The ArcGIS Compendium — Volume B: ArcGIS Online Mastery

ON THIS PAGE

The ArcGIS Compendium — Volume B: ArcGIS Online Mastery

Map Viewer end to end — every panel and tool, every smart-mapping style, the Arcade language, pop-ups and labels, and the full lifecycle of hosted layers.

One of eight volumes. Approximately 22776 words. Chapters cross-reference the whole set by number.

In this volume:

Map Viewer: The Complete Reference

Map Viewer is the web application inside ArcGIS Online — and inside ArcGIS Enterprise's portal, if your organization runs its own (Chapter 35) — where you assemble, style, and share interactive maps. If you have built one simple map before, you have already touched it: you added a layer, maybe changed a color, and clicked Save. This chapter walks the whole machine systematically — every panel, what it actually does, when to reach for it, and (just as important) what happens behind the scenes when you click Save. By the end, the interface should feel less like a collection of buttons and more like a small, coherent system with two halves: one side for *what is in the map*, the other for *how the selected thing behaves*.

A quick note on names: Esri has shipped two generations of this tool. The older one is now called **Map Viewer Classic**, and the modern one — the subject of this chapter — is simply **Map Viewer**. If your screen shows a single toolbar across the top and panels that look dated, you are in Classic; look for a link or setting to switch to the new Map Viewer. Everything below describes the modern application.

The Lay of the Land

Open a map and you will see three regions:

- **The canvas** in the middle — the map itself, which you pan, zoom, and click.
- **The left toolbar**, labeled **Contents** — this is the *inventory* side. Layers, basemap, tables, legend, bookmarks, saving, sharing, printing. Everything here answers the question "what is in this map, and what do I do with the map as a whole?"
- **The right toolbar**, labeled **Settings** — this is the *configuration* side. It is contextual: most of its panels operate on whichever layer you currently have selected in the Layers panel. Styles, filters, pop-ups, labels, editing, analysis. Everything here answers "how should this particular layer look and behave?"

That left/right split is the single most useful mental model for Map Viewer. When you are lost, ask yourself: am I trying to change the map's contents (go left) or a layer's behavior (select the layer, go right)?

One more orientation point. The right toolbar's panels gray out or change depending on what is selected. If a tool seems missing, the usual culprit is that no layer is selected, or the selected layer does not support that tool (you cannot label a plain image tile layer, for instance, because it has no attributes to label with — see Chapter 1, How GIS Thinks, for why vector layers carry attributes and most raster tile layers do not).

> **Tip:** Panels in both toolbars open and close as you click their icons, and you can keep one panel open on each side at once. A productive rhythm is Layers open on the left, and whichever settings panel you are working with open on the right.

A Web Map Is a Recipe, Not a Casserole

Before touring the buttons, you need one concept that explains almost every surprise Map Viewer will ever throw at you: **a web map does not contain data**.

When you save a map, ArcGIS Online stores a small text document (in a format called JSON — plain structured text that software can read) describing your map: which basemap to use, which layers to draw, in what order, with what colors, filters, pop-up configurations, and starting extent (the region of the world the map shows when it opens). The layers themselves are separate **items** — independent pieces of content in your organization, each with its own item page, owner, and sharing settings. The map merely *points* at them.

Think of the map as a recipe card and the layers as ingredients in the pantry. The recipe says "use the parcels layer, draw it orange, filter to 2024." It does not copy the parcels into itself.

This has consequences you will feel constantly:

- **One layer, many maps.** The same hosted feature layer (a layer whose data lives in ArcGIS Online — Chapter 10 covers these in depth) can appear in dozens of maps, styled differently in each. Fix a data error once and every map shows the fix immediately, because they all read from the same source.
- **Deleting a layer breaks every map that references it.** The maps do not get a copy; they get a dangling pointer, and they will open with an error about a layer that cannot be accessed.

- **Sharing a map does not share its layers.** If you share a map with the public but its layers remain private, other people see an empty or partly empty map. Map Viewer usually detects this when you share and offers to update the layers' sharing to match — say yes unless you have a specific reason not to.
- **Styling lives in the map by default.** Change a layer's colors inside a map and you have edited the recipe, not the ingredient. Open the same layer in a different map and it shows its own default look. (You *can* push a style back to the layer itself — more on that in the saving section.)

Watch out: Because maps reference layers rather than contain them, tidying up your content by deleting "old layers" is the most common way people destroy working maps. ArcGIS Online does not reliably show you which maps depend on a layer, so before deleting any layer item, turn on delete protection in the layer's item page settings while you investigate — and treat any layer older than your memory of it as load-bearing. Chapter 34 (Administration) covers content management strategies that prevent this.

The Left Toolbar: Contents

Here is the left toolbar at a glance, then each panel in detail.

PANEL	WHAT IT DOES	REACH FOR IT WHEN
Layers	Lists the map's layers; add, remove, reorder, toggle	Building the map's contents; selecting a layer to configure
Basemap	Swaps or customizes the background map	The backdrop fights your data, or you need imagery
Tables	Shows attribute tables; adds standalone tables	You want to see data as rows, sort, or select
Legend	Decodes the symbols for viewers	Checking what your map communicates

PANEL	WHAT IT DOES	REACH FOR IT WHEN
Bookmarks	Saved camera positions	You keep navigating to the same places
Save and open	Save, Save As, open other maps	Always — early and often
Map properties	Map-wide settings	Adjusting behavior that belongs to the whole map
Share	Sets who can see the map	Publishing, or diagnosing "my colleague sees nothing"
Print	Produces a printable page from the current view	Someone needs paper or a static image

Layers

The Layers panel is home base. It lists every operational layer in the map — "operational" meaning your data, as opposed to the basemap underneath.

Order matters: the list is the **drawing order**, with the top of the list drawn on top of the map. Drag entries to reorder. The standard arrangement puts points above lines above polygons, so small things are not buried under big filled shapes (Chapter 4, Cartographic Design, explains the visual logic).

Each layer row gives you a visibility toggle (the eye icon) and an options menu (usually three dots) with actions like zoom to the layer's extent, rename it *within this map*, duplicate it, group it with others, show its attribute table, and remove it. Removing a layer takes it out of the recipe; it never deletes the underlying item.

Adding layers happens here too. Look for the **Add** button at the bottom of the panel. The main routes:

- `Layers > Add > Browse layers` — search your own content, your organization, your groups, or the Living Atlas (Esri's curated collection of ready-to-use layers; Chapter 5 covers how to evaluate what you find there).

