The ArcGIS Compendium — Volume G: Automation and Administration

ON THIS PAGE

The ArcGIS Compendium — Volume G: Automation and Administration

Scaling up — Python in both its ArcGIS flavors, REST services, web development, running an organization, and ArcGIS Enterprise.

One of eight volumes. Approximately 21149 words. Chapters cross-reference the whole set by number.

In this volume:

Python for ArcGIS: arcpy and the ArcGIS API for Python

There are two completely different Python libraries in the ArcGIS world, and almost every early frustration with scripting comes from mixing them up. The first is **arcpy**: it lives inside ArcGIS Pro on your desktop, and it automates the things Pro does — running geoprocessing tools (the analysis and data-management commands that are Chapter 22's whole subject), reading and writing rows in your data, managing layers and maps in a project. The second is the **ArcGIS API for Python** (imported as `arcgis`): it talks to ArcGIS Online or ArcGIS Enterprise over the web, and it automates the things a **portal** — the web hub where your items, maps, users, and groups live — does: searching for items, publishing layers, managing users and groups, updating web maps.

A useful mental shortcut: arcpy is a robot sitting at your desk clicking through Pro for you; the ArcGIS API for Python is a robot logged into your Online account clicking through the website for you. They can be used together — a script can crunch data with arcpy and then publish the result with `arcgis` — but they are separate tools with separate strengths, and this chapter treats them separately.

You do not need to be a programmer to get real value here. If you can read a recipe, you can read the scripts in this chapter. And the single best on-ramp — copying Python commands out of tools you have already run by hand — requires no programming at all to start.

Where the code actually runs

Before writing anything, it helps to know the three places ArcGIS Python code lives, because each place has different capabilities and different rules.

Inside ArcGIS Pro. Pro ships with its own complete copy of Python, bundled and managed by an environment system called **conda** (a tool that keeps a Python installation and its add-on packages together in a self-contained folder, so different projects can use different package versions without conflict). When you open the Python window in Pro or run a script tool, you are using that bundled Python, and arcpy is already installed and licensed. This is the only place arcpy works out of the box

— arcpy is not something you can freely install with `pip` (Python's standard package installer) on a random machine; it comes with Pro (and with ArcGIS Server) and checks your license when it loads.

On your machine, outside Pro. You can run scripts from a plain command prompt or an editor like VS Code, as long as you point them at Pro's bundled Python environment. This is how scheduled overnight jobs usually run: Windows Task Scheduler launches Pro's Python, which runs your script, no Pro window ever appearing. The ArcGIS API for Python can also be installed into any ordinary Python environment on any machine — Windows, Mac, Linux, a server in the cloud — because it only needs an internet connection to your portal, not a Pro license.

In hosted notebooks. ArcGIS Online and ArcGIS Enterprise can run **notebooks** — interactive documents that mix text, code, and results — on Esri's servers (or your Enterprise servers) rather than your machine. The `arcgis` library is always available there; arcpy is available only in certain notebook tiers. More on this later in the chapter.

> **Watch out:** Do not modify Pro's default Python environment directly. If you need extra packages, clone the environment first — in recent Pro versions this lives in the package management area of the application settings (`Project > Package Manager`) — then install into the clone and set it active. Installing into the default environment can break Pro itself, and repairing it sometimes means reinstalling.

Environments in one paragraph

If the word "environment" is new: think of it as a toolbox. Pro comes with a sealed factory toolbox containing Python plus arcpy plus everything Pro needs. You cannot add or remove tools from the sealed box, but you can duplicate it ("clone" it), add your own tools to the copy, and tell Pro to use the copy. The `arcgis` library is already in the factory box; third-party packages like `openpyxl` for Excel work or `requests` for web calls may need a clone. That is genuinely all most people ever need to know about conda.

The arcpy world: automating ArcGIS Pro

arcpy exposes essentially everything you can do in Pro's geoprocessing framework, plus direct row-level access to your data. The workflow it replaces is you, opening tools one by one, filling in parameters, clicking Run, waiting, and repeating for the next dataset. If you have ever thought "I need to do this exact thing to forty shapefiles," you have found arcpy's reason to exist.

The learning path that actually works: copy the Python command

Every geoprocessing tool you run in Pro records itself in the project's history, and every history entry can be turned into working Python with one right-click. This is the honest secret of how most GIS people learned to script: they never wrote arcpy from scratch — they ran tools by hand, copied the generated commands, and edited them.

The loop looks like this:

1. Run a tool normally in Pro — say, Buffer on a layer of schools.
2. Open the history pane (`Analysis > History` on the ribbon in recent versions).
3. Right-click the completed run and choose `Copy Python Command` .
4. Paste it into the Python window (`Analysis > Python > Python Window`) and look at it.

What you get is one line of real code with every parameter you chose spelled out:

```
arcpy.analysis.Buffer(
    in_features="Schools",
    out_feature_class=r"C:\Projects\Safety\Safety.gdb\Schools_Buffer",
    buffer_distance_or_field="800 Meters",
    dissolve_option="ALL"
)
```

Now change `"800 Meters"` to `"1200 Meters"` , change the output name, press Enter, and you have run your first scripted analysis. Everything else in arcpy is elaboration on this move. You never have to memorize parameter names, because the tool dialog will always generate them for you.

Tip: The tool names in copied commands follow a `module.ToolName` pattern — `arcpy.analysis.Buffer`, `arcpy.management.SelectLayerByAttribute` — where the module matches the toolbox the tool lives in. When Esri's documentation shows a tool page, the Python syntax and code samples are at the bottom of that same page. The tool reference is the arcpy reference.

Setting the stage: the environment settings

Most scripts begin by setting a **workspace** — the default folder or geodatabase where tools look for inputs and write outputs:

```
import arcpy
arcpy.env.workspace = r"C:\Projects\Safety\Safety.gdb"
arcpy.env.overwriteOutput = True
```

Two things to notice. The `r` before the path string makes it a "raw" string, which stops Python from misreading backslashes (in Python, `\n` normally means "new line" — the `r` disables that). And `overwriteOutput = True` lets tools replace existing outputs instead of failing, which is what you almost always want while iterating. These `arcpy.env` settings mirror the geoprocessing environment settings covered in Chapter 22 (Geoprocessing in Depth) — anything you can set in the tool dialog's Environments tab, you can set in code.

Describing data: asking before acting

Robust scripts check what they are dealing with before they act. `arcpy.Describe()` (and its newer sibling `arcpy.da.Describe()`, which returns a plain dictionary) tells you a dataset's geometry type, coordinate system, fields, and more:

```
desc = arcpy.Describe(r"C:\Projects\Safety\Safety.gdb\Schools")
print(desc.shapeType)                    # e.g. "Point"
print(desc.spatialReference.name)        # e.g. "NAD 1983 StatePlane
..."
```

This matters in batch work: before buffering forty datasets, you can skip the ones that are not points, or flag the ones in a different coordinate system (why that matters is Chapter 3's territory). Companion functions like `arcpy.ListFeatureClasses()`, `arcpy.ListRasters()`, and `arcpy.ListFields()` enumerate what is in the current workspace, which is how batch scripts discover their inputs.

Cursors: reading and writing rows directly

Geoprocessing tools operate on whole datasets. Sometimes you need finer control — read every row, inspect a value, change it, move on. That is what **cursors** do. A cursor is an object that walks through a table one row at a time, like a finger moving down a spreadsheet. The modern ones live in the data access module, `arcpy.da`, and come in three kinds:

CURSOR	WHAT IT DOES	TYPICAL USE
`SearchCursor`	Read rows, no changes	Summaries, exports, checks
`UpdateCursor`	Read and modify or delete existing rows	Fixing values, recalculating fields
`InsertCursor`	Add brand-new rows	Loading data from another source

Reading is the gentlest introduction. This prints every school's name and enrollment:

```
with arcpy.da.SearchCursor("Schools", ["NAME", "ENROLLMENT"]) as cursor:
    for name, enrollment in cursor:
        print(name, enrollment)
```

The `with` statement guarantees the cursor is released even if something goes wrong partway — cursors place locks on data, and a script that crashes while holding a cursor can leave a dataset locked until you close Pro. Always use `with`.

Updating follows the same shape, plus a call to save each changed row:

```
        with arcpy.da.UpdateCursor("Schools", ["ENROLLMENT", "SIZE_CLASS"]) as
cursor:
            for row in cursor:
                row[1] = "LARGE" if row[0] > 1000 else "STANDARD"
                cursor.updateRow(row)
```

Two special field names are worth knowing early: `"SHAPE@"` gives you the full geometry object of each row (so you can measure it, move it, or rebuild it), and lighter variants like `"SHAPE@AREA"`, `"SHAPE@LENGTH"`, and `"SHAPE@XY"` give you just the numbers, which is faster. Cursors also accept a `where_clause` — a SQL filter, the same language covered for definition queries elsewhere in this book — so you touch only the rows you mean to.

> **Watch out:** An `UpdateCursor` changes data immediately and there is no undo button in a script. Practice on a copy of your data — `arcpy.management.Copy` or a simple export — until a script behaves, and only then point it at the real thing. For enterprise geodatabases, run update cursors inside an edit session so changes can roll back on failure; for hosted feature layers, prefer the ArcGIS API for Python instead, as described below.

Batch patterns: the whole point

Here is the canonical arcpy pattern — discover inputs, loop, run a tool on each — as a small, honest, complete script. It buffers every point feature class in a geodatabase:

```
        import arcpy
        import os

        arcpy.env.workspace = r"C:\Projects\Safety\Safety.gdb"
        arcpy.env.overwriteOutput = True

        out_gdb = r"C:\Projects\Safety\Buffers.gdb"
        if not arcpy.Exists(out_gdb):
            arcpy.management.CreateFileGDB(os.path.dirname(out_gdb),
```

```
                                        os.path.basename(out_gdb))

    for fc in arcpy.ListFeatureClasses():
        desc = arcpy.Describe(fc)
        if desc.shapeType != "Point":
            print(f"Skipping {fc} ({desc.shapeType})")
            continue
        out_fc = os.path.join(out_gdb, fc + "_800m")
        arcpy.analysis.Buffer(fc, out_fc, "800 Meters", dissolve_option="ALL")
        print(f"Buffered {fc} -> {out_fc}")

    print("Done.")
```

Every long-running arcpy script you ever write will be a variation of this: set the stage, discover, filter, act, report. Wrap the action in `try` / `except` blocks once you trust the basic flow, so one bad dataset does not kill a forty-dataset run.

Where does a script like this live? Three respectable homes: pasted into the Python window for one-off runs; saved as a `.py` file and run on a schedule; or wrapped as a **script tool** — a saved script with a proper tool dialog, so colleagues can run it from a toolbox without seeing code. Script tools are the natural next step after ModelBuilder, and Chapter 25 (ModelBuilder, Complete) discusses when a model should graduate into a script.

One boundary note: arcpy can also manipulate the Pro project itself — maps, layers, layouts, symbology — through its mapping module (`arcpy.mp`). That is a rich topic that mostly matters for automated map production, and it belongs with the layout material in Chapter 24 (Layouts, Map Series, and Print Cartography). Know it exists; reach for it when you need to export two hundred PDFs.

The ArcGIS API for Python: automating your portal

Now the other world. The ArcGIS API for Python does nothing to your local files. Its subject matter is your portal — ArcGIS Online or your organization's ArcGIS Enterprise — and everything in it: items, layers, maps, apps, users, groups, folders, sharing settings. Under the hood it is a friendly wrapper around the REST services described in

Chapter 32 (REST Services and Integration); every call it makes is a web request your portal answers.

The GIS object: your logged-in session

Everything starts with one object:

```
        from arcgis.gis import GIS

        gis = GIS("https://www.arcgis.com", "your_username")   # prompts for
password
        print(gis.users.me.username)
```

The `GIS` object represents a signed-in connection. From it hang the managers you will use constantly: `gis.content` for items, `gis.users` for accounts, `gis.groups` for groups. In a hosted ArcGIS Notebook, `GIS("home")` connects as you automatically with no password at all, which is one of the quiet pleasures of that environment. Inside ArcGIS Pro's Python window, the API can likewise piggyback on Pro's existing sign-in.

> **Watch out:** Never paste a real password into a script file. Scripts get emailed, committed to shared drives, and copied into documentation, and the password travels with them. Let the API prompt interactively, use the Pro or notebook sign-in it can inherit, or use a stored connection profile or token mechanism your organization approves. Treat any script containing a literal password as already leaked.

