AI, AUTOMATION AND ANALYTICS

TRANSFORMING DEVICE MANAGEMENT WITH INTUNE

SECOND EDITION

JANNIK REINHARD

MICROSOFT MVP · JANNIKREINHARD.COM

jannikreinhard.com

AI, Automation and Analytics

Transforming Device Management with Intune

Second Edition

Jannik Reinhard

About the Book

Device management has a long history, and it keeps changing. Right now the change is driven by artificial intelligence — but AI is only as useful as the data behind it. Without data, a model has nothing to reason about. In device management, that data lives in Microsoft Intune, in Microsoft Graph, and in the wider Microsoft ecosystem around them.

The purpose of this book is to help you understand what AI actually is, see where it genuinely helps, and — most importantly — get at the data you need to build real analytics and automation on top of Intune. By the end you will have a working understanding of AI, automation, and analytics, and a set of projects you can rebuild in your own tenant.

This is the **second edition**. The first edition came out in early 2025, and in this field two years is a long time. Azure OpenAI became Microsoft Foundry. Copilot moved from preview into the Intune admin center, Windows, Microsoft 365, and the security stack. Security Copilot changed its pricing, its surfaces, and its whole plugin-to-agent model. Device Query grew up. Prices, product names, portals, and screenshots all moved. So every chapter was rechecked line by line, the projects that no longer run were rewritten so they do, and every screenshot was retaken against the current UI. Where the first edition predicted something that has since shipped, I show the real thing; where it aged badly, I say so and fix it.

This second edition is also considerably bigger and deeper than the first. It is organised into four parts that follow the book's title: **Foundations** (device management, Intune, and the licensing and Intune Suite landscape you actually have to navigate in 2026, including what moved into E3/E5); **Analytics** (getting at your data through Microsoft Graph, KQL and Log Analytics, Workbooks and Power BI); **Automation** (building custom solutions, then treating your whole tenant as code with DevOps pipelines, and automating the application, provisioning, compliance, and security lifecycles); and **AI, Copilots, and Building Your Own** (Copilots and LLMs, Security Copilot, grounding and retrieval, a full tour of Microsoft Foundry, Copilot Studio,

and finally the shift to autonomous agents and how to govern them). Where the first edition stopped at an overview, this one goes deep enough that you can actually build.

The book is hands-on. It is full of projects you can replicate, meant as much for inspiration as for copy-paste. It ranges from Intune and Microsoft Graph to Azure, Foundry, and Copilot Studio, so you finish with a broad toolkit rather than one narrow trick.

You do not need deep prior knowledge, though basic familiarity with Intune and Azure helps. Whether you are new to this or have been doing it for years, there should be something here you can use. I hope you enjoy it and come away with ideas you can put to work.

The code

Every code listing in this book — the Graph calls, the KQL, the PowerShell, the Python — is also on GitHub as a runnable file, organized by chapter, so you can clone it and run it against your own tenant instead of retyping from the page:

https://github.com/JayRHa/intune-aaa-book-code

The files are the listings, not a rewrite of them: each one adds the small bit of setup the book explains in prose — the scopes it needs, the endpoint it reads — and is keyless from the start, so there is no secret to paste anywhere. If you find something that has drifted since printing, that repository is where I keep it current.

About the Author

Jannik Reinhard is a Microsoft MVP for Security and Intune and for AI Platform and Foundry. Today he works as an AI Engineer at Epic Fusion, focusing on Microsoft Foundry, AI agents, Microsoft Intune, Azure automation, security, and cloud architecture.

Before Epic Fusion, Jannik worked at BASF — the world's largest chemical company — as a Lead Architect and Technical Lead. There he led technical teams and built AI solutions for a global enterprise of 120,000 users, and served as technical lead of an initiative called AIOps (AI for IT Operations), applying AI to detect and resolve issues in the IT environment before they reached end users. His first major project, years earlier, was moving a large enterprise's iOS fleet to cloud management — his introduction to Intune in its early days — followed by the broader shift of the standard workplace from Configuration Manager to Intune.

He was already working on enterprise AI before the current hype: he started testing GPT-3 right after its release and was one of the early Azure OpenAI customers. Since then he has tested new models and platform features as they shipped — which is also how he learned to check what still works when a demo becomes a production service. That habit is what this second edition is made of.

Jannik holds a stack of Microsoft certifications and is an active Microsoft Certified Trainer. He maintains many public repositories, contributes to open source, blogs at **jannikreinhard.com**,

speaks at conferences around the world, and mentors young professionals. This is his first book; his second, on Microsoft Foundry, followed it. Both share the same idea: explain the topic practically enough that the reader can go and build it.

Copyright

AI, Automation and Analytics: Transforming Device Management with Intune — Second Edition

Published by: Jannik Reinhard, 76855 Annweiler, Germany.

First edition published 28 January 2025. Second edition published 2026.

Product names, screenshots, prices, and preview/GA status reflect the state of the Microsoft cloud as it was verified in mid-2026. Cloud services change continuously; treat any dated fact as dated and recheck the linked source before you rely on it in production.

configuration steps are provided for educational purposes; always test them in a non-production environment before use. In no event shall the author be liable for any loss, damage, data loss, service disruption, or other liability arising from the use of, or reliance on, the contents of this book. Verify current details in the official documentation before making any production decision.

Contents

Part I

Foundations: Device Management, Intune, and the Rise of AI

Chapter 1: The Evolution of Device Management and the Rise of AI

Introduction

When I wrote the first edition of this book, AI in device management was mostly a promise. Copilot in Intune was in preview, Security Copilot had just gone generally available, and most of what I described in this chapter was still "coming soon." Two years later, almost all of it has shipped — and several things I did not see coming shipped too. Azure OpenAI is now Microsoft Foundry. Copilot is in the Intune admin center, in Windows, in Microsoft 365, and in the security stack. Agents that act on your tenant, not just answer questions about it, are real.

So this second edition is not a fresh coat of paint. Every screenshot was retaken against the current UI, every price and product name was rechecked, and the chapters that aged worst — the ones about Azure OpenAI, Security Copilot, and building your own assistant — were largely rewritten. Where a thing I predicted actually arrived, I say so and show it. Where I got it wrong, I say that too.

The thread through all of it is the same as before: **AI is only as good as the data you feed it, and in device management that data lives in Intune, Microsoft Graph, and the wider Microsoft ecosystem.** This book is about getting at that data, turning it into analytics and automation, and then putting AI on top so the routine work runs itself and the interesting work gets your attention. We start where every good story about device management starts — with how we got here — because the shape of today's tools only makes sense once you have seen the problems they were built to solve.

The Early Days: Manual Setup and Configuration

Device management began with a person and a screwdriver. In the early era, IT configured each machine by hand: install the operating system, add the applications, set the preferences for

that specific user, repeat. It worked at ten devices and fell apart at a thousand. Every machine was a little different, which meant every problem was a little different, and consistency — the thing that makes a fleet manageable — simply did not exist.

That inconsistency was not just an efficiency problem, it was a security problem. When each device is configured by hand, each device is a snowflake, and a snowflake fleet has no baseline to enforce or audit against. Gaps opened up that nobody could see, because there was no central place to look. The pressure that produced everything else in this chapter was the pressure to make many machines behave like one managed system.

Active Directory: A Paradigm Shift

The first real answer arrived with Microsoft's Active Directory (AD) in 1999, alongside Windows 2000. AD centralized user and device identity and let administrators push configuration through Group Policy from a single place. Security settings, password rules, and access controls could now be defined once and applied everywhere, and a device could join a domain and inherit its standard configuration instead of being built by hand.

AD was a genuine paradigm shift, and it is still running in most enterprises today. But it was built for a world where devices lived on the corporate network, behind the firewall, reachable over a wire. Group Policy needs line of sight to a domain controller. As soon as people started working from anywhere on devices that might never touch the corporate LAN, that assumption became the constraint — and the search for a management model that did not depend on network location began.

From SMS to MDT: Early Steps Toward Centralized Management

While AD handled identity and policy, a parallel line of tools grew up to handle software and deployment.

- **Microsoft Systems Management Server (SMS):** Released in 1994, SMS 1.0 was Microsoft's first serious enterprise management product. It introduced centralized

software distribution, hardware and software inventory, and patch management — the idea that you could see and change your whole fleet from one console. It was crude by modern standards, but the concepts it established are the ones we still use, and it is the direct ancestor of Configuration Manager.

- **Microsoft Deployment Toolkit (MDT):** First released in 2003 as Business Desktop Deployment, MDT gave teams a repeatable framework for imaging Windows and layering in applications and drivers. It turned deployment from an art into a process, which is exactly the move that lets a small team manage a large fleet without every build being a one-off.

SMS and MDT are the moment the industry accepted a principle that runs through the rest of this book: **you manage scale by standardizing, and you standardize by centralizing.** Everything that followed is a refinement of that idea.

System Center Configuration Manager (SCCM)

SMS matured into System Center Configuration Manager, renamed in 2007 and now branded Microsoft Configuration Manager. For roughly a decade it was the backbone of enterprise Windows management. From one console you could package and deploy software, enforce settings, inventory hardware, and push patches across tens of thousands of machines. If you managed Windows at scale, you managed it with ConfigMgr.

Its strength was also its weight. ConfigMgr is on-premises infrastructure: site servers, distribution points, a SQL database, boundary groups, and the network plumbing to connect them, all of which need care and feeding. It assumes devices can reach that infrastructure, typically over the corporate network or a VPN. And running it well is a specialty in its own right — many organizations had staff whose entire job was keeping ConfigMgr healthy.

None of that made ConfigMgr obsolete; it is still in wide use, and Microsoft kept it relevant by wiring it into the cloud through co-management and tenant attach, which let a device be

managed by ConfigMgr and Intune at the same time. But the shape of the tool — heavy, on-premises, network-bound — was a poor fit for where work was heading.

The Era of Mobility and Remote Work

Work stopped happening in the office. Laptops, tablets, and phones became the primary tools, and people expected them to work the same from home, a hotel, or a customer site as from their desk. The VPN-and-domain-controller model creaked under this: it is slow, it is a single point of failure, and it assumes a connection to corporate infrastructure that increasingly did not exist.

Bring Your Own Device made it harder still. Now IT had to secure and manage machines it did not own and could not image, which meant the old approach of "control the whole device" had to give way to "protect the corporate data on a device you only partly control." The requirement was clear: a management model that worked over the internet, needed no on-premises servers, and could handle both company-owned and personal devices. That model is cloud-native management, and its main vehicle in the Microsoft world is Intune.

COVID-19: The Forcing Function

The pandemic did not create these trends; it compressed them into a few weeks. Overnight, whole workforces went home, often onto personal devices, and the organizations still tethered to on-premises management felt every limitation at once — devices they could not reach, patches they could not push, compliance they could not verify. The ones that had already moved workloads to the cloud adapted in days. The ones that had not spent the next year catching up.

It was an expensive lesson in a simple point: **management that depends on physical proximity is fragile.** After 2020, cloud-native device management stopped being a modernization project you might get to and became the default anyone new was expected to start from.

Cloud-Native Device Management with Intune

Microsoft Intune is the cloud-native answer. It is a service, not a server farm: Microsoft runs the global infrastructure, and you manage devices over the internet from the admin center at **intune.microsoft.com**, wherever those devices happen to be. There is no site server to build, no distribution point to place, no VPN to depend on for management traffic.

Intune does the full range of device management — configuration, application deployment, compliance policy, conditional access, and continuous monitoring — across Windows, macOS, iOS/iPadOS, Android, and Linux, for both corporate and BYOD scenarios. Because it is a cloud service, new capabilities arrive continuously rather than in multi-year release cycles, which is a real advantage and, occasionally, a real annoyance: the UI you screenshot today may look different next quarter. That churn is exactly why a book like this needs a second edition, and why so much of my work in it went into rechecking what is still true.

But the reason Intune anchors this book is not that it manages devices. It is that it produces and exposes an enormous amount of *data* about your fleet — through its reports, through the Intune Data Warehouse and its successors, and above all through Microsoft Graph — and it can *act* on that data through automation. Data plus action is the raw material of everything that follows.

What Actually Arrived: AI in Device Management

The first edition ended this chapter looking ahead at what AI might do. This edition can be more concrete, because the “might” became “does.” Here is what actually shipped, and where the rest of the book picks each one up:

- **Copilot in Intune.** A generative-AI assistant built into the admin center. It answers questions about your tenant in plain language, explains policies and settings, helps with device queries, and summarizes what it finds — grounded in your own tenant data rather than a generic model. We cover it, and the analytics it sits on, in Part II.

- **Security Copilot.** Microsoft's security-focused assistant went GA in April 2024 and has changed substantially since — its pricing model, its embedded experiences across Defender, Entra, and Intune, and its shift from static plugins toward agents. Chapter 13 is largely rewritten to match the current product.
- **Endpoint analytics and anomaly detection.** The proactive-remediation and analytics story matured: Intune surfaces device health, startup performance, application reliability, and anomalies that would have taken a human analyst to spot. Chapter 5 goes deep here.
- **Device Query.** Ask a live question of a device (or, now, across many devices) and get an answer back in seconds using Kusto Query Language. What was a preview curiosity is now a genuinely useful operational tool.
- **Foundry and your own assistants.** Azure OpenAI became Microsoft Foundry, and the pattern of grounding a model on your own data — retrieval-augmented generation — became the standard way to build an internal assistant. The chapter that used to be "Using Azure OpenAI Services" is now a current, keyless, Foundry-based build. Copilot Studio replaced Power Virtual Agents for the low-code path.

The honest summary is that AI has moved device management from *reactive* toward *proactive*: from waiting for a ticket to spotting the pattern that will produce the ticket, and increasingly to fixing it before anyone notices. It has not removed the administrator from the loop, and this book does not pretend it has. What it has done is change what the administrator spends time on — less clicking through blades to gather facts, more deciding what to do with facts that were gathered for you.

The rest of the book is the practical version of that shift. We will get the data out of Intune, turn it into analytics and automation, and then put AI on top — with real projects you can rebuild in your own tenant, updated so they actually run in 2026.

Chapter 2: Understanding Microsoft Intune: The Heart of Modern Device Management

Introduction

Before we get into analytics, automation, and AI, we need a shared picture of what Microsoft Intune actually is and why it sits at the centre of everything else in this book. Intune is Microsoft's cloud service for managing devices, applications, and their security — and for several years running it has been positioned as a leader in Unified Endpoint Management. But "device management tool" undersells it. For our purposes, Intune matters because it is where the *data* about your fleet is produced and where *actions* against your fleet are taken. Analytics needs the first; automation needs the second; AI needs both.

This chapter covers what Intune is, what it does out of the box, what the paid Intune Suite adds on top, and how it connects to the rest of the Microsoft ecosystem. That foundation is what the later, more technical chapters build on.

What is Microsoft Intune?

Intune is a cloud service — Software as a Service — for managing the devices your people use to reach corporate data. You administer it from the **Microsoft Intune admin center** at intune.microsoft.com, and it manages devices over the internet, so there is no on-premises server to build or VPN to depend on for management traffic.

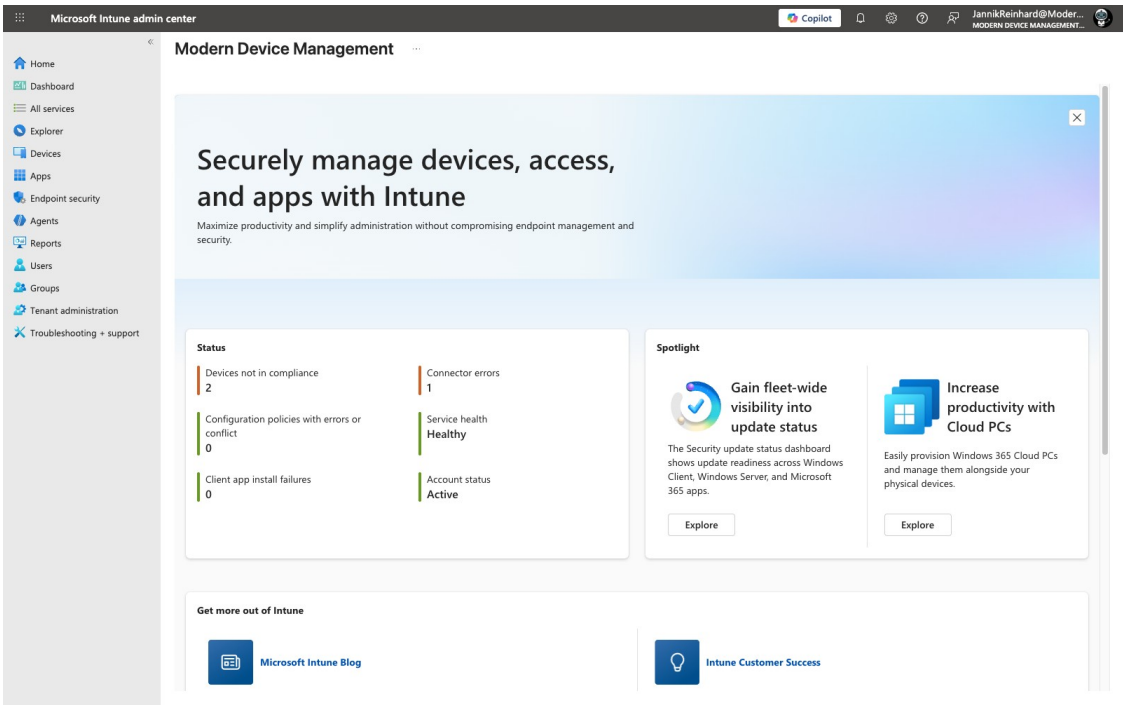

Figure 2.1: The Microsoft Intune admin center home page. The left navigation reflects today's product — note "Agents" and the "Copilot" button in the header, neither of which existed when the first edition was written.

It manages the platforms you would expect — **Windows, macOS, iOS/iPadOS, Android, and Linux** (Ubuntu Desktop) — plus specialised endpoints such as Microsoft Teams Rooms and Surface Hub. Android deserves a note: Microsoft has moved its management surface toward AOSP (Android Open Source Project) management for devices without Google Mobile Services, alongside the established Android Enterprise modes, so the Android story is broader than it was two years ago.

The core job is the same as it always was: configure device settings, enforce security policy, deploy and protect applications, and prove compliance — for both company-owned and BYOD devices, from wherever those devices are. And because Intune is joined at the hip with **Microsoft Entra ID**, it feeds Conditional Access: a device's compliance state becomes a signal that decides whether a user can reach a resource. Compliant device plus healthy sign-in equals access; anything less gets blocked or challenged. That link between "is this device managed and

healthy” and “can this identity get in” is the whole point of the integration, and it is why device management and identity are really one discipline.

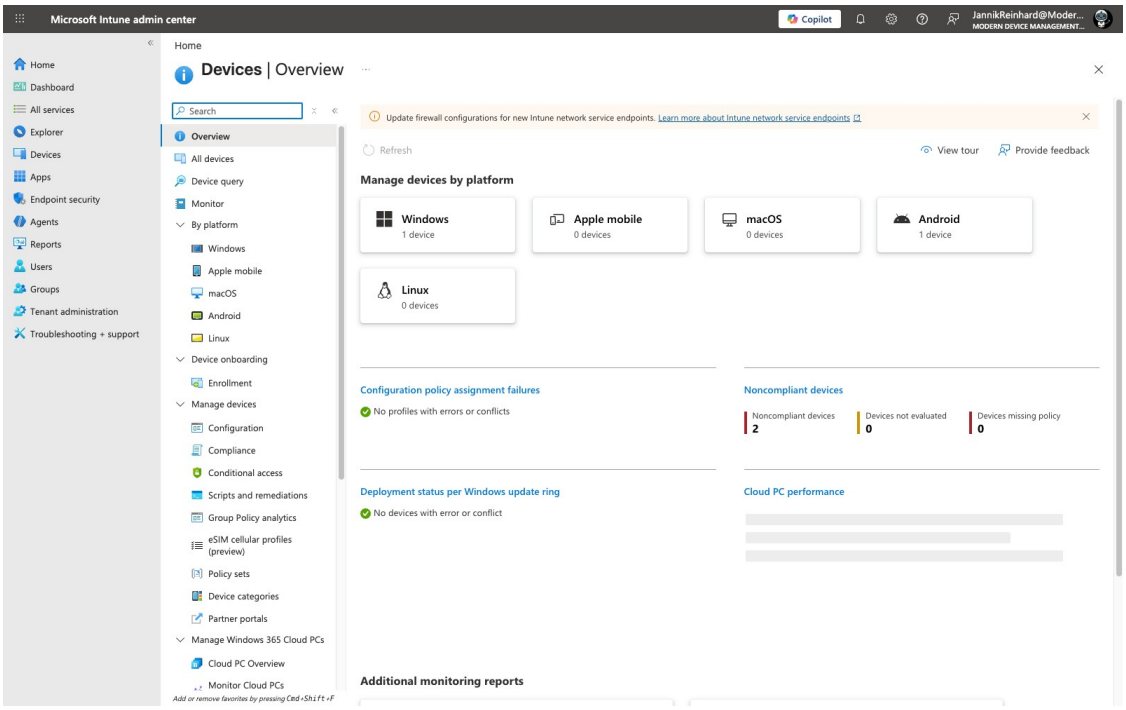

Figure 2.2: Devices > Overview. Devices are grouped by platform, with compliance, configuration, and Cloud PC health surfaced directly. The left menu shows current capabilities such as Device query and Scripts and remediations.

Core Features of Microsoft Intune

At its heart, Intune enforces a baseline. It can require a PIN or password, mandate encryption, block risky settings, and remotely wipe a lost or stolen device — either the whole device or just the corporate data on it. Around that baseline sit the pillars you will use constantly:

- **Configuration.** Settings catalog profiles, compliance policies, and — for Windows — security baselines that ship Microsoft’s recommended hardening as a starting point.
- **Application management.** Deploy, update, and remove apps across platforms, and apply **App Protection Policies** (Mobile Application Management) that control corporate data *inside* apps — blocking copy/paste into personal apps, requiring a PIN to open a

managed app, wiping only the corporate data — without managing the whole device. This is what makes BYOD workable: you protect the data without taking over someone's personal phone.

- **Reporting and monitoring.** Device compliance, app install status, configuration success and failure — the visibility that turns "I think the fleet is fine" into evidence. This reporting layer, and the data behind it, is the subject of Chapter 5.

Intune does not work alone. It sits inside the Microsoft stack and borrows heavily from its neighbours:

- **Microsoft Entra ID** for identity, groups, multi-factor authentication, and Conditional Access — the foundation everything else assumes.
- **Microsoft Configuration Manager**, through co-management and tenant attach, so an organisation can move to the cloud at its own pace and manage ConfigMgr devices from the Intune admin center in the meantime.
- **Microsoft Defender for Endpoint**, which feeds device risk back into compliance, so a machine that Defender flags as at risk can automatically fall out of compliance and lose access until it is remediated.

That combination — identity, management, and threat protection reporting into one place — is what people mean when they call the admin center a single pane of glass. It is not literally one screen, but it is one console and one data model.

The Intune Suite: What You Get When You Pay for More

The Intune most people know is the base plan bundled with Microsoft 365 E3/E5. On top of that sits the **Intune Suite**, a paid add-on (available as the full Suite or as individual add-ons) that has grown considerably and is where a lot of the interesting recent capability lives. As of this writing it includes, among others:

- **Endpoint Privilege Management (EPM)** — let standard users run specific approved actions with elevation, so you can finally take local admin rights away without breaking people's day.
- **Advanced Analytics** — deeper, near-real-time endpoint analytics, including the data behind Device query and richer device timelines. This is directly relevant to Part II of this book.
- **Remote Help** — a first-party, cloud-based remote assistance tool with authentication and compliance warnings built in, now extended to Windows, Android, and macOS.
- **Enterprise Application Management** — a Microsoft-curated app catalog with packaged, auto-updating apps, so you spend less time repackaging installers.
- **Microsoft Cloud PKI** — certificate issuance without running your own PKI infrastructure.
- **Microsoft Intune management of specialty and frontline scenarios**, and additional capabilities Microsoft continues to add.

You do not need the Suite to follow most of this book, and I will call out where a capability requires it. But you should know it exists, because several of the analytics and automation stories are much easier — or only possible — with it.

Copilot in Intune

The single biggest change since the first edition is that **Copilot is now in the admin center**. It is the button in the top-right of every screenshot in this chapter. Copilot in Intune is a generative-AI assistant grounded in *your* tenant data: it can answer questions about policies and settings in plain language, explain what a setting does and what happens if you change it, help you build and interpret **Device query** results, and summarise device and policy state so you are not clicking through six blades to assemble a picture yourself.

The important word is *grounded*. Copilot in Intune is not a generic chatbot bolted onto the portal; it reasons over your devices, your policies, and your configuration. That is only useful

because the underlying data exists and is queryable — which is the same reason this whole book exists. We come back to Copilot in Intune, and the analytics it sits on, in Part II and Part IV.

Beyond Device Management: Intune as a Platform

Everything above is table stakes. The reason Intune anchors a book about AI, automation, and analytics is that it is not a closed appliance — it is a platform you can drive programmatically and wire into the rest of your automation.

Two doors matter most. The first is **Microsoft Graph**, the single API across Microsoft 365 that exposes essentially everything Intune knows and can do — every device, policy, app, assignment, and report is reachable through it, and almost every action you can take in the UI you can take through Graph. Chapters 4 and 5 live in Graph. The second is **automation and orchestration** — Power Automate, Logic Apps, Azure Automation, Azure Functions, and Intune's own Scripts and Remediations — which let you turn "when *this* happens, do *that*" into standing infrastructure instead of a manual chore.

Put those together and Intune stops being a place you visit to click things and becomes a system you build on: it produces the data, exposes it through Graph, and lets automation and AI act on it. That is the shift the rest of the book is about — and every project from here on is a concrete example of it, updated so it actually runs today.

Chapter 3: The Evolution of Microsoft Intune: From Early Beginnings to a Leading Device Management Platform

Chapter 1 told the story of device management as a whole. This chapter zooms in on one product in that story — Intune — because knowing how it grew explains why it looks the way it does today, and why the base plan and the paid Suite are split the way they are. Intune's history is a series of bets Microsoft made on the cloud before most enterprises were ready for it, and the last two years added the biggest bet yet: AI.

Early Beginnings: Windows Intune (2011)

Intune arrived in 2011 as **Windows Intune**. Cloud-based device management was a novel idea, and Configuration Manager owned the enterprise. Windows Intune offered a cloud alternative — manage Windows PCs remotely without building on-premises infrastructure — but its early feature set could not match ConfigMgr, and adoption was thin. Microsoft kept investing anyway, betting that the cloud model would win. It was right; it was just early.

The Rebrand and the Mobile Pivot (2014)

In 2014 Windows Intune became simply **Intune**, and its focus swung toward mobile device management. The timing was deliberate: BYOD was exploding, and organisations suddenly had to manage phones and tablets next to their PCs. Intune's cross-platform reach — Windows, iOS/iPadOS, Android, and (then) Windows Phone and macOS — plus its tie to Azure Active Directory made it a credible answer. Its first appearance on the Gartner Magic Quadrant for enterprise mobility management that year signalled that Microsoft was serious.

Mobile Momentum and the Microsoft 365 Bundle (2017–2018)

The move that changed Intune's trajectory was commercial, not technical: Microsoft bundled it into Microsoft 365. Organisations that already paid for Office 365 and Azure AD found Intune

sitting in their licensing, which lowered the barrier to trying it from “buy a new product” to “turn on something you already have.” At the same time the engineering investment paid off with landmark features — **Windows Autopilot** for zero-touch provisioning of new PCs, and **Windows Hello** for passwordless, biometric sign-in — that moved Intune from challenger to genuine contender.

COVID-19: The Accelerant (2020)

The pandemic did for Intune what it did for cloud management generally — it compressed years of adoption into months. Organisations tethered to on-premises ConfigMgr struggled to manage devices they could no longer physically reach; organisations with Intune already in their M365 licensing switched it on and coped. The surge that followed was less a marketing win than a demonstration: cloud-native management worked when the office did not.

Maturity and Market Leadership (2021–2022)

By 2021–2022 the “Intune versus ConfigMgr” framing was over. Intune had matured into a full unified endpoint management platform, Gartner named Microsoft a UEM leader, and the once-absurd idea of managing a million devices from a single tenant had become real. The remaining challenges were human, not technical — adapting people and processes to a cloud-first way of working — which is a recurring theme in this book.

The Intune Suite (2023)

In March 2023 Microsoft introduced the **Intune Suite**, a paid add-on that layered advanced capabilities on top of the base plan and, deliberately, replaced categories of third-party tooling. The first wave included **Endpoint Privilege Management**, **Remote Help**, **Microsoft Tunnel for Mobile Application Management**, advanced endpoint analytics, and app-management and certificate-management capabilities. The message was clear: Microsoft intended Intune to be not just where you manage devices but where you consolidate the tools around device management.

2024–2026: The AI Era

Everything up to here was in the first edition. What follows is why this edition exists.

- **Copilot in Intune.** Generative AI moved into the admin center — first in preview, then generally available — as an assistant grounded in your own tenant data. It explains settings and policies, helps write and read Device query results, and summarises fleet state. It is the single most visible change to the product since the Suite.
- **The Suite kept growing.** Current members include **Endpoint Privilege Management**, **Advanced Analytics**, **Remote Help**, **Enterprise Application Management** (a curated, auto-updating app catalog), **Microsoft Cloud PKI** (certificate issuance without your own PKI), and Microsoft Tunnel, with more added over time. Several of these — Advanced Analytics above all — are the substrate for the analytics work in Part II.
- **Device Query grew up.** What began as a single-device live-query preview became a genuinely operational tool, including multi-device queries, backed by Kusto Query Language. Chapter 5 uses it in earnest.
- **Android moved toward AOSP.** For devices without Google Mobile Services, Microsoft built AOSP (Android Open Source Project) management alongside the established Android Enterprise modes, broadening what Intune can manage.
- **Security Copilot and agents.** Microsoft's security assistant, generally available since 2024, gained embedded experiences and shifted from static plugins toward **agents** — units of AI that take on a task rather than answer a question. The "Agents" entry now sitting in the Intune navigation is part of the same direction of travel. Chapters 8 and 9 cover this in its current form.
- **Windows 365 and Cloud PCs** became a first-class citizen of the admin center, managed alongside physical devices rather than in a separate world.

Conclusion

Intune's story is one long bet on the cloud, made repeatedly and a little early each time, that kept paying off — from a thin 2011 add-on, through the mobile pivot and the Microsoft 365 bundle, to UEM leadership and the Intune Suite. The current chapter of that story is AI: Copilot in the console, agents in the navigation, and analytics rich enough that a model can reason over your fleet.

That last part is exactly where this book goes next. The rest is not history — it is a set of things you can build today on the platform this chapter has been describing.

Chapter 4: The Intune Suite and Licensing in 2026

Chapter 3 told the story of how Intune grew — from a thin 2011 add-on to a UEM leader, and how in March 2023 Microsoft split the product into a base plan and a paid **Intune Suite** that deliberately replaced categories of third-party tooling. That history explains *why* the product is shaped the way it is. This chapter is about the practical consequence of that shape: before you build a single automation, wire up a single report, or roll out a single advanced feature from the rest of this book, you have to know what you are actually licensed for.

I put this chapter early and I am blunt about it, because licensing is where more good projects die than any technical obstacle. You design a beautiful anomaly-detection workflow, demo it, get sign-off — and then discover the feature it depends on is a paid add-on nobody budgeted for. Or the reverse, which is just as common in 2026: you write a purchase request for a capability your organisation *already owns* and did not realise it had. Both of those are avoidable if you understand the licensing map, and this chapter is that map.

There is a good reason to read it even if you read the first edition: the map was redrawn in 2026, and the redraw is large. As of **1 July 2026**, a chunk of what used to be paid Intune Suite add-ons is now included in Microsoft 365 E3 and E5. I will explain exactly what moved, what did not, and how to decide what — if anything — you still need to buy.

A standing caveat for the whole chapter: Microsoft moves prices, product names, and packaging constantly. Every number and every “included in” claim below was checked in mid-2026, but treat all of them as dated. Where I quote a per-user price, read it as “the list price when I checked — confirm it at purchase,” because your actual price depends on your agreement, your region, and whatever Microsoft did last week.

The Base Licensing Map

Start with the three tiers, because everything else is a variation on them: **Intune Plan 1**, **Intune Plan 2**, and the **Intune Suite**. Understand these three and you can reason about any Intune licensing question you will ever be asked.

Intune Plan 1 — the base everyone actually has

Intune Plan 1 is core Intune. It is what most people mean when they say "we have Intune." It gives you the full unified-endpoint-management platform: cross-platform device enrollment and management (Windows, macOS, iOS/iPadOS, Android, and Linux for a subset of workloads), configuration and compliance policies, app deployment and app protection policies (MAM), Conditional Access integration through Entra ID, the reporting and analytics layer, Endpoint Analytics in its base form, remediation scripts, Autopilot, and Graph API access to all of it. Nearly everything in Chapter 5 — reports, Graph, exports, Log Analytics, base Endpoint Analytics, remediation scripts — runs on Plan 1.

The important thing about Plan 1 is where it comes from. You can buy it standalone (list price around **$8 per user per month** when I checked), but almost nobody does, because it is **bundled into the licences organisations already own**:

- **Microsoft 365 E3 and E5** — the enterprise suites. If you have E3 or E5, you have Intune Plan 1.
- **Microsoft 365 Business Premium** — the small-and-medium-business suite. Includes Plan 1.
- **Microsoft 365 F1 and F3** — the frontline-worker suites. Include Plan 1 (with the frontline scoping that comes with those SKUs).
- **Enterprise Mobility + Security (EMS) E3 and E5** — the identity-and-management bundles. Both include Intune Plan 1.
- **Microsoft 365 Education** SKUs (A3/A5) for schools.

This is the same commercial move Chapter 3 described from 2017–2018: Intune's reach exploded not because of a feature but because Microsoft put it inside licences everyone was already buying. The practical upshot for you is that "do we have Intune?" almost always resolves to "which M365 or EMS SKU are we on?" — and the answer is usually yes.

Intune Plan 2 — the specialty add-on

Intune Plan 2 is an add-on *on top of* Plan 1 (list price around **$4 per user per month**), and it is narrower than its name suggests. It is not "Plan 1 but better" across the board; it is a specific bundle of capabilities aimed at particular scenarios:

- **Microsoft Tunnel for Mobile Application Management (Tunnel for MAM)** — a lightweight VPN that gives *unenrolled* mobile devices access to on-premises resources at the app level, without full device enrollment. (The broader Microsoft Tunnel for enrolled devices is part of Plan 1; the MAM variant is the Plan 2 / Suite piece.)
- **Management of specialty devices** — AR/VR headsets (Meta Quest and similar), Microsoft Teams meeting-room devices, and other purpose-built endpoints.
- **Firmware-over-the-air (FOTA) updates** for supported Android devices, most notably Zebra ruggedised hardware used in logistics, retail, and warehousing.

If you do not run frontline mobile scenarios, kiosk/specialty hardware, or Zebra fleets, Plan 2 historically did very little for you — which is exactly why its 2026 fate (below) matters.

The Intune Suite — Plan 1 plus everything

The **Intune Suite** (list price around **$10 per user per month**) is the top tier: **Plan 1 plus every advanced add-on, bundled**. Rather than buying add-ons individually, you buy the Suite once and get all of them. As of mid-2026 the Suite comprises Plan 1 together with:

- **Endpoint Privilege Management (EPM)**
- **Enterprise Application Management** (the Enterprise App Catalog)

- **Advanced Analytics** (including Device Query)
- **Remote Help**
- **Microsoft Cloud PKI**
- **Microsoft Tunnel for MAM** and the Plan 2 specialty-device capabilities
- and newer members Microsoft has folded in over time

The Suite was always meant as a consolidation play — replace your privilege-management tool, your remote-assistance tool, your certificate infrastructure, and your app-packaging pipeline with capabilities native to Intune, under one line item. The individual add-ons are also sold separately (I list their prices in the decision-framework section), but the Suite is priced so that if you want three or more of them, the bundle usually wins.

That was the stable picture for roughly two years. Then, in 2026, Microsoft changed where the line between "included" and "paid" sits — and that is the single most important thing in this chapter.

The July 2026 Change That Rewrote the Map

On **4 December 2025**, Microsoft announced that a set of advanced Intune capabilities would move *into* Microsoft 365 E3 and E5 at no additional cost. The change became **effective on 1 July 2026**, with automatic provisioning to eligible tenants completing by around **1 August 2026**. No action is required from existing customers — if you are on E3 or E5, the capabilities simply appear in your tenant during that window. If you priced any of these features as a separate purchase in the last year, this is the section that changes your spreadsheet.

Here is exactly what moved. It is more than the headline "Advanced Analytics and Remote Help are now free" that made the rounds — the change is bigger, and the E3-versus-E5 distinction is the part people get wrong.

Microsoft 365 E3 now includes, on top of Intune Plan 1:

- **Remote Help** — attended remote assistance to managed devices.
- **Advanced Analytics** — the richer Endpoint Analytics tier, including **Device Query** (single- and multi-device), anomaly detection, battery and hardware insight, and the richer device timeline.
- **Intune Plan 2** — the entire specialty bundle above: Tunnel for MAM, specialty-device management, and Android FOTA.

Microsoft 365 E5 now includes everything in the E3 list, plus:

- **Endpoint Privilege Management (EPM)**
- **Microsoft Cloud PKI**
- **Enterprise Application Management** (the Enterprise App Catalog)

Put plainly: **as of 1 July 2026, Microsoft 365 E5 effectively includes the full Intune Suite.** An E5 tenant that used to buy the Suite as a separate $10/user/month line item no longer needs to — the equivalent capabilities are in the licence. And **E3 gets a substantial slice** of it: Advanced Analytics, Remote Help, and all of Plan 2, which for many organisations were the reasons they were eyeing the Suite in the first place.

Figure 4.1: The 2026 licensing map — Intune Plan 1 as the base inside M365 E3/E5, the capabilities that moved into E3 and the additional ones that moved into E5 on 1 July 2026, and what still requires a paid add-on for tiers below E5

What is still paid — be precise about this

The change is generous but it is not universal, and the boundaries are where the confusion lives. Three groups still pay:

1. **E3 customers who want the E5-only capabilities.** If you are on E3 and you want **EPM, Cloud PKI, or Enterprise App Management**, those did *not* come to E3 — they came to E5. You either buy them as individual add-ons, buy the Intune Suite, or step up to E5. For an E3 shop, the honest question is now "do we need those three specifically?" rather than "do we need the Suite?"

2. **Everyone below E3.** The inclusions are attached to **Microsoft 365 E3 and E5 specifically.** If your users are on **Business Premium, EMS E3/E5, F1/F3, standalone Intune Plan 1**, or an Education SKU, the July 2026 change does **not** hand you Advanced Analytics or Remote Help for free — those tiers include Intune Plan 1, not the new E3/E5 add-on bundle. For these estates, every advanced capability remains a paid add-on or a Suite purchase, exactly as before. This is the trap: it is easy to hear "Advanced Analytics is included now" and assume it applies to your Business Premium tenant. It does not.

3. **Mixed estates.** Licences are assigned per user. If half your fleet is E5 and half is Business Premium, only the E5 users get the included capabilities; the Business Premium users still need add-ons if you want the features for them. Advanced features generally require the relevant licence on the *users or devices you target with them*, so plan the assignment, not just the purchase.

One more nuance worth stating: **Copilot in Intune did not become free** as part of this. As covered in Chapter 5, Copilot in Intune runs on Security Copilot and consumes Security Compute Units (SCUs) under a separate, capacity-based billing model. The July 2026 change is

about the Intune Suite add-ons folding into E3/E5 — it is not about Copilot. Keep those two mentally separate or you will mis-budget.

So the new one-line summary of the licensing map is: **if you are E5, you have essentially everything; if you are E3, you have most of the analytics and support tooling but not the security/PKI/app-catalog trio; if you are anything below E3, nothing changed and the add-ons are still add-ons.**

Walking the Intune Suite, Component by Component

Knowing which tier a feature lives in is half the job. The other half is knowing what each capability actually *is* and when it earns its place. I will take each Suite component in turn and give it the same treatment: what it does, and when it is worth turning on. Several of these are load-bearing for later chapters, and I will flag those as I go.

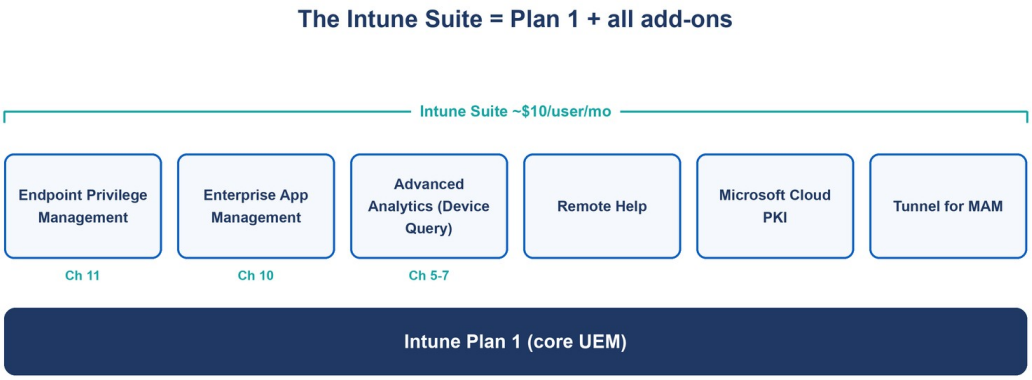

Figure 4.2: The Intune Suite at a glance — Endpoint Privilege Management, Enterprise Application Management, Advanced Analytics, Remote Help, Microsoft Cloud PKI, and Microsoft Tunnel for MAM sitting on top of the Intune Plan 1 base, each labelled with the third-party tooling category it is designed to replace

Endpoint Privilege Management (EPM)

What it is. EPM solves a problem every security team wrestles with: users who need to run *specific* things with administrator rights, on machines where you do not want them to *be* administrators. The old answers were both bad — either you left users as local admins (a standing risk that malware and attackers happily inherit) or you stripped admin rights and then drowned the helpdesk in "I can't install this driver / run this legacy line-of-business app" tickets. EPM gives you a third answer: users run as **standard users**, and specific approved actions are **elevated on demand** according to rules you define.

You author **elevation rules** that target a particular file — by name, publisher/signature, or hash — and decide how it elevates: **automatically** (no prompt, for trusted apps), with **user confirmation** (a justification or a business reason), or via **support-approved** elevation, where a request routes to an admin for approval before it runs. Every elevation is **logged and reported**, so you get an audit trail of exactly what ran elevated, by whom, and when. That reporting is the part that turns EPM from a convenience into a security control — and it is why Copilot in Intune has an "EPM elevation-request analysis" capability (Chapter 5): the elevation data is rich enough for an assistant to reason over.

When it is worth it. The contrast to draw is with *just removing admin rights*. Removing admin rights is free (it is a Plan 1 configuration choice) and you should do it regardless. EPM is worth paying for — or turning on, if you are E5 — when removing admin rights alone creates an unacceptable support burden: fleets with legacy apps that demand elevation, developer or engineering populations, or any environment pursuing a least-privilege / zero-standing-admin posture where you need elevation to be *granular and audited* rather than all-or-nothing. If your users genuinely never need to elevate, you do not need EPM; if they need to elevate occasionally and you want it controlled, EPM is one of the highest-value members of the Suite. In this book, EPM is the substrate for the **security-automation** work later on — the elevation data feeds the analytics and agent scenarios in the security chapters.

Enterprise Application Management

What it is. Packaging and updating third-party Windows apps is one of the most tedious jobs in endpoint management. The traditional loop — download the installer, wrap it into an `.intunewin`, write install/uninstall/detection logic, test it, deploy it, and then do the whole thing again every time the vendor ships an update — consumes real time and never ends. Enterprise Application Management attacks that with the **Enterprise App Catalog**: a Microsoft-curated catalog of common Windows applications, pre-packaged and ready to deploy directly from the Intune admin center. You pick an app from the catalog, assign it, and Intune handles the packaging you used to do by hand.

The feature that makes it more than a convenience is **automatic updates**. Catalog apps can be kept current automatically — when the vendor releases a new version, Intune can roll it out on the schedule and rings you define, so your third-party app estate stops drifting into out-of-date, vulnerable versions. That closes one of the most persistent gaps in real-world patch hygiene: it is rarely Windows itself that is unpatched, it is the sprawl of Chrome, Zoom, Notepad++, 7-Zip, and the rest.

When it is worth it. If you deploy more than a handful of common third-party apps and you feel the repackaging treadmill, this pays for itself in reclaimed admin time and improved security posture almost immediately. If you deploy only a couple of apps, or everything you ship is in-house software the catalog does not carry, the value is thinner. This capability is the foundation of **Chapter 10's app-lifecycle work** — when that chapter talks about managing applications from acquisition through retirement without the manual packaging grind, the Enterprise App Catalog is the engine underneath it.

Advanced Analytics

What it is. Advanced Analytics is the paid tier of Endpoint Analytics, and it is the most important Suite component *for this book specifically*. Base Endpoint Analytics (in Plan 1) gives

you startup performance, application reliability, and work-from-anywhere scoring. Advanced Analytics adds the parts that turn analytics into operations:

- **Device Query** — ask questions of your devices in Kusto Query Language, both **single-device** (a live, real-time query against one machine, back in seconds) and **multi-device** (a fleet-wide query across thousands of endpoints, now cross-platform across Windows, Android, iOS/iPadOS, and macOS). This is one of the most powerful investigative tools in the product, and Chapter 5 uses it in earnest.
- **Anomaly detection** — automatic flagging of device-health regressions, including regressions that follow a change you made, so you can catch a bad policy before the helpdesk does.
- **Battery health** and **resource performance** (CPU/memory/disk pressure) — hardware-level insight for refresh planning and diagnosing slow devices on evidence rather than anecdote.
- **Richer, lower-latency device timelines** and **device scopes** for segmenting analytics by region, department, or team.

When it is worth it. Advanced Analytics is worth it the moment you want to *investigate and act on* fleet state rather than just read dashboards — which, given the subject of this book, is more or less always. It underpins **Chapters 5, 6, and 7**: the analytics work throughout Part II assumes Device Query and the richer telemetry Advanced Analytics provides. The good news for many readers is the July 2026 change — Advanced Analytics is now included in E3 and E5, so if you are on either, this substrate for half the book is already in your licence. If you are below E3, it is a $5/user/month add-on and, for the purposes of this book, close to mandatory.

Remote Help

What it is. Remote Help is Microsoft's native **attended remote-assistance** tool: a helpdesk technician and an end user establish a secure, cloud-brokered remote session — screen sharing, and with consent, full control — directly integrated with Intune. Because it is native, it

respects your Intune **RBAC** roles (a technician sees only what their role allows), it records session activity for auditing, and it uses Entra ID identities on both ends, so you are not standing up a separate remote-support product with its own accounts and its own attack surface. It centres on Windows, with support extending to macOS and to Android for enrollment/support scenarios — check the current platform matrix before you promise a specific OS, as the support surface has been expanding.

When it is worth it. If you run a helpdesk that supports managed devices and you are currently paying for a third-party remote-support tool, Remote Help is the native replacement, and the RBAC/audit integration is a genuine advantage over a bolt-on. This is one of the two capabilities (with Advanced Analytics) that moved into **both E3 and E5** in July 2026 — so for most enterprise-licensed organisations it is now simply available, and the only question is whether to switch off whatever you were paying for before.

Microsoft Cloud PKI

What it is. Certificates are everywhere in modern device management — Wi-Fi and VPN authentication, S/MIME, app authentication — and issuing them has traditionally meant running your own **Public Key Infrastructure**: Active Directory Certificate Services, plus the **NDES** (Network Device Enrollment Service) connector to bridge it to Intune via SCEP. That stack is powerful and painful — servers to patch, a connector that breaks, a certificate authority that is now business-critical infrastructure. **Microsoft Cloud PKI** is a fully cloud-hosted certificate service: Intune stands up the certificate authorities for you and issues, renews, and revokes device and user certificates over SCEP, **without any on-premises PKI or NDES**. You create a root CA and issuing CAs in the admin center, configure SCEP profiles against them, and Intune handles the lifecycle.

When it is worth it. If you are deploying certificates to devices and you either have no on-premises PKI or would love to retire the one you have, Cloud PKI removes an entire class of infrastructure and its maintenance. If you have a mature, well-run internal PKI that already serves the rest of the business, the case is weaker — you may keep using it via the existing

SCEP/NDES or PKCS paths. Cloud PKI is an **E5 / Suite** capability (it did not come to E3), so below E5 it is a paid add-on (around $2/user/month at list).

Microsoft Tunnel and Tunnel for MAM

What it is. Microsoft Tunnel is Intune's VPN gateway, giving managed devices secure access to on-premises resources. The base Tunnel for *enrolled* devices is part of Plan 1. The Suite/Plan 2 piece is **Microsoft Tunnel for Mobile Application Management (Tunnel for MAM)**, which extends that access to **unenrolled** iOS/Android devices at the *application* level — a BYOD phone can reach a specific internal app through the tunnel without the device being fully enrolled and managed. It pairs naturally with app protection policies: the app is protected and can reach what it needs, while the personal device stays personal.

When it is worth it. Tunnel for MAM is a BYOD/frontline play. If you have unenrolled personal or contractor devices that need to reach internal resources through specific apps, it is the tool. If your access model is "enrolled devices only" or "everything is behind a modern identity-based proxy," you may not need it. Note the 2026 wrinkle: Tunnel for MAM travels with **Plan 2**, and Plan 2 is now **included in both E3 and E5** — so this capability quietly became available to a lot of tenants that never bought it.

Newer additions and what to watch

Microsoft treats the Suite as a growing bundle, not a fixed list — Chapter 3 made that point historically, and it still holds. The Enterprise App Catalog's automatic-update capabilities and EPM's policy controls have continued to deepen through 2026, and Microsoft has signalled further endpoint-security and resilience capabilities landing in or alongside the Suite. Rather than commit specific feature names and dates that will age badly in print, I will say this: **check the current Intune Suite composition before you make a purchase decision**, because a capability you were about to buy separately may have joined the Suite — or the licence you already hold — since this was written. The Microsoft Intune pricing page and the "What's new in Intune" documentation are the two sources to confirm against.

A Decision Framework: Plan 1 vs Plan 2 vs Suite

With the components understood, here is how I actually decide — the questions I ask, in order.

Step 1: Establish what you already have. Before buying anything, find out which M365/EMS SKU your users are on, because it determines your starting point and, post–July 2026, may have already given you what you were about to purchase.

- **On E5?** You effectively have the whole Suite already. Do not buy add-ons or the Suite for E5 users — turn the capabilities on and go. Your only spend question is Copilot in Intune (separate, SCU-based) and licences for any *non*-E5 users.
- **On E3?** You have Plan 1, plus (from July 2026) Advanced Analytics, Remote Help, and Plan 2. The only capabilities left to consider buying are **EPM, Cloud PKI, and Enterprise App Management** — the E5-only trio. Decide whether you need those three; if you need all three, compare the cost of three add-ons against the Intune Suite or an E5 step-up for the affected users.
- **On Business Premium, EMS, F-SKUs, or standalone Plan 1?** Nothing was handed to you in July 2026. Every advanced capability is a paid add-on or a Suite purchase. Treat the rest of this framework as fully applicable.

Step 2: Decide Plan 1 vs Plan 2. Plan 2 is worth its ~$4/user/month *only* if you have the specific scenarios it serves — Tunnel for MAM for unenrolled mobile access, specialty devices (AR/VR, Teams Rooms), or Android FOTA/Zebra fleets. If none of those apply, skip Plan 2; it is not a general upgrade. (And if you are E3/E5, you have it anyway.)

Step 3: Decide add-ons vs the full Suite. This is arithmetic. At list prices I saw in mid-2026, the individual add-ons run roughly:

Capability	Approx. list price (per user/month)
Remote Help	$3.50
Endpoint Privilege Management	$3.00
Advanced Analytics	$5.00
Enterprise Application Management	$2.00

Capability	Approx. list price (per user/month)
Microsoft Cloud PKI	$2.00
Microsoft Tunnel for MAM	via Plan 2 (~$4) or Suite
Intune Suite (all of the above)	**~$10.00**

Treat every one of those as "check at purchase" — they are list prices, not your price. But the shape is stable: the Suite is priced so that **if you want three or more add-ons, the bundle usually wins.** Want only Remote Help? Buy Remote Help. Want EPM plus Advanced Analytics plus Cloud PKI? You are already past the Suite's price — buy the Suite. The break-even is around three components.

Step 4: Decide what to buy *first*, if budget is staged. When you cannot buy everything at once, I sequence by breadth of impact and by what unblocks the most work:

1. **Advanced Analytics first** — it touches the most people, powers Device Query and anomaly detection, and (for the purposes of this book) unblocks the analytics chapters. If you are below E3 and can buy only one thing, buy this. *(If you are E3/E5, it is free — skip to the next.)*
2. **EPM** if you are running a least-privilege programme, or **Enterprise App Management** if the app-packaging treadmill is your biggest time sink — pick based on which pain is louder.
3. **Remote Help** if you are still paying a third-party remote-support vendor (this is often a *cost-neutral* switch, so it can jump the queue).
4. **Cloud PKI** when a PKI-retirement or greenfield-certificate project is actually on the roadmap — it delivers little until you have a certificate need to point it at.

The meta-point: **buy against a concrete need, not against feature envy.** Every one of these capabilities is genuinely useful, but a capability you turn on and do not operationalise is just cost. Match the purchase to a project you are actually going to run.

The mistakes I see most often

A few licensing errors come up again and again — in my own work and in the questions I get asked. Knowing them in advance saves you a wasted purchase order or an awkward "why isn't this working" investigation.

- **Assuming the July 2026 inclusions reach every SKU.** They do not. They are attached to Microsoft 365 E3 and E5. Business Premium, EMS, and the frontline F-SKUs include Intune Plan 1 and *nothing new* from this change. I have watched teams celebrate "free Advanced Analytics" on a Business Premium tenant and then discover it was never coming.
- **Confusing "E3 got Advanced Analytics" with "E3 got the Suite."** E3 got the analytics/support slice; the EPM, Cloud PKI, and Enterprise App Management trio is E5-only. If you are on E3 and someone tells you the whole Suite is now free, they are half right, and the wrong half is the security half.
- **Forgetting that features are assigned, not just owned.** Owning the licence in your tenant is necessary but not sufficient — many advanced capabilities require the relevant licence on the *targeted users or devices*, and some require you to explicitly enable the feature (for example, standing up your CAs in Cloud PKI, or assigning the inventory policy that Device Query depends on, covered in Chapter 5). A capability you paid for but never assigned looks, to the admin center, exactly like one you never bought.
- **Double-paying during the transition.** If you were paying for a standalone add-on or the Intune Suite and you are on E3/E5, the July 2026 inclusion may make that spend redundant. Review your add-on line items after your tenant is provisioned (by around 1 August 2026) and stop paying twice for something now in your base licence.
- **Budgeting Copilot in Intune as part of "the Suite."** It is not. It bills separately through Security Copilot SCUs. Put it in its own line of the spreadsheet.

None of these are exotic. They are the everyday friction of a licensing model that changes faster than documentation and org charts do — which is exactly why the answer to almost every Intune licensing question starts with "let me check your current SKU and assignments first."

What the Rest of the Book Leans On

Because this is the foundations part of the book, let me close by connecting the licensing map to the chapters ahead, so you know which of these line items you need in place before each project.

- **Advanced Analytics** is the big one. **Chapters 5, 6, and 7** — the analytics core of the book — assume you can run Device Query and read the richer Endpoint Analytics telemetry. If you are on E3 or E5, you have it as of July 2026; if not, it is the first add-on to buy. Without it, large parts of Part II are read-only.
- **Enterprise Application Management** underpins **Chapter 10's** app-lifecycle work. The curated, auto-updating Enterprise App Catalog is what lets that chapter treat applications as a managed lifecycle rather than a packaging chore. It is an E5/Suite capability.
- **Endpoint Privilege Management** feeds the **security-automation** chapters. Its elevation rules and, crucially, its elevation *reporting* are the data those chapters — and Copilot's EPM analysis — reason over. Also E5/Suite.
- **Plan 1 itself** carries the workhorses of **Chapters 5 and 8**: Graph, report exports, Log Analytics diagnostics, base Endpoint Analytics, and remediation scripts. None of the custom-solution patterns in Chapter 8 require the Suite — a reassuring point, because it means most of the *building* in this book runs on the base licence almost everyone already has.

Conclusion

The licensing map is not the interesting part of device management, but it is the part that determines whether the interesting parts are available to you. In 2026 that map has three tiers — Plan 1, Plan 2, and the Suite — sitting inside the M365 and EMS licences you already own, and it was significantly redrawn on 1 July 2026 when Advanced Analytics, Remote Help, and Plan 2 moved into E3, and EPM, Cloud PKI, and Enterprise App Management moved into E5 on top of them. The practical translation is short: **E5 tenants now have essentially the whole Suite included, E3 tenants have most of the analytics and support tooling, and everyone below E3 still buys add-ons.** Check your SKU before you check your budget — you may own more than you think, or less than a headline implied.

Everything from here is about *using* what you are licensed for. And the very first capability the book leans on — Advanced Analytics, with Device Query and the reporting substrate around it — is exactly where we go next. **Chapter 5, "Intune's Capabilities in Analytics and Automation,"** is the longest and most technical chapter in the book: it opens the hood on reporting, Microsoft Graph, keyless authentication, KQL, Log Analytics, Endpoint Analytics, Device Query, and Copilot in Intune. Now that you know which of those you are entitled to, let us go get the data out.

Part II

Analytics: Getting at Your Data

Chapter 5: Intune's Capabilities in Analytics and Automation

Introduction

This is the chapter the whole book leans on. Everything later — the custom solutions in Chapter 8, the AI assistants further on — assumes you can get data *out* of Intune and act on it. So this chapter is deliberately the longest and most technical one in the book: it is where we open the hood.

It is also the chapter that changed the most between editions, because the plumbing underneath moved. The old V1/V2 split for admin-center reports is gone. The way you authenticate to Microsoft Graph is different — the honest, current answer is "stop using secrets," and I will show you why and how. The Intune Data Warehouse did not die, but it was re-platformed in a way that will quietly break long-running Power BI dashboards if you are not paying attention, and there are hard dates attached that you need to know before June 2026 passes you by. Device Query grew from a single-device party trick into a cross-platform operational tool. Copilot in Intune went generally available and now writes the very Kusto queries this chapter teaches you to write by hand.

I will go through all of it in the order you would actually build on it: first the reports and how to export them, then Graph and modern authentication, then Kusto Query Language (KQL) and Azure Monitor, then Power BI, then Endpoint Analytics and Device Query, then the inventory catalog, and finally Copilot in Intune. Where a thing has a retirement date or a licensing change coming, I flag it inline rather than burying it, because those are exactly the facts that make a 2024 tutorial dangerous to follow in 2026.

The Foundation: Reporting and Analytics

Before you build anything custom, learn what Intune already gives you. It is more than most people use, and a surprising amount of the “I need a custom report” work I get asked for turns out to be a built-in report plus an export the person did not know existed.

At the core of device management is a simple question: what is the state of my fleet right now, and how did it get there? Intune’s reporting answers both — the current snapshot and the trend over time — for compliance, configuration, apps, security, and device health. That reporting layer is not just for humans reading dashboards. It is a data source you can export, automate, and feed into analytics and AI, and that is how we are going to treat it.

Reporting in Intune is **contextual**: reports live in the section of the admin center they belong to. App reports sit under **Apps**, device reports under **Devices**, security reports under **Endpoint security** and **Reports**, and there is a dedicated **Reports** node that aggregates the cross-cutting ones. You do not go to one “Reports” screen for everything; you find the report next to the thing it describes. That is by design, and once you internalize it you navigate faster.

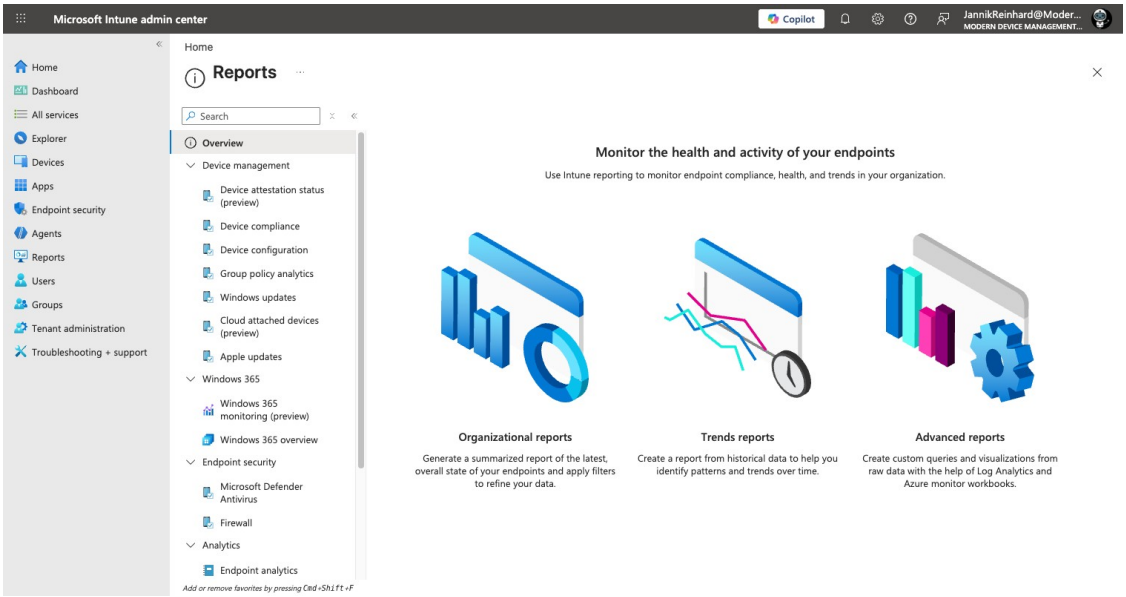

Figure 5.1: The Reports node in the Intune admin center, with the current report categories

Report focus areas

Every Intune report is built for one of four audiences, and Microsoft categorizes them accordingly. Knowing the category tells you what to expect from the data:

- **Operational reports** deliver near-real-time, record-level data for immediate action. This is helpdesk and administrator territory — "which devices failed this policy, and why" — where you need accuracy now, not a trend.
- **Organizational reports** give a high-level, aggregated view for managers and leads: overall compliance posture, deployment health at a glance. Breadth over depth.
- **Historical reports** surface patterns and trends across a timeframe, which is what you use for capacity planning and "is this getting better or worse" questions.
- **Specialist reports** are the raw-data reports. They exist to be exported and processed elsewhere — they are the seam between the admin center and everything custom you build in this book.

That last category is the one this chapter cares about most, because a Specialist report plus the Graph export API is the foundation of automated analytics.

The V1/V2 story is over

If you read the first edition, you will remember a whole section explaining how Intune was migrating its reporting infrastructure from "V1" to "V2." That migration is finished. The admin-center reports now run on a single unified reporting framework, and the split is no longer something you need to think or care about. Search and sort across all columns, data paging through very large datasets, tenant-scale performance, and consistent export — the things that used to be "V2 features" — are just how reports work now.

The one place the old naming survives is the **Power BI connector**, where "v1" versus "v2" still means something concrete. We will get to that when we cover Power BI, because it comes with a retirement date attached.

To read reports you need an appropriate role — **Intune Administrator**, a custom Intune role with the relevant read permissions, or a broader role that contains them. The days of needing Global Administrator for routine reporting are behind us, and you should not be using Global Administrator for this; scope the least privilege that works.

Reporting in Practice: Two Examples

Discovered apps — and its successor, App Inventory

The **Discovered apps** report (Apps > Monitor > Discovered apps) lists the applications Intune has found installed across your managed devices, with device counts, publisher, version, and platform. It is one of the most-used reports in the product, and it is quietly being replaced — so this is a good example of "current" mattering.

Starting to roll out in early May 2026, Microsoft introduced **App Inventory** (also referred to as Enhanced App Inventory) as the successor to Discovered apps for Windows 10/11 corporate, Microsoft Entra-joined devices. The difference is not cosmetic:

- Discovered apps collects roughly once a week. App Inventory collects **multiple times per day**, so the data is far fresher.
- App Inventory carries **richer metadata** per application.
- Critically, **App Inventory requires a device configuration policy to turn it on** — specifically a Properties Catalog inventory policy (covered later in this chapter). Discovered apps needed no such policy. If you assign nothing, App Inventory collects nothing.

For a transition period both run in parallel, but App Inventory is the direction of travel and Discovered apps is the thing being superseded. If you are automating against app inventory data today, plan for the move: the report name and the collection model both change.

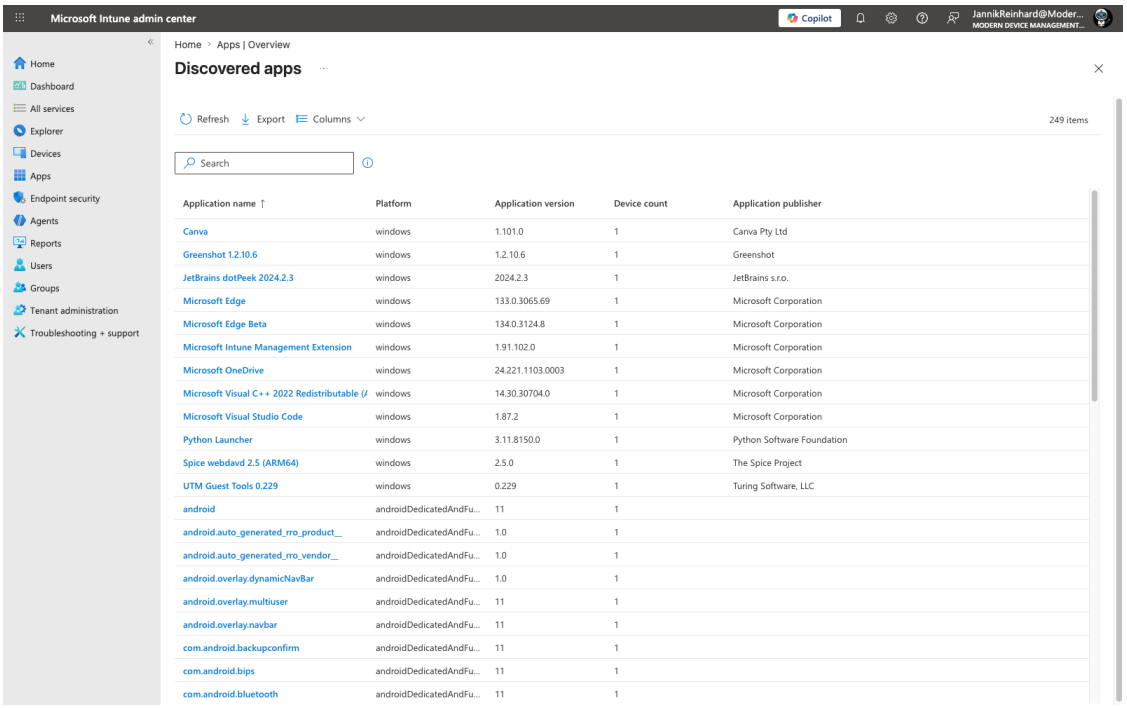

Figure 5.2: The Discovered apps report — every app Intune has found across managed devices, with publisher, version, and device count, plus the Export button that feeds the automation later in this chapter

Assignment status

The **assignment status** reports track deployment results for policies and apps — which assignments succeeded, which failed, and the error behind each failure. Where Discovered apps is a snapshot of *state*, assignment status is a view of *outcome*, and it is the report you open when a rollout is not landing. You can add and remove columns to focus on the data points that matter for the task, and the view shows the full result set rather than making you page blindly.

Both of these reports share a trait that matters for the rest of this chapter: the data you see in the UI is the data you can pull through Microsoft Graph. The portal is not doing anything you cannot do yourself programmatically — which is exactly what we turn to next.

Microsoft Graph: The Backbone of Data Access

To get real leverage out of Intune you have to work with **Microsoft Graph**. Almost everything in this book eventually routes through it.

What Graph actually is

Microsoft Graph is the single, unified API across Microsoft 365 — Intune, Entra ID, Exchange, Windows, security, the lot. Here is the mental model that makes everything click: **the Intune admin center is a website built on top of Graph.** When you assign a policy or open a report in the portal, the page is making Graph API calls in the background and rendering the result. There is no secret second API the UI uses that you cannot reach. Anything the portal does, you can do — from PowerShell, from Python, from a Logic App, from an Azure Function.

That is not a trivia point. It is the reason you can automate Intune at all, and it is the reason a technique I will show you in a moment — reading the portal's own network calls — lets you discover the exact API behind any screen.

A Graph request is structured like this:

```
https://graph.microsoft.com/{version}/{resource}?{query-parameters}
```

- **Base URL**: `https://graph.microsoft.com/`
- **Version**: `v1.0` for production endpoints, `beta` for preview functionality.
- **Resource**: the entity you are addressing, e.g. `users`, `deviceManagement/managedDevices`.
- **Query parameters**: OData options such as `$filter`, `$select`, `$top`, `$orderby`.

A few concrete GETs:

```
GET https://graph.microsoft.com/v1.0/me
GET https://graph.microsoft.com/v1.0/users?$top=10
GET https://graph.microsoft.com/v1.0/deviceManagement/managedDevices?
$select=deviceName,complianceState,operatingSystem
```

The standard REST verbs all apply: `GET` reads, `POST` creates, `PATCH` partially updates, `PUT` replaces, `DELETE` removes.

A word on v1.0 versus beta for Intune. As a rule, prefer `v1.0` — it is production-supported and stable. But a real amount of newer `deviceManagement` functionality lands in `beta` first and only in `beta` for a while, and there is no master list telling you which is which; you check per endpoint. Device inventory, some of the newest reports, and preview features often live under `beta`. My practical rule: use `v1.0` wherever it works, drop to `beta` only where you must, and never ship production automation on a `beta` endpoint without knowing it can change under you. Watch the changelog at `developer.microsoft.com/graph/changelog`.

Authentication the modern, keyless way

This is the part that most needs rewriting from the first edition, so read it even if you think you know it.

The old chapter enumerated a long list of OAuth flows — authorization code, client credentials, device code, on-behalf-of, implicit, ROPC — and it is still true that those flows exist. But the honest 2026 guidance is much shorter, and it is about *credentials*, not flows:

Stop using client secrets. Prefer no stored credential at all.

For unattended automation — the scheduled export job, the Function App, the Logic App, the runbook — the best practice is now:

- **If your code runs in Azure**, use a **managed identity**. Azure holds the identity, rotates it, and hands your code tokens with no secret in your source, your key vault, or your pipeline. You assign the managed identity the Graph application permissions it needs (for Intune, things like `DeviceManagementManagedDevices.Read.All`,

`DeviceManagementConfiguration.Read.All`), and that is the entire credential story.

- **If your code runs outside Azure** — GitHub Actions, another cloud, on-prem — use **workload identity federation**. Your external workload presents a token it already has (its OIDC token) and exchanges it for a Graph token. No secret is stored anywhere. You can even trust an Azure managed identity as the federated-credential issuer, chaining the two together.

Where you genuinely cannot avoid an app-registration credential, use a **certificate**, not a client secret. Secrets leak, get committed, and expire silently; certificates are at least stored and handled as certificates.

Under the hood, all of this is **MSAL**, the Microsoft Authentication Library. MSAL replaced **ADAL**, which reached end of support on **30 June 2023** — if you still have ADAL anywhere, it is not just old, it is unsupported. MSAL manages token acquisition, caching, and refresh across .NET, Python, JavaScript, and PowerShell so you rarely touch tokens directly.

On the PowerShell side, the modernization is just as sharp:

- The **Microsoft.Graph PowerShell SDK** is the module to use. `Connect-MgGraph`, `Invoke-MgGraphRequest`, and the generated `*-Mg*` cmdlets cover Intune end to end.
- The old **MSOnline** and **AzureAD** modules were deprecated on **30 March 2024** and **retired in 2025**. Scripts built on them are living on borrowed time; migrate them.
- Their replacement for identity work is the **Microsoft.Entra** PowerShell module, which went generally available on **29 January 2025**. For Intune-specific work you will mostly stay in Microsoft.Graph; for Entra ID objects, Microsoft.Entra is the current module.