- Adding a **layer from URL** — paste the web address of a GIS service, useful for layers published by other organizations. Chapter 32 (REST Services) explains what those URLs actually are.
- Adding a **sketch layer** — quick freehand points, lines, shapes, and text stored inside the map itself. Sketches are the one exception to "maps contain no data," and they are fine for annotations but wrong for anything you will analyze or reuse. Chapter 11 (Creating Data) covers when a sketch should graduate into a real layer.
- Uploading a **file** — spreadsheets, CSVs, GeoJSON and similar can often be dragged in or added here; depending on choices you make, they either become a real hosted layer or live only in this map.

Selecting a layer in this panel is what arms the right toolbar. Nearly everything in the second half of this chapter starts with "select the layer first."

Basemap

The basemap is the reference backdrop — streets, imagery, terrain, or a deliberately quiet gray canvas. The Basemap panel shows a gallery chosen by your organization's administrator; click one to swap. The choice is cartographically loud: a light gray basemap makes your data pop, imagery grounds field data in reality, and a dark basemap flatters bright, glowing symbols. Chapter 4 goes deep on choosing.

The panel also lets you customize the current basemap — you can typically move an operational layer into the basemap so it behaves as part of the backdrop, or swap the individual layers that compose the basemap (many modern basemaps separate the base drawing from a reference layer of place labels, so you can put labels *above* your data). If your organization allows it, you can save a customized basemap for reuse.

Tables

Every feature layer is secretly a table wearing geometry (Chapter 1). The Tables experience shows that table docked along the bottom of the map: one row per feature, one column per field. Open it from the Tables panel or from a layer's options menu.

Use it to sort, to scan for data problems, to hide or rearrange columns, and to **select** rows — selected rows highlight on the map, and selecting features on the map highlights rows, which is the fastest way to answer "which dot is this record?"

The Tables panel also lets you add a **standalone table**: a table with no geometry at all, such as an inspection log related to facility points. These become genuinely useful once you understand relationships between tables and layers, which is Chapter 12's territory (Schema Design).

Legend

The Legend panel renders the decoder ring: each visible layer with its symbols and what they mean. It is read-only — you change the symbols in Styles, not here — but it is the honest mirror of your map. If the legend is confusing, the map is confusing.

Only currently visible layers appear, and layers can be excluded from the legend individually (look for a "hide in legend" option in the layer's settings) — handy for backdrop layers that viewers do not need explained.

Bookmarks

A bookmark is a saved camera position: a location and zoom level with a name. Navigate somewhere, open Bookmarks, add one, and you can snap back anytime. They are stored in the map and travel with it, which makes them quietly powerful downstream: several app builders (Chapters 26 and 29) can present your map's bookmarks as a navigation menu for end users. If your map serves three field offices, make three bookmarks.

Save and Open

This panel holds **Save**, **Save As**, and options for opening or creating maps. The semantics deserve their own section — see "Saving: What Goes Where" below — but the headline is:

- **Save** overwrites the current map item.
- **Save As** creates a brand-new map item and leaves the original untouched. The new map references the *same* layers — Save As copies the recipe, never the ingredients.

> **Watch out:** Map Viewer does not autosave. The browser tab crashing, your session expiring over lunch, or an accidental navigation click can discard an hour of unsaved styling. Save early, save often, and watch for the unsaved-changes indicator near the save control.

Map Properties

Map properties are the settings that belong to the whole map rather than any layer — for example, the background color shown behind (and through) the basemap, and the time zone used when displaying dates. It is a short panel; skim it once so you know what lives there. The map's *item-level* details — its title, summary, thumbnail, and description — belong to the map's item in the content system; you can reach the item page from the map or from your content list. A well-documented item page is what makes your map findable and trustworthy six months from now; Chapter 5's advice on evaluating other people's data applies equally to documenting your own.

Share

Share controls who can open the map: just you (private), specific groups, your whole organization, or everyone (public). This is the same sharing model every ArcGIS item uses, and Chapter 34 covers it fully — including groups, which are the right tool for almost every real collaboration.

Remember the recipe principle: the sharing dialog will check whether the map's layers are shared at least as broadly as the map, and offer to fix mismatches. A publicly shared map with private layers is the most common "it works for me but not for anyone else" report in all of ArcGIS Online (Chapter 39, the Troubleshooting Encyclopedia, opens with it).

Print

Print generates a static, printable version of the current view — typically a PDF or image, with a choice of page layouts and a title. It uses a printing service to compose a proper page at print resolution rather than screenshotting your screen.

Be clear about what this is and is not. Print is for quick, serviceable output: a page for a meeting, an attachment for a report. It is not a cartographic layout engine — no fine control over margins, multiple map frames, or publication typography. When the output matters, build a layout in ArcGIS Pro (Chapter 24, Layouts, Map Series, and Print Cartography).

The Right Toolbar: Settings

Select a layer in the Layers panel, then work down this toolbar. At a glance:

PANEL	WHAT IT DOES	REACH FOR IT WHEN
Properties	Layer basics: transparency, visible range, blending	The layer shows at wrong zooms, or overpowers the map
Styles	Symbology and smart mapping	Deciding how the data draws
Filter	Shows only features matching conditions	The layer has more than the story needs
Effects	Visual treatments like drop shadow and bloom	Emphasis and polish, sparingly
Aggregation	Clustering and binning of dense points	Thousands of overlapping dots
Pop-ups	What appears when a feature is clicked	Always — defaults are never good enough
Fields	Display names and number formatting	Field names look like machine code
Charts	Bar charts, scatter plots, and more from the layer's attributes	The numbers behind the map need to sit beside it
Labels	Text drawn on features	Names or values should be visible without clicking
Edit	Modify the layer's actual data	Correcting or capturing features
Analysis	Spatial analysis tools	Asking questions the map alone cannot answer

Properties

The Properties panel gathers a selected layer's fundamentals:

- **Visibility range** — the zoom span within which the layer draws. If your layer "disappears" when you zoom out, the visible range is set (sometimes inherited from

the layer's publisher). Widen it here — or respect it, since ranges often exist because drawing ten thousand parcels at state scale is both slow and unreadable.

- **Transparency** — a slider making the whole layer see-through, essential for overlaying polygons on imagery.
- **Blending** — blend modes control how a layer's colors mathematically combine with layers beneath it, like blend modes in photo editors. Multiply is the workhorse: it lets a colored layer soak into terrain shading below rather than pasting over it. Chapter 7 shows blending in anger.
- Layer information and links to the layer's item page, which is your jump point to everything covered in Chapter 10.