Searching and managing items

Everything in your portal — every layer, map, app, file — is an **item** with an ID, an owner, tags, and sharing settings (the item model is unpacked in Chapter 34, Administration). Searching looks like this:

```
        results = gis.content.search(
            query='title:"Street Trees" AND owner:your_username',
            item_type="Feature Layer",
```

```
        max_items=20
    )
    for item in results:
        print(item.title, item.id, item.type)
```

Once you have an item — from a search, or directly via `gis.content.get("the-item-id")` — you can read and change almost anything about it:

```
    item = gis.content.get("abc123...")          # the long ID from the item's URL
    item.update(item_properties={
        "snippet": "Street tree inventory, updated monthly.",
        "tags": "trees, urban forestry, inventory"
    })
    item.share(groups=[some_group])              # or org=True, everyone=True
```

This is where the API starts paying rent. Updating the summary on one item is faster by hand. Updating the summary, tags, and sharing on ninety items — after a department reorganization, say — is a ten-line loop. Housekeeping chores that never get done manually (finding items with no description, no thumbnail, or no views in a year) become one afternoon's script.

Publishing and updating layers

The API can create hosted feature layers (the cloud-hosted data layers explained in Chapter 10) from local files. The pattern is two steps: add the file as an item, then publish it as a service:

```
    csv_item = gis.content.add(
        {"title": "Street Trees 2026",
         "type": "CSV",
         "tags": "trees, inventory"},
        data=r"C:\Projects\Trees\trees_2026.csv"
    )
```

```
trees_layer_item = csv_item.publish()
print(trees_layer_item.url)
```

For a CSV, publishing analyzes the file to detect the coordinate or address columns; if it guesses wrong, `publish()` accepts publish parameters that name the latitude and longitude fields explicitly.

For ongoing maintenance — the weekly refresh of a layer that already feeds maps and apps — you generally do not republish. Republishing creates a new item with a new ID and orphans everything pointing at the old one. Instead you update the data in place. The two common approaches:

- **Overwrite:** if the layer was originally published from a file, upload a new version of that same file and use the layer manager's overwrite capability. Same item ID, same URL, fresh data. Schema changes are risky here — keep field names stable.
- **Truncate and append, or targeted edits:** get the layer object itself (`trees_layer_item.layers[0]`), then use its editing methods to delete and reload rows, or `edit_features()` to add, update, and delete specific records. This is the surgical option and works regardless of how the layer was born.

```
layer = trees_layer_item.layers[0]
layer.edit_features(updates=[{
    "attributes": {"OBJECTID": 42, "CONDITION": "Removed"}
}])
```

The API also queries layers into ordinary Python data structures — and, if you install the optional data-science components, into DataFrames from **pandas** (Python's standard table-handling library) via a spatially enabled DataFrame, which is the bridge between GIS and the wider Python analysis ecosystem. That bridge is a book of its own; here it is enough to know the door exists.

Tip: Chapter 36's worked project publishes and refreshes a real dataset end to end, including the overwrite-versus-append decision on a live layer with

dependent maps. If publishing patterns feel abstract, read this section, then that chapter.

Administering users and groups

The third pillar is administration, and for anyone who manages an organization it is the killer feature. Rosters, audits, and offboarding are all loops:

```
# Every user, with their last login (a raw timestamp)
for user in gis.users.search(max_users=1000):
    print(user.username, user.role, user.lastLogin)

# Create a group and invite members
grp = gis.groups.create(title="Field Crews 2026",
                        tags="field, operations",
                        access="org")
grp.add_users(["mgarcia_org", "jchen_org"])

# Reassign a departing user's content, then handle the account
# (item by item: item.reassign_to("new_owner"))
```

Everything Chapter 34 (Administration: Users, Groups, Security, and Credits) describes doing through the organization pages — inviting users, changing roles, auditing licenses, tracking credit-hungry items — can be scripted. The most common first admin script is a simple weekly report: every item created in the last seven days, who made it, and how it is shared. Ten lines, and suddenly the organization is legible.

A starter script for the API world

Small, honest, and complete — an inventory of your own content, flagging items missing a description:

```
from arcgis.gis import GIS

gis = GIS("home")   # in a hosted notebook; use the URL+username form
```

```
elsewhere


        me = gis.users.me
        print(f"Signed in as {me.username}")

        items = gis.content.search(query=f"owner:{me.username}",
                                   max_items=500)
        missing = [i for i in items if not i.description]

        print(f"{len(items)} items total, {len(missing)} missing a description:")
        for item in missing:
            print(f"  {item.type:25} {item.title}")
```

Run it, fix the worst offenders, run it again. Scripts that measure something and print a list are the gateway drug of portal automation: no risk, immediate value, and each one teaches you a corner of the object model.

ArcGIS Notebooks: Python hosted in the portal

A **notebook** is an interactive document made of cells — some cells hold text, some hold code, and running a code cell shows its result right below it. The format (Jupyter, an open standard) is beloved for exploration and teaching because the narrative and the code live together. ArcGIS supports notebooks in two places: inside ArcGIS Pro (`Analysis > Python > Python Notebook` opens one in a project), and hosted in ArcGIS Online or Enterprise, where notebooks are items like any other, created from the portal's Notebook section.

Hosted notebooks matter for three reasons:

- **Nothing to install.** The environment is ready the moment the notebook opens, with `arcgis` preloaded and `GIS("home")` already authenticated as you. For teaching, for quick admin chores, and for colleagues without Pro, this removes every setup obstacle at once.
- **They run in the portal, near the data.** A hosted notebook talking to hosted layers is not shipping data down to your laptop and back.

- **They can be scheduled.** A hosted notebook can run on a timer — nightly data refresh, weekly admin report — with no machine of yours switched on. This is the portal-native answer to Windows Task Scheduler.

The honest caveats: arcpy is only available in the more advanced hosted runtimes, not the standard one, and the advanced runtimes bill credits for the time they run — scheduled runs included — so learn how your organization's notebook credits work (Chapter 34) before leaving anything on a timer. The standard runtime with `arcgis` covers all the portal automation in this chapter; reach for the advanced tiers only when a hosted script genuinely needs geoprocessing.

> **Tip:** A sensible division of labor: exploratory and portal-administration work in hosted notebooks; heavy geoprocessing on a desktop or server with Pro's Python and arcpy; and the occasional hybrid where a desktop script does the analysis and the `arcgis` library uploads the result. Decide where the work naturally lives, then put the code there.

Choosing your library: a field guide

YOU WANT TO...	USE	WHERE IT RUNS
Run a geoprocessing tool on many datasets	arcpy	Pro / Pro's Python
Read or fix attribute values row by row in a geodatabase	arcpy cursors	Pro / Pro's Python
Export two hundred layout PDFs	arcpy (`arcpy.mp`)	Pro's Python
Publish a hosted feature layer from a file	ArcGIS API for Python	Anywhere with internet
Refresh a hosted layer's data nightly	ArcGIS API for Python	Hosted notebook or scheduled script

YOU WANT TO...	USE	WHERE IT RUNS
Fix tags and sharing on ninety items	ArcGIS API for Python	Anywhere with internet
Audit users, licenses, or credit burn	ArcGIS API for Python	Hosted notebook
Analysis + publish in one pipeline	Both, in one script	Machine with Pro installed

And one clarification that saves confusion: **Arcade is not Python.** Arcade (Chapter 8) is the small expression language that runs inside pop-ups, labels, and symbology in the moment a map draws. Python runs outside the map, before or after, doing the heavy lifting. They look vaguely similar and share no code.

Habits that keep scripting pleasant

A few practices, learned the hard way by everyone eventually, that you may as well adopt on day one.

Print as you go. A script that reports each step (`print(f"Processing {fc}...")`) turns a mysterious ten-minute silence into a progress bar. When something fails at dataset thirty-one of forty, you will know exactly where.

Test on copies, then scale. Run new logic against one small copied dataset. Then three. Then all forty. The cost of this discipline is minutes; the cost of skipping it is a weekend.

Keep scripts in version control (a change-tracking system like Git), or at least dated folders. Six months from now, "which version of the buffer script did we use for the report?" will be a question with consequences.

Fail loudly, not silently. When you add `try` / `except`, always print or log what failed and why. A batch script that swallows errors and reports "Done." is worse than one that crashes, because it manufactures false confidence.

Start from generated code. Copy Python commands from history; copy snippets from the API's documentation and from sample notebooks in Esri's galleries. Professional GIS scripting is mostly assembling and adapting known-good pieces. Writing from a blank file is a stunt, not a virtue.

The two libraries reward slightly different temperaments — arcpy repays care and dry runs, because it touches data directly; the ArcGIS API for Python repays curiosity, because reading and listing things is free and safe. But both share the same payoff: the third time you do any task by hand, you already know it should have been a loop. Now it can be.

REST Services and Integration

When you published your first hosted feature layer, you probably thought of it as "my data, living on a map." Here is the reframe this whole chapter rests on: you also published a web API. Every hosted layer in ArcGIS Online or ArcGIS Enterprise sits on top of a set of URLs that any program can call — a browser, a Python script, a business-intelligence dashboard, a phone app, a spreadsheet. The map viewer you use every day is just one client among many, politely asking those URLs for data and drawing what comes back.

This matters because the moment you understand the URLs underneath your layers, you stop being limited to Esri's own tools. You can pull your data into Power BI, feed it to a scheduled script, wire it into an automation platform, or hand a colleague a link that returns clean GeoJSON. This chapter walks through the anatomy of those URLs, the query language they speak, how security works, and how to decide when a live service beats a downloaded file.

An API, if the term is new to you, is an *application programming interface* — a structured way for one piece of software to ask another for data or actions. REST (short for *Representational State Transfer*) is a popular style of API built entirely on ordinary web addresses and ordinary web requests, the same kind your browser makes when

you visit any page. That is the entire trick: ArcGIS exposes your data as web addresses, and anything that can fetch a web address can use your data.

Every Hosted Layer Is an API

Start with something you can see. Open the item page for any hosted feature layer you own (see Chapter 10 for how hosted feature layers work and how they get published). Scroll toward the bottom of the item page and you will find a **URL** section with a web address ending in something like `/FeatureServer`. That address is the *service endpoint* — the front door of your layer's API.

Paste it into a browser tab. Instead of a map, you get a plain, slightly old-fashioned web page listing the service's properties: its layers, its capabilities, its spatial reference, and links to drill deeper. This is the **Services Directory**, a built-in, human-readable view of the API. Everything a program can ask the service, you can explore here by clicking. It is the single best learning tool for this chapter, because nothing is hidden: the same URLs you click are the ones your scripts will call.

> **Tip:** Keep a Services Directory tab open while you read this chapter. Every concept below — layers, queries, formats — has a clickable counterpart there. If you get lost in a URL, delete pieces from the right-hand end until you land back on a page that makes sense, then work forward again.

Three kinds of service endpoints show up most often:

ENDPOINT TYPE	WHAT IT SERVES	TYPICAL USE
`FeatureServer`	Individual features — geometries plus attributes — that clients can query, filter, and (if allowed) edit	Interactive layers, data access, editing apps
`MapServer`	Pre-drawn map images or tiles rendered on the server	Basemaps, cached reference layers, older Enterprise services

ENDPOINT TYPE	WHAT IT SERVES	TYPICAL USE
`ImageServer`	Imagery and raster data, often with on-the-fly processing	Satellite/aerial imagery, elevation (see Chapter 15)

Hosted feature layers in ArcGIS Online are `FeatureServer` endpoints, and they are the star of this chapter. Map and image services matter more in ArcGIS Enterprise environments (Chapter 35 covers when Enterprise enters the picture).

Anatomy of a Feature Service URL

A hosted feature service URL reads like a file path, and each segment means something. A typical one looks like this:

```
https://services.arcgis.com/AbCd1234/arcgis/rest/services/Hydrants/FeatureServer/0
```

Reading left to right:

- **The host** (`services.arcgis.com/AbCd1234`) identifies where the service lives. For ArcGIS Online, the scrambled-looking segment is your organization's unique identifier. Enterprise servers use your own domain instead.
- **`rest/services`** marks the root of the Services Directory. Everything after it is the catalog of services.
- **The service name** (`Hydrants`) is the name chosen at publishing time. It may differ from the item's display title, which you can rename freely afterward; the URL name is stickier.
- **`FeatureServer`** declares the service type.
- **The layer index** (`0`) picks one layer within the service. A feature service can contain several layers and tables, numbered from zero. A service published from a single layer usually has just `/0` , but multi-layer services are common — a utility service might have hydrants at `/0` , mains at `/1` , and an inspections table at `/2` .

This distinction between the *service* endpoint (ends at `FeatureServer`) and a *layer* endpoint (ends in a number) trips people up constantly. The service endpoint describes

the whole collection; the layer endpoint is where the data lives. Most tools that ask for a layer URL want the numbered version.

> **Watch out:** If a tool complains that it "cannot find a layer" at a URL that looks right, check the ending. A URL that stops at `/FeatureServer` points at the service, not a layer. Add `/0` (or the correct index — the Services Directory page lists them) and try again.

Asking for machine-readable answers: the `f` parameter

Every page in the Services Directory accepts a parameter called `f`, for *format*. Append `?f=json` to any endpoint URL and the friendly HTML page is replaced by raw JSON — *JavaScript Object Notation*, the near-universal text format programs use to exchange structured data. `?f=html` gets you back to the readable page. Later you will meet `f=geojson` for a widely supported open flavor of JSON, and `f=pjson` ("pretty JSON") which is the same data with line breaks added for human eyes.

