# InTune Device Compliance Reporter

Contents

[InTune Device Compliance Reporter 1](#_Toc195283110)

[Introduction 2](#_Toc195283111)

[1. Initial Setup 2](#_Toc195283112)

[1.1. Local Repository 2](#_Toc195283113)

[1.2. Version Control 4](#_Toc195283114)

[2. Requirement Gathering 5](#_Toc195283115)

[2.1. Business Problem 5](#_Toc195283116)

[2.2. Objective 5](#_Toc195283117)

[2.3. Stakeholder Engagement 5](#_Toc195283118)

[2.4. Functional Requirements 6](#_Toc195283119)

[2.5. Technical Requirements 6](#_Toc195283120)

[3. Analysis and Initial Design 7](#_Toc195283121)

[3.1. Introduction 7](#_Toc195283122)

[3.2 MoSCoW 7](#_Toc195283123)

[3.3. Risks and Constraints 7](#_Toc195283124)

[3.4. Identify System Inputs, Processes, and Outputs 8](#_Toc195283125)

[3.5. High-Level Architecture 8](#_Toc195283126)

[3.6. Database Design 9](#_Toc195283127)

[3.7 Chosen Technologies 9](#_Toc195283128)

[4. Development 11](#_Toc195283129)

[4.1. Project Setup and Environment 11](#_Toc195283130)

[4.2. API Connection to Microsoft Graph 11](#_Toc195283131)

[4.3 Storing Credentials Securely with .env and .gitignore 12](#_Toc195283132)

[4.4 Data Retrieval and Authentication 14](#_Toc195283133)

[4.5. Storing the Data 19](#_Toc195283134)

[4.6. Querying the Data 23](#_Toc195283135)

[4.7. Optional Enhancements and Future Improvements 25](#_Toc195283136)

[5. Testing and Debugging 26](#_Toc195283137)

[5.1. Testing Strategy 26](#_Toc195283138)

[5.2. Functional Testing Examples 26](#_Toc195283139)

[5.3. Debugging Process 27](#_Toc195283140)

[6. Reflection 27](#_Toc195283141)

[6.1. What went well 27](#_Toc195283142)

[6.2. What I could have done differently 28](#_Toc195283143)

[6.3. Conclusion 28](#_Toc195283144)

## Introduction

This project, *Intune Device Compliance Reporter*, was initiated as part of the **A1.4 Apply** stage of my Level 4 Software Engineer apprenticeship with QA. It aims to automate the process of retrieving, processing, and reporting on device compliance data for Windows and iOS devices managed via Microsoft Intune- a cloud-based endpoint management solution used by Assura to secure and monitor company devices.

Currently, reporting on device compliance is a **manual, time-consuming task** performed periodically by the IT team. This project will reduce administrative overhead, improve accuracy, and provide timely insights to enable faster IT triage.

My goal is to develop a maintainable, testable, and structured Python-based tool that retrieves compliance data via the Microsoft Graph API, stores it in an SQL database, and outputs clear reports for IT analysis.

The project directly benefits my organisation while also allowing me to evidence key apprenticeship learning outcomes:

* **S1:** Writing logical and maintainable code
* **S4:** Unit testing and debugging
* **S7:** Structured problem solving
* **B4:** Collaborating with internal teams
* **B10:** Continuing professional development through real-world tooling and automation

## 1. Initial Setup

### 1.1. Local Repository

To begin, I designed a clean and modular folder structure to support separation of concerns during development. Each folder has a clearly defined role.

A screenshot of a computer

AI-generated content may be incorrect.

#### Folder-by-Folder Review

|  |  |
| --- | --- |
| **Folder** | **Purpose** |
| Database/ | Storage for my .db file (SQLite) and SQL schema files. |
| Docs/ | Great for requirement gathering notes, design diagrams, and SDLC write-up drafts. |
| Reports/ | Ideal for storing auto-generated output: CSVs, charts (from matplotlib), or summary logs. |
| Screenshots/ | Useful to evidencing progress along the way (e.g., API calls, testing, terminal outputs, debugging, stakeholder communication). |
| Scripts/ | My code will live here with separate modules for fetching, transforming, storing, etc. |
| Tests/ | Where I will write unit tests testing and debug my code. Could use pytest at a later point. |

#### README.md File Creation

A markdown-based README file was added to the root of the project as well as to each sub-folder. It includes a brief overview of the project’s purpose (or folder’s purpose), the technologies used, and how the solution aligns with my apprenticeship learning outcomes (S1, S4, S7, B4, B10). This also serves as documentation for anyone reviewing the code or assessing the project via GitHub.

A screenshot of a computer program

AI-generated content may be incorrect.

### 1.2. Version Control

To manage this project and demonstrate good version control practices, Git and GitHub were used. Git was used locally to initialise and track changes to the project, while GitHub served as the remote repository, allowing for secure cloud-based backup and versioning.

As part of this project:

* The default branch was renamed to **main**, following modern Git conventions.
* Commits were made regularly using **clear, meaningful messages** to describe changes.
* Each core folder (e.g. Docs, Scripts, Tests) includes a placeholder README.md file to ensure they are tracked from the start.