Styles

Styles is where you choose what the data looks like: which attribute (if any) drives the drawing, which drawing style renders it (single symbol, color by category, size or color by number, heat map, and more), and the detailed options within that style. The panel is powered by **smart mapping** — Esri's system that inspects your data and proposes sensible defaults for colors, numeric break points, and symbol size ranges.

This chapter deliberately stops here, because Chapter 7 (Styling and Smart Mapping, Complete) is the full treatment. For now, know the shape of the workflow: pick attributes, pick a style, open style options, refine. And know that everything you do in Styles is saved in the *map* unless you explicitly save it to the layer.

Filter

A filter shows only the features that meet conditions you define — "status is Active," "year is after 2020," "inspection score below 60." You build conditions from the layer's fields, combine them with and/or logic, and the map redraws instantly.

Two things to internalize. First, **filters hide, they never delete** — the data is untouched, and removing the filter brings everything back. Second, a filter set here lives in this map only. If you need a filtered *layer* — say, to share only active records with the public while keeping the full dataset private — that is a hosted feature layer **view**, a fundamentally different and more powerful tool covered in Chapter 10.

Tip: Filters are the fastest data-exploration tool in Map Viewer. Before committing to an analysis workflow, spend five minutes filtering on different fields and watching the map. You will often find the pattern — or the data quality problem — immediately.

Effects

Effects apply visual treatments to a layer: drop shadow, bloom (a glow that flatters bright symbols on dark basemaps), blur, grayscale, and adjustments like brightness and saturation. More interestingly, effects can be **conditional**: features meeting a condition draw one way while everything else draws another — for example, overdue inspections in full color with a drop shadow while the rest fade to gray. That single pattern, vivid-versus-muted, is one of the strongest emphasis tools in web cartography.

Use effects like hot sauce. One deliberate effect focuses a map; three make it look like a video game.

Aggregation

When a point layer has thousands of overlapping features, individual dots stop communicating. The Aggregation panel offers two remedies:

- **Clustering** groups nearby points into a single symbol sized or labeled by how many it contains. Zoom in and clusters break apart into their members. Cluster pop-ups can summarize their contents — count, sums, averages.
- **Binning** divides the map into a regular grid of cells and draws each cell colored by a statistic of the points inside it. It reads more like a density surface and stays stable as users pan. Binning requires support from the underlying layer, so it is not available on every layer.

Aggregation is display-time summarization — the underlying features are untouched, and turning it off restores the dots. For statistical rigor about density and hot spots, rather than visual relief, see Chapter 17.

Pop-ups

The pop-up is what appears when someone clicks a feature, and it is where a map earns or loses trust. The Pop-ups panel lets you enable or disable pop-ups per layer, set the

title (usually driven by a field), and stack content blocks: a formatted list of fields, free text with field values woven in, images, charts, and values computed on the fly with **Arcade**, Esri's expression language (Chapter 8 takes you from zero to fluent).

The default pop-up — a raw dump of every field with database-style names — is never the right answer. Even ten minutes of curation transforms a map. The complete craft, including attribute expressions and content ordering, is Chapter 9's job; the thing to remember here is simply where the panel lives and that pop-up configuration, like styling, is part of the map's recipe by default.

Fields

The Fields panel controls how a layer's fields *display*: friendly display names (aliases) instead of `POP2020_EST`, and formatting for numbers and dates — thousands separators, decimal places, date styles. These settings flow through everywhere the field appears: pop-ups, labels, tables, charts.

Understand the boundary: Fields changes presentation, not schema. It cannot add a field, change a field's type, or touch stored values — that is done on the layer itself, and the design thinking behind it is Chapter 12. If a number displays with absurd precision ("34.29481662 acres"), fix it here once rather than in every pop-up and label separately.

Charts

Map Viewer can also build charts from a selected layer's attributes — bar charts, scatter plots, histograms, and similar — displayed alongside the map. Charts are saved as part of the map, and they are interactive in both directions: select bars or points in the chart and the matching features highlight on the map, and vice versa. For quick "show me the distribution" questions this beats exporting data to a spreadsheet; for rigorous analysis, Volume D still applies.

Labels

Labels draw text on or near features — parcel numbers on parcels, names on trailheads. The Labels panel lets you switch labeling on per layer, choose the field (or write an Arcade expression) supplying the text, style the font and halo, set placement, and restrict the zoom range where labels appear. You can define multiple **label classes**, each with its own filter and style, so major roads label large and minor roads small.

Web labeling is automatic: an engine places what fits and quietly drops what does not, so do not expect every feature to be labeled at every scale. Chapter 9 covers the craft in the web; Chapter 23 covers the far more controllable labeling engine in ArcGIS Pro.

Edit

The Edit panel opens the editor for layers that allow editing: create new features by clicking or sketching geometry, select existing ones to move, reshape, or delete, and update their attribute values through a form (the form itself is configurable — you choose which fields appear, in what order, with what logic — a topic Chapter 13 develops). Snapping options help geometry connect precisely. Whether a layer is editable at all — and *what kinds* of edits are allowed — is governed by settings on the hosted layer itself (Chapter 10) and by views that expose safe editing slices of the data.

Everything you change here changes the actual data, immediately, for every map and app that references the layer. There is no draft state and no save-to-commit step for edits in the web. Editing workflows, forms, and the discipline that keeps shared data safe are Chapter 13's subject.

> **Watch out:** Edits are live and shared the moment you make them. Styling mistakes are private until you save the map; editing mistakes are public instantly. Before experimenting with the Edit panel, work against a copy of the data or an editable view scoped for practice — Chapter 13 shows how to set that up.

Analysis

The Analysis panel exposes spatial analysis tools directly in the browser: create buffers, overlay layers, join by location, find hot spots, calculate drive times, and more. Each tool takes layers as input, runs on Esri's servers, and typically writes its result as a **new hosted feature layer** in your content — a real, reusable item, not a temporary graphic. A history view tracks what you have run.

Two practical notes. First, most analysis tools consume **credits**, ArcGIS Online's usage currency; each tool estimates its cost before you run it, and expensive inputs (very large layers, drive-time calculations) cost more. Chapter 34 explains credits and how administrators budget them. Second, the tools themselves — what they compute and

when each is the right question — occupy all of Volume D. This panel is merely the doorway.

Navigating the Map

Fluent navigation is worth the five minutes it takes to learn, especially the keyboard, which most people never discover.

With the mouse: click-drag pans; the scroll wheel zooms in and out centered on your pointer; double-click zooms in a step; holding Shift while dragging draws a box to zoom into; and right-click-drag rotates the map. If you rotate — deliberately or by accident — a compass control appears on the map; click it to snap back to north-up. (An unexpectedly rotated map is a classic source of quiet panic. It is always the compass.)