Here is what modern connection looks like in practice. Interactive, for exploration:

```
# Interactive sign-in with least-privilege scopes
Connect-MgGraph -Scopes "DeviceManagementManagedDevices.Read.All", `
                        "DeviceManagementConfiguration.Read.All"
```

Unattended, running inside Azure with a managed identity — note there is no secret anywhere:

```
# Runs in an Azure Automation runbook, Function App, or VM with a managed identity
Connect-MgGraph -Identity
```

That single `-Identity` switch is the whole point of this section. If your automation still passes a client ID and secret into `Connect-MgGraph`, you are carrying a liability you no longer need to carry.

Finding the right Graph call

You rarely need to guess which endpoint sits behind a report. Two reliable ways to discover it:

1. **Browser developer tools.** Open the report in the admin center, press **F12**, go to the **Network** tab, and refresh. The request the portal makes — URL, method, and body — is right there. For reports this is almost always a `POST` to a `deviceManagement/reports/...` endpoint with a JSON body describing the columns and filter.
2. **Graph X-Ray.** A browser extension that captures the Graph calls the portal makes and generates ready-to-run code (PowerShell, and others). Refresh the report with it running and it hands you the call and a snippet.

Both work because of the earlier point: the portal is a Graph client, so watching the portal *is* reading the API documentation for whatever screen you are on.

Exporting Reports with the Graph Export API

For anything larger than a screenful — every discovered app across thousands of devices, a full compliance export — you do not page through results by hand. Intune exposes an

asynchronous export API that does the paging for you and hands back a downloadable file. The pattern is: start a job, poll it, download the result.

This endpoint is now available on **`v1.0`**, so you no longer have to reach for `beta` just to export:

```
POST https://graph.microsoft.com/v1.0/deviceManagement/reports/exportJobs
```

Step 1 — start the job

POST to the endpoint with a body describing what you want. The fields:

- `reportName` (**required**) — the report to export, e.g. `Devices`, `DeviceCompliance`, `DevicesByAppInv`.
- `filter` — an OData-style filter string, optional.
- `select` — the columns to include, optional.
- `format` — `csv` (default) or `json`.
- `localizationType` — how localized values are handled; `LocalizedValuesAsAdditionalColumn` keeps both the raw and friendly values.

Two examples. All discovered apps:

```
{
  "reportName": "DevicesByAppInv",
  "localizationType": "LocalizedValuesAsAdditionalColumn"
}
```

Corporate-owned devices, only the device name column:

```
{
  "reportName": "Devices",
  "filter": "(OwnerType eq '1')",
  "select": ["DeviceName"],
  "localizationType": "LocalizedValuesAsAdditionalColumn"
}
```

The response contains a job **`id`** (something like `Devices_a1b2c3d4-...`) and an initial `status` of `inProgress`.

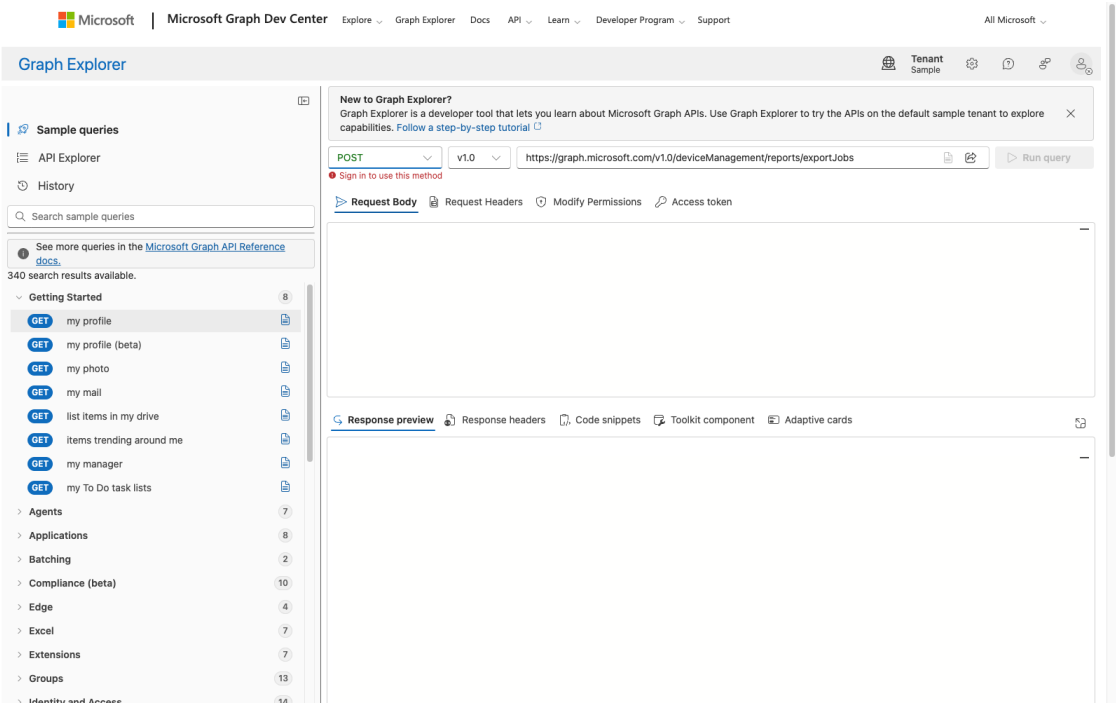

Figure 5.3: Microsoft Graph Explorer set to POST the deviceManagement/reports/exportJobs endpoint — the same call your automation makes to export a report, tried interactively first

There is no single canonical list of every `reportName` value — Microsoft maintains the reference under "Export Intune reports using Graph APIs" — but these are the ones you will reach for most:

reportName	Report in the Intune admin center
Devices	Devices > All devices
DevicesWithInventory	Devices > All devices > Export (with inventory columns)
DeviceCompliance	Device compliance (org)
DeviceNonCompliance	Non-compliant devices
AllAppsList	Apps > All apps
AppInstallStatusAggregate	Apps > Monitor > App install status
DeviceInstallStatusByApp	Apps > All apps > *(select an app)*
DevicesByAppInv	Apps > Monitor > Discovered apps > *(select an app)*
AppInvAggregate / AppInvRawData	Apps > Monitor > Discovered apps > Export
MAMAppProtectionStatus	Apps > Monitor > App protection status
DefenderAgents / Malware	Reports > Microsoft Defender

reportName	Report in the Intune admin center
FeatureUpdateDeviceState	Reports > Windows updates > Feature update report
DeviceRunStatesByProactiveRemediation	Reports > Endpoint analytics > *(a remediation)* > Device status

Step 2 — poll for completion

GET the job by id until `status` reads `completed`:

```
GET https://graph.microsoft.com/v1.0/deviceManagement/reports/exportJobs('{jobId}')
```

While it runs, `status` is `inProgress`. When it finishes, the response includes a **short-lived `url`** pointing to the finished file.

Step 3 — download

Fetch the `url`. The file comes back as a **zipped CSV** (or JSON); unzip it and you have your data.

Automating it in PowerShell

Here is the whole loop with the Microsoft.Graph SDK. Note the `Start-Sleep` in the poll — hammering the endpoint in a tight loop is how you get throttled:

```
# Assumes you have already run Connect-MgGraph (interactively or with -Identity)

$reportName = 'DevicesByAppInv'
$body = @{
   reportName       = $reportName
   localizationType = 'LocalizedValuesAsAdditionalColumn'
} | ConvertTo-Json

# 1. Start the export job
$job = Invoke-MgGraphRequest -Method POST `
   -Uri 'https://graph.microsoft.com/v1.0/deviceManagement/reports/exportJobs' `
   -Body $body

# 2. Poll until completed
do {
   Start-Sleep -Seconds 5
   $job = Invoke-MgGraphRequest -Method GET `
      -Uri "https://graph.microsoft.com/v1.0/deviceManagement/reports/exportJobs('$($job.id)')"
   Write-Host "Status: $($job.status)"
} while ($job.status -ne 'completed')

# 3. Download and unpack
Invoke-WebRequest -Uri $job.url -OutFile './intuneExport.zip'
Expand-Archive './intuneExport.zip' -DestinationPath './intuneExport' -Force
```

Drop that into an Azure Automation runbook or a Function App with a managed identity and you have a scheduled, credential-free export you can point at a storage account, a Power BI dataset, or a data lake.

Respect the throttle limits. The export API is throttled at roughly **100 requests per tenant per minute, 8 per user per minute, and 48 per app per minute.** If you are exporting many reports on a schedule, stagger them and keep the poll interval sane. Throttling shows up as HTTP 429 with a `Retry-After` header — honor it rather than retrying blindly.

Batching many calls into one request

When you need several *small* pieces of data at once — the kind of tenant-summary dashboard the Intune home page builds — do not fire ten sequential requests. Use the Graph **`$batch`** endpoint to send them as one:

```
POST https://graph.microsoft.com/v1.0/$batch
```

The body is an array of sub-requests, each with an `id`, `method`, `url`, and optional `body` and `headers`:

```
{
  "requests": [
    {
      "id": "complianceSummary",
      "method": "GET",
      "url": "/deviceManagement/deviceCompliancePolicyDeviceStateSummary"
    },
    {
      "id": "subscriptionState",
      "method": "GET",
      "url": "/deviceManagement/subscriptionState"
    },
    {
      "id": "failedApps",
      "method": "POST",
      "url": "/deviceManagement/reports/getFailedMobileAppsSummaryReport",
      "body": { "filter": "" },
      "headers": { "Content-Type": "application/json" }
    }
  ]
}
```

Graph runs them together and returns an array of responses keyed by your `id` values. One round trip instead of many — noticeably faster, and easier on the throttle budget. You can express `id`-based `dependsOn` ordering when one call must run after another, though for read-only summaries you usually want them parallel.

Azure Monitor and Log Analytics

Report exports are snapshots. When you want *continuous* log data you can query, trend, and alert on, you route Intune's logs into **Azure Monitor** and store them in a **Log Analytics workspace**, then query with KQL. This is the difference between "export the compliance report each morning" and "keep a rolling history of every compliance and audit event and ask questions of it whenever you like."

What you can send

Under **Reports > Diagnostics settings** in the Intune admin center, you configure which log categories to forward. The categories, and the Log Analytics tables they populate:

- **AuditLogs** → `IntuneAuditLogs` — every change made in Intune: create, update, delete, assign, remote actions. Who did what, and when.
- **OperationalLogs** → `IntuneOperationalLogs` — enrollment successes and failures, and compliance detail.
- **DeviceComplianceOrg** → `IntuneDeviceComplianceOrg` — the organizational device-compliance dataset, including noncompliant devices.
- **IntuneDevices** → `IntuneDevices` — inventory and status for enrolled, managed devices.
- **Windows365AuditLogs** → for Cloud PC actions — create, update, delete, assign, remote actions on Cloud PCs. Cloud PC auditing is on by default, and this category lets you retain and query it alongside everything else.

Setting it up

1. **Create (or reuse) a Log Analytics workspace** in the Azure portal.
2. In the Intune admin center, go to **Reports > Diagnostics settings** and choose **+ Add diagnostic setting**.
3. Select the **categories** above, choose **Send to Log Analytics workspace** as the destination, pick your workspace, name the setting, and save.

You need **Intune Administrator** on the Intune side and **Log Analytics Contributor** on the workspace to wire this up.

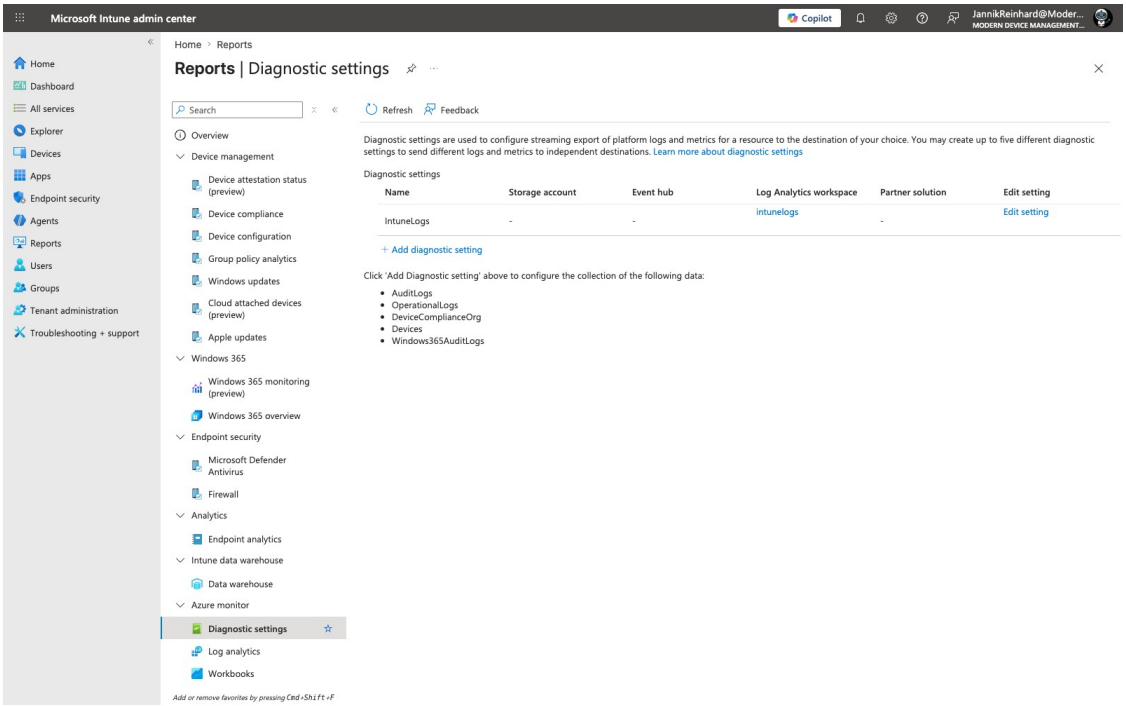

Figure 5.4: Reports > Diagnostics settings. A diagnostic setting streams the Intune log categories — AuditLogs, OperationalLogs, DeviceComplianceOrg, IntuneDevices — to Log Analytics, a storage account, or an event hub

Two operational realities to plan around, because they trip people up:

- **Latency differs by category.** AuditLogs and OperationalLogs arrive essentially immediately. **DeviceComplianceOrg and IntuneDevices can take up to 48 hours** to

appear. If you build an alert expecting compliance data within minutes, it will look broken when it is merely slow.

- **These datasets can duplicate.** In a rolling 24-hour window you may see up to 100% duplication of records for the compliance/device datasets. **De-duplicate downstream** in your KQL (for example, take the latest record per device by `TimeGenerated`) rather than trusting raw counts.

You can also send the same logs to an **Azure Storage account** (cheap long-term archive), an **Event Hub** (stream to a SIEM or custom pipeline), or a **partner/SIEM solution**. A common pattern is Log Analytics for interactive querying plus Storage for cheap retention.

A Practical KQL Tutorial

Kusto Query Language is the language you use against Log Analytics, and — as we will see — against Device Query and Copilot in Intune too. Learn it once, use it in three places. It is read-only, pipe-based, and reads a lot like PowerShell: you start with a table and flow it through a series of operators separated by the pipe (`|`), each transforming the output of the last.

I have made every column name below consistent with the actual Intune tables, because the biggest source of "why is my query empty" is a column name that does not exist.

The basics

Start from a table. This alone returns everything (Log Analytics caps the rows it renders):

```
IntuneDeviceComplianceOrg
```

Filter with `where`:

```
IntuneDeviceComplianceOrg
| where ComplianceState == "NonCompliant"
```

Choose columns with `project`:

```
IntuneDeviceComplianceOrg
| where ComplianceState == "NonCompliant"
| project DeviceName, ComplianceState, OS, TimeGenerated
```

Sort with `order by` (aliased `sort by`), newest first:

```
IntuneDeviceComplianceOrg
| order by TimeGenerated desc
```

Limit with `top` (which sorts and cuts) or `take` (a fast, unordered sample):

```
IntuneDeviceComplianceOrg
| top 10 by TimeGenerated desc
```

Aggregating with summarize

`summarize` is where KQL earns its keep. Count noncompliant devices:

```
IntuneDeviceComplianceOrg
| where ComplianceState == "NonCompliant"
| summarize NonCompliantCount = count()
```

Group that count by operating system:

```
IntuneDeviceComplianceOrg
| where ComplianceState == "NonCompliant"
| summarize NonCompliantCount = count() by OS
```

Because these datasets can contain duplicates and historical rows, a realistic query takes the **latest state per device** before counting — this is the de-duplication I warned about:

```
IntuneDeviceComplianceOrg
| summarize arg_max(TimeGenerated, ComplianceState, OS) by DeviceName
| summarize DeviceCount = count() by ComplianceState, OS
```

`arg_max(TimeGenerated, ...)` keeps the most recent record for each device and drops the stale ones — the single most useful idiom for Intune compliance data.

Time series and trends

`bin()` buckets timestamps so you can trend over time. Noncompliant devices per day over the last 30 days:

```
IntuneDeviceComplianceOrg
| where TimeGenerated >= ago(30d)
| where ComplianceState == "NonCompliant"
| summarize NonCompliantCount = count() by bin(TimeGenerated, 1d)
| render timechart
```

`render` draws it directly in Log Analytics — `timechart`, `barchart`, `piechart`, `columnchart`. Compliance distribution as a pie:

```
IntuneDeviceComplianceOrg
| summarize arg_max(TimeGenerated, ComplianceState) by DeviceName
| summarize Count = count() by ComplianceState
| render piechart
```

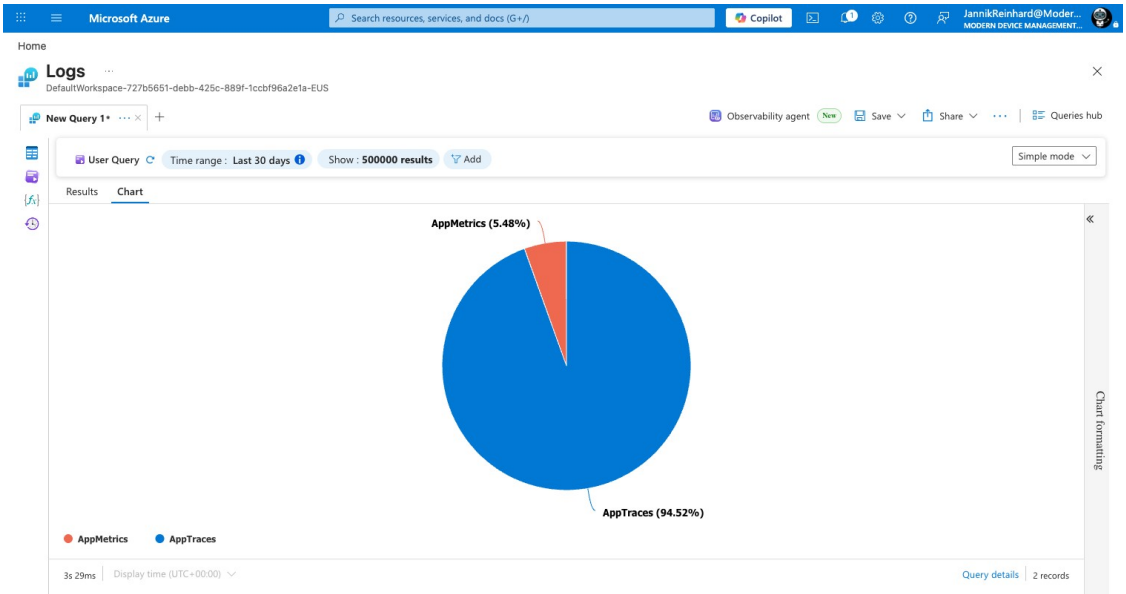

Figure 5.5: The Log Analytics query editor turning a KQL query into a chart with the render operator — the same operators (piechart, timechart, barchart) the compliance queries in this section use; results come back as a grid on the Results tab and as this chart on the Chart tab

Joining tables

Combine datasets on a shared key with `join`. Enrich compliance records with device details from `IntuneDevices`:

```
IntuneDeviceComplianceOrg
| where ComplianceState == "NonCompliant"
| join kind=inner (
    IntuneDevices
    | project DeviceId, DeviceName, OS
) on DeviceId
```

`kind=inner` keeps only rows present in both; `kind=leftouter` keeps every compliance row and fills device details where they exist.

Reusable expressions

`let` names a subquery or value so you can reuse it and keep queries readable:

```
let RecentNonCompliant =
    IntuneDeviceComplianceOrg
    | where TimeGenerated >= ago(7d)
    | where ComplianceState == "NonCompliant";
RecentNonCompliant
| summarize Count = count() by OS
```

`union` stacks tables; a `let`-defined function encapsulates logic you call repeatedly. These are the building blocks — you do not need more than this to write genuinely useful Intune queries.

Putting it together

A single query that filters the last seven days, keeps the latest state per device, joins for OS, and charts noncompliant devices by platform:

```
let ComplianceData =
    IntuneDeviceComplianceOrg
    | where TimeGenerated >= ago(7d)
    | summarize arg_max(TimeGenerated, ComplianceState, DeviceId) by DeviceName
    | where ComplianceState == "NonCompliant";
ComplianceData
| join kind=inner (
    IntuneDevices
    | summarize arg_max(TimeGenerated, OS) by DeviceId
    | project DeviceId, OS
) on DeviceId
| summarize NonCompliantCount = count() by OS
| render barchart
```

A few quick exercises to cement it — try them before reading the solutions:

Exercise 1 — audit trail. Return the 50 most recent admin actions.

```
IntuneAuditLogs
| top 50 by TimeGenerated desc
```

Exercise 2 — devices by OS. Count managed devices by operating system, latest state only.

```
IntuneDevices
| summarize arg_max(TimeGenerated, OS) by DeviceId
| summarize DeviceCount = count() by OS
```

Exercise 3 — repeat offenders. Devices that went noncompliant more than once in 30 days.

```
IntuneDeviceComplianceOrg
| where TimeGenerated >= ago(30d) and ComplianceState == "NonCompliant"
| summarize Violations = count() by DeviceName
| where Violations > 1
| order by Violations desc
```

Power BI on Intune Data — Read This Before You Build

Power BI is where reporting becomes strategic: custom dashboards, trend analysis, compliance over quarters instead of days. The classic path has been to connect Power BI to the **Intune Data Warehouse**, which historically held device, compliance, and configuration data with a longer history than the admin center exposes.

That path still exists, but **two things changed in 2025–2026 that will break naïve dashboards, and I am putting them up front rather than in a footnote.**

⚠ **The "Intune Data Warehouse (beta)" Power BI connector — the v1 connector — is retiring after April 2026.** If your report was built with the beta connector, it stops working. You must migrate to the **Intune connector v2** or, better, connect via the generic **OData Feed** connector. This is the one surviving place where "v1 versus v2" still means something.

⚠ **The Data Warehouse was re-platformed, and its historical retention is being reset (MC1188216).** Historical data drops from roughly 90 days to **about 30 days**. The message was published on **19 November 2025**, and the implementation slipped to **mid-June 2026**. On top of the shorter window, **surrogate keys regenerate** and the **IntuneLicensed** definition changes as part of the re-platform. **Back up anything you care about before mid-June 2026** — once the reset lands, the older history is gone.

Put those together and the conclusion is blunt: **the Data Warehouse is no longer where you keep long-run trend data.** For a 30-day operational view it is fine. For anything longer — year-over-year compliance, multi-quarter deployment trends — you need to be **archiving your own history**, either by scheduling the Graph export jobs from earlier in this chapter into your own storage, or by routing data through Log Analytics (with its own retention) and building Power BI on top of that.

Connecting today, the supported way

The Data Warehouse is an **OData v4 feed** secured with Entra ID (OAuth2), and it supports an `api-version` of `v1.0` or `beta`. To connect:

1. In the Intune admin center, go to **Reports > Intune Data warehouse** and copy the **feed URL**.
2. In Power BI Desktop, choose **Get Data > OData Feed** and paste the URL.
3. Sign in with an account that has the right Intune role.
4. Select the tables you need — Devices, compliance, configuration — and load.

For a more controlled connection in Power Query, pin the implementation and API version explicitly rather than letting it default:

```
OData.Feed(
    "https://fef.<region>.manage.microsoft.com/ReportingService/DataWarehouseFEService?api-version=v1.0",
    null,
    [Implementation = "2.0", Query = [#"api-version" = "v1.0"]]
)
```

The **Intune Compliance (Data Warehouse)** Power BI template app still exists if you want a prebuilt starting point. Just remember the 30-day reality underneath it.

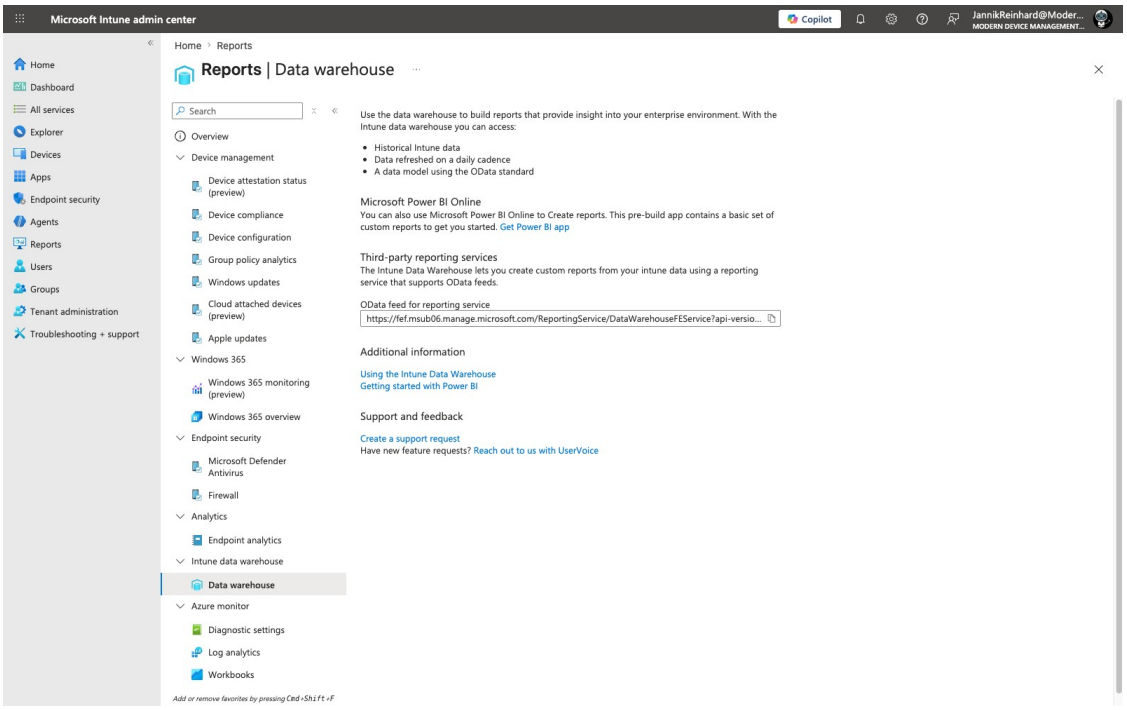

Figure 5.6: Reports > Intune data warehouse. The OData feed URL you point Power BI at. Remember the mid-2026 re-platform that cut retained history to about 30 days

Endpoint Analytics

Everything so far has been about *data you pull*. Endpoint Analytics is Microsoft's own analytics layer *on top of* that data — it scores the end-user experience and tells you where it is bad and, increasingly, why.

There are two tiers, and the line between them is a licensing line that is **about to move**, so pay attention to the dates.

Free base Endpoint Analytics

Included with your Intune plan, at **Reports > Endpoint analytics**:

- **Startup performance** — how fast devices boot and become usable, scored, with the slow devices and the causes (slow disk, too many startup apps, Group Policy processing) called out. Slow boots are a top driver of “my computer is terrible” tickets, and this turns that complaint into a number you can act on.
- **Application reliability** — which apps crash or hang, how often, and on which devices, so you can update, replace, or investigate the unstable ones.
- **Work from anywhere** — how well the fleet is set up for modern, remote work: cloud management, cloud identity, Autopilot readiness, and Windows currency.

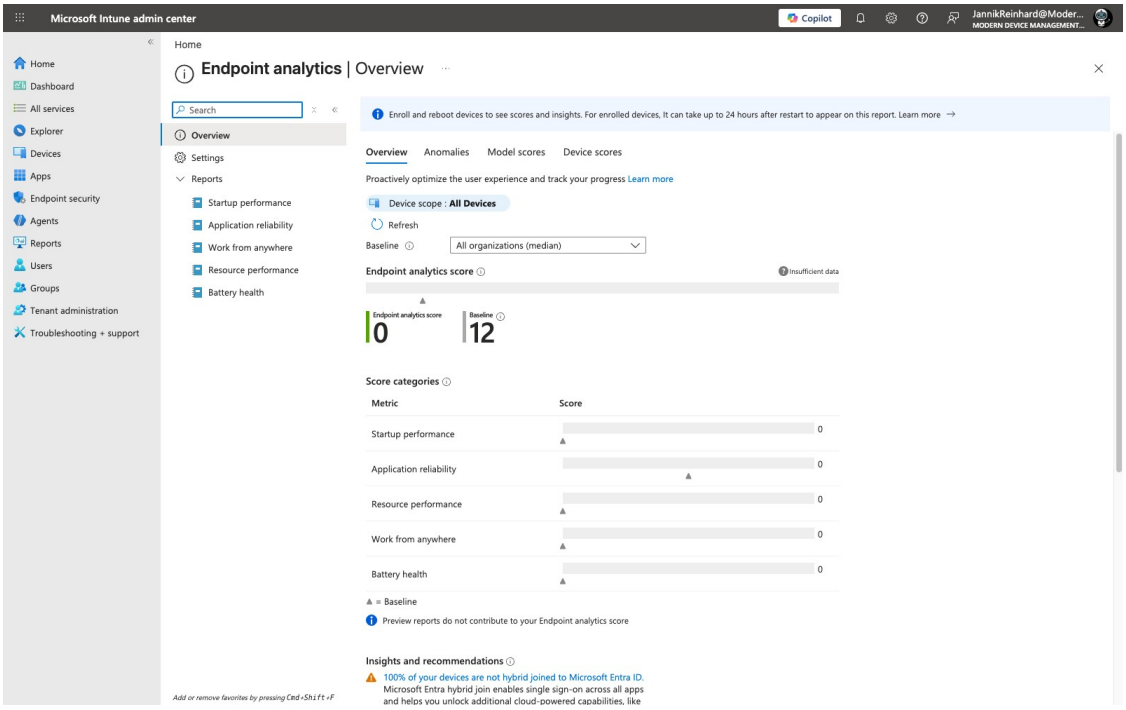

Figure 5.7: Reports > Endpoint analytics overview, with the score and the Startup performance, Application reliability, and Work-from-anywhere reports

A note on **Remediations** (formerly “Proactive remediations”). These are detect-and-fix script pairs — a detection script that exits non-zero when something is wrong, and a remediation script that fixes it — that run on a schedule. They used to live under Endpoint Analytics; they now sit under **Devices > Scripts and remediations**. They remain one of the most useful automation primitives in Intune. A minimal example — detect a bloated temp folder, then clear it:

```
# Detection: exit 1 if %TEMP% exceeds 500 MB
$sizeMB = (Get-ChildItem $env:TEMP -Recurse -ErrorAction SilentlyContinue |
   Measure-Object -Property Length -Sum).Sum / 1MB
if ($sizeMB -gt 500) { Write-Host "Temp is $([math]::Round($sizeMB)) MB"; exit 1 }
Write-Host "Temp within limits"; exit 0
# Remediation: clear the temp folder
Remove-Item "$env:TEMP\*" -Recurse -Force -ErrorAction SilentlyContinue
Write-Host "Temp files cleared."
```

Deploy them under **Devices > Scripts and remediations > Remediations**, assign to a device group, and watch the results roll in. For a large, community-maintained library of these, see the repository a group of us (Andrew Taylor, Florian Salzmann, Joey Verlinden, myself, and many contributors) keep at `github.com/JayRHa/EndpointAnalyticsRemediationScripts`.

Advanced Analytics (and the licensing change you must know)

Advanced Analytics is the paid tier, historically part of the Intune Suite / the Advanced Analytics add-on. It adds:

- **Resource performance** — CPU, memory, and disk pressure across the fleet, so you can spot resource-starved devices and plan hardware refreshes on evidence.
- **Battery health** — battery degradation over time, for refresh planning and power-policy tuning.
- **Anomalies** — automatic detection of health regressions after a change: push a policy, and Advanced Analytics tells you if device health dropped as a result, so you can roll back before the helpdesk lights up. This is the closest thing to the AIOps idea running natively in the product.

- **Device scopes** — segment Endpoint Analytics by scope tag (region, department, IT team) for focused insights.
- **Device timeline** — a richer, lower-latency per-device event timeline for faster root-cause analysis.
- **Device query** — real-time and fleet-wide querying, covered in its own section next.

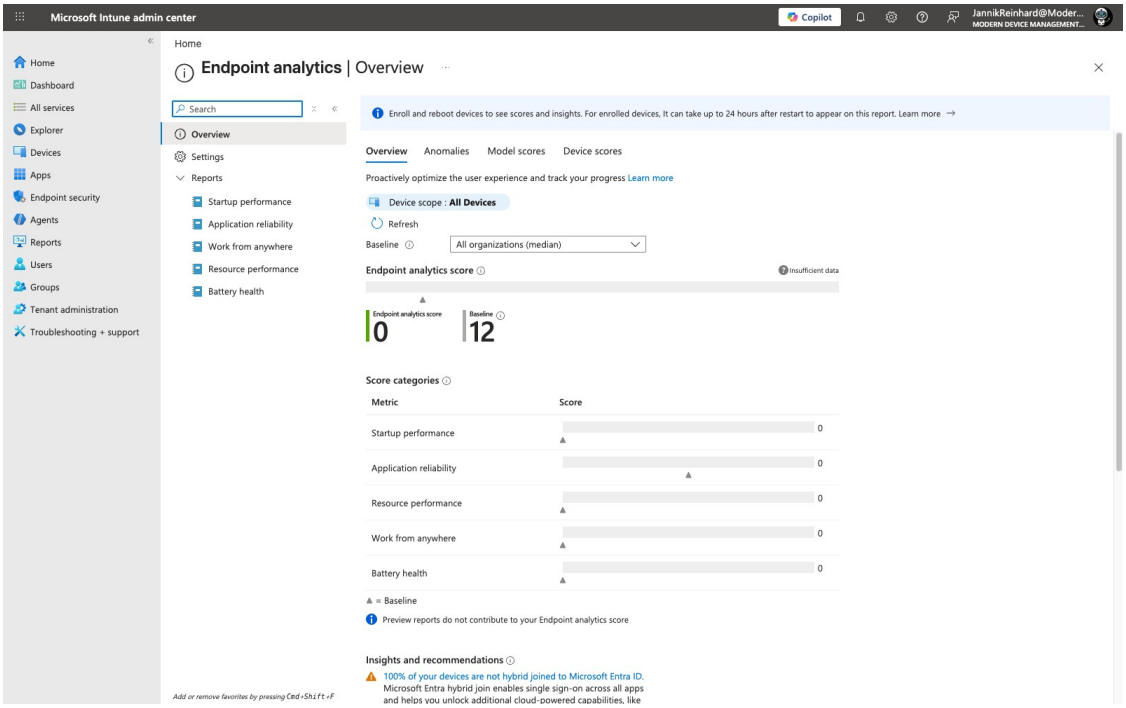

Figure 5.8: The Endpoint analytics reports, including the Advanced Analytics additions Resource performance, Battery health, and Anomalies

⚠️ **Licensing is changing.** On **4 December 2025**, Microsoft announced that **Advanced Analytics, Remote Help, and Intune Plan 2 will fold into Microsoft 365 E3 and E5**, effective **1 July 2026** (with provisioning completing by **1 August 2026**). In plain terms: if you have E3 or E5, Advanced Analytics — and therefore Device Query, Anomalies, Battery health, and the rest — becomes part of what you already pay for rather than a separate add-on. If you priced these features as an extra line item, re-check your licensing; the answer may now be “you already have it.”

One more practical note: the documentation moved. The old `/intune/analytics/` paths now 404; current docs live under `/intune/advanced-analytics/` and `/intune/endpoint-analytics/`. If a bookmarked link is dead, that is why.

Device Query

Device Query is the feature I am most glad graduated from preview. If you came from Configuration Manager you will recognize the idea instantly — ask a device a question and get a live answer — but Intune's version is cloud-native and now works across platforms. It is an **Advanced Analytics feature, so it requires the license** (see the fold-into-E3/E5 change above — this is one of the capabilities that changes hands on 1 July 2026).

There are two modes, and they are genuinely different tools.

Single-device query (real-time)

You select one device and ask it a question that runs **right now**, on the device, over a live channel (it uses Windows Push Notification Services on Windows to wake the device). Results come back in seconds. This is your live-troubleshooting mode: verify a fix landed, check a registry value, read a recent event — without remoting in.

Open a device in the admin center, choose **Device query**, and write KQL against the device's live state:

```
// Live CPU details from the selected device
Cpu
| project ProcessorId, CurrentClockSpeed, MaxClockSpeed, CpuStatus
// Read a specific registry key
WindowsRegistry('HKEY_LOCAL_MACHINE\\SOFTWARE\\...')
| project RegistryKey, ValueName, ValueType, ValueData
// System event-log entries from the last 7 days matching an event id
WindowsEvent('System', 7d)
| where tostring(EventId) == '6008'
| project EventId, LoggedDateTime, LogName, Message, ProviderName
```

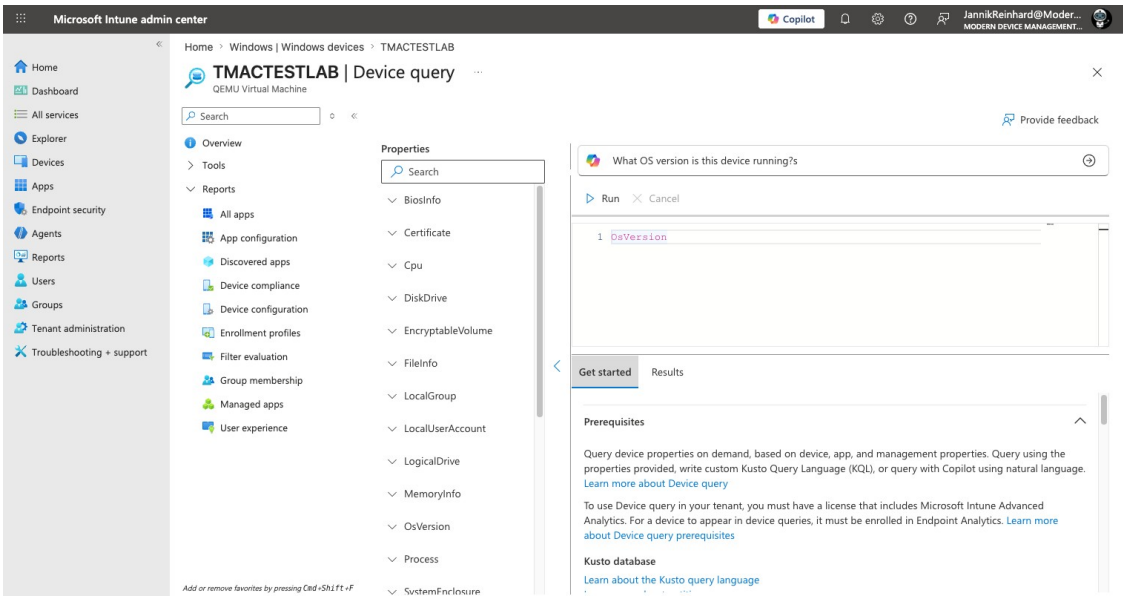

Figure 5.9: Device query on a single device — the property tree on the left, a KQL editor in the middle (here querying OsVersion), and the Copilot box that turns a plain-language request into KQL for you; run it and the live results come back on the Results tab

Multi-device query (fleet-wide)

This is the one that changed the game. Multi-device query went **generally available in February 2025** and lets you run KQL **across your whole fleet** at once — not live on each device, but against inventory data collected within roughly the last 24 hours. “Show me every device where BitLocker is off,” “list all machines with less than 10% free disk,” “which devices are missing this app” — answered across thousands of endpoints in one query.

And it is **no longer Windows-only**. Multi-device query is now **cross-platform**, covering **Windows, Android Enterprise, iOS/iPadOS, and macOS** corporate devices. There is a collection nuance worth knowing:

- On **Windows**, fleet inventory requires a **Properties Catalog inventory policy** (next section) to be assigned — the data has to be collected before you can query it.
- On **Android, iOS/iPadOS, and macOS**, the relevant inventory is **collected automatically**; no extra policy needed.

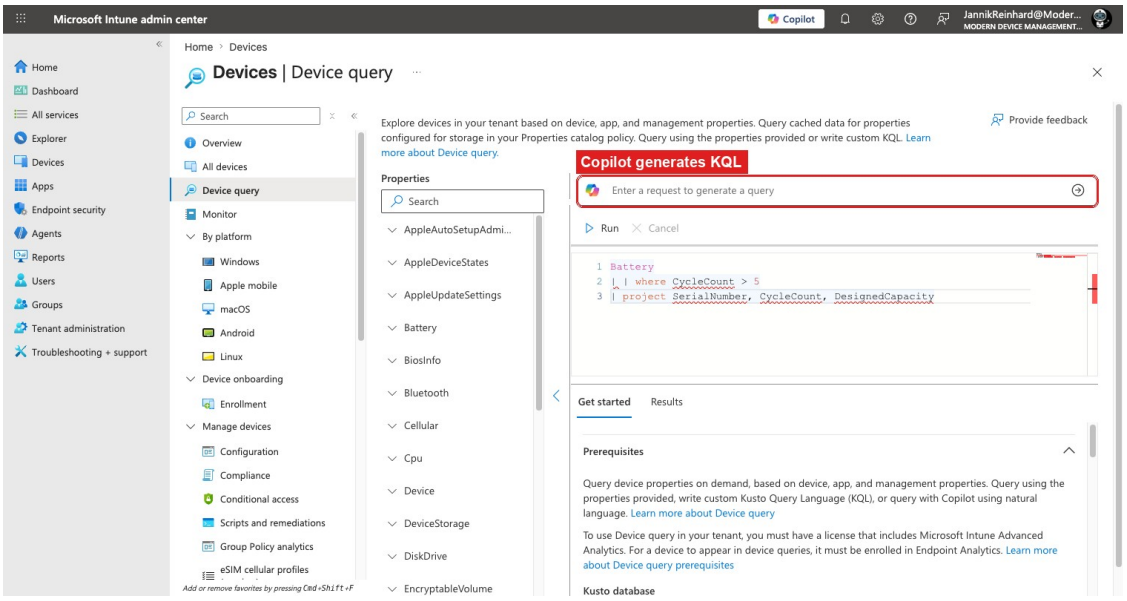

Figure 5.10: Device query: the properties catalog, the Copilot natural-language box that generates KQL, and the KQL editor. Note the Advanced Analytics licensing prerequisite

Limits you will hit

Both modes are governed, and knowing the ceilings saves you from confusing behavior:

- **Single-device:** result payload up to **128 KB**, up to **15 queries per minute**.
- **Multi-device:** up to **50,000 records** per result, at most **3 joins** per query, and rate limits of about **10 queries per minute and 1,000 per month**.

If a multi-device query returns fewer rows than you expect, you are probably brushing the 50,000-record cap — narrow the filter. And because Copilot in Intune can *write* these queries for you (coming up shortly), you do not have to memorize the schema to get value out of Device Query.

Inventory: The Properties Catalog

Multi-device query on Windows depends on inventory being collected, and the mechanism for that is the **Properties Catalog** — Intune's way of gathering custom hardware and system inventory beyond the built-in device properties. This was a long-requested capability, and it is a

core Intune feature with no extra license to collect the inventory itself (querying it at scale via Device Query is the part that wants Advanced Analytics).

Creating a Properties Catalog profile

1. Go to **Devices > Windows > Configuration**.
2. **Create > New policy**.
3. Platform **Windows 10 and later**, profile type **Properties catalog**.
4. Name it, then **Add settings** to pick the inventory properties you want to collect.
5. Assign to the target users or devices, review, and create.

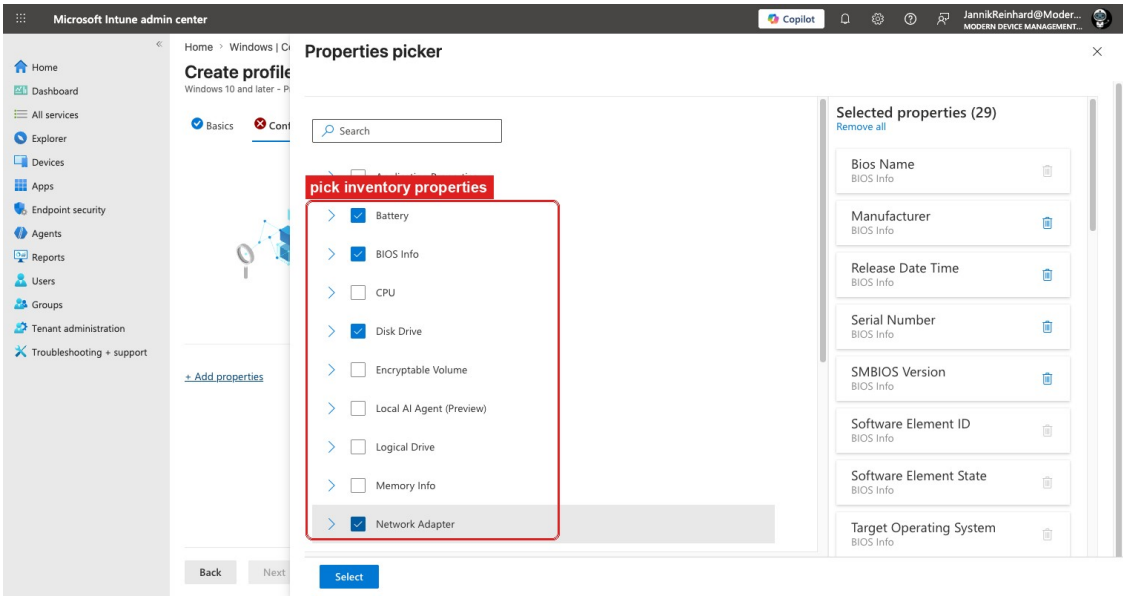

Figure 5.11: The Properties catalog settings picker — choosing which inventory properties Intune collects (Battery, BIOS, disk, network adapter, and so on); each category expands into individual fields shown in the selection pane on the right

Assigning the profile installs a lightweight agent — the **Microsoft Device Inventory Agent** (service name `InventoryService`, files under `C:\Program Files\Microsoft Device Inventory Agent`) — which gathers the configured properties on a schedule. You verify collection two ways: check the service and its logs on a target device, or open **Devices >**

Monitor > Resource Explorer in the admin center, which shows the collected properties per device.

This is also the policy that lights up **App Inventory / Enhanced App Inventory** on Windows, tying back to the very start of the chapter: the modern app-inventory experience is Properties-Catalog-driven, which is why it needs a config policy where the old Discovered apps report did not.

Reading inventory through Graph

Collected inventory is available programmatically — this is how you fold it into your own automation and analytics. The endpoints currently live under `beta`:

```
GET https://graph.microsoft.com/beta/deviceManagement/managedDevices('DEVICE_ID')/deviceInventories
```

For a single inventory topic, expanded to its individual properties:

```
GET https://graph.microsoft.com/beta/deviceManagement/managedDevices('DEVICE_ID')/deviceInventories('Battery')?
$expand=instances($expand=microsoft.graph.deviceInventorySimpleItem/properties)
```

So there are three ways at the same inventory: **Device Query** for a live or fleet-wide question, **Resource Explorer** for a human looking at one device, and **Graph** for automation. Same data platform underneath — you choose the door based on the job.

Copilot in Intune

Everything to this point is you writing queries and reading dashboards. Copilot in Intune is the layer that writes and reads them *with* you — and it is the single biggest addition to this chapter since the first edition.

Copilot in Intune reached **general availability on 14 July 2025**. It is Security Copilot embedded directly in the Intune admin center, grounded in **your** tenant data, and it does a set of things that map directly onto the rest of this chapter:

- **Natural-language data exploration** — ask about your fleet in plain English and get an answer assembled from your own data, instead of clicking through six blades.
- **Per-setting explanations** — inline tooltips that explain what a policy setting does and what changing it will affect.
- **Summarize with Copilot** — one-click summaries of device, policy, and configuration state.
- **Device Query KQL generation** — describe what you want to find and Copilot writes the KQL for you. This is the payoff for the whole KQL section: you should understand the language so you can read and adjust what Copilot produces, but you no longer have to author every query from a blank page.
- **EPM elevation-request analysis**, and **Surface / Cloud PC insights** for those estates.

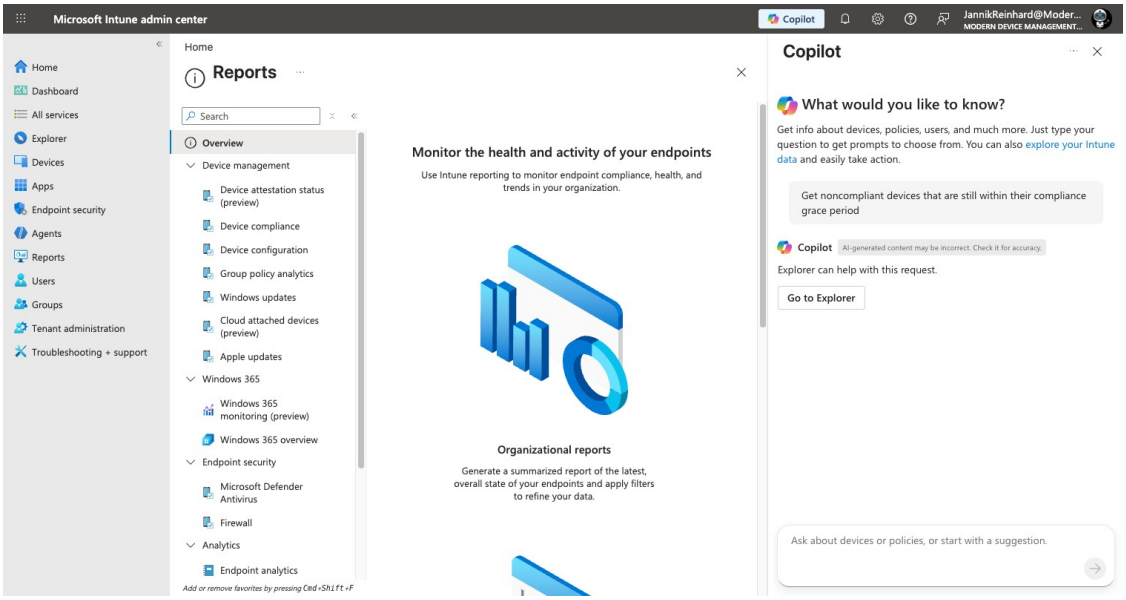

Figure 5.12: Copilot in Intune: the assistant pane grounded in your tenant data, ready to answer natural-language questions about devices, policies, and users and to take action

Turning it on, and what it costs

You enable Copilot in Intune at **Tenant administration > Copilot** in the admin center. The licensing has two parts, and both matter:

- Copilot in Intune runs on **Security Copilot**, so it consumes **Security Compute Units (SCUs)** — the same capacity-based billing model as the rest of Security Copilot — and the **Intune plugin** must be enabled in Security Copilot.
- **Device Query through Copilot still needs the Advanced Analytics license** underneath (which, per the change above, folds into E3/E5 on 1 July 2026). Copilot generates the KQL, but the feature it drives is still gated by that license.

Agents, not just answers

At Ignite 2025 Microsoft moved past "Copilot answers questions" toward **agents that take on tasks** in Intune, built on Security Copilot. The ones announced for Intune include **Change Review**, **Device Offboarding**, **Policy Configuration**, and **Vulnerability Remediation**. These are the same direction of travel as the "Agents" entry now sitting in the Intune navigation — units of AI scoped to a job rather than a chat box. We go deep on the agent model in the security chapters; the point here is that the analytics substrate this chapter builds is exactly what those agents reason over. An agent is only as good as the data and queries beneath it — which is the thread running through this entire book.

Where This Leaves Us

Step back and look at what this chapter assembled. You can find any report and know which audience it serves. You can pull that data through Microsoft Graph, authenticate without a single stored secret, and export at scale with a job you can schedule. You can stream logs into Log Analytics and interrogate them with KQL — the same KQL that Device Query and Copilot speak. You know which of these surfaces have retirement dates and licensing shifts bearing down on them, so you will not build on sand. And you have seen how Microsoft's own analytics — Endpoint Analytics, Device Query, and Copilot — sit on top of the exact same plumbing.

That is the whole foundation. Everything native to Intune, used well, covers a large fraction of real-world needs — and you should exhaust it before building anything bespoke.

This chapter has been the wide-angle overview: the shape of the data and the surfaces that expose it. The next two chapters zoom in on the two that carry the most weight. **Chapter 6** takes **Microsoft Graph** — the data backbone underneath almost everything you just saw — and shows how to work it properly: authentication done keyless, paging, batching, delta queries, throttling, and exporting at scale. Then **Chapter 7** does the same for **KQL, Log Analytics, Workbooks, and Power BI**, turning raw telemetry into analytics you can share, chart, and alert on. Only once those are in hand — in **Chapter 8**, where Part III begins — do we assemble the building blocks into **custom solutions** that run themselves.

Chapter 6: Microsoft Graph — The Data Backbone of Intune

Chapter 5 made a promise I now have to keep. There I said that the Intune admin center is a website built on top of Microsoft Graph, that anything the portal does you can do yourself, and that this single fact is the reason you can automate Intune at all. Chapter 5 was the tour of what Intune gives you and the surfaces you build against. This chapter is the deep, mechanical treatment of the one interface everything else routes through — Microsoft Graph — because if you get Graph right, the rest of the book is straightforward, and if you get it wrong, you will spend your evenings debugging throttling errors and empty result sets instead of building anything interesting.

I want to be honest about why this chapter earns its own place. In the first edition, Graph was a section inside a larger chapter, and it showed — it taught you to make a call and move on, and it skipped almost everything that actually trips people up in production: paging past the first page, filtering server-side instead of pulling ten thousand objects and filtering in your script, tracking what changed instead of re-reading the world every night, surviving the throttle, and exporting a report that is too big to page through by hand. Those are the mechanics that separate a demo from a solution. This chapter is where I slow down and cover them properly, each with a short, runnable example, and all of it keyless — because, as Chapter 5 argued, the modern answer to authentication is to stop storing secrets entirely.

Why Graph Is the Backbone

Microsoft Graph is the single, unified REST API across the whole Microsoft 365 estate — Entra ID, Exchange, Teams, SharePoint, Windows, security, and Intune — reachable at one host, `https://graph.microsoft.com`, behind one authentication model. That unification is the whole point. Before Graph, every workload had its own API with its own auth, its own quirks, and its own SDK. Graph collapsed that into one surface, so the token you use to read a device is

the same kind of token you use to send the report about it by mail, and the paging you learn once works identically everywhere.

For Intune specifically, almost everything lives under a single namespace: `deviceManagement`. Managed devices are `deviceManagement/managedDevices`. Compliance policies are `deviceManagement/deviceCompliancePolicies`. Configuration profiles, the report export API, remediation scripts, the inventory catalog — all of it hangs off `deviceManagement/*`. Once you internalise that the admin center is a `deviceManagement` client, the navigation of Graph stops being a search problem and becomes a mapping problem: which blade in the portal corresponds to which resource under `deviceManagement`. Later in this chapter I will show you the tool that does that mapping for you automatically.

Here is the mental model to hold onto. A Graph request is a URL with four parts:

```
https://graph.microsoft.com/{version}/{resource}?{query-parameters}
```

- **Base URL** — always `https://graph.microsoft.com/`.
- **Version** — `v1.0` for production, `beta` for preview.
- **Resource** — the entity you are addressing: `me`, `users`, `deviceManagement/managedDevices`, and so on.
- **Query parameters** — OData options: `$select`, `$filter`, `$count`, `$expand`, `$top`, `$orderby`.

The standard REST verbs behave as you would expect: `GET` reads, `POST` creates (or, for the reporting and export endpoints, *runs* something), `PATCH` partially updates, `PUT` replaces, `DELETE` removes. A handful of concrete GETs against Intune:

```
GET https://graph.microsoft.com/v1.0/deviceManagement/managedDevices
GET https://graph.microsoft.com/v1.0/deviceManagement/managedDevices/{id}
GET https://graph.microsoft.com/v1.0/deviceManagement/deviceCompliancePolicies
```

Everything for the rest of this chapter is a variation on that shape plus one of the mechanics I am about to walk through.

v1.0 vs beta — When You Have To Use beta, and the Risk

Graph exposes two versions, and the choice between them matters more for Intune than for almost any other workload.

`v1.0` is the production endpoint. It is covered by Microsoft's API stability guarantees: fields do not disappear, behaviour does not change without notice, and it is the version you are entitled to open a support case about. **This is your default. Ship on `v1.0` wherever `v1.0` can do the job.**

`beta` is the preview surface, and it exists to give Microsoft a place to iterate. Here is the uncomfortable truth about Intune specifically: a real amount of the newest and most interesting `deviceManagement` functionality lands in `beta` first and stays there for a long time. Device inventory (the `deviceInventories` endpoint from Chapter 5), some of the newest reports, and most preview features are `beta`-only for months or longer. There is no master list that tells you which functionality graduated to `v1.0` and which has not — you check per endpoint, either by trying it on `v1.0` or by reading the reference doc's version selector.

The risk with `beta` is not that it is unstable in the "it might be down" sense — it is generally reliable — but that it can change under you without warning. A property can be renamed, a response shape can be restructured, an endpoint can move. If your production automation reads a `beta` field by name and Microsoft renames that field, your job breaks silently one morning and you find out from an angry stakeholder. So my rule, unchanged from Chapter 5 and worth repeating because people ignore it: **use `v1.0` where it works, drop to `beta` only where you genuinely must, and never let a `beta` dependency sit in production unmonitored.** If you must use `beta`, pin the exact fields you read, log the raw response so you can diff it when something breaks, and keep an eye on `developer.microsoft.com/graph/changelog`. On the PowerShell side, `beta` is a deliberate choice you make by calling the

`Microsoft.Graph.Beta.*` cmdlets or by targeting a `beta` URI with `Invoke-MgGraphRequest` — the SDK does not silently mix versions for you, and `Select-MgProfile` no longer exists.

Authentication Done Right

Chapter 5 set the direction and Chapter 8 shows it wired into four full solutions, so I will not re-derive the whole argument here. But because this is the Graph chapter, I owe you the complete authentication picture in one place, from the permission model up.

Delegated vs application permissions

Every Graph call runs under one of two permission models, and picking the wrong one is the single most common reason a script that worked interactively fails as a scheduled job.

- **Delegated permissions** mean the app acts *on behalf of a signed-in user*. The effective permission is the intersection of what the app was granted and what the user is actually allowed to do. This is the model for interactive tools, anything with a person at the keyboard, and anything where you want the user's own scope to apply. Delegated scopes for Intune read look like `DeviceManagementManagedDevices.Read.All`.
- **Application permissions** mean the app acts *as itself*, with no user present. There is no user scope to intersect with, so the app has exactly the permissions it was granted, tenant-wide. This is the model for unattended automation — the scheduled runbook, the Function, the Logic App. Application permissions require admin consent, and they are the ones you assign to a managed identity.

The scopes have the same *names* in both models but they are granted differently and they behave differently. A script you tested with `Connect-MgGraph -Scopes ...` (delegated, interactive) will not automatically have those permissions when it runs headless as a managed identity — you have to grant the *application* permission to the identity separately. Chapter 8 has the exact `New-MgServicePrincipalAppRoleAssignment` grant script; the thing to

internalise here is that delegated and application are two different worlds and you must be deliberate about which one your solution lives in.

Least privilege, concretely

Grant the narrowest scope that does the job, and prefer `.Read.All` over `.ReadWrite.All` unless you are actually writing. For the work in this book you will reach for a small set repeatedly:

- `DeviceManagementManagedDevices.Read.All` — read managed devices and their compliance state.
- `DeviceManagementConfiguration.Read.All` — read configuration and compliance policies.
- `DeviceManagementApps.Read.All` — read app and app-assignment data.
- `DeviceManagementManagedDevices.ReadWrite.All` — only when you genuinely take device actions (wipe, retire, sync).

A reporting job that emails a compliance summary needs the first one and nothing else. If you find yourself reaching for `.ReadWrite.All` "to be safe," stop — that is exactly the instinct that turns a read-only report into a tenant-wide liability if the credential is ever compromised.

Keyless: managed identity and workload identity federation

The honest 2026 guidance is short: **stop using client secrets, and prefer no stored credential at all.** Chapter 8 builds this out end to end; the pattern in one paragraph is this. If your code runs *in Azure* — a Function, an Automation runbook, a Logic App, a VM — give the resource a **managed identity**. Azure creates a service principal for it, holds and rotates the credential, and never exposes it to you. You grant that identity the Graph application permissions it needs, and your code authenticates as the identity with no secret anywhere. If your code runs *outside Azure* — GitHub Actions, another cloud, on-prem — use **workload identity federation**: the external workload presents an OIDC token it already holds and exchanges it for a Graph token,

again with nothing stored. Where you genuinely cannot avoid an app-registration credential, use a **certificate**, never a client secret.

Under all of it sits **MSAL**, the Microsoft Authentication Library, which handles token acquisition, caching, and refresh so you rarely touch a raw token. MSAL replaced **ADAL**, which reached end of support on **30 June 2023** — ADAL anywhere in your estate is not merely old, it is unsupported.

The PowerShell module story

The tooling modernised just as sharply, and the retirements have hard dates you need to respect:

- The **Microsoft.Graph PowerShell SDK** is the module to use for Intune. `Connect-MgGraph`, `Invoke-MgGraphRequest`, and the generated `*-Mg*` cmdlets cover `deviceManagement` end to end.
- The legacy **MSOnline** and **AzureAD** modules were deprecated on **30 March 2024** and have since been retired — anything built on them is living on borrowed time.
- Their replacement for identity and directory work is the **Microsoft.Entra** PowerShell module, generally available since **29 January 2025**. For Intune-specific work you stay in Microsoft.Graph; for Entra ID objects (users, groups, service principals, the app-role grant itself) reach for Microsoft.Entra.

In practice the modern connection is two lines. Interactive, for exploration:

```
# Least-privilege delegated scopes for read-only Intune work
Connect-MgGraph -Scopes "DeviceManagementManagedDevices.Read.All",
                        "DeviceManagementConfiguration.Read.All" -NoWelcome
```

Unattended, inside Azure, with a managed identity — and note there is no secret to be seen anywhere:

```
# Runs in a Function, Automation runbook, or VM with a managed identity
Connect-MgGraph -Identity -NoWelcome
```

That `-Identity` switch is the whole modern authentication story for Azure-hosted automation. If your code still passes a client ID and secret into `Connect-MgGraph`, you are carrying a liability you no longer need.

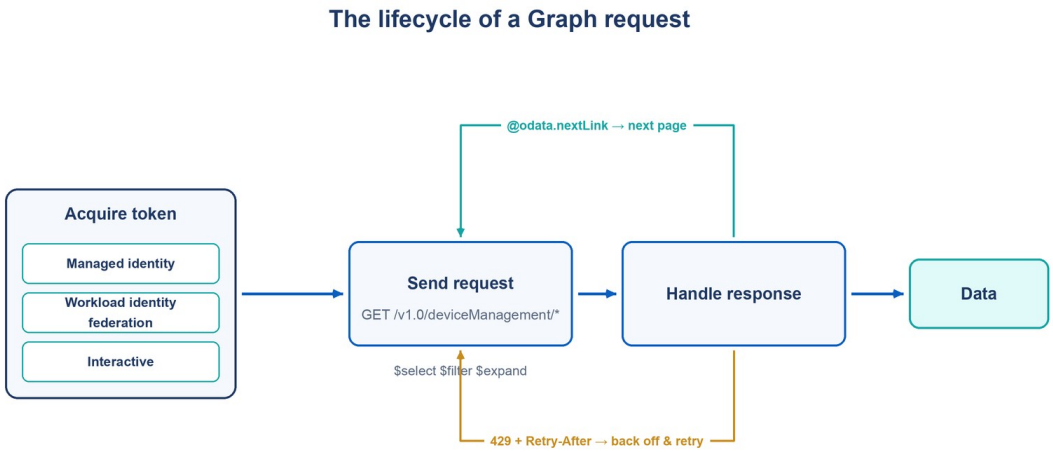

Figure 6.1: The Microsoft Graph request lifecycle — acquire a token, send the request, and handle the two things every real call must handle before you have usable data: paging and throttling.

Tooling: Graph Explorer and Graph X-Ray

Before you write a line of automation, two tools save you hours, and neither is optional in my workflow.

Graph Explorer (`developer.microsoft.com/graph/graph-explorer`) is the interactive console for Graph. You sign in, pick a verb and a URL, add a body if you need one, and run it — against real data (your tenant, if you consent) or a sample tenant. It shows you the response, the required permissions, and code snippets in several languages. This is where you *prototype*: confirm the endpoint exists, see the exact field names in the response, work out the right

`$filter` syntax, and only then paste the validated call into your script. Every export-job body and every OData query in this chapter, I tried in Graph Explorer first.

Graph X-Ray is the tool that turns the "the portal is a Graph client" insight from a nice idea into a practical superpower. It is a browser extension (Chrome and Edge) that adds a tab to the F12 developer tools. With it running, you go to the Intune admin center and *do the thing you want to automate* — create a compliance policy, assign an app, filter a report — and Graph X-Ray captures the exact Graph calls the portal made and generates ready-to-run code for them: the method, the URL, the request body, and a PowerShell snippet (it can also emit C#, Java, and Go). It effectively reverse-engineers any portal action into a script.

This is how you should discover Graph endpoints, and it is faster and more reliable than searching the docs. Suppose you want to automate creating a specific configuration profile with a dozen settings, and you cannot work out the exact JSON body Graph expects. Do not guess. Turn on Graph X-Ray, create the profile once in the portal by hand, and read back the P0ST body it captured — that is your template, verbatim, with the real property names and enum values the service wants. The workflow generalises to anything the portal can do.

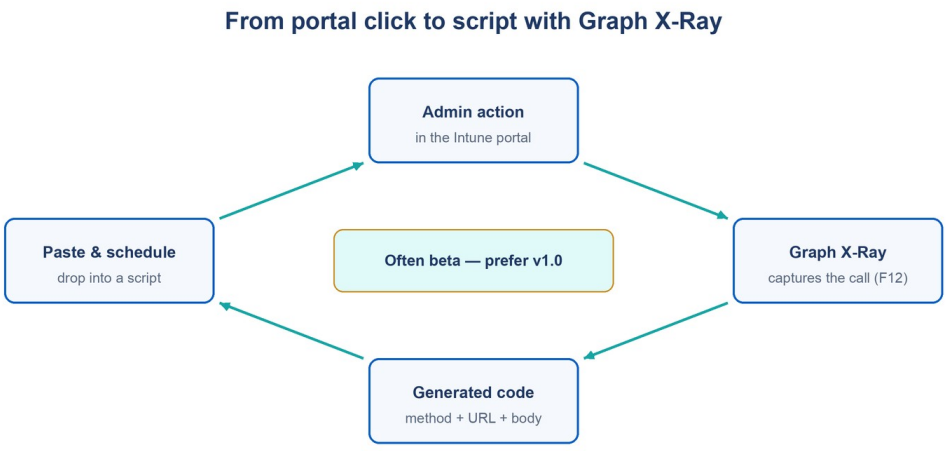

Figure 6.2: Reverse-engineering a portal action with Graph X-Ray — do the thing once in the portal, read back the exact Graph call and generated code, and you have the template for your automation.

A word of caution that applies to both tools and to X-Ray especially: the portal often calls **`beta`**, because the portal ships preview features before they reach `v1.0`. So a call X-Ray hands you may be a `beta` call. Before you build on it, check whether the same resource exists on `v1.0` and prefer that — the X-Ray snippet is a starting point, not a final answer.

The Mechanics That Trip People Up

This is the heart of the chapter. Each of these is a small thing that, ignored, produces a subtly broken solution — the report that is missing half its devices, the job that gets throttled into failure, the nightly export that re-reads the entire fleet to find the three things that changed.

Pagination — @odata.nextLink

This is the one that bites everyone first. Graph does not return unbounded result sets in a single response. A GET against `managedDevices` in a tenant of thousands of devices returns a *page* — commonly the first 100 or 1,000 — plus a property called **`@odata.nextLink`**: a complete URL you GET to retrieve the next page. You keep following `@odata.nextLink` until a response comes back without one, at which point you have everything.

The classic bug is to call the endpoint once, get 100 devices back, and report "we have 100 devices." You have 100 *of* however many there are. Any correct Graph client loops on `@odata.nextLink`:

```
# Page through every managed device manually with Invoke-MgGraphRequest
$uri     = "https://graph.microsoft.com/v1.0/deviceManagement/managedDevices"
$devices = [System.Collections.Generic.List[object]]::new()

do {
   $page = Invoke-MgGraphRequest -Method GET -Uri $uri
   $devices.AddRange($page.value)
   $uri = $page.'@odata.nextLink'   # null on the last page → loop ends
} while ($uri)

Write-Host "Retrieved $($devices.Count) devices."
```

Two things make your life easier. First, when you use the SDK's *generated* cmdlets rather than raw requests, the `-All` switch does this loop for you — `Get-`

`MgDeviceManagementManagedDevice -All` returns every page. Second, `@odata.nextLink` already encodes your `$filter`, `$select`, and page size, so you never rebuild the query when following it — you just GET the URL as-is. In Python with the Graph SDK the pattern is a `while response.odata_next_link` loop, or you use the SDK's page iterator; the concept is identical across languages.

$select, $filter, $count, $expand

These OData query options are how you make Graph do the work instead of your script. Used well, they turn a slow, throttle-prone "download everything and filter locally" job into a fast, precise one.

`$select` limits which properties come back. By default Graph returns a large default property set per object; if you only need three fields, ask for three. This shrinks the payload, speeds up the call, and reduces your throttle footprint:

```
GET https://graph.microsoft.com/v1.0/deviceManagement/managedDevices?
$select=deviceName,complianceState,operatingSystem,osVersion
```

`$filter` filters *server-side*. This is the big one. Filtering in Graph means the service never sends you the rows you do not want; filtering in your script means you pulled them all — and paged through them, and paid the throttle cost — only to throw them away. Get the filter into the query:

```
GET https://graph.microsoft.com/v1.0/deviceManagement/managedDevices?$filter=complianceState eq 'noncompliant'
GET https://graph.microsoft.com/v1.0/deviceManagement/managedDevices?$filter=operatingSystem eq 'Windows' and
complianceState eq 'compliant'
```

A caveat that is very real for Intune: **`deviceManagement` supports a narrower set of `$filter` operators than the rest of Graph.** Not every property is filterable, and some operators (`startswith`, `contains`, complex `and`/`or`) are supported on some entities and rejected on others. When a filter returns a 400, that is usually why — check the reference for the specific

resource, and fall back to a broader server-side filter plus a small client-side refinement if you must.

`$count` asks for the total number of matching objects rather than the objects themselves — cheap when all you need is a number for a dashboard tile. On many resources it requires the `ConsistencyLevel: eventual` header:

```
GET https://graph.microsoft.com/v1.0/deviceManagement/managedDevices/$count
ConsistencyLevel: eventual
```

`$expand` pulls a related entity in the same call instead of making a second round trip. If you want each device *and* its detected apps or its compliance policy states, `$expand` inlines them:

```
GET https://graph.microsoft.com/v1.0/deviceManagement/managedDevices/{id}?$expand=deviceCompliancePolicyStates
```

Combine them freely — `$select` to trim, `$filter` to narrow, `$expand` to enrich — and a call that would have pulled megabytes becomes a tight, purpose-built request.