#### GitHub Repository

I created a new **private GitHub repository** titled **intune-device-compliance-reporter**. I made the repository private as it will include sensitive information such as client secrets from Azure.

#### Connecting Local Repository to GitHub

The newly created repository was linked to my local folder using Git, and the initial commit (including the full folder structure and README) was pushed successfully. This ensures all future code changes are tracked and version controlled, supporting maintainability and collaboration.

Subsequently, I leveraged Visual Studio Code’s built-in Source Control features to streamline visual management of changes.

Finally, I tested it had worked by navigating back to my GitHub repo page and confirming that I was able to see the folders and my README file.

## 2. Requirement Gathering

### 2.1. Business Problem

Currently, the process of extracting device compliance data from Microsoft Intune is manual, repetitive, and inefficient, relying on IT staff to log in, navigate the portal, and export reports manually. This slows down triage efforts, limits visibility of non-compliant devices, and introduces risk through inconsistent reporting intervals.

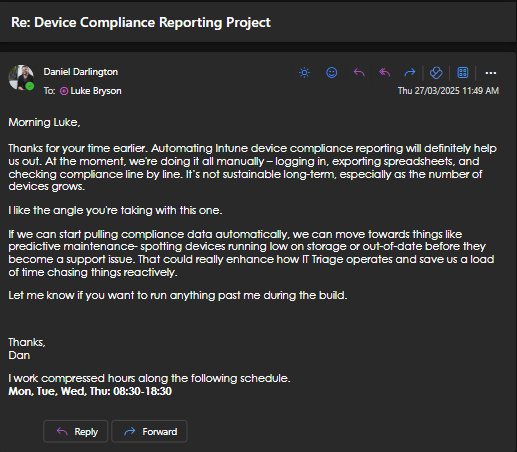
This process is time-consuming and does not support historical trend analysis or automated alerts. Due to its time-consuming nature, it is also only run on a monthly basis- I would like to change this to run weekly.

### 2.2. Objective

To design and build a Python-based tool that automates the retrieval, storage, and reporting of compliance data for Windows and iOS devices managed by Microsoft Intune, providing consistent and timely insights.

### 2.3. Stakeholder Engagement

To shape the requirements and ensure the project delivers value, I consulted with our Head of Digital & Transformation and IT Support Manager regarding reporting needs as well as discussing report structure and compliance categories.



*Figure A.1 – Email from IT Support Manager confirming manual process and value of predictive maintenance.*

A screenshot of a computer

AI-generated content may be incorrect.

*Figure A.2 – Email from Head of Digital Transformation with initial reporting requirements and strategic alignment.*

The project scope and plan were reviewed and approved by my Digital Learning Consultant (DLC) at QA.

### 2.4. Functional Requirements

|  |  |
| --- | --- |
| **Requirement** | **Description** |
| Retrieve Compliance Data | Use Microsoft Graph API to fetch device compliance data (platform, status, last check-in) |
| Store Data | Insert structured records into a SQL database for long-term reference |
| Generate Reports | Provide summaries such as number of compliant vs non-compliant devices, platform breakdown |
| Automate | Schedule periodic execution to keep data up to date |
| Alerting | Email notifications if non-compliance thresholds are exceeded |

### 2.5. Technical Requirements

|  |  |
| --- | --- |
| **Area** | **Technology / Decision** |
| Language | Python 3 |
| API Access | Microsoft Graph API with OAuth 2.0 |
| Data Storage | SQLite (lightweight, file-based SQL database) |
| Data Processing | Python 3 (built-in libraries) |
| Visual Reporting | Power BI |
| Security | Use .env file to store sensitive API credentials securely |

## 3. Analysis and Initial Design

### 3.1. Introduction

Before starting development, I analysed the functional and non-functional requirements gathered from stakeholders, along with the technical constraints and project scope. This helped me structure the solution around high-priority features, identify potential blockers early, and align the development effort to Assura’s needs.

### 3.2 MoSCoW

To prioritise the project requirements, I applied the MoSCoW method, a well-known Agile technique used to categorise features based on their importance to project success:

* Must Have – Essential features required for the solution to function
* Should Have – Important features that add value but aren't critical
* Could Have – Nice-to-have additions if time and resources permit
* Won’t Have – Explicitly excluded from this phase or project scope

This helped me focus on delivering a Minimum Viable Product (MVP) while remaining realistic about time constraints and project scope.

|  |  |
| --- | --- |
| Priority | Requirement |
| Must Have | Fetch device compliance data from Microsoft Intune using the Graph API |
| Must Have | Store the data in a SQL database for later reference and reporting |
| Must Have | Generate summary reports showing compliant vs non-compliant devices |
| Should Have | Filter reports by platform (Windows/iOS) |
| Should Have | Support trend analysis by storing historical data |
| Could Have | Automated email notifications for non-compliant devices |
| Won’t Have | User interface or web dashboard (out of scope) |

