| where TimeGenerated > ago(1h)
| extend Path = tostring(parse_url(Url).Path)
| where Path has_any ("internal", "admin", "secret", "secrets", "config", "env",
"cfg")
| summarize
    RequestCount = count(),
    FailedCount = countif(tobool(Success) == false),
    ResultCodes = make_set(ResultCode)
  by Path
| sort by RequestCount desc
```

> **Note**
>
> This is keyword-based detection. It is intentionally simple. In production, you would tune the query for the actual application routes, expected callers, and known deployment behavior.

> **Question**
>
> Which endpoints appear suspicious in your results? Are they suspicious because of the path name, the number of requests, the result code, or the context?

8. Detect Request Bursts

A short burst of traffic may be normal during testing, but it can also indicate abuse, brute force behavior, scraping, or a denial-of-wallet scenario.

```
AppRequests
| where TimeGenerated > ago(1h)
| summarize RequestCount = count() by bin(TimeGenerated, 1m)
| render timechart
```

Show the busiest minutes:

```
AppRequests
| where TimeGenerated > ago(1h)
| summarize RequestCount = count() by bin(TimeGenerated, 1m)
| top 10 by RequestCount desc
```

> **Question**
>
> Can you identify the minute in which you ran the burst traffic? Would this query be useful in production as-is, or would you need a baseline first?

9. Correlate Activity Log Events with Application Errors

Azure Activity Log records management-plane operations, such as changes to Azure resources. Application Insights records application-level telemetry, such as requests and exceptions.

A useful investigation question is:

“Did a management action happen shortly before or during an application error spike?”

9.1. Recent Management Events

```
AzureActivity
| where TimeGenerated > ago(1h)
| where OperationNameValue has_any ("Microsoft.Web", "Microsoft.KeyVault",
"Microsoft.Storage", "Microsoft.Authorization", "UpdateWebSite")
| project TimeGenerated, OperationNameValue, ResourceGroup, ResourceProviderValue,
Resource, Caller, ActivityStatusValue
| sort by TimeGenerated desc
| take 20
```

9.2. Time-Window Correlation

Do not join on exact timestamps. Exact timestamps rarely match across tables. Instead, bin events into time windows and compare activity within the same window.

```
let activityEvents =
    AzureActivity
    | where TimeGenerated > ago(1h)
    | where OperationNameValue has_any ("Microsoft.Web", "Microsoft.KeyVault",
"Microsoft.Storage", "Microsoft.Authorization", "UpdateWebSite")
    | summarize ManagementEvents = count(), Operations =
make_set(OperationNameValue), Callers = make_set(Caller)
      by TimeWindow = bin(TimeGenerated, 5m);
let appErrors =
    AppRequests
    | where TimeGenerated > ago(1h)
    | where toint(ResultCode) >= 400
    | summarize AppErrors = count(), ResultCodes = make_set(ResultCode)
      by TimeWindow = bin(TimeGenerated, 5m);
activityEvents
| join kind=inner appErrors on TimeWindow
| project TimeWindow, ManagementEvents, Operations, Callers, AppErrors, ResultCodes
| sort by TimeWindow desc
```

> **Note**
>
> This query shows correlation, not causation. If a resource change and an error spike occur in the same time window, that is a reason to investigate further. It does not automatically prove that the change caused the errors.

Question

> Did any management-plane events occur near the same time as your application errors? What extra evidence would you need before blaming a deployment or configuration change?

10. Detect Key Vault Access Patterns

This section only works if Key Vault diagnostic settings send audit logs to the Log Analytics workspace.

Modern resource-specific logging commonly uses the `AZKVAuditLogs` table. Some environments may still route logs to `AzureDiagnostics`. Try the first query first.

10.1. Key Vault Audit Events

```
AZKVAuditLogs
| where TimeGenerated > ago(1h)
| project TimeGenerated, OperationName, ResultType, CallerIpAddress, Identity
| sort by TimeGenerated desc
| take 20
```

10.2. Failed Key Vault Operations

```
AZKVAuditLogs
| where TimeGenerated > ago(1h)
| where ResultType !in ("Success", "Succeeded")
| project TimeGenerated, OperationName, ResultType, CallerIpAddress, Identity
| sort by TimeGenerated desc
```

10.3. Fallback for AzureDiagnostics

Use this if your Key Vault logs are routed to the legacy `AzureDiagnostics` table.