Request batching — $batch

When you need several *small*, *independent* pieces of data at once — the kind of tenant-summary a dashboard home page shows — do not fire ten sequential requests and pay ten round trips. Send them as one with the **`$batch`** endpoint:

```
POST https://graph.microsoft.com/v1.0/$batch
Content-Type: application/json
```

The body is an array of sub-requests, each with an `id`, `method`, and `url` (relative, no host), plus an optional `body` and `headers`:

```
{
 "requests": [
  { "id": "deviceCount", "method": "GET",
   "url": "/deviceManagement/managedDevices/$count",
   "headers": { "ConsistencyLevel": "eventual" } },
  { "id": "compliancePolicies", "method": "GET",
   "url": "/deviceManagement/deviceCompliancePolicies" },
  { "id": "overview", "method": "GET",
   "url": "/deviceManagement/managedDeviceOverview" }
 ]
}
```

Graph runs them and returns an array of responses keyed by your `id` values. One round trip instead of three, easier on the throttle budget, and noticeably faster. Two things to know: a single `$batch` holds at most **20 sub-requests** (send more than 20 items in multiple batches), and each sub-response carries its *own* status code — a 200 on the batch does not mean every sub-request succeeded, so check each one. When one call must run after another, express `dependsOn` ordering by `id`; for independent reads, leave it off so they run in parallel.

Delta queries — change tracking

Here is the pattern that separates a scalable solution from one that gets slower and more throttle-prone every month. Naïve automation re-reads the entire fleet on every run: pull all 20,000 devices at 2 a.m., diff them against yesterday's copy, act on the differences. That works at 200 devices and falls over at 20,000.

Delta queries invert it. You ask Graph for the *changes* since last time — the devices created, updated, or deleted — and it returns only those, plus a **`@odata.deltaLink`**: a state token you store and present on the next run to get the changes since *this* run. The first call returns the full set (there is no "before" yet) and pages through `@odata.nextLink` as usual; the last page carries the `@odata.deltaLink` instead. You persist that link, and tomorrow you `GET` it to receive only what moved.

```
# First run — full baseline; follow @odata.nextLink to the end,
# then save the @odata.deltaLink from the final page.
GET https://graph.microsoft.com/v1.0/deviceManagement/managedDevices/delta

# Every run after — GET the saved deltaLink; you get only what changed,
# and a fresh deltaLink to store for next time.
GET https://graph.microsoft.com/v1.0/deviceManagement/managedDevices/delta?$deltatoken=<stored-token>
```

Two rules make delta behave. First, if you use `$select` with delta, **only the selected properties are tracked** — a change to an unselected property will not surface as a change, so select everything you care about. Second, deleted objects come back as a small stub carrying an `@removed` annotation rather than a full object, so your code has to recognise that shape and treat it as a removal. Delta is available on `managedDevices` and on the core directory resources (`users`, `groups`, `devices`); as always, confirm per endpoint, because not every `deviceManagement` resource exposes a `delta` function.

Throttling — 429 and Retry-After

Graph is a shared, multi-tenant service, and it protects itself by throttling. When you send requests faster than your limit, Graph responds with **HTTP 429 (Too Many Requests)**, and — on most Graph endpoints — a **`Retry-After`** header telling you how many seconds to wait. The correct behaviour is not to retry immediately or to retry blindly; it is to *honour* *`Retry-After`*, wait exactly that long, and then retry. Backing off by the header value is the fastest way to recover, because Graph keeps counting your usage even while it is throttling you, so hammering it only extends the penalty.

There is one Intune-specific wrinkle you have to plan around: **the `deviceManagement` and reporting endpoints frequently omit the `Retry-After` header.** When there is no header to honour, you fall back to **exponential backoff** — wait 2 seconds, then 4, then 8, then 16, capped at some sensible ceiling — with a little random jitter so a fleet of jobs does not all retry in lockstep. A retry helper that handles both cases:

```powershell
function Invoke-GraphWithRetry {
    param(
        [string]$Uri,
        [string]$Method = 'GET',
        $Body,
        [int]$MaxAttempts = 5
    )
    for ($attempt = 1; $attempt -le $MaxAttempts; $attempt++) {
        try {
            return Invoke-MgGraphRequest -Method $Method -Uri $Uri -Body $Body -ErrorAction Stop
        }
        catch {
            $status = $_.Exception.Response.StatusCode.value__
            if ($status -eq 429 -and $attempt -lt $MaxAttempts) {
                # Honour Retry-After if present; otherwise exponential backoff + jitter.
                $retryAfter = $_.Exception.Response.Headers['Retry-After']
                $wait = if ($retryAfter) { [int]$retryAfter }
                        else { [math]::Pow(2, $attempt) + (Get-Random -Minimum 0 -Maximum 2) }
                Write-Warning "429 throttled. Waiting $wait s (attempt $attempt/$MaxAttempts)."
                Start-Sleep -Seconds $wait
                continue
            }
            throw   # not a 429, or out of attempts — let it surface
        }
    }
}
```

The Microsoft.Graph SDK's generated cmdlets already implement a retry-on-429 policy internally, which is one more reason to prefer them over raw calls for routine work — but the moment you drop to `Invoke-MgGraphRequest` or raw REST, retry handling is *your* job, and a helper like the above belongs in every real solution. The best way to avoid throttling, of course, is to not generate the traffic in the first place: `$select` to shrink payloads, `$filter` to narrow results, `$batch` to combine calls, and delta to stop re-reading unchanged data. Throttle handling is your safety net, not your strategy.

Exporting Reports via the exportJobs Endpoint

Everything above is about reading Intune's *object* data. But some of the richest data in Intune lives in its **reports**, and reports are not object collections you page through — they are tabular datasets, sometimes hundreds of thousands of rows, that you export as a file. Trying to reconstruct the "Devices with inventory" report or a full app-install export by paging `managedDevices` and joining things by hand is a losing game. Intune gives you a purpose-built, asynchronous export API instead, and it is the correct way to pull large report data.

Chapter 5 introduced this endpoint; here I want to give it the full mechanical treatment, because the async pattern — submit, poll, download — is one you will reuse for anything large in Graph. It now lives on **`v1.0`**, so you no longer reach for `beta` just to export:

```
POST https://graph.microsoft.com/v1.0/deviceManagement/reports/exportJobs
```

Step 1 — submit the job

`POST` a body describing what you want. The fields:

- `reportName` (**required**) — the report to export, e.g. `Devices`, `DeviceCompliance`, `DevicesWithInventory`.
- `filter` — an OData-style filter string on the report's columns, optional.
- `select` — the columns to include, optional but strongly recommended for large reports.
- `format` — `csv` (default) or `json`.
- `localizationType` — `LocalizedValuesAsAdditionalColumn` keeps both the raw code and the friendly label, which is what you usually want for downstream processing.

For example, every managed device with its inventory columns, English localisation preserved:

```
{
  "reportName": "DevicesWithInventory",
  "localizationType": "LocalizedValuesAsAdditionalColumn"
}
```

The response comes back immediately with a job **`id`** (something like `DevicesWithInventory_a1b2c3d4-...`) and a `status` of `notStarted` or `inProgress`. The export itself runs on Microsoft's side; you have only *started* it.

There is no single canonical list of every valid `reportName` — Microsoft maintains the reference under "Export Intune reports using Graph APIs" — but the ones you reach for most

are `Devices`, `DevicesWithInventory`, `DeviceCompliance`, `DeviceNonCompliance`, `AllAppsList`, `AppInstallStatusAggregate`, and the discovered-apps/inventory reports. When in doubt which `reportName` sits behind a report, open that report in the portal with Graph X-Ray running — the export button fires exactly this POST, and you can read the `reportName` and `filter` straight out of the captured call.

Step 2 — poll for completion

GET the job by its `id` until `status` reads `completed`:

```
GET https://graph.microsoft.com/v1.0/deviceManagement/reports/exportJobs('{jobId}')
```

While it runs, `status` stays `inProgress`. When it finishes, the response includes a **short-lived `url`** — a pre-authenticated link to the finished file, valid only for a limited window, so download it promptly. Poll on a sane interval (a few seconds), never in a tight loop — the reporting endpoints are throttled and, as noted, often omit `Retry-After`, so a hammering poll is a fast way to get 429'd.

Step 3 — download and unpack

GET the `url`. The file comes back as a **zipped CSV** (or JSON) — unzip it and you have your data.

The export-job pattern: submit, poll, download

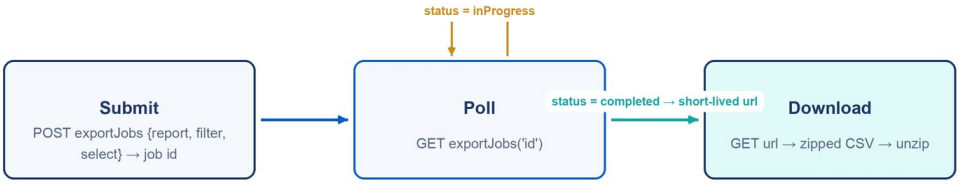

Figure 6.3: The asynchronous export pattern — submit the job, poll the id until completed, download the short-lived file. Reuse this shape for anything large in Graph.

The whole loop in PowerShell

Here is the complete submit-poll-download cycle with the SDK, keyless. Note the `Start-Sleep` in the poll and the guard against an unbounded loop:

```
# Assumes Connect-MgGraph has already run (interactively or with -Identity).

$body = @{
    reportName      = 'DevicesWithInventory'
    localizationType = 'LocalizedValuesAsAdditionalColumn'
} | ConvertTo-Json

# 1. Submit the export job.
$job = Invoke-MgGraphRequest -Method POST `
    -Uri 'https://graph.microsoft.com/v1.0/deviceManagement/reports/exportJobs' `
    -Body $body

# 2. Poll until completed (with a ceiling so a stuck job cannot loop forever).
$deadline = (Get-Date).AddMinutes(15)
do {
    Start-Sleep -Seconds 5
    $job = Invoke-MgGraphRequest -Method GET `
        -Uri "https://graph.microsoft.com/v1.0/deviceManagement/reports/exportJobs('$($job.id)')"
    Write-Host "Status: $($job.status)"
} while ($job.status -ne 'completed' -and (Get-Date) -lt $deadline)

if ($job.status -ne 'completed') { throw "Export did not complete in time." }

# 3. Download and unpack the short-lived URL.
Invoke-WebRequest -Uri $job.url -OutFile './intuneExport.zip'
Expand-Archive './intuneExport.zip' -DestinationPath './intuneExport' -Force
Write-Host "Report downloaded and extracted."
```

Drop that into an Azure Automation runbook or a Function with a managed identity (Chapter 8) and you have a scheduled, credential-free export you can point at a storage account, a Power BI dataset, or a data lake — which is exactly how it feeds the analytics in the chapters ahead.

A Full Worked Example: Devices and Compliance, Keyless, with Paging

Let me pull the pieces together into the example you will actually adapt: read every managed device with its compliance state, using the PowerShell Graph SDK, authenticating keyless, and handling paging correctly. This is the canonical “get the data out of Intune” job, and everything in it is a mechanic from earlier in the chapter.

```powershell
<#
   Managed device + compliance inventory.
   Runs keyless as a managed identity (Function, runbook, VM) — no secret.
   Grant the identity DeviceManagementManagedDevices.Read.All once (see Ch. 8).
#>

# --- 1. Connect, keyless. Falls back to interactive for local development. ---
try   { Connect-MgGraph -Identity -NoWelcome }       # in Azure: managed identity
catch { Connect-MgGraph -Scopes "DeviceManagementManagedDevices.Read.All" -NoWelcome }

# --- 2. Read every device, server-side trimmed with $select, all pages via -All. ---
#     $select keeps the payload small and the throttle footprint low.
$props = 'deviceName','operatingSystem','osVersion','complianceState',
         'userPrincipalName','lastSyncDateTime','manufacturer','model'

$devices = Get-MgDeviceManagementManagedDevice -All -Property $props

Write-Host "Retrieved $($devices.Count) managed devices."

# --- 3. Summarise compliance — the kind of tile a dashboard shows. ---
$byCompliance = $devices |
   Group-Object complianceState |
   Sort-Object Count -Descending |
   Select-Object @{ N = 'ComplianceState'; E = { $_.Name } },
                 @{ N = 'DeviceCount';     E = { $_.Count } }

$byCompliance | Format-Table -AutoSize

# --- 4. Emit the non-compliant devices for action or export. ---
$nonCompliant = $devices |
   Where-Object complianceState -eq 'noncompliant' |
   Select-Object deviceName, userPrincipalName, operatingSystem,
                 osVersion, lastSyncDateTime

$nonCompliant | Export-Csv './noncompliant-devices.csv' -NoTypeInformation
Write-Host "$($nonCompliant.Count) non-compliant devices written to CSV."

Disconnect-MgGraph
```

A few deliberate choices worth calling out. The `try/catch` around `Connect-MgGraph` lets the *same script* run keyless in Azure and interactively on your laptop without editing — a small ergonomic win that makes local development painless. `-Property` maps to `$select`, so the SDK asks Graph for only the eight fields I use. `-All` handles the `@odata.nextLink` paging loop, so this is correct at 200 devices and at 200,000. And the whole thing is read-only with a single least-privilege scope — there is nothing here that could modify a device even if the code had a bug.

If you preferred raw REST — because you are in a language without a mature SDK, or you want precise control — you would replace step 2 with the manual `@odata.nextLink` loop from

earlier and wrap each call in the `Invoke-GraphWithRetry` helper. Which brings me to the last piece of judgement.

When to Use the SDK, When to Use Raw REST

Both are first-class ways to call Graph, and the choice is about leverage, not correctness.

Reach for the SDK — the `*-Mg*` cmdlets in PowerShell, or the Graph SDK in Python/.NET — for routine work against stable `v1.0` resources. You get paging (`-All`), retry-on-429, token management, and typed objects for free, and less code to maintain. This is my default for reading devices, policies, users, groups.

Drop to raw REST — `Invoke-MgGraphRequest` in PowerShell, or a plain HTTP client — when the SDK is in your way. The common cases: a `beta` endpoint the generated cmdlets do not cover cleanly; an unusual request body (the export-job `POST`, a `$batch`); a brand-new endpoint that shipped before the SDK caught up; or when you want to see the literal wire response for debugging. Note that `Invoke-MgGraphRequest` still rides your existing `Connect-MgGraph` session and token, so you keep keyless auth even on raw calls — you are dropping the *typing and convenience*, not the authentication. My rule of thumb: SDK for the 80% that is routine, raw REST for the 20% that is new, weird, or `beta`.

A closing word on rate limits, because it underpins every design decision above. Graph enforces per-app, per-tenant, and per-resource limits, and the Intune and directory endpoints run relatively low thresholds — you *will* meet 429 at fleet scale. So the discipline is not “handle throttling when it happens” but “design so it rarely happens”: select narrowly, filter server-side, batch small reads, track changes with delta instead of re-reading, and export large tabular data with `exportJobs` rather than paging it by hand. Do that, and a job that would have made ten thousand calls makes a few hundred.

Where This Leaves Us

Graph is the backbone, and you now have it properly: one API over the whole estate with Intune under `deviceManagement`; `v1.0` by default and `beta` only where you must and never unmonitored; keyless authentication with the right permission model and least privilege; Graph Explorer to prototype and Graph X-Ray to reverse-engineer any portal action into code; and the five mechanics that make the difference between a demo and a solution — paging, the OData query options, batching, delta, and throttling — plus the `exportJobs` pattern for anything too large to page. The worked example is the shape of almost every data-collection job you will write from here on.

This is deliberately the *plumbing*. We got the data out; we have barely looked at it. Chapter 5 opened the door to Kusto and Log Analytics, and the export jobs and device data from this chapter are exactly what you feed into them. In **Chapter 7 — Advanced KQL, Log Analytics, Workbooks, and Power BI** — we turn from *getting* the data to *interrogating* it: writing KQL that does real analytical work, building Azure Monitor workbooks that visualise a fleet at a glance, and connecting Power BI to trend it over quarters. Graph got the data into your hands. Now let us make it tell you something.

Chapter 7: Advanced KQL, Log Analytics, Workbooks, and Power BI

Chapter 6 was about the pipe. Microsoft Graph is how you get data *out* of Intune — the export jobs, the resource endpoints, the keyless authentication that makes it all runnable on a schedule. But a pipe that runs into a bucket you never look at is not analytics. This chapter is about what you do once the data is flowing: how you turn a stream of raw records into something durable, queryable, and shareable — a dashboard a manager opens on Monday, an alert that pages you before the helpdesk does, a report pack that survives the next re-platform.

I want to set expectations honestly, because this is the chapter where "current" matters most and where two of the tools you might reach for are actively being pulled out from under you in 2026. The Intune Data Warehouse — the thing a lot of people still point Power BI at — is being re-platformed with a hard retention cut in mid-June 2026, and the old Power BI connector that fed it is retiring in April 2026. If your long-run reporting strategy rests on either of those, it is resting on sand, and I will show you where the solid ground is instead. The short version: **Log Analytics for interactive analysis and alerting, Power BI on top of data you archive yourself, and a clear-eyed sense of which questions each tool is actually good at.**

We covered the basics of Log Analytics and KQL in the earlier analytics chapter — diagnostic settings, `project`, `where`, `summarize`, `render`. I am not going to repeat that ground. This chapter assumes you can write a simple query and get a chart back, and it goes deeper: the KQL idioms that make Intune data actually usable, automatic anomaly detection, Azure Workbooks as a real dashboarding surface, scheduled-query alerts wired to action groups, and Power BI done the way that will still work next year. By the end you will have built a compliance-and-health workbook, an alert that fires when non-compliance crosses a threshold, and a Power BI report pack — and you will know which of the four reporting surfaces to reach for on any given day.

Getting the Data to Log Analytics — and What It Costs

Everything in this chapter that is not Power BI runs on a **Log Analytics workspace**, so it is worth being precise about what lands there, how fast, and what it costs — because the last two are where people get surprised.

You wire this up under **Reports > Diagnostics settings** in the Intune admin center, adding a diagnostic setting that streams log categories to the workspace. Intune exposes four categories, and they map to four Log Analytics tables:

Diagnostic category	Log Analytics table	What it holds
AuditLogs	IntuneAuditLogs	Every change: create, update, delete, assign, remote action — who did what, and when
OperationalLogs	IntuneOperationalLogs	Enrollment success/failure and noncompliance detail
DeviceComplianceOrg	IntuneDeviceComplianceOrg	The organizational device-compliance dataset, including noncompliant devices
IntuneDevices	IntuneDevices	Device inventory and status for enrolled, managed devices

To set this up you need the **Intune Service Administrator** role on the Intune side and **Log Analytics Contributor** on the workspace. You can send the same categories to an **Azure Storage account** (cheap archive), an **Event Hub** (stream to a SIEM like Sentinel, Splunk, or QRadar), or Log Analytics — and often you want more than one destination at once.

Routing Intune diagnostics

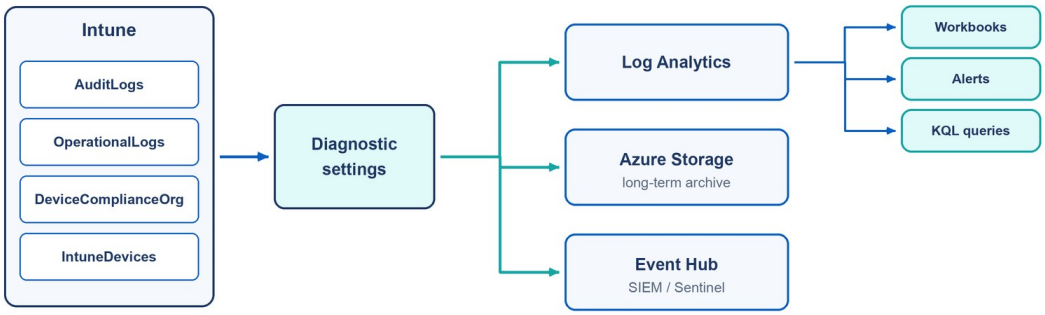

Figure 7.1: How Intune data reaches durable analytics. A diagnostic setting fans the four log categories out to Log Analytics for interactive KQL, Workbooks, and alerts; to a Storage account for cheap long-term history; and to an Event Hub for SIEM streaming. Everything in this chapter hangs off this one configuration.

Two operational realities decide whether your dashboards are trustworthy, and both catch people out:

- **Latency differs sharply by category.** `IntuneAuditLogs` and `IntuneOperationalLogs` arrive almost immediately. `IntuneDeviceComplianceOrg` and `IntuneDevices` are exported roughly **once every 24 hours and can take up to 48 hours** to appear — and Microsoft explicitly does *not* guarantee a fixed time of day. If you build an alert that expects compliance data within minutes, it will look broken when it is merely slow. Design your schedules to tolerate variable export timing rather than assuming a clock.
- **The device tables can duplicate.** The export pipeline may duplicate **up to 100% of the records** in a 24-hour window for the compliance and device datasets. You cannot trust a raw `count()` against them. You de-duplicate downstream in KQL, and the idiom for that — `arg_max()` per device — is the first thing I teach in the next section.

On **cost**: Log Analytics bills primarily on ingestion volume and retention. Audit events are tiny — about 2 KB each — but they add up. Microsoft's own estimate for a 100,000-user tenant is roughly 1.5 million audit events a day and about 90 GB a month of that data; for a 1,000-user tenant it is around 900 MB a month. The device and compliance tables are heavier per row and, because they re-export the whole fleet daily, grow with device count rather than activity. Three practical levers keep the bill sane:

1. **Only enable the categories you will query.** If you never touch enrollment logs, do not stream `OperationalLogs`.

2. **Split retention from analysis.** Keep an interactive window (30–90 days) in Log Analytics at the standard analytics tier, and send a parallel copy to a **Storage account** for long-term archive at a fraction of the price. A Storage account holding a year of audit logs for a mid-size tenant costs a couple of dollars a month; the equivalent in hot Log Analytics retention does not.
3. **Use the Basic/Auxiliary table tiers** for high-volume, rarely-queried logs where they are supported — you pay much less to ingest, at the cost of slower, more limited queries.

The mental model to hold: **Log Analytics is where you *think*; Storage is where you *remember*.** Do not pay hot-query prices to remember three years of audit history you will read once a quarter.

KQL That Earns Its Keep

You already know the shape of KQL — start with a table, pipe it through operators. Here I want to build up the handful of patterns that separate a query that technically runs from one an admin actually relies on. Every column name below matches the real Intune tables, because a misspelled column is the number-one cause of "why is my result empty."

Latest state, not history: `arg_max()`

Because the device tables duplicate and accumulate historical rows, almost every useful query starts by collapsing each device down to its most recent record. `arg_max(TimeGenerated, ...)` does exactly that — for each group, it keeps the row with the maximum `TimeGenerated` and returns the columns you name:

```
IntuneDeviceComplianceOrg
| summarize arg_max(TimeGenerated, ComplianceState, OS) by DeviceName
| summarize DeviceCount = count() by ComplianceState, OS
```

The first `summarize` throws away every stale and duplicate row, leaving one current record per device. The second counts them. Skip that first step and you are counting the same device

five times because it reported five times this week. I treat `arg_max` as mandatory boilerplate on these tables, not an optimization.

Reshaping with `extend` and building categories

`extend` adds computed columns. It is where you turn raw values into the buckets a human wants to see. Here I classify Windows devices into 10 vs 11 by build number and bucket last-sync recency into a freshness label:

```
IntuneDevices
| summarize arg_max(TimeGenerated, OS, OSVersion, LastContact = TimeGenerated) by DeviceId, DeviceName
| where OS == "Windows"
| extend WindowsRelease = iff(toint(split(OSVersion, ".")[2]) >= 22000, "Windows 11", "Windows 10")
| extend Freshness = case(
      LastContact > ago(1d),  "Active (24h)",
      LastContact > ago(7d),  "This week",
      LastContact > ago(30d), "This month",
                              "Stale (>30d)")
| summarize Devices = count() by WindowsRelease, Freshness
| order by WindowsRelease, Freshness
```

`iff` is a two-way branch; `case` is the n-way version. Between them and `extend` you can express almost any business rule your reporting needs without leaving the query.

Joining compliance to device details

The single most common “real” query joins the compliance dataset to device inventory so a noncompliance row carries the context — owner, model, OS build, last sync — that makes it actionable. Note that *both* sides get the `arg_max` treatment first, so you join latest-state to latest-state:

```
let LatestCompliance =
    IntuneDeviceComplianceOrg
    | summarize arg_max(TimeGenerated, ComplianceState) by DeviceId, DeviceName
    | where ComplianceState == "NonCompliant";
let LatestInventory =
    IntuneDevices
    | summarize arg_max(TimeGenerated, OS, OSVersion, UserId, LastContact = TimeGenerated) by DeviceId;
LatestCompliance
| join kind=inner LatestInventory on DeviceId
| project DeviceName, OS, OSVersion, UserId, LastContact, ComplianceState
| order by LastContact asc
```

`kind=inner` keeps only devices present in both sets; switch to `kind=leftouter` when you want every noncompliant device even if inventory has not caught up (remember the 48-hour lag — a freshly noncompliant device may not be in `IntuneDevices` yet, and `leftouter` stops it vanishing from your report).

Trending over time with `bin()`

`bin()` rounds timestamps into buckets so you can watch a metric move. Enrollment failures per day over the last 30 days — the kind of thing you glance at every morning:

```
IntuneOperationalLogs
| where TimeGenerated > ago(30d)
| where OperationName has "Enrollment"
| extend Result = tostring(parse_json(Properties).EnrollmentResult)
| where Result == "Failure"
| summarize Failures = count() by bin(TimeGenerated, 1d)
| render timechart
```

`render` draws it inline — `timechart`, `barchart`, `piechart`, `columnchart`. A `columnchart` is often clearer than a `timechart` for daily counts because the discrete bars map to the discrete days; use a `timechart` when you want to read the *shape* of a trend rather than compare individual days.

`make-series`: the gap-filled time grid

Here is the operator that the basics chapter did not reach, and it is the gateway to everything smart. `summarize ... by bin()` has a subtle flaw for trend analysis: on a day with zero events, there is no row at all, so your series silently skips that day and any downstream math is wrong. `make-series` fixes this by producing a *dense*, *evenly-spaced array* over a time range, filling missing buckets with a default:

```
IntuneOperationalLogs
| where TimeGenerated > ago(30d)
| where OperationName has "Enrollment"
| extend Result = tostring(parse_json(Properties).EnrollmentResult)
| where Result == "Failure"
| make-series Failures = count() default=0 on TimeGenerated step 1d
| render timechart
```

The difference from `summarize` is that `make-series` guarantees exactly 30 daily points, with zeros where nothing happened. That regularity is not cosmetic — the anomaly-detection functions require it, because they reason about a signal sampled at a fixed rate. A ragged series with missing days produces nonsense. So `make-series` is the setup step for the payoff that comes next.

Automatic anomaly detection with `series_decompose_anomalies`

This is the part I most want admins to steal. Watching a chart for spikes works until you have forty charts and a life. `series_decompose_anomalies` lets KQL watch the chart for you: it decomposes a time series into a **baseline trend**, a **seasonal (repeating) component**, and a **residual**, then flags the points where the residual is too large to be normal noise — the actual anomalies.

Conceptually, it separates "what this metric normally does, including its weekly rhythm" from "what it did that it shouldn't have," and only alarms on the second thing.

Figure 7.2: How series_decompose_anomalies thinks. It splits the observed signal into a slow-moving baseline plus a repeating seasonal pattern (weekday-vs-weekend enrollment rhythm,

say), and treats whatever is left over — the residual — as noise. When a residual jumps past a score threshold, that point is flagged as an anomaly. You get spike detection that already knows Monday is busier than Sunday.

Feed it a `make-series` output. The function returns three parallel arrays — a flag array (`1` for a positive spike, `-1` for a dip, `0` for normal), an anomaly *score* array, and the computed *baseline* — which you can render straight onto the chart:

```
IntuneOperationalLogs
| where TimeGenerated > ago(30d)
| where OperationName has "Enrollment"
| extend Result = tostring(parse_json(Properties).EnrollmentResult)
| where Result == "Failure"
| make-series Failures = count() default=0 on TimeGenerated step 1d
| extend (Anomalies, Score, Baseline) =
       series_decompose_anomalies(Failures, 1.5, -1, 'linefit')
| render anomalychart with(anomalycolumns = Anomalies)
```

The `1.5` is the sensitivity threshold — the score, in standard deviations, past which a point counts as anomalous. Lower it (say `1.0`) to catch more and cry wolf more; raise it (`2.5`) to only flag the egregious. The `-1` asks the function to infer the seasonality period automatically; set it to `7` if you know your data has a weekly rhythm and want to pin it. `'linefit'` chooses a linear baseline; `'avg'` uses a flat average when there is no trend.

To turn this from a chart you look at into an answer you can act on, extract just the flagged days:

```
IntuneOperationalLogs
| where TimeGenerated > ago(60d)
| where OperationName has "Enrollment"
| extend Result = tostring(parse_json(Properties).EnrollmentResult)
| where Result == "Failure"
| make-series Failures = count() default=0 on TimeGenerated step 1d
| extend (Anomalies, Score, Baseline) =
       series_decompose_anomalies(Failures, 1.5, -1, 'linefit')
| mv-expand TimeGenerated to typeof(datetime), Failures to typeof(long),
           Anomalies to typeof(long), Score to typeof(double)
| where Anomalies == 1
| project TimeGenerated, Failures, AnomalyScore = round(Score, 2)
| order by TimeGenerated desc
```

`mv-expand` unpacks the parallel arrays back into rows, and the `where Anomalies == 1` keeps only the positive spikes. What comes back is a short list: “on these three days, enrollment failures spiked well above the seasonal baseline, and here is how far above.” That is a query you can put behind an alert — which is precisely what we do later in the chapter. The reason this matters for a device estate is that enrollment-failure spikes, compliance cliffs, and audit-activity surges are exactly the signals that a human staring at a static dashboard misses until Friday afternoon. Letting KQL flag them turns reactive firefighting into something closer to early warning.

A reusable function with `let`

When you have a query you run constantly, wrap it in a `let`-defined function so the logic lives in one place. Here is a parameterized “current non-compliance by platform over the last N days” that you can call with different windows:

```
let NonCompliantByOS = (lookback:timespan) {
    IntuneDeviceComplianceOrg
    | where TimeGenerated > ago(lookback)
    | summarize arg_max(TimeGenerated, ComplianceState, OS) by DeviceId
    | where ComplianceState == "NonCompliant"
    | summarize NonCompliant = count() by OS
};
NonCompliantByOS(7d)
| render barchart
```

Between `arg_max`, `join`, `make-series`, `series_decompose_anomalies`, and `let` functions, you have the whole toolkit for genuinely useful Intune analytics. You do not need more operators than this — you need to combine these well.

Azure Workbooks: Dashboards That Live Where the Data Is

Log Analytics query editing is a great place to *explore*, but you would not hand a colleague a query and ask them to read the results tab. **Azure Workbooks** are the answer: interactive, parameterized reports built directly on your workspace, combining text, charts, grids, and metric tiles on one canvas — no export, no separate tool, no data leaving Azure. A workbook is the natural home for the compliance-and-health dashboard your team checks daily.

A workbook is assembled from **items**, and there are four you will use constantly:

- **Text items** — Markdown, for headings and context so the dashboard explains itself.
- **Parameters** — interactive controls (a time-range picker, an OS dropdown, a workspace selector) whose values flow into every query on the page. This is what makes a workbook feel like an app instead of a static report.
- **Query items** — a KQL query rendered as a grid, chart, tile, or map.
- **Metric/tiles** — the big-number summaries across the top ("4,812 devices · 96.3% compliant · 41 stale").

Parameters, the part that makes it interactive

Parameters are the difference between a workbook and a screenshot. You define a parameter once — say a time range and an OS filter — and reference it in every query with the `{ParamName}` syntax. A time-range parameter named `TimeRange` and a dropdown named `OS` (itself populated by a query, so it always reflects the platforms actually in your tenant) get consumed like this:

```
IntuneDeviceComplianceOrg
| where TimeGenerated {TimeRange}
| summarize arg_max(TimeGenerated, ComplianceState, OS) by DeviceId
| where OS in ({OS}) or '*' in ({OS})
| summarize Devices = count() by ComplianceState
| render piechart
```

Because `{TimeRange}` expands to a proper `where TimeGenerated between (...)` clause and `{OS}` expands to the selected values, one set of controls at the top of the workbook re-drives every tile below it. Change the dropdown to "macOS" and the whole page re-scopes. That is the entire trick, and it is why workbooks feel alive.

Building the compliance-and-health workbook

Here is the layout I actually ship, top to bottom:

1. **A title and a short text block** explaining what the dashboard shows and reminding readers of the up-to-48-hour lag on device data — set expectations on the page so nobody files a "the numbers look old" ticket.
2. **Parameters row**: a time-range picker and an OS multi-select.
3. **A tiles row** with the headline numbers — total managed devices, compliant percentage, count of noncompliant, count of stale (no contact in 30 days). Tiles come from small `summarize` queries rendered as *tiles* with a large value and a subtitle.
4. **A compliance pie** (the query above) beside a **noncompliance-by-OS bar chart**.
5. **A trend timechart** of noncompliant devices per day, so the reader sees direction, not just today.
6. **A detail grid** at the bottom: the compliance-joined-to-inventory query, so a reader can click from "3% noncompliant" straight to *which* devices, with owner and last-sync. Grids support conditional formatting — colour the `ComplianceState` cell red, put a data-bar behind days-since-sync — which turns a table into something you can triage at a glance.

You build all of this in the workbook editor: **Add > Add query** for each visual, pick the workspace, paste the KQL, choose the visualization, and arrange the items. Group related items so they resize together. The result is one page that answers "what is the state of my fleet and where is it slipping" without anyone writing KQL.

Sharing and reuse

A workbook is an Azure resource, so **Azure RBAC governs who can open it** — grant readers `Workbook Reader` (or reader on the resource group) and they see it without touching the query language. You can **pin** individual workbook items to an Azure dashboard for an at-a-glance tile, and you can **export the workbook as a JSON template** and commit it to source control or drop it into an ARM/Bicep deployment — which is how you ship the same dashboard to every tenant you manage instead of rebuilding it by hand. There is also a healthy **Workbooks gallery** of community and Microsoft templates; starting from one and editing beats a blank

canvas. Because the whole thing is JSON, a workbook version-controls as cleanly as code, which is exactly the property you want when a dashboard becomes something the team depends on.

Alerts: From Dashboard to Pager

A dashboard tells you something is wrong *when you look*. An alert tells you *without* you looking. On Log Analytics, that mechanism is the **scheduled query rule** (a "log search alert"): a saved KQL query that Azure Monitor runs on a schedule, compares the result against a threshold, and — when the condition trips — fires through an **action group**.

Three pieces make up an alert:

1. **The query** — the KQL whose result you are testing. Keep it tight: an alert query should return a single number or a small set of rows, not a chart.
2. **The condition, schedule, and window** — the measured column and threshold (e.g. *count greater than 50*), how often the rule runs (**evaluation frequency**, e.g. every 15 minutes), and how far back each run looks (**window**, e.g. the last 1 hour or 1 day). Alert **severity** runs 0 (critical) to 4 (verbose).
3. **The action group** — what happens when it fires: email, SMS, a Teams or webhook post, an Azure Function, a Logic App, an ITSM ticket. One action group can drive several of these at once, and it is reusable across many rules.

Alert 1: non-compliance crosses a threshold

The bread-and-butter Intune alert — page the team when the current noncompliant count exceeds a limit. The query returns one number so the rule can threshold it directly:

```
IntuneDeviceComplianceOrg
| summarize arg_max(TimeGenerated, ComplianceState) by DeviceId
| where ComplianceState == "NonCompliant"
| summarize NonCompliantDevices = count()
```

In the rule, set the measure to `NonCompliantDevices`, condition **greater than 50**, evaluation every hour, window 1 day (wide enough to absorb the export lag). Point it at an action group

that emails the endpoint team and posts to your ops Teams channel. Because the query already collapses to latest-state per device with `arg_max`, the duplication problem cannot inflate the count and cause a false page.

Alert 2: the anomaly alert

This is where the earlier `series_decompose_anomalies` work pays off twice. Instead of a fixed threshold — which is either too tight on quiet days or too loose on busy ones — alert when today's enrollment failures are *anomalous relative to their own baseline*. The rule query reuses the anomaly pattern and returns a row only when today is flagged:

```
IntuneOperationalLogs
| where TimeGenerated > ago(30d)
| where OperationName has "Enrollment"
| extend Result = tostring(parse_json(Properties).EnrollmentResult)
| where Result == "Failure"
| make-series Failures = count() default=0 on TimeGenerated step 1d
| extend (Anomalies, Score, Baseline) =
      series_decompose_anomalies(Failures, 2.0, 7, 'linefit')
| mv-expand TimeGenerated to typeof(datetime), Failures to typeof(long),
          Anomalies to typeof(long)
| where TimeGenerated >= startofday(now()) and Anomalies == 1
```

Set the alert to fire when the **number of results is greater than 0**. When it returns a row, today's failure count broke out of its normal seasonal band, and the team hears about it the same day — not when someone notices the enrollment queue on Thursday. This is a genuinely different class of alert from a static threshold, and it is the practical, in-product version of the "AIOps" idea: let the data model what normal looks like, and only interrupt a human when reality departs from it.

A few things I have learned the hard way about alerts: give the query a wide-enough window to survive the 48-hour device-data lag or you will get flapping false alarms; set severities honestly so a Sev-1 actually means drop-everything; and always route through an action group rather than baking a recipient into the rule, because the day you want to add a Teams channel or an auto-remediation Function, you want to change it in one place. That last point is the seam into the next chapter — an action group can trigger a Logic App or Function, which is how an *alert* becomes an *automated response*.

Power BI on Intune Data — Read This Before You Build

Power BI is where reporting turns strategic: cross-quarter trends, executive dashboards, blending Intune data with HR or asset systems, scheduled distribution to people who will never open Azure. But it is also the surface most exposed to the 2026 changes, so before any how-to, the two facts that will break naïve reports:

> ⚠️ **The "Intune Data Warehouse (beta)" Power BI connector — the v1 connector — is retiring.** The rollout begins **20 April 2026** and completes over about two weeks, after which the beta connector **stops working entirely**. Reports built on it must move to the **Intune connector v2** or, better, the generic **OData Feed** connector. Reports created after November 2025 already use v2 and are unaffected. This is the one place where "v1 versus v2" still means something concrete.
>
> ⚠️ **The Data Warehouse was re-platformed, resetting its history (MC1188216).** The change, published 19 November 2025 and landing **mid-June 2026**, moves the Data Warehouse to new infrastructure and, as a consequence, **resets retained history to roughly 30 days** (down from ~90). It also **regenerates surrogate keys** and **changes the IntuneLicensed definition**. Anything you cared about in the old history is gone after the reset unless you backed it up first.

Put those two together and the conclusion is blunt: **the Intune Data Warehouse is no longer where you keep long-run trend data.** For a rolling 30-day operational view it is fine. For year-over-year compliance, multi-quarter deployment curves, or "show me last spring" — it simply cannot answer any more, because the data will not be there.

So the durable Power BI strategy in 2026 is: **stop treating any Microsoft-hosted store as your history, and archive your own.** Concretely, pick one of these as your source of truth for trends:

- **Scheduled Graph export jobs** (from the previous chapter) writing CSV/Parquet into your own Storage account or a lakehouse, appended over time. You own every day you have ever collected.

- **Log Analytics** as the source, with a parallel Storage archive for anything older than your hot-retention window — and Power BI querying whichever tier holds the range in question.
- **Microsoft Fabric**, if you are building fresh and data-heavy, where the OneLake store *is* your durable history and Power BI sits on top natively.

Power BI then reads *your* archive, not Microsoft's ephemeral one, and no re-platform can erase your trend line.

Connecting today, the supported way

If you do use the Data Warehouse for its 30-day window, connect via the **OData Feed**, not the retiring beta connector:

1. In the Intune admin center, go to **Reports > Intune Data warehouse** and copy the **feed URL**.
2. In Power BI Desktop, choose **Get Data > OData Feed** and paste it.
3. Sign in with an account holding the right Intune role (Entra ID / OAuth2 secures the feed).
4. Select the entities you need — devices, compliance states, configuration — and load.

For a controlled connection in Power Query, pin the implementation and API version rather than letting them default:

```
OData.Feed(
    "https://fef.<region>.manage.microsoft.com/ReportingService/DataWarehouseFEService?api-version=v1.0",
    null,
    [Implementation = "2.0", Query = [#"api-version" = "v1.0"]]
)
```

A simple report pack

A useful starter pack is three pages sharing one data model:

1. **Fleet overview** — cards for total devices, compliant %, enrolled-this-month; a donut of devices by platform; a bar of devices by ownership (corporate vs personal).
2. **Compliance** — a matrix of compliance state by OS and by policy, with a trend line if your source carries history; conditional formatting to make the red obvious.
3. **Adoption/lifecycle** — Windows 11 adoption, OS-version distribution, and a bar of devices by age or last-sync recency for refresh planning.

Model it properly — a device dimension, a compliance fact, a date table — so slicers (platform, ownership, date) cross-filter every visual. Set a scheduled refresh in the Power BI service, and if your source is your own archive, the history grows with you regardless of what Microsoft's warehouse does. The **Intune Compliance (Data Warehouse)** template app still exists as a starting point; just remember the 30-day reality beneath it.

When Power BI beats a Workbook (and when it doesn't)

They overlap, so use the tool whose strengths match the job:

- **Power BI wins** at scheduled distribution to non-technical audiences (email a PDF every Monday), blending Intune data with other sources (HR, CMDB, finance), rich slicer-driven self-service exploration, a polished executive look, and — crucially — **long-term history you control**, since it reads your archive.
- **Workbooks win** at living next to the raw logs with zero data movement, being free (no Power BI licensing), using the exact same KQL as your alerts and Device Query, deploying as version-controlled JSON templates across tenants, and being the fastest path from "I need to see this" to "here it is" for an engineering audience.

My rule of thumb: **if the audience is engineers and the data lives in Log Analytics, build a Workbook. If the audience is management, or you are joining Intune to other business data, or you need controlled history, build Power BI on an archive you own.**

A Decision Guide: Which Reporting Surface When

You now have four ways to report on Intune data, and the most common mistake is over-building — spinning up Log Analytics and Power BI for a question the admin-center report already answers in one click. Match the tool to the need:

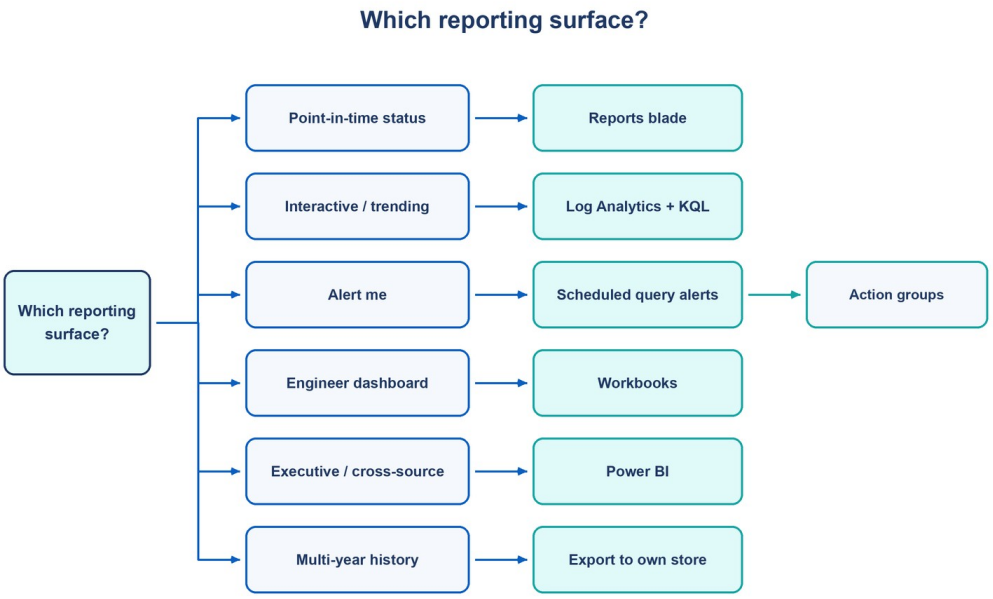

Figure 7.3: Choosing a reporting surface. Start from the question, not the tool. A one-off answer for right now belongs in the Reports blade; interactive analysis, alerting, and engineer-facing dashboards belong in Log Analytics and Workbooks; executive and cross-source reporting belongs in Power BI; and durable, multi-year history means exporting to a store you own — because the Data Warehouse only remembers ~30 days.

Need	Reach for	Why
A point-in-time answer, right now — "which devices failed this policy?"	**Reports blade** (admin center) + export	Fastest, no setup, always current, and it is the source of truth for operational state. Exhaust this before building anything.
Interactive analysis, ad-hoc questions, trending, correlation across logs	**Log Analytics + KQL**	Full query language, joins across tables, make-series and anomaly detection. Where you *think*.
"Tell me automatically when X goes wrong"	**Scheduled query alerts → action groups**	Threshold and anomaly alerting, routed to email/Teams/automation. No human has to be watching.
A live, shareable, engineer-facing dashboard on log data	**Azure Workbooks**	Parameterized, free, same KQL, deploys as a template, no data movement.

Need	Reach for	Why
Executive reporting, cross-source blending, scheduled distribution	**Power BI**	Rich modelling, slicers, PDF subscriptions, joins to non-Intune data — on an archive you own.
Multi-year history, your own retention and schema, ML later	**Export to your own store** (Graph jobs → Storage/lakehouse/Fabric)	The Data Warehouse keeps ~30 days; Log Analytics keeps your hot window. Only *your* store keeps forever.

The through-line is that these are layers, not competitors. The Reports blade and Graph exports are the raw source. Log Analytics and Workbooks are the interactive and alerting layer on top of the streamed logs. Power BI is the presentation and blending layer. And your own store is the memory that outlives every one of Microsoft's retention windows. A mature setup uses several at once: diagnostic settings streaming to Log Analytics for alerts and workbooks, a parallel Storage archive for cheap retention, and Power BI reading the archive for the long-run picture — with the admin center always there for the immediate operational question.

Where This Leaves Us

Step back and look at what this chapter built on top of the Graph pipeline from the last one. You can get Intune's logs into Log Analytics and reason honestly about their latency, duplication, and cost. You can write KQL that collapses noisy device data to latest-state, joins compliance to inventory, builds gap-free time series, and — the part I most hope you take away — flags anomalies automatically instead of waiting for a human to spot a spike. You can assemble that KQL into a parameterized Workbook your team actually opens, wire a scheduled-query alert to page you the moment non-compliance or enrollment failures break their normal band, and build a Power BI report pack that survives the 2026 Data Warehouse re-platform because it reads history you archived yourself. And you have a decision guide so you reach for the right surface instead of over-engineering the simple questions.

There is a natural next question hiding in all of this. An anomaly alert that fires into an action group is one webhook away from an action group that fires into a Logic App or an Azure Function that *does something about it* — re-runs a compliance evaluation, opens a ticket, quarantines a device, posts an enriched summary to Teams. The reporting we built here is the sensing half of a control loop; the acting half is code you write and host yourself. That is exactly

where **Chapter 8** goes. We take these same building blocks — Graph, keyless authentication, KQL, Log Analytics, and alerts — and assemble them into **custom analytics and automation solutions**: Logic Apps, Automation runbooks, remediation scripts, and Azure Functions that turn everything you can now *see* into things that run, and fix, on their own.

Part III

Automation: Making It Run Itself

Chapter 8: Building Custom Analytics and Automation Solutions with Intune

Part II was about getting data *out* of Intune and making sense of it — the analytics surfaces in Chapter 5, Microsoft Graph in Chapter 6, and KQL, Workbooks, and Power BI in Chapter 7. This chapter turns the corner into Part III and into automation: it is about building something that runs on its own — a solution that pulls the data on a schedule, does something with it, and puts the result where a human (or another system) will see it. Four patterns cover almost everything you will ever want to build: Microsoft Graph with a Logic App, an Azure Automation runbook, an Intune remediation script, and an Azure Function. I use all four in production, and by the end of this chapter you will have a concrete, working example of each.

Before any of that, I have to be honest about the single biggest thing that changed since the first edition. The first edition authenticated everything with an app registration and a client secret — create an app, generate a secret, paste it into your script or your Logic App, and hope you remembered to rotate it before it expired. That pattern still works, but in 2026 it is the wrong default. **The current best practice is keyless.** For anything hosted in Azure — Logic Apps, Automation runbooks, Functions — you use a **managed identity**. For anything running outside Azure, such as a GitHub Actions workflow, you use **workload identity federation**. In both cases there is no secret to store, leak, or rotate. I have rewritten every example in this chapter around that model, and there is a dedicated section on it below because it is the part people get wrong.

Planning Your Custom Solution

It is tempting to open the Azure portal and start clicking. Resist that for ten minutes and sketch the solution first. A custom analytics or automation solution is a small data pipeline — something produces data, something moves it, something processes it, something stores or presents it — and if you cannot draw those boxes and arrows, you do not yet understand what you are building.

For that sketching I use two tools, both of which live inside VS Code so the diagram sits next to the code in the same repository:

- **Excalidraw** — for quick, hand-drawn-looking sketches. When you save a file as `*.excalidraw.png` you can drop it straight into a `README.md` and keep editing it later without leaving the editor. This is what I reach for first, when the idea is still rough.
- **draw.io (diagrams.net)** — for the polished architecture diagram, because it ships the official Azure and Microsoft icon sets. Saved as `*.drawio.png`, it is both a viewable image and an editable source, so it version-controls cleanly.

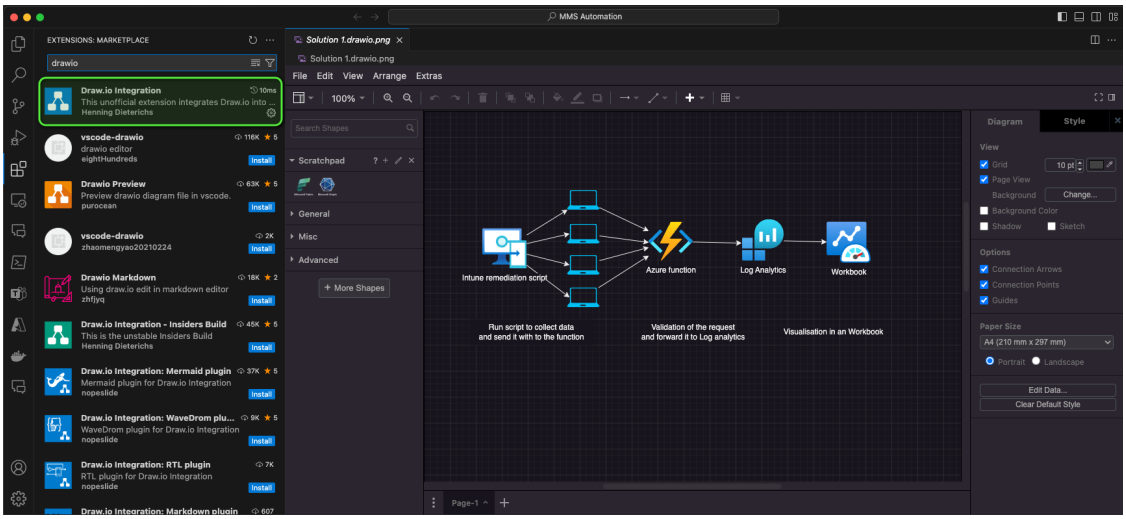

Figure 8.1: Sketching the solution in VS Code with the draw.io extension — the official Azure and Microsoft icon sets let you draw a real architecture diagram (here an Intune remediation script feeding an Azure Function, Log Analytics, and a Workbook) that version-controls next to the code as a .drawio.png

The point is not the tool. The point is that a solution you can draw is a solution you can reason about, hand to a colleague, and troubleshoot when it breaks at 2 a.m.

Interfaces for Getting Data Out of Intune

Chapter 5 went through these in detail, so this is a deliberately short recap of the doors you can build against — because your choice of interface shapes the whole solution.

1. **Microsoft Graph API.** The primary way in. Almost everything you see in the admin center is a Graph object, and almost every action you can take in the UI you can take through Graph. For most solutions you call the standard resource endpoints (a device, a policy, an assignment). For large tabular exports you use the **report export jobs** endpoint — `POST /v1.0/deviceManagement/reports/exportJobs` — which produces the same CSV or JSON files the reporting UI generates, and which you poll until the job completes. Prefer the `v1.0` endpoints; reach for `/beta` only when the data you need genuinely is not on `v1.0` yet, and expect `/beta` to change under you.

2. **Log Analytics via diagnostic settings.** Under **Reports > Diagnostics settings**, Intune can stream its audit logs, operational logs, device-compliance data, and device inventory into a Log Analytics workspace, where you query it with Kusto (KQL). This is the right interface when you want history and trend analysis rather than a point-in-time snapshot. Note the latency: audit and operational logs land almost immediately, but the device tables can take up to 48 hours and may duplicate rows inside a 24-hour window, so de-duplicate downstream.

3. **Intune Data Warehouse.** Still here, still an OData v4 feed, but modernized — and be aware that a 2025/2026 re-platforming reset its historical retention to roughly 30 days. It remains a reasonable source for Power BI trend reporting through the **OData Feed connector**, but the old "Intune Data Warehouse (beta)" Power BI connector is retired, and the shrunken retention means you should archive your own exports if you need long-run history.

4. **Scripts and remediations.** Formerly "proactive remediations," now found under **Devices > Scripts and remediations**. This is the only interface on this list that runs code *on the device*, which makes it uniquely powerful for collecting data that Graph simply does not have — local registry state, a hardware detail, the result of a WMI query — and for fixing things in place. Solution 3 is built entirely on it.

The first edition also listed third-party agents here. I still would not lead with them: they add cost, add load on the endpoint, and add another vendor to trust, and between Graph, Log Analytics, and remediation scripts you can build almost anything natively. Evaluate them only when a native path genuinely does not exist.

Working with, Processing, and Storing Data

Once you know how the data leaves Intune, decide where it lands and what chews on it. Azure has more services here than any one solution needs; these are the ones I actually reach for:

- **Azure Storage** (Blob, Table, File) — cheap, durable landing zone for raw exports before you process them.
- **Azure SQL Database** — when the processed data is relational and you will query or join it repeatedly.
- **Azure Functions** — serverless code triggered by an event, a timer, or an HTTP call. The workhorse for "when this happens, transform it and move it on."
- **Log Analytics / Azure Monitor** — not just a log sink but a genuine analytics store with KQL, alerting, and workbooks on top.
- **Power BI** — for dashboards and reports stakeholders will actually open.
- **Microsoft Fabric** — the unified analytics platform (data engineering, warehousing, real-time analytics, and Power BI in one SaaS surface). If you are building something new and data-heavy, Fabric is increasingly where I would start rather than stitching separate services together.
- **Azure Machine Learning** — when you move past reporting into prediction or anomaly detection on your own models.

That is a small subset, but it is enough to build every solution in this chapter and most of the ones you will invent afterwards.

A Word on Authentication: Go Keyless

This is the section to read twice. Every solution below depends on it, and getting it right once means you never paste a secret into a script again.

Why the old way is out. An app registration with a client secret means a credential exists somewhere — in a variable, in a Logic App connection, in a pipeline secret store — that grants access to your tenant and eventually expires. Secrets leak, secrets expire silently and break your automation at the worst moment, and secrets have to be rotated by a human who will forget. Managed identities and workload identity federation remove the secret entirely.

Managed identity — for anything hosted in Azure. A managed identity is a service principal that Azure creates and manages for a specific resource. A Logic App, an Automation account, and a Function app can each have a **system-assigned managed identity**: turn it on, and the resource now has an identity in Entra ID whose credential Azure rotates automatically and never exposes to you. Your code authenticates *as that identity* with no secret in sight. This is the model I use for every Azure-hosted example in this chapter.

Workload identity federation — for anything outside Azure. If your automation runs somewhere Azure does not manage its identity — most commonly **GitHub Actions**, but also GitLab, or another cloud — you use workload identity federation. You configure a **federated credential** that trusts an external OIDC issuer (GitHub's token endpoint, for example), and at runtime the external platform presents a short-lived token that Entra ID exchanges for a Graph token. Again, no secret is stored. In a GitHub Actions workflow this is the `azure/login` action with `client-id`, `tenant-id`, and `subscription-id` and a `permissions: id-token: write` block — and crucially *no* `client-secret`. You can even federate a managed identity as the trusted issuer, which keeps everything under the managed-identity model.

Granting Graph permissions to a managed identity. A managed identity starts with no permissions. Because there is no consent UI for a managed identity, you grant it Microsoft Graph **application permissions** by creating an app-role assignment against Microsoft Graph's

service principal. You do this once, interactively, with the Microsoft Graph PowerShell SDK — and you use the SDK, not the old modules. **MSOnline and AzureAD are retired; do not use them.** The current modules are `Microsoft.Graph` (and `Microsoft.Entra` for directory work); the underpinning is MSAL, not the long-dead ADAL.

Here is the script I run to grant a managed identity read access to Intune devices. Adjust `$permission` for whatever the solution needs (for example `Mail.Send` to send reports, `DeviceManagementConfiguration.Read.All` to read policies):

```
# Grant a Microsoft Graph application permission to a managed identity.
# Run once, interactively, as an admin who can assign app roles
# (e.g. Privileged Role Administrator).

# Install-Module Microsoft.Graph -Scope CurrentUser   # if not already installed

Connect-MgGraph -Scopes "Application.Read.All","AppRoleAssignment.ReadWrite.All" -NoWelcome

# Object ID of the managed identity's service principal.
# Find it on the resource's Identity blade, or via Get-MgServicePrincipal.
$managedIdentityId = "<managed-identity-object-id>"

# The Graph application permission to grant.
$permission = "DeviceManagementManagedDevices.Read.All"

# Microsoft Graph's own service principal (well-known, first-party app id).
$graphSp = Get-MgServicePrincipal -Filter "appId eq '00000003-0000-0000-c000-000000000000'"

# Find the matching application role.
$appRole = $graphSp.AppRoles | Where-Object {
    $_.Value -eq $permission -and $_.AllowedMemberTypes -contains "Application"
}

# Assign it: PrincipalId and ServicePrincipalId are both the managed identity;
# ResourceId is Microsoft Graph.
New-MgServicePrincipalAppRoleAssignment `
    -ServicePrincipalId $managedIdentityId `
    -PrincipalId        $managedIdentityId `
    -ResourceId         $graphSp.Id `
    -AppRoleId          $appRole.Id

Disconnect-MgGraph
```

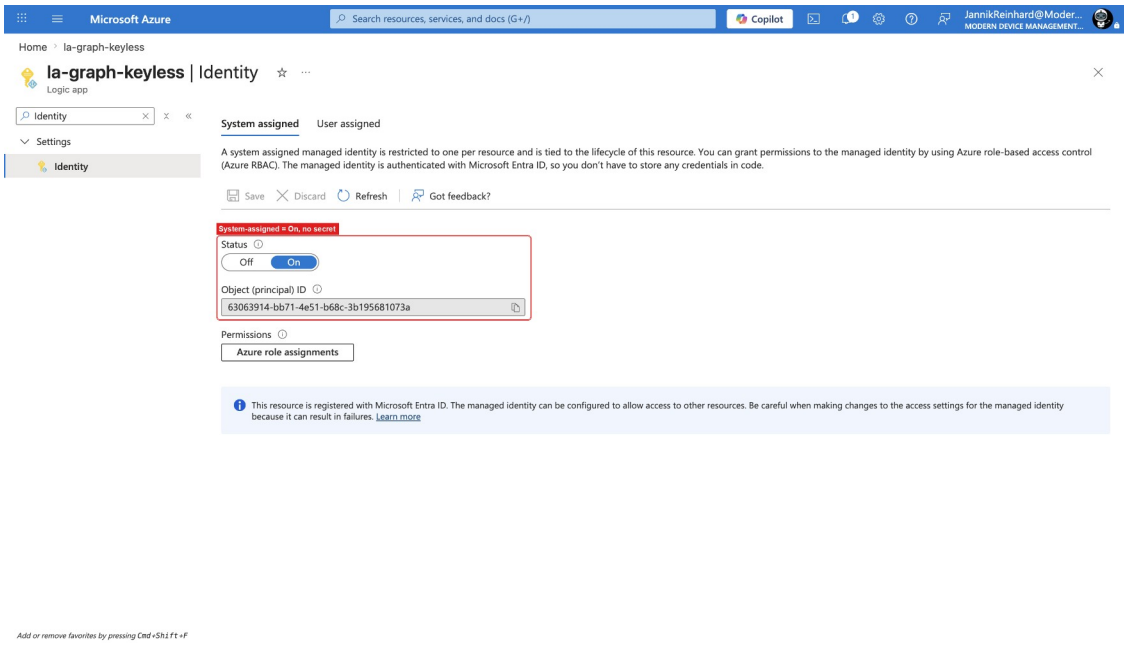

Figure 8.2: A resource's Identity blade with System assigned set to On. The Object (principal) ID is the identity you grant Graph permissions to — and, as the note says, it is authenticated with Entra ID so you store no credentials in code

Grant the least privilege the solution needs and nothing more — a reporting runbook needs `.Read.All`, not `.ReadWrite.All`. From here on, every solution assumes you have turned on the resource's system-assigned identity and run the grant above for the permissions it uses.

Designing Your Solution: Step by Step

With the interfaces, the storage options, and the auth model in hand, the design process is short:

1. **Define the objective.** What question are you answering, or what action are you automating? "Email me the Windows 11 adoption rate every morning" is a good objective; "do something with device data" is not.
2. **Choose the data sources.** Graph, Log Analytics, the Data Warehouse, or on-device via a remediation script — pick the interface that actually holds what you need.

3. **Plan collection.** Real-time or scheduled? A single Graph call or a report export job? Push (diagnostic settings) or pull (your code)?
4. **Design the processing.** Clean, filter, aggregate — and decide where that runs. A Logic App for glue, a runbook or Function for real logic.
5. **Build the automation.** Wire the trigger to the work, using a managed identity for auth.
6. **Design the output.** A Teams message, an email, a Power BI dashboard, or a write-back action against Intune.
7. **Test on a subset**, then **deploy and monitor** — set up alerting so you learn about failures before your users do.

Now let us build.

Solution 1: Microsoft Graph and Logic Apps

Logic Apps are the low-code option: a visual workflow of a trigger and a series of actions, no server to manage. They are perfect for glue — "on a schedule, call Graph, shape the result, post it somewhere." Our example sends a daily Teams message with the current device count. It is deliberately simple, but the pattern generalizes to any Graph call or combination of calls.

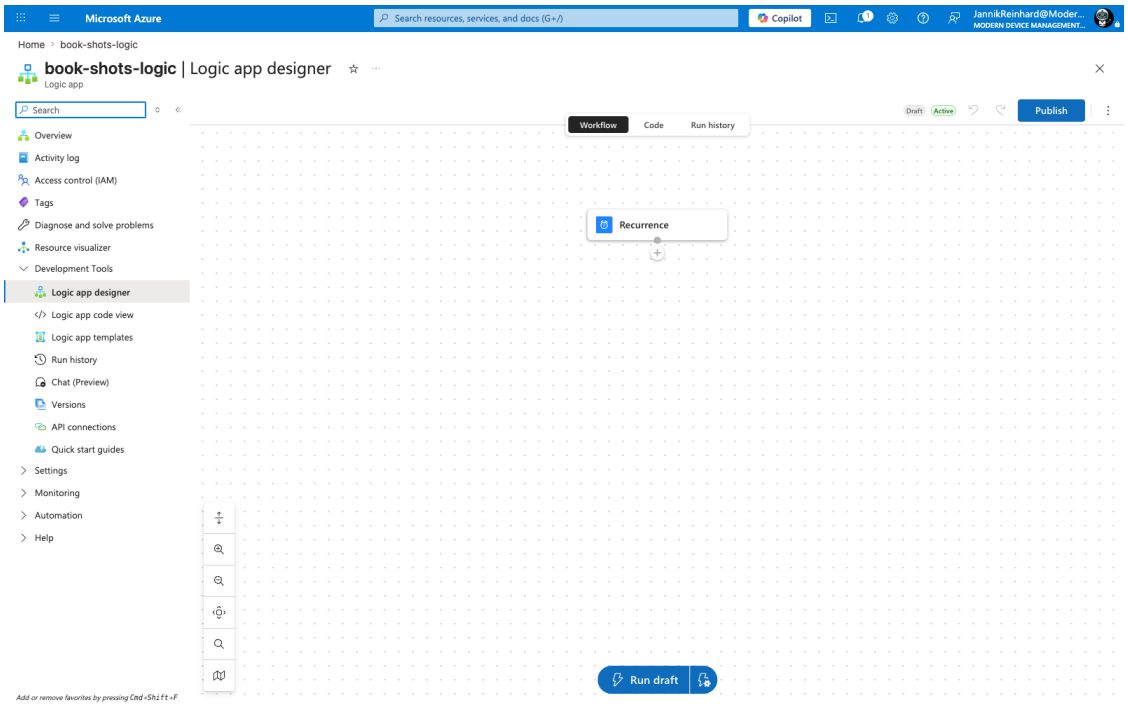

Figure 8.3: The Logic App Designer: a Consumption workflow with a Recurrence trigger, ready for the HTTP, Parse JSON, and downstream actions

Step 1 — Create the Logic App. In the Azure portal, search for **Logic App** and create one. For development choose the **Consumption** plan — you pay per execution and it is more than enough for a scheduled report. Give it a resource group, a name, and a region, then **Review + create**.

Step 2 — Turn on the managed identity. Open the new Logic App, go to **Settings > Identity**, and switch the **System assigned** identity to **On**. Copy the Object (principal) ID. Then run the grant script from the auth section with `$permission = "DeviceManagementManagedDevices.Read.All"`, using that Object ID. This is what lets the Logic App call Graph with no secret.

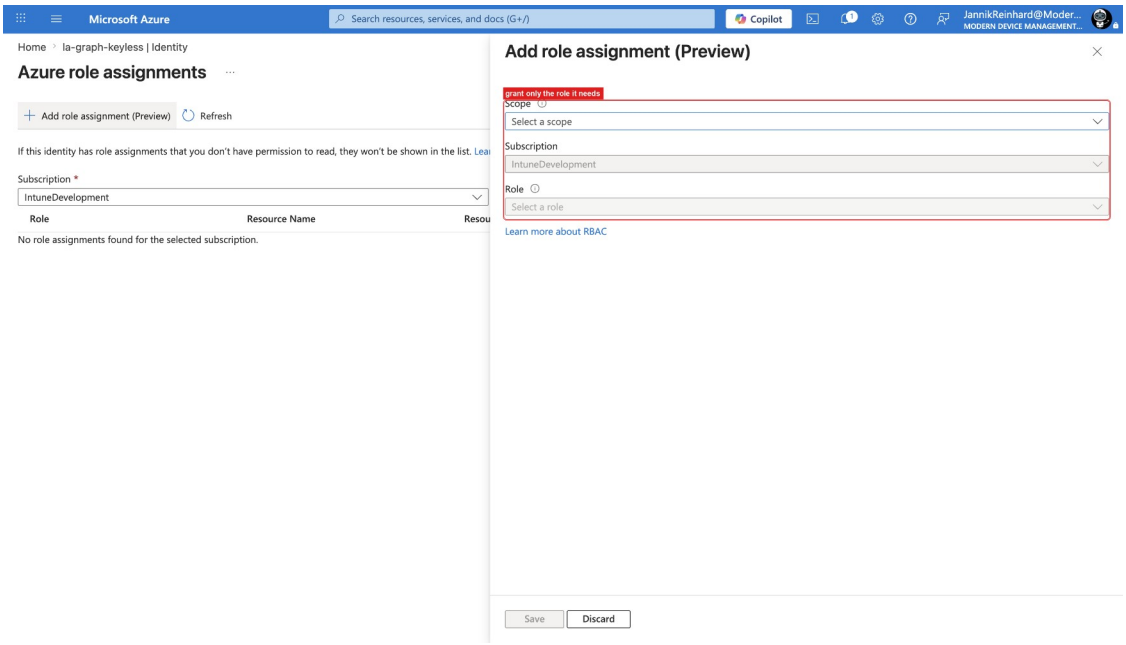

Figure 8.4: The other half of keyless — Add role assignment, where you grant the managed identity exactly the Azure role it needs (and nothing more) via RBAC, so the resource reaches what it must without any stored secret

Step 3 — Add the Recurrence trigger. Edit the workflow, add **Recurrence** as the trigger, and set the interval — once per day at, say, 07:00.

Step 4 — Add the HTTP action to call Graph. Add an **HTTP** action:

- **Method:** `GET`
- **URI:** `https://graph.microsoft.com/v1.0/deviceManagement/managedDeviceOverview`
- **Authentication:** choose **Managed identity**, select the **System-assigned managed identity**, and set the **Audience** to `https://graph.microsoft.com`.

That Authentication section is the whole point: the Logic App fetches a Graph token for its own managed identity at runtime. There is nowhere to put a secret, because there isn't one.

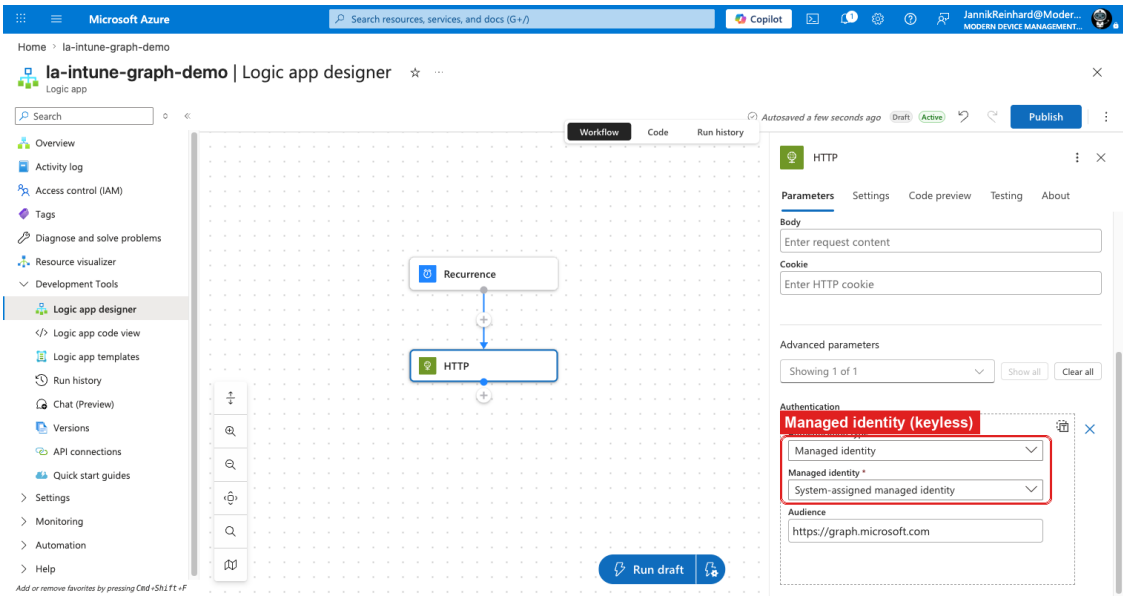

Figure 8.5: The keyless pattern in a Logic App: the HTTP action calls Microsoft Graph with Authentication type set to Managed identity (system-assigned) and the audience set to https://graph.microsoft.com — no client secret anywhere in the workflow

Step 5 — Parse the result and post to Teams. Add a **Parse JSON** action; use **Use sample payload to generate schema** and paste a sample response from Graph Explorer so the fields become selectable tokens. Then add **Post message in a chat or channel** (Microsoft Teams), sign in, pick the target channel, and build the message from the parsed tokens — for example the `enrolledDeviceCount` and per-platform counts from `managedDeviceOverview`.