### 3.3. Risks and Constraints

During analysis, I also identified potential risks and technical constraints that could affect development or project delivery. These are documented below with mitigation plans.

|  |  |  |
| --- | --- | --- |
| Risk/Constraint | Impact | Mitigation |
| API Authentication | OAuth2 token access may require approval or permissions | Request delegated access and test in a secure environment |
| Rate Limiting | Frequent API calls could be throttled | Batch requests sensibly and cache where possible |
| Data Sensitivity | Compliance data may contain identifiers | Avoid storing PII; use secure storage |
| Time Constraints | Project must be completed in 4 weeks (~30 hours) | Time-boxed features and focused Minimum Viable Product (MVP) delivery |
| No UI Planned | Project is CLI and reporting-based only | Stakeholders aware this is a backend/reporting tool |

### 3.4. Identify System Inputs, Processes, and Outputs

To design the application effectively, I analysed the expected data inputs, the internal processes the system must perform, and the desired outputs for end users. This Input-Process-Output (IPO) model helps ensure the solution is well structured, maintainable, and aligned with user and business needs.

#### Inputs

* Microsoft Intune device compliance data retrieved via the Microsoft Graph API
* Authentication credentials (OAuth2 access token)
* Configuration parameters (e.g., filters, date range, environment variables)
* Historical compliance records (for trend analysis, if available)

#### Processes

* Authenticate with the Microsoft Graph API using secure credentials/ client secret
* Fetch device compliance data and parse the response
* Transform and clean the data using Python/ SQLite3 (e.g., convert timestamps, standardise values)
* Insert the structured data into a SQL database (SQLite)
* Run summary queries to calculate compliance rates, trends, and other metrics
* Optionally trigger automated notifications for non-compliance thresholds

#### Outputs

* Tabular compliance summary reports (e.g., device ID, status, last check-in date)
* Aggregate statistics (e.g., number of non-compliant devices per OS)
* Visual charts (e.g., bar charts showing compliance over time)
* Email alerts or log messages for internal IT reporting
* Persistent record storage for historical analysis and audit

### 3.5. High-Level Architecture

#### System Architecture Diagram

The following diagram shows a high-level overview of the components that make up the solution and how they interact. It includes the Python scripts for data retrieval and processing, Microsoft Graph API as the external data source, the SQLite database for local storage, and the reporting layer used to generate outputs for the IT team. This architecture supports a modular, testable, and scalable design that can evolve over time as our needs grow.

A blue cylinder with white text

AI-generated content may be incorrect.

#### Data Flow Diagram (DFD)

A group of blue rectangular objects with text

AI-generated content may be incorrect.This DFD shows how data moves through the system from external source (Microsoft Graph API) through internal processing, storage, and reporting workflows initiated by the relevant member of IT. The diagram reflects a clean, modular structure to support maintainability and testability.

### 3.6. Database Design

This section outlines the design of the database that will store compliance data retrieved from Microsoft Intune. A clear and logical schema is essential for ensuring that data is stored efficiently, is easy to query, and supports meaningful reporting and analysis. This database design also contributes to the maintainability and scalability of the overall system.

#### Table: device\_compliance

A single table named device\_compliance will be designed to hold the retrieved data from the Graph API. This table includes the most critical fields required for reporting on device compliance and monitoring trends over time.

#### Entity Relationship Diagram (ERD)

I decided that an ERD was not necessary as my design only involves one table and the table structure I have written (with clear fields, types, and descriptions) already meets the expectations for demonstrating structured database design.

### 3.7 Chosen Technologies

This section explains the technologies selected for this project and provides a rationale for each, alongside an evaluation of possible alternatives. The chosen tools enable me to develop a solution that is maintainable, testable, and aligned with both the project's requirements and my current technical skillset at an early stage of my apprenticeship.

#### Chosen Tools and Rationale

|  |  |  |  |
| --- | --- | --- | --- |
| **Technology** | **Reasoning** | **Alternatives** | **Trade-offs** |
| **Python** | Easy to learn, excellent for scripting and APIs, strong library ecosystem | Java, C# | Slower than compiled languages; whitespace-sensitive |
| **SQLite** | Lightweight, no server required, ideal for local data storage; integrates easily with Python | PostgreSQL, MySQL | Not ideal for large-scale or multi-user systems |
| **Microsoft Graph API** | Official method to retrieve Intune data; secure and well-documented | Manual Intune exports | Requires authentication setup and API learning curve |
| **Power BI** | Efficient tools for data manipulation and visualisation | pandas, matplotlib, Excel, seaborn | May require additional learning |
| **Task Scheduler / Cron** | Enables scheduled automation of scripts | Manual script execution | Platform dependent; needs basic config |

While Power Automate could achieve similar results with a no-code/low-code approach, here are some key benefits of using Python and SQL for this project:

* Fine-grained control: Python gives you complete control over how data is fetched, transformed, and stored. You can implement custom transformation logic, robust error handling, and precise data filtering, which might be more difficult to achieve using pre-built connectors in Power Automate.
* Flexibility and customisation: using Python, you can easily adapt and extend your solution. For example, I can integrate machine learning components in the future, complex data processing algorithms, or automated testing routines. This level of customisation is often beyond the scope of a low-code platform.
* Reproducibility and transparency: a Python/SQL solution is fully scriptable, and version controlled. This means that every step of the process is documented in code, making it reproducible, auditable, and easier to maintain over time.