With the keyboard, once the map has focus (click it first): the arrow keys pan, plus and minus zoom, and additional keys handle rotation and resetting north. Map Viewer publishes its current shortcut list in its help — worth a skim, since exact bindings evolve. Keyboard navigation matters beyond convenience: it is how users who cannot operate a mouse experience your map, which makes it an accessibility obligation for anything public-facing.

The **search box** geocodes — turns typed text into a map location: enter an address or place name and the map flies there, dropping a temporary pin. Search can also be configured to look inside your layers' attributes, so users can find "Parcel 042-118" as easily as a street address — look for search settings among the map's or app's configuration.

On touch screens, drag pans, pinch zooms, and two-finger twist rotates.

Saving: What Goes Where

You now know enough to make the saving rules precise. There are three distinct places a change can land, and every save-related mystery in Map Viewer dissolves once you name which one you are looking at.

The map item. Save writes the recipe: layer list, order, styling done in this map, filters, pop-ups, labels, effects, bookmarks, extent, basemap choice. Save As writes a new

recipe card and leaves the old one alone. Nobody else's maps are affected, because your recipe is yours.

The layer item. Some changes can be pushed from the map down into the layer itself, so the layer *defaults* to your new configuration everywhere it is used from then on. After styling a layer, look in the layer's options for a save-to-layer choice. Do this when you own the layer and the new style genuinely is its best default. Do not do it casually to shared layers — you are changing the starting point for every future map and every colleague who adds that layer.

The data. Edits made through the Edit panel — geometry and attribute changes — go straight into the layer's data. They do not wait for a map save; they are simply done, everywhere, immediately.

One subtlety completes the picture: when a map's recipe includes styling for a layer, the map's styling **overrides** the layer's default. If a colleague later improves the layer's default symbology, your map keeps showing your saved override. That is a feature — maps are stable — but it explains the occasional "I updated the layer, why does the old map look the same?"

> **Tip:** When a map matters — it feeds a dashboard, a StoryMap, or the public — treat Save As as your undo insurance. Before a big rework, Save As a dated copy. Maps are tiny (they are just recipes), so copies cost effectively nothing, and Chapter 36's worked project shows how this habit fits a real production workflow.

A Working Rhythm

Everything above compresses into a sequence you will run hundreds of times. Add layers and order them (left: Layers). Choose a backdrop that serves the data (left: Basemap). Select each layer and work down the right side: Properties for visibility range and transparency, Styles for the drawing, Filter to trim to the story, Fields to civilize the names and numbers, Pop-ups and Labels to make features speak, Aggregation or Effects if density or emphasis demands them. Set bookmarks for the places that matter. Save. Check the Legend as a stranger would. Share — letting Map Viewer fix layer sharing — and only then build the app or dashboard on top.

Map Viewer is the hinge of the whole ArcGIS Online experience: every StoryMap, Dashboard, Instant App, and Experience Builder app in Volume F begins life as a web map configured exactly the way this chapter describes — and Map Viewer's own Create app option will hand your finished map straight to those builders. Time invested in the right side of this screen pays out in every product built downstream.

Styling and Smart Mapping, Complete

A layer's data decides what your map *knows*. Its style decides what your map *says*. Two people can publish the same table of census tracts and produce maps that make opposite impressions — one a calm reference map, the other a screaming alarm — purely through styling choices. This chapter is the complete tour of how styling works in ArcGIS Online's Map Viewer: every smart-mapping style family, every option buried in the style panels, plus effects, blend modes, and the strategy of styling across zoom levels. For the design principles behind these tools — color theory, visual hierarchy, when a map is honest — see Chapter 4 (Cartographic Design). This chapter is about the machinery.

How Smart Mapping Thinks

Smart mapping is Esri's name for the guided styling system built into Map Viewer. The idea: instead of handing you a blank symbol editor and wishing you luck, the software looks at your data — the geometry type, the fields you pick, the statistical spread of the values — and offers a short list of styles that actually fit, each with sensible defaults already computed.

To reach it, select a layer in your map, then open `Styles` from the settings toolbar (the panel of icons on the right side of Map Viewer). The Styles pane asks you two questions, in order:

1. **Choose attributes.** Which field (or fields) should drive the styling? An *attribute* is just a column in the layer's table — population, land use category, inspection date. You can pick none (style by location only), one, or several. You can also click `+`

`Expression` here to drive the style with an Arcade expression instead of a raw field — Arcade is ArcGIS's small scripting language, and Chapter 8 (Arcade) takes it from zero to fluent. For now, know that anything a field can do, an expression can do too, including combining fields ("injuries plus fatalities") or bucketing values on the fly.

2. **Pick a style.** Based on what you chose, Map Viewer lists the style families that apply, with one marked as the suggested default. Each has a `Style options` button that opens the deep settings.

The defaults are genuinely good — smart mapping reads your data's distribution and places its color and size breakpoints at statistically meaningful positions rather than arbitrary round numbers. But defaults are a starting bid, not a finished map. The rest of this chapter is about knowing what every knob does so you can negotiate.

> **Tip:** Styling is saved in the *web map*, not in the layer itself, unless you deliberately push it down. When you restyle a layer inside a map and save the map, only that map changes. If you want the new look to become the layer's default everywhere it's used, look for the option to save the layer's properties back to the layer item (on the layer's options menu or its item page). Chapter 10 (Hosted Feature Layers) covers why one layer often feeds many maps.

Choosing a Style Family

Every style in Map Viewer belongs to a family, and the right family follows directly from the question your map answers:

YOU WANT TO SHOW...	FAMILY	WORKS WITH
Where things are, all drawn alike	Location (single symbol)	Points, lines, polygons
What kind of thing each feature is	Types (unique symbols)	Points, lines, polygons
How much, using color	Counts and Amounts (color)	Points, lines, polygons

YOU WANT TO SHOW...	FAMILY	WORKS WITH
How much, using size	Counts and Amounts (size)	Points, lines, polygons
Two "how much" questions at once	Color and Size	Points, lines, polygons
How two variables relate	Relationship (bivariate color)	Mostly polygons
Quantities spread inside areas	Dot Density	Polygons only
Density of many points	Heat Map	Points only
Many points, summarized on the fly	Clustering / Binning (aggregation)	Points only

The sections below take each in depth.

Location: The Honest Default

Location styling draws every feature with the same symbol. No attribute drives anything. It sounds trivial, but it is the correct choice more often than people admit: when the message is simply "here is where our sites are," a single well-chosen symbol beats a rainbow.