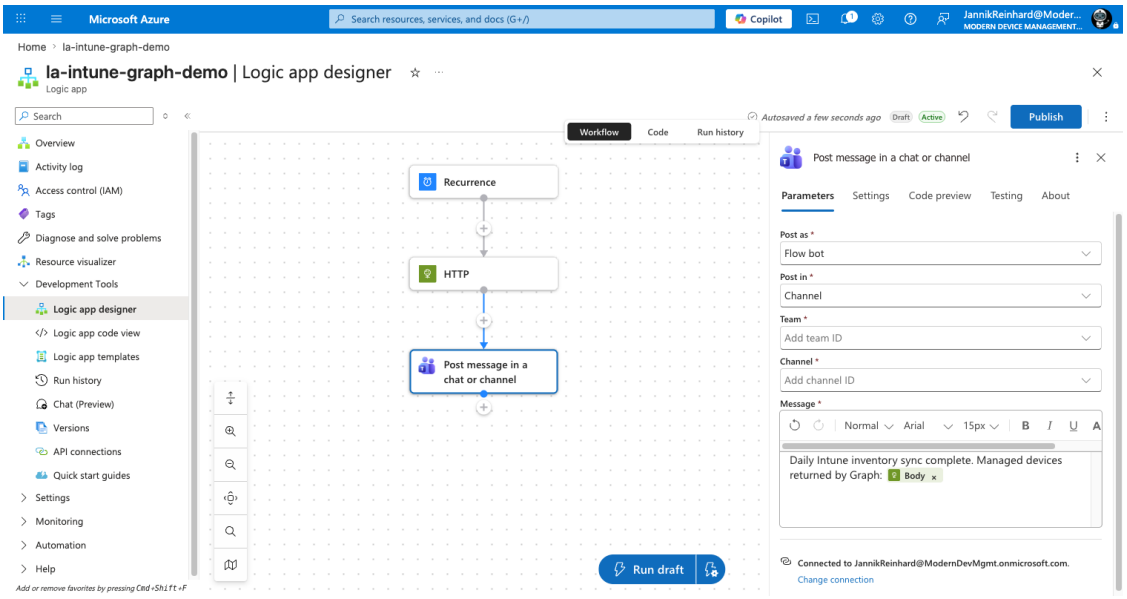

Figure 8.6: The Teams Post message in a chat or channel action, composed with a dynamic-content token — the green Body chip pulls the Graph response from the earlier HTTP step straight into the message, so the workflow posts live tenant data into a channel

Run it. Every morning a message like “Enrolled devices: 4,812 (Windows 3,900 / iOS 610 / Android 290 / macOS 12)” lands in your ops channel — with not a single credential stored anywhere.

Solution 2: Azure Automation Runbooks

When the logic outgrows a Logic App — real PowerShell, loops, HTML generation, multiple Graph calls — an **Azure Automation runbook** is the next step up. It runs scripts (PowerShell, Python) on a schedule on Azure-managed infrastructure, and like the Logic App it has a system-assigned managed identity, so it authenticates to Graph the keyless way. Our example emails a daily Windows 11 adoption report.

Step 1 — Create the Automation account. Search **Automation Accounts** in the portal and click **+ Create**. Pick a subscription, resource group, name, and region. On the **Advanced** tab, enable the **System assigned** managed identity, then **Review + create**.

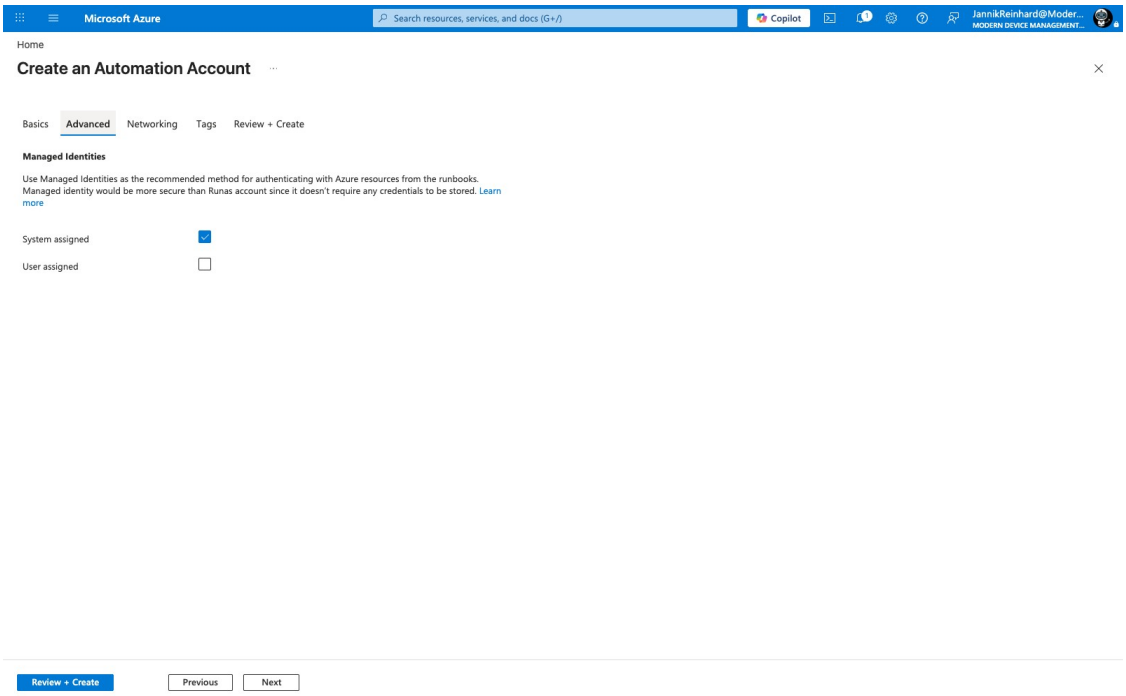

Figure 8.7: The Create Automation Account wizard, Advanced tab — turning on the system-assigned managed identity at creation, so the runbook authenticates to Graph with no stored secret

Step 2 — Grant Graph permissions. Copy the managed identity's Object ID (Automation account > **Account Settings > Identity**) and run the grant script twice — once for `DeviceManagementManagedDevices.Read.All` (to read devices) and once for `Mail.Send` (to send the report).

Step 3 — Import the Graph modules. A runbook has only the modules you give it. Under **Shared Resources > Modules**, add `Microsoft.Graph.Authentication` and `Microsoft.Graph.DeviceManagement` (and `Microsoft.Graph.Users.Actions` for `Send-MgUserMail`) from the gallery, and make sure the runbook's **Runtime version** is a current PowerShell 7.x. Importing only the sub-modules you need keeps cold-start times sane rather than importing the whole SDK.

Step 4 — Create the runbook. Under **Process Automation > Runbooks**, click **+ Create a runbook**, choose **PowerShell** and the matching runtime, and paste the script below. Note the

authentication line: `Connect-MgGraph -Identity`. That one flag replaces the entire old dance of `Connect-AzAccount`, `Get-AzAccessToken`, and passing a token into `Connect-MgGraph` — the SDK acquires the token for the managed identity itself.

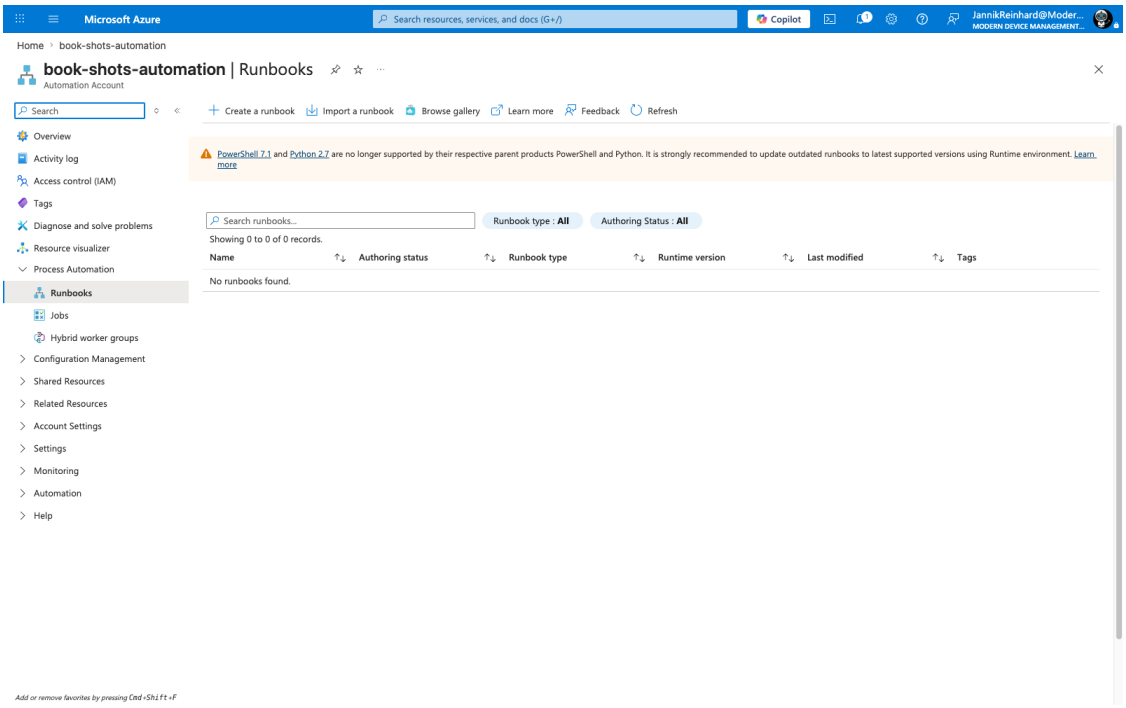

Figure 8.8: The Automation account Runbooks blade, where a PowerShell 7.4 runbook authenticates keyless with Connect-MgGraph -Identity

```powershell
# Runbook: Windows 11 adoption report
# Authenticates as the Automation account's system-assigned managed identity.

param(
    [string]$MailSender = "reports@contoso.com",
    [string]$MailTo     = "admin@contoso.com"
)

# Keyless connect — no secret, no token juggling.
Connect-MgGraph -Identity -NoWelcome

# Pull all Windows devices from Intune.
$devices = Get-MgDeviceManagementManagedDevice -All `
    -Filter "operatingSystem eq 'Windows'"

# Windows 11 is build 22000 and above.
$windows11 = $devices | Where-Object { [version]$_.OsVersion -ge [version]"10.0.22000" }

$total      = $devices.Count
$win11Count  = $windows11.Count
$win11Pct    = if ($total) { [math]::Round(($win11Count / $total) * 100, 1) } else { 0 }

# Build a simple HTML report.
$rows = ($devices | ForEach-Object {
    "<tr><td>$($_.DeviceName)</td><td>$($_.UserPrincipalName)</td>" +
    "<td>$($_.LastSyncDateTime)</td><td>$($_.OsVersion)</td></tr>"
}) -join "`n"

$html = @"
<html><head><style>
body{font-family:Segoe UI,Arial,sans-serif}
h1{color:#0078d4}
table{border-collapse:collapse;width:100%}
th{background:#0078d4;color:#fff;padding:8px;text-align:left}
td{border:1px solid #ddd;padding:8px}
tr:nth-child(even){background:#f2f2f2}
</style></head><body>
<h1>Windows 11 Adoption Report</h1>
<p><b>$win11Count of $total</b> Windows devices are on Windows 11 (<b>$win11Pct%</b>).</p>
<table>
<tr><th>Device</th><th>User</th><th>Last sync</th><th>OS version</th></tr>
$rows
</table>
</body></html>
"@

# Send the report with the Graph SDK cmdlet (needs Mail.Send).
$mail = @{
    Message = @{
        Subject      = "Windows 11 adoption report"
        Body         = @{ ContentType = "HTML"; Content = $html }
        ToRecipients = @( @{ EmailAddress = @{ Address = $MailTo } } )
    }
    SaveToSentItems = $false
}

Send-MgUserMail -UserId $MailSender -BodyParameter $mail
Write-Output "Report sent to $MailTo ($win11Pct% on Windows 11)."

Disconnect-MgGraph
```

Step 5 — Schedule it. Publish the runbook, then under **Schedules** link a daily schedule. Every morning the report arrives by email. If you would rather post it to Teams, swap `Send-MgUserMail` for an outgoing webhook or an Incoming Webhook connector — the data-gathering half of the script is unchanged.

A couple of modernization notes if you are porting an old runbook: `Send-MgUserMail` replaces the hand-built `sendMail` REST call, `Connect-MgGraph -Identity` replaces the `Get-AzAccessToken` pattern (and avoids the newer Az breaking change where the token is returned as a `SecureString`), and `Select-MgProfile` no longer exists — if you need beta endpoints, use the `Microsoft.Graph.Beta.*` cmdlets instead.

Solution 3: Intune Remediation Scripts

Everything so far ran in Azure and talked to Intune through Graph. Remediation scripts are different: they run *on the device*, in the Intune management extension, which lets you inspect and fix things Graph never sees. They live under **Devices > Scripts and remediations** (this was called "proactive remediations" in the first edition), and each one is a **pair** of scripts — a **detection** script that decides whether the device has the problem, and a **remediation** script that runs only when detection reports one.

You do **not** need Advanced Analytics to run basic remediations — the detection/remediation engine and its reporting are part of the base capability. Advanced Analytics adds the richer device timeline and anomaly detection on top, which is worth having but is not a prerequisite here.

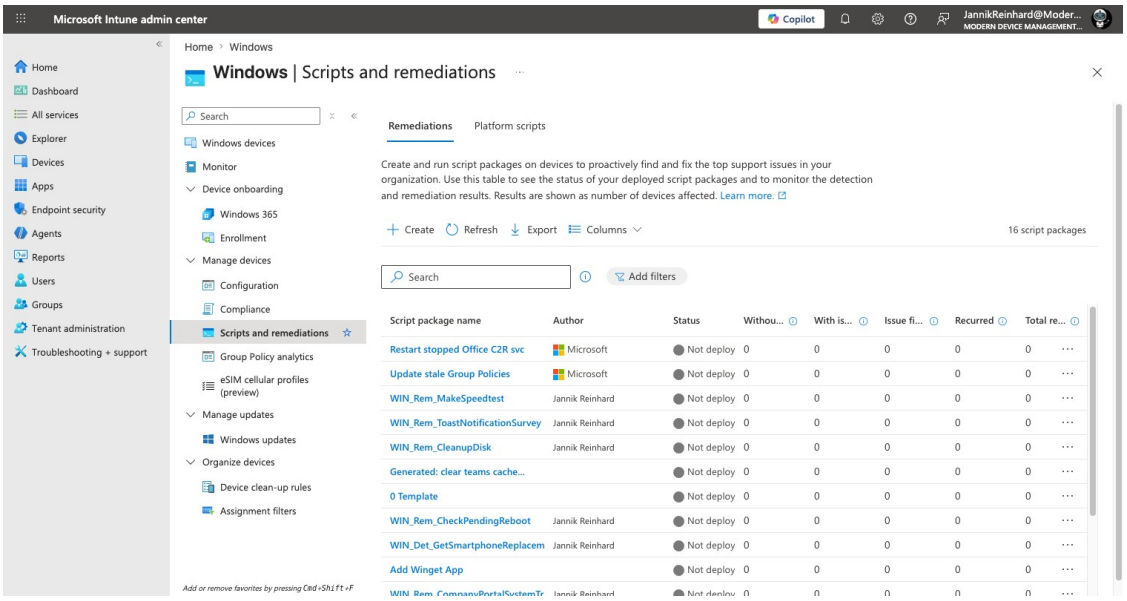

Figure 8.9: Devices > Scripts and remediations, where detection-and-remediation script pairs run on a schedule across the fleet

The detection/remediation contract. The engine reads two things from your script: its **exit code** and its **standard output**. Detection exits `0` to mean "compliant, do nothing" and non-zero to mean "issue found, run remediation." Whatever the detection script writes to stdout is captured and surfaced in the report — which is the mechanism you exploit to *collect data*, not just fix it.

Here is a compact, current pair. Detection checks whether the device has rebooted in the last seven days; remediation prompts the user to reboot. (The full toast-notification version — the one that pops a branded dialog and captures the user's choice — is in the book's companion repository; it is long, but the detection/remediation skeleton is exactly this.)

```powershell
# Detection script — exit 1 if uptime exceeds the threshold.
$maxDays = 7
$lastBoot = (Get-CimInstance Win32_OperatingSystem).LastBootUpTime
$uptime   = (Get-Date) - $lastBoot

if ($uptime.TotalDays -gt $maxDays) {
    Write-Output "Uptime $([math]::Round($uptime.TotalDays,1)) days exceeds $maxDays."
    exit 1   # triggers remediation
}
Write-Output "Uptime within policy."
exit 0
# Remediation script — notify the user that a reboot is due.
# Runs in the logged-on user's context so the toast is visible.
$title = "Contoso IT"
$text  = "Your device has been running for over a week. Please save your work and restart when convenient."

[Windows.UI.Notifications.ToastNotificationManager, Windows.UI.Notifications, ContentType = WindowsRuntime] | Out-Null
[Windows.Data.Xml.Dom.XmlDocument, Windows.Data.Xml.Dom.XmlDocument, ContentType = WindowsRuntime] |
Out-Null

[xml]$toast = @"
<toast scenario="reminder">
  <visual><binding template="ToastGeneric">
    <text>$title</text>
    <text>$text</text>
  </binding></visual>
</toast>
"@

$xml = New-Object Windows.Data.Xml.Dom.XmlDocument
$xml.LoadXml($toast.OuterXml)
[Windows.UI.Notifications.ToastNotificationManager]::CreateToastNotifier($title).Show($xml)
exit 0
```

Deploy it. In **Scripts and remediations > Remediations**, click **+ Create**, give it a name, and on the next page upload the **detection** and **remediation** scripts. Set **Run this script using the logged-on credentials** to **Yes** (a toast has to be shown in the user's session), leave the architecture as needed, and — for a Windows PowerShell 5.1 script that emits Unicode — set **Enforce script signature check** and the 32/64-bit option appropriately. Assign it to a device or user group and choose a schedule (Once, Hourly, or Daily). The results, including anything your detection script wrote to stdout, show up in the remediation's report and on each device's timeline.

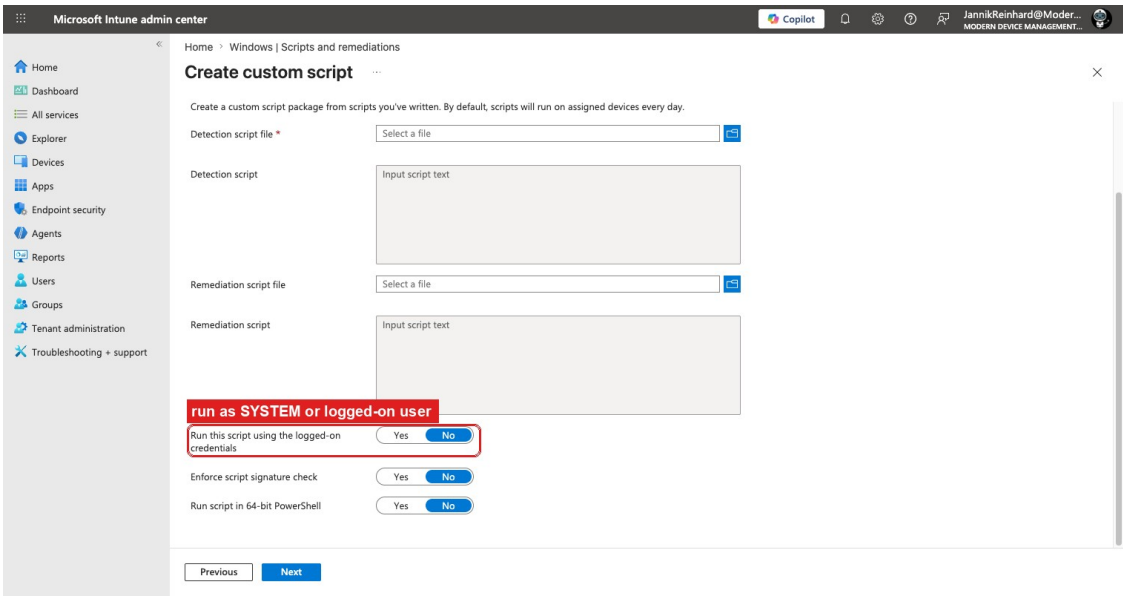

Figure 8.10: The Create custom script (remediation) wizard, Settings step — you upload a detection script and a remediation script, and decide whether the pair runs as SYSTEM or with the logged-on user's credentials, along with signature-check and 64-bit options

A note on collecting the data centrally. In the first edition this section had the remediation script write straight into a Log Analytics workspace using the workspace ID and a shared key (the HTTP Data Collector API at `*.ods.opinsights.azure.com/api/logs`). That still runs today, but **the HTTP Data Collector API is deprecated and on a retirement path**, and it was never a great idea: it bakes a workspace key into a script that ships to every device, leaving a credential footprint on the endpoint. The modern replacement is the **Logs Ingestion API**, which sends data through a **Data Collection Endpoint** governed by a **Data Collection Rule** and authenticates with Entra ID rather than a shared key. But you do not want every device holding *any* ingestion credential. The clean answer is Solution 4: have the device call an Azure Function, and let the Function — with its managed identity — do the authenticated write. The device holds nothing but a function URL.

Solution 4: Azure Functions

An **Azure Function** is serverless code triggered by an HTTP request, a timer, a queue, or another event. In our context it is the secure broker between the device and your data store: the

remediation script POSTs its inventory to the function's HTTPS endpoint, and the function — authenticating with its **managed identity** — validates and writes that data to Log Analytics (via the Logs Ingestion API) or wherever you keep it. The device never holds an ingestion credential; it only knows a URL. This is the most robust of the four patterns and the one I use for anything collecting from the whole fleet.

The reference implementation for this is the community project **Intune Enhanced Inventory** by Sandy Zeng and Jan Ketil Skanke (MSEndpointMgr) — a packaged combination of a remediation script, an Azure Function, and a Log Analytics workspace that collects rich client inventory the secure way. It ships a **Deploy to Azure** button that provisions the Function app, workspace, and supporting resources in one click, which is the fastest way to see the pattern working end to end.

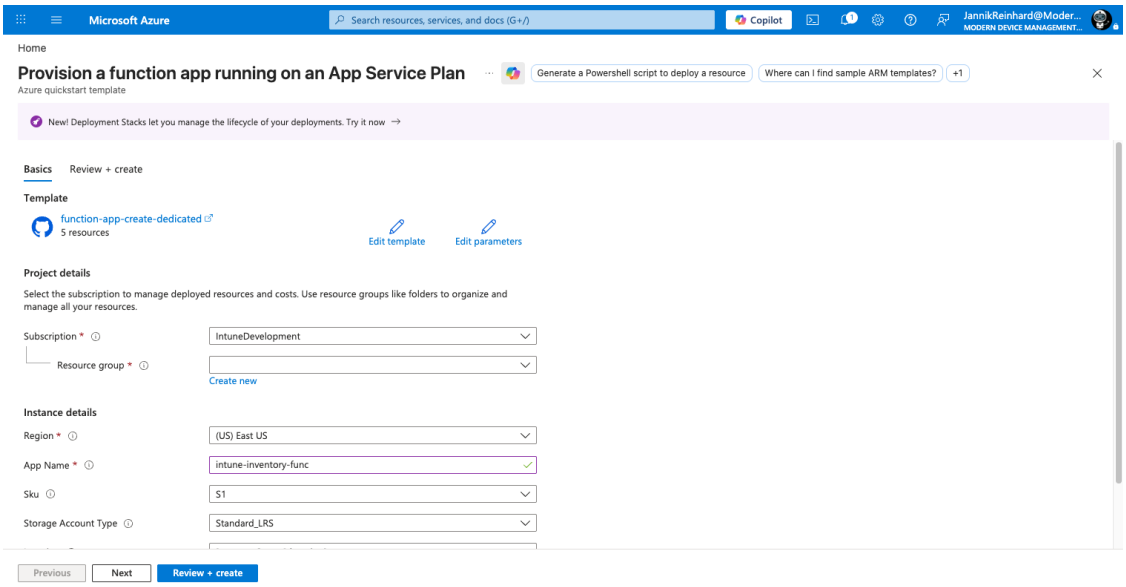

Figure 8.11: A repo's Deploy to Azure button lands you here — the portal's Custom deployment page for an ARM/Bicep template, where you pick the subscription and resource group and fill the template's parameters (function app name, region, SKU) before it provisions the whole solution in one click

After deployment, turn on the **Function app's system-assigned managed identity** (**Settings > Identity**) and grant it whatever Graph or Monitor permissions it needs with the same grant

script from the auth section. The repository includes an `Add-MSIGraphPermissions.ps1` helper that does exactly this — the same `New-MgServicePrincipalAppRoleAssignment` pattern I showed earlier, just wrapped in a loop over multiple scopes.

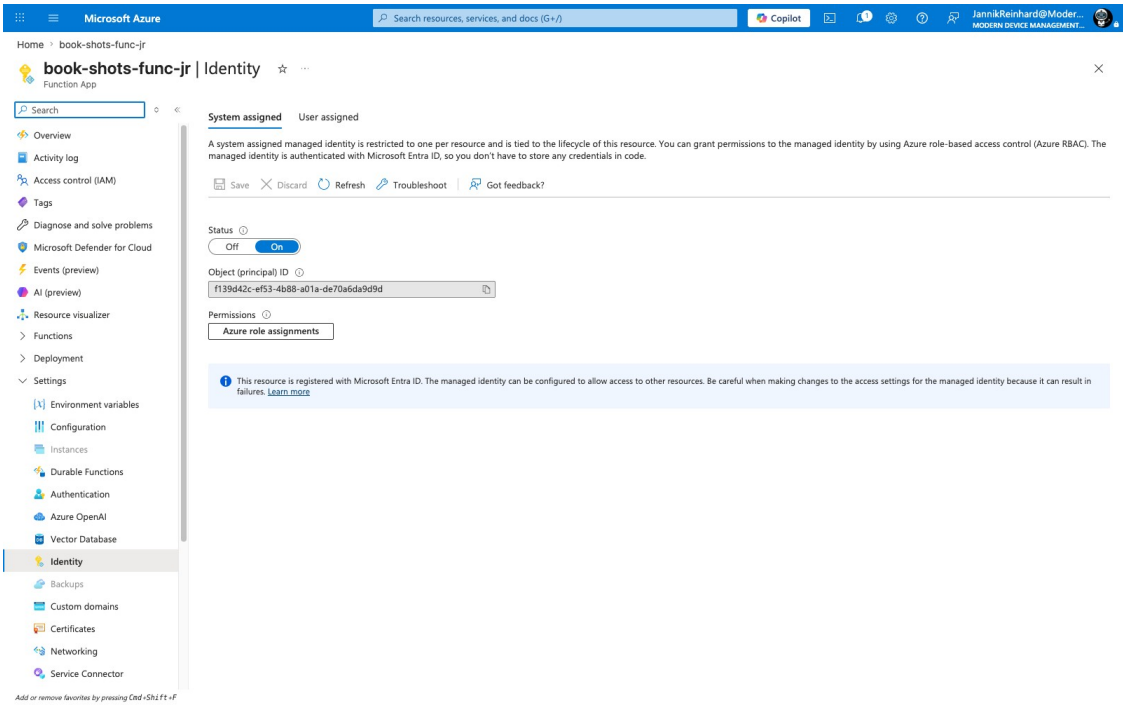

Figure 8.12: The Function app's Identity blade — the system-assigned identity that the function uses to reach Microsoft Graph and Azure Monitor without a stored secret

The device side is a single remediation script that gathers the inventory and POSTs it to the function. You paste the function's URL — retrieved from the function's **Get function URL** button, which includes a pre-authenticating function key — into that script, and deploy it as a detection script through Scripts and remediations exactly as in Solution 3. Run it in the system context (64-bit) unless you specifically need per-user data.

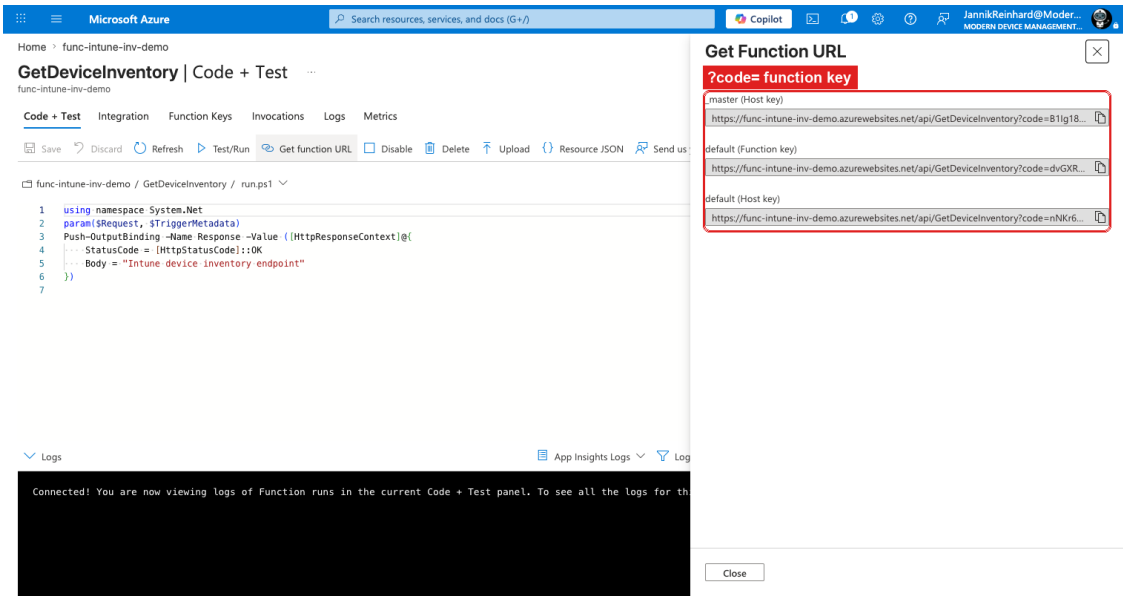

Figure 8.13: An HTTP-triggered Azure Function's Code + Test view with the Get function URL flyout open — the invoke URL carries the function key as a ?code= parameter, which is the pre-authenticated URL you paste into the caller (and exactly why you protect it like a secret)

The shape of the function itself is straightforward — an HTTP trigger that reads the posted JSON, connects with its managed identity, and writes onward. In PowerShell it starts like this:

```
using namespace System.Net
param($Request, $TriggerMetadata)

# Inventory posted by the device's remediation script.
$inventory = $Request.Body

# Authenticate as the Function app's managed identity — no secret.
Connect-MgGraph -Identity -NoWelcome

# ... validate $inventory, enrich it from Graph if needed, then write it
#     to Log Analytics via the Logs Ingestion API (Data Collection Endpoint) ...

Push-OutputBinding -Name Response -Value ([HttpResponseContext]@{
    StatusCode = [HttpStatusCode]::OK
    Body       = "Inventory received."
})
```

The simpler, less secure variant is to skip the function and have the remediation script write directly into Log Analytics itself. It is fewer moving parts, but — as noted in Solution 3 — it puts an ingestion credential on every device and rides the deprecated Data Collector API. I only

reach for it in a lab or a tightly scoped, short-lived collection; for anything that lives in production, the function-brokered pattern is worth the extra half hour to set up.

And if it runs outside Azure: the same function logic in a GitHub Actions workflow or a self-hosted job would use **workload identity federation** instead of a managed identity — a federated credential trusting the platform's OIDC issuer, `azure/login` with no client secret, and the identical Graph app-role grants. The principle holds across all four solutions: the credential is never a secret you store; it is an identity the platform proves at runtime.

Conclusion

Four patterns, one idea: get the data out of Intune, do something useful with it, and put the result where it matters — on a schedule, running itself. Logic Apps for low-code glue, Automation runbooks for real scripting, remediation scripts for anything that has to happen on the device, and Functions for secure, fleet-scale collection. Which one you pick is a question of where the logic lives and how much of it there is.

The thread through all four is the authentication model, and it is the thing I most want you to take from this chapter. The first edition taught these solutions with app registrations and client secrets because that is what we all did in 2023. In 2026 that is a liability. Turn on a managed identity, grant it the least Graph permission it needs with `New-MgServicePrincipalAppRoleAssignment`, connect with `Connect-MgGraph -Identity`, and there is simply no secret left to leak. Build your solutions that way from the start, and you spend your time on the interesting part — what the automation *does* — instead of babysitting credentials that were never supposed to be your problem.

These four patterns each build one solution at a time. The next chapter asks the bigger question that shows up the moment you have more than a handful of them, or more than one tenant to run: how do you manage *all* of it — every policy, profile, script, and solution — as code, in version control, deployed and checked by a pipeline instead of by hand? That is where automation grows up into **DevOps for Intune**, and it is where Chapter 9 goes next.

Chapter 9: DevOps for Intune — Infrastructure as Code, CI/CD, and Backup

Chapter 8 was about building automations that *run against* Intune — Logic Apps, runbooks, remediation scripts, and Functions that pull data out and push actions back in, all authenticated keyless with a managed identity or workload identity federation. This chapter is about something one level up: managing Intune's own *configuration* the way you would manage infrastructure. Not the automations that touch the tenant, but the compliance policies, configuration profiles, app assignments, and Autopilot profiles that *are* the tenant. The idea is simple and, once you have lived it, hard to give up: your tenant configuration lives in git, changes go through a pull request, and a pipeline deploys them — no more clicking around the admin center and hoping you remember what you changed.

I want to be honest up front about why this matters, because "DevOps for Intune" can sound like ceremony for its own sake. It is not. Five concrete problems push almost everyone here eventually.

Configuration drift. You make a change in the portal to fix something at 4 p.m. on a Friday, and eighteen months later nobody — including you — remembers it exists or why. Multiply that by every admin who has ever had access, and your tenant becomes an archaeology site. When configuration is code, the current state *is* the repository, and nothing is a mystery because everything is in the history.

Audit trail. "Who changed this policy, when, and why?" is a question the admin center answers poorly and git answers perfectly. Every change is a commit with an author, a timestamp, and — if you insist on it — a linked pull request explaining the reasoning.

Peer review. A conditional access change or a compliance policy tweak can lock thousands of people out of their laptops. That change deserves a second pair of eyes *before* it lands, not an incident review afterwards. A pull request is where that review happens.

Rollback. When a change does cause an incident, “revert the commit and let the pipeline redeploy” is a very different Tuesday from “reconstruct the old settings from memory and screenshots.”

Replicating a known-good baseline. This is the one that sells it to management. If you run a dev tenant and a production tenant — or if you are an MSP running fifty customer tenants — config-as-code lets you define a baseline once and deploy it everywhere, consistently, with the differences between tenants expressed as deliberate, reviewable overrides rather than accidental divergence.

What “Infrastructure as Code” Even Means for Intune

Before we pick tools, I have to clear up a confusion that trips up almost everyone coming from the Azure side, because getting it wrong sends you down a dead end.

When people say “infrastructure as code” in Azure, they usually mean **Bicep** or **Terraform** — declarative templates that create and manage Azure Resource Manager (ARM) resources: virtual machines, storage accounts, key vaults, networks. That model is real and excellent, and it absolutely applies to the *surrounding* Azure resources your Intune automations depend on — the Function app, the Log Analytics workspace, the storage account, the managed identity from Chapter 8. Those are ARM resources, and you should manage them in Bicep or Terraform like any other infrastructure.

But **Intune policy is not an ARM resource.** A compliance policy, a Settings Catalog profile, an app assignment — none of these live in Azure Resource Manager. They live behind Microsoft Graph, as JSON objects under `deviceManagement`. There is no `Microsoft.Intune/compliancePolicies` ARM type you can `terraform apply`. So when you hear “IaC for Intune,” mentally translate it to **Graph-configuration-as-code**: exporting those Graph JSON objects to files, versioning the files, and deploying them back through Graph. It is the same *philosophy* as Bicep — declarative, versioned, pipeline-deployed — but a completely different *mechanism*.

That distinction defines the whole toolchain:

- **Bicep / Terraform** (with the `azurerm` and `azapi` providers) manage the **Azure resources around** Intune — the plumbing your Chapter 8 solutions run on.
- **IntuneCD, Microsoft365DSC, the new native Graph configuration APIs, or your own Graph SDK code** manage the **Intune configuration itself**.

Do not try to force Intune policy into Terraform's `azurerm` provider; there is a community Terraform provider that talks to Graph (`microsoft/microsoft365`, formerly the Kaymera/`Microsoft365DSC`-adjacent efforts), and it is maturing, but for most teams the JSON-and-Graph tools below are the pragmatic path. I will show you all of it and be clear about where each one fits.

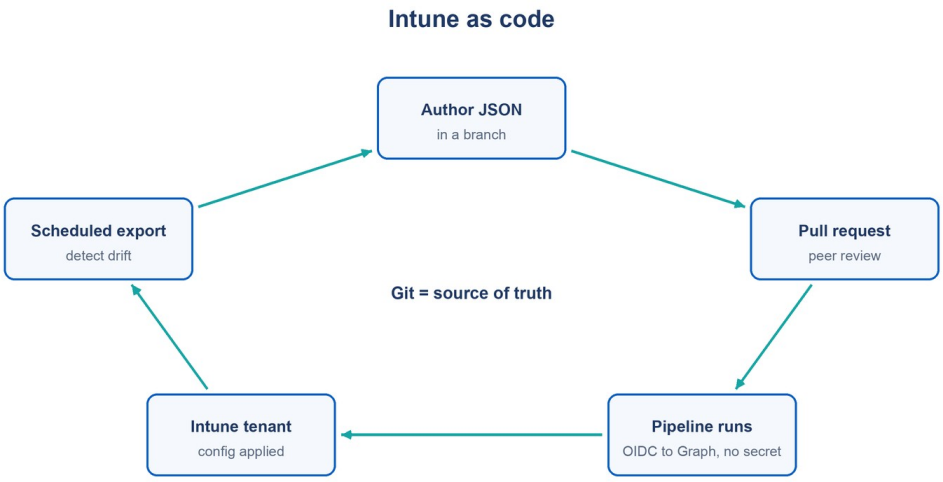

Figure 9.1: The config-as-code loop for Intune — author, review, deploy, and detect drift, all anchored in git as the single source of truth.

The Tooling Landscape, Honestly Compared

There is no single winner here. I run different tools for different jobs, and I would rather you understand the trade-offs than hand you one recommendation that fits your situation badly.

There are four realistic options for the Intune-configuration side, plus the Bicep/Terraform layer for the Azure side.

IntuneCD — backup, restore, and documentation as JSON in git

IntuneCD ("Intune Continuous Delivery," by Tobias Almén — `almenscorner/IntuneCD`) is the tool I reach for first, and the one most teams should start with. It is a Python package that does three things extremely well:

1. **Backup** — export a whole tenant's Intune configuration to a folder tree of JSON (or YAML) files, one file per object, organized by type. This is your source of truth in git.
2. **Update** — take that folder tree and push it into a target tenant, creating what is missing and updating what has drifted. This is the deploy half.
3. **Document** — generate human-readable Markdown documentation of the entire configuration, which is worth the price of admission on its own for handing an "as-built" document to auditors or a new team member.

What makes IntuneCD a natural fit for git is that the backup output is *diffable*. Each policy is its own JSON file, so a change to one compliance policy is a one-file diff in a pull request that a human can actually read. It covers the breadth of Intune — compliance, configuration profiles, Settings Catalog, apps and assignments, Autopilot, enrollment, scripts, filters, scope tags, and more — and it is explicitly designed around the dev-tenant-to-prod-tenant promotion flow this chapter is about. It is actively maintained, it ships as a container image and a `pip` package, and it has first-class documentation for both GitHub Actions and Azure DevOps pipelines.

Its limitation is the flip side of its strength: it is opinionated about *its own* JSON format and folder layout. You are managing Intune the IntuneCD way, which is fine — but it is a tool-specific representation, not a Microsoft-blessed schema.

Microsoft365DSC — desired-state configuration across all of M365

Microsoft365DSC takes a different philosophy. It is built on PowerShell **Desired State Configuration**, and its scope is not just Intune but *all* of Microsoft 365 — Entra ID, Exchange Online, SharePoint, Teams, Purview, Defender, and Intune together in one configuration language. You *extract* a tenant's configuration into a DSC configuration (a `.ps1`/`.psd1`), you *deploy* that configuration to bring a tenant into the desired state, and you *monitor* for drift with a scheduled compliance check that tells you — and can act on — anything that has diverged from the declared state.

If your remit is broader than Intune — if you own conditional access, Exchange settings, and Teams policy as well as devices — Microsoft365DSC is genuinely compelling because it unifies all of it under one model. As of early 2026 it covers over a thousand Intune Settings Catalog categories, so its Intune fidelity is high and keeps improving. The trade-off is that DSC is a heavier conceptual model than "a folder of JSON files," the extracted configurations are large and can be harder to diff cleanly in a PR, and you are committing to the PowerShell DSC way of thinking. For a pure-Intune team I lean IntuneCD; for a team that owns the whole M365 estate as code, Microsoft365DSC earns its complexity.

The new native option: Tenant Configuration Management APIs

Worth knowing about, and worth watching, is Microsoft's own entry into this space. On **27 January 2026** Microsoft shipped the **public preview** of the **Tenant Configuration Management APIs** (you will also see it called **Unified Tenant Configuration Management, UTCM**) in Microsoft Graph. This is Microsoft building config-as-code *into the platform* rather than leaving it to the community.

In preview it does two of the four things you want:

- **Snapshot (baseline)** — an asynchronous job that captures your tenant's configuration for chosen resource types into a JSON template. This is a native backup.

- **Drift monitoring** — configuration *monitors* that run on a schedule and flag when the live tenant diverges from a baseline.

The other two — **deploying** configuration from a template and **auto-remediating** drift — were on the roadmap at the time of writing but not yet shipped. It spans Intune, Entra, Exchange Online, Defender, Purview, and Teams, with 300-plus resource types in the published schema. Be honest with yourself about the preview limits before you build on it: snapshots are capped at roughly 20,000 resources per tenant per month and are auto-deleted after seven days; monitors run on a fixed six-hour cadence you cannot change; you get at most 30 monitors, and monitoring is capped around 800 settings per day.

My take for mid-2026: it is the strategic direction, and in a year or two it may well be *the* answer — a native, supported, no-third-party-tool way to do exactly what this chapter describes. Today it is a preview with real limits and no deploy capability, so I would use it to *complement* IntuneCD (native drift monitoring is a nice signal) rather than to replace a working pipeline. Watch it.

Rolling your own with the Microsoft.Graph SDK

The fourth option is to write it yourself, and sometimes that is the right call — when you need to manage a narrow, specific slice of configuration exactly your way, or when you want to embed config management inside a larger automation. You already have every building block from Chapter 8: `Connect-MgGraph`, `Invoke-MgGraphRequest`, keyless auth, and the knowledge that anything the portal does is a Graph call you can make. Exporting a category of policy is a `GET` that you serialize to JSON; deploying it is a `POST` or `PATCH` of that JSON back. Here is the skeleton of a hand-rolled backup for compliance policies, keyless as always:

```
# Minimal roll-your-own backup: export all compliance policies to JSON files.
# Runs in a pipeline with a workload-identity-federated login (no secret).
Connect-MgGraph -Identity -NoWelcome   # or the OIDC flow shown later

$outDir = "./backup/compliancePolicies"
New-Item -ItemType Directory -Force -Path $outDir | Out-Null

$policies = Invoke-MgGraphRequest -Method GET `
   -Uri "https://graph.microsoft.com/v1.0/deviceManagement/deviceCompliancePolicies"

foreach ($p in $policies.value) {
   # Strip volatile fields that would create noise in every diff.
   $p.Remove('lastModifiedDateTime')
   $p.Remove('createdDateTime')
   $p.Remove('version')
   $file = Join-Path $outDir ("{0}.json" -f ($p.displayName -replace '[^\w\-]', '_'))
   $p | ConvertTo-Json -Depth 20 | Set-Content -Path $file -Encoding utf8
}
```

The catch — and the reason most people end up on IntuneCD instead — is everything that skeleton *doesn't* handle: assignments, dependencies between objects (a policy that references a filter that references a scope tag), the difference between create and update, ID remapping when you move between tenants (the object IDs are tenant-specific), and the dozen other object types. IntuneCD is, in effect, several thousand lines of exactly this, already written and tested. Roll your own for a surgical need; use a tool for the whole tenant.

Bicep and Terraform for the Azure side

Finally, do not forget the layer beneath. The Function app, storage account, Log Analytics workspace, Data Collection Endpoint, and user-assigned managed identity that your Chapter 8 automations run on *are* ARM resources, and they belong in Bicep or Terraform, deployed by the same pipeline that manages your Intune config. Keeping both in one repository means a single pull request can, say, stand up a new Function app *and* wire up the remediation config that feeds it. Terraform's `azurerm` provider (and `azapi` for the newest resources) or a Bicep module — either is fine; pick the one your organization already uses. The point of this chapter is the pattern, and the pattern is identical whether the thing being deployed is a Bicep template or an IntuneCD backup folder.

Authentication: Workload Identity Federation, No Secrets

Everything in this chapter runs from a pipeline that lives *outside* Azure — GitHub Actions or Azure DevOps — which means the keyless model we use is **workload identity federation (OIDC)**, exactly as introduced in Chapter 8. This is the single most important thing to get right, so I am going to show the full setup rather than gesture at it.

The mechanism is a trust relationship. You create an app registration (or a user-assigned managed identity) in Entra ID, and you attach to it a **federated credential** that says, in effect: "I trust tokens issued by GitHub's OIDC provider, but *only* for this specific repository, on this specific branch or environment." At runtime, GitHub mints a short-lived JSON Web Token describing the workflow — which repo, which branch, which environment — and hands it to the `azure/login` action. Entra ID validates that token's issuer, audience, and subject against the federated credential you configured, and if they match, it issues a real Azure/Graph access token. No secret is ever stored, in GitHub or anywhere else. The credential is a *proof of identity the platform generates fresh each run*, not a password sitting in a vault waiting to leak.

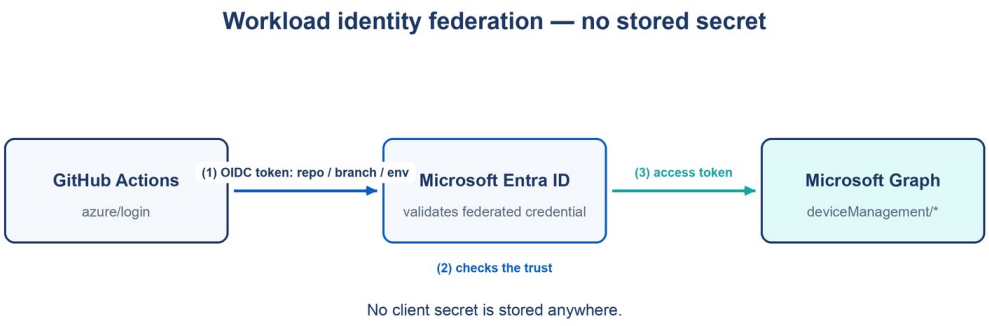

Figure 9.2: The keyless trust chain — GitHub proves the workflow's identity with a short-lived token, Entra validates it against the federated credential, and Graph gets called with no stored secret.

Setting up the federated credential

You do this once, interactively, as an admin. Create the app registration, then add a federated credential scoped to your repo and branch. Here it is with the Azure CLI:

```
# 1. Create the app registration for the pipeline identity.
appId=$(az ad app create --display-name "intune-config-pipeline" \
      --query appId -o tsv)

# 2. Create a service principal for it.
az ad sp create --id "$appId"

# 3. Add a federated credential trusting GitHub Actions
#    for the main branch of this repo (used for deploys).
az ad app federated-credential create --id "$appId" --parameters '{
  "name": "github-main",
  "issuer": "https://token.actions.githubusercontent.com",
  "subject": "repo:contoso/intune-config:ref:refs/heads/main",
  "audiences": ["api://AzureADTokenExchange"]
}'
```

The `subject` is the security boundary and the part to get exactly right. `repo:contoso/intune-config:ref:refs/heads/main` means only workflows running on the `main` branch of `contoso/intune-config` can use this identity. If you gate deploys behind a GitHub **environment** (recommended — more on that under governance), use an environment-scoped subject instead, so only jobs targeting that protected environment can deploy:

```
{
  "name": "github-prod-environment",
  "issuer": "https://token.actions.githubusercontent.com",
  "subject": "repo:contoso/intune-config:environment:production",
  "audiences": ["api://AzureADTokenExchange"]
}
```

Granting the pipeline identity least-privilege Graph permissions

The app registration starts with no permissions, exactly like the managed identity in Chapter 8, and you grant it Graph application permissions the same way — with `New-MgServicePrincipalAppRoleAssignment`. The permissions differ by job: a **drift-detection** pipeline that only ever reads needs `.Read.All` scopes, while a **deploy** pipeline needs `.ReadWrite.All`. Grant them separately, and if you can, use *two* identities — a read-only one

for the scheduled export and a read-write one gated behind your protected production environment. For Intune configuration management the scopes you will typically need are:

```
# Grant the pipeline's service principal the Intune config scopes it needs.
# Run once, interactively, as an admin who can assign app roles.
Connect-MgGraph -Scopes "Application.Read.All","AppRoleAssignment.ReadWrite.All" -NoWelcome

$spId   = (Get-MgServicePrincipal -Filter "displayName eq 'intune-config-pipeline'").Id
$graph  = Get-MgServicePrincipal -Filter "appId eq '00000003-0000-0000-c000-000000000000'"

# Read-only for a drift pipeline; add the ReadWrite equivalents for a deploy pipeline.
$scopes = @(
   "DeviceManagementConfiguration.ReadWrite.All",
   "DeviceManagementApps.ReadWrite.All",
   "DeviceManagementServiceConfig.ReadWrite.All",
   "DeviceManagementManagedDevices.Read.All",
   "DeviceManagementRBAC.ReadWrite.All"
)

foreach ($scope in $scopes) {
   $role = $graph.AppRoles | Where-Object {
      $_.Value -eq $scope -and $_.AllowedMemberTypes -contains "Application"
   }
   New-MgServicePrincipalAppRoleAssignment `
      -ServicePrincipalId $spId -PrincipalId $spId `
      -ResourceId $graph.Id -AppRoleId $role.Id
}
Disconnect-MgGraph
```

Grant the least the job needs and nothing more — this identity can rewrite your entire device-management configuration, so it deserves the same scrutiny you would give a domain admin account.

A Worked End-to-End Pipeline

Now let us assemble the whole thing in GitHub Actions. The design is two workflows against one repository:

1. A **scheduled export** workflow that reads the live tenant and, if anything changed, opens a pull request with the diff. This is how portal changes get *captured* back into git, and it doubles as drift detection.
2. A **deploy** workflow that, when a pull request is merged to `main`, pushes the committed configuration into the target tenant.

I will use IntuneCD as the engine because it does the heavy lifting, but the pipeline *shape* is identical whichever tool you choose.

The repository layout

```
intune-config/
├── .github/workflows/
│   ├── export.yml        # scheduled: tenant -> PR
│   └── deploy.yml        # on merge to main: repo -> tenant
├── backup/               # IntuneCD JSON output, the source of truth
│   ├── Compliance Policies/
│   ├── Device Configurations/
│   ├── Settings Catalog/
│   ├── Applications/
│   └── ...
└── README.md
```

Deploy on merge

The deploy workflow runs on a push to `main` (i.e. after a PR merges), authenticates with OIDC — note the `permissions` block and the *absence* of any `client-secret` — and runs IntuneCD's update command against the target tenant.

```yaml
# .github/workflows/deploy.yml
name: Deploy Intune configuration

on:
  push:
    branches: [main]
    paths: ['backup/**']

permissions:
  id-token: write      # REQUIRED for OIDC — lets the job mint the GitHub token
  contents: read

jobs:
  deploy:
    runs-on: ubuntu-latest
    environment: production        # protected environment: approval gate lives here
    steps:
      - uses: actions/checkout@v4

      - name: Azure login (workload identity federation, no secret)
        uses: azure/login@v2
        with:
          client-id: ${{ vars.AZURE_CLIENT_ID }}
          tenant-id: ${{ vars.AZURE_TENANT_ID }}
          allow-no-subscriptions: true

      - name: Set up Python
        uses: actions/setup-python@v5
        with:
          python-version: '3.12'

      - name: Install IntuneCD
        run: pip install IntuneCD

      - name: Deploy configuration to the target tenant
        env:
          # IntuneCD reads the token from the azure/login session — no secret.
          TENANT_NAME: ${{ vars.TARGET_TENANT }}
        run: |
          IntuneCD-startupdate \
            --path "./backup" \
            --interactiveauth false \
            --report true
```

The three things that make this keyless and safe are worth calling out explicitly. `id-token: write` is what lets the job obtain the GitHub OIDC token in the first place — without it, `azure/login` has nothing to present. `azure/login@v2` with only `client-id` and `tenant-id` (and no `client-secret`) triggers the federated flow. And `environment: production` ties the job to a GitHub environment where you have configured a required reviewer, so the deploy pauses for human approval before it touches the tenant.

Export on a schedule, open a PR

The export workflow is the mirror image. It runs on a timer, backs the live tenant up into the `backup/` folder with a *read-only* identity, and if that produced any change, opens a pull request. Reviewing and merging that PR is how a portal change becomes a committed, reviewed part of your baseline.

```yaml
# .github/workflows/export.yml
name: Export Intune configuration

on:
  schedule:
    - cron: '0 6 * * *'    # every day at 06:00 UTC
  workflow_dispatch: {}      # allow manual runs

permissions:
  id-token: write
  contents: write           # to commit the export
  pull-requests: write      # to open the PR

jobs:
  export:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Azure login (read-only pipeline identity, OIDC)
        uses: azure/login@v2
        with:
          client-id: ${{ vars.AZURE_READONLY_CLIENT_ID }}
          tenant-id: ${{ vars.AZURE_TENANT_ID }}
          allow-no-subscriptions: true

      - uses: actions/setup-python@v5
        with:
          python-version: '3.12'

      - run: pip install IntuneCD

      - name: Back up the live tenant into the repo
        run: |
          IntuneCD-startbackup \
            --path "./backup" \
            --interactiveauth false \
            --exclude-assignments false

      - name: Open a PR if the tenant diverged from the baseline
        uses: peter-evans/create-pull-request@v6
        with:
          branch: drift/auto-export
          title: "Tenant configuration drift detected"
          commit-message: "chore: capture live tenant configuration"
          body: |
            The scheduled export found differences between the live tenant
            and the committed baseline. Review the diff below: approve to
            adopt the change into the baseline, or close and let the deploy
            pipeline revert the tenant to the committed state.
```

That is the entire loop. An admin authors a change (either by editing JSON in a branch, or by making it in the portal and letting the export catch it), a reviewer reads the diff in a pull request, and on merge the deploy pipeline applies it — all keyless, all audited, all reversible.

Drift Detection

Look again at that export workflow, because it is doing double duty. Its *primary* purpose is capturing intended changes back into git, but its *side effect* is drift detection: if someone makes an out-of-band change in the portal — a well-meaning fix, a mistake, or something more concerning — the next scheduled export sees the live tenant no longer matches the committed baseline, and it opens a PR showing exactly what diverged. Drift stops being invisible.

You then get to make a *decision* about every drift, which is the whole point. When that PR appears, you either **adopt** the change (merge the PR — the portal change was legitimate and now it is properly recorded and reviewed) or you **reject** it (close the PR and let the deploy pipeline push the committed baseline back over the drift, reverting the tenant to its known-good state). Either way the divergence surfaced within a day and a human decided, deliberately, rather than the change silently becoming permanent.

For something louder than a quiet PR, wire an alert onto the export job so a divergence pings a Teams channel. A minimal step at the end of the export workflow:

```
- name: Alert on drift
  if: steps.cpr.outputs.pull-request-number != ''
  run: |
    curl -H 'Content-Type: application/json' \
      -d '{"text":"⚠️ Intune drift detected — review PR #${{ steps.cpr.outputs.pull-request-number }}"}' \
      "${{ secrets.TEAMS_WEBHOOK }}"
```

And this is exactly where the new native **UTCM configuration monitors** complement the tool-based approach: if you want a second, Microsoft-native signal, point a UTCM monitor at your production tenant and let it flag drift on its own six-hour cadence, independent of your export pipeline. Belt and braces. Just remember its preview limits — it is a signal, not yet the enforcement mechanism.

Backup, Restore, and Disaster Recovery

Everything above gives you disaster recovery almost for free, but it is worth stating plainly because “we have IaC” and “we can actually recover” are not the same claim until you have tested the second one.

Your **backup** is the scheduled export, committed to git. Because it runs daily and git keeps history, you do not have one backup — you have every daily state going back as far as your repository does. Restoring the tenant to how it looked three weeks ago is `git checkout` of that commit followed by a deploy run. That is a genuinely powerful position to be in, and it costs you nothing beyond the pipeline you already built.

Your **restore** is the deploy pipeline pointed at the same or a rebuilt tenant. This is the scenario that justifies the whole exercise to a risk officer: if a tenant is compromised, corrupted, or has to be rebuilt, you are not reconstructing hundreds of policies from documentation and memory — you run the deploy pipeline against the fresh tenant and it recreates your entire device-management configuration from the JSON in git.

Two honest caveats so you plan for reality rather than the demo:

- **IDs and references do not travel.** Object IDs are tenant-specific, and so are the group IDs that assignments target. A good tool (IntuneCD does this) remaps references by name or by a mapping you supply, but a full cross-tenant restore always needs the target tenant’s *groups* to exist first. Restore your Entra groups (or your group-provisioning automation) before you restore the Intune config that assigns to them. In practice this means your DR runbook has an ordering: identity and groups first, then Intune configuration on top.
- **Not everything round-trips perfectly.** Secrets inside profiles (VPN pre-shared keys, certificate payloads, token-based store connections like Apple VPP/ABM or the Managed Google Play link) are not exported in a form you can just replay — they either need re-entering or re-establishing the connector. Know which of your objects carry

these, and document the manual steps for those specifically. Test a real restore into a scratch tenant at least once, so you discover these gaps in a drill and not in an outage.

Environment Strategy and Promotion

The reason to have separate tenants is that you never want to test a compliance-policy change on the tenant that controls whether your CFO can open their laptop. The mature pattern is a **dev/test → production** promotion flow, and it maps cleanly onto git branches and pull requests.

- **Dev tenant** — a cheap, low-license tenant where admins experiment freely in the portal. Changes originate here. The export pipeline captures them into a branch.
- **Test/staging tenant** (optional but valuable at scale) — where a change is validated against realistic groups and pilot devices before it goes wide.
- **Production tenant** — the real one, which *only* ever receives configuration through a merged, reviewed pull request deployed by the pipeline. Nobody makes changes directly in the production portal; if they do, drift detection catches them and the change goes through review retroactively or gets reverted.

Promotion, then, is just moving configuration through branches: a change proven in dev becomes a PR into `main`, review happens, and the merge deploys it to production. The subtlety is that tenants are *similar but not identical* — production targets different group IDs, maybe a different scope-tag structure, sometimes a different value for a handful of settings. Handle that with per-tenant **overrides** rather than divergent baselines: keep one canonical configuration and a small, explicit layer of per-tenant differences (IntuneCD supports assignment and value scoping for exactly this). The goal is that the *difference* between two tenants is itself a reviewable, version-controlled artifact — never an accident.

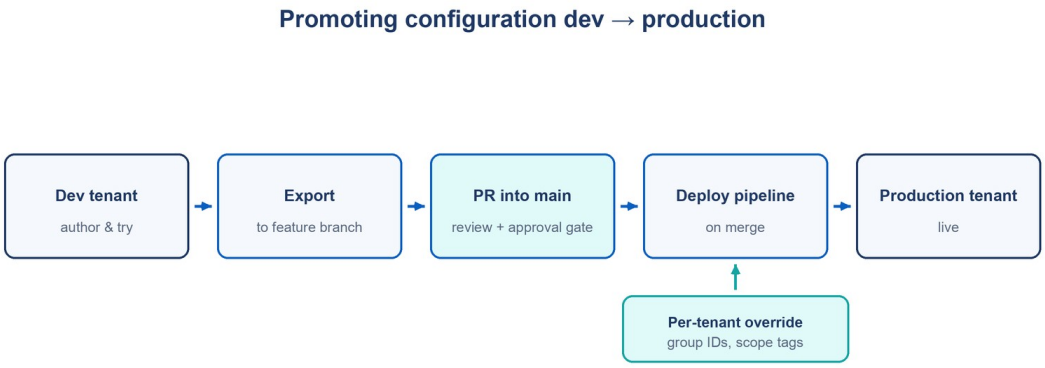

Figure 9.3: Promotion is a pull request — a change proven in dev flows into main through review, and the merge is what deploys it to production, with deliberate per-tenant differences expressed as explicit overrides.

For MSPs the same shape generalizes sideways instead of upward: one baseline repository, a per-customer override layer, and a deploy job per customer tenant, each with its own federated credential. Onboarding a new customer to your standard becomes "add their tenant's overrides and credential and run the deploy" rather than a week of clicking.

Governance: Who Can Merge, and the Least-Privilege Pipeline

The pipeline is now the *only* sanctioned way configuration reaches production, which means the pipeline's controls *are* your change-management process. Get these right or the rest is theater.

Protect the `main` branch. In the repository's branch protection (or rulesets) require pull requests before merging, require at least one approving review, dismiss stale approvals when new commits land, and — importantly — do not let anyone, including admins, push directly to `main`. If the only path to production is a reviewed PR, then peer review is not a nice-to-have you can skip under pressure; it is enforced by the platform.

Use CODEOWNERS for the sensitive bits. A `CODEOWNERS` file lets you require specific reviewers for specific paths — so a change under `backup/Compliance Policies/` or a conditional-

access object *must* be approved by your security lead, even if a general reviewer would do for a wallpaper profile. Match the review rigor to the blast radius.

Gate production behind a GitHub environment. Configure the `production` environment (the one the deploy job references) with a **required reviewer** and, if you like, a wait timer. This adds a second, deployment-time approval on top of the code review, and — because you scoped the federated credential to that environment — it means the production-deploy *identity cannot even be used* except from an approved run targeting that environment. The approval gate and the credential boundary reinforce each other.

Treat the pipeline identity as least-privilege, and split read from write. I said it under authentication and I will say it again here because it is a governance control, not just a technical one: the deploy identity can rewrite your whole device configuration, so it gets *only* the scopes it needs and it lives behind the environment gate. The scheduled export identity is read-only and can run freely because the worst it can do is open a PR. Two identities, two blast radii. Rotate the humans who can approve, review who has access to the app registrations periodically, and remember that with workload identity federation there is no secret to leak — the attack surface is "who can merge and approve," which is precisely the surface you want your governance focused on.

One last cultural note, because tools do not fix behavior on their own: the discipline only holds if the team agrees that the portal is *read-mostly* for production. People can and should explore in the portal on the dev tenant. But the moment production changes routinely happen by clicking and only sometimes get captured, you are back to drift and archaeology with extra steps. The payoff — the audit trail, the reviews, the one-command rollback — comes only when the pipeline is genuinely the front door.

Conclusion

Managing Intune as code is the same mental shift the software world made years ago, arriving late but very welcome to device management: your configuration is not a state you maintain by

hand in a web UI, it is an artifact you version, review, and deploy. The tooling in mid-2026 gives you real choices — **IntuneCD** for a diffable, JSON-based backup-and-deploy flow that most teams should start with, **Microsoft365DSC** when your remit spans all of M365, the new native **Tenant Configuration Management APIs** as the strategic direction to watch, and **Bicep or Terraform** for the Azure resources underneath — but the pattern is constant across all of them: git is the source of truth, a pull request is the change control, a pipeline is the deploy mechanism, and **workload identity federation** means none of it stores a secret. Drift detection closes the loop, separate tenants and PR-based promotion keep production safe, and branch protection plus a least-privilege pipeline identity turn "who changed this?" from a mystery into a commit.

We have now automated the *configuration* of the tenant. The obvious next question is what runs *on* the devices that configuration manages — and specifically the applications, which are their own lifecycle of packaging, testing, deploying, updating, and retiring. In **Chapter 10, "Automating the Application Lifecycle,"** we take the same DevOps instincts — version control, pipelines, keyless authentication, promotion through environments — and point them at apps: building `.intunewin` packages in a pipeline, testing them, and deploying them to Intune automatically, so that shipping a new version of an application becomes a pull request rather than an afternoon of manual packaging.

Chapter 10: Automating the Application Lifecycle

Chapter 9 was about treating Intune itself as code — defining your configuration in Bicep and JSON, running it through a CI/CD pipeline, and backing it up so a bad afternoon does not become a bad quarter. This chapter takes that same DevOps instinct and points it at the single most tedious, most repetitive, most never-actually-finished job in endpoint management: applications. Getting them packaged, getting them deployed, keeping them current, and eventually retiring them.

I want to be honest about why this chapter exists at all. In every environment I have ever worked in, apps are where the time goes. Not the glamorous stuff — not the Conditional Access policy or the shiny new Autopilot flow — but the endless grind of "Chrome released a new version," "the vendor changed their installer switches," "this MSI needs a transform," "who packaged 7-Zip and why does the detection rule point at the wrong path." A fleet of any size has dozens to hundreds of applications, each of which updates on its own schedule, and every one of those updates is a small packaging-and-testing task that somebody has to do. Multiply that out and you have a full-time job that produces nothing visible when it goes well and a flood of tickets when it goes badly.

So the goal here is not to teach you to package one app by hand. The goal is to make the application lifecycle *boring* — to push as much of it as possible onto the platform, onto a pipeline, and onto Graph, so that the humans only touch the cases that genuinely need a human. We will go deep on Win32 apps because that is still where the real complexity lives, look hard at PSADT because it is how you make installs behave consistently, and then spend real time on the native, no-extra-cost paths — winget and the Enterprise App Catalog — that in 2026 let you stop packaging a surprising number of apps entirely.

The App Problem and the Lifecycle

Every application in your tenant moves through the same loop, whether you manage it deliberately or not:

1. **Acquire** — get the installer and the licensing sorted, from the vendor, a store, or an internal build.
2. **Package** — wrap it so Intune can deliver it silently, with install and uninstall logic and a way to tell whether it is present.
3. **Test** — prove it installs, uninstalls, and detects correctly on a real device, not just your machine.
4. **Deploy** — assign it to the right users or devices, in the right order, without breaking anyone.
5. **Update** — replace the old version with the new one when the vendor ships, ideally without anyone noticing.
6. **Retire** — pull it cleanly when it is end-of-life or no longer needed, without leaving orphaned files or broken detections.

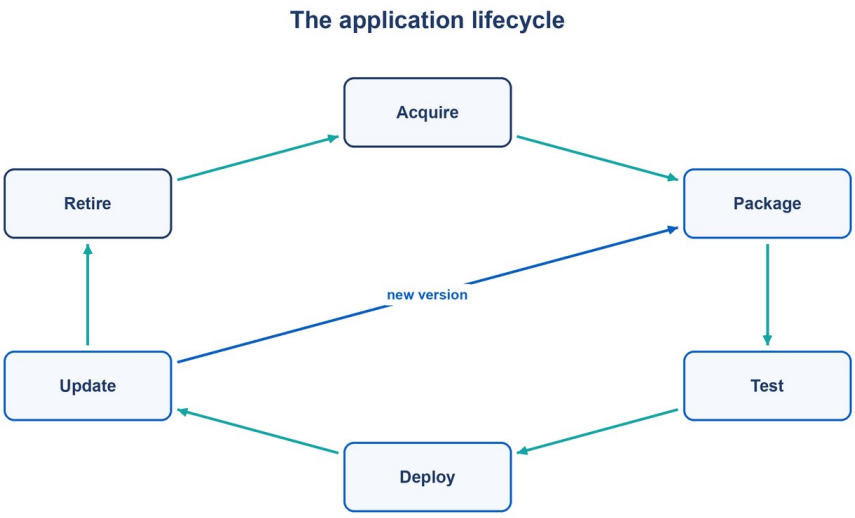

Figure 10.1: The application lifecycle loop — acquire, package, test, deploy, update, retire — with update feeding back into package, because an update is just the lifecycle running again on a new version. The whole point of this chapter is to automate every arrow so humans only handle the exceptions.

The reason this is a *loop* and not a line is step 5. An update is not a special event; it is the lifecycle starting over with a new version of the same thing. Every argument for automating the first pass is doubled for the tenth. If packaging Chrome once takes you an hour, packaging it every three weeks forever is the thing that eats your year. That realization is what pushes you toward the native catalog for the long tail and toward a pipeline for everything you still package yourself.

Intune gives you several delivery mechanisms for apps, and choosing the right one per app is half the battle:

- **Win32 apps** (`.intunewin`) — the workhorse. Anything with an `.exe` or `.msi` installer, any custom install logic. Maximum control, maximum effort.
- **Microsoft Store apps (new)** — winget-backed store delivery, auto-updating, zero packaging.
- **Enterprise App Catalog apps** — curated, Microsoft-hosted Win32 packages with native updating, part of the Intune Suite.
- **Line-of-business (LOB) apps** — a raw `.msi`/`.appx`/`.msix` uploaded directly; the older, simpler path, largely superseded by Win32 for Windows.
- **Platform stores** — managed Google Play, Apple VPP/App Store, macOS `.pkg`/`.dmg`.

The decision tree is short and worth internalizing: *if the native catalog or the store has it and keeps it current, use that; only package a Win32 app yourself when nothing else will do.* Most teams have this backwards — they package everything out of habit, then drown in updates. We will spend the first half of this chapter on Win32 because you must understand it, and the second half convincing you to use it less.

Win32 Apps End to End

The Win32 app is the most capable delivery format Intune has for Windows, and everything about it is designed around one hard constraint: the Intune Management Extension (IME) has

to install your app on a device it has never seen, with no human present, silently, and then be able to prove afterwards whether the install worked. Every concept below exists to serve that constraint.

The .intunewin format and IntuneWinAppUtil

Intune does not deploy your raw installer. It deploys a `.intunewin` file — an encrypted, compressed wrapper around your entire source folder. You create it with the **Microsoft Win32 Content Prep Tool**, `IntuneWinAppUtil.exe`, a small command-line utility Microsoft publishes on GitHub. It takes three inputs: the folder holding your installer and supporting files, the name of the setup file inside that folder, and an output location. It encrypts the lot into a single `.intunewin` you upload to Intune.

```
# Wrap a source folder into an .intunewin package.
# -c source folder, -s the installer inside it, -o output folder, -q quiet.
.\IntuneWinAppUtil.exe `
   -c "C:\Packaging\7zip\source" `
   -s "7z2408-x64.msi" `
   -o "C:\Packaging\7zip\output" `
   -q
```

Two things trip people up. First, everything in the source folder goes into the package, so keep it clean — no stray installers, no test logs, no `Thumbs.db`. Second, when you point `-s` at an `.msi`, the tool reads the MSI's product code and metadata into the package, which Intune then offers to use for the detection rule automatically; point it at an `.exe` and you get none of that, so you supply the detection logic yourself. That difference matters more than it looks.

For anything you do repeatedly, you do not run this by hand. The whole thing is a command-line call, so it drops straight into a build script or a pipeline stage — which is exactly what we wire up in the worked example later, and exactly the CI/CD muscle memory from Chapter 9.

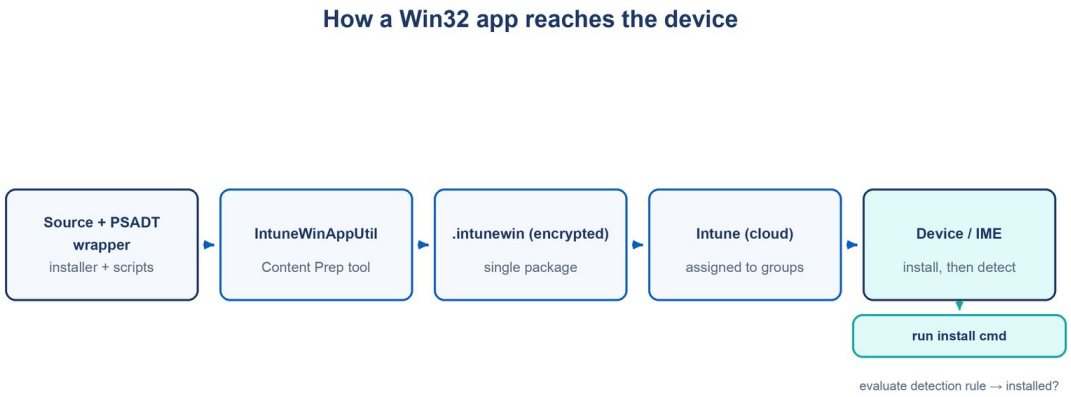

Figure 10.2: The Win32 packaging pipeline — source plus a PSADT wrapper is compiled by IntuneWinAppUtil into an encrypted .intunewin, uploaded to Intune, and pulled down by the Intune Management Extension, which runs the install command and then evaluates the detection rule to decide whether the app is present.

Detection rules — how Intune knows the app is there

This is the single most important part of a Win32 app, and the part people get wrong most often. Intune does not track "did the installer exit successfully" as the source of truth. It runs a **detection rule** after the install command and *that* is what decides whether the app counts as installed. Get the detection rule wrong and you get the classic failure modes: an app that installs perfectly but reports "failed," or an app that never installs because Intune already thinks it is there.