## 4. Development

### 4.1. Project Setup and Environment

To begin development, I ensured my local environment was fully configured for Python development, version control, and structured project management.

* **IDE Used:** I used Visual Studio Code for its Python support, Git integration, and terminal access.
* **Folder Structure:** As defined in Section 1, I created a modular folder structure (Scripts, Docs, Reports, etc.) to separate concerns and support scalability.
* **Version Control:** The project was tracked using Git from the beginning. All changes were committed with meaningful messages, and the repository was hosted privately on GitHub.
* **Python Environment:** I planned to use the system-wide Python 3.9.13 installation and create a virtual environment with built-in modules as well as additional packages.

This setup phase allowed me to begin development with confidence, knowing that the core tooling was in place and aligned with the design decisions laid out in earlier sections.

### 4.2. API Connection to Microsoft Graph

The first development task was establishing a secure connection to the **Microsoft Graph API**, which is the recommended method for retrieving compliance and device management data from Microsoft Intune.

To configure API access logged into Azure, navigated to **App registrations** and selected **New registration** before entering a name and selecting **Register**. I then granted it the required Graph API permissions (**DeviceManagementManagedDevices.Read.All**).

#### Authentication Approach

Microsoft Graph uses OAuth 2.0 for secure access. I selected **Application permissions** as this will allow it to run as a background service without a signed-in user.

A screenshot of a computer

AI-generated content may be incorrect.

#### Client Secret

A client secret was generated during the Microsoft Azure app registration process to authenticate against the Microsoft Graph API. A **client secret** is essentially a password used by your application to authenticate against Microsoft’s identity platform. It's used in combination with your **Client ID** and **Tenant ID** when requesting access tokens.

The secret was set to expire in 12 months, following development best practices. This expiry balances security with convenience and encourages regular credential rotation. To prevent unexpected authentication failures, the expiry date was documented.

### 4.3 Storing Credentials Securely with .env and .gitignore

#### To follow secure coding practices and protect sensitive API credentials, I implemented a .env file for secure local storage, excluded it from source control using .gitignore, and managed Python dependencies within a virtual environment. This ensured that my authentication secrets were never exposed publicly and that my project could be easily reproduced on any machine.

#### Step 1: Creating and Populating .gitignore

To avoid accidentally committing sensitive information, I ensured relevant details were excluded from Git version control:

# Python cache and virtual environment

\_\_pycache\_\_/

\*.pyc

\*.pyo

\*.pyd

.venv/

env/

# Ignore environment variables

.env

# Ignore virtual environment folder

.venv/

# OS-specific

.DS\_Store

Thumbs.db

# VS Code settings

.vscode/

# Ignore device output

cleaned\_devices.json

# Ignore screenshots

Screenshots/

# Ignore all SQLite database files

\*.db

\*.db-journal

\*.sqlite3

This was added to my .gitignore file at the root of the repository.

#### Step 2: Setting up the Virtual Environment

I followed Python best practices by isolating my project dependencies using a virtual environment. This avoids conflicts with global packages and increases reproducibility.

python -m venv .venv

To create and activate the virtual environment:

To activate it in PowerShell:

powershell -ExecutionPolicy Bypass -File .\.venv\Scripts\Activate.ps1

#### Step 3: Install required packages inside the virtual environment

The next step was to install my required packages for the project using pip.

pip install python-dotenv requests sqlite-utils

These packages were chosen because:

* python-dotenv loads environment variables securely from .env.
* requests handles HTTP requests to the Microsoft Graph API.
* sqlite-utils provides extra utilities for working with SQLite databases via Python.

#### Step 4: Freeze Dependencies for Reproducibility

I generated a requirements.txt file so that I can easily recreate the same environment:

pip freeze > requirements.txt

python-dotenv

requests

sqlite-utils

sqlite3

Contents of the file:

#### 

#### Step 5: Creating a .env File to Store API Credentials

CLIENT\_ID=my-client-id-here

TENANT\_ID=my-tenant-id-here

CLIENT\_SECRET\_ID=my-client-secret-id-here

CLIENT\_SECRET\_VALUE=my-client-secret-value-here

This file was used to store key-value pairs for CLIENT\_ID, TENANT\_ID, CLIENT\_SECRET\_ID and CLIENT\_SECRET\_VALUE, which are loaded securely using the python-dotenv package.

### 4.4 Data Retrieval and Authentication

To securely access the Microsoft Graph API and retrieve device compliance data from Intune, I implemented a modular Python-based authentication and data retrieval workflow. This section outlines how I developed, structured, and validated the components responsible for secure API access and data extraction.

#### Step 1: Authentication via auth.py

To access Microsoft Graph securely using the client credentials flow, I created a Python script (auth.py) responsible for authenticating against the Microsoft Identity Platform.   
  