The craft is in the symbol itself. In `Style options` you set the shape, fill and outline colors, size (for points and line width), and overall transparency. Two habits pay off. First, size points for the density of your data — thirty points can wear a bold 12-pixel marker; thirty thousand need something small and translucent or they become a smear (or a candidate for clustering, covered below). Second, mind the outline: a thin white or dark outline is what keeps point symbols legible over busy basemaps.

Types: Categories in Color and Shape

Types styling (Esri also calls it *unique values*) assigns one symbol per category in a field — zoning class, pipe material, inspection result. Map Viewer lists every distinct value it finds, most common first, and assigns each a color from a categorical palette.

Everything worth knowing lives in `Style options`:

- **Reorder and group values.** Drag values to control legend order. You can group several raw values under one label — useful when the data distinguishes "Vacant-Residential" from "Vacant-Commercial" but your story only needs "Vacant."
- **The Others bucket.** Categorical color works up to roughly ten distinguishable hues; beyond that, colors recycle and the legend turns to noise. Map Viewer sweeps low-frequency values into an "Others" class. Resist the urge to un-collapse it. If you truly have dozens of meaningful categories, the map probably needs filtering or multiple maps, not more colors.
- **Per-value symbols.** Click any value's swatch to change its color, shape, or size individually — including swapping in a custom icon (see Symbol Libraries below).

> **Watch out:** Types styling reads the *stored* values, not your intentions. If the field contains "Residential", "residential", and "RES ", you get three legend entries. Fix the data (Chapter 14, Data Quality) or paper over it with an Arcade expression that normalizes the text — but know that the paper-over version silently hides a data problem.

Counts and Amounts (Color): The Choropleth Engine

This family shades features by a numeric value — the classic *choropleth* map when applied to polygons (a choropleth is any map where areas are shaded by a statistic). It is probably the most-used thematic style in existence, and the one with the most levers.

Open `Style options` and you'll meet the centerpiece of smart mapping: a **histogram** of your data with draggable **handles** on a color ramp beside it. The histogram shows how your values are distributed — where they bunch, where the outliers sit. The handles set where the ramp's colors are anchored. Smart mapping places them initially around the data's average and spread, which is why the default map usually looks reasonable. Drag a handle and the map recolors live; this direct manipulation is the fastest way to explore what your data is hiding. When outliers squash the useful part of the histogram into a sliver, look for the histogram's zoom control to spread out the crowded range — or type exact handle values instead of dragging.

Above the histogram sits the **theme** selector, which controls the *shape* of the color logic:

THEME	WHAT IT EMPHASIZES	TYPICAL USE
High to low	A smooth sweep from lowest to highest	General "how much" maps
Above and below	Divergence from a meaningful midpoint	Change over time, above/below average, profit vs. loss
Above / Below	Only one side of a midpoint	"Show me only where it got worse"
Extremes	The tails; the middle fades out	Outlier hunting

"Above and below" deserves special attention: it uses a *diverging* ramp (two hues meeting at a neutral center) and is the honest choice whenever your data has a true midpoint — zero change, the national average, a regulatory threshold. Set the center handle to that meaningful value, not just the mean.

Two more controls in this pane matter enormously:

- **Divided by (normalization).** A count map of raw totals mostly shows where people live — big places have big numbers of everything. Dividing by a second field (population, area, households) converts counts to rates and makes places comparable. The style pane offers this directly as `Divided by`.
- **Classify data.** By default, modern Map Viewer uses an *unclassed* (continuous) ramp — every value gets its own point on the gradient. Toggle `Classify data` to switch to discrete classes, then choose a method: **natural breaks** (finds gaps in the data — good default), **equal interval** (uniform-width bins — good for well-known scales), **quantile** (equal feature counts per bin — always looks dramatic, sometimes dishonestly so), **standard deviation** (distance from the mean), or **manual** breaks you type yourself. Classed maps are easier to read precisely ("this tract is in the 20–

40% band"); unclassed maps show texture and avoid the arbitrariness of bin edges. Chapter 4 (Cartographic Design) argues the trade-off in full.

> **Tip:** When mapping anything per-capita, per-household, or per-area, set `Divided by` before you touch a single handle. Every other adjustment depends on whether you're mapping counts or rates, and switching later resets your careful breakpoints.

Counts and Amounts (Size): Graduated Symbols

Same idea, different visual channel: the numeric value drives symbol size instead of color. Points grow into *proportional symbols*; lines thicken; polygons get a sized marker at their center.

The style options mirror the color version — histogram, handles, theme — but the handles now map values to a size range (minimum and maximum symbol size in pixels, or line widths). Size is the better channel than color when the audience must compare magnitudes ("this city is roughly triple that one") because human eyes judge relative size better than relative shade. It's the worse channel when features are dense, because big symbols overlap. Practical remedies: lower the maximum size, add transparency so overlaps read as darker stains, or let clustering take over at small scales (see below).

For polygons, prefer size symbols over choropleth shading when mapping raw counts — a sized circle over each county says "quantity" without the misleading effect where geographically huge, sparsely populated areas visually dominate.

Color and Size Together

Pick two numeric attributes and Map Viewer offers **Color and Size**: one variable drives symbol color, the other drives symbol size. Each gets its own full options panel — themes, handles, normalization — exactly as described above.

The classic, and most defensible, use is *rate plus magnitude*: color shows the rate (unemployment percentage), size shows the base (labor force). The reader sees at a glance both how bad and how big. You can also assign the *same* field to both channels,

which produces a redundantly-encoded map that reads clearly even for colorblind viewers and in grayscale printouts.

Two variables per symbol is the comfortable ceiling. Three (color, size, plus rotation or transparency) is technically possible and almost always mud.

Relationship: The Bivariate Grid

The Relationship style answers "do these two variables move together?" — income and broadband access, age of housing and code violations. It blends two color ramps into a grid: features high on both variables get the strong blended corner color, low-low gets the pale corner, and the mismatched corners (high-low, low-high) get the two pure hues. The legend is itself a small grid, and reading it takes most audiences a beat — this is a style for engaged readers, not glance-and-go dashboards.

In `Style options` you choose the grid dimensions — 2×2, 3×3, or 4×4. Bigger grids give more nuance and much harder legends; 3×3 is the sweet spot for most work. You can also rotate which corner is the "focus," flip either variable's direction, and adjust each variable's breakpoints on its own histogram, just as in the color style. A `Show legend as` option can express the same classes as discrete named combinations ("High income, Low access") which some audiences find friendlier than the color grid.