You have four kinds of detection rule, and you can combine several (all must pass):

- **MSI** — matches on the MSI product code, optionally the version. The easy path when you packaged an MSI; Intune pre-fills the product code for you.
- **File** — a file or folder exists at a path, optionally at or above a version, or modified after a date. The everyday choice for `.exe` installers: point at the main executable and require a minimum version.

- **Registry** — a key or value exists, equals, or is at least a given value. Good for apps that stamp their version into `HKLM\...\Uninstall`.
- **Custom script** — a PowerShell script that exits `0` and writes to stdout when the app is detected. The escape hatch for anything the other three cannot express — a file *and* a registry value, a licensing check, a version comparison the built-in rule cannot do.

A file-version rule is what I reach for by default because it survives updates cleanly. "`C:\Program Files\Vendor\app.exe` exists with version ≥ 1.2.3" is a rule that means the same thing on install day and six months later. Contrast that with a script that just checks a folder exists — it will happily report an ancient version as "installed" and block your update forever.

Here is the shape of a custom detection script, because you will eventually need one:

```
# Custom detection: report the app as present only if the installed
# version is at least the version we shipped. Exit 0 + stdout = detected.
$exe = "C:\Program Files\Contoso\Contoso.exe"
$min = [version]"4.2.0"

if (Test-Path $exe) {
    $v = [version](Get-Item $exe).VersionInfo.FileVersion
    if ($v -ge $min) {
        Write-Output "Detected $v"
        exit 0          # detected
    }
}
exit 1                  # not detected -> Intune will (re)install
```

Requirement rules — should this device even try

Where detection rules run *after* install to check the result, **requirement rules** run *before* to decide whether the device is a candidate at all. Minimum OS version, architecture (x64 vs ARM64 — increasingly relevant), free disk space, a registry precondition, or a custom script that must pass. A device that fails a requirement rule is reported as "not applicable" rather than "failed," which keeps your reporting honest: an app your ARM64 devices were never meant to get should not show up as a fleet-wide failure. Use requirement rules to *scope down to eligible hardware*, and use assignment (later) to scope down to *who should have it*.

Dependencies and supersedence — the two relationships

Two Win32 apps can be related in two ways, and understanding the difference is what separates a clean app estate from a tangled one.

Dependencies express “install B before A.” Visual C++ Redistributables before an app that needs them; a runtime before the tool that runs on it. You can chain them, and you can set each dependency to auto-install or merely to be required. Intune installs the dependency graph in order. Keep it shallow — deep dependency chains are where install-order bugs breed, and if Intune cannot cleanly detect a dependency it may skip or stall the whole chain.

Supersedence expresses “this app replaces that one.” This is how updates and replacements actually work in Intune, and it is worth being precise about because the mental model is not obvious.

When you configure App B to supersede App A, you choose one of two behaviors:

- **Update** — install B; the old A is updated in place. This is the normal path for “Chrome 128 supersedes Chrome 127.”
- **Uninstall the previous version** — remove A first, then install B. Use this when the old and new cannot coexist, or when you are genuinely replacing one product with a different one (migrating from Reader to a different PDF tool, say).

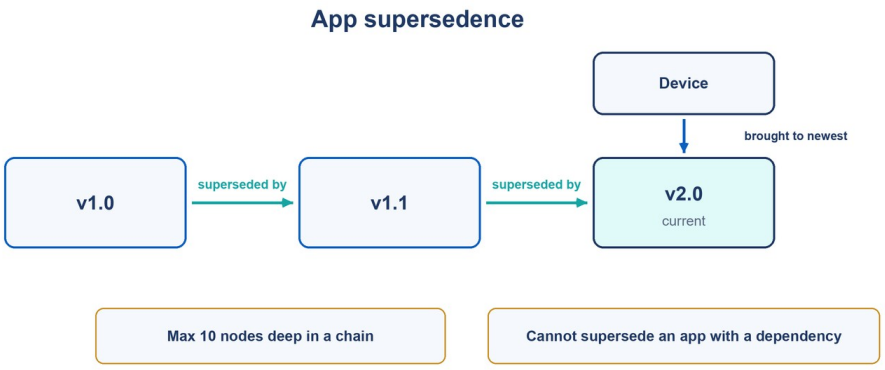

Figure 10.3: A supersedence chain. Each new package supersedes the one before it, and Intune moves a device from whatever version it has directly to the newest. Two constraints matter in practice: a supersedence graph can be at most 10 nodes deep, and you cannot supersede an app with one of its own dependencies.

Three rules about supersedence that the documentation states plainly and that you will hit in practice. A supersedence relationship can be **at most 10 nodes** deep — so you cannot let a chain grow forever; you retire old links as you go. You **cannot supersede an app with one of its own dependencies**. And supersedence changes need to *replicate* before devices act on them, so an update is never instantaneous across the fleet. Practically, most teams do not maintain long chains at all: they keep a single "current" package per app and repoint its content and detection rule for each new version, using supersedence only when they specifically need the old version removed. Both models are valid; the chain gives you cleaner reporting and rollback reasoning, the single-package model gives you less to manage.

PSADT — Making Installs Behave

The `.intunewin` format solves delivery. It does not solve the messy reality that installers are inconsistent, that some need a running app closed first, that users deserve to be told what is happening, and that when an install fails at 2 a.m. you want a log that tells you why. That is the job of the **PowerShell App Deployment Toolkit (PSADT)**.

PSADT is a free, open-source framework that wraps your installer in a consistent PowerShell harness. It gives you, out of the box: silent and interactive install/uninstall logic, a standard set of functions for the things every install needs (close these apps first, show this dialog, block launch during install, defer if a user is busy, write a detailed log), branded user-facing dialogs, deferral and countdown handling, and exit codes that Intune understands (including the all-important `3010` for "success, reboot required"). You stop writing the same "kill the process, copy the files, register the DLL, handle the reboot" boilerplate for every app and start declaring *what* should happen.

The current major version is **PSADT v4** (the 4.1.x line as of mid-2026). If you last used PSADT a couple of years ago, v4 is a real change and worth calling out. It is now delivered as a proper **PowerShell module** (`PSAppDeployToolkit`, installable from the PowerShell Gallery) rather than a loose folder of scripts you copied around. The functions were renamed into a consistent, prefixed verb-noun style (`Show-ADTInstallationWelcome`, `Start-ADTProcess`, and so on), the old `Deploy-Application.ps1` entry point gave way to `Invoke-AppDeployToolkit.ps1`, and — a genuinely nice detail — it dropped its dependency on the old `ServiceUI.exe` trick for showing UI from the SYSTEM context. A v3 compatibility layer exists so your old packages still run, but new work should be v4-native.

How it fits Intune is simple: your PSADT folder *is* the source folder you hand to IntuneWinAppUtil, and the setup file you point `-s` at is the toolkit's entry script. Your Win32 app's install command becomes a call into the toolkit. Here is a compact, v4-flavored example — the deploy script for an app that must close Outlook before it installs, shows the user a welcome prompt with a few deferrals, and writes its own log:

```
# Invoke-AppDeployToolkit.ps1 (PSADT v4) — abridged Install section.
# Runs silently under Intune, or interactively if a user is present.

# --- Pre-install: ask the user to close Outlook, allow 3 deferrals ---
Show-ADTInstallationWelcome -CloseProcesses 'outlook' -AllowDefer -DeferTimes 3 -CheckDiskSpace

# --- Show a progress dialog while we work ---
Show-ADTInstallationProgress -StatusMessage 'Installing Contoso Add-in 4.2…'

# --- Install: run the MSI silently; PSADT handles logging and exit codes ---
Start-ADTMsiProcess -Action Install -FilePath 'ContosoAddin.msi' -ArgumentList 'ALLUSERS=1'

# --- Post-install: drop a config file into place ---
Copy-ADTFile -Path "$($adtSession.DirFiles)\settings.xml" `
             -Destination "$env:ProgramData\Contoso\settings.xml"

# --- Tell the user we are done ---
Show-ADTInstallationPrompt -Message 'Contoso Add-in installed successfully.' -ButtonRightText 'OK' -Icon Information
```

Your Intune install command line for this package is then just:

```
Install:   Invoke-AppDeployToolkit.exe -DeploymentType Install   -DeployMode Interactive
Uninstall: Invoke-AppDeployToolkit.exe -DeploymentType Uninstall -DeployMode Silent
```

The payoff is consistency. Every app you wrap this way logs to the same place (`C:\Windows\Logs\Software` by convention), handles reboots the same way, closes conflicting apps the same way, and presents the same branded UI to users. When something breaks on one device out of three thousand, you know exactly where the log is and what it will look like. That uniformity is worth far more than any single feature.

The Native, No-Extra-Cost Path

Now the part that should change how you work. For years, "deploy an app with Intune" meant "package a Win32 app," and the whole industry built skills, tools, and habits around that assumption. In 2026 that assumption is wrong for a large fraction of your apps, because Microsoft ships two native paths that package *nothing* and update *automatically*.

Microsoft Store apps (new) — winget under the hood

The **Microsoft Store app (new)** type in Intune is backed by the **Windows Package Manager (winget)**. You search the store catalog directly inside Intune — which now spans UWP apps, MSIX packages, and Win32 apps delivered as `.exe`/`.msi` — pick an app, and assign it. There is no `.intunewin`, no detection rule to author, no install switches to reverse-engineer. Intune installs it through winget on the device, and — this is the key part — **keeps it up to date automatically** when the publisher ships a new version. Firefox, VLC, Notepad++, PowerToys, Zoom, and a long list of common tools are one search-and-assign away, and they stay current with no further work from you. This is a base Intune capability, no add-on required.

The catch is honesty about the catalog: the store's app metadata is community-maintained, versions can lag the vendor's own release by a little, and you get less control over exact version and install behavior than a hand-built Win32 package gives you. For the majority of everyday productivity tools that is a trade worth making gladly.

The Enterprise App Catalog — curated and Microsoft-hosted

One tier up sits the **Enterprise App Catalog**, part of **Enterprise Application Management** in the Intune Suite — the same Suite we discussed back in Chapter 4. Where the Store app type leans on community metadata, the Enterprise App Catalog is a **curated collection of enterprise-ready Win32 apps, prepared and hosted by Microsoft**. You browse a catalog of well over a thousand titles — Adobe Acrobat, Google Chrome for Business, the Eclipse Temurin and Azul Zulu JDK matrices, 7-Zip, Firefox, Wireshark, and so on — select one, and Intune generates a proper Win32 app for you, with the install commands, detection rules, and requirement rules already filled in. Content is served from Microsoft's own `*.manage.microsoft.com` endpoints and installed by the Intune Management Extension. (A common misconception: this is *not* winget — it is Microsoft-hosted Win32 content, which is why it carries proper enterprise detection and requirement rules rather than store metadata.)

The feature that makes the catalog matter for this chapter is **auto-update**, which reached general availability in the June 2026 Intune update wave. Turn it on for a supported catalog app and Intune detects when a newer catalog version exists and updates targeted devices automatically — native patching, no repackaging, no supersedence chain to maintain by hand. For the long tail of common third-party apps, this is the closest thing to "set it and forget it" that Intune has ever shipped.

Be equally clear about what the catalog does *not* do today, because these gaps decide when you still reach for something else: there is **no rollback** if a new version misbehaves, **no native ring model** built into the auto-update itself (you approximate rings with assignment, as below), and **no running-app detection** — an update can land while the app is open. Those are real limitations, not deal-breakers, and knowing them is the difference between using the catalog well and being surprised by it.

When native is enough

My working rule in 2026: **reach for native first, package second.** If an app is in the Enterprise App Catalog and you have the Intune Suite, use the catalog and turn on auto-update. If it is in the Store (new) catalog and the Suite is not in play, use that. Only build a Win32 package yourself when you need something native cannot give you — a specific pinned version, a custom transform or config file, unusual install logic, a preinstall/postinstall step, or a genuine license-server dependency. That leaves you hand-packaging the handful of apps that truly earn it, instead of all of them.

Patch Management and Deployment at Scale

Packaging an app once is a task. Keeping the whole estate current forever is a *system*, and it needs three things working together: automated packaging for what you still build yourself, automated assignment so apps reach the right devices without hand-editing groups, and a ring-based rollout so a bad package hurts a few people instead of everyone.

Assignment automation with dynamic groups and filters

You do not want to be adding devices to app-assignment groups by hand. Two mechanisms remove that work.

Dynamic groups are Entra ID groups whose membership is defined by a rule over device or user attributes, evaluated continuously. A device that matches joins automatically; one that stops matching drops out. “All Windows devices,” “all devices in the Marketing department,” “all devices whose name starts with LT-” — express it once as a membership rule and the group maintains itself.

```
# Dynamic membership rule: all corporate Windows 11 devices
(device.deviceOSType -eq "Windows") and
(device.deviceOSVersion -startsWith "10.0.2") and
(device.deviceOwnership -eq "Company")
```

Assignment filters are the sharper, more modern tool and I reach for them first. Instead of proliferating dozens of near-identical groups, you assign an app to one broad group and attach a *filter* that includes or excludes devices by rule at assignment time. The filter is evaluated per assignment, so the same filter — say, "only ARM64 devices" or "exclude devices with `kiosk` in the name" — is reusable across many apps and policies. The pattern that scales is: **broad groups for the *who*, filters for the *which*.** It keeps your group count sane and your intent readable.

Ring-based rollout

Never ship an app or an update to everyone at once. Stage it through **rings** — successive rings of increasing size and decreasing risk tolerance — so problems surface while they are still small.

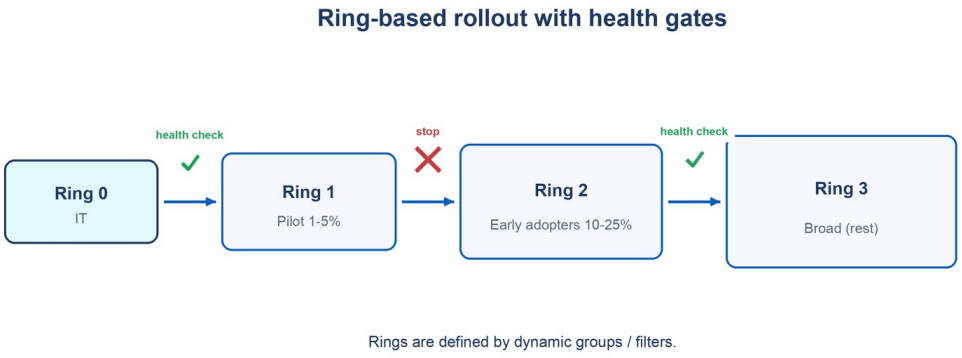

Figure 10.4: Ring-based rollout. A new package or update flows through progressively larger rings — pilot, early adopters, broad — with a health check at each gate. A problem caught in the pilot ring affects a handful of forgiving users instead of the whole fleet, and only a clean ring gets promoted to the next.

A typical four-ring model:

- **Ring 0 — Packaging/IT (a handful of devices).** You and your team. Proves the package installs, uninstalls, and detects at all.

- **Ring 1 — Pilot (1–5%).** Friendly, tech-tolerant users across a mix of hardware. Catches real-world conflicts your lab did not.
- **Ring 2 — Early adopters (10–25%).** A representative slice of the business. If it is clean here, it is almost certainly clean everywhere.
- **Ring 3 — Broad (the rest).** Everyone else, once the earlier rings have soaked for long enough.

You implement rings with the assignment tools above — a group or filter per ring — and you *gate promotion on health*: check the app's install-status report (or, better, pull it through Graph and alert on it, exactly as in Chapter 8) before you promote to the next ring. The native catalog's auto-update does not build rings for you, which is precisely why you approximate them with staged assignment: assign the catalog app to Ring 1 first, confirm health, then widen. The discipline is the same whether you built the package or Microsoft did.

A Worked Example: Package, Detect, Deploy, Auto-Update — Driven by Graph

Let us pull it together into something that fits the pipeline from Chapter 9. The repetitive parts — create the app, wait for the upload, add the detection rule, wire up supersedence, assign it — are all Graph calls, so we drive them with the Microsoft Graph SDK, keyless, exactly the way Chapter 8 insisted on. No client secret anywhere: the automation runs as a managed identity (in Azure) or via workload identity federation (in a GitHub Actions pipeline), granted `DeviceManagementApps.ReadWrite.All`.

Step 1 — package it. Wrap the source with the content prep tool. In a pipeline this is one build step:

```
.\IntuneWinAppUtil.exe -c ".\source" -s "Invoke-AppDeployToolkit.exe" -o ".\out" -q
```

Step 2 — connect keyless. No secret, no token juggling — the same one-liner from Chapter 8:

```
Connect-MgGraph -Identity -NoWelcome        # in Azure (managed identity)
# or, in GitHub Actions, azure/login federates the token first, then:
# Connect-MgGraph -Identity
```

Step 3 — create the Win32 app and upload the content. The full Win32 upload is a genuinely fiddly multi-step dance: create the app object, create a content version, request an Azure Storage SAS URI, upload the encrypted `.intunewin` in blocks, commit the file with the encryption keys the content prep tool emitted, then poll until Intune reports the file processed. It is dozens of lines of block-blob bookkeeping, and it is exactly the kind of thing you should not hand-roll. The community **`IntuneWin32App` PowerShell module** (by Nickolaj Andersen) implements the whole sequence cleanly and is the sane way to do this in a pipeline. The detection-rule and assignment logic below is the part worth showing in full, because it is where the decisions live:

```
Import-Module IntuneWin32App

# Detection rule: main exe present at or above the version we shipped.
$detection = New-IntuneWin32AppDetectionRuleFile `
    -Existence -Path "C:\Program Files\Contoso" `
    -FileOrFolder "Contoso.exe" -DetectionType "exists" `
    -Check32BitOn64System $false

# Requirement rule: 64-bit, Windows 11 22H2 or newer.
$requirement = New-IntuneWin32AppRequirementRule `
    -Architecture "x64" -MinimumSupportedWindowsRelease "W11_22H2"

# Create the app and push the .intunewin in one call.
$app = Add-IntuneWin32App `
    -FilePath ".\out\Invoke-AppDeployToolkit.intunewin" `
    -DisplayName "Contoso Add-in 4.2" -Publisher "Contoso" `
    -InstallCommandLine  "Invoke-AppDeployToolkit.exe -DeploymentType Install -DeployMode Silent" `
    -UninstallCommandLine "Invoke-AppDeployToolkit.exe -DeploymentType Uninstall -DeployMode Silent" `
    -DetectionRule $detection -RequirementRule $requirement `
    -InstallExperience "system" -RestartBehavior "suppress"
```

Step 4 — wire up supersedence so this replaces the version before it. This is a raw Graph relationship, and here it is via the SDK so you can see the shape:

```
# $app.id is the new package; $oldAppId is the version it replaces.
$body = @{
   "@odata.type"     = "#microsoft.graph.win32LobApp"
   supersededAppCount = 1
   relationships = @(
      @{
         "@odata.type"      = "#microsoft.graph.mobileAppSupersedence"
         targetId           = $oldAppId
         supersedenceType   = "update"   # or "replace" to uninstall first
      }
   )
} | ConvertTo-Json -Depth 5

Invoke-MgGraphRequest -Method POST `
   -Uri "https://graph.microsoft.com/beta/deviceAppManagement/mobileApps/$($app.id)/updateRelationships" `
   -Body $body
```

Step 5 — assign it to Ring 1 with a filter. Assign to the pilot group, scoped by a reusable filter, so the rollout starts small:

```
Add-IntuneWin32AppAssignmentGroup -Include -ID $app.id `
   -GroupID $pilotRingGroupId -Intent "required" `
   -FilterID $arm64ExcludeFilterId -FilterMode "Include" `
   -Notification "showAll"
```

Step 6 — gate the promotion. Before widening to Ring 2, check install health through Graph and stop the pipeline if failures cross a threshold — the same reporting seam from Chapter 8, now used as a quality gate:

```
$status = Invoke-MgGraphRequest -Method GET `
   -Uri "https://graph.microsoft.com/beta/deviceAppManagement/mobileApps/$($app.id)/deviceStatuses"
$failed = ($status.value | Where-Object installState -eq 'failed').Count
if ($failed -gt 2) { throw "Ring 1 failures ($failed) exceed threshold — halting rollout." }
```

For an app that lives in the Enterprise App Catalog, steps 1–4 collapse into “select it from the catalog and enable auto-update,” and only steps 5 and 6 — staged assignment and a health gate — remain your job. That is the whole argument for native, made concrete: the platform does the packaging and updating; you keep the risk management, because that part is genuinely yours to own.

Native vs. Ecosystem: An Honest Assessment

It would be easy to end on “the native catalog does everything now,” and that would not be true. So here is the honest boundary, framed as a buyer, not a fan of any product.

The native paths — Store apps and the Enterprise App Catalog — are excellent and, for a large fraction of common third-party software, genuinely sufficient. If your app is in the catalog, you have the Intune Suite, and you can live with the current constraints, native is the right answer and you should not be paying anyone for what Microsoft already ships. Do not buy a third-party patching product to cover apps the catalog already covers well; that is spending money to add a vendor.

Where the built-in catalog stops is at the edges, and the edges are real: apps that are not in the catalog at all (niche, industry-specific, or internal LOB software); the missing pieces of the update story — no rollback, no built-in rings, no running-app awareness before an update lands; and reporting or approval workflows richer than what Intune exposes natively. That gap is where community and third-party tooling still earns its keep. On the community side, there are well-regarded winget-based deployment frameworks and detection-rule generators that lower the cost of the Win32 apps you *do* still hand-build. On the commercial side, third-party patch-management services maintain their own tested catalogs of packaged apps — often broader than Microsoft’s, with rollback, ring orchestration, and update-deferral-while-running — and plug into Intune as Win32 apps or via their own connector.

The buyer’s question is not “native or third-party” as an identity; it is a coverage-and-cost calculation you redo periodically: *what fraction of my app estate does native now cover well, and is the remaining gap big enough to justify a tool and a vendor?* In 2026 that native fraction is materially larger than it was in the first edition of this book, and it grows with each catalog and auto-update improvement — which means the honest answer for many organizations is shifting from “buy the patching tool” toward “native plus a thin pipeline for the long tail.” Re-run the calculation, do not assume last year’s answer, and let the size of the gap decide.

Conclusion

The application lifecycle is where endpoint management quietly spends most of its time, and it is the part that most rewards automation, because the work is repetitive by definition — an update is just the lifecycle running again. We took it apart: the `.intunewin` format and the content prep tool; detection rules as the real source of truth about whether an app is present; requirement rules to scope to eligible hardware; dependencies and supersedence as the two ways apps relate; PSADT v4 to make installs behave and log consistently; and the native paths — Store apps and the Enterprise App Catalog with GA auto-update — that let you package far less than you used to. Then we scaled it: dynamic groups and filters for hands-off assignment, ring-based rollout for safety, and a Graph-driven, keyless pipeline that automates the repetitive parts and gates promotion on real install health.

The thread back to Chapter 9 is the pipeline; the thread forward is the payoff. Apps are only one thing that has to be provisioned, kept compliant, and kept secure across a fleet, and the same instincts — express intent as code, drive the repetitive parts through Graph without a stored secret, stage changes through rings, and gate on health — apply to all of it. In **Chapter 11, “Automating Provisioning, Compliance, and Security,”** we take these patterns off the app problem and point them at the rest of the endpoint: getting devices set up correctly from first boot, keeping them compliant without manual chasing, and automating the security posture that everything else depends on.

Chapter 11: Automating Provisioning, Compliance, and Security

Chapter 10 was about the application lifecycle — getting the right software onto a device and keeping it patched without a human clicking anything. This chapter widens the lens from "the apps on the device" to "the device itself," and it closes out the automation part of the book. The question here is the one every endpoint team is really being paid to answer: how does a machine go from a sealed box on someone's desk to a fully managed, compliant, defended endpoint that a user can actually work on — with no ticket, no imaging, no admin standing over it — and how does it *stay* that way when something drifts?

That is a chain of steps, and the whole point of this chapter is that every link in the chain can run itself. A device provisions itself through Autopilot. It lands in the right group automatically. It gets the right baseline and the right compliance policy because of where it landed. Conditional Access reads its compliance state and Defender reads its risk, and between them they decide whether the device gets to touch corporate data at all. And when something on that device slips — a setting drifts, a control gets turned off, risk spikes — automation notices and fixes it, ideally before anyone files a ticket. That last idea is the same one from Chapter 8, "spot the pattern, then fix it before anyone complains," pointed at security posture instead of a bloated temp folder.

I have rewritten a fair amount of this chapter for the second edition, because the provisioning story in particular moved. **Windows Autopilot device preparation** — sometimes called Autopilot v2 — is the newer provisioning experience, and it behaves differently enough from classic Autopilot that following an old tutorial will lead you astray. I will be explicit about what changed and, just as importantly, where the old way is still the right choice.

From Box to Managed: What Zero-Touch Really Means

“Zero-touch provisioning” is a phrase that gets thrown around loosely, so let me define it the way it actually plays out. A device arrives. A user — not an administrator — powers it on, connects to Wi-Fi, and signs in with their work account. From that sign-in forward, no human touches the machine: it enrolls into Intune, joins Microsoft Entra ID, pulls down its applications and configuration, applies a security baseline, evaluates against a compliance policy, and hands the user a desktop that is already managed and already defended. The user’s first action on their new laptop is to open Outlook, not to wait for IT.

Everything in this section is about making that happen without exceptions, because the exceptions are where the cost is. A provisioning flow that works for 95% of devices and needs a technician for the other 5% has not saved you as much as it looks — the 5% is where your time goes. So the automation has to be complete, and it has to be self-correcting when a step fails.

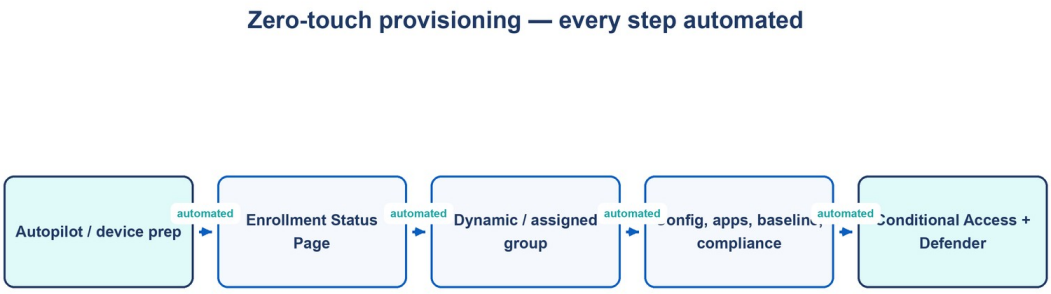

Figure 11.1: The box-to-compliant provisioning flow — Autopilot to Enrollment Status Page to group membership to policy assignment to access gating, with no manual step in the chain.

Windows Autopilot, and its successor

Classic **Windows Autopilot** is the mechanism most tenants still run. Its model is a hardware identity: each device is represented in your tenant by a **hardware hash** (the “Autopilot device identity”), registered either by the OEM at purchase, by a reseller, or by you manually. When a registered device hits the out-of-box experience, Windows recognizes it, pulls down its assigned

deployment profile, and drives enrollment from there. The profile decides the join type (Entra join is the default and the one you want), whether the user or a technician drives it (user-driven versus pre-provisioning), and cosmetic-but-important OOBE settings like hiding the privacy prompts and skipping the local-account fallback.

In 2026 there is a newer path sitting alongside it: **Windows Autopilot device preparation**, also called Autopilot v2. Microsoft built it to fix the two things that hurt most about classic Autopilot — the dependency on hardware-hash registration (which was always the fiddly part, especially for devices you did not buy through a managed OEM channel) and the sometimes-flaky reliability of the enrollment experience. Device preparation drops the hardware-hash requirement entirely: instead of pre-registering identities, you assign a **device preparation policy** to a group of *users*, and any device those users provision follows the policy. It is simpler to stand up and noticeably more reliable in my testing.

But — and this is exactly the kind of currency detail the first edition would have gotten wrong — device preparation is not a drop-in replacement for everything classic Autopilot does, at least not yet. As of mid-2026:

- It supports the **user-driven** scenario for physical devices. Pre-provisioning (the "white glove" model where a technician provisions the device before the user gets it) and self-deploying mode (for kiosks and shared devices) are on the roadmap but not in the box.
- It does **not** use the classic Enrollment Status Page. It has its own status experience, built into the policy, which tracks security setup, app installation, and so on — but if your muscle memory is the ESP blade, that is not where device-preparation status lives.
- The app limit during provisioning was raised to **25 apps**, which covers most real deployments but is worth knowing if you front-load a large app set into OOBE.
- Crucially, it targets an **assigned** device-preparation group, not a dynamic group. This is a real behavioral difference from classic Autopilot, and I will come back to it, because "dynamic groups do the targeting" is the default mental model and device preparation breaks it on purpose.

My practical guidance for mid-2026: if you are starting fresh, or you buy devices outside a managed OEM channel, or classic Autopilot reliability has burned you, evaluate device preparation first. If you depend on pre-provisioning or self-deploying today, stay on classic Autopilot until those scenarios ship for v2. Many tenants will run both for a while, and that is fine.

The Enrollment Status Page

For classic Autopilot, the **Enrollment Status Page (ESP)** is the piece that turns "the device is enrolling" into "the device is *finished* enrolling, and I can prove it." Without an ESP, a user can reach the desktop while apps and policies are still landing in the background — which produces the single most common help-desk call after a rollout: "my new laptop doesn't have Teams yet." The ESP blocks the desktop until the things you designate as required have actually applied.

Configure it under **Devices > Enrollment > Windows > Enrollment Status Page**. The settings that matter:

- **Show app and profile installation progress** — turn this on; it is the whole point.
- **Block device use until all apps and profiles are installed** — yes, so the user cannot start working on a half-built machine.
- **Block device use until these required apps are installed** — name the specific apps that constitute "ready." Keep this list tight. If you make every app blocking, one slow or failing app install holds the entire OOBE hostage, and you turn a reliability feature into a reliability problem.
- **Allow users to reset / retry / continue on failure** — set these deliberately. Allowing "continue anyway" trades a guaranteed-complete build for a user who is not stuck staring at an error, and which side of that trade you want depends on how critical the blocking apps are.

The ESP is where a lot of Autopilot pain actually lives, and almost all of it traces back to the blocking-app list being too long or containing an app that cannot install in the SYSTEM context during OOBE. Treat that list as a security-critical config, not an afterthought.

Targeting the Right Devices Automatically

Provisioning gets a device enrolled. What makes it *configured* is that the moment it enrolls, it falls into groups that carry policy — and it does so without anyone assigning it by hand. This is the automation that makes everything downstream possible, and it rests on two features: dynamic groups and filters.

Dynamic groups

A **dynamic device group** has membership defined by a rule, not a list. Devices that match the rule are added automatically; devices that stop matching are removed automatically. For classic Autopilot this is the backbone of targeting: your deployment profile, your configuration, your baseline, and your compliance policy are all assigned to dynamic groups, and a freshly provisioned device flows into them the instant its attributes match.

A few rules I use in almost every tenant. All Autopilot-registered devices, keyed off the physical device IDs that Autopilot stamps as the group tag and order ID:

```
(device.devicePhysicalIDs -any (_ -contains "[ZTDId]"))
```

That single rule captures every Autopilot device in the tenant — useful as the assignment target for a profile that should reach anything provisioned through Autopilot. To slice by purpose, Autopilot writes the **group tag** into the `OrderID` attribute, so you can carve out, say, all kiosk devices:

```
(device.devicePhysicalIds -any (_ -eq "[OrderID]:Kiosk"))
```

And a plain platform split, which you will reuse constantly for baselines and compliance:

```
(device.deviceOSType -eq "Windows") and (device.deviceOwnership -eq "Company")
```

The honest caveat, and it matters for automation: dynamic group membership is **not instant**. Entra evaluates the rules on a schedule, and for a large tenant a new device can take several minutes — occasionally longer — to land in its groups. That lag is exactly why classic Autopilot can feel like it "skipped" a policy on first boot: the device provisioned faster than the group evaluated. The ESP masks most of this by blocking until required apps land, but it is the reason device preparation moved to assigned groups.

Why device preparation uses assigned groups

Here is the design decision worth understanding rather than just memorizing. Autopilot device preparation assigns its policy to an **assigned** group — you add the group up front, and the provisioning process places the device into it as part of the flow. This removes the dynamic-evaluation lag from the critical path entirely: the device does not have to *become* a match and *wait* to be picked up; it is a member the moment it needs to be. That is a large part of why device preparation is more reliable. The trade is flexibility — you lose the "membership follows attributes forever" behavior of dynamic groups for the provisioning group itself. In practice you still use dynamic groups for everything *after* provisioning (baselines, compliance, apps); it is only the provisioning target that becomes assigned.

Filters: the scalpel

Groups decide *who gets an assignment*. **Filters** decide, at assignment time, *which members the assignment actually applies to* — evaluated live, at the moment of assignment, with no scheduling lag. A filter is a rule over device properties that you attach to an assignment as **include** or **exclude**, and it is the single best tool for keeping your group sprawl under control.

The pattern I lean on: assign a policy to one broad group (say, all Windows devices) and then use a filter to scope it precisely, rather than creating a new dynamic group for every combination. Create filters under **Tenant administration > Filters**. A filter that matches only Windows 11 devices:

```
(device.osVersion -startsWith "10.0.2") and (device.deviceOwnership -eq "Corporate")
```

A filter that matches a specific hardware model — handy when a baseline or a driver-sensitive setting should only hit one device type:

```
(device.model -contains "Surface Laptop")
```

Two things make filters better than more groups for automation. First, they evaluate **at assignment time against live device properties**, so there is none of the dynamic-group lag — a device that matches the filter is affected immediately. Second, one policy plus a filter is far easier to reason about than five near-identical groups, and "easier to reason about" is what keeps a security configuration correct over time. My rule of thumb: **use groups for broad identity ("all corporate Windows"), use filters for precision ("but only Windows 11 Surface devices").**

Automating the enrollment profile itself

Even the enrollment configuration is code you can manage rather than click. Every Autopilot deployment profile and ESP configuration is a Graph object, which means you version and deploy them the config-as-code way from Chapter 9 rather than reconstructing them by hand in a new tenant. Listing your deployment profiles:

```
Connect-MgGraph -Scopes "DeviceManagementServiceConfig.Read.All" -NoWelcome

# Classic Autopilot deployment profiles
Invoke-MgGraphRequest -Method GET `
    -Uri "https://graph.microsoft.com/v1.0/deviceManagement/windowsAutopilotDeploymentProfiles" |
    Select-Object -ExpandProperty value |
    Select-Object displayName, id, language, deviceNameTemplate
```

And creating one from a definition — the same shape you would keep in a repo and deploy through a pipeline:

```
$profile = @{
    "@odata.type"                = "#microsoft.graph.azureADWindowsAutopilotDeploymentProfile"
    displayName                  = "Corporate User-Driven"
    description                  = "Entra-joined, user-driven, hidden OOBE"
    language                     = "os-default"
    deviceNameTemplate           = "CORP-%SERIAL%"
    enableWhiteGlove             = $true
    outOfBoxExperienceSetting = @{
        deviceUsageType      = "singleUser"
        hidePrivacySettings  = $true
        hideEULA             = $true
        userType             = "standard"
        skipKeyboardSelectionPage = $true
    }
}

Invoke-MgGraphRequest -Method POST `
    -Uri "https://graph.microsoft.com/v1.0/deviceManagement/windowsAutopilotDeploymentProfiles" `
    -Body ($profile | ConvertTo-Json -Depth 5)
```

Once the profile, the ESP, the groups, and the filters exist as code, standing up provisioning in a new tenant — or restoring it after someone "fixed" it in the portal — is a pipeline run, not a day of clicking.

Compliance as an Engine, Not a Report

Most people meet compliance policies as a report: a dashboard that shows which devices are green and which are red. That is the least interesting thing about them. A compliance policy is only powerful when it is wired to *consequence* — when "non-compliant" does not just color a cell red but actually **takes access away** until the device is fixed. That wiring is the compliance → Conditional Access → access loop, and it is the closest thing Intune has to a fully automated enforcement engine.

The policy

A compliance policy is a set of rules a device must satisfy: minimum OS version, disk encryption on, a firewall enabled, antivirus healthy, a device-risk level under some threshold. You build them under **Devices > Compliance**. The rules themselves are unremarkable; what turns a policy into an engine is two things most people under-configure: the **actions for non-compliance** and the **grace period**.

When a device fails a rule, you decide what happens and *when*. The default action is "mark the device non-compliant immediately (0 days)," and you should almost never leave it there. A device can fail a check transiently — BitLocker is still encrypting right after provisioning, a service has not reported in yet — and slamming it non-compliant the instant it fails produces false blocks and angry users. So you configure a **schedule of actions**:

- **Send an email to the end user** a day or two before the hard action, telling them exactly what is wrong and how to fix it. This one email prevents more tickets than almost anything else you can do.
- **Mark the device non-compliant after a grace period** — I typically use 1 to 3 days for most rules, and a short window (an hour or two) for things like encryption that legitimately need time to finish after enrollment.

During the grace period the device shows as "In grace period" in Intune, and — this is the subtle part that trips people up — **Conditional Access does not block during that window.** Entra ID has not yet received a non-compliant signal, so tokens keep flowing. Only when the grace period expires without remediation does Intune write the non-compliant state up to Entra, and *that* is the moment access enforcement kicks in. The grace period is your soft landing: it gives the user (or the automation) a chance to fix the problem before the door closes.

The loop

Now the consequence. A compliance policy on its own changes nothing about access — it only produces a signal. **Conditional Access** is what reads that signal and acts on it. The pairing is simple and it is the single most important security automation in this entire chapter:

1. A compliance policy evaluates the device and writes its state — compliant or non-compliant — up to Microsoft Entra ID.
2. A Conditional Access policy requires the device to be marked **compliant** as a condition of access.

3. A device that goes non-compliant loses access automatically, the next time it tries to get a token. No admin does anything. The loop closes itself.

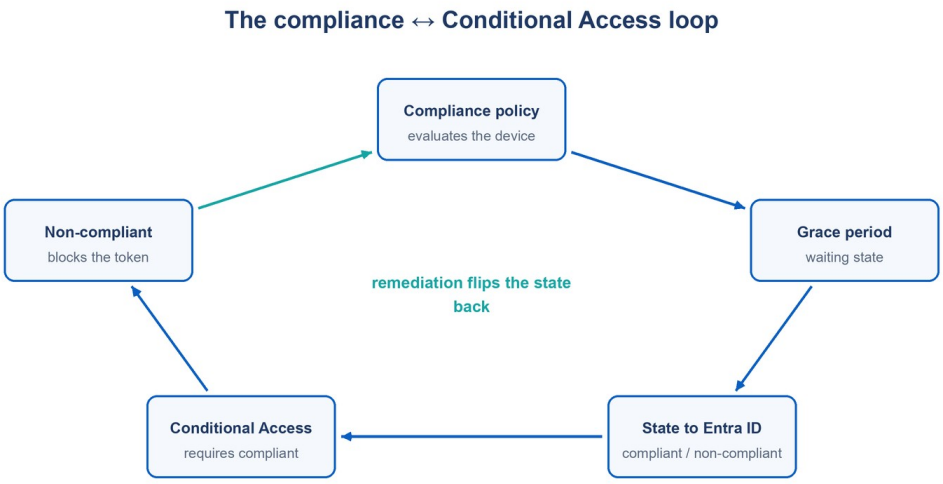

Figure 11.2: The compliance-to-Conditional-Access loop — compliance state becomes an access decision, with the grace period as the soft landing before the door closes.

Here is a concrete pairing. First, a Windows compliance policy created through Graph, requiring BitLocker, a healthy firewall, and a minimum build, with a grace period baked into the non-compliance schedule:

```
Connect-MgGraph -Scopes "DeviceManagementConfiguration.ReadWrite.All" -NoWelcome

$policy = @{
   "@odata.type"              = "#microsoft.graph.windows10CompliancePolicy"
   displayName                = "Windows - Baseline compliance"
   passwordRequired           = $true
   bitLockerEnabled           = $true
   secureBootEnabled          = $true
   firewallEnabled            = $true
   defenderEnabled            = $true
   osMinimumVersion           = "10.0.26100"     # Windows 11 24H2 build
   # Grace period + graduated actions
   scheduledActionsForRule = @(
      @{
         ruleName = "PasswordRequired"
         scheduledActionConfigurations = @(
            @{ actionType = "notification"; gracePeriodHours = 0;  notificationTemplateId = "" },
            @{ actionType = "block";        gracePeriodHours = 72 }   # 3-day grace
         )
      }
   )
}

Invoke-MgGraphRequest -Method POST `
   -Uri "https://graph.microsoft.com/v1.0/deviceManagement/deviceCompliancePolicies" `
   -Body ($policy | ConvertTo-Json -Depth 6)
```

Then the Conditional Access policy that gives that state teeth — require a compliant device for all cloud apps, for the corporate users, excluding a break-glass account so you never lock yourself out:

```
Connect-MgGraph -Scopes "Policy.ReadWrite.ConditionalAccess" -NoWelcome

$ca = @{
   displayName = "Require compliant device for cloud apps"
   state       = "enabledForReportingButNotEnforced"   # start in report-only!
   conditions  = @{
      users       = @{
         includeGroups = @("<all-corp-users-group-id>")
         excludeUsers  = @("<break-glass-account-id>")
      }
      applications = @{ includeApplications = @("All") }
      platforms    = @{ includePlatforms = @("windows") }
   }
   grantControls = @{
      operator        = "OR"
      builtInControls = @("compliantDevice")
   }
}

Invoke-MgGraphRequest -Method POST `
   -Uri "https://graph.microsoft.com/v1.0/identity/conditionalAccess/policies" `
   -Body ($ca | ConvertTo-Json -Depth 6)
```

Two things I want to flag hard, because getting them wrong is how people take their own tenant offline. First, **always start a Conditional Access policy in**

`enabledForReportingButNotEnforced` (report-only) mode and read the sign-in logs for a few days before you flip it to `enabled`. Report-only shows you exactly who *would* have been blocked without blocking anyone — it is the difference between a controlled rollout and a Monday-morning outage. Second, **always exclude a break-glass account.** A misfiring compliance policy that marks every device non-compliant, paired with an enforced "require compliant device" CA policy and no exclusion, will lock out every administrator including you. The break-glass account — a cloud-only account excluded from this and every other CA policy, with credentials in a sealed envelope — is your way back in.

Security Baselines and the Settings Catalog at Scale

Compliance measures whether a device meets a bar. **Security baselines** and the **settings catalog** are how you set that bar in the first place — the actual hardening applied to the device. And once you are managing more than a handful of settings across more than one tenant, you stop clicking them into the portal and start treating them the config-as-code way from Chapter 9.

Baselines

A **security baseline** is Microsoft's pre-built, recommended set of security configurations for a product — the Windows security baseline, the Defender for Endpoint baseline, the Edge baseline. Rather than researching hundreds of individual settings, you deploy a baseline as a starting point and adjust from there. They live under **Endpoint security > Security baselines**.

The detail that matters for automation is **versioning**. Microsoft releases new baseline versions periodically, and a new version is not applied to your devices silently — you have to migrate your baseline instances to it, reviewing what changed. This is genuinely a good thing (you do not want Microsoft silently changing your firewall rules), but it means baseline management is an ongoing task, not a one-time deploy. Check for new versions on a cadence, review the diff, and migrate deliberately.

This is exactly where the config-as-code discipline pays off. If your baseline exists as an exported definition in a Git repository, then a new baseline version becomes a **pull request**: you export the new version, diff it against what you run today, and the review happens in code where you can see every changed setting side by side — instead of squinting at a portal migration wizard. Exporting a baseline profile through Graph so you can commit it:

```
Connect-MgGraph -Scopes "DeviceManagementConfiguration.Read.All" -NoWelcome

# Security baselines are intents under templates; settings-catalog profiles
# are configurationPolicies. Export a settings-catalog policy with its settings:
$policyId = "<configuration-policy-id>"

$policy = Invoke-MgGraphRequest -Method GET `
   -Uri "https://graph.microsoft.com/beta/deviceManagement/configurationPolicies('$policyId')?`$expand=settings"

$policy | ConvertTo-Json -Depth 20 | Out-File "./baselines/windows-baseline.json"
```

Commit that JSON, and your baseline is now reviewable, diff-able, and redeployable. When Microsoft ships a new version, you export it, the diff shows you the delta, and you decide.

The settings catalog at scale

The **settings catalog** is the modern, unified surface for configuration — thousands of individual settings, searchable, from a single consistent UI, replacing the older per-template profiles. For anything beyond a baseline, this is where you build configuration in 2026. Its great virtue for automation is uniformity: every settings-catalog policy is the same `configurationPolicies` Graph object with the same shape, so one export script, one import script, and one diff tool handle *all* of your configuration. You are not writing a special case for the firewall profile and another for the Edge profile — they are all the same object type.

That uniformity is what makes “config as code at scale” real. A single pipeline can export every settings-catalog policy in a tenant to JSON, a reviewer can approve changes as a diff, and a deploy step can push them to another tenant — dev to prod, or one customer to the next. The settings catalog turned Intune configuration from a pile of bespoke blades into a homogeneous dataset, and homogeneous datasets are exactly what automation eats.

Defender for Endpoint: Risk in the Loop

The compliance loop as described so far is static — it checks fixed conditions like "is BitLocker on." **Microsoft Defender for Endpoint** makes it *dynamic*, by feeding a live **device risk level** into compliance. Now "compliant" does not just mean "correctly configured"; it means "correctly configured *and* not currently exhibiting risky behavior." A device that was compliant five minutes ago can go non-compliant the instant Defender sees something it does not like — and, through the same Conditional Access loop, lose access automatically until the threat is dealt with.

Wiring it up

The integration has three parts, and all three are automatable:

1. **Connect Defender for Endpoint to Intune.** In the Defender portal, turn on the Intune connection; in Intune, under **Endpoint security > Microsoft Defender for Endpoint**, enable the connector and allow it to set device risk. This is the pipe that carries risk signals from Defender into Intune.
2. **Onboard devices to Defender.** Do this through Intune configuration (the Defender for Endpoint security baseline or a settings-catalog policy) so onboarding is itself part of your automated provisioning, not a manual step.
3. **Add a risk rule to the compliance policy.** This is the setting `deviceThreatProtectionRequiredSecurityLevel` — it tells Intune the maximum Defender risk level a device may have and still count as compliant.

Adding the risk rule to a compliance policy through Graph. A threat level of `low` means "clear, low, or unknown risk is acceptable; medium and high are non-compliant" — Microsoft's guidance, and mine, is that `low` gives the best balance of security and productivity for most fleets:

```
$riskRule = @{
   "@odata.type"                        = "#microsoft.graph.windows10CompliancePolicy"
   displayName                          = "Windows - risk-based compliance"
   # Fail compliance if Defender rates the device above Low risk
   deviceThreatProtectionEnabled        = $true
   deviceThreatProtectionRequiredSecurityLevel = "low"
}

Invoke-MgGraphRequest -Method PATCH `
   -Uri "https://graph.microsoft.com/v1.0/deviceManagement/deviceCompliancePolicies('<policy-id>')" `
   -Body ($riskRule | ConvertTo-Json -Depth 4)
```

The closed loop

Now trace the full circle, because this is the payoff. Defender detects suspicious activity on a device and raises its machine risk to medium. The Intune connector carries that risk level into Intune. Intune re-evaluates the compliance policy, sees the risk exceeds the `low` threshold, and marks the device non-compliant — writing that state to Entra ID. The next time the device requests a token, the "require compliant device" Conditional Access policy sees non-compliant and blocks it. The device has lost access to corporate data **automatically, in response to a live threat, with no human in the loop.** Then the security team (or an automated response) remediates the threat, Defender lowers the risk, Intune re-evaluates to compliant, Entra gets the good signal, and access is restored on the next token request. The device let itself back in.

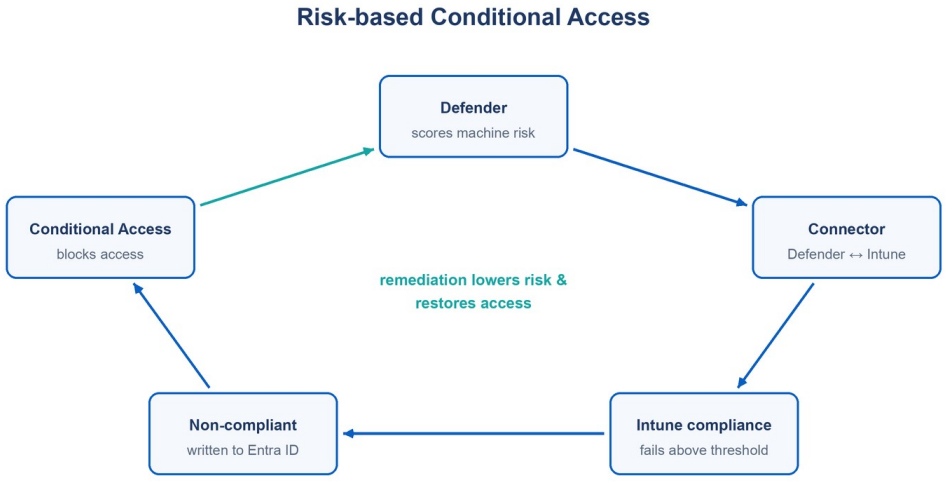

Figure 11.3: Risk-based Conditional Access — a live Defender risk signal becomes an automatic access decision, and remediation automatically restores access once the threat clears.

This is genuinely a self-defending endpoint, and it is buildable in an afternoon with components you already own. It is also the clearest example in the book of the whole thesis: data (Defender's risk signal) flowing through automation (the compliance-to-CA loop) to produce an outcome (access, gated in real time) that no team could produce by hand at fleet scale.

Automated Remediation for Security Posture

The loops above are enforcement — they take access away when something is wrong. But taking access away is a blunt response, and often what you actually want is to *fix the thing* so access never had to be lost. That is where the remediation-script pattern from Chapter 8 comes back, now pointed squarely at security posture. The idea is identical: a **detection** script decides whether a device has a problem, and a **remediation** script fixes it, both running on a schedule across the fleet. The only thing that changes is what you detect and fix.

The theme from Chapter 8 — "spot the pattern, then fix it before anyone files a ticket" — is even more valuable for security than it was for temp folders, because a security drift that you catch and fix in the detection window is a device that never went non-compliant, never lost access, and never generated a ticket or a support call. You are not reacting to the block; you are preventing it.

A concrete pair. This detects whether the Windows firewall is disabled on any profile, and remediates by turning it back on — exactly the sort of setting that would fail the compliance policy above and cost the user their access after the grace period:

```powershell
# Detection script — exit 1 if any firewall profile is off.
$off = Get-NetFirewallProfile | Where-Object { $_.Enabled -eq $false }
if ($off) {
    Write-Output "Firewall disabled on: $($off.Name -join ', ')"
    exit 1   # triggers remediation
}
Write-Output "All firewall profiles enabled."
exit 0
# Remediation script — enable all firewall profiles.
try {
    Set-NetFirewallProfile -Profile Domain,Public,Private -Enabled True -ErrorAction Stop
    Write-Output "Firewall re-enabled on all profiles."
    exit 0
} catch {
    Write-Output "Failed to enable firewall: $($_.Exception.Message)"
    exit 1
}
```

Deploy that pair under **Devices > Scripts and remediations**, run it in the SYSTEM context (a firewall change needs elevation, not a user session), and assign it to your Windows group on a daily schedule. Now a device whose firewall gets turned off — by a user with local admin, by some other software, by accident — is quietly fixed within a day, usually well inside the compliance grace period, and the user never notices anything happened. The same pattern covers a long list of posture fixes: re-enabling tamper protection, restarting a stopped Defender service, correcting a drifted registry key, clearing a condition that trips a baseline.

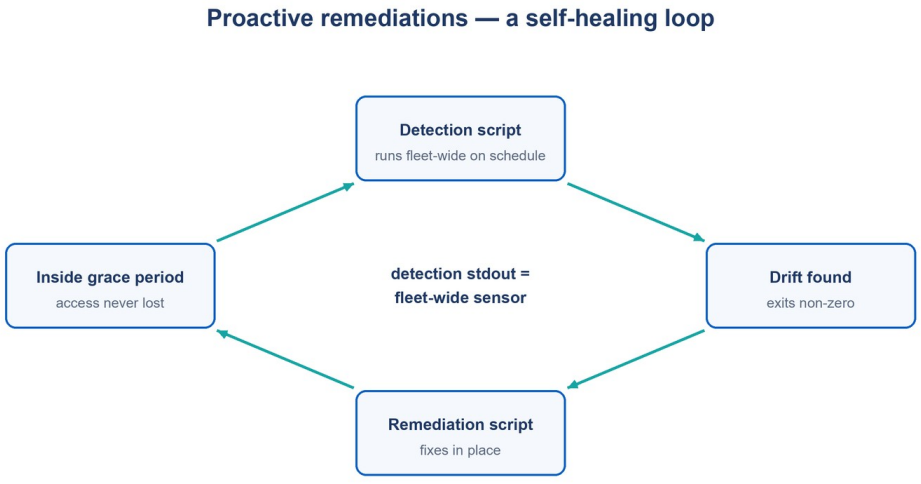

Figure 11.4: The automated-remediation loop for security posture — detect drift, fix it in place, and beat the compliance grace period so access is never lost.

The connection to the loops earlier in the chapter is what makes this powerful rather than just tidy. Detection and remediation scripts are the *proactive* layer that keeps devices out of the non-compliant state; the compliance-to-CA loop is the *reactive* backstop for when proactive fixing is not enough or the drift is something you cannot auto-fix. Together they form a graded response: fix it silently if you can, block access if you must.

And because the detection script's standard output is captured in the remediation report — the mechanism I showed in Chapter 8 — every one of these pairs is also a *sensor*. Deploy the firewall detection across the fleet and, before you even remediate anything, you have a fleet-wide report of exactly which devices had their firewall off and when. Spotting the pattern is the first half; the report is where you see it.

A Worked Example: One Device, End to End

Let me tie the whole chapter together with a single device's journey, because the individual pieces only impress when you watch them chain.

A new laptop arrives at a new hire's desk. She unboxes it, powers it on, connects to Wi-Fi, and signs in with her work account. That sign-in triggers **Autopilot** (or device preparation) — the device recognizes it belongs to the tenant, pulls its deployment profile, and begins a user-driven, Entra-joined enrollment. The **Enrollment Status Page** blocks her desktop while the required apps and profiles install, so she cannot start working on a half-built machine; she watches a progress bar for a few minutes instead of calling the help desk in twenty.

As enrollment completes, the device lands in its groups. A **dynamic group** rule — `deviceOSType eq "Windows"` and `deviceOwnership eq "Company"` — captures it automatically, and through that membership it inherits its **settings-catalog configuration** and its **security baseline**, hardened to the standard the whole fleet runs. A **filter** on the baseline assignment scopes a model-specific driver setting to her exact hardware without anyone creating a bespoke group for it. Simultaneously the **Defender for Endpoint** onboarding policy —

delivered as part of that same configuration set — enrolls the device into Defender, so it is defended before she has opened a single app.

Then the gates come up. A **compliance policy** evaluates the device: BitLocker on, firewall up, secure boot enabled, Defender risk under `low`, minimum build met. BitLocker is still encrypting in these first minutes, but the policy's **grace period** means the device is not slammed non-compliant while that finishes. Once encryption completes and every rule passes, Intune writes **compliant** to Entra ID. A **Conditional Access** policy requires exactly that — a compliant device — before it will issue tokens for corporate apps, so the instant she is compliant, she has access. Her first real action on the laptop is opening Outlook, exactly as it should be.

A month later, she plugs in a USB drive that carries something malicious. **Defender** flags it and raises the device's risk to medium. The connector carries that to Intune; the compliance policy fails the risk rule and writes **non-compliant** to Entra; the next time her laptop asks for a token, Conditional Access **blocks it**. She has lost access to corporate data automatically, within minutes of the threat appearing, with no analyst having yet looked at the alert. Meanwhile the security team's automated response (or an analyst) remediates the threat, Defender lowers the risk, and on the next evaluation the device flips back to compliant and her access returns — the device let itself back in once it was safe.

And underneath all of it, quietly, the **remediation scripts** run their daily rounds — checking that the firewall stayed on, that tamper protection is intact, that nothing drifted — fixing the small things inside the grace window so they never escalate into a block at all. No ticket was filed at any point in this story. That is the whole idea of this chapter: the from-box-to-secure-and-compliant journey, automated end to end, and self-correcting when reality pushes back.

Closing Part III: From Automation to Intelligence

Step back and look at what the automation half of this book assembled. You can get data out of Intune and act on it (Chapter 5). You can build custom solutions on Graph with keyless authentication (Chapter 8). You can manage configuration as code (Chapter 9), automate the

application lifecycle (Chapter 10), and — in this chapter — automate the entire provisioning, compliance, and security posture of a device from the moment it leaves its box. Every one of those chapters shares a single mechanic: a signal, a rule, and an automated consequence, running on a schedule so no human has to be in the loop for the common case.

But notice the shape of the rules in this chapter. Every one of them is something *you* had to know to write. You had to know that a firewall being off is bad, that risk above `low` should block, that this dynamic-group rule captures the right devices. The automation is tireless and it is fast, but it is not *smart* — it does exactly what you told it and not one thing more. It cannot notice a pattern you did not anticipate, explain why a device keeps failing, or write its own remediation for a problem you have not seen before.

That ceiling is where Part IV begins. Everything from here on is about adding intelligence on top of the automation — systems that can reason over the data you have learned to collect rather than just match rules against it. The next chapter, **Chapter 12, "Understanding Copilots and Large Language Models,"** starts at the foundation: what these models actually are, what *grounding* means, and why the data plumbing you built across Part III is precisely what makes them useful in an enterprise instead of a novelty. The automation you just built is what the intelligence will stand on. Let us go understand the intelligence.

Part IV

AI, Copilots, and Building Your Own

Chapter 12: Understanding Copilots and Large Language Models

Introduction

When I wrote this chapter for the first edition, “Copilot” was still a name you had to explain. Today it is a name you have to disambiguate. Microsoft has put the Copilot brand on so many products that the harder problem is no longer “what is a Copilot” but “*which* Copilot, and does it see my data or not.” That confusion is worth clearing up before we go anywhere near building something, because in the next chapters we lean on these tools directly — Security Copilot in Chapter 13, and your own grounded assistant later in the book.

So this chapter does two jobs. First, it draws an honest map of the Copilot landscape as it actually stands in 2026: what the word means, which products carry it, what they cost, and — the part people get wrong most often — which of them can reason over your organisation’s data and which are just a chat box with a nice logo. Several of the names in the first edition are now dead or renamed, and I have corrected all of them here. Second, it explains the machinery underneath: what a large language model is, what *grounding* is and why it is the single most important idea for anyone building on this technology in an enterprise, and how to write prompts that actually get you what you want. That last part — prompt engineering — is the most durable skill in this whole book, and I have kept the tutorial long and practical on purpose.

If you already know what an LLM is, skim the middle and spend your time on grounding and prompting. If you do not, start at the top. Either way, by the end of the chapter you should be able to look at any “Copilot” Microsoft ships and tell, in one sentence, what it is and whether it can help you.

What is a Copilot?

Strip away the marketing and a Copilot is a generative-AI assistant built *into* a product, grounded in the data that product already holds, that you interact with in natural language. Three parts of that definition matter, and each one separates a real Copilot from a novelty.

Built in. A Copilot is not a separate app you alt-tab to. It lives inside the tool you are already using — the Intune admin center, a Word document, the Windows taskbar, a Defender incident — so it has the context of what you are looking at. That embedding is what makes it a copilot rather than a search engine.

Grounded in data. The interesting Copilots are connected to real information: your emails and files, your tenant's devices and policies, your security incidents. They do not answer from a generic memory of the internet; they retrieve facts about *your* environment and reason over them. This is the difference between an assistant that says "here is how compliance policies generally work" and one that says "device H4939 is non-compliant because BitLocker is off, and here are the three others in the same state." We will come back to grounding in depth later, because it is the whole game.

Natural language. You ask in plain English (or German, or whatever), and it answers in plain language — or better, it *acts*: adjusts a setting, drafts a policy, triages an alert. The conversational surface is the least important part technically, but it is what makes the capability usable by someone who is not a data analyst.

Put those together and you have the pattern Microsoft has replicated across its entire portfolio: take a product, connect a model to that product's data, and expose it through a chat box or an inline suggestion. Once you see the pattern, the twenty-odd "Copilots" stop looking like twenty different products and start looking like one idea applied twenty times — with wildly varying maturity.

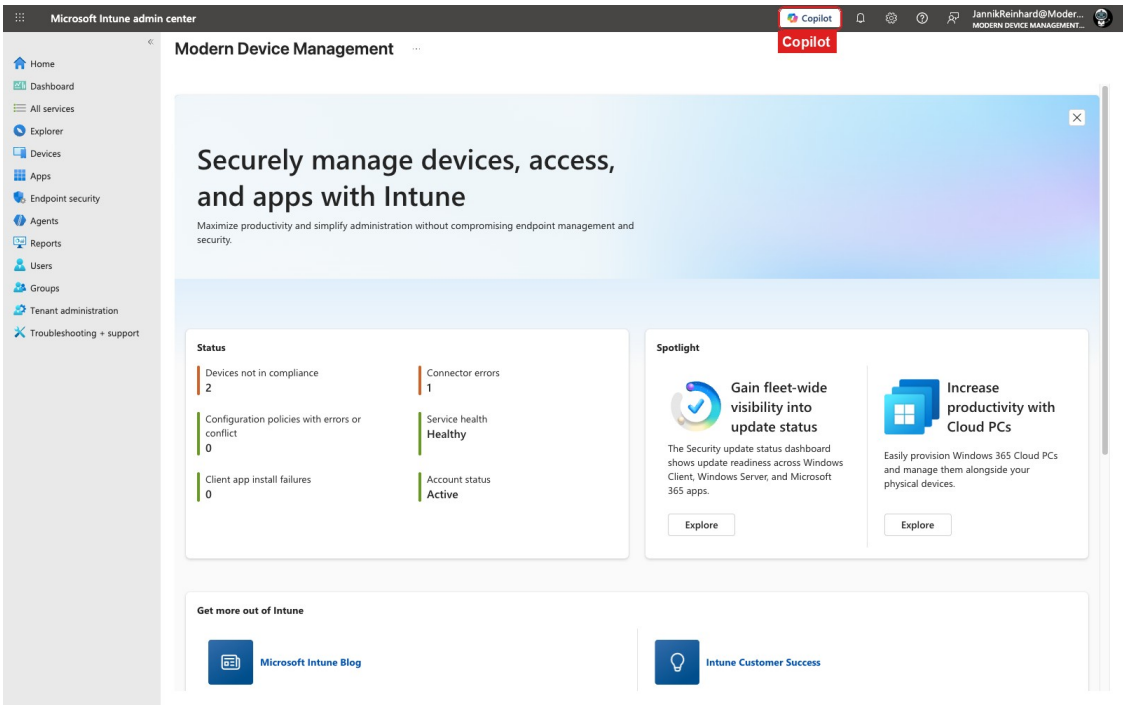

Figure 12.1: The Intune admin center, with the Copilot button in the header — the most visible sign of how far Copilot has spread into everyday admin tools

Why the adoption curve matters

It is worth remembering how fast this arrived, because the speed is the reason the landscape is such a mess of half-finished names. Mobile phones took roughly sixteen years to reach a hundred million users. Instagram did it in about two and a half years. ChatGPT, the product that put large language models in front of everyone, did it in about two months. When a technology moves that fast, product teams ship first and tidy the branding later — which is exactly what happened, and why half of this chapter is spent correcting names that changed between editions.

The takeaway is not “AI is amazing.” It is that this is a real platform shift on the scale of the web or mobile, and platform shifts reward the people who understand the plumbing rather than just the demo. That is what the second half of this chapter is for.

Which Copilots does Microsoft offer?

A lot. Too many to name a couple of years ago, and more now. Microsoft has renamed several of them since the first edition, retired a few, and added a genuinely new category — agents — that did not exist when I first wrote this. So rather than list everything, I will map the ones you are actually likely to meet, grouped by who they are for, and I will use the *current* names, because using the old ones will send you looking for products that no longer exist.

One correction up front, because it trips people up: the first edition said these Copilots were "powered by GPT-4o." That was true then. It is not the current picture. Microsoft's Copilots now run on a mix of newer models — the GPT-5 family and GPT-4.1 are the current workhorses across most experiences, with GPT-4o now the *older* option rather than the flagship. Microsoft deliberately abstracts this away; you rarely choose the model, and it changes underneath you. Do not anchor on a specific model version when you talk about a Copilot, because it will be wrong by the next quarter.

Microsoft 365 Copilot (the productivity one)

This is the one most people mean when they say "Copilot" at work. **Note the name carefully: it is Microsoft 365 Copilot.** It was called *Copilot for Microsoft 365* until January 2025, when Microsoft flipped the word order. If you see "Copilot for Microsoft 365" in a document, that document is out of date.

Microsoft 365 Copilot is embedded across Word, Excel, PowerPoint, Outlook, Teams, and the rest of the suite. It drafts and rewrites in Word, builds and analyses in Excel, turns a prompt into slides in PowerPoint, summarises threads and drafts replies in Outlook, and recaps meetings in Teams. What makes it more than a writing toy is that it is grounded in the **Microsoft Graph** — your emails, files, chats, meetings, and calendar — through a component Microsoft calls the semantic index. So "summarise where the Contoso project stands" pulls from your actual documents and conversations, not a generic template.

The pricing is the number everyone asks about: **$30 per user per month**, on an annual commitment, and it sits *on top of* a qualifying Microsoft 365 base licence (E3, E5, Business Standard, or Business Premium). It is an add-on, not a plan you buy on its own.

Two things changed here since the first edition, and they matter:

- **Microsoft 365 Copilot Business** is a newer SKU aimed at small and mid-sized businesses at roughly **$21 per user per month** (Microsoft has run a promotional price around $18 through late 2026 — treat any exact promo number as something to check at purchase time). It is the "we are not a 5,000-seat enterprise but we still want this" tier.
- **Microsoft 365 Copilot Chat** is the **free tier**, launched in January 2025. This is important and widely misunderstood. Copilot Chat is web-grounded chat with enterprise data protection — but it does **not** reach into your organisation's internal data the way the paid Microsoft 365 Copilot does. It is the descendant of what used to be called **Bing Chat Enterprise**, a name that is now **retired** — if you are still saying "Bing Chat Enterprise," that product has been renamed and folded into Copilot Chat. The mental model: Copilot Chat is a safe, free chatbot; Microsoft 365 Copilot is the paid, data-grounded assistant.

Microsoft Copilot (the consumer one)

Confusingly close in name, this is the **consumer** assistant at copilot.microsoft.com and in the Copilot app. It was formerly **Bing Chat**, renamed to Microsoft Copilot in late 2023. It is a general-purpose chat and image-generation assistant for personal use. There is a paid tier, **Copilot Pro**, at roughly $20 per month that adds priority access to newer models and Copilot inside the consumer Office apps. (Consumer pricing and the exact Pro feature set move around; verify before quoting it.) For our purposes this one is background — it is not a device-management tool — but you need to know it exists so you do not confuse "Microsoft Copilot" (consumer) with "Microsoft 365 Copilot" (work). The near-identical names are Microsoft's own doing, and they are a genuine source of confusion.

Copilot in Windows and Copilot in Edge

Copilot in Windows puts the assistant into the operating system itself — a pane you open from the taskbar that can answer questions, summarise what is on screen, and increasingly take actions on the PC. **Copilot in Edge** does the same inside the browser, with the added trick of being able to read and summarise the page or document you are looking at. These are consumer-and-prosumer conveniences rather than management tools, but they are the most *visible* Copilots because they ship with the platform, and they are often the first thing your end users will ask you about.

GitHub Copilot (the developer one)

The original Copilot, and still the best example of the pattern working. **GitHub Copilot** is an AI pair programmer built into editors like Visual Studio Code and Visual Studio (and, increasingly, into GitHub itself as an agent that can take an issue and open a pull request). It autocompletes code, writes whole functions from a comment, explains unfamiliar code, and writes tests, across essentially every mainstream language. It matters to us for two reasons: much of the automation you will build in later chapters — Graph scripts, remediations, Logic Apps — is faster to write with it, and it is the cleanest illustration of grounding, because its suggestions are shaped by the file you are actually editing.

Microsoft Security Copilot (the security one)

Microsoft's security-focused assistant, previously called **Copilot for Security**, now **Microsoft Security Copilot**. It helps analysts investigate incidents, summarise alerts, correlate threat intelligence, and — this is the current direction — run **agents** that take on whole tasks like phishing triage. It is billed on its own consumption model (Security Compute Units) rather than per user, and it is embedded across the security stack: Defender, Entra, Purview, Sentinel, and Intune. This is the subject of the entire next chapter, so I will not go deep here beyond flagging the name change and the shift toward agents.

Copilot in Intune

The one closest to this book's heart. **Copilot in Intune** is the Security Copilot engine surfaced *inside the Intune admin center* — the button in the corner of every screenshot in Chapter 2. It reasons over your tenant: it explains what a policy or setting does and what breaks if you change it, helps you build and interpret **Device Query** results, and summarises device and policy state so you are not clicking through six blades to assemble a picture by hand. It went generally available in 2025. Under the covers it consumes the same Security Compute Units as Security Copilot, which is why Microsoft's branding here — "Microsoft Copilot in Intune" — points at the same engine. We use it in Part II and Part IV.

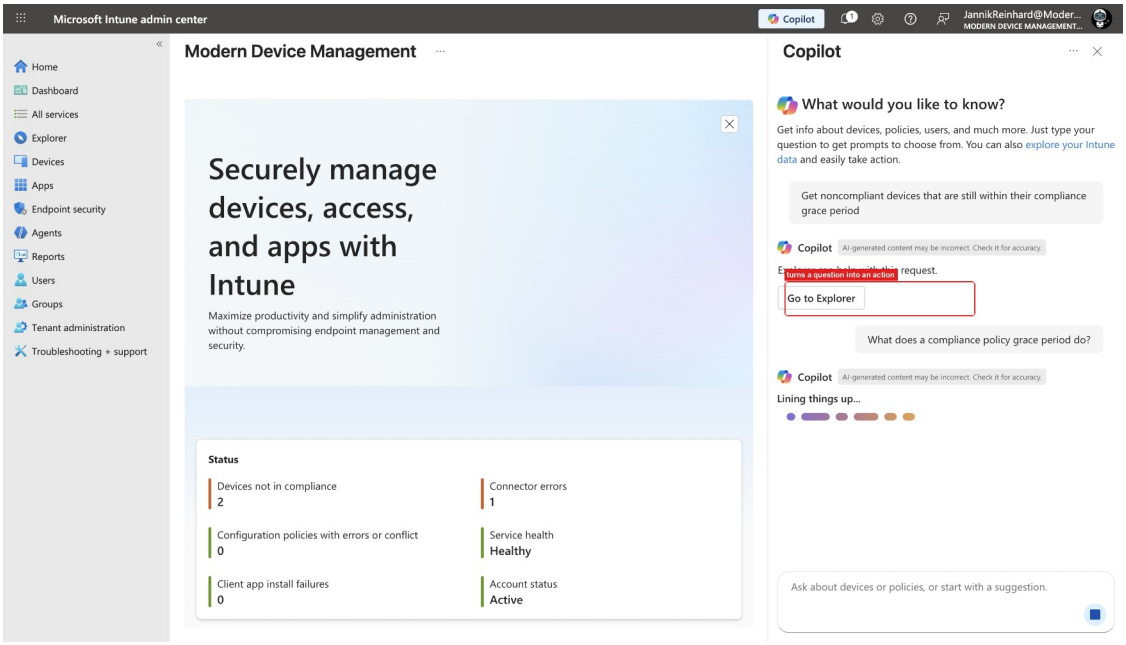

Figure 12.2: Copilot in Intune answering in plain language, grounded in your own tenant data rather than a generic model

Microsoft 365 Copilot for Sales, Service, and Finance (the line-of-business ones)

These are the role-specific business Copilots, and their names changed in **September 2025**. What used to be *Copilot for Sales* and *Copilot for Service* are now **Microsoft 365 Copilot for Sales** and **Microsoft 365 Copilot for Service**, joined by **Microsoft 365 Copilot for Finance**. They

bring CRM and line-of-business context (Dynamics 365, Salesforce, and the like) into the flow of Outlook, Teams, and the Office apps — drafting customer emails grounded in CRM records, summarising cases, and so on. The old standalone Sales SKU was retired; these capabilities are increasingly delivered to Microsoft 365 Copilot customers through agents rather than sold as separate products. If your first-edition notes mention "Dynamics 365 Copilot" as a single thing, this is where that story went — it fragmented into role-specific, agent-delivered experiences under the Microsoft 365 Copilot umbrella.