This script performs the following:

* Loads credentials from the .env file using python-dotenv

# auth.py

import os

from dotenv import load\_dotenv

import requests

# Load environment variables from .env file

load\_dotenv()

* Reads TENANT\_ID, CLIENT\_ID, and CLIENT\_SECRET\_VALUE from environment variables

# Get credentials from environment variables

TENANT\_ID = os.getenv("TENANT\_ID")

CLIENT\_ID = os.getenv("CLIENT\_ID")

CLIENT\_SECRET\_VALUE = os.getenv("CLIENT\_SECRET\_VALUE")

* Next, I created a get\_access\_token() function in auth.py which sends a POST request to the Microsoft Identity token endpoint to retrieve a bearer token

# Microsoft Identity token endpoint

TOKEN\_URL = f"https://login.microsoftonline.com/{TENANT\_ID}/oauth2/v2.0/token"

def get\_access\_token():

    """

    Retrieves an OAuth2 access token from Microsoft Identity Platform

    using the client credentials flow.

    """

    headers = {

        "Content-Type": "application/x-www-form-urlencoded"

    }

    body = {

        "client\_id": CLIENT\_ID,

        "scope": "https://graph.microsoft.com/.default",

        "client\_secret": CLIENT\_SECRET\_VALUE,

        "grant\_type": "client\_credentials"

    }

    response = requests.post(TOKEN\_URL, headers=headers, data=body)

* Then I parsed and returned the access\_token for use in subsequent API calls

    if response.status\_code == 200:

        token = response.json().get("access\_token")

        print("Access token retrieved.")

        return token

    else:

        print(" Failed to retrieve access token.")

        print(response.text)

        return None

if \_\_name\_\_ == "\_\_main\_\_":

    token = get\_access\_token()

    print(token)

Step 2: Device Data Retrieval (fetch\_devices.py)Once authenticated, I developed fetch\_devices.py to retrieve data on all registered Intune devices. Here is a breakdown of the script:

* Firstly, I imported and called the get\_access\_token() function from auth.py

# fetch\_devices.py

import requests

import json

from auth import get\_access\_token  # Reuse token function from auth.py

* The raw response from Microsoft Graph contains many fields, so I created transform\_device() to extract only the required attributes. This includes device name, operating system (OS), compliance status, OS version, total and free storage (in bytes), and last check-in date. It also calculated values such as free storage percentage and compliance status.

# Transform and clean the raw device data

def transform\_device(device):

    """Cleans and extracts only the fields needed for reporting."""

    try:

        total = int(device.get("totalStorageSpaceInBytes") or 0)

        free = int(device.get("freeStorageSpaceInBytes") or 0)

        free\_pct = (free / total \* 100) if total else None

    except (ValueError, TypeError, ZeroDivisionError):

        total, free, free\_pct = 0, 0, None

    return {

        "deviceId": device.get("id"),

        "deviceName": device.get("deviceName") or "unknown",

        "emailAddress": device.get("emailAddress") or "unknown",

        "model": device.get("model"),

        "operatingSystem": device.get("operatingSystem"),

        "osVersion": device.get("osVersion"),

        "complianceState": device.get("complianceState"),

        "isCompliant": device.get("complianceState") == "compliant",

        "totalStorageBytes": total,

        "freeStorageBytes": free,

        "freeStoragePct": round(free\_pct, 2) if free\_pct is not None else None,

        "lowStorage": free\_pct is not None and free\_pct < 10,

        "lastSyncDateTime": device.get("lastSyncDateTime")

    }

* After that, I sent a POST request to the Microsoft Graph API's /deviceManagement/managedDevices endpoint and stored devices in a list.

# Step 1: Get access token

access\_token = get\_access\_token()

if access\_token:

    # Step 2: Define the Microsoft Graph API endpoint

    endpoint = "https://graph.microsoft.com/v1.0/deviceManagement/managedDevices"

    headers = {

        "Authorization": f"Bearer {access\_token}",

        "Content-Type": "application/json"

    }

    all\_devices = []  # Store all device records here

* The @odata.nextLink property was also included in the script to paginate responses- this is also discussed in the next step of this section.

    # Step 3: Handle pagination to fetch all devices

    while endpoint:

        response = requests.get(endpoint, headers=headers)

        if response.status\_code == 200:

            data = response.json()

            devices = data.get("value", [])

            all\_devices.extend(devices)

            # Continue if there's another page of results

            endpoint = data.get("@odata.nextLink", None)

        else:

            print("❌ Failed to retrieve device data:", response.status\_code)

            print(response.text)

            break

* Finally, the program outputs the total device count, transforms and cleans the data and saves the cleaned data to cleaned\_devices.JSON.

    # Step 4: Output total device count

    print(f"✅ Retrieved {len(all\_devices)} devices")

    # Step 5: Transform and clean the data

    cleaned\_devices = [transform\_device(device) for device in all\_devices]

    # Step 6: Save cleaned data to JSON

    with open("cleaned\_devices.json", "w") as f:

        json.dump(cleaned\_devices, f, indent=2)

    print("✅ Cleaned device data saved to cleaned\_devices.json")

else:

    print("❌ Authentication failed. Cannot retrieve device data.")