The layer endpoint's JSON is worth skimming once. It describes the layer's fields, geometry type, spatial reference, editing capabilities, and its **maximum record count** — a server-imposed cap on how many features a single request may return. That cap (often in the low thousands, though the layer owner or an administrator can change it) drives the pagination discussion below.

The Query Endpoint: Asking Questions with a URL

Underneath each layer endpoint sits the operation you will use more than all others combined: `query`. Add `/query` to a layer URL and you can ask the layer questions — filter by attributes, filter by location, choose which fields come back, sort, count, and page through results. On the Services Directory page for any layer, a **Query** link at the bottom opens a form with every parameter laid out; fill it in, submit, and the browser shows you both the answer and the exact URL that produced it. That form is how most people learn the query language, and it never stops being useful.

A minimal query looks like this:

```
.../FeatureServer/0/query?where=1=1&outFields=*&f=json
```

Let's decode the parameters that matter, in plain terms.

`where` — the attribute filter

The `where` parameter takes a filter written in SQL syntax — SQL being the standard language of database queries, but you only need a sliver of it here. `STATUS = 'Active'` returns features whose STATUS field equals Active. `POP2020 > 50000` compares numbers. Combine conditions with `AND` and `OR`. Text values go in single quotes; numbers don't. The odd-looking `where=1=1` is a deliberate always-true condition — a conventional way of saying "give me everything."

If you have written filter expressions in Map Viewer, you have already been generating `where` clauses without seeing them; the viewer builds these strings for you. Arcade (Chapter 8) is a different language for a different job — it runs on individual features for pop-ups and symbology, while `where` runs on the server to decide which features come back at all.

`outFields` — which columns you want

By default — if you don't ask for fields at all — a query may return little more than each feature's ID. `outFields=*` requests every field; `outFields=NAME,POP2020` requests just those two. Asking only for the fields you need makes responses smaller and faster — a courtesy that becomes a necessity on large layers.

`returnGeometry` — do you want the shapes?

Set `returnGeometry=false` when you only need the attribute table — feeding a chart, counting things, exporting to a spreadsheet. Geometry is usually the bulkiest part of a response, and skipping it can shrink a reply dramatically.

Geometry filters — filtering by location

Alongside attribute filters, the query endpoint accepts a spatial filter: pass a `geometry` (commonly a rectangle, called an *envelope*, given as min-x, min-y, max-x, max-y), a matching `geometryType`, and a `spatialRel` — the spatial relationship to test, such as "intersects" (the default: return features that touch or overlap the shape) or "contains."

A related parameter pair, `distance` and `units`, turns a point into a "within this far of here" search. This is the raw mechanism behind every "features near me" experience you have ever used. The companion parameters `inSR` and `outSR` declare which coordinate system your filter shape uses and which one you want results in — coordinate systems being the subject of Chapter 3, and a classic source of "my filter returns nothing" bugs when the numbers you send are in the wrong system.

Sorting, counting, and statistics

`orderByFields=POP2020 DESC` sorts results (DESC for descending, ASC for ascending). `returnCountOnly=true` skips the data entirely and returns just how many features match — cheap, fast, and perfect for dashboards or sanity checks. The `outStatistics` parameter goes further, asking the server to compute sums, averages, minimums, maximums, or counts, optionally grouped by a field — a pivot table over the web. The Query form exposes all of these, and experimenting there beats memorizing syntax.

Pagination — getting past the record cap

Here is the rule that surprises everyone once: **a single query never returns more than the layer's maximum record count**, no matter what your `where` clause matches. If your layer holds tens of thousands of features and you ask for all of them, you get the first chunk and a flag in the response called `exceededTransferLimit` set to true — the server's way of saying "there's more."

The remedy is *pagination*: request the results in pages. `resultRecordCount` sets the page size and `resultOffset` says how many records to skip, so page two of a 1,000-record page size is `resultOffset=1000&resultRecordCount=1000`, and so on until `exceededTransferLimit` disappears from the response. Every serious client library — the ArcGIS API for Python (Chapter 31), the JavaScript SDK (Chapter 33) — handles this loop for you, but when you wire up a generic tool yourself, the loop is yours to write. (One wrinkle: some older Enterprise services don't support paging at all; the layer's JSON advertises whether yours does.)

> **Watch out:** Tools that don't understand pagination will silently truncate your data. If a spreadsheet or BI dashboard connected to a big layer shows a suspiciously round number of rows, you are almost certainly seeing exactly one

page — the record cap, not the dataset. Always compare a `returnCountOnly=true` result against what your tool actually loaded.

Here is a compact reference for the parameters covered above:

PARAMETER	PLAIN-ENGLISH MEANING
`where`	Attribute filter in SQL syntax; `1=1` means "everything"
`outFields`	Comma-separated fields to return; `*` for all
`returnGeometry`	`false` to skip shapes and return attributes only
`geometry`, `geometryType`, `spatialRel`	Spatial filter: a shape and how features must relate to it
`distance`, `units`	"Within this far of the geometry" searches
`inSR`, `outSR`	Coordinate system of your input shape / desired output
`orderByFields`	Sort order
`returnCountOnly`	Return only the number of matching features
`outStatistics`	Server-side sums, averages, and other summaries
`resultOffset`, `resultRecordCount`	Pagination: skip N records, return up to M
`f`	Response format: `json`, `pjson`, `geojson`, `html`

Tokens and Authentication, Conceptually

Everything so far assumed you could just call the URL. Whether you actually can depends on the layer's sharing level, and this is where the API meets security.

A layer shared with **Everyone (public)** answers any request, no identification needed. Public URLs can go straight into BI tools, open-data catalogs, and other people's apps. A layer shared with your **organization** or with specific **groups** demands proof of identity before answering — and on the web, that proof is a **token**.

A token is a temporary pass: a long string of characters that says "the bearer of this string has been authenticated as this user" and expires after a set time. When you browse a private layer in Map Viewer, the software obtains and attaches tokens for you invisibly. When some other program wants access, it needs to obtain a token through one of a few sanctioned routes:

- **OAuth sign-in** — the industry-standard flow where the app bounces the person to an ArcGIS sign-in page, they log in, and the app receives a token on their behalf. This is what "Sign in with ArcGIS" means, and it is the right pattern for apps used by people. The app never sees the password.
- **API keys** — long-lived credentials you create in your developer or organization settings, scoped to particular services and capabilities. These suit unattended access: a script, a server, an embedded public-facing map that needs location services.
- **Application credentials** — an app authenticating as itself rather than as a person, used for server-to-server integrations.

You do not need to memorize the mechanics; you need the mental model: *people sign in via OAuth, machines use scoped keys, and both end up sending a token with each request.* Chapter 34 covers the administrative side — who can create keys, how sharing levels interact with credits and members, and how to audit access.

> **Watch out:** Never paste a username and password into another tool's "connection settings," and never hard-code a token or API key into anything shared — a public repository, an email, a StoryMap. A leaked key works for whoever holds it until it is revoked. If a tool can only accept a bare URL with no authentication support, the honest options are: share the layer publicly, create a filtered public *view* of it (see Chapter 10), or use a scoped API key that you treat like a password.

One more practical note: a token travels as just another parameter (`token=...`) or a request header. If you ever see a raw service URL with a long `token=` string attached, treat that whole URL as a secret — it *is* the credential.

Consuming Services from Other Software

The payoff for all this plumbing is interoperability. Because a feature service speaks plain HTTP — the web's ordinary request-and-response protocol — and JSON, an enormous range of software can consume it, with three broad patterns.

Native connectors

Some tools know what ArcGIS is and speak to it directly. Microsoft **Power BI** offers ArcGIS-aware integration; **Tableau** can connect to Esri sources; **QGIS**, the leading open-source GIS, adds `FeatureServer` endpoints as first-class layers (look for its ArcGIS REST connection option and paste the URL that ends at `/FeatureServer`); **Excel** and other Microsoft tools can reach services through Esri add-ins or plain web queries. ArcGIS Pro, of course, connects to any service URL — `Insert > Connections` is the neighborhood to explore, or simply add data from a path pasted into the catalog. With native connectors, you supply the service URL and sign in when prompted; the connector handles queries and pagination.

Generic JSON consumers

Any tool with a "get data from web" or "from JSON" feature — Power Query in Excel and Power BI, low-code platforms, automation tools like Make, Zapier, n8n, or Power Automate — can call a query URL you construct by hand. This is where the parameter fluency from the previous section pays off: you build a query URL that returns exactly the fields and rows you need (`returnGeometry=false` is your friend for tabular tools), test it in a browser, then paste it into the tool. Remember the pagination caveat: generic tools fetch one page unless you teach them the offset loop.

Scripted access

For anything beyond a one-off, scripts are the durable answer. The ArcGIS API for Python wraps all of this — authentication, querying, pagination, editing — in friendly objects, and Chapter 31 is devoted to it. Custom web applications use the Maps SDK for JavaScript, covered in Chapter 33. Both are, under the hood, generating exactly the

REST calls described in this chapter, which is why learning the raw API makes every higher-level tool easier to debug.

> **Tip:** When any integration misbehaves, drop down a level. Take the URL the tool is trying to call (most tools will show it in a log or error message), paste it into a browser, and read the raw response. Nine times out of ten the answer is right there: an authentication error, an invalid `where` clause, or an empty result because a spatial filter used the wrong coordinate system.

Export Endpoints: KML, GeoJSON, and Friends

Sometimes the receiving software doesn't want a live service — it wants a file. ArcGIS offers two distinct routes to that file, and knowing both saves you from fighting the wrong one.

Route one: format parameters on the query endpoint. The same `/query` operation can return open formats directly. `f=geojson` yields **GeoJSON**, the open, widely adopted JSON dialect for geographic data that web tools, GitHub, and most modern software understand natively. A carefully built query URL with `f=geojson` is effectively a "live download link": every time someone fetches it, they get current data in an open format. Two caveats. First, the record cap still applies — a GeoJSON query is still a query. Second, coordinates: the GeoJSON standard expects latitude–longitude values in the WGS 84 system, but the service does not re-project for you — it returns whatever coordinate system the query asks for, which can produce technically invalid GeoJSON from layers stored in a projected system. Add `outSR=4326` (the WGS 84 code) to GeoJSON queries so downstream tools get what the standard promises (Chapter 3 explains the distinction).

Route two: item-level export. On the layer's item page — the owner sees an `Export Data` option — you can generate a downloadable file in several formats: CSV (a plain spreadsheet-friendly table), shapefile (the venerable GIS exchange format), **KML** (the format Google Earth made famous, good for handing to people who live in Google's tools), GeoJSON, Excel, and file geodatabase (the richest option, preserving the most fidelity). Whether *other* people can export your layer is a setting you control on the item — look for the option that allows others to export the data, in the item's settings.

Item export produces a snapshot: a real file, frozen at the moment of export, with no record-cap gymnastics because the export machinery handles the whole dataset.

KML deserves one extra note: it is a *display* format as much as a data format — it can carry styling and descriptions — and round-tripping data through KML tends to flatten rich attribute tables. Use it as a final handoff to Earth-style viewers, not as a working format.

Webhooks: Services That Call You

Everything above follows a *polling* pattern: your tool asks, the service answers, and if you want to know about changes you must keep asking. A **webhook** inverts this. You register a URL of *yours* with the service, and when a chosen event happens — features edited, a survey submitted — the service sends a message to your URL, promptly and unprompted. Instead of "are we there yet?" every five minutes, the service taps you on the shoulder.

Two webhook flavors matter in the ArcGIS world:

- **Survey123 webhooks** are the most approachable: in the survey's management settings you can register a webhook so that each submitted survey fires a notification carrying the submitted values. Paired with an automation platform (Power Automate, Make, Zapier, n8n), this becomes "when a field crew submits an inspection, post to the team channel, email the supervisor, and append a row to the tracking sheet" — with no code. Chapter 30 covers Survey123 itself.
- **Feature service webhooks** fire when a hosted feature layer's data changes. The notification is deliberately lightweight — essentially "something changed, here is where to ask for details" — and your receiving system then queries for the specifics. These are configured by the layer owner or an administrator, typically through the service's administrative settings rather than the everyday item page, and availability and configuration details differ between ArcGIS Online and ArcGIS Enterprise — check what your environment supports before designing around them.

The conceptual points to carry away: webhooks require *you* to have a URL that can receive messages (automation platforms provide one with a click, which is why they pair so well); they deliver notifications, not guarantees, so critical pipelines still

reconcile against the service periodically; and they are the right tool whenever "how fresh?" matters more than "how much?"

Rate, Scale, and Being a Good API Citizen

A hosted service feels infinite until the day it doesn't. A few realities govern how services behave under load, and designing around them is the difference between an integration that hums and one that melts on launch day.