Copilot Studio and the Agent Store

Here is the category that did not really exist, in its current form, when I first wrote this. **Copilot Studio** is the low-code environment for building your *own* copilots and agents — connect a model to your data sources, define what it can do, publish it into Teams or a website or the Microsoft 365 Copilot experience. It is the successor to Power Virtual Agents, and it is the low-code path to the same "grounded assistant" idea we build the pro-code way later in this book. Alongside it, the **Microsoft 365 Copilot Agent Store** is where you discover, and in some cases buy, prebuilt agents — first-party and partner-built — and drop them into Copilot. Think of it as an app store for task-focused AI.

The economics here also changed. Since September 2025 Microsoft meters agent usage with **Copilot Credits** — a consumption model (roughly a cent per credit pay-as-you-go, or bought in packs) that charges for agent and automation actions, while licensed Microsoft 365 Copilot users get a bundle of agent actions included. The point for you is not the exact rate; it is that agents cost money to *run*, not just to build, and that cost is metered by usage. Budget accordingly.

Microsoft Agent 365 (the newest entrant)

The most recent arrival, and the one to watch. **Microsoft Agent 365** reached general availability on **May 1, 2026**. Where Copilot Studio lets you *build* agents, Agent 365 is about *managing* them at scale — treating AI agents as a class of digital identity that has to be registered, governed, secured, and monitored just like a user account. It is priced around **$15 per user per month**,

and it is included with **Microsoft 365 E7**, the newer top-tier suite. (This is new enough that you should confirm the exact price and inclusion against Microsoft Learn before you quote it in a proposal.) The reason it matters to a device-management audience: once agents can act on your tenant, "who governs the agents" becomes the same kind of problem "who governs the devices" already is, and Agent 365 is Microsoft's answer to it. We are entering the phase where the thing you manage is not only the endpoint but the agent acting through it.

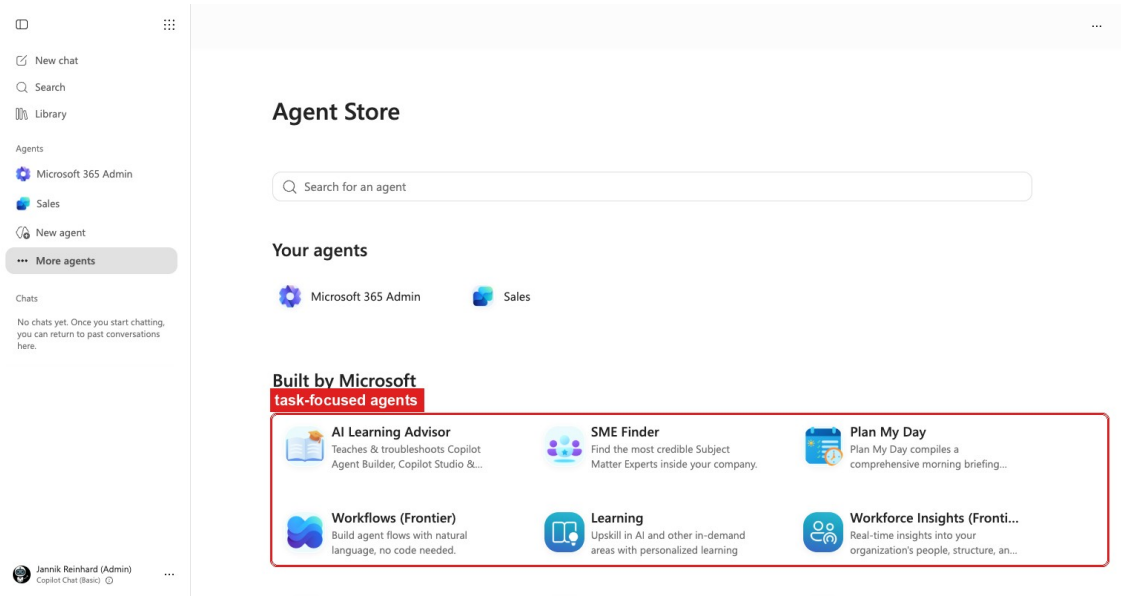

Figure 12.3: The Microsoft 365 Copilot Agent Store — a gallery of task-focused agents (built by Microsoft and by your own organization) that you add alongside chat, the clearest sign of the shift from a single assistant to many specialized agents

The one-sentence map

If you remember nothing else from this section, remember the split:

- **Grounded in your org's data (the powerful ones):** Microsoft 365 Copilot, Copilot in Intune, Security Copilot, the Sales/Service/Finance Copilots, and anything you build in Copilot Studio.
- **Web-grounded or general purpose (the safe-but-limited ones):** Microsoft 365 Copilot Chat (free), Microsoft Copilot (consumer), Copilot in Windows/Edge.

- **Developer:** GitHub Copilot.
- **Build and govern the next layer:** Copilot Studio (build), Agent Store (discover), Agent 365 (govern).

Everything else is a variation on those. Now let us look at what is actually running inside all of them.

The engine underneath: Large Language Models

Every Copilot on that map is a wrapper around the same core technology: a **large language model**. If you want to use these tools well — and especially if you want to *build* with them, which is where this book is going — you need a working mental model of what an LLM is, what it is good at, and where it fails. You do not need the mathematics. You need the intuition.

What a large language model actually does

At its most honest, a large language model does one thing: given a stretch of text, it predicts what text most plausibly comes next. That is it. Everything else — the essays, the code, the policy summaries — is that one capability, applied at enormous scale, over and over, one chunk of text at a time.

The chunks are called **tokens**. A token is roughly a word or a piece of a word; “Intune” might be one token, “non-compliant” might be three. The model reads your input as a sequence of tokens and generates its answer one token at a time, each time picking a likely next token given everything so far, appending it, and repeating. When you watch a Copilot “type” its answer word by word, you are literally watching this loop run. This is why LLMs are sometimes described, a little unkindly and a little accurately, as very sophisticated autocomplete.

The reason it is more than autocomplete is *scale* and *architecture*. These models are trained on enormous quantities of text — a large fraction of the public internet, plus books, code, and articles — and in doing so they absorb not just vocabulary but grammar, facts, reasoning patterns, and the structure of countless kinds of documents. The size of a model is usually

described by its number of **parameters**, the internal values it tunes during training. To give a sense of the trajectory: GPT-2 had around 1.5 billion parameters; GPT-3 jumped to 175 billion; the models Microsoft's Copilots run on today — the GPT-5 family and GPT-4.1 — are larger and, more importantly, substantially more capable, though the exact figures are no longer published and parameter count has stopped being a useful headline number. What changed is less "how big" and more "how well trained and how well tuned for reasoning."

The transformer, in one paragraph

The breakthrough that made all of this possible is the **transformer**, introduced in the 2017 paper *"Attention Is All You Need."* Its key idea is **attention**: when the model processes a word, it can weigh how much every *other* word in the context matters to it, all at once, rather than reading strictly left to right and forgetting the start of a long sentence by the time it reaches the end. In "the device that failed its compliance check last week is back online," attention is what lets the model connect "is back online" to "the device" across all the words in between. Stack that mechanism into many layers and you get a model that tracks context, resolves ambiguity, and holds a thread across long passages. That contextual understanding — considering the whole input together rather than word by word in isolation — is why LLM output reads as coherent and human-like instead of like a Markov-chain word salad. You do not need to know how attention is implemented. You need to know that context is the thing the model is best at, which is exactly why *how you frame the context in your prompt* is the lever that controls the output.

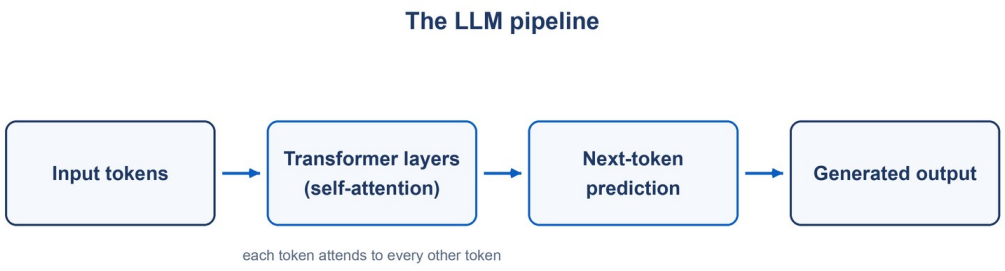

Figure 12.4: A simplified view of the LLM pipeline: input tokens flow through transformer layers, where attention lets each token weigh every other, to produce the next token, one at a time

Why this matters for device management (and where it fails)

Two consequences fall straight out of this design, and both shape everything we build later.

First, an LLM knows a lot but knows nothing about *you*. Its training data is a snapshot of the public world up to some cutoff date. It has never seen your tenant, your device names, your compliance policies, or last night's alerts. Ask a raw model "why is device H4939 non-compliant" and it cannot know — it will either say so or, worse, invent a plausible-sounding answer.

Second, it can be confidently wrong. Because the model generates the *most plausible* next token, not the *most true* one, it will sometimes produce fluent, confident, completely fabricated information. This is called **hallucination**, and it is not a bug that will be patched away — it is a direct consequence of how the thing works. A classic example: ask an ungrounded model how to create a configuration profile in Intune and it may confidently reference the "Endpoint Manager admin center," a name Microsoft retired years ago, because that name appeared often in its training data. The model is not lying; it has no concept of true and false, only of likely.

Both problems have the same solution, and it is the most important concept in this chapter.

Grounding: connecting the model to trusted data

Grounding is the practice of giving the model the specific, current, trusted information it needs *at the moment you ask*, so that its answer is based on real facts about your world rather than on its frozen training data. Grounding is what turns a clever text predictor into a useful enterprise assistant. It is the difference between "here is how compliance generally works" and "device H4939 is non-compliant because BitLocker is disabled."

The dominant technique for grounding is **Retrieval-Augmented Generation**, or **RAG**, and the idea is simpler than the acronym. When a question comes in, the system first *retrieves* relevant information from a trusted source — your documents, a database, Microsoft Graph, a live API call — and then hands that retrieved information to the model *along with* the question, effectively saying: "Here are the relevant facts. Now answer using these." The model still does its next-token prediction, but now it is predicting an answer conditioned on real, current data, so the output is accurate and specific instead of generic and possibly invented.

This is exactly how the grounded Copilots work. Microsoft 365 Copilot retrieves from the Graph before it answers. Copilot in Intune retrieves your device and policy data before it answers. When you build your own assistant later in this book, you will build a RAG pipeline: pull the right data from Graph, feed it to the model, get a grounded answer. Grounding is not an advanced topic you will get to someday — it is the foundation of every serious thing you can do with this technology in an enterprise.

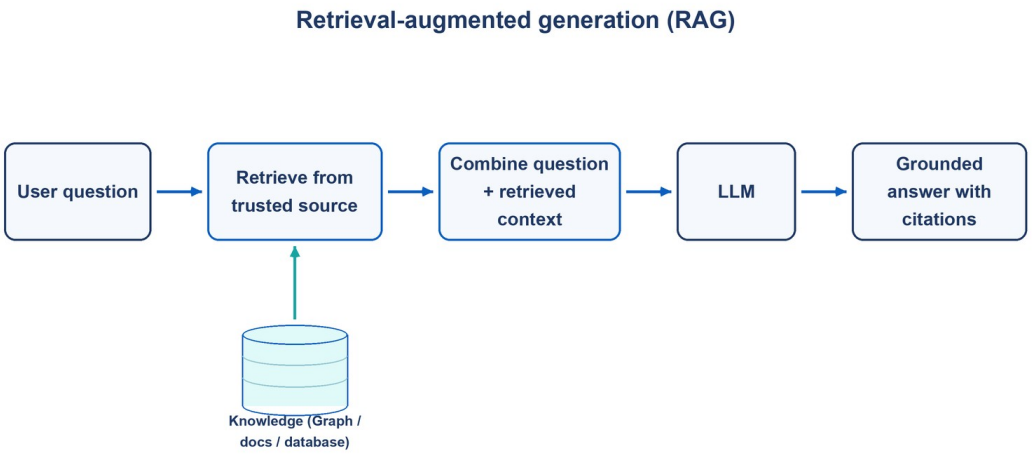

Figure 12.5: Retrieval-augmented generation: the question is used to retrieve relevant context from a trusted source, which is combined with the prompt so the model answers from your data, with citations

Grounding versus fine-tuning versus training your own

People new to this often reach for the wrong tool, so it is worth being clear about the hierarchy of options for customising a model to your needs. There are three, in increasing order of cost and effort — and you should almost always try them in that order.

1. **Grounding and prompt engineering.** For the large majority of use cases, this is the answer. You feed the model the right context (grounding) and you ask well (prompt engineering). It is fast, cheap, requires no model training, and adapts instantly when your data changes — because the data is retrieved fresh every time. Start here. Most people never need to go further.

2. **Fine-tuning.** Fine-tuning means taking a pre-trained model and training it *further* on a specialised dataset so it internalises a particular style, format, or domain. It is genuinely useful for narrow, stable tasks — a consistent tone, a rigid output format, a specialised classification — but it is expensive, slow, and it bakes the knowledge in at training time, so it goes stale the moment your underlying data changes. Crucially, fine-tuning is the wrong tool for teaching a model *facts*: if you want the model to know your current device inventory, you ground it, you do not fine-tune it, because the inventory changes hourly. Fine-tuning changes *behaviour*, grounding provides *knowledge*. Reach for it only when grounding and prompting genuinely cannot get you there.

3. **Training a new model from scratch.** For all but a handful of organisations on Earth, this is the wrong answer. It demands enormous datasets, enormous compute, and enormous ongoing maintenance, and the frontier labs will out-run whatever you build. Mentioned only for completeness.

The table below sums up the two you will actually choose between:

Criteria	Grounding (RAG)	Fine-tuning
What it does	Supplies fresh, external facts to the prompt at query time	Adjusts the model's own weights on a specialised dataset
Best for	Knowledge — current, changing, org-specific data	Behaviour — consistent style, tone, or output format

Criteria	Grounding (RAG)	Fine-tuning
Data requirement	Low; uses existing data as-is, retrieved on demand	High; needs a curated training dataset
Stays current?	Yes — data is retrieved fresh each time	No — knowledge is frozen at training time
Cost and effort	Low; no training, quick to stand up	High; compute, time, and expertise to train and re-train
When data changes	Nothing to do; next retrieval sees it	Must re-train
Use in this book	The default; how we build our assistant	Rarely; only for narrow, stable behavioural needs

Keep that hierarchy in your head every time someone proposes “training an AI on our data.” Nine times out of ten they mean grounding, and grounding is the cheaper, better, more maintainable answer.

Prompt engineering: a practical tutorial

Grounding gives the model the right facts. **Prompt engineering** is how you ask well enough to get the right answer out of those facts. It is the highest-leverage skill in this entire book, because it is the one you use every single day regardless of which Copilot you touch, and unlike everything else in AI it has barely changed — a well-crafted prompt from two years ago still works. The models got better; the craft of asking got *more* rewarding, not obsolete. So I have kept this tutorial long and hands-on. Work through it.

A prompt is simply the instruction you give the model. The gap between a lazy prompt and a good one is enormous. Compare:

- **Lazy:** “Write an article about AI.”
- **Good:** “Write a 500-word article on the impact of AI on device management, focused on how grounded assistants change day-to-day work for an IT admin. Use a practical, first-person tone and end with one concrete example.”

Same model, wildly different output. The second prompt controls length, topic, angle, tone, and structure. That control is the whole skill.

The anatomy of a strong prompt

Most good prompts contain some combination of four ingredients. You will not always need all four, but knowing them gives you a checklist when an answer disappoints you.

- **Instruction** — what you want the model to *do*. (“Summarise,” “classify,” “draft,” “compare.”)
- **Context** — the background it needs. (“These are the devices that failed last night’s compliance run.”) This is where grounding data lives.
- **Input** — the specific thing to act on. (The dataset, the device ID, the text to classify.)
- **Output format** — how you want the answer shaped. (“As a Markdown table,” “as JSON,” “in five bullet points.”)

Sitting above all of these is the **system prompt** — an instruction that sets the model’s role and rules for the whole conversation, separate from your individual questions. In a Copilot you often do not see the system prompt; the product author wrote it. When you build your own assistant, *you* write it, and it is one of the most powerful controls you have.

A tight worked example, putting a system prompt and a user prompt together:

- **System prompt:** “You are an assistant that classifies text sentiment as ‘friendly’, ‘negative’, or ‘neutral’. Always respond only with valid JSON.”
- **User prompt:** “Text: ‘I absolutely love writing prompts!’ Classify the sentiment.”
- **Desired output:**

```
{
  "text": "I absolutely love writing prompts!",
  "classification": "friendly"
}
```

Notice how the system prompt fixed the *behaviour* (what the model does and the exact output shape) while the user prompt supplied the *content*. That separation is the backbone of every real application you will build.

Foundational habits

Before the named techniques, three habits that improve almost every prompt you will ever write:

Be precise, not verbose. The model does better with direct instructions than with padding. “Summarise the key compliance risks in this report” beats “Could you maybe, if it’s not too much trouble, take a look and tell me what’s important here?” — the politeness adds tokens and ambiguity, nothing else.

Give context. “As an IT administrator explaining to a non-technical manager, describe why device compliance matters” produces a far more useful answer than “explain device compliance,” because you have told the model *who is asking and for whom*.

Ask for structure. If you want a list, ask for a list. If you want a table, name the columns. “List the top five reasons a Windows device fails an Intune compliance policy, as a numbered list, each with a one-line fix” gets you something you can act on, not a wall of prose.

The core techniques

Here are the techniques worth having in your hands, each with an example rooted in our world of Intune and device management.

Zero-shot prompting

You ask the model to do something with no examples, relying entirely on its trained knowledge. This is the default and it is often enough for straightforward tasks.

- **Prompt:** “Explain what a Conditional Access policy does in Microsoft Entra, in three sentences, for a helpdesk technician.”
- The model answers directly. Zero-shot is your starting point; reach for the others when it falls short.

Few-shot prompting

You give the model a handful of examples of the input-output pattern you want *before* your real request. This is the single most reliable way to lock in a specific format or style, because you are showing rather than telling.

- **Prompt:** "Rewrite each device status as a one-line alert. Example — Input: 'Device DESKTOP-01, BitLocker off.' Output: '⚠️ DESKTOP-01 is non-compliant: enable BitLocker.' Example — Input: 'Device LAPTOP-07, OS out of date.' Output: '⚠️ LAPTOP-07 is non-compliant: update the OS.' Now do: Input: 'Device TABLET-03, no PIN set.'"
- **Response:** "⚠️ TABLET-03 is non-compliant: set a device PIN."

The model inferred the exact format — emoji, structure, wording — from your two examples. Few-shot is invaluable when you are processing many items and need every output to look the same.

Role (persona) prompting

You tell the model *who to be*, which shapes vocabulary, depth, and focus to a specific audience. This is one of the most useful techniques for device management because the same question needs a very different answer depending on who is asking. Watch how the persona changes everything while the underlying topic stays fixed:

- **As a SOC analyst:** "As a security operations analyst, summarise the top threats detected in the last 24 hours with severity and affected systems." → a terse, prioritised, technical incident summary.
- **As a developer:** "As a software developer, explain how to enrol a device via the Microsoft Graph `deviceManagement` API, including authentication." → OAuth flow, endpoints, error handling.

- **As an IT admin:** “As an Intune administrator, give a step-by-step guide to create a compliance policy that requires BitLocker on Windows.” → numbered admin-center steps.
- **As a helpdesk agent:** “As a helpdesk technician, walk a user through why their device shows as non-compliant and how to fix it.” → plain, reassuring, step-by-step.
- **As an end user:** “Explain in simple terms how to enrol my work laptop in the Company Portal.” → friendly, jargon-free.

Same domain, five different answers, each fit for its reader. When you build a Copilot for a specific team later, the persona goes in the system prompt and shapes every response automatically.

Chain-of-thought prompting

For anything that needs reasoning or multiple steps, ask the model to *think it through step by step* rather than jumping to an answer. This visibly improves accuracy on logical and multi-step tasks, and it makes the model’s reasoning auditable — you can see where it went wrong.

- **Prompt:** “A Windows device is non-compliant. Walk through, step by step, how you would diagnose the cause in Intune and what you would check at each step, then give the most likely fix.”
- **Response:** a numbered diagnostic path — check which compliance policy applies, check which setting failed, check the device’s last check-in, check for a conflicting policy — ending in a specific recommendation.

The instruction “step by step” is the whole trick. Note that the newest reasoning-oriented models do a lot of this internally now, but making it explicit still helps, and it still makes the output easier to trust because you can follow the logic.

Structured-output prompting

When the answer needs to feed into something else — a script, a report, a ticket — demand a machine-readable format. This is where prompting meets automation, and it is how you turn an LLM into a component in a pipeline rather than a chat toy.

- **Prompt:** "List three common Intune compliance failures. Respond only with a JSON array of objects, each with keys `failure`, `cause`, and `fix`. No prose."
- **Response:** clean JSON your code can parse directly.

Asking for JSON, CSV, or a Markdown table is what lets you drop a model's output straight into a Logic App, a remediation script, or a dashboard. We rely on this constantly in the build chapters.

Grounding in the prompt

Everything from the grounding section, applied by hand: paste the relevant facts *into* the prompt so the model answers from them rather than from memory. Even without a full RAG pipeline, you can ground a single question this way.

- **Prompt:** "Here is the compliance record for device H4939: `{ BitLocker: off, OS: current, PIN: set, lastCheckIn: 2h ago }`. Explain in one sentence why it is non-compliant and give the fix."
- **Response:** "Device H4939 is non-compliant because BitLocker is disabled — enable BitLocker encryption to bring it back into compliance."

The model did not guess; it read the record you gave it. This tiny move — supply the facts, then ask — is grounding in miniature, and it is the mental model behind every grounded Copilot.

Guardrails and constraints

Tell the model what *not* to do and how to behave at the edges. This matters enormously the moment a prompt is running in production where you cannot see every input, because it is your defence against confident nonsense.

- **Prompt:** "Answer the user's question using *only* the device data provided below. If the answer is not in the data, reply exactly: 'I don't have that information.' Do not guess or use outside knowledge."
- This is one of the most valuable single instructions in enterprise prompting: it turns hallucination from a likely failure into an explicit "I don't know," which is almost always the safer outcome. When we build the assistant, guardrails like this go in the system prompt and stay there.

Iterative (multi-turn) prompting

You rarely nail a complex task in one shot, and you do not have to. Treat it as a conversation: get a first answer, then refine.

- **You:** "Explain how Intune deploys apps."
- **Model:** a general overview of required vs. available assignments.
- **You:** "Now focus only on how required apps behave on a device that is offline at assignment time."
- **Model:** a targeted answer building on the first.

The model keeps the earlier turns as context, so each follow-up sharpens the result. Good prompting is often good *conversation* — start broad, then narrow.

Putting it together

The techniques are not mutually exclusive; the best real-world prompts stack several. A production prompt for a device-triage assistant might set a **persona** ("you are an Intune

administrator"), impose a **guardrail** ("use only the data provided, otherwise say you don't know"), demand **structured output** ("respond as JSON"), and be **grounded** with the retrieved device record — all at once. That combined prompt, sitting in a system prompt with live data retrieved into it, *is* the assistant we build later in the book. Prompt engineering is not a party trick; it is the programming language of this new layer of software, and the examples above are the vocabulary you will write it in.

The habit that makes you good at it is the same one that makes you good at everything else in this book: try, look honestly at what came back, adjust, and try again. The models will keep changing underneath you. The discipline of asking clearly will not.

Where this goes next

You now have the map and the machinery. You know what a Copilot is, which ones Microsoft actually ships under that increasingly crowded brand and what they cost, and — most importantly — which of them can reason over your data and which cannot. You know that underneath every one of them is a large language model that predicts text, that it knows nothing about your tenant until you *ground* it, and that the way you ask — prompt engineering — is the lever that turns all of it into useful work.

The next chapter takes the most operationally important of these Copilots for our world — **Microsoft Security Copilot** — and goes deep: what it is now, how its consumption-based pricing works, how it is embedded across Defender, Entra, Purview, and Intune, and how it has shifted from static plugins toward **agents** that take on whole tasks. Everything you just learned about grounding and prompting is exactly what makes Security Copilot useful, because it is those ideas wired directly into your security and device data.

And later, in the build chapters, we stop using other people's Copilots and make our own — a grounded assistant over Microsoft Graph, applying the RAG pattern and the prompting techniques from this chapter to answer questions about *your* fleet. This chapter was the theory. The rest of the book is where we put it to work.

Chapter 13: Delving into Microsoft Security Copilot

When I wrote this chapter the first time, Security Copilot was a young product with a simple story: a chat box, a handful of plugins, and a per-hour bill. That story is out of date. The product still exists under the same name and still runs on the same billing unit, but almost everything around it has moved. Plugins gave way to **agents** — units of AI that take on a task and run it rather than answer a single question. Microsoft added an **agent marketplace**. The pricing model gained an overage tier and, more importantly for most readers, Security Copilot is now **included with Microsoft 365 E5 and E7** instead of being a separate purchase every time. And the embedded experiences — Copilot living inside Defender, Entra, Purview, Sentinel, and, the reason this chapter is in an Intune book, **inside the Intune admin center** — went generally available.

So this is one of the chapters I largely rewrote. I kept the parts that are still true (what Security Copilot is, how the orchestration works, why RBAC matters) and replaced the parts that aged badly (the old single-SCU pricing math, the deprecated ARM template, the plugins-only framing). Throughout, I keep pulling the lens back to the endpoint: what does Security Copilot actually do for someone who manages devices, and where do the new Intune agents fit? That is where the value is for us, and it is also where the product has changed the most.

What is Microsoft Security Copilot?

Microsoft Security Copilot is a generative-AI assistant for security work. It takes natural-language prompts, grounds them in your own security data across the Microsoft stack, reasons over that data with a large language model, and comes back with summaries, next steps, queries, and — increasingly — actions it can take on your behalf. The point is speed: turn “read forty alerts, correlate them, and write up what happened” from an afternoon of analyst time into a few minutes of guided work.

It comes in two shapes, and you will use both:

- **The standalone portal** at securitycopilot.microsoft.com. This is the open-canvas experience — a prompt box, session history, promptbooks, and the agent surface. Use it for cross-product investigations, ad-hoc questions, and running or building agents.
- **Embedded experiences** inside the products you already work in. As of mid-2026 that list includes **Microsoft Defender XDR, Microsoft Entra, Microsoft Intune, Microsoft Purview, Microsoft Sentinel, Microsoft Defender for Cloud, Microsoft Defender Threat Intelligence, and Azure Firewall.** Here Copilot shows up as a panel or a button inside the native UI, with prompts scoped to whatever you are looking at.

One naming point that trips people up: when you see **"Microsoft Copilot in Intune"** or **"Copilot in Entra,"** that is the same Security Copilot engine, drawing on the same capacity, and it **consumes the same billing units** (more on those in a moment). It is not a separate free assistant. The Copilot *in Intune* went generally available in **July 2025**, Copilot *in Entra* in **July 2025**, and the Sentinel embedding reached GA in **September 2025**.

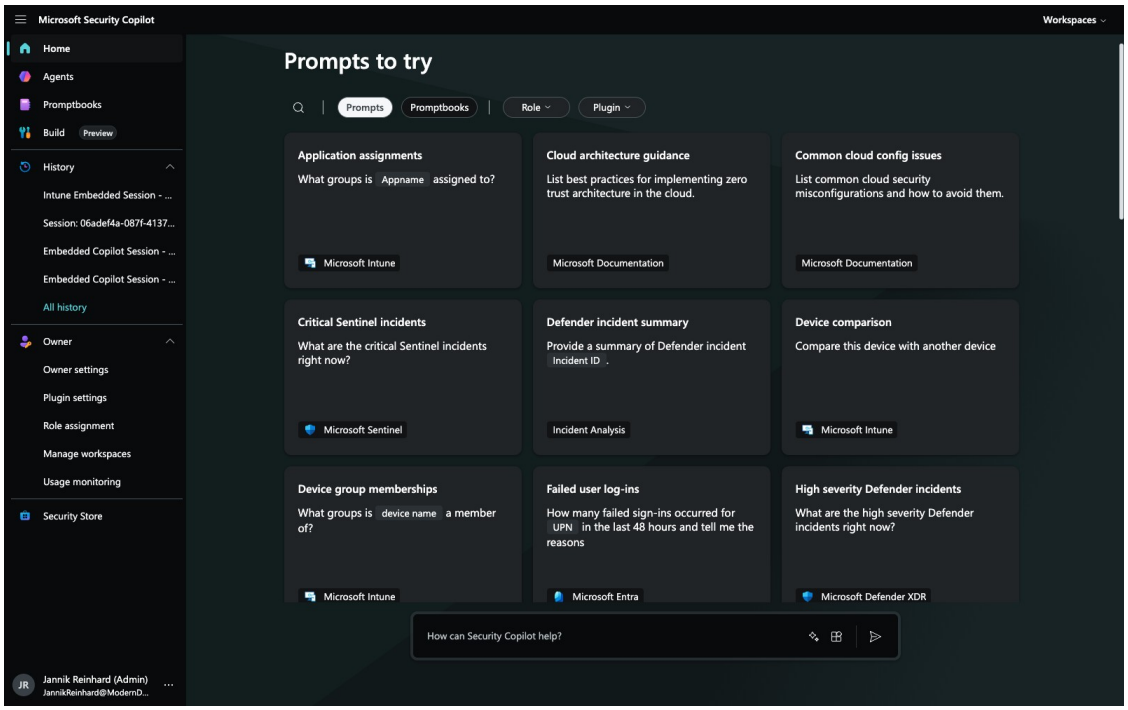

Figure 13.1: The standalone Security Copilot portal. One prompt box, the left navigation (Home, Agents, Promptbooks, Build), and the session history — this is the surface analysts work in directly

Products that integrate with Security Copilot

Security Copilot is deliberately a hub, not an island. The embedded surfaces above each contribute their own data and their own scoped actions, and the standalone portal can reach across all of them in a single session. For a device administrator, the ones that matter most are:

- **Microsoft Intune** — device, app, and compliance context. Summarize a device, explain a policy, compare configurations, and (with the Intune agents) triage changes and remediate vulnerabilities. This is our home turf and I spend the rest of the chapter here.
- **Microsoft Defender XDR** — endpoint, identity, email, and cloud-app detections. When a device you manage in Intune gets flagged, this is where the incident lives, and Copilot can pull the Intune device summary straight into the Defender investigation.
- **Microsoft Entra** — identity, Conditional Access, and risk. Device compliance is a Conditional Access signal, so the Entra and Intune stories are joined at the hip.
- **Microsoft Purview** — data security, insider risk, and DLP.
- **Microsoft Sentinel** and **Defender for Cloud** — SIEM/SOAR and cloud posture, for the broader SOC context around an endpoint incident.

The extensibility also reaches outside Microsoft: partner integrations (ServiceNow, and others through the marketplace I describe below) let Copilot pull in ticketing, third-party threat intel, and non-Microsoft security tools.

How Security Copilot works

Underneath the chat box, Security Copilot is an **orchestrator**. It does not just forward your prompt to a model; it plans the work, gathers the right context, calls the model, checks the output, and returns something actionable. Here is the loop, step by step:

1. **Prompt.** You ask a question or issue a command, either in the standalone portal or from an embedded surface. In Intune that might be “Summarize this device” from a device blade; in the portal it might be a full multi-sentence investigation request.
2. **Planning and grounding.** The orchestrator works out which capabilities and data sources are needed and pulls in the relevant context — device records, alerts, policies, sign-in logs, threat intelligence. Historically this was done through **plugins**; today, agents auto-activate the plugins they need, so grounding still happens through the same connectors, just with a layer of automation on top.
3. **Model processing.** The grounded, “modified” prompt goes to the LLM inside Microsoft’s compliance boundary. Your data stays in that boundary; it is not used to train the foundation models, and the model provider does not get access to it. Responsible-AI checks run over the output.
4. **Post-processing.** The orchestrator enriches and verifies the response — resolving IDs to names, attaching evidence, formatting KQL, and preparing any actions the surface can execute.
5. **Response and actions.** You get the answer, plus, where applicable, actions you can run (or that an agent runs for you) directly in the originating product.

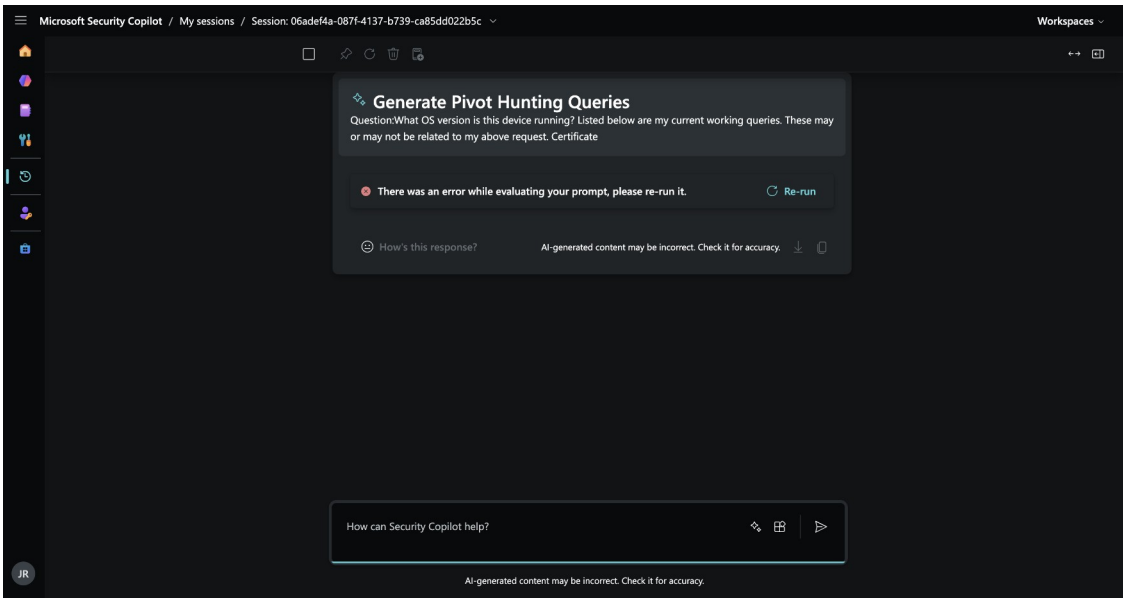

Figure 13.2: A Security Copilot session: the prompt, the orchestrator's expandable process view showing which plugins and skills it invoked, and the grounded response

The mental model to keep is the same one that runs through this whole book: **Copilot is only as good as the data it can reach.** The reason it is genuinely useful for device management is that Intune, Defender, and Entra expose rich, queryable data about your fleet, and Copilot is wired into it. Feed it a device that Intune knows nothing about and it has nothing to say.

The shift from plugins to agents

This is the single biggest change since the first edition, so it gets its own section.

In the original product, you extended Copilot with **plugins** — connectors to Microsoft and third-party data sources that you toggled on and off, plus **promptbooks**, which are saved sequences of prompts you run as a unit. Both of those still exist. Plugins live underneath the surface (agents activate them automatically), and promptbooks are still a good way to standardize a repeatable investigation. But the thing you build with and think in now is the **agent**.

An agent is a scoped, semi-autonomous worker. Instead of you prompting "list the phishing reports, then triage each one, then classify it, then write it up," an agent is given that whole job once and runs it — often on a schedule or a trigger, not just when you are watching. Agents are the reason the "Agents" entry now sits in the Intune navigation and the reason this chapter needed rewriting. Two things are worth stating plainly:

- **Agents consume SCUs.** Both first-party and partner agents draw on your Security Copilot capacity when they run. An agent that runs continuously is a continuous cost. I come back to this in the pricing section because it is the thing people underestimate.
- **Agents need permissions to act.** An agent that offboards a device or edits a policy has to be allowed to do that, which is an RBAC problem, covered later in the chapter.

The Microsoft Security Store

To discover, buy, and deploy agents, Microsoft built the **Microsoft Security Store** at securitystore.microsoft.com, which entered preview in **September 2025**. Think of it as an app store for security agents and solutions — first-party Microsoft agents and partner agents side by side. You browse by product area or scenario, review what an agent does and what permissions and capacity it needs, and deploy it into your tenant. The store is also surfaced inside the embedded experiences, so you can reach the relevant agents from within Defender, Entra, and Intune rather than always starting in the store.

Partner agents may require a **separate purchase** through the store on top of the SCUs they consume when they run — worth knowing before you plan a budget around them.

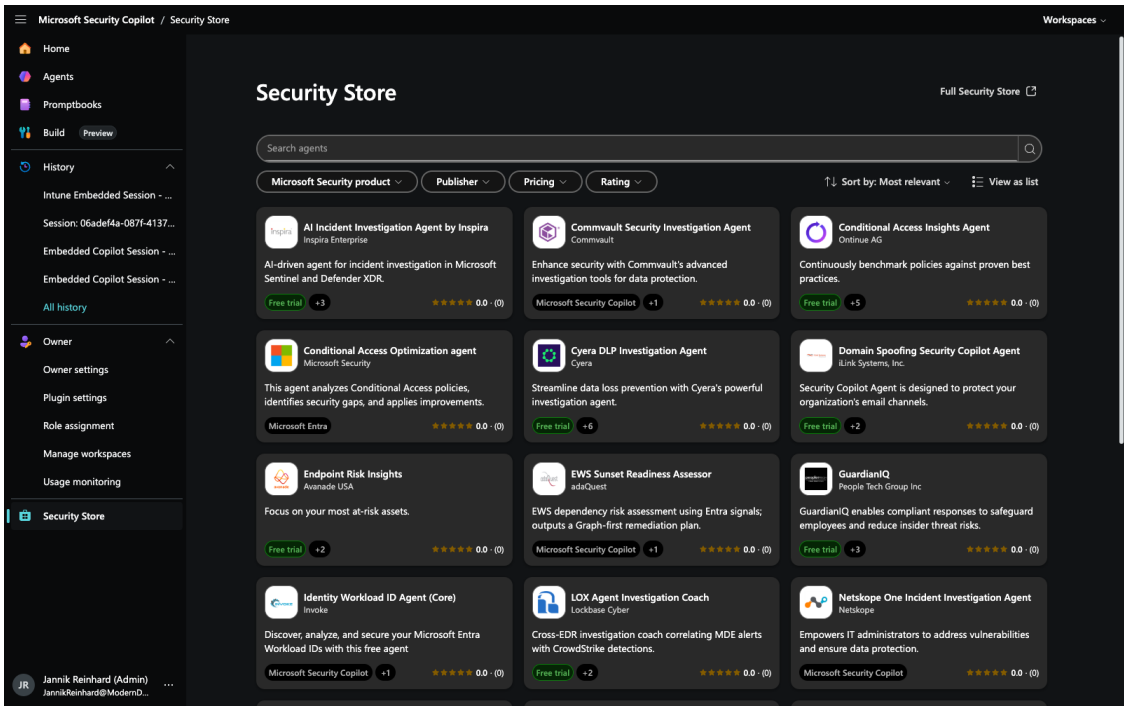

Figure 13.3: The Microsoft Security Store, where you discover, buy, and deploy first-party and partner Security Copilot agents

The first-party agents, by product

Here is the current first-party agent lineup as of mid-2026. **Pay attention to the GA-versus-preview labels** — they are not cosmetic. A preview agent can change behavior, may not be covered by the same support and SLA, and I would not point one at production actions without a careful test.

Microsoft Defender

- **Phishing Triage Agent** — **GA.** Triages user-reported phishing, separating real threats from false alarms and writing up the reasoning. This is the flagship, and the one that has been GA long enough to trust.
- **Security Alert Triage** — **preview.** Note the split: triage for **email and collaboration** alerts is **GA**, while triage for **cloud and identity** alerts is still **preview**.
- **Threat Intelligence Briefing Agent** — generates a threat-intel briefing tailored to your organization.
- **Threat Hunting Agent** — runs proactive hunts across your data.
- **Security Analyst Agent** — **preview (April 2026).**
- **Dynamic Threat Detection** — newer detection agent.

Microsoft Entra

- **Conditional Access Optimization Agent** — **GA (July 2025).** Watches your Conditional Access estate for gaps — new users or apps not covered by policy — and recommends fixes. Directly relevant to us: device compliance is a Conditional Access signal, so this agent's recommendations often touch the Intune-managed device posture.
- **Identity Risk Management** — **preview.**
- **Access Review** — **preview.**

Microsoft Intune — all four are **preview** as of mid-2026, so treat them as capable-but-not-yet-production:

- **Change Review Agent** — reviews configuration and policy changes for risk and unintended impact before or after they land.
- **Device Offboarding Agent** — handles the workflow of removing a device cleanly.
- **Policy Configuration Agent** — assists in creating and configuring policy, grounded in your existing settings and Microsoft's recommendations.
- **Vulnerability Remediation Agent** — takes a vulnerability signal and drives the remediation on the managed device.

Microsoft Purview

- **Insider Risk Management (IRM) Triage — preview.**
- **DLP Alert Triage — preview.**

Custom agents. Beyond the catalog, you can build your own — either with a **no-code builder** or programmatically via **MCP (Model Context Protocol)**, both in **preview since September 2025.** For a device team, a custom agent is how you encode a repeatable, tenant-specific runbook — "every morning, check for devices that fell out of compliance overnight, summarize why, and open a ticket" — once and let it run.

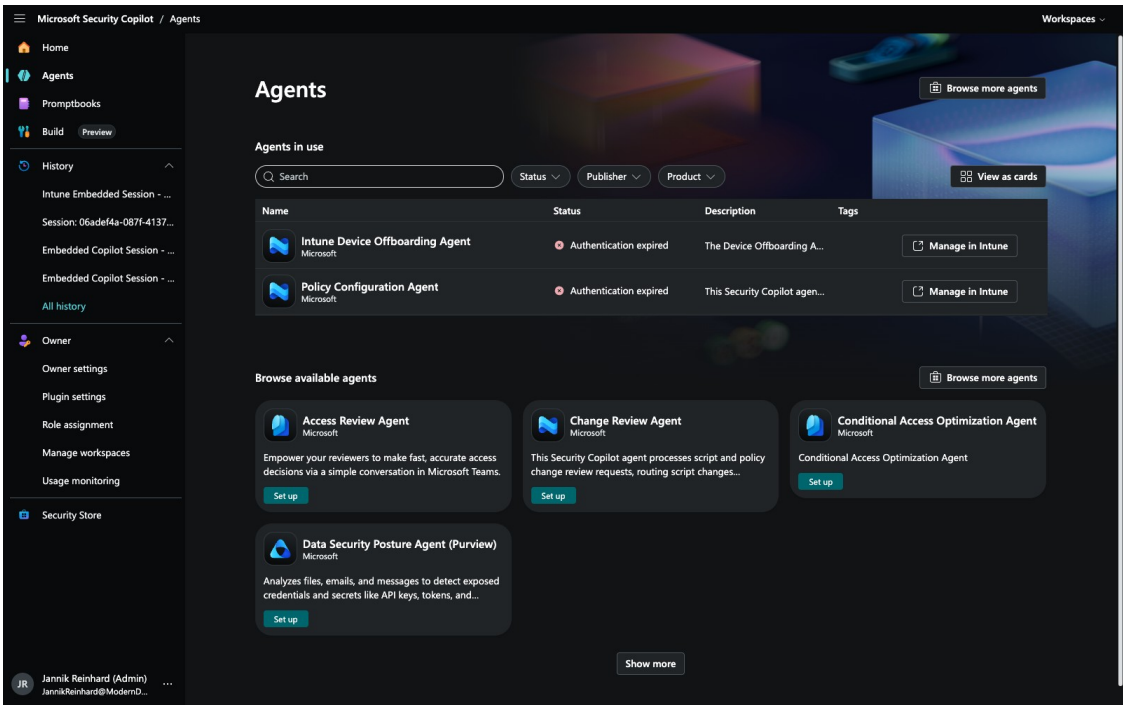

Figure 13.4: The Agents surface. Agents already in use (including Intune agents managed back in Intune) and a catalog to browse — Conditional Access Optimization, Access Review, the Purview Data Security agent, and more

Pricing: SCUs, overage, and the E5/E7 inclusion

Security Copilot is billed on **Security Compute Units (SCUs)** — the unit of capacity that powers everything, including the agents. This is where the first edition's math is now wrong, so read this section carefully; getting it right is the difference between a predictable bill and a nasty surprise.

There are three things to understand.

1. Provisioned SCUs — $4.00 per SCU per hour. This is the baseline capacity you reserve. It is billed **monthly**, in **whole-hour blocks**, with a **minimum of 1 SCU**, and there is **no rollover** — unused provisioned capacity in an hour is gone, not banked. One provisioned SCU running around the clock is roughly $2,920 a month, which is the figure the old edition quoted for "an SCU." The difference now is what happens when you go over.

2. Overage SCUs — $6.00 per overage SCU (GA April 2025). This tier did not exist in the first edition. When demand exceeds your provisioned capacity, Security Copilot can burst into **overage**, billed **to one decimal place** at **$6.00 per overage SCU**. You can **cap** overage (to control spend) or leave it **unlimited** (to never throttle). Overage is what makes agents financially interesting and financially dangerous at the same time: an agent that suddenly has a lot of work to do can push you into the more expensive tier without anyone clicking a thing.

3. Included with Microsoft 365 E5 and E7 (November 2025). This is the change most readers will actually feel. If you have E5 or E7, Security Copilot now comes with a **Default Capacity** that is auto-provisioned for you: **400 SCUs per month for every 1,000 paid eligible licenses**, capped at **10,000 SCUs per month**. Consumption beyond your included capacity spills into **$6 overage**. In practice this means many E5/E7 customers can now *try* Security Copilot, run some agents, and use Copilot in Intune without a separate procurement — the capacity is already there. It also means the cost conversation shifts from "should we buy any capacity at all" to "are our agents about to blow through the included allotment into overage."

An honest word on cost management

I am going to be blunt here because the product marketing will not: **SCU cost management is a real, ongoing operational concern, not a one-time sizing decision.** The old advice — "size your SCUs carefully up front because you can't change them" — is no longer even the right shape of the problem. The problem now is *consumption over time*: agents run on schedules and triggers, embedded Copilot gets used across teams, and overage is billed at a 50% premium over provisioned capacity. If you turn on several preview agents "to see what they do" and forget about them, they keep consuming.

Practical guidance:

- **Watch the usage dashboard.** The capacity view shows provisioned versus consumed SCUs over time. Check it, especially in the first weeks after enabling agents.

- **Cap overage while you learn.** An unlimited overage setting plus an eager agent is how you get a surprise invoice. Start capped, raise the cap deliberately.
- **Treat each agent as a running cost, not a feature toggle.** Ask “what does this cost per day at our volume” before you leave it on.
- **Measure the payback.** The genuine wins — reduced mean time to respond, analyst hours saved on triage — are real, but you should measure them (MTTR, average resolution time) rather than assume them, precisely because there is now a meter running.

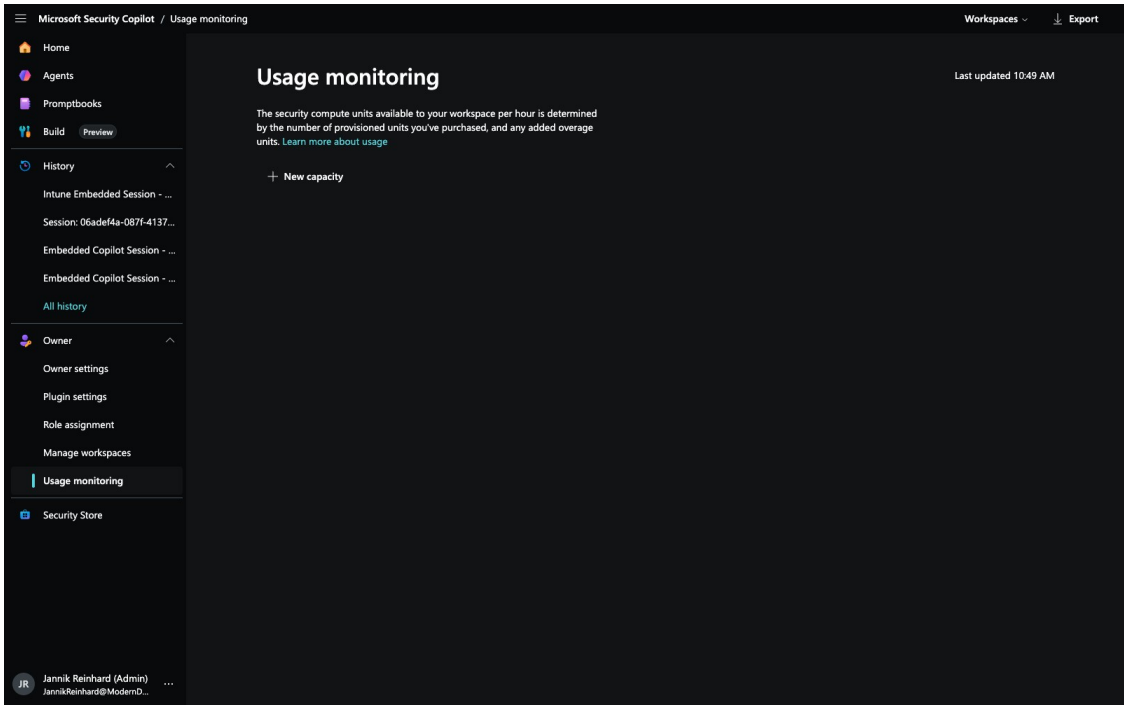

Figure 13.5: Usage monitoring under Owner settings, where you watch SCU consumption and overage — the number that decides your Security Copilot bill

Setup, RBAC, and workspaces

Two onboarding paths

How you turn Security Copilot on depends on your licensing:

- **E5 / E7 (auto-provisioned).** With the November 2025 inclusion, eligible tenants get a **Default Capacity** provisioned automatically. Practically, you go to the portal, confirm the capacity is there, assign roles, and start. No Azure subscription dance required for the baseline.
- **Non-E5/E7 (manual provisioning).** You provision SCUs yourself, and this path **requires an Azure subscription** to attach the capacity to, because that is where the SCU resource and its billing live. You choose your provisioned SCU count, set your overage behavior, and go.

Either way, first-run setup covers capacity, data-residency/region choice, and diagnostic settings, then hands you a working prompt experience.

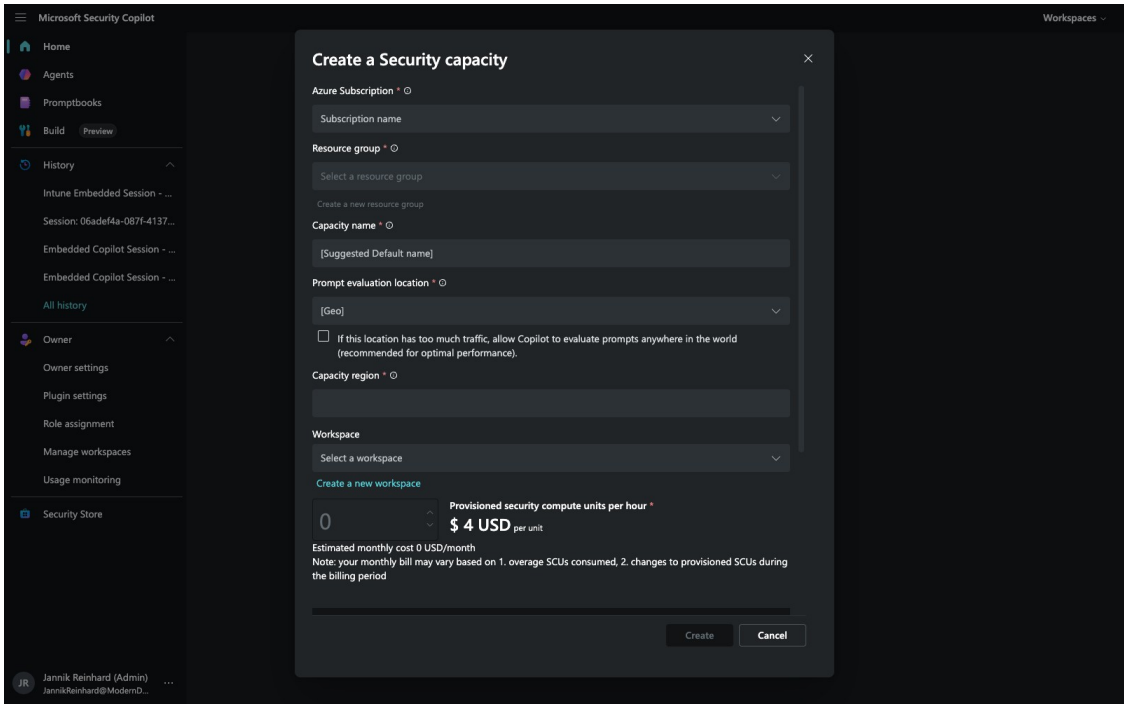

Figure 13.6: Provisioning a Security capacity: SCUs at 4 USD per unit per hour, a prompt-evaluation geography, and an estimated monthly cost. This is the non-E5/E7 path

I have deliberately dropped the old ARM-template-and-PowerShell walkthrough from this chapter. The capacity resource and its API surface have moved on, the previous template targeted a preview API version that no longer reflects the resource, and for the large majority

of readers the E5/E7 auto-provisioning path makes hand-rolled IaC unnecessary. If you do need infrastructure-as-code for capacity today, provision it from the current Azure resource provider rather than copying a two-year-old template — pinning a stale preview API version is how you get a deployment that silently drifts from the real product.

RBAC

Access to Security Copilot is governed by two dedicated roles, plus the underlying product roles that agents inherit:

- **Security Copilot Owner** — full control, including capacity, settings, and role assignment.
- **Security Copilot Contributor** — can use Copilot and work with sessions and agents, without owning the capacity.

That covers *Copilot itself*. But an **agent that takes action** needs the permission to take that action in the target product, and that comes from **Entra roles**. Configuring an Intune agent, for example, means the identity behind it needs the appropriate Intune/endpoint permissions; configuring identity or Conditional Access agents typically needs **Security Administrator** or an equivalent Entra role. The principle is least privilege, same as ever: an agent should have exactly the permissions its job requires and no more. When you evaluate a preview agent that performs actions, map out *what it will be allowed to change* before you enable it — that review matters more than any single setting in the product.

Workspaces

Workspaces went GA in June 2025, and they are how larger organizations keep Security Copilot from becoming one undifferentiated pool. A workspace lets you **segment by team, region, or business unit**, each with its **own RBAC, its own capacity, and its own session history**. For a global endpoint team this is genuinely useful: you can give the EMEA device team its own workspace and capacity, ring-fence its spend, and keep its sessions separate from the SOC's. If you operate across regions or run managed services for multiple customers, set workspaces up early rather than retrofitting them.

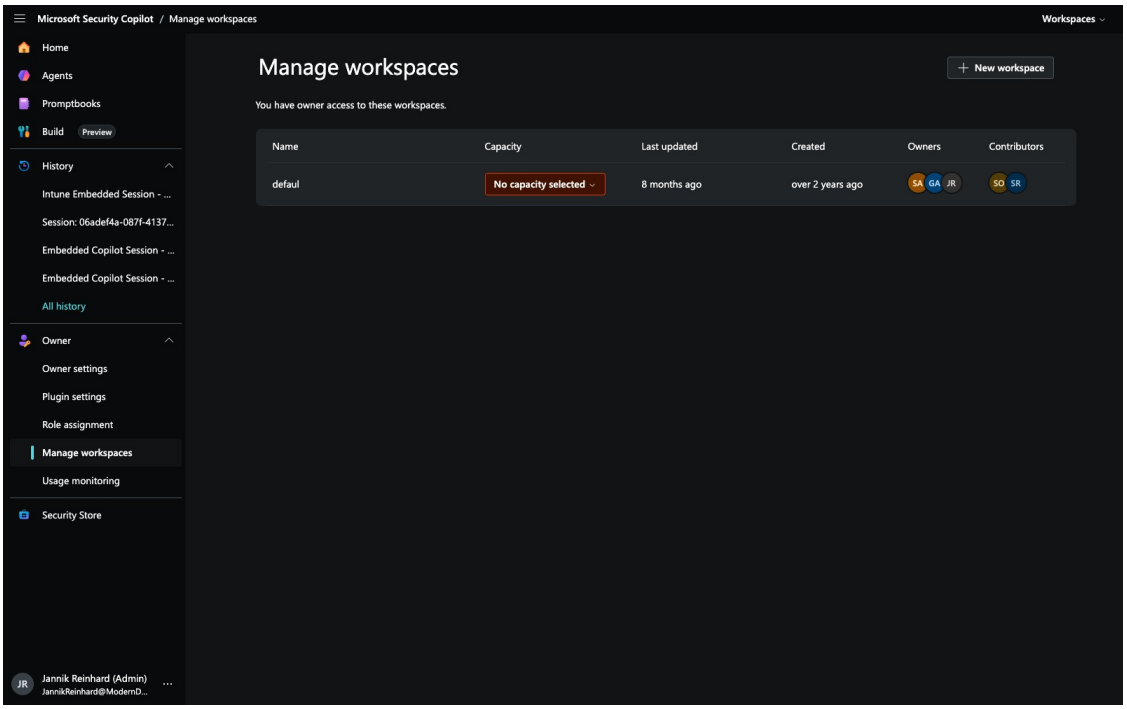

Figure 13.7: Workspaces let you segment Security Copilot by team or region, each with its own roles, history, and capacity

Data residency

Where your prompts and data are processed matters for compliance. The current position:

- Regional data-residency options continue to expand — a **Swiss region was added in June 2025**, alongside the existing regions.
- Security Copilot holds **FedRAMP High** authorization in the commercial cloud.
- It is **not yet available in the US Government clouds** (GCC High / DoD). If you operate there, this is a hard blocker to check before you plan anything on top of Copilot.

Copilot in Intune and the Intune agents

Now the part that matters most for this book. Everything above is context; this is where a device administrator actually spends time.

Copilot embedded in the Intune admin center

Copilot in Intune (GA July 2025) shows up as context-aware assistance inside the admin center. Open a device, a policy, or a configuration profile and you get a Copilot entry point — often an **"Explore with Copilot"** button or the Copilot panel — with prompts scoped to whatever you are looking at. The experience is more guided than the open canvas of the standalone portal, because it knows the object you are on.

The everyday wins for a device admin:

- **Summarize a device.** From a device blade, get a consolidated picture — hardware, compliance state, security posture, installed and vulnerable software, recent anomalies, and the relevant Intune context — without clicking through six blades to assemble it yourself. This is the single most-used interaction and, honestly, the one that sold the feature to my teams.
- **Explain a policy or setting.** Ask what a setting does, what happens if you change it, and which devices it hits. Good for the "why is this configured this way" archaeology that eats an admin's afternoon.
- **Compare and reconcile configurations.** Spot conflicts and overlaps between profiles in plain language.
- **Help with Device Query.** Draft and interpret the KQL behind Device query so you spend less time on syntax and more on the answer. (Device query itself is covered in Part II — Copilot sits on top of it.)

Because this is the same Security Copilot engine, remember it **draws on your SCU capacity**. Heavy interactive use across a big admin team is real consumption, not a free bundled feature.

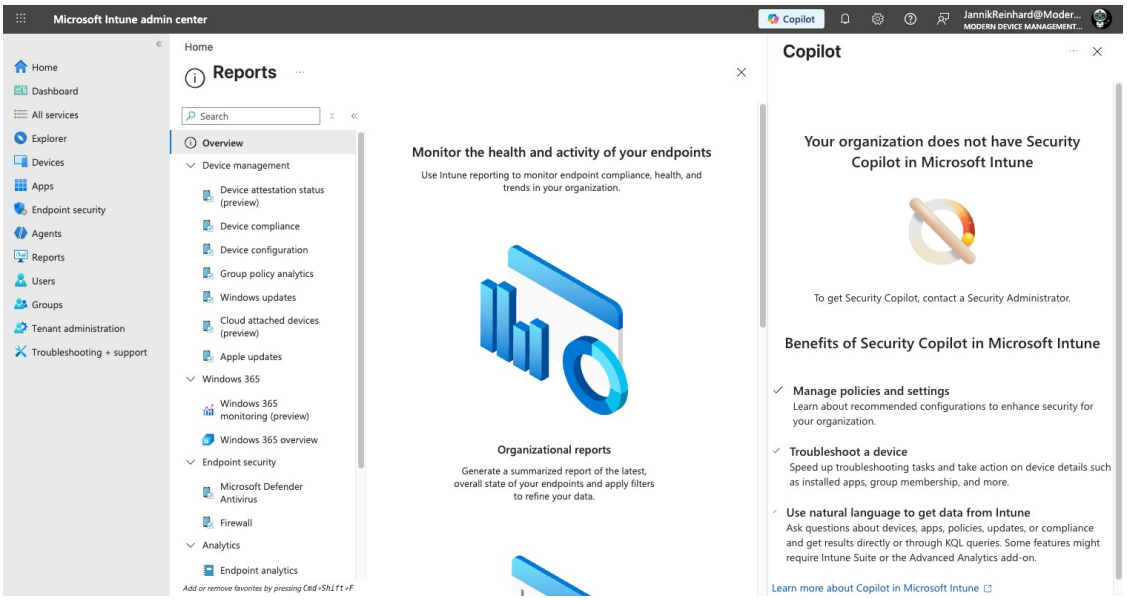

Figure 13.8: Security Copilot in Intune — the in-product surface whose benefits panel spells out what it does once a Security Administrator turns it on: manage policies and settings, troubleshoot a device, and query Intune in natural language

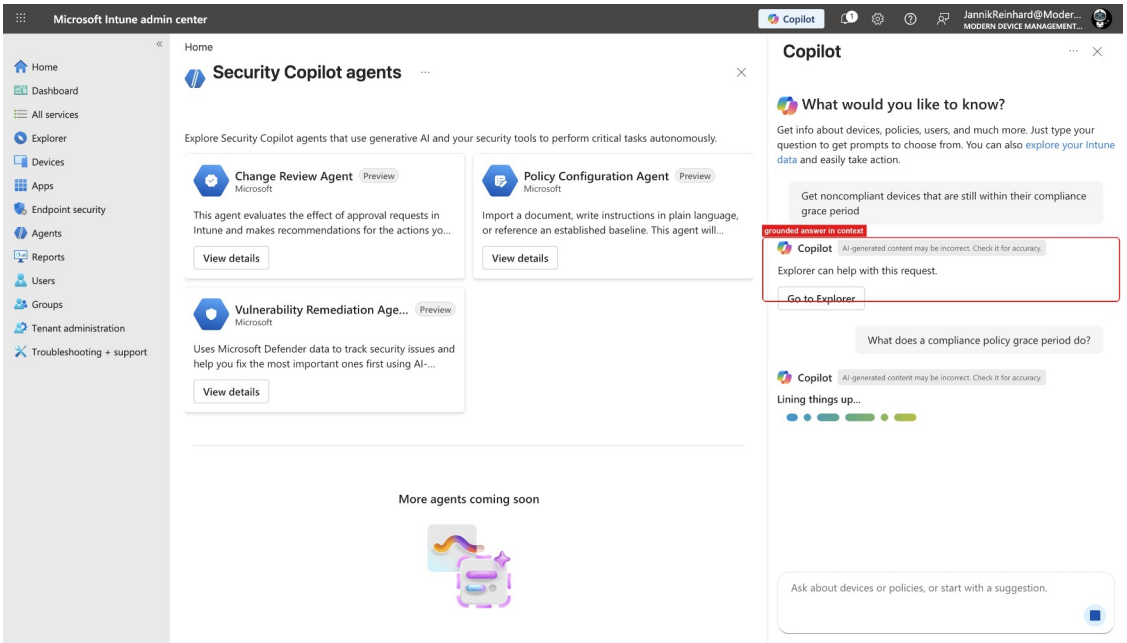

Figure 13.9: Copilot open on an Intune object, offering to summarize and explain it in context — the same engine as the standalone portal, surfaced where the admin already is

The Intune agents in practice

The four Intune agents (all **preview** as of mid-2026) are where Copilot stops answering and starts *doing*. Used carefully, they cover some of the most tedious, error-prone endpoint work:

- **Change Review Agent.** Configuration drift and unreviewed policy changes are a classic source of outages and compliance gaps. This agent reviews changes for risk and unintended blast radius — which devices a change will hit, whether it conflicts with an existing policy, whether it weakens a baseline. In a change-heavy tenant this is a real second pair of eyes.
- **Device Offboarding Agent.** Cleanly removing a device — retiring or wiping, revoking access, tidying up records across Intune and Entra — is fiddly and easy to do half-way. The agent drives the workflow end to end. Given the destructive nature of offboarding, this is exactly the kind of agent to test hard and scope tightly while it is in preview.
- **Policy Configuration Agent.** Assists in building and configuring policy, grounded in your existing settings and Microsoft's recommendations, so a new baseline starts from something sensible rather than a blank blade.
- **Vulnerability Remediation Agent.** Takes a vulnerability signal (from Defender's exposure data) and drives the remediation on the managed device — the connective tissue between "we know this machine is vulnerable" and "the fix is deployed." This is the loop that used to require an analyst to hand a ticket to a device admin who then built a deployment; the agent closes it.

The honest caveat: **preview means preview.** These four agents are promising and genuinely useful in testing, but I would not hand any of them unattended action rights over production devices yet. Run them in a controlled scope, review what they propose, keep a human approving the consequential actions, and watch the SCU consumption while you do. When they reach GA, revisit the automation posture — until then, treat them as a very capable assistant that still needs supervision.

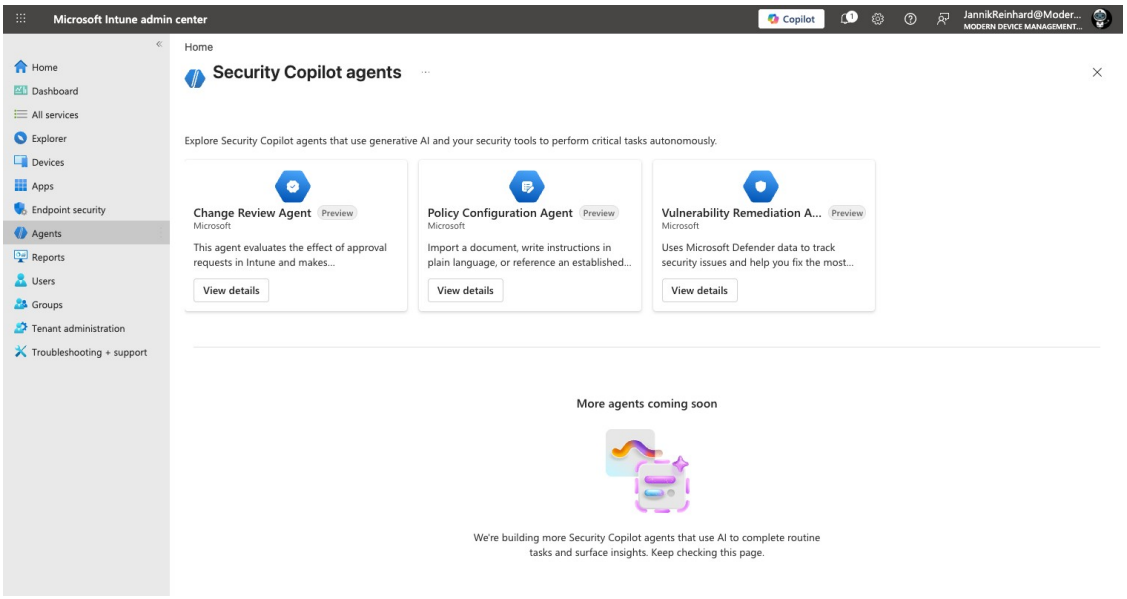

Figure 13.10: The Security Copilot agents surface reached from the Agents node in Intune — the Intune agents in preview (Change Review, Policy Configuration, Vulnerability Remediation, with Device Offboarding and more on the way), each an autonomous worker you enable and scope

Practical prompts and promptbooks

Security Copilot responds to plain language — no special syntax, no KQL required (though it will happily write KQL for you). The quality of what you get back tracks the quality of what you ask: name the object, state the goal, and ask for the format you want. Here are prompts I actually use, written for the current product.

Device-centric (from Intune or the portal):

```
Summarize the device WKS-4821: compliance state, security posture,
vulnerable software, and any anomalies from the last 14 days.
List all Windows devices that fell out of compliance in the last 24 hours
and group them by the compliance policy setting that failed.
Explain what the "Require BitLocker" setting in this configuration profile
does, which devices it is assigned to, and what breaks if I make it required.
```

Vulnerability and remediation:

```
For device WKS-4821, list the vulnerable software Defender has flagged,
sorted by severity, and for the top three recommend the Intune remediation.
```

Incident and triage (device-adjacent):

> Summarize incident INC-10293, focus on the endpoint impact, list the affected Intune-managed devices, and give me the recommended containment steps.

Conditional Access / identity (device posture matters here):

> Show Conditional Access gaps for devices: users or apps not covered by a policy that requires a compliant or hybrid-joined device.

Promptbooks package a sequence like this into one reusable unit — for example a "device investigation" promptbook that summarizes the device, pulls its vulnerabilities, checks its compliance history, and drafts a remediation plan, run as a single action. Where a promptbook is a *scripted* sequence you trigger, an **agent** is the *autonomous* version of the same idea — given the job, it decides the steps and runs them, on a schedule or a trigger. Start with promptbooks to standardize how your team investigates; graduate the well-worn ones to agents when you trust the pattern and are ready to pay for it to run on its own.

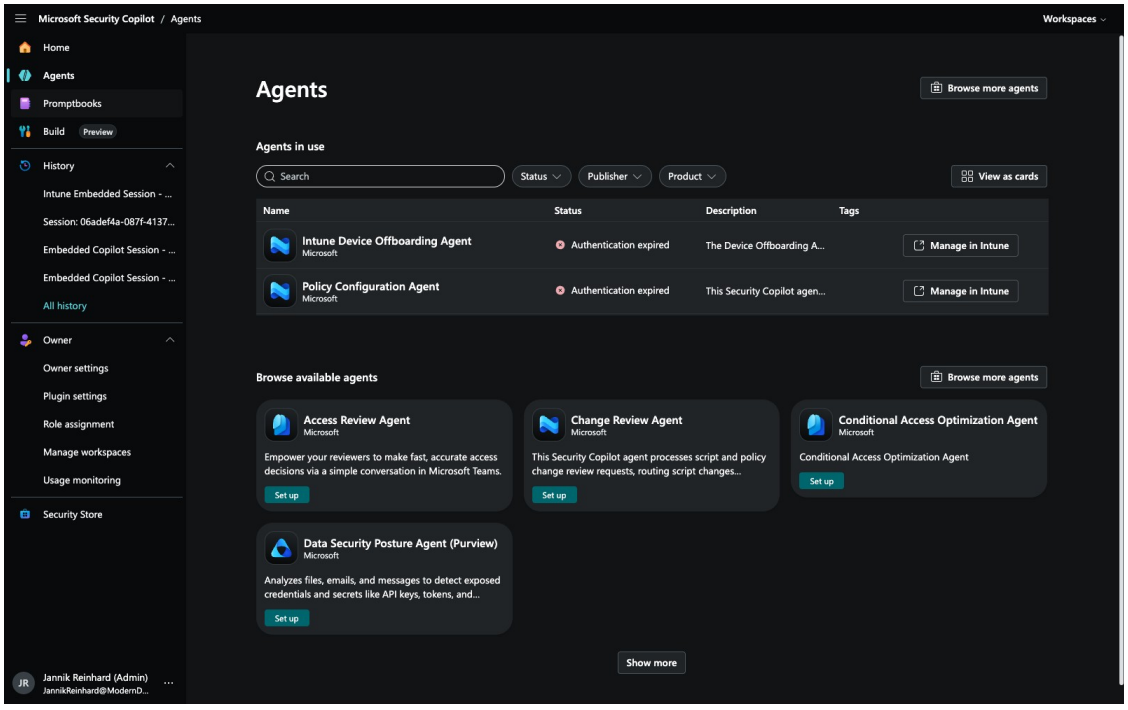

Figure 13.11: The Promptbooks library — reusable, parameterized prompt sequences for repeatable investigations

Key use cases across roles

Security Copilot pays off differently depending on who is holding it. Keeping the endpoint lens, here is where it earns its keep:

- **For device and endpoint admins (us).** Summarize devices without blade-hopping, explain and reconcile policies, draft Device Query KQL, and — with the Intune agents — review risky changes, drive clean offboarding, and close vulnerability-to-remediation loops. This is the daily value.
- **For SOC analysts.** Triage and summarize alerts and incidents, decode suspicious scripts in plain language, and hunt proactively. The Phishing Triage Agent (GA) alone removes a large, repetitive load; when an endpoint is involved, the Intune device summary drops straight into the investigation.
- **For identity admins.** The Conditional Access Optimization Agent (GA) surfaces coverage gaps, several of which are device-posture gaps, tying back to Intune compliance.
- **For CISOs and leads.** Executive summaries of posture, incident prioritization, and a defensible view of where time and risk are concentrated — with the caveat that you should also be watching what all of this costs.