* The program also prints key status messages to confirm successful data acquisition and storage or to confirm it was unable to authenticate.

#### Step 3: Output Verification and Pagination Handling

Upon initial testing, only a subset of devices was retrieved. By inspecting the raw JSON output, I identified the need for pagination handling- a common scenario when querying large datasets from Graph. I updated the script to recursively follow the @odata.nextLink property and retrieve all available pages, ultimately capturing a full dataset of 180 devices.

All device data was successfully exported into cleaned\_devices.json, stored in the root of the project directory, for use in later processing and visualisation stages.

#### Step 4: Reflections and Testing Considerations

Although detailed testing and debugging are discussed in Section 5, I used frequent print() statements and controlled output preview such as:

print(json.dumps(devices[:3]))

During development this helped me to isolate issues such as partial responses, empty tokens, or indentation errors which allowed for rapid identification and resolution of logical flaws.

### 4.5. Storing the Data

With the device compliance data cleaned and transformed in Section 4.4, the next step was to store it in a structured format to enable filtering, reporting, and future data queries. I selected SQLite as the storage solution due to its simplicity, portability, and suitability for a lightweight, read-intensive reporting tool.

#### Step 1: Selecting a Database Solution

SQLite was chosen for the following reasons:

* No server configuration required- ideal for local development or small-scale automation.
* Lightweight and embeddable- integrates directly with Python without needing a running service.
* Supports SQL syntax- allowing the use of powerful queries, indexes, and joins if required in future enhancements.

This aligns with the project's scope and reduces complexity. It also makes it easier to port the logic to a cloud-native database later if scalability is required.

#### Step 2: Designing the Table Schema

I designed a schema to match the fields prepared in cleaned\_devices.json, which were the result of the transformation step. These fields provide a foundation for compliance reporting across both Windows and iOS devices.

The table was named device\_compliance and includes the following fields:

|  |  |  |
| --- | --- | --- |
| **Column** | **Data Type** | **Description** |
| deviceId | TEXT | Unique device identifier (used as primary key) |
| emailAddress | TEXT | User’s email address |
| Model | TEXT | Device model |
| deviceName | TEXT | User-friendly device name |
| operatingSystem | TEXT | OS type (e.g. Windows, iOS) |
| osVersion | TEXT | Operating system version |
| complianceState | TEXT | Raw compliance state string from Intune |
| isCompliant | BOOLEAN | Boolean flag derived from complianceState |
| totalStorageBytes | INTEGER | Total storage reported by the device |
| freeStorageBytes | INTEGER | Free space reported by the device |
| freeStoragePct | REAL | Calculated % of storage space remaining |
| lowStorage | BOOLEAN | True if free space < 10% |
| lastSyncDateTime | TEXT | Last reported sync date from InTune |

The primary key was set to deviceId as each record in Intune is uniquely identified by this field.

#### Step 3: Writing Cleaned Data to SQLite

To produce the transformed device data for reporting, I developed a Python script called insert\_into\_sqlite.py. Th*e* script loads the structured device records from cleaned\_devices.json and writes them into a local SQLite database (intune\_devices.db).

The script performs the following steps:

* Connects to SQLite and creates the database file intune\_devices.db if it doesn't exist.

# insert\_into\_sqlite.py

import json

import sqlite3

# Load the cleaned device data

with open("cleaned\_devices.json", "r") as file:

    devices = json.load(file)

# Connect to the SQLite database (or create it)

conn = sqlite3.connect("intune\_devices.db")

cursor = conn.cursor()

* Defines the table schema (device\_compliance) and creates it if not already present.

# Create the table (if it doesn't already exist)

cursor.execute("""

CREATE TABLE IF NOT EXISTS device\_compliance (

    deviceId TEXT PRIMARY KEY,

    deviceName TEXT,

    emailAddress TEXT,

    model TEXT,

    operatingSystem TEXT,

    osVersion TEXT,

    complianceState TEXT,

    isCompliant BOOLEAN,

    totalStorageBytes INTEGER,

    freeStorageBytes INTEGER,

    freeStoragePct REAL,

    lowStorage BOOLEAN,

    lastSyncDateTime TEXT

);

""")

* Reads the cleaned JSON file and iteratively inserts each device record into the table using INSERT OR REPLACE, ensuring no duplicates on deviceName.

# Insert each device one at a time

for device in devices:

    cursor.execute(f"""

    INSERT OR REPLACE INTO device\_compliance (

        deviceId, deviceName, emailAddress, model, operatingSystem, osVersion,

        complianceState, isCompliant, totalStorageBytes, freeStorageBytes,

        freeStoragePct, lowStorage, lastSyncDateTime

    ) VALUES (

        ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?