The record cap is a feature, not a bug. It exists so that no single greedy request can monopolize the service. Fight it with pagination when you truly need everything, but first ask whether you do: `returnCountOnly`, `outStatistics`, and tight `where` clauses let the server do the heavy lifting and send you a summary instead of a haul.

Repeated identical questions deserve cached answers. If a public dashboard asks the same query every few seconds on behalf of every visitor, consider whether the answer actually changes that often. Options, roughly in order of effort: query less often; enable caching or tile-based delivery options offered on the layer (hosted feature layers can serve data in an efficient cached fashion when the data is static or on a controlled update schedule); or, for genuinely static data, publish a tile layer and let the original feature layer rest (Chapter 10 discusses performance options for hosted layers in depth).

Editing traffic is heavier than reading traffic. Services that accept edits do more bookkeeping per request. High-volume field collection (Chapter 30) benefits from thoughtful sync settings and from keeping heavy analytical queries off the same layer during collection windows — layer *views* let you give analysts a read-only window while the crews write to the source.

Credits and quotas exist. In ArcGIS Online, storage and certain kinds of usage consume credits — the platform's internal currency — and organizations can face throttling if usage spikes pathologically. The everyday query traffic of a modest app is not the concern; the concern is the accidental infinite loop, the tool polling every second, the script that re-downloads a large layer nightly when a webhook or a change-detection query would do. Chapter 34 covers how administrators monitor and budget for this.

Tip: Before wiring any external tool to a production layer, create a *view* of the layer (Chapter 10) and point the tool at the view instead. A view lets you restrict fields, filter rows, and cap capabilities — and if the integration ever misbehaves, you can throttle or delete the view without touching the source layer or any other consumer.

Services or Exports: Choosing the Right Handoff

The final judgment call this chapter equips you for: when someone says "can I get that data?", do you hand them a live service URL or a file? Both are legitimate; they fail in different ways.

CONSIDERATION	LIVE SERVICE	EXPORT (SNAPSHOT FILE)
Freshness	Always current	Frozen at export time
Works offline	No	Yes
Load on your service	Ongoing, per consumer	One-time
Consumer needs auth handling	Yes, unless public	No — the file is self-contained
Reproducibility (audits, reports)	Answers change over time	Exact record of a moment
Very large one-time transfers	Painful (pagination)	Natural
Recipient's tooling	Must speak HTTP/JSON or have a connector	Anything that opens files

The pattern that emerges: **services for relationships, exports for transactions.** A dashboard that must reflect this morning's inspections needs the live service. A consultant doing a one-time analysis is better served by a file geodatabase export — they get everything at once, your service carries no ongoing load, and their results are

reproducible because their input can't shift underneath them. A report filed with a regulator should cite a snapshot, not a URL whose contents will drift. A partner organization building their own app on your data wants the service — with a view, a scoped key, and a conversation about expected traffic.

And frequently the answer is *both*: a live service as the system of record, plus scheduled exports (a small scripted task, once you've read Chapter 31) that drop dated snapshots somewhere safe. Freshness for the people who need it, history for the people who will need it later.

Everything downstream in this volume builds on this foundation. The Python chapter automates these same endpoints; the JavaScript chapter renders them; the administration chapter governs who may call them. When any of those layers gets confusing, come back to the plain URL underneath — paste it in a browser, read the JSON, and the mystery usually dissolves.

Building Web Apps: Maps SDK for JavaScript Primer

Everything you have built so far in this book — web maps, dashboards, StoryMaps, Experience Builder apps — was assembled by configuration. You clicked, you chose options, and Esri's software wrote the underlying code for you. This chapter is about the moment you step behind the curtain and write that code yourself, using the **ArcGIS Maps SDK for JavaScript**: Esri's official library for putting live, interactive maps inside web pages you build.

Let's be honest up front, because this chapter exists to save you time in both directions. Most people who think they need to write a custom map application do not. The configurable app builders — Instant Apps (Chapter 26), Dashboards (Chapter 28), Experience Builder (Chapter 29) — cover an enormous range of needs, and they come with free maintenance: when Esri updates them, your app updates too. But a real minority of projects genuinely need code, and for those projects, nothing else will do. The goal here is to give you an accurate picture of what the SDK is, how it thinks, what a working app actually looks like, and a clear-eyed way to decide whether you belong in

this chapter at all. You do not need to be a professional programmer to follow along. If you can read a recipe, you can read this code.

What the SDK Actually Is

A **software development kit**, or SDK, is a bundle of pre-written code plus documentation that lets developers build on top of someone else's platform without starting from scratch. The ArcGIS Maps SDK for JavaScript is a large JavaScript library — JavaScript being the programming language that runs inside every web browser — that knows how to draw maps, load ArcGIS layers, render symbols, handle clicks and pop-ups, and talk to ArcGIS Online or ArcGIS Enterprise on your behalf.

One naming note before you go searching: for many years this library was called the **ArcGIS API for JavaScript**, and you will still find that name all over older tutorials, forum posts, and books. Esri renamed it the ArcGIS Maps SDK for JavaScript in recent years. Same product, same lineage, new name. When you see "JS API" in a search result, it means this.

Here is the important conceptual point. The SDK is not a separate mapping system. It is a friendlier face on the exact same machinery you have been using all along. When Map Viewer draws a hosted feature layer, it is making requests to a REST endpoint — a web address that serves data — exactly as described in Chapter 32 (REST Services and Integration). When your custom app draws that same layer with the SDK, it makes the same requests to the same endpoint. The SDK simply wraps those raw web requests in convenient, well-documented JavaScript objects so you write `new FeatureLayer(...)` instead of hand-assembling URLs. Your hosted layers, your web maps, your symbology, your Arcade expressions from Chapter 8 — all of it works in a custom app, because a custom app is just another client of the same services.

What the SDK gives you that the configurable builders do not is *total control of the page*. The map can be any size, anywhere in your layout. The buttons can look like your organization's buttons. The map can react to things happening elsewhere on the page — a form, a chart, a search box you built — and the page can react to the map. That freedom is the entire value proposition, and also the entire cost, because everything the builders did for you automatically is now your job.

Tip: If you have never written any JavaScript, spend an evening with any free introductory tutorial before returning here. You need only the basics — variables, functions, objects — to be productive with the SDK. You do not need to master the language first.

The Mental Model: Map, View, and Layers

The best news in this chapter is that you already understand the SDK's architecture, because it deliberately mirrors what you know from Map Viewer (Chapter 6). Three concepts carry almost all of the weight.

The Map is a container of content. It holds a basemap and an ordered list of operational layers — the same distinction you know from the `Layers` panel and the basemap picker in Map Viewer. A Map object knows *what* to display but nothing about *where on the screen* to display it. It has no size, no zoom level, no rotation. It is pure content.

The View is the window onto that content. A view is bound to one specific spot in your web page — an HTML element you designate — and it owns everything about presentation: the current center point, the zoom level, the rotation, what happens on click and scroll. The SDK offers two kinds: a **MapView** for flat, two-dimensional maps, and a **SceneView** for 3D globes and scenes. The same Map object can be shown in either, which is a tidy demonstration of why content and presentation are kept separate.

Layers are the individual datasets, exactly as you know them: feature layers, tile layers, imagery layers, and so on. A **FeatureLayer** object in the SDK corresponds directly to a hosted feature layer from Chapter 10 — you typically point it at a service URL or an item ID and it handles the rest, including renderers (the styling rules from Chapter 7), labels, and pop-up configuration.

The correspondence with Map Viewer is close enough to use as a translation table:

MAP VIEWER CONCEPT	SDK EQUIVALENT
The map you're editing	`Map` (or `WebMap`) object

MAP VIEWER CONCEPT	SDK EQUIVALENT
The visible map area and zoom	`MapView`
`Layers` panel entries	`map.layers` collection
Basemap picker	`map.basemap` property
Styles pane (renderers)	`layer.renderer`
Pop-up configuration	`layer.popupTemplate`
Effects, filters, visibility ranges	Layer and view properties with similar names

One more idea rounds out the mental model: **reactivity**. Nearly everything in the SDK is a live property you can read, set, or watch. If you set `view.zoom` to a new number, the map animates there. If you watch `view.center`, your code gets notified every time the user pans. Building an app is largely a matter of wiring properties together: when *this* changes, do *that*. If you have used a spreadsheet, where changing one cell updates the formulas that reference it, the flavor is similar.

Your First Map, Line by Line

Here is a complete, working web page that displays a map. It is short enough to read in full, and we will walk through every piece.

```
        <!doctype html>
        <html>
        <head>
          <link rel="stylesheet"
                href="https://js.arcgis.com/<current-
version>/esri/themes/light/main.css">
          <script src="https://js.arcgis.com/<current-version>/"></script>
          <style>
            html, body, #viewDiv { margin: 0; padding: 0; height: 100%; }
          </style>
```

```
  </head>
  <body>
    <div id="viewDiv"></div>
    <script>
      require(["esri/Map", "esri/views/MapView"], (Map, MapView) => {

        const map = new Map({
          basemap: "topo-vector"        // a named Esri basemap
        });

        const view = new MapView({
          container: "viewDiv",         // the div to draw into
          map: map,                     // the content to show
          center: [-93.26, 44.98],      // longitude, then latitude
          zoom: 12
        });

      });
    </script>
  </body>
  </html>
```

Reading from the top: the `<!doctype html>` line tells the browser to use its modern rendering rules — leave it off and browsers fall back to a legacy mode where the full-height styling below can quietly misbehave. The next two lines in the `head` load the SDK from Esri's **content delivery network** (CDN) — servers that host the library so you don't have to download or install anything. One line brings in the stylesheet that makes maps and widgets look right; the other brings in the code itself. The `<current-version>` placeholder stands for a version number; copy the exact URLs from the current Getting Started page in Esri's documentation rather than from any book or old tutorial, because the library is updated on a regular cycle and you want a real, current version pinned in those URLs.

The small `style` block deserves more respect than its size suggests. Web page elements have no height by default, and a map drawn into a zero-height box is

invisible. Forgetting to give the container a height is, without exaggeration, the single most common beginner failure with this SDK — a blank page, no error message, nothing. Here we stretch the page and the container to full height.

The `div` with the id `viewDiv` is an empty box in the page. It is where the map will live. You could just as easily make it a small rectangle in the corner of an existing page — that placement freedom is the point of writing code.

Then the script. The `require(...)` call is the classic way this SDK loads the specific modules you need — think of it as fetching just two tools, `Map` and `MapView`, from a very large toolbox. You will see this pattern throughout years of samples and community code, though Esri's newer materials increasingly lead with the web-component style described below; both drive the same objects underneath. Inside, we create a Map with a named basemap (Esri publishes a set of well-known basemap names), then create a MapView that binds that map to our `viewDiv`, centered on a longitude-latitude pair at a chosen zoom level. Save this as an `.html` file, open it in a browser, and you have a pannable, zoomable map of Minneapolis.

> **Watch out:** The `center` property takes coordinates as `[longitude, latitude]` — x before y — while most humans say "lat, long." Swapping them puts your map in the ocean or the polar wastes. If your map opens somewhere absurd, check the order first. Chapter 3 (Coordinate Systems) explains why software consistently prefers x-y order.

> **Tip:** Recent versions of the SDK also offer **web components** — custom HTML tags like an `arcgis-map` element that wrap these same objects, so a basic map takes one line of HTML and no visible JavaScript. They are pleasant, and Esri's own tutorials now feature them prominently. Everything in this chapter about maps, views, layers, and web maps applies identically underneath; the components are a thinner way in, not a different system.

One word about authentication. Esri's basemaps and location services — geocoding (turning addresses into map locations) and routing — require your app to identify itself, typically with an **API key** — a long token string you generate in your ArcGIS account

and paste into your code — or through a user sign-in flow. Esri's tutorials show exactly where the key goes, and Chapter 34 (Administration) covers how keys are created, scoped, and what usage they meter. For quick experiments inside a signed-in sandbox like Esri's own, you can often defer this; for anything deployed, plan on it.

The Big Shortcut: Loading a Web Map by Item ID

The first example built a map from raw parts. You will almost never do that in real work, because there is a far better way: load an entire **web map** — the saved, configured map item you already know how to make in Map Viewer — by its item ID.

Every item in ArcGIS Online has a unique ID, the long string of letters and numbers visible in the item page's web address. The SDK's `WebMap` class accepts that ID and reconstructs the whole map:

```
require(["esri/WebMap", "esri/views/MapView"], (WebMap, MapView) => {

  const map = new WebMap({
    portalItem: { id: "paste-your-web-map-item-id-here" }
  });

  const view = new MapView({
    container: "viewDiv",
    map: map
    // no center or zoom needed — the web map remembers its own
  });

});
```

Pause on what just happened, because it is the most important practical idea in this chapter. Every choice you made in Map Viewer comes along for free: layer list and drawing order, renderers and smart mapping styles, pop-up configurations, Arcade expressions, labels, filters, visibility ranges, bookmarks, the saved initial extent. All of it. You configured it once with pleasant visual tools, and your code inherits the result in three lines.