Be honest about what this map shows: visual co-occurrence, not statistical correlation, and certainly not causation. If you need the statistics, that's Chapter 17 (Statistical and Pattern Analysis).

Dot Density

Dot density styling — polygons only — scatters dots inside each polygon, each dot representing a fixed quantity: one dot per 100 households, say. Where dots crowd, there's a lot; where they thin out, little. It's uniquely good at showing *internal texture* that a choropleth flattens: a county that's half empty ranch land and half dense suburb reads as exactly that.

Key options:

- **Dot value** — how much one dot represents. Smart mapping ties this to scale by default so the density impression stays stable as you zoom (the dot value falls as you

zoom in). You can pin it if you need a fixed legend statement.

- **Multiple attributes** — add several count fields, each with its own dot color, and you get the famous multi-group dot density map (one color per demographic group, all mingled in the same polygons). Keep it to a handful of high-contrast colors on a muted basemap.
- **Blending and background** — dot density lives or dies on contrast; a dark neutral basemap with luminous dots is the proven combination.

> **Watch out:** Dot density requires *counts* — whole quantities that can be divided into dots. Feeding it a rate, percentage, or average produces a map that looks plausible and means nothing. Also remember the dots are placed *randomly within each polygon*: a dot is not a household at that spot. Readers routinely over-interpret dot positions, so say so in the map's description or pop-up.

Heat Map

Heat maps take a point layer and render a smooth surface of density: blue-through-yellow-through-red blobs (or whatever ramp you choose) where blue means sparse and red means crowded. Under the hood it's a *kernel density* rendering — each point spreads a small halo of influence, and overlapping halos add up.

Options are few but consequential. **Area of influence** sets how far each point's halo spreads: small values give a speckled, precise surface; large values give broad, smoothed blobs. There is no objectively correct setting — nudge it until the pattern is visible at your map's working scale without dissolving into one big blob. **Weight by a field** makes high-value points count more (a $1M sale contributes more heat than a $100K one). The ramp's handles control which densities read as "cool" versus "hot."

Heat maps shine for large point sets at small scales — thousands of incident reports across a metro. They fail when points are few (a dozen points make misleading blobs) or when the reader needs to identify individual features — the points literally aren't drawn. Plan a scale transition (last section of this chapter) so zooming in eventually reveals real points.

Watch out: A heat map is a *picture of density*, not an analysis of significance. A red blob may just mean "more people live here, so more of everything happens here." Hot spot analysis — the statistical test that asks whether clustering exceeds what population or chance would predict — is different machinery entirely; see Chapter 17 (Statistical and Pattern Analysis). Don't present a styled heat map as a statistical finding.

Aggregation: Clustering and Binning

Everything so far restyles individual features. Aggregation *replaces* them on screen with summaries, recomputed live as you pan and zoom. In Map Viewer, select a point layer and open `Aggregation` from the settings toolbar; choose **Clustering** or **Binning**. Aggregation is display-only — the underlying data is untouched, and turning it off restores the raw points.

Clustering

Clustering gathers nearby points into a single cluster symbol whose size reflects how many points it swallowed, with a count label. Zoom in and clusters break apart into smaller clusters and eventually raw points; zoom out and they merge.

The essential settings:

- **Cluster radius** — how aggressively points merge. Larger radius, fewer and bigger clusters.
- **Size range** — the pixel sizes clusters scale between.
- **Cluster pop-ups and labels** — clusters get their *own* pop-up and label configuration, separate from the features'. By default a cluster reports its count, but you can surface summary statistics of a field — average inspection score, predominant category — through the aggregate fields the pop-up editor exposes, or with Arcade. Chapter 9 (Pop-ups, Fields, and Labels) covers the mechanics.
- **Inherited styling** — clusters respect the layer's style family where it makes sense: cluster a types-styled layer and each cluster takes the color of its *predominant* category, which is genuinely informative.

Binning

Binning divides the map into a fixed grid of cells (the grid comes from a spatial indexing scheme, so bins stay put as you pan rather than reshaping the way clusters do) and draws one polygon per cell, styled by the count — or another statistic — of the points inside. Where clustering produces organic, shifting blobs, binning produces a stable, regular grid, which makes side-by-side and over-time comparison much easier. Bins can be styled with the full Counts and Amounts machinery — ramps, themes, breakpoints — and get their own pop-ups and labels like clusters do. Binning also lets you choose a zoom threshold past which the raw features draw instead — use it; clustering handles that transition on its own as clusters dissolve into points.

Choose clustering when the audience will interact and drill down; choose binning when the audience needs to compare densities across areas or dates at a glance.

Style Options Deep Dive

The controls in this section appear across many of the families above. This is the reference for each.

Color Ramps

Every color-driven style has a `Symbol style` panel with a ramp gallery. Three things to know. First, the gallery is filtered by your basemap's tone — ramps that read well on dark basemaps are flagged, and switching basemaps re-sorts the suggestions. Second, ramps come in the three grammatical types from Chapter 4: *sequential* (light to dark, for ordered magnitudes), *diverging* (two hues from a neutral center, for above/below themes), and *categorical* (distinct hues, for types). The theme you picked constrains which type you're offered — one of smart mapping's quiet guardrails. Third, any ramp can be flipped (reverse direction), and you can override individual ramp colors if the house palette demands it.

Breakpoints and the Histogram

The histogram-with-handles appears in every numeric style, and mastering it is most of mastering smart mapping. The handles are *anchors*: values at or beyond the top handle get the ramp's strongest color (or largest size); values at or below the bottom handle get the weakest; everything between interpolates. This means outliers don't stretch your ramp thin — everything beyond a handle just saturates. Deliberately pulling the

top handle *down* is a legitimate technique to make an upper tier "max out" and let midrange variation show; just remember the legend then reads "value or higher."

Smart mapping's initial handle positions are statistical — typically bracketing the average by a spread on either side — which is why untouched defaults usually look sane. When a break must land exactly on a threshold (a regulatory limit, a zero line), type the value rather than dragging. When `Classify data` is on, the handles become class *edges* and the count of features per class is shown — watch those counts to avoid classes containing two features and classes containing two thousand.

Transparency by Value

Separate from a layer's overall transparency slider, most styles let a *second* attribute drive per-feature transparency: `Style options`, then look for the transparency-by-attribute control. Features with high values of that field draw opaque; low values fade toward invisible, along a range you set with two handles.