    );

    """, (

        device["deviceId"],

        device["deviceName"],

        device["emailAddress"],

        device["model"],

        device["operatingSystem"],

        device["osVersion"],

        device["complianceState"],

        device["isCompliant"],

        device["totalStorageBytes"],

        device["freeStorageBytes"],

        device["freeStoragePct"],

        device["lowStorage"],

        device["lastSyncDateTime"]

    ))

* Commits the changes and closes the database connection once all records are processed, whilst printing a message to confirm the number of records inserted into device\_compliance.

# Save changes and close the connection

conn.commit()

conn.close()

print(f"✅ {len(devices)} records inserted into device\_compliance table.")

I then verified that intune\_devices.db had been created by viewing it with Visual Studio Code’s SQLite Viewer Extension.

A screenshot of a computer

AI-generated content may be incorrect.

Choosing SQLite kept the solution self-contained and aligned with the constraints of a single-user, internal reporting tool. Storing the data prepares the project for integration with visualisation tools or automation layers.

### 4.6. Querying the Data

The goal of this stage was to query and validate device compliance data from the SQLite database, ensuring that the correct insights could be retrieved by IT. These queries support the Business Analyst team, who are responsible for visualising and presenting the data using tools such as Power BI.

I used the SQLite extension within VS Code to write and run test queries against intune\_devices.db.

-- Query to get all data

SELECT

    deviceId AS "Device ID",

    deviceName AS "Device Name",

    emailAddress AS "User",

    operatingSystem AS "Operating System",

    osVersion AS "OS Version",

    complianceState AS "Compliance State",

    isCompliant AS "Is Compliant",

    totalStorageBytes AS "Total Storage (Bytes)",

    freeStorageBytes AS "Free Storage (Bytes)",

    freeStoragePct AS "Free Storage (%)",

    lowStorage AS "Low Storage",

    lastSyncDateTime AS "Last Sync Date"

FROM device\_compliance;

-- Query to get Windows devices with low storage that have checked in within the last 3 months

SELECT

    emailAddress AS "User",

    deviceName AS "Device Name",

    freeStoragePct AS "Free Storage (%)"

FROM device\_compliance

WHERE lowStorage = 1 AND operatingSystem = 'Windows' AND date(lastSyncDateTime) >= date('now', '-3 months');

-- Query to get Windows devices that are not compliant and have checked in within the last 3 months

SELECT

    emailAddress AS "User",

    deviceName AS "Device Name",

    Model AS "Model"

FROM device\_compliance

WHERE isCompliant = 0 AND operatingSystem = 'Windows' AND date(lastSyncDateTime) >= date('now', '-3 months');

-- Query to get iOS devices with low storage that have checked in within the last 3 months

SELECT

    emailAddress AS "User",

    deviceName AS "Device Name",

    freeStoragePct AS "Free Storage (%)"

FROM device\_compliance

WHERE model != "iPhone 13" AND lowStorage = 0 AND operatingSystem = 'iOS' AND date(lastSyncDateTime) >= date('now', '-3 months');

-- Query to get iOS devices that are not compliant and have checked in within the last 3 months

SELECT

    emailAddress AS "User",

    deviceName AS "Device Name",

    Model AS "Model"

FROM device\_compliance

WHERE isCompliant = 0 AND operatingSystem = 'iOS' AND date(lastSyncDateTime) >= date('now', '-3 months');

By executing these SQL queries, I confirmed that the device data was stored accurately, and that the essential compliance information was readily accessible.

While visual reporting was not implemented as part of this project, the queried data is now available for use in stakeholder reporting via CSV export, SQL queries, or Power BI dashboards.

### 4.7. Optional Enhancements and Future Improvements

To extend the capabilities of the InTune Device Compliance Reporter and provide more value to stakeholders, I outlined some enhancements to automate and elevate the reporting process.

#### Weekly Automation of Data Collection

To ensure the device compliance data is refreshed regularly without manual input, I planned to schedule the two core Python scripts- fetch\_devices.py and insert\_into\_sqlite.py- to run automatically every Friday using Windows Task Scheduler.

**Script 1:** fetch\_devices.py

1. Open Task Scheduler and select Create Basic Task.
2. Name the task: Fetch Intune Devices
3. Set the trigger to Weekly, every Friday at 09:00.
4. Set the action to Start a program.
5. Program/script:

Point to the Python executable in the .venv folder:

C:\Path\To\Project\.venv\Scripts\python.exe

1. Add arguments:

-- Query to get Windows devices with low storage that have checked in

**Script 2:** insert\_into\_sqlite.py

Repeat the same steps above to create a second scheduled task:

* Name: Insert Device Data into SQLite
* Trigger: Same day/time, or a few minutes later
* Arguments:

#### Export to CSV for Power BI Reporting

To support integration with tools like Power BI, a SQL query can be run within VS Code or DB Browser for SQLite to export the latest device compliance snapshot:

.mode csv

.headers on

.output device\_compliance.csv

SELECT \* FROM device\_compliance;

.output stdout

This CSV can be used as the data source for a Power BI dashboard, refreshed manually or synced via OneDrive/SharePoint for automated updates.