```
AzureDiagnostics
| where TimeGenerated > ago(1h)
| where ResourceProvider == "MICROSOFT.KEYVAULT"
| where Category == "AuditEvent" or Category == "KeyVaultAuditLogs"
| project TimeGenerated, OperationName, ResultType, ResultSignature, CallerIPAddress,
ResourceId
| sort by TimeGenerated desc
| take 20
```

10.4. Table-Agnostic Key Vault Query

If you are not sure which table is being used, try this version:

```
union isfuzzy=true AzureDiagnostics, AZKVAuditLogs
| where TimeGenerated > ago(24h)
| where ResourceProvider has "KEYVAULT"
   or ResourceProvider == "MICROSOFT.KEYVAULT"
   or _ResourceId has "Microsoft.KeyVault/vaults"
| order by TimeGenerated desc
| take 50
```

> **Question**
>
> Which Key Vault operations are visible? Which operation would be suspicious if it came from an unexpected identity, location, or application path?

11. Example: Detect Unusual Sign-in Patterns

This section is optional because Microsoft Entra sign-in logs are tenant-level logs. They do not appear just because Key Vault logging is enabled.

You need Microsoft Entra diagnostic settings that send sign-in logs to the Log Analytics workspace, and you need sufficient tenant permissions to query them.

For this reason, you might not have access to them. If you do, try these queries. If not, just read through them to see how sign-in patterns can be analyzed with KQL.

11.1. Failed Sign-ins

```
SigninLogs
| where TimeGenerated > ago(24h)
| where ResultType != "0"
| summarize FailedSignIns = count() by UserPrincipalName, IPAddress, AppDisplayName,
ResultType
| sort by FailedSignIns desc
```

11.2. Azure CLI or PowerShell Sign-in Failures

```
SigninLogs
| where TimeGenerated > ago(24h)
| where AppDisplayName has_any ("Azure CLI", "Azure PowerShell", "Microsoft Azure
PowerShell")
| where ResultType != "0"
| summarize FailedSignIns = count() by UserPrincipalName, IPAddress, AppDisplayName,
ResultType
| sort by FailedSignIns desc
```

11.3. Group Related Sign-in Records

Log Analytics may show multiple records for a single interactive sign-in flow. Grouping by correlation ID can make investigation clearer.

```
SigninLogs
| where TimeGenerated > ago(24h)
| summarize
    FirstSeen = min(TimeGenerated),
    LastSeen = max(TimeGenerated),
    Attempts = count(),
    FinalResult = arg_max(TimeGenerated, ResultType, ConditionalAccessStatus)
  by CorrelationId, UserPrincipalName, IPAddress, AppDisplayName
| sort by LastSeen desc
```

> **Note**
>
> `SigninLogs` are useful for identity investigation, but they are not produced by Key Vault. They come from Microsoft Entra ID and must be routed separately to Log Analytics.

Question

> What would make a sign-in pattern suspicious: the number of failures, the application used, the IP address, the location, the time of day, or the account involved?

12. Build a Lightweight Detection Workbook

So far, you have run individual KQL queries. In practice, defenders rarely want to rerun every query manually. They often collect useful queries into a dashboard or workbook.

In this section, you will create a simple Azure Monitor Workbook that visualizes the most important signals from this lab.

The workbook should help answer four questions:

- Is application traffic arriving?
- Are errors increasing?
- Which endpoints look unusual?
- Did management-plane activity happen near application errors?

12.1. Create the Workbook

In the Azure portal:

- Go to **Monitor**.
- Open **Workbooks**.
- Select **Empty**.
- Select **Edit**.
- Give the workbook a name such as `Lab 5.6 - Lightweight Detection`.

Useful reference:

https://learn.microsoft.com/en-us/azure/azure-monitor/visualize/workbooks-overview

> **Note**
>
> A workbook is better than a static dashboard for this lab because it can combine explanatory text, KQL queries, tables, charts, and parameters in one place.

12.2. Add a Time Range Parameter

Add a parameter so you can reuse the same workbook for different time windows.

In the workbook editor:

- Select **Add → Add parameters**.
- Add a parameter named `TimeRange`.
- Use parameter type **Time range picker**.
- Save the parameter.

When adding each query below, set the query control's time range to the `TimeRange` parameter.

12.3. Tile 1: Request Count

Add a query tile that shows whether application telemetry is arriving.

In the workbook:

- Select **Add → Add query**.

- Data source: **Logs**.
- Resource type: **Log Analytics** or **Application Insights**, depending on where your `AppRequests` table is available.
- Select the lab workspace or Application Insights resource.
- Set the timerange to use our property parameter: `TimeRange`.
- Set the visualization to **Time chart**.

Use this query:

```
AppRequests
| summarize RequestCount = count() by bin(TimeGenerated, 5m)
| order by TimeGenerated asc
```

Suggested title:

```
Application request volume
```

This chart should show when you ran the demo traffic generator.

12.4. Tile 2: Error Count

Add a second query tile for failed requests.

Visualization: **Time chart**

```
AppRequests
| where toint(ResultCode) >= 400
| summarize ErrorCount = count() by bin(TimeGenerated, 5m), ResultCode
| order by TimeGenerated asc
```

Suggested title:

```
HTTP 4xx and 5xx responses
```

This makes repeated failed requests easier to spot than reading individual log rows.

12.5. Tile 3: Top Suspicious Paths

Add a grid that shows suspicious-looking endpoint access.

Visualization: **Grid**

```
AppRequests
| extend Path = tostring(parse_url(Url).Path)
| where Path has_any ("internal", "admin", "secret", "secrets", "config", "env",
"cfg")
| summarize
    RequestCount = count(),
    FailedCount = countif(tobool(Success) == false),
    ResultCodes = make_set(ResultCode)
  by Path
| sort by RequestCount desc
```

Suggested title:

```
Suspicious endpoint names
```

> **Note**
>
> This is not a real incident detector yet. It is a visibility panel that highlights endpoint names worth investigating.

12.6. Tile 4: Busiest Minutes

Add a grid that shows request bursts.

Visualization: **Grid**

```
AppRequests
| summarize RequestCount = count() by TimeWindow = bin(TimeGenerated, 1m)
| top 10 by RequestCount desc
```

Suggested title:

```
Busiest request minutes
```

This tile helps students identify the minute in which the burst traffic was generated.

12.7. Tile 5: Recent Management Activity

Add a grid for Azure Activity Log events.

Visualization: **Grid**

```
AzureActivity
| where OperationNameValue has_any ("Microsoft.Web", "Microsoft.KeyVault",
"Microsoft.Storage", "Microsoft.Authorization", "UpdateWebSite")
| project TimeGenerated, OperationNameValue, ResourceGroup, ResourceProviderValue,
Resource, Caller, ActivityStatusValue
| sort by TimeGenerated desc
| take 20
```

Suggested title:

```
Recent management-plane activity
```

> **Note**
>
> If this tile is empty, check whether the subscription Activity Log is exported to the same Log Analytics workspace and whether you created a recent management-plane event.

12.8. Tile 6: Management Events Near App Errors

Add a grid that correlates management-plane events and application errors in the same 5-minute window.

Visualization: **Grid**

```
let activityEvents =
    AzureActivity
    | where OperationNameValue has_any ("Microsoft.Web", "Microsoft.KeyVault",
"Microsoft.Storage", "Microsoft.Authorization", "UpdateWebSite")
    | summarize
```

```
        ManagementEvents = count(),
        Operations = make_set(OperationNameValue),
        Callers = make_set(Caller)
      by TimeWindow = bin(TimeGenerated, 5m);
let appErrors =
    AppRequests
    | where toint(ResultCode) >= 400
    | summarize
        AppErrors = count(),
        ResultCodes = make_set(ResultCode)
      by TimeWindow = bin(TimeGenerated, 5m);
activityEvents
| join kind=inner appErrors on TimeWindow
| project TimeWindow, ManagementEvents, Operations, Callers, AppErrors, ResultCodes
| sort by TimeWindow desc
```

Suggested title:

```
Management activity near application errors
```

This tile is useful during investigation, but it still shows correlation rather than causation.

12.9. Optional Tile: Key Vault Audit Events

Add this tile only if Key Vault diagnostic settings are enabled.

Visualization: **Grid**

Try the resource-specific table first:

```
AZKVAuditLogs
| project TimeGenerated, OperationName, ResultType, CallerIpAddress, Identity
| sort by TimeGenerated desc
| take 20
```

If your environment sends Key Vault logs to the legacy diagnostics table, use this version instead:

```
AzureDiagnostics
| where ResourceProvider == "MICROSOFT.KEYVAULT"
| where Category == "AuditEvent" or Category == "KeyVaultAuditLogs"
| project TimeGenerated, OperationName, ResultType, ResultSignature, CallerIPAddress,
ResourceId
| sort by TimeGenerated desc
| take 20
```

Suggested title:

```
Key Vault audit events
```

12.10. Save the Workbook

When the workbook is useful:

- Select **Done editing**.
- Select **Save**.
- Save it in the lab resource group if possible.
- Use a clear name such as `Lab 5.6 - Lightweight Detection Workbook`.

> **Question**
>
> Which workbook tile would be most useful during a real investigation? Which tile would you convert into an alert rule, and which tile is mainly useful for manual exploration?

13. Create an Alert for a Sudden Error Increase

A workbook helps you investigate, but an alert helps you notice a problem without manually checking the workbook.

In this section, you will create a log search alert that compares two time windows:

- The last 10 minutes.
- The 30 minutes before that.

The alert should fire when the number of HTTP errors suddenly increases compared with the previous baseline.

Useful reference:

https://learn.microsoft.com/en-us/azure/azure-monitor/alerts/alerts-create-log-alert-rule

13.1. Test the Detection Query

Open the Log Analytics workspace and run this query:

```
let RecentWindow = 10m;
let BaselineWindow = 30m;
let MinimumRecentErrors = 5;
let IncreaseFactor = 2.0;
let recentErrors =
    AppRequests
    | where TimeGenerated between (ago(RecentWindow) .. now())
    | where toint(ResultCode) >= 400
    | summarize RecentErrors = count();
let baselineErrors =
    AppRequests
    | where TimeGenerated between (ago(RecentWindow + BaselineWindow) ..
ago(RecentWindow))
    | where toint(ResultCode) >= 400
    | summarize BaselineErrors = count();
recentErrors
| extend JoinKey = 1
| join kind=inner (baselineErrors | extend JoinKey = 1) on JoinKey
| extend
    BaselinePer10Minutes = todouble(BaselineErrors) / (todouble(BaselineWindow) /
todouble(RecentWindow)),
    IncreaseRatio = todouble(RecentErrors) / iif(BaselinePer10Minutes == 0.0, 1.0,
BaselinePer10Minutes)
| where RecentErrors >= MinimumRecentErrors
| where IncreaseRatio >= IncreaseFactor
| project
    TimeGenerated = now(),
    RecentErrors,
    BaselineErrors,
    BaselinePer10Minutes,
    IncreaseRatio
```

This query only returns a row when both conditions are true:

- There are at least `MinimumRecentErrors` in the recent window.

- The recent error count is at least `IncreaseFactor` times higher than the earlier baseline.

> **Note**
>
> The baseline is normalized to a 10-minute window. This makes the comparison fair even though the baseline window is longer than the recent window.

13.2. Understand the Alert Logic

The alert is intentionally simple:

Setting	Meaning
`RecentWindow = 10m`	Look at the last 10 minutes
`BaselineWindow = 30m`	Compare against the 30 minutes before the recent window
`MinimumRecentErrors = 5`	Avoid alerting on tiny numbers
`IncreaseFactor = 2.0`	Alert when recent errors are at least twice the baseline rate

Example:

- Previous 30 minutes: 6 errors.
- Normalized baseline: 2 errors per 10 minutes.
- Last 10 minutes: 8 errors.
- Increase ratio: 4.0.

This should trigger because the recent error count is both high enough and much higher than the baseline.

13.3. Create the Alert Rule in Azure Monitor

In the Azure portal:

- Go to the **Log Analytics workspace**.
- Open **Logs**.
- Paste and run the query above.
- Select **New alert rule**.
- Scope: the lab Log Analytics workspace.
- Condition: custom log search.
- Measurement: **Table rows** or **Number of results**.
- Operator: **Greater than**.
- Threshold value: `0`.
- Evaluation frequency: `5 minutes`.
- Lookback period: `45 minutes` or more.
- Severity: `3 - Informational` or `2 - Warning`.

The alert fires when the query returns at least one row.

> **Note**
>
> The query itself contains the detection logic. The alert rule only checks whether the query returned any rows.

13.4. Add an Action Group

For a classroom lab, you can either skip notification actions or use an email action group.

If you add an action group:

- Select **Create action group**.
- Use a clear name such as `lab56-error-increase-ag`.
- Add an email notification to your own address.
- Save the action group.

> **Warning**
>
> In a real environment, avoid noisy alerts. A sudden increase query should be tuned with realistic thresholds, suppression, severity, and routing rules.

13.5. Test the Alert

Run the traffic generator again, or intentionally create repeated failed requests.

Then wait for the alert evaluation cycle.

Use this query to check whether the alert condition would currently be true:

```
let RecentWindow = 10m;
let BaselineWindow = 30m;
let MinimumRecentErrors = 5;
let IncreaseFactor = 2.0;
let recentErrors =
    AppRequests
    | where TimeGenerated between (ago(RecentWindow) .. now())
    | where toint(ResultCode) >= 400
    | summarize RecentErrors = count();
let baselineErrors =
    AppRequests
    | where TimeGenerated between (ago(RecentWindow + BaselineWindow) ..
ago(RecentWindow))
    | where toint(ResultCode) >= 400
    | summarize BaselineErrors = count();
recentErrors
| extend JoinKey = 1
| join kind=inner (baselineErrors | extend JoinKey = 1) on JoinKey
| extend
    BaselinePer10Minutes = todouble(BaselineErrors) / (todouble(BaselineWindow) /
todouble(RecentWindow)),
    IncreaseRatio = todouble(RecentErrors) / iif(BaselinePer10Minutes == 0.0, 1.0,
BaselinePer10Minutes)
| project
    RecentErrors,
    BaselineErrors,
    BaselinePer10Minutes,
    IncreaseRatio,
    WouldAlert = RecentErrors >= MinimumRecentErrors and IncreaseRatio >=
IncreaseFactor
```

This version always returns one row and shows whether the alert condition would currently be true.

Question

Why is this alert better than a simple fixed threshold such as `errors > 10`? When could this alert still produce false positives?

14. Logs versus Metrics

Logs and metrics both help with detection, but they answer different questions.

Signal	Useful for	Example
Logs	Detailed investigation of events	`Which endpoint returned 404 and when?`
Metrics	Aggregated trends and thresholds	`Did request count spike above normal?`

> **Note**
>
> A production detection strategy usually combines both. Metrics are useful for fast threshold alerts. Logs are useful for investigation, context, and security detections.

15. Reflection Questions

Your answer:

1. Which query was most useful for identifying the demo traffic? Why?
2. Which generated traffic pattern was easiest to detect?
3. Which pattern could also happen during normal operations or deployment?
4. What extra context would you need before calling one of these patterns a real incident?
5. How did correlating AzureActivity events with application errors change your understanding of the traffic patterns?
6. Which workbook tile was most useful for investigation?
7. Which query would you convert into an Azure Monitor alert or Microsoft Sentinel analytics rule?
8. Why is a time-window comparison often better than a fixed threshold?

16. Summary

In this lab, you deployed the Day 5 apps, exported the subscription Activity Log to Log Analytics, generated controlled demo traffic, created harmless management-plane events, and used KQL to detect suspicious patterns across multiple log tables.

You queried:

- `AppRequests` for failed requests, suspicious endpoints, and request bursts.
- `AppExceptions` for application exceptions.
- `AzureActivity` for management-plane events after enabling Activity Log export.
- `AZKVAuditLogs` or `AzureDiagnostics` for Key Vault audit events, if enabled.
- `SigninLogs` for Microsoft Entra sign-in patterns, if enabled.

You also built a lightweight Azure Monitor Workbook and created a query-based alert that compares recent errors with a previous baseline window.

The most important lesson is that anomaly detection starts with context. A spike, failure, or unusual path is not automatically an incident. It is a signal that needs investigation, baselining, and correlation.

> **Note**
>
> In a production environment, useful KQL queries can be converted into Azure Monitor alert rules, workbook visualizations, or Microsoft Sentinel analytics rules. Security automation can then connect those detections to response actions such as notifications, ticket creation, or Logic App workflows.