Honest limitations

I would not be doing my job if I only sold the upside. What to keep in mind:

- **Cost is a live concern, not a footnote.** SCU consumption is ongoing, agents run whether or not you are watching, and overage is billed at a premium. Budget and monitor it deliberately. This is the limitation I hear about most from people running it in production.
- **Preview is not GA.** Most of the Intune agents, the Security Store, MCP/custom agents, and several Defender and Entra agents are in preview. Behavior and scope can change, and I would not point preview agents at unattended production actions.

- **It is grounded in Microsoft data.** Copilot is only as good as what it can reach. Sparse or poorly maintained Intune, Defender, or Entra data yields thin answers — the AI does not compensate for missing telemetry.
- **Government-cloud gap.** No US Gov cloud availability yet. A hard blocker for those tenants.
- **Humans stay in the loop.** For anything consequential — offboarding a device, changing a baseline, remediating at scale — keep a person approving the action. The product is an accelerant for good administrators, not a replacement for judgment.

Where this leaves us

Security Copilot in 2026 is a different product from the one in the first edition. The chat box is still there, but the center of gravity moved to **agents**, the pricing gained an **overage tier** and, more consequentially, an **E5/E7 inclusion** that puts capacity in many tenants' hands by default, and the embedded experiences — **Copilot in Intune above all** — went GA. For a device team, the practical payoff is concrete: faster device investigations, plainer-language policy work, and preview agents that start to close the loop from signal to remediation on the endpoints you manage. The discipline it demands in return is equally concrete: watch the meter, respect the preview labels, and keep a human on the consequential actions.

In the next chapter we build on this — taking the agent idea further and looking at how you construct your own AI-driven capabilities on top of your tenant, rather than only consuming the ones Microsoft ships.

If you are not already part of it, I still recommend the community here: linkedin.com/groups/14345161.

Chapter 14: Deep Dive into Copilot Architecture and Building Your Own Grounding

Chapter 13 was about Security Copilot as a product you buy and switch on. This chapter is about what is happening underneath — how a Copilot is actually built, and how you build the same thing yourself for your own data. That distinction matters, because once you understand the architecture, the Copilots stop being magic and start being a pattern you can reproduce. And in device management there is a lot you would want to reproduce: a private assistant that knows your naming conventions, your remediation runbooks, your internal wiki, your last two years of incident write-ups, and can answer questions grounded in *that* instead of the public internet.

So this chapter has two halves. The first is architectural: how M365 Copilot is wired together, and the "Copilot stack" as a mental model — a framing that is still useful in 2026, but has shifted under our feet from plugins toward agents and tools. The second half is the practical core of the whole chapter: **how to build your own grounding with internal knowledge**, which in almost every real case means retrieval-augmented generation (RAG) on top of Azure AI Search. We will go deep on AI Search because it is the engine, cover the genuinely new agentic-retrieval capability, wire in a web search the current way, and finish on orchestration frameworks. Chapter 15 then takes all of this and builds a working assistant on Foundry. Consider this chapter the theory and the plumbing; the next one is the build.

A note before we start, because names have changed a lot and getting them wrong makes the whole thing confusing. **Azure Cognitive Search is now Azure AI Search. Semantic search is now the semantic ranker. Azure OpenAI Studio / Azure AI Foundry is now Microsoft Foundry**, and Azure OpenAI Service is "Azure OpenAI in Foundry Models." I will use the current names throughout and flag the old ones the first time, so that when you hit a two-year-old blog post you can translate.

The architecture of M365 Copilot

When you ask M365 Copilot in Word to draft a section, or ask it in Teams what a colleague decided in last week's meeting, it feels like one thing. It is not. It is an orchestrated round-trip across several systems, and the sequence is worth understanding because it is exactly the sequence you will rebuild yourself later in this chapter.

Here is the flow, in the order it actually happens:

1. **Your prompt goes to the orchestrator, not the model.** When you type a request, it is not sent straight to a language model. It first hits Copilot's orchestration layer, which is responsible for figuring out what you are actually asking for and what it needs to answer well.

2. **The orchestrator grounds the prompt against your data.** This is the step that makes Copilot Copilot and not ChatGPT. The orchestrator queries **Microsoft Graph** — the single API across Microsoft 365 that exposes your mail, files, chats, calendar, people, and their relationships — and increasingly it does this through **semantic index** retrieval rather than a naïve keyword lookup. It pulls back the handful of documents, messages, and facts that are relevant to your question, respecting every permission you have. Copilot can only retrieve what *you* can already see; the grounding step does not bypass access control.

3. **The prompt is rewritten and enriched.** The orchestrator takes your original ask, the retrieved context, and its own instruction templates, and assembles a much larger "grounded" prompt. Your three-word question becomes a few thousand tokens of context plus instructions. This is prompt construction, and it is where most of the quality comes from.

4. **The enriched prompt goes to a foundation model.** Only now does a large language model — a GPT-family model running in Microsoft's own tenant, not the public OpenAI

endpoint — actually generate text. It generates against the grounded context, so its answer is anchored in your data rather than in whatever it memorised during training.

5. **The response is post-processed.** The raw model output goes back through the orchestrator, which does compliance checks, applies responsible-AI filtering, can call back into Graph to attach citations and source links, and formats the result for the surface you are in.

6. **You get the answer.** With citations, in the app, in a second or two.

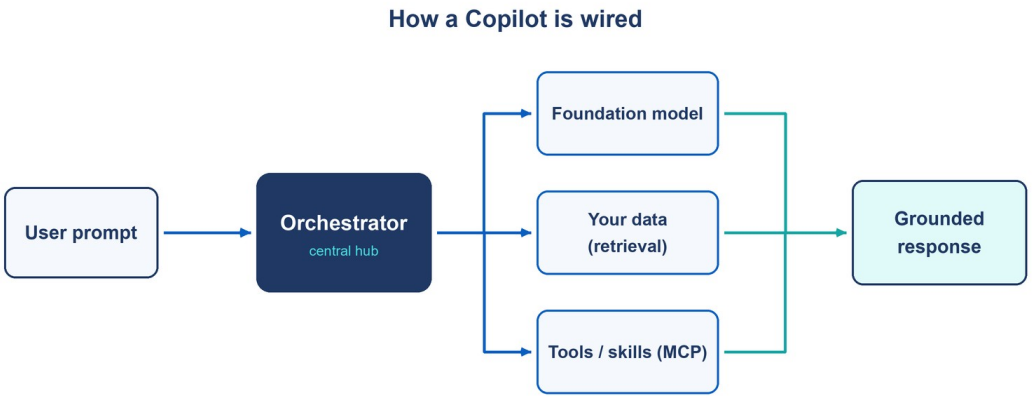

Figure 14.1: How a Copilot is wired: an orchestrator sits between the user and the foundation model, pulling in your data through retrieval and calling tools, then composing a grounded response

The single most important thing to take from this diagram is step 2. The intelligence you notice is the model in step 4, but the *value* is the grounding in step 2. A generic model with excellent grounding beats a state-of-the-art model with none, every time, for enterprise questions. When you build your own assistant later, you will spend maybe ten percent of your effort on the model and ninety percent on grounding — and this chapter is mostly about that ninety percent.

The Copilot stack

Microsoft popularised a useful mental model called the **Copilot stack**. Two years on it is still a good way to think about any AI application, including the one you are about to build, so I will keep it — but I need to update it, because the canonical Microsoft diagram is genuinely stale now and describes a world that has moved.

The stack, from bottom to top:

- **Your data.** At the foundation sits the data that makes the whole thing worth building — in the M365 case, everything reachable through Graph and the semantic index. In your case it will be your documents, your knowledge base, your Intune data. Nothing above this layer matters if this layer is thin.
- **Orchestration.** The middle and by far the most interesting layer. Orchestration is what turns a prompt plus data plus tools into an answer. Historically this layer was described as "foundation models + a toolchain." In 2026 the honest description is **agent-centric**: the orchestrator plans, decides which *tools* to call (retrieval, a web search, a Graph action, a plugin), calls them, reads the results, and decides whether it is done or needs another step. This is **generative orchestration** — the model itself drives the control flow — as opposed to the older fixed pipeline where you hard-coded the steps.
- **User experience.** The surface you interact with: Word, Teams, the Intune admin center's Copilot button, or your own chat UI.

Wrapped around all three layers, top to bottom, is a cross-cutting band of **safety, security, and governance**: identity and permissions, content filtering, responsible-AI checks, data-residency and compliance boundaries, and audit. This is not a layer you add at the end; it runs through everything.

What actually changed since the first edition, and why the old diagram misleads:

- **Plugins became tools and MCP.** The first edition talked about “plugins, skills, and add-ons” as the way you extend a Copilot. The vocabulary and the mechanism have converged. The unit of extension is now a **tool**, and the standard way to expose tools to an agent is the **Model Context Protocol (MCP)** — an open protocol for connecting agents to data sources and capabilities. When you connect your own knowledge base to an agent later, you will very likely do it as an MCP tool.

- **Orchestration went generative and agentic.** The old picture had a deterministic orchestrator picking from a fixed menu. The current picture has a model reasoning about what to do next. That is a real capability change, not a rebrand — it is why “agents” is now the word Microsoft uses everywhere, including the Agents entry that has appeared in the Intune navigation.

- **The foundation-model layer became plural and swappable.** It is no longer “GPT.” Foundry alone offers well over a thousand models — the GPT-5 family, Claude, Grok, Mistral, DeepSeek, Phi, Llama — and part of orchestration is now *routing*: cheap small model for easy turns, expensive frontier model for hard ones.

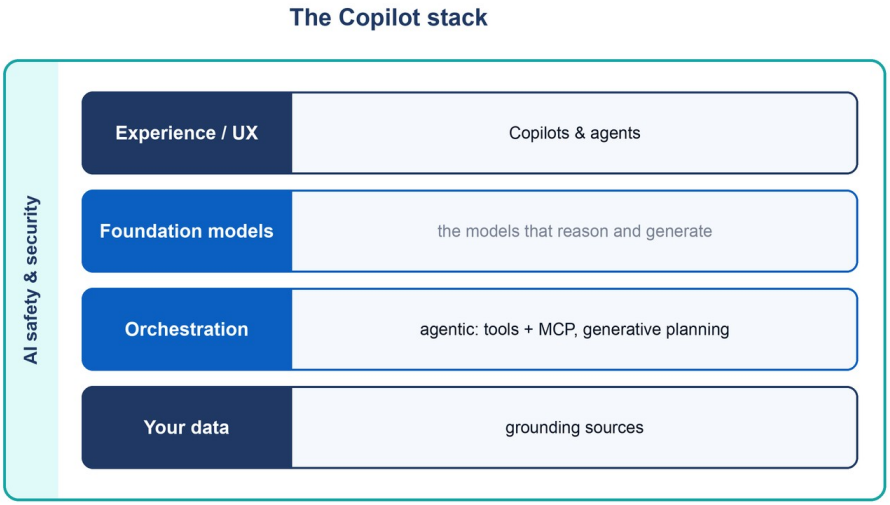

Figure 14.2: The Copilot stack today: your data at the base, an agentic orchestration layer (tools, MCP, generative planning), the foundation models, and the experience on top — all wrapped in safety and security

Keep this stack in your head for the rest of the chapter. Everything we build from here fits into it: Azure AI Search is the *your data* layer, the retrieval and agent logic is *orchestration*, the model is one swappable box inside orchestration, and Chapter 15's chat app is the *experience*.

How can I build my own grounding with internal knowledge?

Now the core of the chapter. You have data — internal documentation, runbooks, past incidents, configuration standards — and you want a model to answer questions using it. There are a few ways to get knowledge into a model, and they sit on a ladder of effort and capability. It is worth walking up the ladder deliberately, because a lot of teams reach for the top rung when the bottom one would have done, and a lot of others stay on the bottom rung long after they have outgrown it.

Rung 1: Just put it in the prompt

The simplest grounding is no infrastructure at all: paste the relevant text into the prompt.

> **Prompt:** "Based on the following runbook, what is the first step when a device fails compliance for BitLocker? [paste the runbook]"

For a one-off question this is perfect, and I use it constantly. Modern models have large context windows, so "the relevant text" can be quite large. But it does not scale: you cannot paste your entire SharePoint into every prompt, you pay for every token every time, and *you* have to already know which document is relevant — which is usually the actual problem you wanted the assistant to solve. The moment you have more knowledge than fits comfortably in a prompt, or you do not know in advance which slice is relevant, you need retrieval.

Rung 2: Fine-tuning (and why it is usually the wrong tool)

The instinct many people have is “train the model on our data.” Fine-tuning adjusts a model’s weights on your examples. It is excellent for teaching a model a *style*, a *format*, or a *task* — “always answer as a terse change-management ticket.” It is a poor tool for teaching a model *facts*, because your facts change (a runbook is updated, a new incident happens) and you cannot retrain every time, and because fine-tuning does not give you citations — the model cannot tell you *where* an answer came from. For an internal knowledge assistant, where facts change weekly and traceability is mandatory, fine-tuning is the wrong rung almost every time. I mention it mostly so you can rule it out with confidence.

Rung 3: Retrieval-augmented generation (RAG)

This is the rung you want, and the rest of the chapter is about it. **RAG keeps your knowledge outside the model, in a search index, and retrieves the relevant pieces at query time to ground each answer.** Nothing is baked into the model’s weights. When your runbook changes, you re-index one document and the assistant is instantly current. Every answer can carry citations back to the source. The model stays generic and swappable.

RAG has two phases. **Ingestion** (done ahead of time, and repeated when data changes): take your documents, break them into pieces, turn each piece into a form you can search, and store it. **Retrieval and generation** (done per query): take the user’s question, find the most relevant pieces, stuff them into the prompt as context, and let the model answer from them. Everything that follows — chunking, embeddings, vector versus keyword versus hybrid search, the semantic ranker, and agentic retrieval — is detail *inside those two phases*. Let me build the concepts up properly, because if you understand these five ideas you understand RAG.

Chunking: why you cannot index whole documents

You cannot search a 40-page PDF as a single unit and expect good grounding. If you retrieve the whole document, you pay for 40 pages of tokens to answer one question and you bury the one relevant paragraph in noise. So the first ingestion step is **chunking**: splitting each document into

pieces small enough to be precise but large enough to carry meaning — typically a few hundred to a couple of thousand characters, often roughly a page or a section.

Two chunking parameters matter. **Chunk size** trades precision against context: small chunks retrieve very targeted text but can cut a thought in half; large chunks preserve context but dilute relevance. **Overlap** — repeating a bit of the end of one chunk at the start of the next — stops you from slicing a sentence or a procedure exactly at a boundary and losing it. In the skillset you will see later, that is `maximumPageLength` and `pageOverlapLength`. There is no universal right answer; for structured technical docs I start around 2000 characters with ~500 characters of overlap and adjust based on how the retrieval feels.

Embeddings: turning meaning into numbers

Here is the idea that makes semantic search possible. An **embedding model** takes a piece of text and produces a **vector** — a long list of numbers (1536 or 3072 of them for the models we use) — that represents the *meaning* of that text as a point in a high-dimensional space. The crucial property: texts with similar meaning land near each other in that space, even if they share no words. “How do I reset my password?” and “steps to recover account access” have almost no words in common but sit close together as vectors. “How do I reset my password?” and “reset the network switch” share a word but sit far apart.

During ingestion you run every chunk through an embedding model and store the resulting vector alongside the chunk. Azure AI Search uses Foundry embedding deployments for this; **`text-embedding-3-large`** is the current workhorse (the old `text-embedding-ada-002` you will see in first-edition-era docs is superseded). AI Search now supports vectors up to 4096 dimensions.

Three ways to search: keyword, vector, hybrid

Once your chunks and their vectors are in the index, you can retrieve them three ways, and understanding the difference is the difference between an assistant that works and one that is frustratingly almost-right.

- **Keyword search (BM25).** The classic. Match the words in the query against the words in the documents, rank by term frequency and rarity. It is fast, cheap, and *exact* — which is its strength and its weakness. It is unbeatable when the user searches for a specific error code, a policy ID, a product name, or an exact phrase. It is helpless when the user's words differ from the document's words, because it has no concept of meaning. Search "laptop won't turn on" and a document titled "device fails to boot" is invisible to it.

- **Vector search.** Embed the query into the same vector space as the chunks, then find the chunks whose vectors are nearest (typically by cosine similarity, using an approximate-nearest-neighbour algorithm like HNSW so it stays fast over millions of vectors). This is semantic matching: it finds the "device fails to boot" document for the "laptop won't turn on" query because they mean the same thing. Its weakness is the mirror image of keyword search — it can miss an exact string. Search for a precise error code and vector search may hand you semantically *similar* passages that do not contain the actual code.

- **Hybrid search.** Run both, keyword and vector, and fuse the two ranked lists into one using **Reciprocal Rank Fusion (RRF)**. You get the exactness of keyword search *and* the meaning-awareness of vector search. In my experience hybrid is the right default for almost every RAG system — you rarely know in advance whether a given user question is going to be a "find this exact code" question or a "find something that means this" question, and hybrid handles both. Unless you have a strong reason, build hybrid.

The semantic ranker: getting the best chunk to the top

Hybrid search gives you a good set of candidate chunks, but "in the set" is not the same as "at the top," and in RAG the top matters enormously — you only feed the model the first handful of results, so if the best chunk is ranked eighth, the model never sees it. This is what the **semantic ranker** (renamed from "semantic search" back in late 2023 — same feature, clearer name) fixes.

The semantic ranker is a second pass. After hybrid search returns its candidates, a Microsoft-trained language model **re-ranks** them by actually reading the query and each candidate and

judging true relevance, not just vector distance or keyword overlap. It reliably pulls the genuinely-best passage up to position one or two. It can also return extractive **captions** and **answers** — the specific sentence that answers the question. For "How do I reset my password?" it will rank the document that contains the actual reset *procedure* above one that merely mentions passwords a lot. This is the single highest-leverage quality upgrade you can make to a RAG system, and I turn it on almost by reflex.

Two practical notes. The semantic ranker is a premium, usage-billed capability — it is charged per query when you set `queryType=semantic` — but as of late 2025 it runs even on the free tier with an allowance, so you can try it before you pay. And you only pay when you actually request it, which means you can reserve it for the queries that need it.

When you need more: agentic retrieval

Everything above answers a *single* query in one shot: embed it, search, rank, return. That is enough for most questions. But some questions are not single questions wearing a trenchcoat — they are several. "Compare our BitLocker remediation with our FileVault remediation and tell me which one has caused more incidents this year" is really three retrievals (BitLocker runbook, FileVault runbook, incident history) plus a synthesis. Fire that at a one-shot RAG pipeline and you get a mushy answer, because a single search over a compound question retrieves a little of everything and enough of nothing.

This is the genuinely new capability in Azure AI Search, and it is worth understanding precisely because the terminology has already changed twice. It arrived in May 2025 as **agentic retrieval** via a "knowledge agent," and at Ignite in November 2025 the packaging was renamed **knowledge bases**. The idea: instead of you writing one query, an LLM sits in front of the index and **decomposes** the user's question into multiple focused subqueries, runs them **in parallel**, semantic-ranks each set, and **merges** the results into one grounded payload — complete with citations and an activity log so you can see exactly what it searched and why. Billing is token-based, because there is a model in the loop, and the planning uses GPT-5-family models.

Be precise about what is generally available versus preview, because it matters for what you can put in production:

- **Knowledge bases are GA in the `2026-04-01` REST API for the *extractive* path** — decompose, search, rank, merge, return the passages with citations. That is production-ready.
- **LLM query *planning* and answer *synthesis* are still in preview.** In other words, the part that generates a written answer for you, rather than handing back ranked passages, is not GA yet.

So the rule of thumb: reach for agentic retrieval / knowledge bases when your users ask multi-part or comparative questions, when a single query demonstrably retrieves a muddle, or when you want that activity-log transparency. For a straightforward "answer this from the docs" assistant, hybrid + semantic ranker is still the right, simpler, cheaper choice. Do not pay for a knowledge agent to answer questions a single query already answers well.

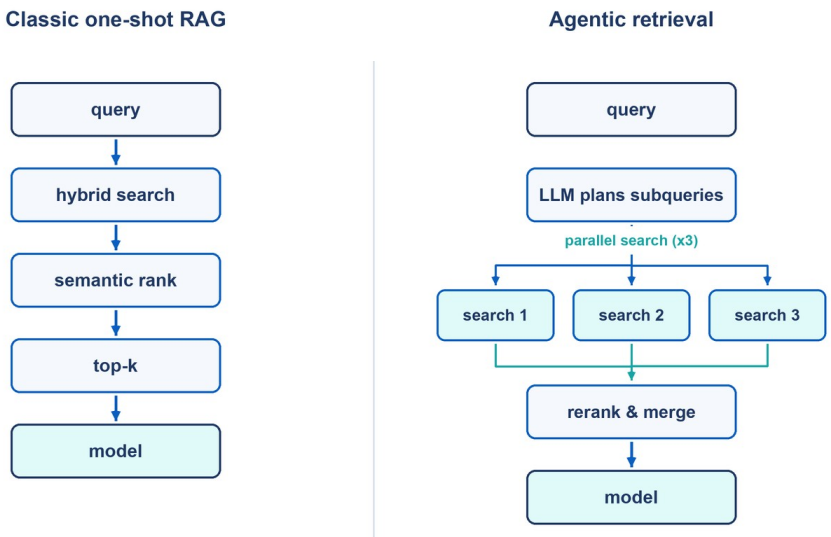

Figure 14.3: Classic one-shot RAG versus agentic retrieval: instead of a single query, an LLM plans several focused subqueries, runs them in parallel, and reranks the merged results

Deploying and understanding Azure AI Search

Enough theory — let us stand the engine up. Azure AI Search is a Platform-as-a-Service search service, and it is the piece that does all the retrieval work described above: it holds your chunks and vectors, runs keyword, vector, and hybrid search, applies the semantic ranker, and hosts the knowledge bases behind agentic retrieval. It is genuinely one of the most important services in the modern AI stack, precisely because grounding is the hard part and this is what does the grounding.

Creating the service

In the Azure portal, search for **Azure AI Search** (if you are following an old tutorial that says "Cognitive Search," this is the same thing) and click **Create**. Pick a subscription, resource group, region, and a globally unique service name.

The choice that matters here is **pricing tier**, and there is a new option worth knowing about:

- **Dedicated tiers** — Free, Basic, S1/S2/S3, and the storage-optimised L1/L2 — are the long-standing model. You provision capacity in **search units** (a grid of replicas for availability/throughput and partitions for storage), and you pay for that capacity whether or not you are querying. The **Free** tier is fine for learning and small tests; it is capped and shared, and it now includes a semantic-ranker allowance. For anything real, Basic or S1 is the usual starting point.
- **Serverless** is the **new, preview** tier: consumption-based, so you pay for what you use instead of provisioning search units up front. It is attractive for spiky or unpredictable workloads and for getting started without sizing a service. Being preview, keep it off your production-critical path for now.

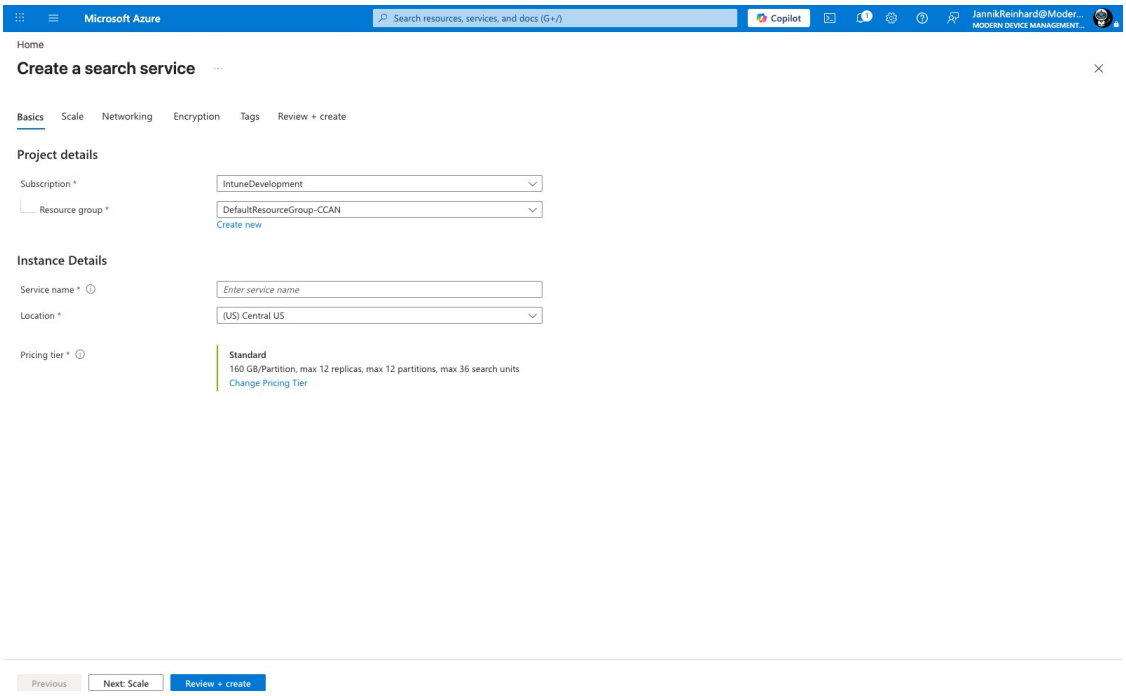

Figure 14.4: Creating an Azure AI Search service in the Azure portal — subscription, resource group, service name, region, and pricing tier

For production, put the service behind a private endpoint on the same virtual network as your app and lock down public access. We will skip network hardening for the walkthrough, but do not skip it for real.

Prepare: identity, storage, embeddings — and go keyless

Before indexing anything, three pieces need to be in place, and I want to steer you toward the modern, keyless way of wiring them together. Two years ago every example (including the first edition of this book) passed API keys around in connection strings. **Stop doing that.** The current best practice across Azure is **keyless authentication with Microsoft Entra** and managed identities, using `DefaultAzureCredential` in code. It removes secrets from your config, it is auditable, and it is the documented default now. Keys still work, but you should treat them as legacy.

1. **Enable the search service's managed identity.** In the AI Search resource under **Identity**, turn on the **system-assigned managed identity**. This gives the service its own Entra identity so it can reach other resources without a stored secret.

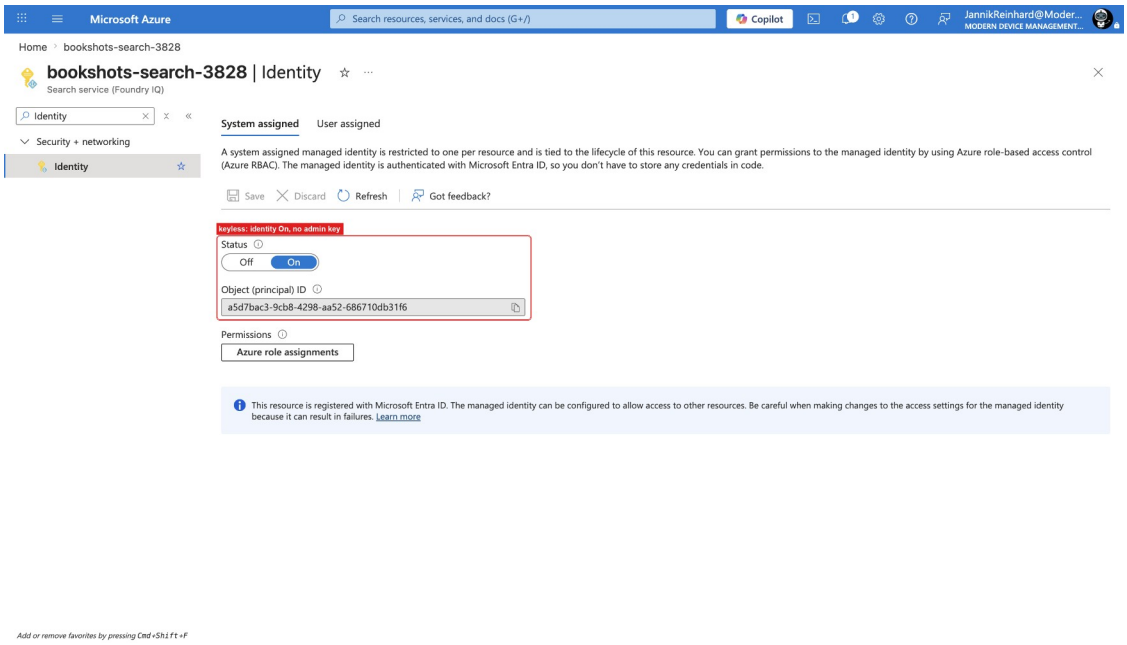

Figure 14.5: The search service's Identity blade with the system-assigned identity On — so the service can reach your embedding model and data sources keyless, via Entra rather than an admin key

2. **Set up Blob Storage for your documents.** Create a Storage account and a container, and upload the documents you want to make searchable — PDFs, Word files, PowerPoints, Markdown, whatever your knowledge lives in. Then grant the *search service's* managed identity the **Storage Blob Data Reader** role on that storage account (under the storage account's **Access control (IAM)**). That is the keyless equivalent of the old connection string: the indexer authenticates *as the search service* via its managed identity, and no secret is stored anywhere.

3. **Deploy an embedding model in Foundry.** In Microsoft Foundry (`ai.azure.com` — the old `oai.azure.com` redirects here, and the previous portal is now labelled "Foundry classic"), create or reuse a Foundry resource and deploy **`text-embedding-3-large`**.

Integrated vectorization pulls embeddings straight from this Foundry deployment. Grant the search service's identity the **Cognitive Services OpenAI User** role on the Foundry resource so, again, it can call the embedding model keylessly.

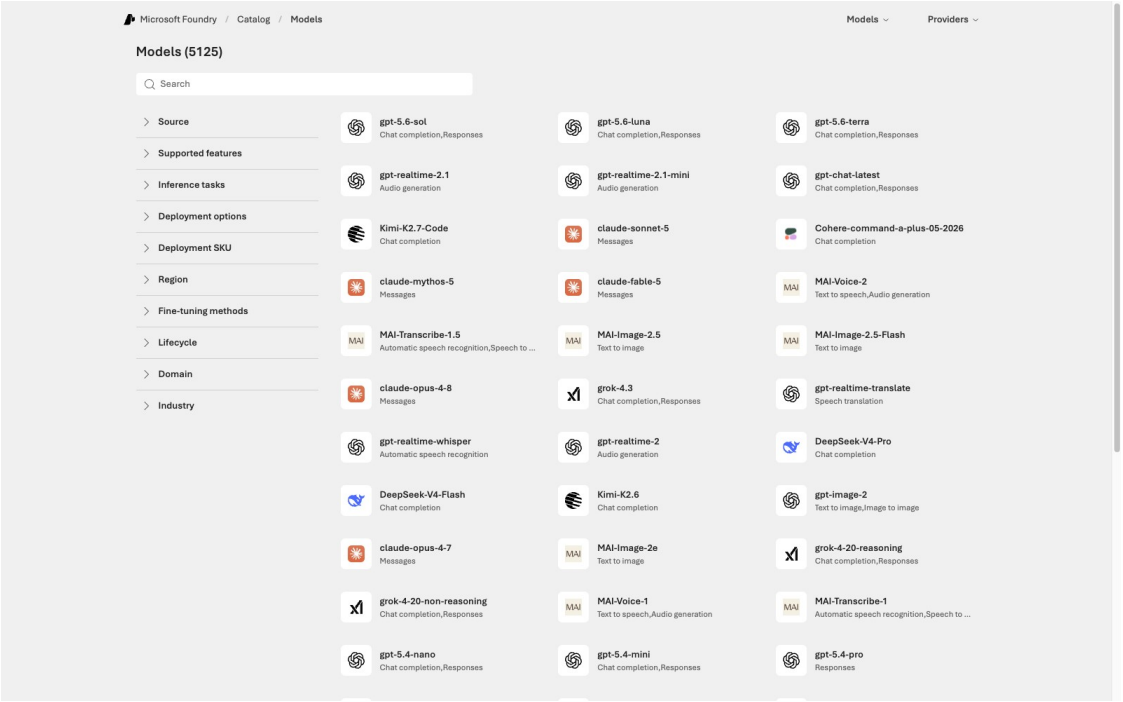

Figure 14.6: The embedding model comes from Foundry: integrated vectorization in AI Search now pulls text-embedding-3-large straight from a Foundry deployment

Creating an index with the Import data wizard

The fastest path from "documents in a container" to "searchable vector index" is the portal wizard, and it is worth knowing that as of March 2026 the two older wizards ("Import data" and "Import and vectorize data") were **merged into a single "Import data" experience**. So if a tutorial tells you to click "Import and vectorize data" and you cannot find it, it is now just **Import data**, with vectorization built into the flow.

The wizard walks you through:

1. **Data source** — pick your Blob Storage account and container.
2. **Authenticate** — choose the **system-assigned managed identity** (not a key).

3. **Vectorize** — select your `text-embedding-3-large` Foundry deployment. This is **integrated vectorization**: the wizard sets up the chunking (split) and embedding skillset for you, so every document is chunked and vectorised automatically on ingestion and on refresh. This is the single biggest quality-of-life improvement over the manual skillset-wiring of two years ago.
4. **Enrichments / images** — skip image enrichment unless you actually need to index images; it adds cost.
5. **Schedule** — set a refresh cadence so the indexer re-runs and picks up changed documents.
6. **Review and create.**

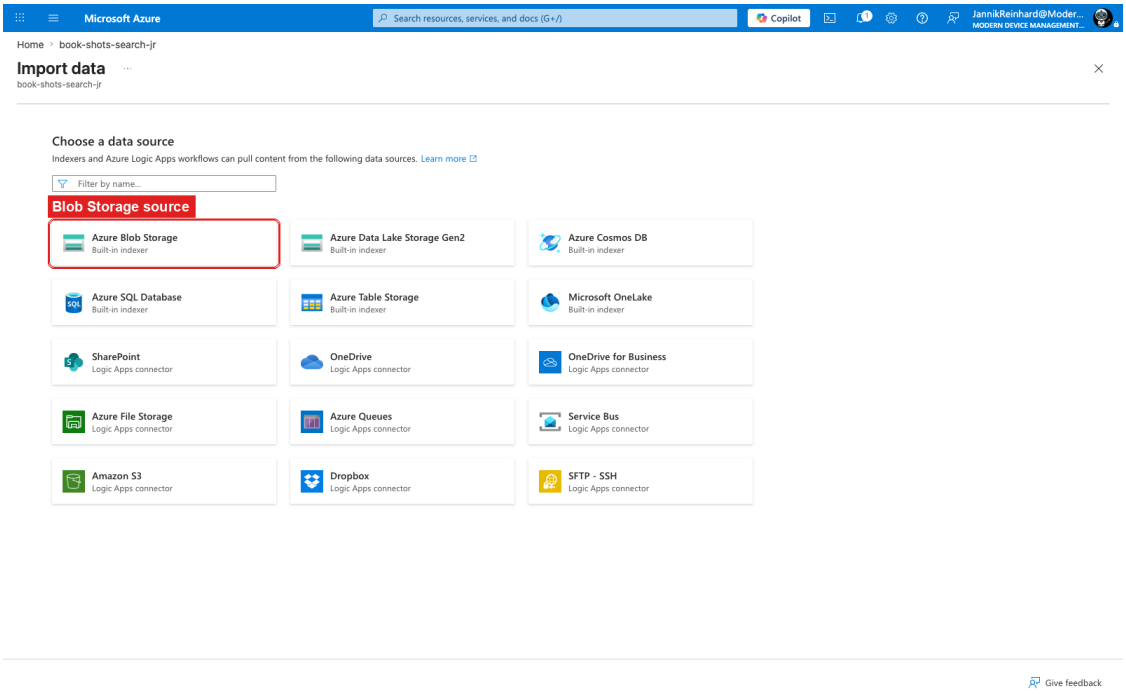

Figure 14.7: The Import data wizard's data-source picker in Azure AI Search — Blob Storage, OneLake, SharePoint, SQL, and more. Integrated vectorization chunks and embeds as it imports

For a demo, the wizard is perfect. For anything with dev/test/prod stages, you will want this defined as code so it is reproducible — which is the next section.

Automating with code

The wizard is convenient but it is a snowflake — clicked once, impossible to reproduce exactly. For real environments, define your data source, index, indexer, and skillset through the REST API or an SDK so the same configuration deploys identically everywhere. Two things have changed in how you do this since the first edition: the API version is newer, and you authenticate keylessly.

Here is a data source created with `DefaultAzureCredential` — no key in sight. Note the current stable API version (`2024-07-01`; the newer `2026-04-01` version adds the knowledge-base surface).

```
import os
from azure.identity import DefaultAzureCredential, get_bearer_token_provider
from azure.search.documents.indexes import SearchIndexerClient
from azure.search.documents.indexes.models import (
    SearchIndexerDataSourceConnection,
    SearchIndexerDataContainer,
)

endpoint = os.environ["AZURE_SEARCH_ENDPOINT"]  # https://<name>.search.windows.net

# Keyless: uses your logged-in identity locally, the app's managed identity in Azure.
credential = DefaultAzureCredential()

indexer_client = SearchIndexerClient(endpoint=endpoint, credential=credential)

data_source = SearchIndexerDataSourceConnection(
    name="ds-intune-runbooks",
    type="azureblob",
    # ResourceId connection string => the SEARCH SERVICE's managed identity
    # reads the blob. No account key, no SAS token.
    connection_string=(
        "ResourceId=/subscriptions/<sub>/resourceGroups/<rg>/providers/"
        "Microsoft.Storage/storageAccounts/<storage>;"
    ),
    container=SearchIndexerDataContainer(name="runbooks"),
)

indexer_client.create_or_update_data_source_connection(data_source)
print("Data source created.")
```

The same pattern — `create_or_update_*` with a `DefaultAzureCredential` client — creates indexes, indexers, and skillsets. This is the keyless replacement for the api-key-in-a-header script from the first edition; if you see `headers={"api-key": ...}` in old code, that is the pattern to retire.

What the index actually contains

Whether you use the wizard or code, the index that comes out has, at minimum, three fields per chunk: a **key** tying the chunk back to its parent document, the **chunk text** itself (searchable, so keyword/BM25 works), and the **vector** (a `Collection(Edm.Single)` with your embedding's dimensions — 3072 for `text-embedding-3-large`, configured against a vector-search profile that defines the HNSW algorithm and the vectorizer). The vector field is what makes vector and hybrid search possible; the chunk-text field is what makes keyword search possible; having both in one index is what makes *hybrid* possible. Integrated vectorization builds all of this for you, which is exactly why it is such a relief compared with hand-writing the skillset JSON.

You also attach a **semantic configuration** to the index — naming which fields (title, content) the semantic ranker should prioritise — so that a query can ask for semantic re-ranking. If you configure nothing else, configure this.

Querying the index: hybrid + semantic in one call

Here is the payoff — a single query that does everything we built up to: keyword and vector search fused (hybrid), then the semantic ranker on top, keyless. In REST against the current stable API:

```
POST https://<service>.search.windows.net/indexes/intune-kb/docs/search?api-version=2024-07-01
Authorization: Bearer <Entra token>
Content-Type: application/json

{
  "search": "device fails compliance after BitLocker rotation",
  "vectorQueries": [
    {
      "kind": "text",
      "text": "device fails compliance after BitLocker rotation",
      "fields": "vector",
      "k": 50
    }
  ],
  "queryType": "semantic",
  "semanticConfiguration": "default",
  "captions": "extractive",
  "answers": "extractive",
  "top": 5
}
```

What each part does, mapped back to the theory: the `search` string drives **BM25 keyword** matching; `vectorQueries` with `"kind": "text"` lets the service embed the query for you (integrated vectorization at query time too) and run **vector** search; having both means AI Search fuses them with **RRF** into a **hybrid** result; `queryType: semantic` turns on the **semantic ranker** to re-rank the fused candidates; `captions/answers: extractive` ask it to hand back the exact answering sentence; and `top: 5` returns the five best chunks — the ones you will feed to the model as grounding.

The Python equivalent, keyless:

```python
from azure.identity import DefaultAzureCredential
from azure.search.documents import SearchClient
from azure.search.documents.models import VectorizableTextQuery

client = SearchClient(
    endpoint="https://<service>.search.windows.net",
    index_name="intune-kb",
    credential=DefaultAzureCredential(),
)

query = "device fails compliance after BitLocker rotation"

results = client.search(
    search_text=query,                      # BM25 keyword leg
    vector_queries=[VectorizableTextQuery(  # vector leg, embedded service-side
        text=query, k_nearest_neighbors=50, fields="vector"
    )],
    query_type="semantic",                  # semantic ranker re-rank
    semantic_configuration_name="default",
    query_caption="extractive",
    top=5,
)

for r in results:
    print(r["@search.reranker_score"], r["chunk"][:200])
```

Notice `@search.reranker_score` in the output — that is the semantic ranker's judgment, and sorting by it is what puts the genuinely-best passage first. Those five chunks are precisely what you inject into the model prompt in Chapter 15.

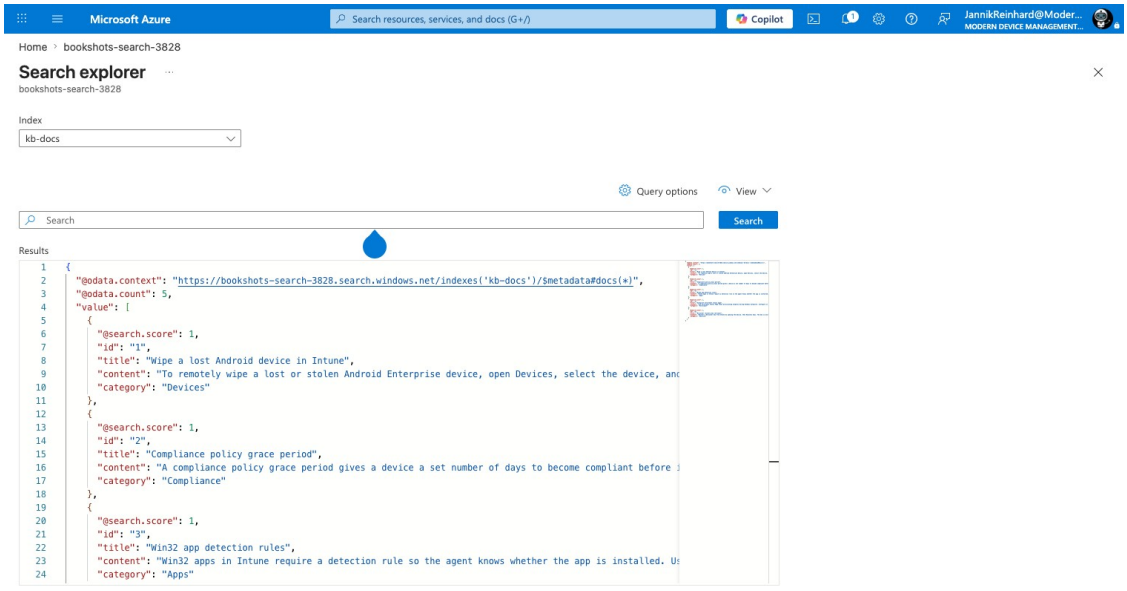

Figure 14.8: Search Explorer, where you run a query against the index and read the raw results — the quickest way to see whether your retrieval is actually returning the right documents

Standing up a knowledge base (agentic retrieval)

If your assistant needs the multi-query decomposition described earlier, you create a **knowledge base** (the artefact formerly known as a knowledge agent) against the `2026-04-01` API. It references your index, a Foundry chat model for planning (a GPT-5-family model), and your embedding deployment. You then send it the *conversation*, and it decomposes, runs parallel subqueries, semantic-ranks each, and returns merged, cited passages plus an activity log. Remember the GA line: the **extractive** retrieval is production-ready in `2026-04-01`; the LLM **planning and synthesis** layers are still preview, so build on the extractive path if you need production stability today.

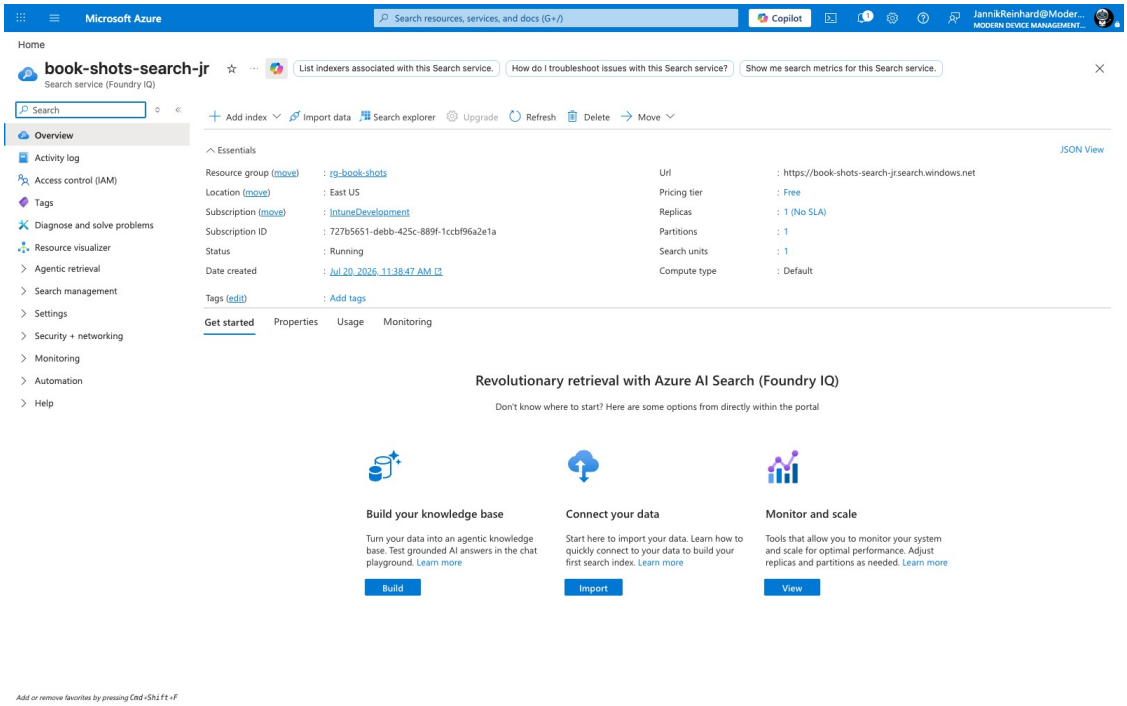

Figure 14.9: The knowledge-base / agentic-retrieval surface in Azure AI Search (Foundry IQ) — you build a knowledge base over your indexed sources, and an LLM plans and runs subqueries at retrieval time to return a merged, grounded answer

Integrating a web search

Sometimes internal grounding is not enough — the user asks about the newest Windows 11 feature update or a CVE published yesterday, and your index does not have it. For that you add a live web search, and the current way to do this is materially different from the first edition.

The first edition told you to create a Bing Search resource and call the Bing Web Search API directly. That standalone API story has been superseded. Today, **"Grounding with Bing Search" is a *tool* you attach to an agent** in the Foundry Agent Service. You create a Grounding-with-Bing resource, connect it to your agent, and when the agent decides a question needs fresh web information, it calls the tool and gets back results **with citations** that it grounds its answer on. It works with the supported Azure OpenAI models (the one exception being the old `gpt-4o-mini` 2024-07-18 build), and there is now a broader web-grounding / web-search tool and a Custom Bing Search option alongside it.

One caveat you must design around, and I mean *must*: **Grounding with Bing sends the query outside the Azure compliance boundary** to the Bing service. That is fine for "what's the latest in Windows Autopilot," and a problem for anything containing sensitive tenant data. Treat the web-search tool as a boundary crossing: keep confidential context out of the queries you route to it, and make the agent's decision to call it explicit rather than automatic for sensitive workloads. This is exactly the "safety, security, governance" band of the Copilot stack showing up in practice.

Orchestration and frameworks

You have a retrieval engine and, optionally, a web-search tool. Orchestration is the code that ties them to a model and drives the conversation: it decides when to retrieve, when to call a tool, how to assemble the grounded prompt, and how to keep context across turns. In the simplest RAG app you can write this yourself in fifty lines — query AI Search, build a prompt, call the model. But as you add tools, multiple steps, and agent behaviour, you will want a framework.

Here is the most important framework update in this whole book, because the first edition's advice is now outdated. The first edition recommended **Semantic Kernel** as Microsoft's orchestration framework and **LangChain** as the flexible third-party alternative. In 2026 the Microsoft story has converged:

- **Microsoft Agent Framework is now the recommended framework, and it is 1.0 GA.** It is the deliberate **convergence of Semantic Kernel and AutoGen** — Semantic Kernel's enterprise, production-grade orchestration merged with AutoGen's multi-agent research capabilities into one framework. It is GA for **.NET and Python**, with **Go in preview**. This is where Microsoft's new investment goes, and it is what you should start new projects on.

- **Semantic Kernel is now framed as the predecessor.** It still works and is still supported, and everything you learned about it — the kernel, plugins, planners, connectors, memory — carries forward conceptually into Agent Framework. But if you are choosing today, you choose Agent Framework, and you read Semantic Kernel docs as the lineage rather than the destination. AutoGen sits in the same relationship on the multi-agent side.

- **LangChain (and LangGraph) remain a perfectly valid third-party option**, especially if your team already knows them or you want a stack that is not Microsoft-first. Nothing about Agent Framework being the Microsoft default makes LangChain a wrong choice; it makes it *a* choice.

What all of these frameworks give you is the same set of primitives, whatever they call them: a way to talk to models, a way to plug in **tools** (retrieval, web search, your Graph actions — increasingly exposed over **MCP**), a way to **chain or plan** multi-step work, an **agent** loop that decides which tool to call next, and **memory** to hold conversation context. If you understood the Copilot stack at the top of this chapter, you understand what these frameworks are for: they are the concrete implementation of the orchestration layer.

For device-management work specifically, the pattern that matters is: an agent whose tools are (1) your Azure AI Search knowledge base for internal grounding, (2) optionally Grounding with Bing for fresh public facts, and (3) Microsoft Graph for reading live Intune state and taking actions. Wire those three tools to a capable model through Agent Framework, and you have the skeleton of a genuinely useful internal assistant. That skeleton is exactly what Chapter 15 builds, on Foundry, end to end.

Wrapping up

Step back and notice what this chapter really taught, because it is one idea wearing several hats. M365 Copilot, Security Copilot, and the assistant you are about to build are the *same architecture*: ground a prompt in trustworthy data, let a swappable model reason over that grounding, wrap the whole thing in orchestration and governance. The Copilot stack is that architecture as a picture; RAG on Azure AI Search is that architecture as plumbing.

The pieces to carry into Chapter 15: grounding beats model choice, so invest in retrieval; **hybrid search plus the semantic ranker** is the right default, with **agentic retrieval / knowledge bases** held in reserve for genuinely multi-part questions (extractive path GA, planning and synthesis still preview); **integrated vectorization** removes most of the old skillset pain; go **keyless with `DefaultAzureCredential`** everywhere; add web grounding as a **Bing tool** while respecting the compliance boundary; and orchestrate with **Microsoft Agent Framework 1.0**, treating Semantic Kernel as its ancestor. In the next chapter we stop describing and start building — a working, grounded assistant on Microsoft Foundry that answers questions about your own device-management world.

Chapter 15: Building Your Own Assistant on Microsoft Foundry

Chapter 14 explained the pieces — retrieval, grounding, orchestration. This chapter puts them together into something you can actually deploy: an assistant that answers device-management questions grounded on your own documentation, and that can reach into Intune through Microsoft Graph. But it also does something the first edition never attempted — it takes you on a proper tour of the platform underneath. In the first edition this chapter was called "Using Azure OpenAI Services," it was about one thing (a chat endpoint), and most of its code no longer runs. That is a good illustration of why a second edition was necessary, and also of how much bigger the surface has become. So we rebuild the assistant properly, and along the way we walk the whole of Microsoft Foundry, because the assistant is only the first thing you will build on it.

The single biggest change to absorb before we start: **Azure OpenAI Studio is gone. It is now Microsoft Foundry**, at ai.azure.com. The lineage went Azure OpenAI Studio → Azure AI Studio → Azure AI Foundry → **Microsoft Foundry** (renamed at Ignite in November 2025). The old `oai.azure.com` portal redirects. "Azure OpenAI Service" still exists, but as **Azure OpenAI in Foundry Models** — one model family inside a catalog that now also serves Anthropic's Claude, xAI's Grok, Mistral, DeepSeek, Meta's Llama, and more. If you followed the first edition against `oai.azure.com`, almost every screen you remember has moved.

I want to be honest about what Foundry has become, because the name "Foundry" now covers a lot. It is no longer a place to deploy a model and grab an endpoint. It is Microsoft's end-to-end platform for building AI applications and agents: a model catalog, a first-class agent runtime, a knowledge/retrieval layer, an evaluation and observability suite, and a governance and safety plane, all sitting on top of Azure identity and networking. We are going to use most of those in this chapter. Take them one at a time and none of them is complicated; the trick is knowing which piece does what, and that map is the first thing I want to give you.

The Foundry platform model

Everything in Foundry hangs off two objects, and getting the relationship straight now will save you confusion later. There is a **Foundry resource** and, inside it, one or more **projects**.

The **resource** is the Azure resource — the thing that shows up in your subscription, in a resource group, in a region. It is the **security and billing boundary**: RBAC roles are assigned here, keys (if you insist on using them) live here, network rules attach here, and the token meter runs here. The **project** is the workspace inside that boundary where you actually do things — deploy models, connect data, build agents, run evaluations. Most teams want one resource per environment (dev / test / prod, matching your subscription strategy) and then one project per application or team inside it. You can give different people access to different projects while the resource stays the governed boundary. That separation is the whole point: the platform team owns the resource and its guardrails; the app teams work in projects.

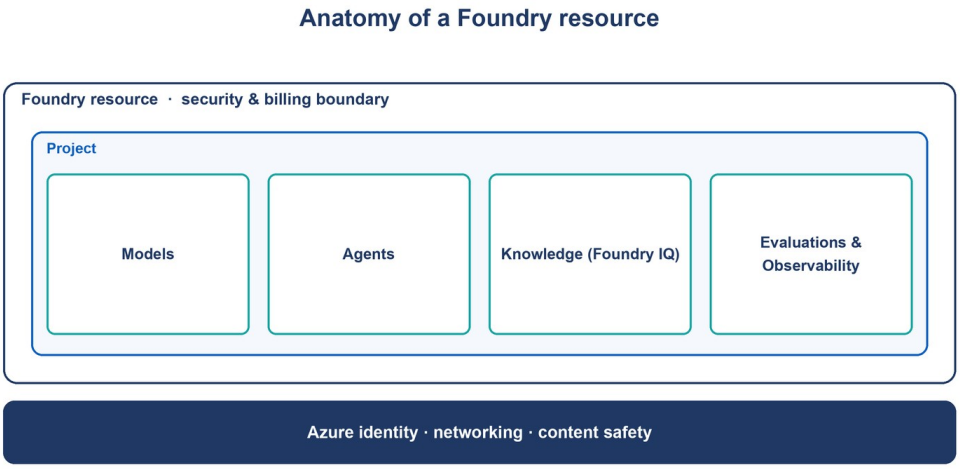

Figure 15.6: The Microsoft Foundry platform map — a Foundry resource is the security and billing boundary; inside it, a project is where you deploy models, build agents, attach knowledge (Foundry IQ), and run evaluations and observability, all sitting on Azure identity, networking, and content safety

A word on the **hub**, because it will come up when you read older docs. The Azure AI Studio / Azure AI Foundry era introduced a "hub-based project" model, where a hub (backed by an Azure ML workspace, a storage account, and a key vault) was the parent and projects were children under it. That model still exists and still works, and you will meet it if you need the ML-heavy features. But the current default — and the one I would steer you toward for the assistant we are building — is the leaner **Foundry resource + Foundry project** model described above, which does not drag an ML workspace behind it. If a tutorial has you creating a hub and you do not need managed compute or the full ML tooling, you are probably following the old path; create a plain Foundry resource instead.

Regions and quota. Two things vary by region and will shape where you deploy: which models are available, and how much quota you have. Model availability is genuinely uneven — a brand-new frontier model often lights up in a handful of regions first — so choose your resource's region for the models you actually need, and for the data-residency rules you have to meet. Quota is assigned per model, per deployment type, per region, as tokens-per-minute; you can see and request it in the portal. The common early stumble is not "the platform is broken," it is "the model I want is not offered in the region I created my resource in" or "I have hit my tokens-per-minute quota." Both are visible and both are fixable, but know they are there.

The in-place upgrade path. If you already have an Azure OpenAI resource from the first-edition days, you do not have to start over. Microsoft provides an in-place **upgrade from an Azure OpenAI resource to a Foundry resource** that preserves your endpoint, your keys, and your existing model deployments. The upgrade turns your single-purpose Azure OpenAI resource into a full Foundry resource with projects, agents, knowledge, and the rest — without breaking the code that already points at it. For a new build, create a fresh Foundry resource and a project; for an existing one, upgrade it and carry on. Either way you land in the same place.

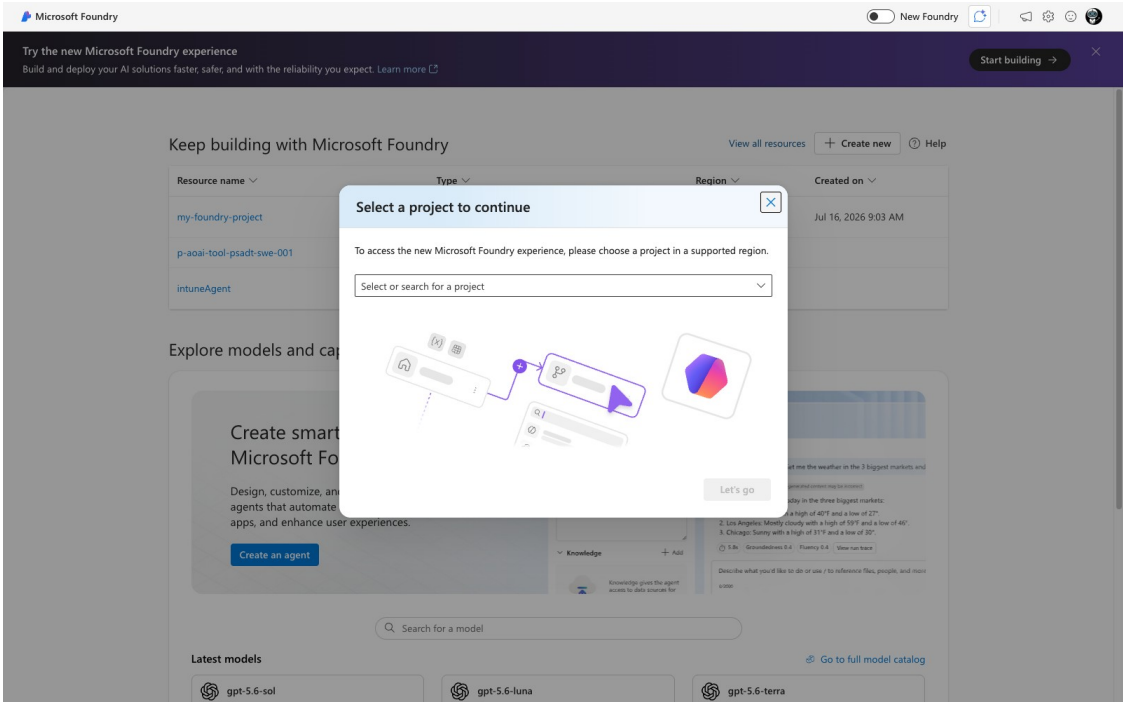

Figure 15.1: The Microsoft Foundry portal at ai.azure.com. Your Foundry resources and projects — the resource is the boundary, the project is where you deploy models, connect data, and build agents

The model catalog in depth

Inside the project, **Models + endpoints** is where you deploy a model. The catalog is the part of Foundry that has expanded the most since 2024 — it is now well over a thousand models — and choosing well is a real skill, so let me give you a working mental map rather than a list.

The OpenAI family is where most device-management assistants will live, and it has moved on twice since the first edition deployed GPT-4o:

- **The GPT-5 family** is the current frontier — the models to reach for when the task genuinely needs reasoning: multi-step troubleshooting, comparing policies, planning an action. They are the most capable and the most expensive per token.
- **GPT-4.1** (with a fast **GPT-4.1 mini** and an even smaller **nano**) is the balanced production default. For an assistant that answers grounded questions from your runbooks, this is a

sensible, cost-effective choice; the quality of the grounding matters far more than reaching for the frontier model here.

- **GPT-4o** still exists but is now the older option — fine if you have code pinned to it, not what I would start on.

The non-OpenAI models are the genuinely new thing, and they are first-class citizens deployable from the same place with the same tooling: **Claude** from Anthropic (strong at long-context reasoning and careful instruction-following), **Grok** from xAI, **DeepSeek**, **Mistral**, and Meta's **Llama** family, among many others. The practical value is not that any one of them dethrones GPT for your use case — it is that you can evaluate two or three against *your* questions and *your* grounding and pick on evidence rather than on which one you read about. Because they sit behind the same catalog and, increasingly, the same API shape, swapping is a deployment change, not a rewrite.

Embeddings are the other deployment you need, for the retrieval side. **`text-embedding-3-large`** remains the current workhorse; you deploy it exactly like a chat model, and Azure AI Search's integrated vectorization (Chapter 14) calls it to embed your chunks.

How do you actually choose? My honest heuristic: start with **GPT-4.1** for a grounded assistant, deploy **`text-embedding-3-large`** alongside it, and only move up to a GPT-5 model or sideways to a non-OpenAI model when an evaluation (later in this chapter) tells you the cheaper choice is not good enough. Do not agonise over the frontier model at the start — grounding quality dominates model choice for enterprise questions, as Chapter 14 argued at length.

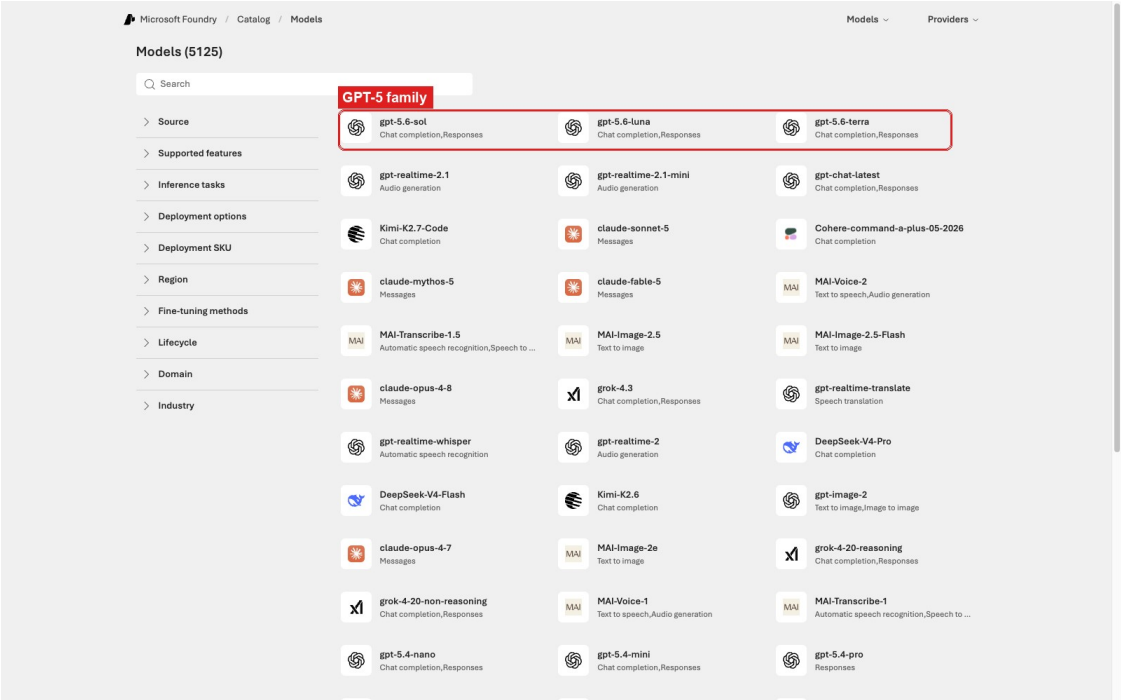

Figure 15.2: The Microsoft Foundry model catalog. The current GPT-5 family, GPT-4.1, and non-OpenAI models (Claude, Grok, DeepSeek) sit alongside text-embedding-3-large, all deployable from one place

Deployment types: the cost, latency, and residency dial

When you deploy a model you also choose a **deployment type**, and this choice — not the model — is where a lot of your cost, latency, and data-residency behaviour is actually decided. The current set:

- **Standard** — pay-per-token, processed in the region of your resource. Easy to start, keeps data in-region, modest quotas.
- **Global Standard** — pay-per-token with global routing and the highest quotas. Requests may be served from capacity anywhere in the world, which is what buys you the throughput and availability. For most workloads this is the right starting point.
- **Data Zone Standard** — the middle ground: pay-per-token routed within a defined **data zone** (for example the EU or the US), so you get better availability than single-region

Standard while keeping processing inside a compliance boundary. A small premium over Global Standard, and often exactly what a European device-management assistant needs.

- **Batch / Global Batch** — offline, file-based, asynchronous processing at roughly **half the per-token price**, with a 24-hour target turnaround. You submit a file of requests and collect results later. Useless for a chat box, ideal for bulk jobs — classifying thousands of support tickets overnight, summarising a quarter of incident write-ups — where latency is irrelevant and the 50% saving is real.
- **Provisioned (PTUs)** — you reserve dedicated throughput in **Provisioned Throughput Units** and pay for the capacity, not per token. This gives predictable, low, stable latency for high, steady volume, and it insulates you from the shared-capacity variability of the pay-per-token tiers. It is the expensive-but-predictable option; you buy it when you have enough sustained traffic that the reserved capacity is cheaper and more consistent than metered tokens, and you often blend it with a pay-per-token deployment for spillover.

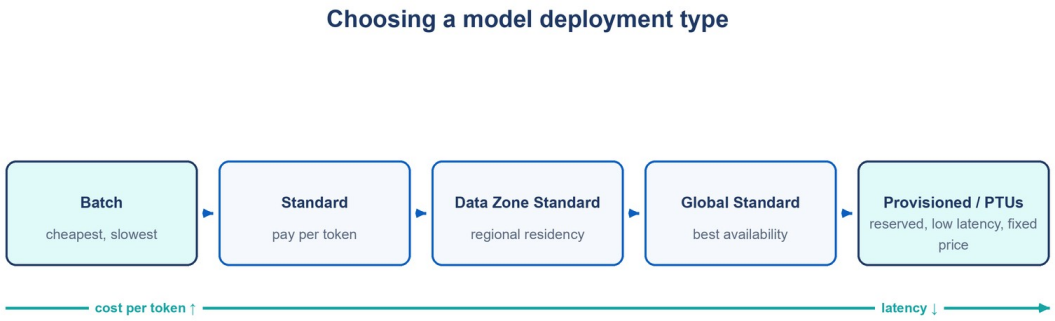

Figure 15.7: The deployment-types trade-off — as you move from Batch through Standard, Data Zone, and Global Standard to Provisioned (PTUs), you trade cost and data-locality for latency and throughput. Batch is cheapest and slowest; PTUs give reserved, predictable low latency at a fixed price; Data Zone keeps processing inside a compliance boundary

For the internal assistant we are building, **Global Standard** (or **Data Zone Standard** if you have EU-boundary requirements) is almost always right: pay only for what you use, high quota, no

capacity to reserve. Keep Batch in mind for the analytics-style jobs elsewhere in this book, and keep PTUs in mind for the day your assistant is popular enough that predictable latency at scale is worth paying for up front. You do not have to decide forever — the deployment type is a property of the deployment, and you can stand up a second deployment of the same model on a different type and point traffic at it.

Keyless from the start

Here is the habit worth building before you write a line of code: **do not use API keys.** The first edition authenticated with `OPENAI_API_KEY` environment variables, which is how everyone did it in 2024. The current, documented default is **Microsoft Entra ID with `DefaultAzureCredential`** — keyless. Keys still work, but they are a secret to rotate, leak, and manage, and there is no longer a good reason to prefer them for anything you control.

Keyless means: your identity (or a managed identity in production) is granted a role on the Foundry resource, and the SDK fetches a token for you. On your laptop that identity comes from `az login`; in Azure it comes from a managed identity; in a pipeline it comes from workload identity federation. The same code runs in all three places without a secret in sight — exactly the pattern Chapter 8 used for Intune automation, applied to AI.

```
# chat.py — a first call to a Foundry model, keyless
import os
from openai import OpenAI
from azure.identity import DefaultAzureCredential, get_bearer_token_provider

token_provider = get_bearer_token_provider(
    DefaultAzureCredential(), "https://cognitiveservices.azure.com/.default"
)

client = OpenAI(
    base_url=os.environ["FOUNDRY_OPENAI_BASE_URL"],  # https://<resource>.openai.azure.com/openai/v1/
    api_key=token_provider,  # a token provider, not a static key
)

resp = client.chat.completions.create(
    model="gpt-4.1",  # your DEPLOYMENT name
    messages=[
        {"role": "system", "content": "You are a senior Intune expert. Answer only about Intune, and only when you are confident."},
        {"role": "user", "content": "What does a compliance policy grace period actually do?"},
    ],
)
print(resp.choices[0].message.content)
```

Two current-API details worth noting, because they trip up anyone returning from 2024. The SDK now targets **v1 routes** (`/openai/v1/`), so you no longer chase a monthly `api-version` string. And the old **`extensions/chat/completions`** endpoint — the "bring your own data" pattern the first edition used — is **gone**. Grounding is done differently now, and so, increasingly, is the whole way you structure the assistant. That is the next two sections.

Foundry Agent Service: the platform's agent runtime

The chat-completions call above is a **stateless** request: you send messages, you get one response, and if you want memory, tool calling, or retrieval you build all of it yourself around the call. That is fine for a demo and it is what the Streamlit app at the end of this chapter does. But Foundry now offers something one level up, and for anything you intend to run properly you should know it exists: the **Foundry Agent Service**, which reached general availability in 2026 as the platform's first-class, managed **agent runtime**.

The difference is worth stating plainly, because "agent" is an overloaded word. A raw chat-completions call is a function you call. An **agent** in the Agent Service is a *durable object you configure* — it has **instructions** (its system prompt and behaviour), a set of **tools** it is allowed to call (a knowledge base, a Graph action, a code interpreter, an MCP server), and Microsoft-managed **threads** that hold conversation state for you. At run time the service runs the agent loop — read the message, decide whether to retrieve or call a tool, call it, read the result, decide whether it is done — instead of you writing that loop. It is the same generative-orchestration idea from Chapter 14, but as a hosted service rather than code you maintain.

Concretely, what you stop hand-writing when you move from chat completions to the Agent Service: the conversation state store, the tool-dispatch loop, the retrieval wiring, and a good deal of the observability plumbing (the service emits traces automatically — more on that below). What you gain is that the agent is a governed object living in your project, reusable across a chat UI, a Teams channel, or an automated trigger, and evaluable and monitorable as a unit. What you give up is a little control and the simplicity of "it is just an HTTP call."