The canonical use is *confidence fading*: color shows the estimate, transparency shows the margin of error or sample size, so shaky data literally recedes. It's also a gentle alternative to filtering — fade the irrelevant instead of deleting it. Keep the minimum transparency above roughly the barely-visible threshold unless you truly want features to vanish, and mention the encoding in the map description, because legends express transparency poorly.

Rotation by Value

Point symbols can rotate to match a numeric field — wind direction, vehicle heading, aspect of a slope. In `Style options`, the rotation control asks for the field (or an Arcade expression) and the angle convention: **geographic** (0° is north, angles run clockwise, the convention of compasses and most sensor data) or **arithmetic** (0° is east, counterclockwise, the convention of mathematics). Picking the wrong convention produces arrows that are wrong by a consistent, maddening offset — if your arrows look systematically skewed, this toggle is the first suspect. Use a symbol with an obvious "front," like an arrow or teardrop; rotating a circle accomplishes nothing.

Symbol Libraries and Custom Symbols

Click any point symbol's swatch and a symbol browser opens with several built-in libraries: basic geometric shapes, a large set of point-of-interest icons (schools,

hospitals, transit), arrows, and themed sets. All are vector symbols — they scale cleanly and most accept your fill and outline colors.

When the built-ins won't do, use a custom image: the symbol browser accepts a web-accessible image URL (a PNG with a transparent background is the reliable choice). Practical rules: host the image somewhere permanent and HTTPS-served — if the URL dies, your map shows broken symbols; keep the source image modest in pixel dimensions and let the size slider do the scaling; and design for legibility at small sizes, meaning bold silhouettes, not detailed logos. For a whole custom symbol *set* mapped to categories, set each category's symbol individually in the Types style options. (Pro's symbol engine, with its multi-layer symbols and effects, is a different and deeper world — Chapter 23.)

Effects and Blend Modes

Styling decides how features draw; effects and blend modes decide how the drawn layer is *processed and composited* into the final image. Used sparingly, they add a level of polish that reads as professional cartography. Used enthusiastically, they read as a lens-flare phase.

Layer Effects

With a layer selected, open `Effects` from the settings toolbar. Effects apply to the layer's rendered image as a whole. The roster includes **bloom** (a soft glow around bright features), **drop shadow** (lifts features off the basemap), **blur**, **grayscale** and **sepia** (drain color), **saturation**, **brightness and contrast**, **hue rotate**, and **invert**. Each has small sliders for intensity.

Some effects can also be applied *conditionally* — one effect for features matching a filter, another for the rest — which turns effects into an emphasis tool: features meeting a condition get a drop shadow, everything else gets grayscale and a touch of blur. That single pattern (sharpen the subject, mute the context) is the highest-value use of the entire effects system.

Blend Modes

A blend mode changes how a layer's pixels combine with the layers *below it* in the drawing order — the same concept as blend modes in photo editors. Find it with the

layer selected in the `Properties` pane, where the blending control sits alongside transparency. Normal mode simply paints over. The useful alternatives:

- **Multiply** darkens: the layer's colors combine with what's beneath, so whites become transparent-ish and shading accumulates. This is the workhorse.
- **Screen / Lighten** brighten instead — useful for glowing features on dark basemaps.
- **Overlay / Soft light** boost contrast between layer and basemap in both directions.
- **Destination-in and its siblings** are masking modes: the layer acts as a stencil that reveals or hides what's below rather than painting anything itself.
- **Luminosity / Color** transfer only brightness or only hue between layers — niche, occasionally magic for imagery work.

A blend mode affects everything below the layer, so drawing order (drag layers in the Layers pane) is part of the recipe.

Three Tasteful Recipes

Terrain-aware choropleth. Put your polygon choropleth above a hillshade basemap (a hillshade is a grayscale shaded-relief rendering of terrain) or an imagery basemap, and set the polygon layer's blend mode to multiply. The terrain's shading shows through the thematic colors, giving a hand-shaded-atlas feel for free. Nudge layer transparency if the result runs dark.

The spotlight. Draw a polygon of your study area in a layer above the basemap and set its blend mode to destination-in. The basemap now renders *only inside* the polygon; everything outside disappears. Add a muted solid-color layer underneath as the "outside" backdrop. This is the cleanest way to say "only this county matters here."

Firefly points. On a very dark basemap, style points small and vividly colored, then add a bloom effect. Dense point data becomes a glowing constellation — genuinely striking for citywide incident or activity data. The failure mode is using it when a plain map would communicate better; fireflies are for density-as-drama, not for maps people must read precisely.

> **Tip:** Effects and blend modes are rendered live by the viewer's browser and carry into most, but not every, downstream context — some export paths, older apps, or

print flows may ignore them. Before you build a workflow around a blend-mode look, test it in the actual app or export format you'll deliver (Chapter 26 for Instant Apps, Chapter 24 for print).

Scale-Dependent Styling Strategy

A web map is not one map; it's a stack of maps across zoom levels wearing the same name. A style tuned for the statewide view is usually wrong for the street view, and vice versa. Map Viewer gives you three tools to manage this, and good maps use all three deliberately.

Visible range. Every layer has a visibility slider (`Properties > Visibility` , with the layer selected) setting the scale range in which it draws at all. This is the bluntest tool: parcels off until zoomed in, regional summary polygons off once you're close.

Scale-aware styles. Several styles adapt to scale on their own: dot density adjusts its dot value, heat maps recompute density for the current view, clustering re-clusters continuously. Others adjust symbol sizes automatically as you zoom — look for the size-by-scale controls in the size style options if you want to tune or override the curve. Labels get their own visible ranges too (Chapter 9).

The layer-swap pattern. The most powerful technique costs nothing but forethought: add the *same* layer to the map two or three times, style each copy for a different zoom band, and set non-overlapping visible ranges. Zoomed out: a binned density surface. Mid zoom: clustered points. Zoomed in: individual features with types styling and full pop-ups. To the reader it feels like one intelligent layer that changes character as they approach. Because each copy references the same hosted layer, there's no data duplication — just three renderings (and performance considerations for multi-copy maps live in Chapter 10).

Design the transitions at the scales where the previous style *breaks* — when a choropleth's polygons get too big to compare, when points start overlapping, when clusters would contain only one or two features. Then zoom slowly through the whole range and watch for the awkward middle band where neither style looks right; that band is where most published maps quietly fail, and where fifteen minutes of tuning visible ranges separates a competent map from a considered one.

Styling is where your data meets your reader. The families give you the grammar, the options give you the vocabulary, and scale strategy makes the whole thing hold together in motion. When you're ready to put words in the map's mouth — pop-ups, labels, and the fields behind them — continue to Chapter 9.