This suggests a division of labor that experienced teams treat as doctrine: **configure in Map Viewer, present in code.** Keep the cartography, pop-ups, and layer settings in the web map item, where non-programmers can maintain them, and keep your JavaScript focused on the things only code can do — custom interface, page integration, unusual behavior. Done this way, a colleague can recolor a layer or reword a pop-up in Map Viewer and your deployed app picks up the change on next load, with no code edit, no redeployment, and no developer in the loop.

> **Tip:** Resist the urge to rebuild styling in code just because you can. Every renderer you hard-code in JavaScript is a renderer that now requires a developer to change. The web map is your configuration file; treat it as the single source of truth, in the spirit of Chapter 10's advice about layers and views.

> **Watch out:** Sharing levels still apply. If the web map or any layer inside it is private, visitors to your app will be prompted to sign in, and people without access get errors or missing layers. Before you blame your code, check the sharing settings of the web map *and each layer in it* — a public map containing one private layer is a classic source of "it works for me but not for anyone else." Chapter 34 covers sharing in depth.

You can also load a single hosted layer directly, without a web map, by giving `FeatureLayer` either a service URL or a `portalItem` ID. That is the right move when your app supplies all its own presentation and you just need the data. `map.add(layer)` puts it on the map, exactly like adding a layer from the `Layers` panel in Map Viewer.

Widgets: Prebuilt Interface Pieces

You could write your own legend from scratch — query the renderer, build the swatches, draw the labels. Nobody does. The SDK ships with **widgets**: prebuilt, tested interface components that you drop onto a view with a couple of lines. They are the same components you see inside Map Viewer and Esri's configurable apps, exposed for your use.

```
require(["esri/widgets/Legend"], (Legend) => {
  const legend = new Legend({ view: view });
  view.ui.add(legend, "bottom-left");
});
```

That is the whole pattern: construct the widget with a reference to your view, then add it to one of the view's corner positions. The widget stays synchronized automatically — the legend updates as layers toggle, the search widget pans the map to results, and so on. The commonly used ones:

WIDGET	WHAT IT DOES
Legend	Explains the symbols currently drawn
Search	Address and place search box (geocoding)
LayerList	Toggle layers on and off, like Map Viewer's panel
BasemapGallery / BasemapToggle	Switch basemaps
ScaleBar	Distance scale
Locate	Zoom to the user's device location
Home	Return to the initial extent
Sketch	Draw temporary graphics on the map
Editor	Full feature editing against editable layers (see Chapter 13)
Expand	Collapses any other widget behind a small button

A map plus a web map ID plus three or four widgets is genuinely most of a useful application, and it fits in a single screen of code. The craft in larger apps lies in the

interface *around* the map, and for that Esri offers a companion design system called **Calcite** — a set of buttons, panels, and layout components that match ArcGIS styling — which you can adopt when your ambitions grow beyond widget corners.

Where to Run It

A common early confusion: where does this code live? The answer is more relaxed than beginners expect.

For experiments: a plain `.html` file on your own disk, opened in a browser, works for simple cases like the examples above. Even easier are online code sandboxes — CodePen and similar sites — where Esri's own tutorials open with one click and you can fork and fiddle without saving anything locally. This is the recommended way to learn: change a value, see the map react, repeat.

For real development: as apps grow, developers run a small **local development server** — a program that serves your files to your own browser the way a real web server would — because some browser features behave differently for files opened directly from disk. Modern JavaScript tooling (the npm package manager and a **bundler** such as Vite — a tool that packages your code and everything it depends on into files a browser loads efficiently) can also install the SDK as the `@arcgis/core` package instead of loading it from the CDN, which is how professional projects usually consume it. None of that is required to start. The plain script-tag approach in this chapter works and is widely deployed, but Esri does shift its recommended loading patterns over the years — check the current Getting Started page before committing a long-lived project to one.

For deployment: here is the fact that surprises people — *ArcGIS Online does not host your custom code.* It hosts data, maps, and its own configurable apps, but a hand-written HTML-and-JavaScript app must live on a web server you arrange: your organization's web server, a university account, or any of the free-tier static hosting services (GitHub Pages, Netlify, and their kin). "Static hosting" just means serving fixed files, which is all this kind of app needs — there is no server-side code, because the SDK does its work in the visitor's browser and talks directly to ArcGIS services. ArcGIS Enterprise (Chapter 35) organizations sometimes have an internal web server designated for exactly this.

Once deployed, if your app uses sign-in or scoped API keys, you register the app in your ArcGIS account so the platform knows the app's web address is allowed to authenticate — a few minutes of setup covered under application items in Chapter 34.

> **Watch out:** Code you deploy is code you maintain. The SDK releases updates on a regular cycle, and while Esri is careful about compatibility, APIs do evolve and old versions eventually retire. If you pin a CDN version and walk away for three years, expect some homework when you return. This ongoing cost is real, and it is the strongest argument for the configurable builders, which Esri maintains for you.

Making It Look Right: Styling and Theming Basics

The SDK's visual layer is ordinary web styling — **CSS**, the language of fonts, colors, and layout on the web — which means anyone on your team who styles web pages can style your map app.

Three basics cover most needs. First, the stylesheet you load from the CDN comes in a **light theme** and a **dark theme**; swapping the word `light` for `dark` in that stylesheet URL restyles every widget for dark interfaces, a one-word change that makes night-mode dashboards look intentional. Second, the map container is just an HTML element, so all the normal layout tools apply: make the map full-screen, or a fixed panel beside a sidebar, or a card in a grid of cards. Responsive design — adapting to phone screens — is likewise standard CSS work, and the widgets themselves are touch-friendly out of the box. Third, widget appearance can be adjusted with CSS overrides, and recent versions expose tidy hooks for colors and fonts so you can nudge the components toward your brand without rebuilding them.

What the SDK does *not* restyle is the map itself — that is cartography, not CSS. Symbol colors, label fonts on features, basemap appearance: those live in the web map and its renderers, which is exactly where Chapter 7 (Styling and Smart Mapping) and Chapter 4 (Cartographic Design) taught you to control them. A custom basemap style, made with Esri's vector basemap style editor, can complete the branding: your colors under your data inside your interface.

Keep the height lesson from earlier in mind whenever you restructure a page: any layout change that collapses the map container's height produces the dreaded silent

blank map. It is such a rite of passage that you should simply expect to hit it once and know where to look.

Do You Actually Need Code? The Honest Ladder

Now the decision this chapter promised. Think of your options as a ladder, and stop at the lowest rung that solves the problem — every rung you climb trades maintenance burden for control.

RUNG	WHAT IT IS	EFFORT	YOU MAINTAIN
Embed	An `iframe` snippet pasted into an existing page	Minutes	Nothing
Instant Apps	Template apps configured from a web map (Ch 26)	Minutes to hours	Configuration only
Experience Builder	Drag-and-drop app layouts, multi-page, widgets (Ch 29)	Hours to days	Configuration only
Maps SDK (this chapter)	Hand-written web application	Days to months	Everything

Embedding deserves a special word because it is so often overlooked. From a web map or an app's share options you can copy an `iframe` snippet — a standard HTML tag that displays one web page inside another — and paste it into a blog post, an intranet page, or a content management system. If the need is "put this map on our existing site," embedding answers it with zero code and zero new hosting. Its limits are equally clear: the map is a sealed rectangle, and the surrounding page cannot interact with it in any deep way.

Instant Apps and Experience Builder own the broad middle ground. If your requirement sounds like "a map with a sidebar, filters, and some charts," describe it to Chapter 26 and Chapter 29 before writing a line of JavaScript; the odds are good one of them does it, and does it with accessibility, mobile support, and Esri-funded upkeep built in.

So when is code actually warranted? Honest signals, from years of watching teams choose:

- **The map is a component inside a larger application** — an existing customer portal, an internal tool, a public website with its own frameworks and login — rather than the application itself. Builders make whole apps; the SDK makes pieces that fit inside yours.
- **The interaction is genuinely custom.** Clicking a feature must update three other page elements you built; dragging a slider must re-run a client-side computation and restyle the layer; the workflow is an opinionated sequence no template anticipates.
- **Branding requirements are strict.** Not "our logo in the corner" — builders do that — but "must be pixel-identical to our design system," which only code delivers.
- **You need behavior beyond configuration**, such as bespoke client-side geometry work, custom data-fetch logic against services (Chapter 32), or performance tuning for an unusual data pattern.
- **You have, or are, someone who will maintain it** — not just build it. An unmaintained custom app is a slow-motion outage.

If none of those describe your project, closing this chapter and opening Chapter 26 is not a defeat; it is the correct engineering decision, and the one Esri's own solution engineers would recommend. If several describe it, welcome — the SDK is mature, thoroughly documented, and among the best mapping libraries in existence, and you will not outgrow it.

It is also worth knowing what this chapter's tool is *not* for. Automating analysis and administration is Python's territory — arcpy and the ArcGIS API for Python, covered in Chapter 31. Native mobile apps have their own Esri SDKs (Chapter 2 surveys the product line). The JavaScript SDK is specifically for things that run in a web browser.

Where to Go From Here

Your path onward, in order of usefulness:

Esri's tutorials and samples. The SDK's documentation site includes a large gallery of live, editable samples — nearly every capability demonstrated in a small page you can

fork and modify. Working programmers use these constantly; treating the sample gallery as a parts bin is not cheating, it is the intended workflow.

A deliberate first project. Take a web map you already own, load it by item ID, add a Search widget, a Legend behind an Expand, and a Home button, style the page to your organization's colors, and deploy it to a free static host. That single exercise touches everything in this chapter and produces something real.

The reactive habit. When you are ready to go deeper, study how to watch view and layer properties and respond to changes — it is the idiom that separates programs that fight the SDK from programs that flow with it.

The neighbors. Chapter 32 explains the REST services your app is quietly consuming, which demystifies every network request in your browser's developer tools. Chapter 8's Arcade skills transfer directly, since Arcade expressions execute inside SDK-rendered pop-ups and labels. And Chapter 34's coverage of API keys, app registration, and sharing is the difference between an app that works on your machine and one that works for your audience.

The distance between "I configured a map" and "I programmed a map" turns out to be shorter than it looks — a container, a view, an item ID, and a handful of widgets. The judgment about *whether* to cross that distance is the harder skill, and now you have both.

Administration: Users, Groups, Security, and Credits

Somebody has to run the place. Every ArcGIS Online organization — the shared account that holds your people, your content, and your settings — has at least one administrator, and if you are reading this chapter, that person is probably you. Administration is not glamorous work, but it is the difference between an org where people find what they need, share safely, and never hit a surprise bill, and an org that slowly fills with orphaned layers, over-privileged accounts, and a credit balance that drains for reasons nobody can explain.

The good news: ArcGIS administration is mostly a handful of decisions made once, plus a light routine repeated forever. This chapter walks through both. It covers who can do what (user types and roles), how sharing actually works (groups), what happens to content when people leave (ownership transfer), where the money goes (credits), and the security and organizational settings that quietly break things when they are set wrong. Everything here applies to ArcGIS Online; ArcGIS Enterprise — the version you install on your own servers — shares most of these concepts but adds its own layers, covered in Chapter 35 (ArcGIS Enterprise and When You Need It).

What You Are Actually Administering

An ArcGIS Online organization (usually just "the org") is a subscription that bundles three things:

- **Members** — the named accounts that sign in. Each one has a user type, a role, and possibly a credit allocation.
- **Content** — every map, layer, app, and file any member has ever saved. Each item has exactly one owner.
- **Settings** — org-wide switches for security, sharing, branding, defaults, and licensing.

As an administrator, your home base is the `Organization` tab in the top navigation of your org's website. It has sub-tabs — typically `Overview`, `Members`, `Licenses`, `Status`, and `Settings` — and nearly everything in this chapter lives under one of them. Esri rearranges these pages from time to time, so if a control has moved since this was written, it is almost certainly still somewhere under `Organization > Settings`; search the settings page rather than hunting menu by menu.

One mental model to carry through the whole chapter: **ArcGIS permissions are a stack of filters, and the most restrictive one wins**. What a member can actually do is whatever survives all three: their *user type* (what their license permits), their *role* (what the org allows them to do), and the *sharing settings* of each individual item (what they can see). A person can be blocked at any layer, and troubleshooting "why can't Maria edit this layer?" is almost always a walk down through those three filters in order.

User Types and Roles: Who Can Do What

New administrators routinely confuse user types with roles because both sound like job descriptions. They are different mechanisms, purchased and assigned separately.

User Types: The License Layer

A **user type** is the license attached to a member's account. It determines which apps the person can use and sets the *ceiling* on what capabilities they can ever be granted. You buy user types; they are the line items on your Esri invoice.