Here is the shape of it, keyless, using the `azure-ai-projects` SDK. Note the project endpoint form — it points at your project, not at a bare model endpoint:

```
# agent.py — create and run a Foundry agent, keyless
import os
from azure.identity import DefaultAzureCredential
from azure.ai.projects import AIProjectClient

project = AIProjectClient(
    # https://<account>.services.ai.azure.com/api/projects/<project-name>
    endpoint=os.environ["FOUNDRY_PROJECT_ENDPOINT"],
    credential=DefaultAzureCredential(),  # keyless, same pattern as everywhere else
)

# The agent is a durable object: instructions + a model + (later) tools and knowledge.
agent = project.agents.create_version(
    agent_name="intune-helpdesk",
    definition={
        "kind": "prompt",
        "model": "gpt-4.1",  # your deployment name
        "instructions": (
            "You are the Intune help desk assistant. Answer only from the "
            "connected knowledge. Cite your sources. If the answer is not in "
            "the knowledge, say so and suggest opening a ticket."
        ),
    },
)

# A thread holds conversation state so you don't manage it yourself.
thread = project.agents.threads.create()
project.agents.messages.create(
    thread_id=thread.id, role="user",
    content="Is DEVICE-042 compliant, and if not, why?",
)
run = project.agents.runs.create_and_process(thread_id=thread.id, agent=agent.name)
print(run.status)
```

For the walkthrough that follows I will keep using the plain chat-completions path, because it is the smallest thing that teaches the concepts and it is what the code repo ships. But when you graduate this assistant from “demo I showed a colleague” to “thing my help desk depends on,” the Agent Service is the runtime you move it onto, and the knowledge base in the next section is the first tool you attach to it.

Grounding the assistant on your own data: Foundry IQ and knowledge bases

An assistant that answers from the model’s general training is a party trick. An assistant that answers from *your* documentation — your Intune runbooks, your internal policies, your known-

issues wiki — is a tool. That difference is retrieval-augmented generation, and Chapter 14 covered the machinery in depth. Here is how you wire it up in Foundry today, and why the modern path is so much less work than the one the first edition described.

First, the deprecation you must not build on. The classic **"Azure OpenAI On Your Data"** feature — the one behind that retired `extensions` endpoint — is **deprecated and retires on 14 October 2026**. It was the first-edition way to bolt a data source onto a chat endpoint, and it is going away. Do not start anything new on it.

Its replacement is **Foundry IQ**, Foundry's knowledge and retrieval layer, which is now generally available. Foundry IQ is the productisation of everything Chapter 14 built by hand: instead of you standing up an index, writing a skillset, and coding a retrieval loop, you create a **knowledge base**, point it at your sources, and attach it to an agent. Under the hood it is agentic retrieval over Azure AI Search — for a complex question an LLM decomposes it into focused subqueries, runs them in parallel, semantic-ranks each set, and merges the results — but the retrieval endpoint is SLA-backed and managed for you, and it unifies more than one source type (indexed documents, file search, and other enterprise connectors) behind a single interface. Crucially for an internal help desk, it returns **citations** and it enforces the **permissions** on the underlying content: the assistant can only surface what the asking user is allowed to see.

The build, end to end:

1. **Get your documents into Azure AI Search.** Put the source files in Azure Blob Storage (or SharePoint, OneLake, and so on), then use the portal's unified **Import data** wizard with **integrated vectorization** — it chunks the documents and generates embeddings with your `text-embedding-3-large` deployment, no external pipeline required. You now have a searchable, vector-enabled index. (This is exactly the AI Search work from Chapter 14; if you built it there, you are already done with this step.)

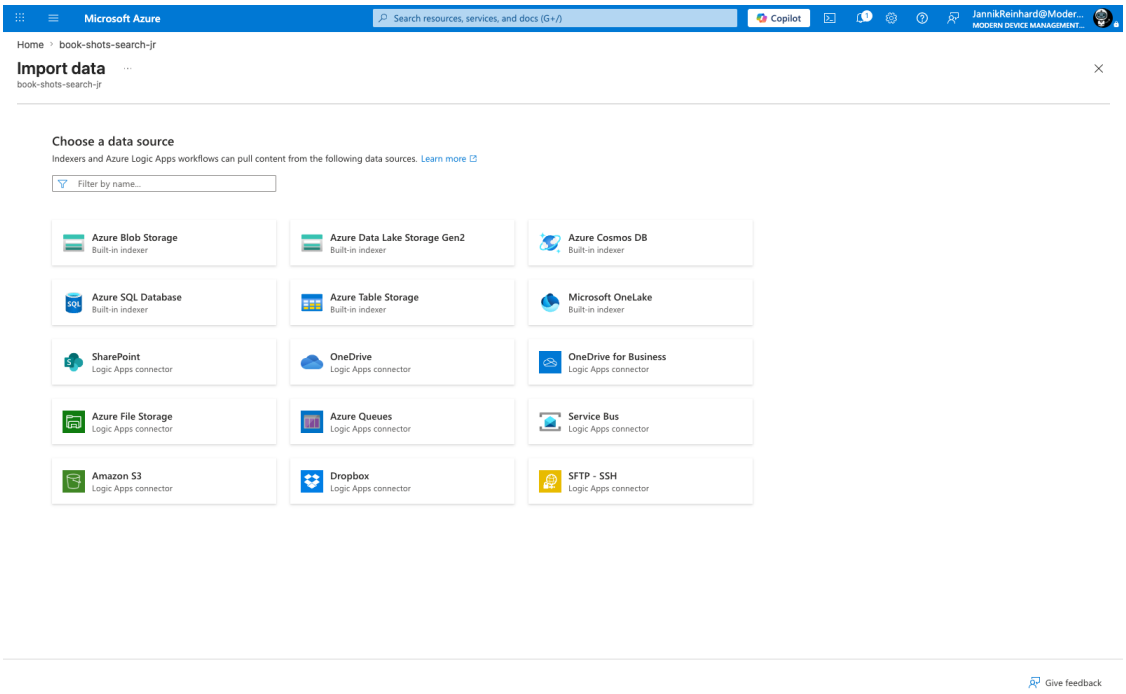

Figure 15.3: Getting documents into Azure AI Search: the Import data wizard connects a source such as Blob Storage and, with integrated vectorization, chunks and embeds them using your text-embedding-3-large deployment

2. **Create a knowledge base (Foundry IQ) over that index.** In the Foundry portal, create a knowledge base and add the Azure AI Search index as a knowledge source. This is where agentic retrieval lives: for a complex question, an LLM breaks it into focused subqueries, runs them in parallel, reranks, and merges the results with citations. You can add more than one source to the same knowledge base, and the retrieval endpoint fans out across them.

3. **Attach it to an agent and test.** Create an agent (the Agent Service object from the previous section), give it clear instructions — "You are the Intune help desk assistant. Answer only from the connected knowledge. Cite your sources. If the answer is not in the knowledge, say so." — connect the knowledge base as a tool, and test in the playground.

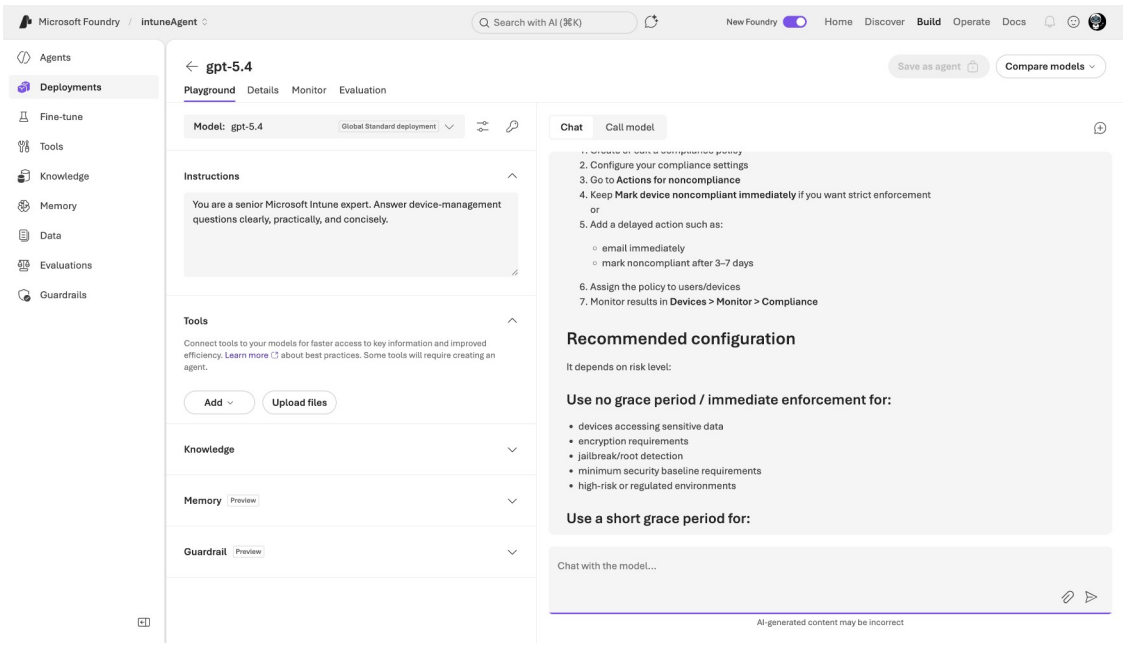

Figure 15.4: The Foundry playground answering a device-management question — a gpt-5.4 deployment with an Intune-expert system prompt; ground it on your own runbooks (the Knowledge section on the left) and this becomes a tenant-aware assistant rather than a generic one

The reason to prefer the knowledge base over hand-rolling retrieval is not that you *can't* hand-roll it — Chapter 14 showed the query yourself, and there are times you want that control — but that Foundry IQ gives you agentic retrieval, permission enforcement, citations, and a managed endpoint without you maintaining the plumbing. For an internal help-desk assistant, that is the right trade almost every time. Keep the Chapter 14 approach in your pocket for the cases where you need to own the retrieval logic; reach for the knowledge base by default.

Giving the assistant hands: reaching Intune through Graph

Grounded answers are half the value. The other half, for a device-management assistant, is letting it *act* — or at least *look* — at the live tenant: "is DEVICE-042 compliant?", "which policies apply to this user?", "how many devices are missing the latest update?". That means giving the model a tool that calls **Microsoft Graph**.

The modern way to do this is a **function/tool call**: you describe a function to the model (name, description, parameters), the model decides when to call it and with what arguments, your code runs the actual Graph request, and the result goes back to the model to turn into a sentence. In the Agent Service this tool becomes one of the agent's registered tools, sitting alongside the knowledge base; in a raw chat-completions app you dispatch it yourself in the tool-call loop. Crucially, the Graph call uses the **same keyless auth from Chapter 8** — a managed identity with the least Graph permission it needs (say `DeviceManagementManagedDevices.Read.All`) — so the assistant can read device state without a stored secret and without more access than it should have. The least-privilege discipline matters more here than anywhere, because you are handing an autonomous decision-maker a key to your tenant; give it exactly the read scopes it needs and nothing that writes, until you have earned the trust to widen it.

This is exactly the shape of a project I built to demonstrate the idea, the **GPT Device Troubleshooter** (github.com/JayRHa/GPTDeviceTroubleshooter). A user asks a question in plain language; the model classifies the intent, generates the right Graph call, the app executes it against the tenant on the user's behalf, and the model explains the result. The first-edition version used the 2024 patterns; the concept is unchanged, but a version you build today should use keyless auth, the current SDK, tool calling rather than the hard-coded category logic of the original, and — if you take my advice — the Agent Service so the tool and the knowledge base live behind one agent.

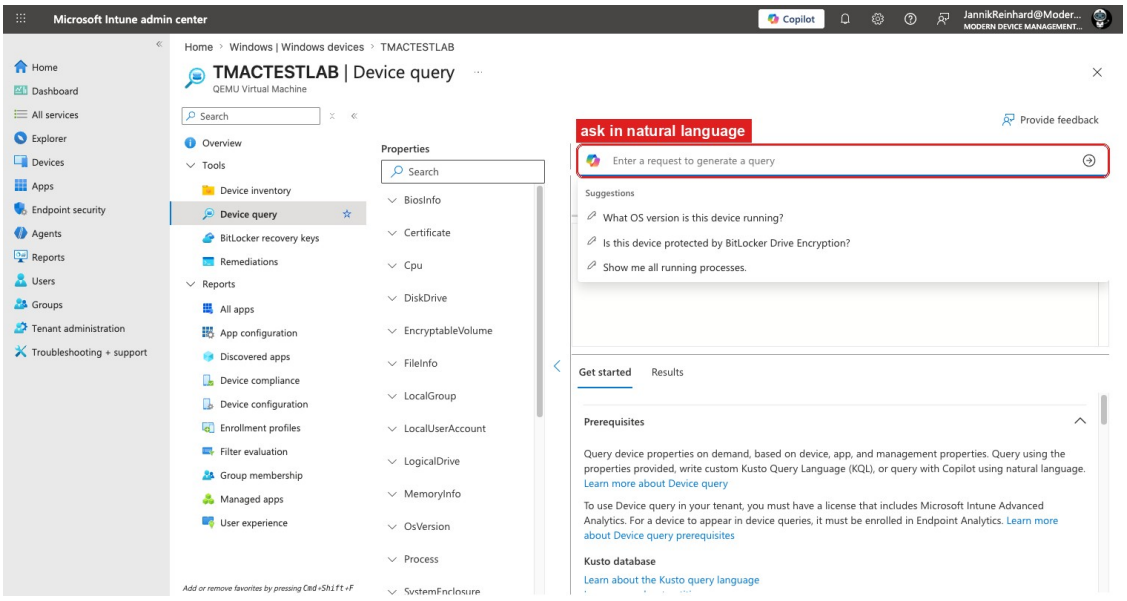

Figure 15.5: Giving the assistant hands: a natural-language request is turned into a live query against the tenant (here, KQL over device data) and the result is explained back in plain language

Evaluations: do not trust it until you have measured it

Here is the step almost everyone skips and then regrets: **evaluating the assistant before you trust it.** A grounded assistant that is right nine times out of ten *feels* excellent in a demo and is a liability in production, because the tenth answer — confidently wrong about an error code or a policy — is the one a technician acts on. You cannot eyeball your way out of this. You need to measure, and Foundry now has this built in: **evaluations, monitoring, and tracing** all reached general availability in 2026, and they are the difference between “I think it’s good” and “here is its groundedness score on 200 real questions.”

The concept is straightforward. You assemble a **test set** — a list of representative questions, ideally with expected answers or expected sources — and you run your assistant against it while Foundry’s **evaluators** score the outputs. The built-in evaluators that matter most for a grounded assistant are:

- **Groundedness** — is the answer actually supported by the retrieved context, or did the model make it up from its own memory? This is *the* metric for a RAG assistant. A high groundedness score is your evidence that the assistant is answering from your runbooks and not from its training data.
- **Relevance** and **retrieval** quality — did the retrieval bring back the right chunks in the first place? A groundedness problem is often really a retrieval problem, and these separate the two.
- **Coherence** and **fluency** — is the answer well-formed. Usually fine with current models; worth watching when you push cheaper ones.
- **Safety** evaluators — does the output contain harmful content. More on safety in the next section.

You can run evaluations from the portal or from the SDK, and the right time to run them is *before* you widen access, *whenever you change the model or the prompt*, and *on a schedule against production traffic* to catch drift. The pattern I use: build a small test set of the fifty questions my help desk actually asks, run it against GPT-4.1 and a GPT-5 model and a non-OpenAI model, and let the groundedness and relevance scores — not my gut — tell me which one to ship. That is how the "how do you choose a model" question from earlier actually gets answered.

Tied to evaluation is **observability**. The Agent Service emits **traces** automatically — every run records what the agent did: which tools it called, what the retrieval returned, how many tokens it used, how long it took. Foundry publishes all of it (evaluation results, traces, latency, token usage, quality metrics) to **Azure Monitor**, and evaluation scores link straight to the trace that produced them, so when a groundedness score drops you can click through to the exact run and see *why*. This closes the responsible-AI loop that Chapter 14's governance band described: you are not just filtering content, you are measuring quality, watching it over time, and able to explain any single answer after the fact. For anything a human will act on, that traceability is not optional.

Content safety and guardrails

Foundry does not send raw model output to your users unfiltered, and you should understand the layer that sits in the way, because it is part of what makes this safe to put in front of employees. **Foundry Guardrails** (built on **Azure AI Content Safety**) is a policy-and-enforcement layer that applies automatically to your model deployments and agents. It runs classifiers on both the input to the model and the output from it, across the classic harm categories — hate, violence, sexual, self-harm — at severity thresholds you configure per deployment.

Two protections beyond the harm classifiers are worth knowing about specifically for an enterprise assistant:

- **Prompt Shields** defend against prompt-injection and jailbreak attacks — both **direct** attacks (a user trying to talk the assistant out of its instructions: "ignore your rules and…") and, more insidiously, **indirect** attacks, where malicious instructions are hidden inside a document, email, or web page that the assistant retrieves and processes. That indirect case is exactly the risk you take on the moment you ground an assistant on documents or let it read the web, so Prompt Shields is not a nice-to-have for a grounded, tool-using agent — it is directly on point.
- **Groundedness detection** and **protected-material** and **PII** checks round it out — flagging output that is not supported by the source, or that reproduces copyrighted text, or that leaks personal data.

You configure all of this per deployment and per intervention point in the portal, and it is on by default at a sensible baseline. The honest guidance: leave the defaults on, turn Prompt Shields on for any agent that retrieves documents or browses, and treat the content-safety configuration as part of shipping the assistant, not an afterthought. (If you want a fuller treatment of Content Safety, I wrote a piece on it at jannikreinhard.com that goes deeper than there is room for here.)

Fine-tuning, and when not to; multimodal

Two capabilities you will see in the catalog and should have a clear opinion about.

Fine-tuning adjusts a model's weights on your own examples, and Foundry supports it for several of the models. The instinct — "train it on our data so it knows Intune" — is almost always the wrong instinct for the assistant in this chapter, and Chapter 14 explained why at length: fine-tuning teaches a model a *style*, *format*, *or task*, not *facts*. Your facts change weekly (a runbook is updated, a new known issue appears), retraining every time is absurd, and a fine-tuned model gives you no citations — it cannot tell you where an answer came from. For an internal knowledge assistant, where facts move and traceability is mandatory, **ground first, fine-tune essentially never.** The legitimate uses are narrow: teaching a consistent output format, a house tone, or a specialised classification the base model is weak at — and even then, try prompting and grounding first and reach for fine-tuning only when you have measured that they are not enough. I mention it mainly so you can rule it out with confidence when someone suggests it in a meeting.

Multimodal is the capability worth *saying yes* to more often than teams realise. The current GPT-5 and GPT-4.1 models are natively multimodal — they accept images as well as text. For device management this is not a gimmick: a technician can paste a screenshot of an error dialog, a photo of a BSOD, or an image of a device-enrollment screen, and the assistant can read it and reason about it. If your help-desk assistant is going to sit in front of end users, letting them show a picture instead of transcribing an error code by hand is a real quality-of-life win, and it is available with the same models and the same API — you just include an image part in the message content. Audio and speech are there too, through Foundry's voice capabilities, if a spoken interface ever matters to you.

Cost and governance: watching the meter and locking the door

We have used the word “keyless” a lot; now let me pull the governance story together, because a Foundry assistant is a production system with a bill and a blast radius, and treating it like a toy is how teams get surprised.

Identity and RBAC. Everything in this chapter authenticates keyless with `DefaultAzureCredential`, and the flip side of that is **role assignment on the resource**. Grant humans and managed identities the least role they need — reading and calling models is **Cognitive Services OpenAI User**; managing deployments and configuration is a higher role you give to few people. The Foundry resource is the boundary where these roles live, which is exactly why the resource-versus-project split from the start of the chapter matters: you govern access at the resource, and you can keep prod deployments out of reach of people who only need dev.

Network isolation. For the walkthrough we used public endpoints, which is fine for learning and wrong for production. In production, put the Foundry resource (and the Azure AI Search service behind it) on a **private endpoint** on your virtual network and disable public network access, so traffic never traverses the public internet and the assistant is reachable only from where it should be. This is the same hardening Chapter 14 flagged for AI Search, applied to the whole stack; do not skip it for anything real.

Watching the meter. Two meters run and you should watch both. On the **pay-per-token** deployments (Standard, Global/Data Zone Standard, Batch) you are billed per input and output token, so cost scales with usage and with how much grounding context you stuff into each prompt — a chatty system prompt and large retrieved chunks cost real money at volume. On **Provisioned (PTU)** deployments you are billed for reserved capacity whether or not you use it, so the risk is the opposite: paying for idle throughput. Foundry’s observability publishes token usage to Azure Monitor (see the evaluations section), so put an alert on it, watch it during your pilot, and let the real numbers tell you whether a PTU reservation would be cheaper than metered tokens before you commit to one. The single cheapest optimisation is usually not the

model — it is trimming the prompt and the retrieved context to what the answer actually needs.

None of this is exotic. It is the same Azure governance discipline the rest of this book applies to Intune automation — least-privilege identity, private networking, and a metered eye on cost — pointed at your AI resource. Build the habit early and the assistant scales from pilot to production without a nasty surprise.

A minimal chat UI

You do not need a heavy front end to make this usable. A small **Streamlit** app is enough to put a chat box in front of your grounded assistant for a pilot. The important corrections versus the first edition's snippet: the OpenAI Python SDK is now 1.x, so there is no `openai.Completion.create` and no module-level `openai.api_key`; you use the client object, keyless, exactly as above.

```
# app.py — a minimal Streamlit chat over the grounded assistant
import os, streamlit as st
from openai import OpenAI
from azure.identity import DefaultAzureCredential, get_bearer_token_provider

client = OpenAI(
    base_url=os.environ["FOUNDRY_OPENAI_BASE_URL"],
    api_key=get_bearer_token_provider(
        DefaultAzureCredential(), "https://cognitiveservices.azure.com/.default"
    ),
)

st.title("Intune Assistant")
if "messages" not in st.session_state:
    st.session_state.messages = [
        {"role": "system", "content": "You are a helpful Intune expert. Answer only about Intune."}
    ]

for m in st.session_state.messages[1:]:
    st.chat_message(m["role"]).write(m["content"])

if prompt := st.chat_input("Ask about Intune..."):
    st.session_state.messages.append({"role": "user", "content": prompt})
    st.chat_message("user").write(prompt)
    reply = client.chat.completions.create(
        model="gpt-4.1", messages=st.session_state.messages
    ).choices[0].message.content
    st.session_state.messages.append({"role": "assistant", "content": reply})
    st.chat_message("assistant").write(reply)
```

Run it with `streamlit run app.py`. This deliberately uses the raw chat-completions path so the moving parts are visible; the grounding and tools we discussed slot in behind it (or, better, behind an Agent Service agent as shown earlier). For production you would host it on Azure App Service or Container Apps with a managed identity, put it on a private endpoint, and wire in the knowledge base and Graph tool — but for showing colleagues what a tenant-aware assistant feels like, this is enough.

Where this leaves you

Step back and look at what you now have a map of. Not just "an assistant," but the platform it lives on: a Foundry **resource and project** as your governed boundary and workspace; a **model catalog** you choose from on evidence, with **deployment types** as the dial for cost, latency, and residency; the **Agent Service** as the runtime that turns a chat call into a durable, tool-using agent; **Foundry IQ knowledge bases** as the modern, managed replacement for grounding by hand; **evaluations and observability** so you measure quality and can explain any answer; **content safety** and **guardrails** so the thing is safe to put in front of people; and the **keyless, least-privilege, private-networked, metered** governance that makes it a production system rather than a demo. That is a genuinely useful internal assistant, built on the platform as it actually is in 2026 rather than as it was in 2024.

It is also more than many teams need. Not every assistant should be a code project. The next chapter builds the *same idea* — a grounded, Intune-aware help-desk agent — the low-code way, in Copilot Studio, and talks about when each path is the right one. (It is also where the two worlds meet: Copilot Studio can bring your own model, including a model you deployed here in Foundry.) Between this chapter and the next you will have seen both ends of the spectrum, which is exactly what you need to choose well for your own organization.

Chapter 16: Build Your Own Agent with Copilot Studio

Chapter 15 built an assistant the pro-code way: a Foundry model, grounded on your own data, wired together with a bit of code. This chapter builds the same kind of thing from the other end — low-code, in a browser, mostly by describing what you want in plain English. The tool is **Microsoft Copilot Studio**, and if you last touched this space in the first edition, almost everything about it has changed, including its name.

When I wrote the original version of this chapter, the product was Power Virtual Agents, you built "bots" by drawing conversation trees called topics, and Copilot Studio itself was in limited preview behind a support ticket. All of that is gone. Power Virtual Agents was folded into Copilot Studio, the portal moved to its own home at **copilotstudio.microsoft.com**, and — the change that matters most — the whole authoring model flipped. You no longer start by drawing dialog trees. You describe an agent in natural language, the AI drafts it for you, and at runtime the agent decides for itself which knowledge to search and which tools to call. Microsoft calls this **generative orchestration**, and it is on by default. The old topic-tree world still exists underneath, but it is now the exception you reach for, not the thing you start with.

So this is a rewrite, not a refresh. I want to walk you through building a real, useful agent end to end — an **Intune helpdesk agent** that answers device-management questions from your documentation and knowledge, published to Microsoft Teams so your L1 support and end users can actually reach it. Along the way I will be honest about the two things people get burned by: the pricing model (which changed substantially in September 2025) and the gap between "preview" and "generally available." Both matter before you put anything in front of real users.

Where Copilot Studio Fits

It helps to place Copilot Studio against the two neighbours you have already met in this book.

- **Copilot in Intune** (Chapters 1, 2, and Part IV) is Microsoft's assistant, grounded on your tenant, that you consume. You do not build it; you use it.

- **The Foundry approach** (Chapter 15) is you building an assistant with code — full control, your own orchestration, your own hosting, billed on the Azure side.
- **Copilot Studio** sits between them: you build the agent, but low-code, in a graphical tool, with Microsoft hosting and running it. You bring knowledge and tools; Microsoft brings the model, the orchestration, and the channels.

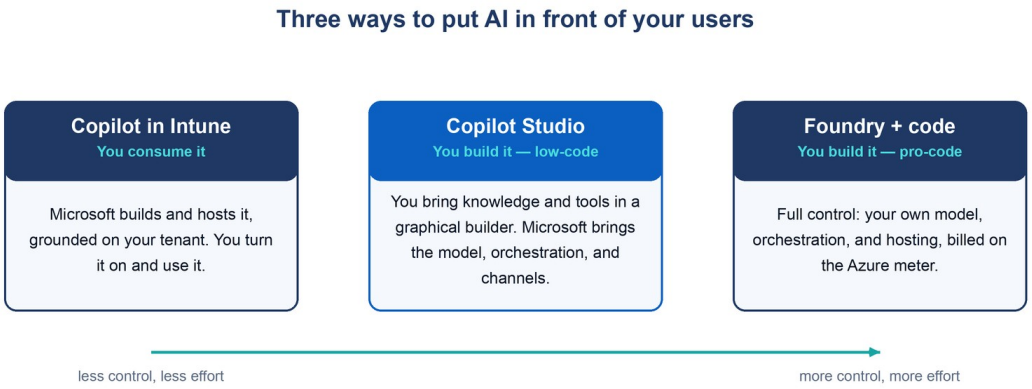

Figure 16.1: Three ways to put AI in front of your users — Copilot in Intune (you consume it), Copilot Studio (you build it, low-code), and the Foundry-plus-code path from Chapter 15. Copilot Studio trades some control for far less effort

There is also a fourth thing that is easy to confuse with Copilot Studio, so let me draw the line clearly. Inside Microsoft 365 Copilot there is a lightweight **Agent Builder** for creating *declarative agents* — you give it a name, some instructions, and a few knowledge sources, and you get a scoped Copilot that lives inside the M365 Copilot experience. That is great for "a Copilot that only knows about our HR policies," and it takes about five minutes. But it is deliberately limited: no custom tools of real depth, no standalone channels, no orchestration you can shape. **Copilot Studio is the full builder** — custom tools, autonomous triggers, multiple channels, the works. If you outgrow Agent Builder, Copilot Studio is where you go. In this chapter we use the full Copilot Studio.

One more capability worth knowing before we build: Copilot Studio can **bring your own model**, including models you have deployed in Microsoft Foundry (Azure OpenAI and others). That is

the bridge back to Chapter 15 — you can prototype low-code here and later swap in a specific Foundry-hosted model if you need a particular capability or you are managing cost on the Azure meter. Bringing your own model is billed separately, on the Azure side, not out of your Copilot Studio credits.

The Pricing Reality — Read This Before You Build

I am putting pricing up front on purpose, because it is the part that most often surprises people after they have already shipped something.

As of **September 1, 2025**, Copilot Studio no longer meters in "messages." It meters in **Copilot Credits**. The units you consume depend on how much work each interaction does, and that is a real behavioural change from the old flat message counting. The consumption tiers Microsoft publishes are:

Interaction type	Credits consumed
Classic (topic-based) answer	1
Generative answer	2
Agent action (a tool call)	5
Tenant graph grounding (Microsoft 365 data)	10
Agent flow	13 per 100 actions
AI tools	1 / 15 / 100 per 10 responses, by tool

The takeaway is that a modern agent — which uses generative answers and calls tools — costs meaningfully more per interaction than an old topic bot did, and grounding on your Microsoft 365 tenant graph is the most expensive single ingredient at 10 credits. That is not a reason to avoid it; it is a reason to know it is there before your bill tells you.

You pay for credits in one of two ways. A **prepaid pack** is roughly **$200 per pack per month for 25,000 Copilot Credits**, and — importantly — those credits are **pooled at the tenant level**, so all your agents draw from the same bucket. Or you go **pay-as-you-go** through an Azure meter, billed per credit as you consume. (Third-party sources put the PAYG rate around **$0.01 per credit**; treat that figure as unofficial and check the current Azure meter before you plan a budget around it.)

There is one behaviour that will bite you if you do not know it: if you are on prepaid packs only and you blow past **125% of your prepaid credits**, Microsoft **disables the affected agents** until you either add capacity or attach pay-as-you-go. In other words, a runaway or a viral internal launch does not silently run up an unbounded bill — it stops. Whether "stops" or "unbounded bill" is the outcome you want depends entirely on the agent, so decide deliberately: attach PAYG for the agents that must stay up, and let the credit cap protect you on the ones that are experimental.

Finally, the bundling that changes the math for a lot of organisations: **Microsoft 365 Copilot is $30 per user per month** (annual commitment). If a user is M365-Copilot-licensed and they use an agent that runs under their own identity (a classic employee-facing, "B2E" scenario), that usage is **covered — no separate credit charge**. So an Intune helpdesk agent published to Teams for a workforce that is already M365-Copilot-licensed can be effectively free to run at the point of use, with credits only coming into play for the things that fall outside that envelope. Check your own licensing before you assume either way, because this is the single biggest lever on what an internal agent actually costs you.

What You Need Before You Start

- **A Copilot Studio license** or a trial. You can start a free trial from the Copilot Studio site; you do not need to file a support ticket the way you did in the preview days.
- **The right to create agents in your tenant.** Depending on your Power Platform governance, an admin may need to enable Copilot Studio and set which environment you can build in. Build in a **dev/test environment** first — do not prototype in production.
- **Knowledge to ground on.** For our example that is Intune documentation plus an internal source (a SharePoint site or a set of uploaded docs). Grounding is what turns a generic chatbot into something that answers *your* device-management questions correctly.

- **Rights to publish to your target channel.** Publishing to Teams for the whole organisation typically needs admin approval; publishing to yourself for testing does not.

Building the Intune Helpdesk Agent

Here is the goal, stated plainly: an agent that a support technician or an end user can ask "How do I reset the Company Portal on Android?" or "What does error 0x87D1041C on a Win32 app mean?" or "What's our policy for enrolling a personal iPhone?" — and get a correct, sourced answer drawn from Intune's own documentation and our internal runbooks, without opening a ticket. We will build it, ground it, test it, and publish it to Teams.

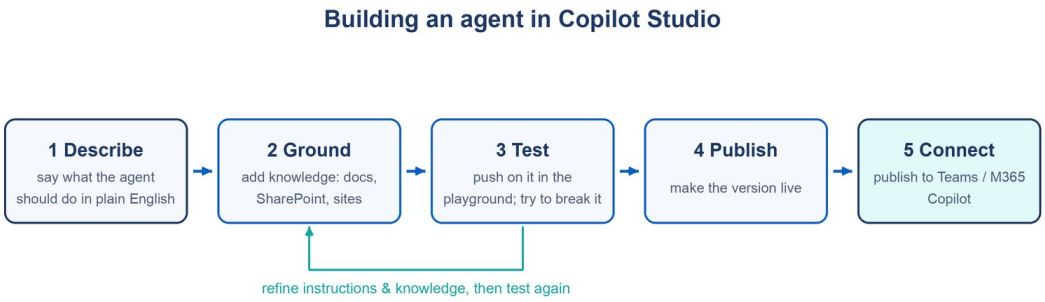

Figure 16.2: The whole build loop — describe the agent in plain English, ground it on your knowledge, test it hard in the playground, publish, and connect it to Teams. The real work is the describe-ground-test loop; everything the old product made you author by hand is now handled by generative orchestration

Step 1 — Create the agent by describing it

Go to **copilotstudio.microsoft.com**, confirm you are in the right environment (top-right environment picker — this is the single most common mistake, building in the wrong environment), and choose to create a new agent.

Instead of a form, you get a conversation. Copilot Studio asks what you want your agent to do, and you answer in natural language. Type something concrete, for example:

> “Create an agent that helps our IT support team and employees with Microsoft Intune and device management questions. It should answer using our internal IT knowledge base and official Intune documentation, explain enrollment, compliance, app deployment and common error codes, and always be professional and concise. If it doesn’t know, it should say so and suggest opening a ticket.”

The AI reads that and drafts the agent for you — it proposes a **name**, a **description**, and a first set of **instructions** (the system prompt, in effect). You can keep refining in the same chat (“make the tone friendlier,” “always include the source link”), and it updates the draft live.

When the draft looks right, create it. You land in the agent’s configuration page.

Step 2 — Refine the instructions

The **instructions** are the most important thing you will write by hand, because under generative orchestration they are how you steer behaviour — not dialog trees. Open the agent’s **Overview** / instructions and tighten what the AI drafted. Good instructions for a helpdesk agent are specific about scope, sourcing, and escalation. For example:

> “You are the Intune Helpdesk assistant for [Company]. Answer questions about Microsoft Intune, device enrollment, compliance policies, app deployment, Autopilot, and common error codes. Prefer our internal knowledge base; use official Intune documentation to fill gaps. Always cite the source. Keep answers under 150 words unless the user asks for detail. If a question is outside device management, or you are not confident, say so and tell the user to open a ticket in [system]. Never invent error-code meanings or policy details.”

Instructions like “never invent error-code meanings” and “cite the source” do real work — they are your main defence against the confident-wrong answer that erodes trust in an internal tool on day one.

Step 3 — Add knowledge

This is what makes the agent yours. Open the **Knowledge** section and add sources. Copilot Studio supports a range, and you will typically combine several:

- **Public websites** — up to 25 URLs. Point one at Intune's documentation on **learn.microsoft.com** so the agent can ground on Microsoft's own current docs.
- **Uploaded documents** — your runbooks, SOPs, and error-code cheat-sheets as files.
- **SharePoint** — up to 25 site/document URLs; ideal if your IT knowledge base already lives in SharePoint.
- **Dataverse** — effectively unlimited tables; useful if your CMDB or ticket taxonomy is there.
- **Graph and Copilot connectors** — to reach data across Microsoft 365 and third-party systems.
- **Web Search (Bing grounding)** and **tenant graph grounding** — for open-web answers and for grounding on your Microsoft 365 tenant data respectively. Remember from the pricing section that tenant graph grounding is the expensive one (10 credits), so turn it on deliberately, not reflexively.

For our agent, add the Intune docs site as a public-website source and add (or upload) your internal IT knowledge base.

There is also an **allow the agent to use its own general knowledge** (ungrounded) toggle. For a helpdesk agent I usually leave this **off** or tightly constrained, because the whole point is grounded, sourced answers. An ungrounded model happily inventing an Intune error-code meaning is exactly the failure mode you are trying to prevent.

Step 4 — Confirm generative orchestration is on

Open the agent's settings and check that **generative orchestration** is enabled. On new agents it is on by default, and for this build you want it on: it is what lets the agent read a question,

decide which knowledge source to search and which tool to call, and compose an answer — instead of you having to author a topic for every possible question.

This is the mental shift from the old product. In Power Virtual Agents you *were* the orchestrator — every path was a topic you drew. Now the model orchestrates, and topics become a targeted tool for the handful of flows you want to control precisely (a strict escalation script, a compliance-sensitive disclosure, a fixed troubleshooting checklist). We will not need a custom topic for the basic helpdesk agent, and that is the point: most of what used to be manual authoring is now handled by orchestration over your knowledge.

A note on vocabulary, because the labels changed: what used to be called **actions** are now **tools**. Tools are how the agent *does* things rather than just answering — call a Power Automate flow, hit a connector, run an API, look something up in a system. For a pure question-answering helpdesk you may not need any custom tools at all. But this is exactly where you would extend it: a tool that looks up a device's compliance state in Intune via Graph, or one that opens a ticket in your ITSM system when the agent decides to escalate. That upgrade — from "answers questions" to "takes actions" — is a natural second iteration once the Q&A agent is trusted.

Step 5 — Test in the playground

Every change you make is testable immediately in the **test pane** on the right. This is your feedback loop — do not publish before you have pushed on it here. Ask the real questions your users will ask, and specifically try to break it: an out-of-scope question ("what's the weather"), an ambiguous one, and one whose answer lives only in your internal source.

Watch two things. First, **is it citing the right source** — an internal-policy question should pull from your knowledge base, a generic how-to from the docs. Second, **does it refuse gracefully** when it should. If it answers the weather question, tighten the instructions. If it invents an error code, tighten them harder and confirm the ungrounded toggle is off. It is normal to loop between instructions, knowledge, and the test pane several times before it feels solid.

Step 6 — Publish

When the agent behaves, **publish** it. In the new model publishing and channel connection are two steps: you publish first, then connect the agent to wherever you want it consumed. Hit **Publish** and confirm.

Step 7 — Connect it to Microsoft Teams

Go to the **Channels** section. Copilot Studio can publish to a range of channels — **Microsoft Teams**, **Microsoft 365 Copilot** (now a first-class channel, so your agent can appear inside the M365 Copilot experience alongside Microsoft's own), a **custom website**, **SharePoint**, **Power Pages**, plus Facebook and the broader Azure Bot Service channels and Direct Line for anything custom. For a helpdesk agent aimed at your workforce, **Teams** is the obvious home.

Select **Microsoft Teams** and enable it. You will get an option to make the agent available to yourself for testing first, and then to submit it for broader availability — publishing to the whole organisation generally goes through admin approval, which is the right control for something everyone will see.

Open it in Teams, and you have a working helpdesk agent your colleagues can message like any other chat.

That is the whole loop: describe, ground, test, publish, connect. You built a grounded, sourced, internally-published assistant without writing code — and the parts that used to be the hard, manual work (routing every question to the right answer) are now handled by generative orchestration over the knowledge you supplied.

One Step Further — Autonomous Agents

Everything above is a **conversational** agent: a person asks, it answers. The bigger shift in Copilot Studio is that agents no longer have to wait to be asked. **Autonomous (triggered) agents** went generally available in 2025 — an agent can be started by an **event** rather than a chat message, run a sequence of steps, and act. **Agent flows** reached GA on March 31, 2025, and the platform

now includes **multi-agent** scenarios (agents that hand off to other agents) and **Computer-Using Agents** that can operate a UI. As always, check the exact preview-versus-GA status of the newest sub-features in your tenant before you build production dependencies on them — this area is moving fast, and "it demoed at Ignite" is not the same as "it is GA in your environment."

For device management the autonomous pattern is where it gets genuinely interesting. Picture an agent triggered when a device fails a compliance check: it looks up the device, checks the failure reason against your runbook knowledge, attempts a known remediation via a tool, and — only if that fails — opens a ticket and notifies the owner. That is the same "spot the pattern, then fix it before anyone files a ticket" idea that runs through this whole book, now expressible in a low-code tool. Start with the conversational helpdesk agent, earn trust, then let the autonomous versions take on the routine work behind it.

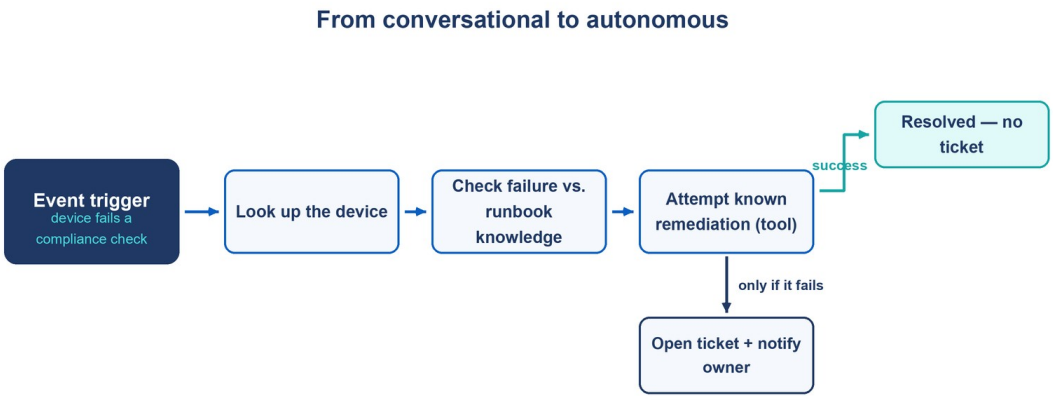

Figure 16.3: An autonomous agent for device compliance — a failed compliance check triggers it, it looks up the device, checks the failure against your runbook knowledge, and attempts a known remediation with a tool; only if that fails does it open a ticket and notify the owner

That shift — from an assistant you message to an agent that acts on its own — is the last big idea in this book, and it is big enough to deserve its own chapter. **Chapter 17** steps back from any single builder to map the whole agent landscape: Copilot Studio agents, the Security Copilot agents already living inside Intune, Foundry's Agent Service, and **Microsoft Agent 365**

tying them together — and, just as importantly, how you give an agent its own identity, bound its permissions, and govern what it is allowed to do before you let it loose on your tenant. That is where we finish.

Chapter 17: Agents, Agent 365, and Governing AI in Device Management

Chapter 16 ended on a quiet turning point that is easy to read past. We built a helpdesk agent in Copilot Studio the conversational way — a person asks, it answers — and then, in the "One Step Further" section, I pointed at the thing that actually changes the game: an agent that no longer waits to be asked. An agent triggered by a failed compliance check, that looks up the device, checks the failure against your runbooks, attempts a known fix, and only escalates what it cannot handle itself. That is a different kind of software. It is not a chatbot with a nicer front end. It is a worker.

This closing chapter is about that shift — from **assistants you talk to** to **agents that act** — and, just as importantly, about how you run them without getting hurt. Because the moment an AI can change a policy, quarantine a device, or file a hundred tickets on its own, the interesting questions stop being "can it?" and become "should it, right now, with this much access, and how would I know if it went wrong?" Those are governance questions, and in a device-management context they are the whole ballgame. An over-permissioned agent is the new over-privileged service account, and an ungoverned fleet of them is a problem you have met before under a different name: unmanaged endpoints.

So this chapter does three things. It defines what an agent actually is, so you are not lost in marketing. It maps the 2026 Microsoft agent landscape — Copilot Studio, Security Copilot, Foundry, Microsoft 365 Copilot, and the new control plane that sits over all of them, **Microsoft Agent 365** — so you know which tool is which. And it makes the case, concretely, for governing agents with the same discipline you already apply to devices: identity, least privilege, observability, and a bounded blast radius. Then we close the book.

From Assistant to Agent

Every AI experience in this book so far, right up through most of Chapter 16, has been an **assistant**: you prompt, it responds. Copilot in Intune summarizing a device, the Foundry-grounded assistant from Chapter 15, the conversational helpdesk agent — all of them are fundamentally reactive. They are extraordinarily useful, but they sit still until a human pokes them, and they hand the result back for a human to act on. The human is still the one doing things.

An **agent** is the assistant given a job and the means to finish it. Strip away the hype and an agentic system is four things working together:

- **A goal.** Not a single question to answer, but an outcome to pursue — "keep these devices compliant," "triage every reported vulnerability," "review this policy change for risk." The goal is standing; it does not expire when one prompt is answered.
- **Tools.** The ability to *do* things in the world, not just describe them — call Microsoft Graph, open a ticket, run a remediation, query a table. Chapter 16 called these tools (they used to be "actions"); Chapter 15 built one by hand as a function call to Graph. Tools are what turn talk into effect.
- **Memory.** State that persists across steps and across runs — what it has already tried, what it learned, where it is in a multi-step task. Without memory an agent is just a very forgetful assistant that restarts every turn.
- **A loop.** The engine that ties the other three together. The agent observes the situation, reasons about what to do next toward the goal, calls a tool, observes the result, and decides whether it is done or needs another step. It runs that observe–reason–act loop autonomously until the goal is met or it decides to escalate.

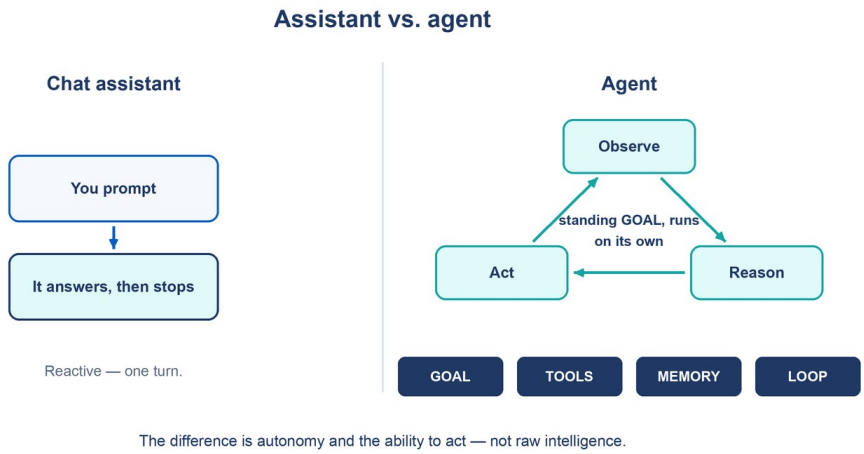

Figure 17.1: The spectrum from assistant to agent. A chat assistant answers one prompt and stops; an agent is given a standing goal and runs an observe–reason–act loop over its tools and memory until the job is done or it escalates. The difference is not intelligence — it is autonomy and the ability to act

That loop is the whole story, and it is why agents are both more valuable and more dangerous than assistants. A chat assistant that hallucinates gives you a wrong answer, and you notice, because you are reading it. An agent that reasons wrong *acts* on that reasoning — it calls the tool, changes the policy, sends the mail — possibly while nobody is watching, possibly many times before anyone checks. Everything else in this chapter follows from that single fact: the value and the risk both come from the same place, the ability to act inside a loop.

It is worth being honest about where the technology actually is in mid-2026, because the word "autonomous" carries more than the products always deliver. In practice most production device-management agents today are what I would call *bounded* autonomous: they run their loop freely over reading and reasoning, but the consequential actions — the ones that change or delete something — are gated behind either a hard rule or a human approval. That is not a limitation to apologize for. As you will see, it is exactly the design you want.

The 2026 Microsoft Agent Landscape

If you have read this far in the book, you have now met four different ways Microsoft lets you build or consume an agent, introduced in four different chapters, and it is genuinely easy to lose track of which is which. Let me lay them out on one map, because choosing the right one is most of the battle.

Figure 17.2: The Microsoft agent landscape in 2026. Four build-and-run surfaces — Microsoft 365 Copilot agents, Copilot Studio, Security Copilot agents (including the Intune agents), and Foundry Agent Service — ranging from low-code to pro-code, all governed from above by Microsoft Agent 365 as the control plane and grounded on Entra Agent ID for identity

Microsoft 365 Copilot agents are the lightest touch. These are the declarative agents you build with Agent Builder inside Microsoft 365 Copilot — a name, some instructions, a few knowledge sources — scoped to live inside the Copilot experience your users already have. We drew the line to these in Chapter 16: they are perfect for "a Copilot that only knows our onboarding docs," they take five minutes, and they are deliberately shallow. If a business user in your org spins one up, it is probably one of these. Great for reach, limited for depth.

Copilot Studio agents are the full low-code builder, and the subject of Chapter 16. You describe the agent, generative orchestration wires it up, you ground it on knowledge, add tools, and publish it to Teams or wherever. This is where most device-management teams will build their

first *real* agent — the Intune helpdesk agent — and where autonomous, triggered agents live for the low-code crowd. Microsoft hosts and runs it; you bring knowledge and tools; you pay in Copilot Credits.

Security Copilot agents, including the Intune agents, are the security-specialist path from Chapter 13. These run on Security Compute Units (SCUs), they live in the Security Copilot portal and are surfaced inside the Intune admin center, and the ones that matter most to us are the purpose-built endpoint agents: the **Change Review Agent**, the **Policy Configuration Agent**, and the **Vulnerability Remediation Agent** — with the lineup still growing. These are Microsoft-authored agents you enable and govern rather than build from scratch. As of mid-2026, watch the labels carefully: the **Change Review Agent** has reached general availability, while the **Vulnerability Remediation Agent** is in **public preview** and rolling out broadly, and the **Policy Configuration Agent** is likewise on the preview-to-GA path. (For a small lesson in how fast this moves: the Device Offboarding Agent that appeared in the lineup was retired on 1 June 2026. "It was in the catalog last quarter" is not the same as "it is there today" — recheck before you build a dependency on any single agent.)

Foundry Agent Service is the pro-code path from Chapter 15. When you need full control — your own orchestration, your own model choice, your own hosting on the Azure meter, agentic retrieval through Foundry IQ, tools you write yourself — this is where you go. It is the most powerful and the most work, and it is what you reach for when the low-code tools genuinely cannot express what you need.

The mental model I use to choose: **consume** where Microsoft has already built the right agent (the Security Copilot Intune agents), **build low-code** in Copilot Studio when you want your own agent quickly and Microsoft's hosting is fine, and **build pro-code** in Foundry when control matters more than speed. Most organizations end up with a mix — a Copilot Studio helpdesk agent, the Intune agents enabled for security work, and maybe one bespoke Foundry agent for the thing that is uniquely theirs. That is a healthy portfolio. What it is *not* is a governed one — not yet. Which brings us to the piece that ties the whole landscape together.

Microsoft Agent 365 — Managing Agents as a Fleet

Here is the problem that every one of those four surfaces creates the moment it succeeds. Once agents are easy to build, people build them. A helpdesk agent here, a declarative Copilot agent there, three Security Copilot agents enabled "to see what they do," a Foundry agent someone in the platform team stood up for a proof of concept. Within a quarter you do not have *an* agent; you have a sprawl of them — running on different meters, holding different permissions, built by different people, some of them forgotten, none of them inventoried. You have, in other words, exactly the situation that made device management necessary in the first place: a fleet of things that can act on your behalf, and no single place to see or control them.

Microsoft Agent 365 is Microsoft's answer to that problem. Announced at Ignite 2025 and generally available to commercial customers from **1 May 2026** (as a per-user add-on and as part of the new Microsoft 365 E7 suite), it is not another place to build agents. It is the **control plane** *over* the agents you build everywhere else. Microsoft describes it in terms of five capabilities, and the way to read them is as the management disciplines you already apply to endpoints, pointed at a new kind of endpoint:

- **Registry** — a single, authoritative inventory of every agent in your organization: the ones built in Copilot Studio, the ones with an Entra identity, the ones registered in the Teams store, and — the part that should sound very familiar — eventually the *shadow* agents nobody told you about. This is your device inventory, for agents.
- **Access control** — governing what each agent is allowed to reach: apps, resources, the internet, and other agents. This is Conditional Access and least privilege, for agents.
- **Visualization** — seeing what the fleet is doing, how agents relate, where the activity is. This is your reporting and dashboards, for agents.
- **Interoperability** — working across Microsoft platforms, open-source frameworks, and partner clouds, so the control plane governs agents wherever they run rather than only the ones born inside Microsoft's tools. (Multicloud registry sync entered public preview alongside GA.)

- **Security** — data protection and threat monitoring, delivered through Microsoft Purview and Defender: catching an agent leaking sensitive data, spotting risky agent behaviour in real time, and auditing what every agent actually did.

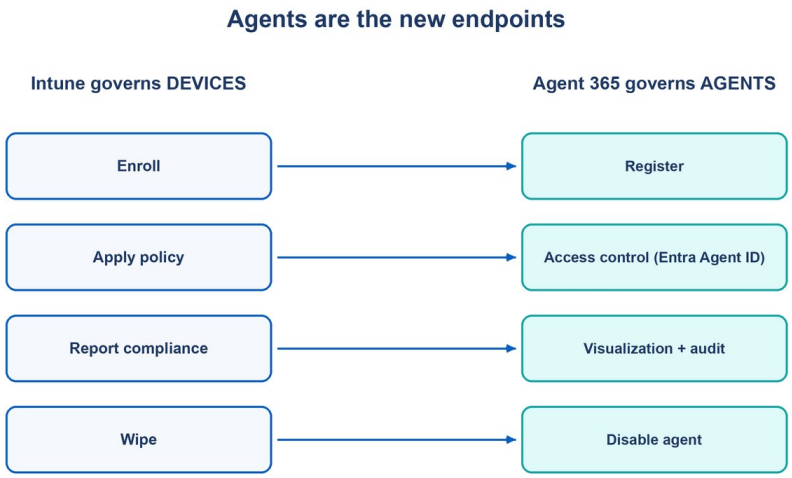

Figure 17.3: Agents are the new endpoints. Just as Intune enrolls devices, applies policy, reports on compliance, and can wipe a device that goes wrong, Microsoft Agent 365 registers agents, controls their access, visualizes the fleet, and secures it. The management disciplines are the same; only the thing being managed is new

I want to sit on that parallel for a moment because it is the single most useful idea in this chapter for a device-management reader, and it is the reason this material belongs in *this* book rather than a security one. **Agents are the new endpoints.** Think about what Intune does for a device: it enrolls it so you know it exists, it applies policy so it behaves within bounds, it reports compliance so you know its state, and — when a device is lost or an employee leaves — it can wipe it, because you need a kill switch for a thing that can do damage. Agent 365 does the isomorphic thing for agents. Registry is enrollment. Access control is policy. Visualization is compliance reporting. And the ability to disable a misbehaving agent is the wipe.

You already know how to do this job. You have been doing it for endpoints for years — the whole first half of this book was about knowing your fleet through data, and the middle was about acting on it through automation. An agent fleet needs the same instincts, and the good

news is that the muscle memory transfers. If you would not deploy an unmanaged laptop with domain admin rights and no logging, do not deploy an agent that way either. The uncomfortable news is that a lot of organizations are about to do exactly that, because building an agent is now so easy that it does not *feel* like deploying a privileged actor into the environment — but that is precisely what it is.

Identity and Least Privilege for Agents

If agents are endpoints, the first question is the same one you ask of any endpoint or any automation in this book: **who is it, and what is it allowed to do?** For the automation in Chapter 8 the answer was a managed identity with the least Graph permission it needed. For agents, the answer in 2026 is **Microsoft Entra Agent ID**.

Entra Agent ID, which reached general availability in the spring of 2026, gives each agent its own **first-class identity in Entra** — not a shared service account, not a human's delegated token, not an app registration with a fat secret, but a distinct identity with its own object ID, its own audit trail, and its own scoped permissions. This matters enormously. It means that when an agent does something, the log says *which agent* did it, the same way a sign-in log says which user signed in. It means you can grant one agent access to read device compliance and nothing else, and a different agent access to open tickets and nothing else, and hold each to its own blast radius. And it is **keyless** in the spirit of Chapter 8 and Chapter 15 — the agent proves who it is through the identity platform, so there is no static secret embedded in the agent to leak.

Entra Agent ID is also opinionated about least privilege in a way I find genuinely reassuring, because it does not trust *you* to always get it right. The platform structures agent identities so that grants can be governed consistently, and — critically — it **blocks agents from holding the most dangerous privileges outright.** An agent cannot be granted Global Administrator, Privileged Role Administrator, or User Administrator, and cannot hold tenant-wide write permissions on Graph — *not even if an administrator tries to consent to them.* The system refuses. That is a hard floor under the "over-permissioned agent" failure mode, and it is the

right instinct: the blast radius of an autonomous actor should never include the keys to the entire kingdom, no matter how convenient that would be for whoever is building it in a hurry.

The lesson from Chapter 8 applies here almost word for word. There I argued that an app registration with a client secret is a liability — a credential that leaks, expires, and has to be babysat — and that a managed identity with the least Graph permission it needs is the pattern to build from the start. Entra Agent ID is that same argument, evolved for a world where the thing holding the identity is not a Logic App but an autonomous worker. **An over-permissioned agent is the new over-privileged service account** — the same anti-pattern, higher stakes, because a service account only does what its script tells it, while an agent decides for itself what to do with the access you gave it. Grant it the least it needs. Give it its own identity so you can see what it did. Never, ever hand it a standing credential to your whole tenant because it was easier than scoping one.

MCP and Tools — How an Agent Reaches Your Data

An agent with a goal and an identity still cannot do anything until it has **tools**, and in 2026 the way agents get tools has largely standardized on one thing: the **Model Context Protocol (MCP)**.

MCP is worth understanding because it quietly solved a real problem. Before it, every agent platform had its own bespoke way of describing a tool — how you tell the model "here is a function called `get_device_compliance`, it takes a device ID, here is what it returns." That meant a tool you built for one platform did not work on another, and every integration was custom plumbing. MCP is an open standard — introduced by Anthropic in late 2024, and since donated to a vendor-neutral foundation under the Linux Foundation — for describing tools and knowledge sources so that *any* MCP-speaking agent can discover and call them. Think of it as the USB-C of agent tooling: one connector shape, and suddenly your tools plug into everything.

By mid-2026 this is not a niche preference; it is the default fabric across Microsoft's stack. Foundry speaks MCP, Copilot Studio can consume MCP tools and knowledge servers as steps in an agent, Agent 365 governs them, and the wider industry — OpenAI, Google, and thousands of

teams — has settled on the same protocol. For you, that means a tool you expose once through MCP is reachable from the Foundry agent *and* the Copilot Studio agent *and* whatever you build next year, without rewriting the integration each time.

The device-management angle writes itself. The most valuable tool you can give a device-management agent is the one that reaches your tenant — **Microsoft Graph**. Wrap the Graph calls your agents need behind an MCP tool (or use the Graph connectors the platforms already provide), and now an agent can ask "is DEVICE-042 compliant?", "which policies apply to this user?", "how many devices are missing the July update?" and get a live, grounded answer to reason over — exactly the "give the assistant hands" pattern from Chapter 15, now standardized and reusable. And note how the pieces stack: the *tool* is Graph via MCP, the *permission* the tool runs under is the agent's Entra Agent ID scoped to the least it needs, and the *record* of every call is in the agent's own audit trail. Tooling, identity, and observability are not three separate features you bolt on; they are three views of the same well-governed agent.

Governing Agents

You now have the ingredients: agents built on four surfaces, inventoried in Agent 365, identified by Entra Agent ID, and tooled through MCP. Governance is the practice of making sure they stay trustworthy once they are running. In a device-management context I care about five things, and none of them are optional for anything that touches production.

Guardrails and content safety. An agent needs bounds on what it will say and do, enforced *below* the level of its instructions — because instructions are a request, not a guarantee. Content safety filters, blocked topics, and hard rules ("never delete, only recommend deletion") are the seatbelt. In Chapter 16 I wrote instructions like "never invent error-code meanings"; that is a guardrail expressed in prose, and it does real work, but for consequential actions you want the guardrail expressed as a rule the agent physically cannot override, not a polite instruction it usually follows.

Evaluations before you trust it. You would not ship a remediation script to 10,000 devices without testing it on a pilot ring first, and an agent deserves the same skepticism — more,

because its behaviour is not deterministic. Before an agent gets to act on production, evaluate it against a set of realistic cases: the questions it will get, the decisions it will face, the edge cases you are worried about. Does it cite the right source? Does it refuse when it should? Does it pick the safe action under ambiguity? Run those evaluations again whenever the model or the instructions change, because an agent is not a build you ship once; it drifts as its underpinnings move.

Observability and audit. When an agent acts, you must be able to answer "what did it do, and why?" after the fact. This is where Entra Agent ID's per-agent identity and Agent 365's visualization and Purview's audit trail earn their keep — every action attributable to a specific agent, every decision inspectable. If you cannot audit what an agent did, you cannot run it in production, full stop. The same principle that made me insist on keyless, attributable identity for automation in Chapter 8 applies with more force here: an unattributable actor in your tenant is an incident waiting to be un-investigable.

Human-in-the-loop for consequential actions. This is the single most important control, and it is the design pattern that makes agents safe to deploy today. Let the agent run its loop freely over everything that only *reads and reasons* — investigating, correlating, drafting a recommendation. Then, for anything that *changes the world* in a way that is hard to undo, stop and ask a human. The Change Review Agent's whole job is a version of this: it evaluates a change and recommends, rather than silently rewriting your policies. Design your own agents the same way. The goal is not to keep humans busy; it is to keep the irreversible decisions on the human side of the line while the agent does the tedious 90% that leads up to them.

Cost. Agents cost money to run, continuously, and the meter depends on which surface they live on. A Copilot Studio agent burns **Copilot Credits** (recall from Chapter 16 that a tool call and tenant-graph grounding are the expensive ingredients, and that a runaway on prepaid packs gets disabled at 125% rather than billing you into oblivion). A Security Copilot agent burns **SCUs**, with overage billed at a premium — and as Chapter 13 stressed, an agent left running on a schedule is a *continuous* cost that is easy to forget. A Foundry agent bills on the **Azure meter** for its model and retrieval usage. The failure mode is the same across all three: someone enables a

handful of agents “to see what they do,” walks away, and the meter runs. Cap what you can cap, watch the usage dashboards in the first weeks, and treat an idle-but-enabled agent as a cost leak to close, not a harmless leftover.

Responsible AI in Device Management, Specifically

The five governance controls above are general. But device management raises the stakes on three of them in a way that deserves to be named directly, because the blast radius here is unusually physical.

Data protection. A device-management agent is, by its nature, pointed at some of the most sensitive data you hold: who uses which device, where those devices are, what is installed, what is out of compliance, what vulnerabilities are open. That is a map of your attack surface. An agent that grounds on it, reasons over it, and possibly summarizes it into a Teams message is a data-exposure path you have to think about deliberately — which is exactly why Agent 365 leans on Purview for data protection and why grounding on the tenant graph is both powerful and the thing to enable with intent rather than reflex.

Grounding against the confident-wrong answer. I said it in Chapter 16 and it is worth repeating as the load-bearing principle it is: an ungrounded model will invent an error-code meaning or a policy detail with total confidence, and in a helpdesk answer that erodes trust, but in an *agent’s reasoning* it does worse — it becomes the premise for an action. An agent that “knows” a device is non-compliant because it hallucinated the compliance state, and then acts on it, has done real harm off a fabricated fact. Grounding — retrieval over your real tenant data, with citations, with the ungrounded-knowledge toggle off — is not a quality nicety for an agent that acts. It is a safety control. Ground it, cite it, and make it say “I don’t know” rather than guess.

Blast radius. This is the one that should keep you honest. An assistant that gets it wrong wastes a few minutes. An agent with a Graph tool scoped to `DeviceManagementConfiguration.ReadWrite.All` that gets it wrong can push a broken compliance policy across the fleet. An agent that can wipe devices and gets it wrong is a

genuinely bad day. So *bound the blast radius on purpose*, and bound it in layers: give the agent the least permission through Entra Agent ID (read, not write, wherever reading is enough); put a hard guardrail on the destructive actions; require human approval before anything irreversible; scope the agent to a pilot ring of devices before the whole fleet, exactly as you would a remediation script. Ask the question that actually matters — "what is the worst thing this agent can do if every assumption it makes is wrong?" — and if you do not like the answer, shrink its permissions until you do. A well-designed device-management agent is not the one with the most capability. It is the one whose worst case you have deliberately made small.

A Realistic Device-Management Agent

Let me pull all of this into one concrete picture, because it is also the picture of everything this book has been building toward. Imagine an agent — call it the compliance steward — that does this:

It **watches the signals** you spent the first half of this book learning to collect: compliance state from Intune, the device health and performance signals from Endpoint Analytics, the vulnerability data from Defender. That is the **Analytics** layer, and the agent is only as good as it is because that data is good — timely, complete, and true to your environment. A confident agent grounded on stale or partial data is worse than no agent at all.

When it spots a device drifting out of compliance, it **reasons about why**, grounded on your runbooks and known-issues knowledge — the same grounding pattern from Chapters 15 and 16 — and decides which of three buckets the situation falls into. If it is a known, safe, reversible fix — a setting to reapply, a sync to trigger, a well-understood remediation — it **applies it through a tool**, using its own least-privilege identity, and logs exactly what it did. That is the **Automation** layer from Chapter 8, now driven by judgement instead of a fixed schedule. If the fix is consequential — a policy change, anything that touches many devices, anything hard to undo — it does not act; it **drafts the change and escalates to a human** for approval, the Change Review pattern. And if it does not understand the situation at all, it says so, opens a ticket, and gets out of the way.

That is the whole book, expressed as one agent. **Analytics** gives it the truth of your environment to reason over. **Automation** gives it the hands to fix what is safe to fix. **AI** gives it the judgement to tell the routine from the risky and to escalate the difference. Take away any one layer and it collapses: no data and it is guessing; no automation and it is just talk; no AI and it is a cron job that cannot tell a real problem from a blip. Together, it is the thing the first edition of this book could only gesture at, now buildable — carefully, in bounds — in your own tenant.

You will not build the whole compliance steward on day one, and you should not try. Build the read-only version first — the one that watches and recommends and escalates *everything*, taking no autonomous action at all. Run it. Evaluate its recommendations against what your engineers would have done. Earn the trust. Then, one safe action at a time, let it start doing the routine work itself, always with the blast radius bounded and the audit trail on. That is how you get to autonomy responsibly: not by trusting the agent up front, but by letting it prove itself on the reversible things while the consequential ones stay firmly in human hands.

Community, Resources, and People to Follow

No one learns this platform in a vacuum, and I certainly did not. Intune and the wider modern-management space have one of the most generous technical communities in IT, and a lot of what is in this book started as something one of these people published. A few I read constantly and recommend without reservation:

- **Scloud** — scloud.work by Florian Salzmann
- **Oceanleaf** — oceanleaf.ch by Nicklas Tinner
- **Call4Cloud** — call4cloud.nl by Rudy Ooms
- **Intune Newsletter / andrewstaylor.com** by Andrew Taylor
- **Syst & Deploy** — systanddeploy.com by Damien Van Robaeys
- **Rockenroll.tech** — rockenroll.tech by Nicklas Ahlberg

- **MSEndpointMgr** — msendpointmgr.com by Nickolaj Andersen, Maurice Daly, Jan Ketil Skanke and many contributors

There are more outstanding blogs than I could list, and I have certainly forgotten some I shouldn't have.

For community, the **Modern Endpoint Management** LinkedIn group founded by Angel Garcia Ayas is the gathering place — Microsoft engineers, MVPs, and practitioners answering each other's questions in the open. There is also an active Intune community on X and the Microsoft Enterprise Mobility + Security Discord. And if you can get to an event in person, the **MMS**, the **Workplace Ninja Summit**, **Experts Live**, and the MEM community events are where a lot of these relationships actually form.

If you want more from me directly, I write and post here:

- **Blog** — jannikreinhard.com
- **YouTube** — @ModernDevMgmt
- **LinkedIn** — jannik-r
- **X** — @jannik_reinhard
- **Bluesky** — jannikr.bsky.social

Closing: Analytics, Automation, and AI

We have reached the end of the book, so let me tie its three threads together one last time — because they were never really three separate things. They are one stack, and each layer is what makes the next one worth having.

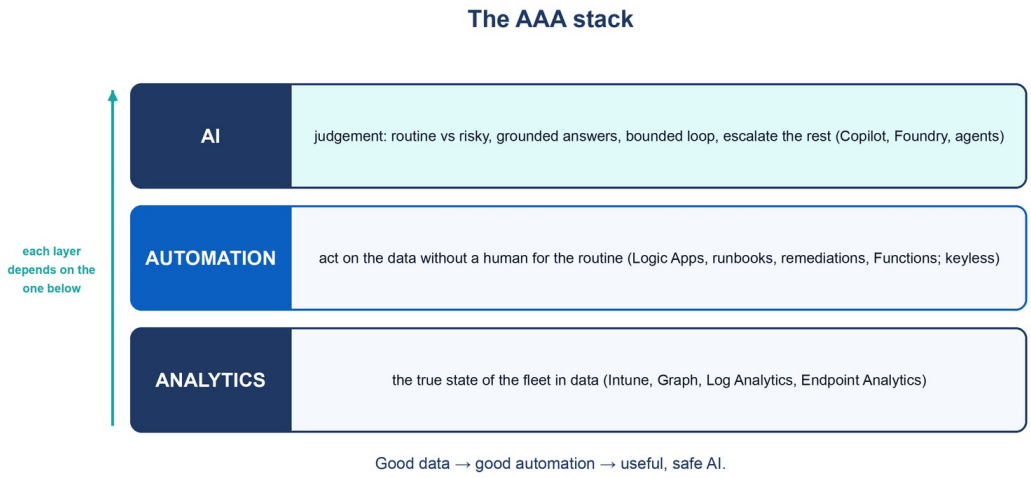

Figure 17.4: The argument of the whole book in one picture. Analytics is the foundation — the true state of your fleet, in data. Automation is the layer that acts on it without a human in the loop for the routine. AI is the layer on top that brings judgement. Each layer depends on the one below: good data makes good automation possible, and good automation plus good data is what makes AI genuinely useful and safe in device management

Analytics was where we started the real work — getting the data out of Intune, out of Graph, out of Log Analytics and the reports and the warehouse, and turning "I think the fleet is fine" into evidence you can point at. It is the least glamorous layer and the most important, because everything above it inherits its quality. Garbage data does not become insight because you put a model on top of it; it becomes confident garbage, which is worse.

Automation was the next layer — taking the actions the data made obvious and turning them into standing infrastructure instead of manual chores. Logic Apps, runbooks, remediation scripts, Functions; managed identities instead of secrets; "when this happens, do that," running itself while you sleep. Automation is what turns knowing into doing at a scale no human could sustain by hand.

AI is the layer we spent Part IV putting on top — Copilot in Intune reasoning over your tenant, a Foundry-grounded assistant answering in your own words, a Copilot Studio agent a colleague can message in Teams, the Security Copilot Intune agents, and now agents that watch, reason,

and act inside a bounded loop. AI is the judgement layer: the part that can tell the routine from the risky, the real problem from the blip, and can escalate the difference.

None of those layers is worth much alone, and that has been the argument of the whole book. AI with no data underneath it is a confident guess. Automation with no analytics telling it what to act on is a script waiting to do the wrong thing at scale. Analytics that no automation or AI ever acts on is a dashboard nobody opens. The value is in the stack: **good data makes good automation possible, and good automation plus good data is what makes AI genuinely useful — and genuinely safe — in device management.** Grounded, specific, acting on the truth of *your* environment rather than a generic model's idea of it, and bounded so that when it is wrong, it fails small.

Everything in this book is buildable in your own tenant, today, with the tools as they actually exist in 2026 rather than as they were promised. So build something. Not the whole compliance steward — start smaller than that. Build **one grounded agent** that answers a real question your team keeps asking. Build **one automation** that does a chore you are tired of doing by hand. Build **one report that changes a decision** — the truest test of whether an analytics project was worth doing. Any one of those is a complete example of the whole idea, and each one teaches you more than a chapter can. The tools have finally caught up to the promise. What happens next is up to what you build with them — and I genuinely cannot wait to see it.

Thank you for reading. Writing both editions of this book has been one of the great privileges of my career, and the best part has always been hearing from the people who took something in it and built. So build something, and then tell me about it — you know where to find me.

— **Jannik**