## 5. Testing and Debugging

### 5.1. Testing Strategy

This project primarily used manual and functional testing, as the scripts were developed for single-run, process automation purposes. However, I recognise that some logic- such as data transformation in fetch\_devices.py- could be refactored into functions suitable for unit testing using pytest. If the project were to scale, I would add unit tests for key transformation logic, API response handling, and edge case validation.

For example, I could add a test like this to validate my data transformation logic:

# test\_transform\_device.py

from Scripts.fetch\_devices import transform\_device

def test\_low\_storage\_flag():

    device = {

        "id": "123",

        "deviceName": "Test Device",

        "emailAddress": "test@example.com",

        "operatingSystem": "Windows",

        "osVersion": "10.0.22631.3447",

        "complianceState": "compliant",

        "totalStorageSpaceInBytes": 100000000,

        "freeStorageSpaceInBytes": 5000000,

        "lastSyncDateTime": "2024-12-31T10:30:00Z"

    }

    result = transform\_device(device)

    assert result["lowStorage"] == True

While unit testing was limited, manual testing effectively validated key data processing tasks.

### 5.2. Functional Testing

auth.py

* Test: Access token successfully retrieved using valid client ID/secret.
* How: Verified 200 OK response and inspected access token output.
* Validation: Output showed first 80 chars of access token.

fetch\_devices.py

* Test: Retrieved correct number of devices.
* How: Confirmed length of list matched expected count (~180 devices).
* Validation: Printed “Retrieved X devices”, saved to cleaned\_devices.json.

insert\_into\_sqlite.py

* Test: Inserted all devices into SQLite with no duplicates or overwrites.
* How: Ran SQL query:

SELECT COUNT(\*) AS total\_devices

FROM device\_compliance;

* Validation: Count matched number of devices retrieved.

### 5.3. Debugging Process

|  |  |  |
| --- | --- | --- |
| **Issue** | **Cause** | **Fix** |
| .venv not activating | VS Code defaulted to old OneDrive path | Reopened VS Code from correct local path and selected local interpreter |
| requests not found | Installed in global env, not .venv | Activated .venv and reinstalled dependencies |
| OperationalError: table already exists | Schema mismatch (columns changed mid-project) | Dropped and recreated SQLite table during development |
| Duplicate devices not being stored | deviceName used as primary key (not unique) | Replaced with deviceId to ensure uniqueness |
| Output from auth.py not showing | Indentation error | Moved if response.status\_code == 200: outside of function block |

## 6. Reflection

### 6.1. What went well

The project met its core objectives by automating the retrieval, transformation, and storage of Intune device data. Breaking down the work into modular, testable scripts (auth.py, fetch\_devices.py, insert\_into\_sqlite.py), I made it easier to troubleshoot and maintain throughout development.

Working with the Microsoft Graph API gave me practical experience with real-world data integration, and I was able to apply structured problem solving to handle challenges such as pagination, authentication, and data inconsistencies. Using SQLite as the storage solution proved to be a good choice- it offered a lightweight yet robust way to store device records for reporting and made querying the data straightforward using standard SQL.

I also became more confident using development tools such as VS Code, virtual environments, Git, and Task Scheduler. These were essential for maintaining code organisation, managing dependencies, and planning for automation.

Tracking the project in Git and pushing it to a private GitHub repository helped me demonstrate version control best practices and clear documentation of my changes.

### 6.2. What I could have done differently

This section reflects on areas of the project that could be improved or approached differently if I were to scope or deliver a similar solution in future.

#### Automatically Pull Latest OS Versions for Comparison

* **Goal:** evaluate device OS version based on live data
* **Method:** Scrape or API-poll those in a future version of the tool:
  + Microsoft releases latest Windows OS builds via public update history:  
    <https://learn.microsoft.com/en-us/windows/release-health/>
  + Apple lists iOS versions here:  
    <https://support.apple.com/en-us/HT201222>

#### Anomaly Detection for Low-Storage Devices

* **Goal:** Detect “outliers” among devices based on free storage or OS version.
* **Method:** A simple k-means clustering or Z-score approach to see which devices differ significantly from the group (e.g., extremely low storage).
* **Libraries:** pandas, numpy

#### Predicting Future Non-Compliance (Logistic Regression)

* **Goal:** Predict if a device might become noncompliant soon, based on patterns in OS version, storage usage, or last check-in time.
* **Method:** A basic logistic regression with historical data for each device. If I keep snapshots over time in my database, I could label examples as “stayed compliant” or “became noncompliant.”
* **Libraries:** pandas, scikit-learn

#### Summarising Compliance Text

* Use Azure OpenAI for:
  + Summaries of device compliance state.
  + Natural language reports (e.g., “We have 18 Windows devices out of compliance. iOS devices show 30% free storage on average.”)

### 6.3. Conclusion

This project has enabled me to apply my learning from A1.3 in a meaningful, real-world context. By developing a Python-based solution to automate the reporting of Intune device data, I’ve demonstrated key software development skills including coding, problem solving, debugging, and working with APIs and databases.

The project also gave me the opportunity to engage with stakeholders, consider future enhancements, and reflect on what I would do differently next time. Thank you for taking the time to read this document.