Esri renames and repackages user types often enough that any list printed here would age badly, but the pattern has been stable for years and is what actually matters:

PATTERN	WHAT IT IS FOR	TYPICAL APPS INCLUDED
Viewer-level	People who only look at maps and apps others made	Web viewing, dashboards, configured apps
Editor/contributor-level	People who edit data in apps someone else built, but do not create content	Web editing, some field apps
Field-worker-level	Mobile crews collecting and updating data in the field	Field Maps, Survey123, QuickCapture (see Chapter 30)
Creator-level	People who make maps, layers, and apps — the standard "GIS user" license	Map Viewer, app builders, most of the platform
Professional-level	Creators who also need ArcGIS Pro on the desktop	Everything above plus Pro and its extensions

The single most common (and most expensive) mistake is defaulting everyone to a creator-level type. Most organizations discover that the majority of their members only ever *consume* maps, and viewer-level types cost a fraction of creator-level ones. Audit honestly: if a person has never published anything, they probably do not need a license that lets them.

Tip: You can change a member's user type later — `Organization > Members`, find the member, and use the options on their row — as long as you have an unassigned license of the target type. Downgrading someone who owns content may be blocked until their content is transferred or their role is adjusted, because the lower type cannot support their current privileges.

Roles: The Privilege Layer

A **role** is a bundle of privileges — individual permissions like "create content," "publish hosted feature layers," "perform analysis," or "invite members." Where the user type is what you *paid for*, the role is what you have *decided to allow*. ArcGIS ships with default roles that cover most needs:

- **Viewer** — can see items shared with them, join groups, and nothing else. Cannot create or own content.
- **Data Editor** — everything Viewer can do, plus editing features in editable layers shared to them.
- **User** — can create their own maps and content and share it, but cannot publish hosted layers.
- **Publisher** — the everyday GIS role: create content, publish hosted feature layers (layers whose data lives in Esri's cloud and is served on demand; see Chapter 10), and run analysis tools that consume credits.
- **Administrator** — everything, including member management, org settings, and billing visibility.

A role can never exceed the user type's ceiling. You cannot make a viewer-type account a Publisher; the license simply does not support publishing, and the org page will not offer the combination. This is the filter model in action: user type sets the maximum, role selects a point at or below it.

Watch out: Keep the Administrator role rare — a good target is two or three people, never one and never ten. One administrator is a single point of failure (vacations, departures, forgotten passwords); many administrators means nobody

feels responsible and destructive mistakes get harder to trace. For people who need *some* admin powers, build a custom role instead, as described next.

Custom Roles: Least Privilege Without Friction

The default roles are coarse. Real orgs quickly hit cases like "our interns should make maps but never share anything publicly" or "the office manager should reset passwords but not touch content." Custom roles solve this.

You create them in the organization settings under the member roles section (`Organization > Settings > Member roles` in current layouts). Start from an existing role as a template, then check or uncheck individual privileges. Privileges come in two broad families:

- **General privileges** — things members do with content and collaboration: create, publish, share publicly, edit, run analysis, use premium content like geocoding (converting addresses into map points) and GeoEnrichment (a paid service that appends demographic data to your features).
- **Administrative privileges** — things done *to the org*: view all members, manage licenses, update member roles, manage security settings, view the credit status pages.

A few custom roles earn their keep in almost every organization:

- **Publisher-minus-public.** Everything Publisher has, with "share with public" unchecked. New staff get this until they have learned the org's publication standards; it prevents the classic accident of internal data going public. Pair it with the org-wide public-sharing setting discussed under security below.
- **Analyst-no-premium.** Publisher privileges minus GeoEnrichment and premium content, for people who run spatial analysis but should not be able to trigger the pricier per-record credit charges by accident.
- **Admin-lite.** Viewer or User privileges plus a small set of administrative ones — view members, reset passwords, manage licenses — for a front-desk or IT person who handles account requests but should never touch content or security settings.

Tip: Name custom roles for what they permit, not for departments ("Publisher — no public sharing" rather than "Planning Dept role"). Departments reorganize; the privilege sets outlive them, and future administrators need to know at a glance what a role actually does.

Resist the urge to create a custom role per person. Every role you add is something a future administrator must understand before they can safely change anything. Three to five custom roles is typical; fifteen is a maintenance problem.

Inviting and Managing Members

Members enter the org through the `Organization > Members` page, via an invite or add flow. You will encounter three patterns:

1. **Invite by email.** The person receives a link and creates their own account. Least work for you; you are trusting them to finish the process.
2. **Add directly with credentials you set.** You create the username and an initial password, then hand it over. Useful for batches — the flow accepts a file listing many members at once — and for accounts that are not really people, like a shared display account for a lobby kiosk.
3. **Organization-specific logins (SSO).** If your organization already has central logins — Microsoft, Google, or another identity provider — the org can be connected to it via SAML or OpenID Connect (industry standards for single sign-on) so people use their existing work credentials. This is configured in the security settings and is worth doing early if you have more than a couple dozen members, because it means departures are handled by your IT identity system automatically.

Whichever route you use, set the **new member defaults** first (in organization settings, under new member defaults). These decide what every incoming member gets automatically: user type, role, credit allocation, the groups they join, and even the apps licensed to them. Orgs that skip this step end up with every new hire as a full creator-and-publisher, which is how privilege sprawl starts. Set the defaults to the most common *low* configuration — say, viewer type with the Viewer role and membership in a few org-wide groups — and upgrade individuals deliberately.

Day-to-day member management is mostly three verbs:

- **Disable**, which locks the account but preserves everything — the right first move whenever someone leaves or an account looks compromised.
- **Delete**, which removes the account permanently. ArcGIS will refuse to delete a member who still owns content or groups; you must transfer or delete their items first (covered below). This refusal is a feature, not an obstacle — it is the platform preventing you from orphaning live data.
- **Categorize.** Member categories are labels you define (departments, sites, employment status) that make the members list filterable. In an org above about thirty people, categories are the difference between "audit the field crew's licenses" being a five-minute filter or an afternoon of guesswork.

Groups: The Sharing Backbone

Everything about sharing in ArcGIS runs through four levels: an item is **private** (only the owner and administrators see it), shared to one or more **groups**, shared to the whole **organization**, or shared with the **public** (everyone on the internet, no sign-in required). Every new item starts private — nothing becomes visible to anyone until its owner deliberately shares it, which is the platform's one merciful default. You met this model when you shared your first map; administration is where you design it on purpose.

A **group** is a named collection of members and a named collection of items, joined together: members of the group can see the items shared to it. That simple mechanism carries almost every collaboration pattern on the platform:

- **Project groups** — the people and content for one project, created at kickoff and archived at closeout.
- **Department or team groups** — standing groups matching your org chart, usually joined automatically via new member defaults.
- **App-backing groups** — several parts of the platform (Instant Apps gallery templates, your org home page's featured content, ArcGIS Hub content libraries) present "whatever is in this group" as their content, so the group becomes a publishing channel: share an item to the group and it appears in the app.

- **Cross-org groups** — groups can admit members from *other* ArcGIS organizations, which is the lightweight way to collaborate with a partner agency without giving them accounts in your org.

When you create a group, three settings matter most, and they answer three separate questions: **who can find and join it** (invitation-only, request-to-join, or open), **who can see its content** (members only, or everyone in the org, or the public), and **who can contribute items** (all members, or only the owner and managers). Most confusion about groups traces back to one of these three being set on autopilot.

Shared Update Groups

Ordinary groups let members *see* each other's items. A **shared update group** goes further: members can *edit* each other's items — open a colleague's web map, change it, and save it in place, without saving a copy. This is how a team maintains one authoritative map with multiple hands on it, and it is essential for shift-based operations (a dashboard maintained by whoever is on duty) and for vacation coverage.

Because it hands out edit rights to other people's work, creating this kind of group requires administrative privilege (or a custom role granted it), and the update capability is chosen when the group is created — in many versions it cannot simply be toggled on later, so plan for it at creation rather than assuming you can upgrade an existing group.

> **Watch out:** Shared update groups do not keep a version history of web maps and apps. If two people edit the same map, the last save wins, silently. Keep shared update groups small, populated only by people who genuinely co-maintain the content, and make sure the team knows to communicate before big edits. For the *data* underneath — hosted feature layers — editing is governed separately by the layer's own settings; see Chapter 10 (Hosted Feature Layers).

One administrative habit worth adopting: have important groups owned by an administrator account or a designated org account rather than by whichever staff member happened to create them. Group ownership can be reassigned later (from the group's settings or the admin member tools), but starting with deliberate ownership avoids the scramble when the group's owner resigns.

Content Ownership and Transfer

Every item has exactly one owner, and by default only the owner (and administrators) can modify or delete it. That is a fine model right up until the owner leaves the organization — and their account owns the hosted feature layer that feeds the public dashboard the director checks every morning.

The mechanics of transfer are simple. An administrator can reassign a single item from the item's settings page (there is a change-owner control), or reassign *everything* a member owns at once from `Organization > Members` using the transfer-content option on the member's row. Item IDs — the unique identifiers baked into every ArcGIS URL — do not change when ownership transfers, so maps and apps that reference a transferred layer keep working. Group memberships are handled separately: transferring content does not automatically put the new owner into the groups the items were shared with, so check group membership as part of the same operation.

The strategy matters more than the mechanics:

- **Do it before the person's last day**, while they can still tell you what each item is and whether it is safe to delete.
- **Transfer to a role, not a friend.** Content dumped onto a random colleague's account just moves the problem. Many orgs maintain a small number of shared "steward" accounts (for example, one per department, creator-typed, credentials held by the team lead) that exist to own production content. People come and go; the steward account persists.
- **Delete the junk during the transfer.** A departing member's account is usually one-third production content and two-thirds experiments. Transferring everything indiscriminately just relocates the clutter — and hosted layers keep consuming storage credits wherever they live.

Once a member owns nothing — no items, no groups — they can be deleted cleanly. Until then, disable the account and work the transfer list.

Credits: The Organization's Currency

Credits are the metered currency of ArcGIS Online. Your subscription includes a pool of them; certain operations draw the pool down; when it is empty, those operations stop

working until you buy more. Administrators do not need to memorize rate tables (Esri publishes and revises them; never trust a number you read in a book), but you absolutely need to know *which categories of activity* consume credits, because they are wildly unequal.

ACTIVITY	CREDIT BEHAVIOR
Viewing maps, using apps, drawing on maps	Free — no credit cost
Hosting web maps and most apps	Free — the items themselves cost nothing to keep
Storing hosted feature layers	Recurring — charged continuously based on data size, forever, until deleted
Storing files and tile layers (pre-rendered map image tiles)	Recurring, at a much lower rate than feature storage
Spatial analysis tools in Map Viewer	Per run, scaled to how many features are processed
Batch geocoding (turning address tables into points)	Per record — large tables add up fast
GeoEnrichment (appending demographic data)	Per record and per variable — among the fastest ways to spend credits
Publishing tile layers	Per tile generated — large areas at many zoom levels get expensive
Hosted notebooks and premium services	Metered while running or per use

Two structural facts fall out of that table. First, **storage is the only recurring charge** — everything else is a one-time hit when someone clicks a button. An org whose credits drain steadily with nobody running analysis has a storage problem: old hosted layers, forgotten copies, or feature layers bloated with photo attachments. Second, **the expensive operations are per-record**, which means the danger is not the tool, it is the

tool pointed at a big table. Geocoding fifty addresses is trivial; geocoding half a million is a budget event.

Your two defensive tools:

Credit budgeting. In the organization settings you can enable credit budgeting and give each member (or each new member, via defaults) an allocation — a personal spending cap. When a member exhausts their allocation, their credit-consuming operations stop; everyone else is unaffected. This converts "one intern geocoded the entire customer file and drained the org" from a crisis into a support ticket. Give generous allocations to trained analysts and small ones to everyone else; you can top up an individual in seconds.

The status dashboard. `Organization > Status` shows credit consumption over time, broken down by category and by member, along with usage reports you can export. Look at it monthly at minimum. The shape of the chart tells you the story before any drill-down: a flat steady slope is storage; spikes are analysis or geocoding runs, and the member breakdown tells you whose.

> **Tip:** Before anyone runs an analysis tool in Map Viewer, the tool panel shows an estimate of the credits the run will consume. Teach every Publisher in your org to read that number before clicking run. It is the single highest-leverage piece of credit training you can give, and it costs nothing.

Security Settings: The Ones That Matter

Everything in this section lives in the organization's security settings (`Organization > Settings > Security` in current layouts). Most of the page can be left at defaults; a handful of switches deserve a deliberate decision.

Public sharing. There is an org-wide setting controlling whether members can share content with the public at all. For any org handling sensitive data, the strong pattern is: turn org-wide public sharing *on* but restrict *who* can do it — via custom roles without the share-publicly privilege — and route public publication through a small set of trained people. If instead you leave every Publisher able to share publicly, assume that sooner or later an internal layer will be public, discovered not by you but by someone

searching ArcGIS Online. Public items are indexed and findable; "nobody knows the URL" is not a security model.

Anonymous access. Separately from item-level sharing, the org itself has a website (your org's home page, galleries, and search), and a setting controls whether people who are not signed in can see it. Turning anonymous access off makes your org's site invitation-only — appropriate for internal-facing orgs — but understand the coupling: with anonymous access disabled, public sharing from your org effectively stops working, because visitors who are not signed in cannot reach your org's content at all, and the public sharing level generally becomes unavailable to members. Nothing you host can be embedded on a public website in this state. Orgs that want a public face and a private interior keep anonymous access on and control exposure through item sharing levels and the share-publicly privilege instead.

Multifactor authentication (MFA). For members with ArcGIS-managed logins, the org can allow or require a second sign-in factor from an authenticator app. If your members sign in through your company's SSO instead, MFA is your identity provider's job and the ArcGIS setting does not apply to them. Either way, someone should be enforcing a second factor, especially for administrators — an admin account hijack exposes every private layer in the org. Esri requires the org to designate more than one administrator with a working email before enabling org-managed MFA, so lockouts can be recovered.

Allowed origins. By default, web applications running on any website can talk to your org's services from the browser (the mechanism is CORS — cross-origin resource sharing — the web's rules for which sites may call which services). The allowed-origins list restricts that: once you add any entries, *only* the listed websites can make browser-based requests to your org. It is a real hardening measure, and also a classic foot-gun — see the next section.

Password policy and access notice. If you use ArcGIS-managed logins, set the password policy (length, complexity, expiration) to match your IT standards, and consider the sign-in access notice if your sector requires terms-of-use banners. With SSO, skip both; your identity provider owns them.

Organizational Settings That Bite

A grab-bag of settings that are configured once, forgotten, and then blamed on the wrong thing when they misfire:

- **Allowed origins set too tightly.** The team builds a custom web app (Chapter 33), deploys it to a new domain, and every map in it fails with cryptic browser errors — because the new domain was never added to the allowed-origins list. If web maps work in ArcGIS itself but die inside a custom app, check allowed origins before you debug anything else.
- **New member defaults never configured.** Covered above, but it bites late: eighteen months in, you discover forty creator licenses assigned to people who have only ever opened a dashboard, and unwinding it means forty individually awkward conversations.
- **Public sharing disabled org-wide as a "temporary" measure.** Weeks later, someone spends a day debugging why their finished StoryMap "won't share" — the option is simply missing from their sharing dialog, with no explanation pointing at the org setting. If a sharing level is mysteriously absent from someone's dialog, the org setting or their role removed it.
- **Default basemap, extent, and units.** Cosmetic, but every new map every member creates starts from these. Setting your region's extent and your preferred basemap gallery once saves the whole org a small annoyance forever.
- **Administrative contacts.** The org lists contacts who receive Esri's operational emails — renewal notices, credit warnings, security notifications. If those point at someone who left two years ago, your first warning that credits ran out will be the platform refusing to geocode. Check the contacts whenever an administrator departs.

Watch out: Settings changes take effect org-wide and immediately, and there is no undo history for the settings pages. Before flipping anything under `Organization > Settings > Security`, note the current value somewhere (a screenshot is fine) so you can restore it if the change breaks a workflow you did not know existed.

Housekeeping: The Healthy-Org Routine

None of the above stays healthy by itself. The difference between a five-year-old org that works and one that terrifies its new administrator is a boring, recurring routine. A workable cadence:

CADENCE	TASKS
Monthly	Review the credit status dashboard; investigate any new slope or spike. Check for members who have never signed in and disable stale invites. Scan newly public items (filter org content by sharing level) and confirm each was intentional.
Quarterly	License audit: compare each member's user type against actual activity, downgrade the over-provisioned. Review role assignments, especially Administrator. Run member and content reports (exportable from the status/reports area) and archive them as a baseline.
Annually or at staff changes	Content spring-cleaning: find items not modified or viewed in a year, confirm with owners, delete or archive. Verify administrative contacts, admin count (two or three), and MFA enforcement. Re-read the security settings page top to bottom — Esri adds settings between visits, and new ones arrive with defaults you did not choose.
Every departure	Disable the account immediately; transfer content to a steward account within the notice period; delete the member once they own nothing.

Two habits make all of this dramatically easier. First, **use the reports**: the status area can generate exportable reports of members and items, and a saved quarterly snapshot turns "how did we get here?" into a diff between two files. Second, **write the decisions down**: a one-page internal doc listing your custom roles and what they are for, your steward accounts, your allowed origins and why each is there, and your credit allocation tiers. Administration transfers badly through folklore and very well through one page of notes.

Run the routine, keep the notes current, and the org stays the way you built it: people with exactly the access they need, content with owners who exist, credits going where you decided they should, and nothing public that should not be. That is the whole job — and done this way, it fits comfortably in a few hours a month.

ArcGIS Enterprise and When You Need It

Almost everything in this book so far has quietly assumed ArcGIS Online: Esri runs the servers, you sign in through a browser, and your maps and layers live in Esri's cloud. ArcGIS Enterprise is the other half of the story. It is the same web GIS platform — the same maps, layers, apps, sharing model, and REST services (the standard web endpoints apps use to request maps and data, covered in Chapter 32) — but installed on computers that *you* control. Your servers, your network, your databases, your rules, and, inescapably, your maintenance burden.

This chapter explains what Enterprise actually is in plain architectural terms, how it differs from Online in ways that matter for real decisions, and — most importantly — how to tell whether you need it. Plenty of organizations buy Enterprise because it sounds more serious and then drown in server administration they never wanted. Plenty of others struggle for years to force sensitive workflows into Online when Enterprise was the honest answer from day one. The goal here is to help you land in the right camp deliberately.

What ArcGIS Enterprise Actually Is

Strip away the branding and Enterprise is a set of server software products that, installed together, recreate the ArcGIS Online experience inside your own infrastructure. When it is set up, your users open a browser, go to an address on your network (something like `gis.yourcompany.com/portal`), sign in, and see a home page, a content gallery, groups, Map Viewer, and app builders that look and behave almost exactly like ArcGIS Online. Most of what you learned in Chapters 6 through 10 transfers directly. The difference is entirely in what sits behind that browser page.

Behind ArcGIS Online sits Esri's global cloud, which you never see and never manage. Behind ArcGIS Enterprise sit machines your organization owns or rents — physical servers in a back room, virtual machines in your data center, or cloud instances in an Amazon, Microsoft, or Google account that your IT team controls. Esri writes the software; you run it.

The four base components

A standard Enterprise deployment — Esri calls the minimal version a *base deployment* — is built from four pieces of software. Understanding what each one does demystifies almost every Enterprise conversation you will ever sit through.

Portal for ArcGIS is the front door and the brain of the sharing model. It provides the website your users sign into, and it keeps track of *who* exists (users), *what* exists (items: maps, layers, apps), and *who may see what* (groups and sharing levels). If you have used ArcGIS Online, you already know Portal — it is functionally the self-hosted twin of the Online website. When someone says "share it to the portal," they mean the same thing an Online user means by "share it to my organization."

ArcGIS Server is the workhorse that actually serves maps and data over the web. It takes GIS resources — a map, a set of features, a geoprocessing tool, an image collection — and publishes them as *web services*: live endpoints that apps can draw from, query, and edit through the REST interface described in Chapter 32 (REST Services and Integration). In a base deployment, one ArcGIS Server is designated the *hosting server*, which gives the portal the ability to host feature layers the way ArcGIS Online does — users can upload a spreadsheet or publish from Pro, and the layer just works, without anyone hand-configuring a service.

ArcGIS Data Store is a managed database that the hosting server uses to store the data behind those hosted layers. You install it, point it at the hosting server, and then largely leave it alone — Esri manages its internal schema, upgrades, and structure. It comes in a few flavors: a *relational* store for ordinary hosted feature layers, a *tile cache* store for 3D scene layers, and a *spatiotemporal* store for very high-volume observation data (think millions of sensor readings or vehicle pings). Most organizations start with just the relational and tile cache types. The key thing to grasp: Data Store is Esri's database, managed Esri's way, for *hosted* content. Your own corporate databases are a separate topic we will get to shortly.

ArcGIS Web Adaptor is the least glamorous piece: a small component that sits with your organization's web server and forwards incoming browser traffic to Portal and Server on standard web addresses and ports. It is essentially plumbing — it lets users reach the system through a normal-looking URL, integrates with your organization's

existing web infrastructure, and keeps the internal machines from being directly exposed. You rarely think about it after setup, except when it breaks.

> **Tip:** When an Enterprise diagram overwhelms you, reduce it to this sentence: *Portal is the catalog and login, Server does the map-serving work, Data Store holds the hosted data, and the Web Adaptor is the front-door plumbing.* Every fancier architecture is these four ideas, multiplied and rearranged.

How a request flows through the stack

A concrete walkthrough makes the architecture stick. Suppose a colleague opens a web map of water mains on your internal portal.

Their browser first hits the Web Adaptor, which forwards the request to Portal. Portal checks who they are and whether the map is shared with them, then hands their browser the map's definition — which layers it contains, how they are styled, where their services live. The browser then requests features from those services, which are running on ArcGIS Server. For a hosted layer, Server pulls the features out of the Data Store; for a layer referencing your utility database (more on that soon), Server queries that database directly. Features flow back through the same chain and appear on screen. Every click, query, and edit repeats some version of this loop.

Nothing in that flow touched the internet. That single fact is the root of most reasons organizations choose Enterprise.

All on one machine, or spread across many

A base deployment can live entirely on a single well-resourced machine — all four components installed side by side. That configuration is real, supported, and appropriate for small teams. Larger organizations split the components across multiple machines, add second copies for *high availability* (so the system survives a machine failing), and add extra ArcGIS Server machines for specific workloads. Recent versions of Enterprise can also be deployed on Kubernetes, a container-orchestration platform, for organizations whose IT teams already operate that way — the architecture concepts stay the same even though the installation mechanics differ completely.

The scaling story is genuinely flexible, but hold on to the honest corollary: every machine you add is a machine someone must patch, monitor, back up, and troubleshoot. We will price that out — in effort, not dollars — before this chapter ends.

Online versus Enterprise: The Decision Factors

Chapter 2 (The ArcGIS Ecosystem) introduced the Online/Enterprise split at a glance. Here is the full decision, factor by factor. Notice that most of these factors are not about GIS capability at all — the mapping features overlap heavily. They are about *data governance, networks, and control*.

FACTOR	POINTS TOWARD ARCGIS ONLINE	POINTS TOWARD ARCGIS ENTERPRISE
Where data must live	No legal or policy constraint on cloud storage	Data must stay in your country, your data center, or your security boundary
Data sensitivity	Public or low-sensitivity data	Regulated, classified, or contractually restricted data
Existing corporate databases	Little or no existing database estate	Large investment in SQL Server, PostgreSQL, or similar databases that GIS must read and write directly
Network environment	Users have reliable internet	Air-gapped (physically isolated from the internet), offline, ship-board, or field-forward networks
IT capacity	Small or no server-administration staff	Capable IT team willing to own another production system
Upgrade appetite	Comfortable with Esri updating the platform on Esri's schedule	Need to control exactly when and whether anything changes
Scaling model	Prefer paying as you grow with no hardware	Prefer owning capacity, or need compute close to huge datasets

A few of these deserve more than a table row.

Data residency and sovereignty

Many governments, utilities, healthcare organizations, and defense contractors operate under rules that say certain data may not leave a jurisdiction or may not sit on shared cloud infrastructure at all. ArcGIS Online stores your content in Esri's cloud regions; you get some say over which region, but the infrastructure is fundamentally shared and Esri-operated. If your legal or compliance team reads a data-handling policy and says "this cannot go to a third-party cloud," the conversation is effectively over — Enterprise (or no web GIS at all) is your path. This is the cleanest and least arguable reason to choose Enterprise: it is a rule, not a preference.

Sensitive data short of "forbidden"

More common is the gray zone: data that *could* legally live in a vetted cloud but that your organization is nervous about — customer records, infrastructure locations, pre-decision land acquisition parcels. Here the decision is genuinely a judgment call. ArcGIS Online has robust access controls, and Esri maintains an extensive set of security certifications, so "cloud" does not mean "public." But some organizations simply sleep better when sensitive layers live behind their own firewall, and that comfort has real organizational value. Be honest about whether your constraint is a rule, a defensible risk posture, or an unexamined habit — the answer changes what you should buy.

Integration with internal databases

This is, for many organizations, the deciding factor in practice. If your asset records live in a SQL Server database that feeds your billing system, your work-order system, and your GIS, you want the GIS to read and write *that database directly* — not a copy of it. ArcGIS Online cannot connect to a database inside your network; it only serves data that has been copied up into its cloud. ArcGIS Enterprise can sit right next to your databases and serve them live. The next major section covers this in depth, because it is the single most consequential architectural difference between the two platforms.

Disconnected and offline networks

Some environments have no internet at all: secure government networks, ships, mines, remote construction sites, disaster-response field operations. ArcGIS Online is by definition unreachable from such a network. Enterprise can be installed entirely inside

it — including basemaps, geocoding (address-to-coordinate matching) services, and routing services built from data you load yourself — and deliver a full web GIS to users who will never touch the public internet. If this describes your world, the decision was made for you.

Credits, and the absence of them

ArcGIS Online meters certain operations — storage, some analysis tools, geocoding, and similar premium services — in *credits*, a consumable currency covered in Chapter 34 (Administration: Users, Groups, Security, and Credits). Enterprise, running on your own hardware, has essentially no credit meter for the work your own servers do. Hosting a gigantic feature layer or running heavy analysis costs you disk space and processor time you already own, not credits. This sounds like a pure win, and for very heavy, steady workloads it can be. But do not treat it as free: you traded a metered bill for hardware, software licensing, and staff time. Organizations that move to Enterprise "to save credits" and then hire an administrator have often made an expensive trade.

> **Watch out:** An Enterprise portal can still consume ArcGIS Online credits if you configure it to use Esri's cloud utility services — for example, using Esri's World Geocoder for address matching instead of building your own locator. "We run Enterprise" does not automatically mean "nothing meters." Check which utility services your portal is actually pointed at.

The choice is not exclusive

Before treating this as a fork in the road, know that many mature organizations run *both*: Enterprise inside the firewall for authoritative data and sensitive editing, Online outside it for public engagement, lightweight collaboration, and ready-made content like the Living Atlas (Chapter 5). The hybrid section later in this chapter shows how the two are formally connected.

Federation: How Servers Join the Portal

You will hear the word *federated* constantly in Enterprise settings, and it sounds more complicated than it is. Federation is the act of formally joining an ArcGIS Server

machine to a Portal, so that the two stop being separate products and start acting as one system.

Before federation, an ArcGIS Server is a free agent: it has its own list of administrators, its own security model, and services that it manages independently. After federation, the portal takes over. The server's services become *items* in the portal's content system — searchable, shareable with groups, ownable by users — and the portal's users and sign-in rules govern who can touch them. One identity, one catalog, one sharing model, across every machine in the deployment.

In a base deployment there is one federated server playing the *hosting server* role. Growing deployments federate additional servers and assign them *roles* — specialized jobs licensed and configured for particular workloads. The names and packaging of these roles have shifted across versions, so hold the concepts rather than the labels: there are server roles for serving and analyzing *imagery* at scale, for processing *real-time data feeds* like vehicle locations and sensor streams, for running *hosted notebooks* (the Python environments discussed in Chapter 31), and for other specialized workloads. Each role is the same underlying ArcGIS Server engine wearing a different uniform, federated to the same portal, showing up in the same catalog.

Why should you, a non-administrator, care about federation at all? Two reasons. First, it explains the mental model: when your Enterprise colleagues talk about "the portal," they mean the whole federated organism, not just the website. Second, it explains certain permanence warnings you may encounter — federation is close to a one-way door.

> **Watch out:** Federating a server, and especially designating a hosting server, is effectively permanent in any deployment with real content. Un-federating deletes or orphans hosted content and breaks item references. If you are ever involved in planning an Enterprise rollout, treat the federation design as a decision to get right on paper *before* anyone clicks the button — not something to refactor later.

Your Databases, Your Data: Referenced versus Hosted

Here is the capability that most decisively separates Enterprise from Online, and it deserves careful unpacking because the vocabulary is subtle: the difference between

hosted data and *referenced* (also called *registered*) data.

The enterprise geodatabase

First, a definition. An *enterprise geodatabase* is not a separate database product you buy from Esri. It is an ordinary industrial database — Microsoft SQL Server, PostgreSQL, Oracle, and several others are supported — into which Esri's software has installed a set of tables, functions, and conventions that teach it to store and manage geographic data properly. Your database administrators keep running the database exactly as they always have: same backups, same security, same monitoring. The geodatabase layer on top adds the spatial intelligence — geometry storage, spatial indexing, and the advanced behaviors covered elsewhere in this volume, such as the versioned editing workflows of Chapter 13 and the attribute rules and topology of Chapters 12 and 14.

For organizations with decades of records in SQL Server or PostgreSQL, this is the crown jewel of the Esri stack: the GIS does not demand that your data move into some proprietary silo. It moves *in with* your data.

Hosted layers: the copy model

A *hosted* feature layer — whether on ArcGIS Online or on Enterprise's hosting server — follows a copy model. When you publish, your data is copied into the system's managed storage (Esri's cloud for Online; the ArcGIS Data Store for Enterprise), and the layer serves from that copy. The original file on your desktop is now irrelevant; the hosted copy is the live data. Chapter 10 (Hosted Feature Layers) covers this world thoroughly, and everything there applies to Enterprise's hosted layers nearly unchanged.

The copy model is wonderfully simple — no database administration, fast publishing, good performance — and its weakness is exactly its strength: it is a copy. If the authoritative version of the data lives in a corporate database that other systems also use, a hosted copy immediately begins to drift out of date, and now you own a synchronization problem.

Referenced layers: the live-connection model

A *referenced* layer works differently, and only Enterprise can do it against your internal databases. You register your enterprise geodatabase with a federated ArcGIS Server — essentially telling the server, "here is a database you are allowed to talk to." Then you publish services whose layers *point at the tables in that database* rather than copying

them. When a user pans the map, the server queries your database live. When a field crew edits a feature through a web app, the edit lands in your database, in the same tables your other business systems read, the moment it is saved.

There is no copy, no sync job, no drift. The web map and the billing system and the engineering desktop are all looking at one set of rows.

Choosing between them

Both models are legitimate, and mature Enterprise shops use both deliberately.

CONSIDERATION	HOSTED (COPIED INTO DATA STORE)	REFERENCED (LIVE FROM YOUR GEODATABASE)
Authoritative source	The hosted layer itself	Your corporate database
Database administration needed	Essentially none	Real database-administrator involvement
Publishing effort	Minimal — upload and go	Moderate — register, configure, publish from Pro
Other business systems see edits	Only via export or sync you build	Immediately, natively
Advanced geodatabase behaviors (versioning, topology, rules)	Limited	Full
Typical use	Project data, survey results, one-off layers, public copies	Authoritative asset and records data shared across systems

A reasonable rule of thumb: data that is *born in the GIS* and lives only there can be hosted; data that is *shared with the rest of the business* should be referenced from an enterprise geodatabase. Organizations that get this backwards — authoritative utility networks in hosted layers, or throwaway project scratch data ceremonially loaded into the corporate database — create friction in both directions.

Tip: When someone proposes a new layer, ask one question first: "If this data changes, what else in the organization needs to know?" If the answer is "nothing," host it and move on. If the answer involves other departments or other software, you are describing referenced data in an enterprise geodatabase.

Hybrid Patterns: Distributed Collaboration

Choosing Enterprise does not exile you from ArcGIS Online, and the formal bridge between the two is a feature called *distributed collaboration*. It lets two or more web GIS organizations — Enterprise to Online, or Enterprise to Enterprise — establish a trusted relationship and share content across the boundary in a controlled, ongoing way.

Mechanically, the participants agree on shared *workspaces*, and each organization links one of its groups to a workspace. Items placed in the linked group flow to the partner organization according to the collaboration's rules. Depending on the item type and configuration, content is shared either as a *reference* (the partner sees your live service) or as a *copy* that synchronizes at intervals — feature data crossing a network boundary generally travels as synchronized copies, since the partner's users may not be able to reach your internal servers directly.

Why this matters is best seen through the patterns organizations actually run:

The publish-outward pattern. A city runs Enterprise internally, where staff edit authoritative data against the enterprise geodatabase all day. A collaboration pushes selected, cleaned layers to the city's ArcGIS Online organization on a nightly sync, where they feed the public open-data site, StoryMaps (Chapter 27), and dashboards (Chapter 28). Sensitive fields and layers simply never enter the shared group. The public gets fresh data from Esri's fast, resilient cloud; the crown-jewel database never faces the internet.

The field-bridge pattern. Field crews collecting data with the mobile apps from Chapter 30 sometimes work far from any VPN (the secure tunnel that lets remote devices reach the internal network). Some organizations have crews collect against ArcGIS Online — reachable from any cell connection — and use collaboration to bring

the collected features back into Enterprise, where they are reviewed and merged into the authoritative database.

The federation-of-agencies pattern. Several independent organizations — say, a state agency and county governments, each with their own Enterprise — collaborate to share regional layers with each other, each remaining sovereign over its own systems.

> **Tip:** Design collaborations around *groups you create specifically for sharing*, never around existing working groups. The linked group is a conveyor belt: anything placed in it ships. A dedicated, deliberately named group ("Collab — Public Nightly Sync") makes accidental sharing much harder.

The strategic point of the hybrid pattern is that it dissolves the false either/or. The real question is rarely "Online or Enterprise?" but "which workloads belong on which side of the firewall, and what flows between them?"

The Administration Burden: An Honest Accounting

Every capability described above is real. So is the cost, and glossing over it is how organizations end up with a limping, unpatched portal that everyone resents. Here is what you actually take on when you run Enterprise — the work Esri does invisibly for you when you use Online.

Installation and architecture design. Someone must size the machines, design the topology, install four-plus software components in the right order, configure certificates, and federate the pieces. Done carefully by someone experienced, a base deployment is days of work; a highly available multi-machine deployment is a genuine project, often involving Esri Professional Services or a partner.

Certificates and web security. Enterprise runs on encrypted connections, which means TLS certificates — the digital credentials that put the padlock in the browser's address bar — on every tier. Certificates expire on their own schedule, and an expired certificate can take the whole system down for every user at once. Certificate renewal is the single most common self-inflicted Enterprise outage, and it is entirely preventable with a calendar reminder and a documented procedure.

Patching and upgrading. Esri releases Enterprise versions on a regular cadence, plus security patches between releases. On Online, you wake up to new features. On Enterprise, *you* are the one who schedules the maintenance window, snapshots the machines, runs the upgrades in the right order across every component, and tests afterward. Organizations that defer this drift years behind, then face a risky multi-hop upgrade — while running with known, published security vulnerabilities in the meantime. Budget real, recurring time for it.

Backup and recovery. Esri ships a command-line utility (`webgisdr`) that exports the state of the whole web GIS — portal items, hosted data, configuration — for disaster recovery. It only helps if someone schedules it, stores the exports somewhere safe, and *tests a restore* before the day it matters. Your enterprise geodatabases are backed up separately by your database team, on their own regimen.

Monitoring and troubleshooting. Services crash, machines fill their disks, and logs grow. Someone must notice before your users do. Esri offers a monitoring product for exactly this, and generic IT monitoring helps, but tooling only assists — it does not replace a responsible human.

People. Summing up: a small, stable base deployment might consume a meaningful fraction of one competent person's time, provided that person also has general IT support behind them for networking, virtual machines, and databases. A large, highly available, multi-role deployment is one or more full-time jobs. The most reliable failure mode in the entire Enterprise world is buying the software without staffing the role — the platform then decays until a crisis forces the investment that should have been made up front.

> **Watch out:** "Our IT department will handle it" is only a plan if IT has explicitly agreed, has capacity, and understands that Enterprise is a specialized, multi-component production system — not just another web server. Get the commitment in writing, with named people, before the purchase.

Against all that, the return: complete control over your data's location, live integration with your corporate databases, service to networks the cloud cannot reach, freedom from the credit meter for your own compute, and the ability to change nothing until

you are ready. For the organizations that need those things, the burden is simply the price of admission, and it is worth paying.

So: Do You Need It?

Pull the threads together into a working decision.

You very likely need Enterprise if any of these are true: a law, regulation, or binding policy forbids your data from living in a third-party cloud; your GIS must read and write databases inside your network as the single source of truth; your users work on networks with no internet access; or your organization requires absolute control over upgrade timing for validated or mission-critical systems. Any one of these is sufficient by itself, and the first and third are non-negotiable — no amount of ArcGIS Online configuration works around them.

You very likely do not need Enterprise if: your data is not legally constrained, your authoritative data can live happily as hosted layers, your team is small, and nobody is signed up to administer servers. ArcGIS Online with well-configured sharing, groups, and roles (Chapter 34) is more capable — and more private — than newcomers assume. "Shared to the organization" content on Online is not public; it is visible only to your signed-in members.

Weak reasons that should not decide it alone: vague unease about "the cloud" without a policy behind it; the expectation that Enterprise is cheaper because credits go away (total it honestly — hardware, licensing, and staff time against your actual credit spend); or the sense that serious organizations self-host (many large governments and enterprises run enormous workloads on Online).

And remember the middle path. A modest Enterprise deployment for the sensitive, database-connected core, collaborated with an Online organization for everything public-facing, is not a compromise — for organizations past a certain size, it is the pattern that fits reality best. Start by listing your actual constraints (legal, network, database), let those pick the platform for each workload, and let distributed collaboration stitch the halves together.

Whichever way you land, everything you have learned in this book remains yours. The maps, the layers, the Arcade expressions, the apps, the analysis — they behave the same

on either side of the firewall. Enterprise is not a different GIS. It is the same GIS, in your house, with the keys — and the chores — that come with ownership.