Arcade: From Zero to Fluent

Sooner or later, every map runs into the same wall: the data does not quite say what you want the map to say. The parcels layer stores owner names in shouting capitals. The inspection dates are raw timestamps nobody can read. The field you want to color by does not exist — it is the ratio of two fields that do. You could go back and rebuild the data, but often you should not have to. Arcade is Esri's answer: a small expression language, built into the ArcGIS platform, that computes new values on the fly from the data you already have.

An expression is just a short formula that produces a value. You write it once, and ArcGIS evaluates it for every feature in a layer — every parcel, every hydrant, every survey point — whenever the map draws, a pop-up opens, or a label appears. Arcade expressions travel with the map itself, so anything you build works identically in Map Viewer, ArcGIS Pro, dashboards, Instant Apps, and mobile apps in the field. That portability is Arcade's whole reason for existing: JavaScript and Python each run in only some parts of the platform, while Arcade runs in essentially all of them.

This chapter takes you from your first one-line expression to fluent, debuggable, cross-layer logic. You do not need any programming background. If you can write a spreadsheet formula, you can write Arcade.

Where Arcade Runs: Profiles

Arcade shows up in more places than most people realize. Each place is called a **profile** — a context that defines what information your expression receives, what functions it may use, and what kind of value it must hand back. You do not choose a profile explicitly; you get one automatically based on where you open the expression editor.

But knowing the profile matters, because an expression that works beautifully in a pop-up may be rejected in a labeling context.

PROFILE	WHERE YOU MEET IT	WHAT YOUR EXPRESSION RETURNS	TYPICAL USE
Visualization (styles)	`Styles > + Expression` in Map Viewer; symbology in Pro	A number or category text to symbolize by	Color by a computed ratio; group codes into classes
Labeling	Label settings on a layer	The text of the label	Multi-line labels; abbreviations; conditional labels
Pop-ups	Attribute expressions or Arcade content elements in pop-up settings	Text, a number, or (in Arcade elements) rich content	Friendly summaries; cross-layer lookups
Forms (smart forms)	Calculated values, visibility, and constraints in form settings	A value for a field, or true/false	Auto-fill fields while editing; show/hide questions
Field calculation	`Calculate` on a field in a table (Online or Pro)	The value written permanently into the field	One-time data cleanup; derived columns
Attribute rules	Rules attached to the data itself, set up in Pro	A value, or a pass/fail validation	Enforce data standards automatically on edit
Dashboard data expressions	Data options in ArcGIS Dashboards	A whole table of features	Reshape data feeding a chart or indicator

Two big distinctions run through this table. First, **transient versus permanent**: styles, labels, pop-ups, and dashboard expressions compute values fresh every time and never

touch your data; field calculation and attribute rules write real values into real fields. A mistake in a pop-up expression costs you nothing — delete it and it is gone. A mistake in a field calculation overwrites data. Second, **per-feature speed**: styles and labels run for every visible feature every time the map redraws, so those profiles deliberately forbid slow operations like querying other layers. Pop-ups run for one feature at a time, when clicked, so they are allowed to do much more.

> **Tip:** When you are unsure whether something is possible, check the profile first. The single most common Arcade frustration — "this function worked yesterday!" — is usually a function that exists in the pop-up profile but not in the labeling or visualization profile.

The globals: $feature and friends

Every expression receives a few starting ingredients called **global variables**, all prefixed with a dollar sign. The one you will use constantly is `$feature`: the feature currently being evaluated. Its attributes are available with dot notation:

```
$feature.POPULATION
$feature.OwnerName
$feature["Field Name With Spaces"]
```

Arcade is case-insensitive, so `$feature.ownername` and `$feature.OwnerName` are the same thing — a small mercy when field names are inconsistent. Use the square-bracket form whenever a field name contains spaces or special characters.

Depending on the profile, you may also get `$map` (the web map, used to reach other layers), `$layer` (the layer the expression lives on), `$datastore` (the underlying database, in some contexts), and in forms and attribute rules, variables describing the edit in progress, such as the feature's original values before the edit. The expression editor lists exactly which globals your current profile provides — look for the variables panel alongside the function list.

Syntax Fundamentals

Arcade's syntax will feel familiar if you have seen any programming language, and learnable in an afternoon if you have not. A few ground rules: statements usually go one per line; semicolons at line ends are optional; anything after `//` on a line is a comment the engine ignores; and the expression as a whole must produce a value, which you hand back with the `return` keyword.

```
// The simplest useful expression: return a field's value
return $feature.STATUS
```

Variables

A variable is a named box you store a value in, declared with `var`:

```
var pop = $feature.POPULATION
var area = $feature.AREA_SQKM
var density = pop / area
return Round(density, 1)
```

Nothing here changed the data — `density` exists only for the instant the expression runs. Variables make expressions readable: instead of one dense formula, you build the answer in named steps. Arcade's value types are the ones you would expect: numbers, text (in single or double quotes), true/false values (called booleans), dates, arrays (ordered lists, written `[1, 2, 3]`), dictionaries (labeled bundles of values), geometries, and the special values `null` (no value at all) and empty text.

Making decisions: IIf, When, and Decode

Most real expressions boil down to "if this, then that." Arcade gives you three compact decision functions, plus full `if/else` blocks when logic gets long.

`IIf` is a two-way fork — condition, value-if-true, value-if-false:

```
return IIf($feature.INSPECTED == 'Y', 'Inspected', 'Needs inspection')
```

Note the double equals sign: `==` asks "are these equal?" while a single `=` would be an assignment. Comparison operators are the usual set: `==`, `!=` (not equal), `<`, `>`, `<=`, `>=`, and you combine conditions with `&&` (and) and `||` (or).

`When` is a chain of tests, evaluated top to bottom, with a required catch-all at the end. It reads like a decision ladder and is the workhorse for binning numbers into categories:

```
var d = $feature.DEPTH_M
return When(
  d < 2,  'Shallow',
  d < 10, 'Moderate',
  d < 50, 'Deep',
  'Very deep'   // the default, if nothing above matched
)
```

`Decode` is `When`'s cousin for exact matching: compare one value against a list of candidates and return the paired result. It is ideal for translating codes:

```
return Decode($feature.ZONE,
  'R1', 'Residential – single family',
  'R2', 'Residential – multi family',
  'C1', 'Commercial',
  'Unknown zone')
```

For anything more elaborate, standard `if`, `else if`, and `else` blocks work exactly as you would guess, and `for` loops let you walk through arrays or query results one item at a time. One quirk to memorize now: a `for...in` loop over an *array* hands you index positions (0, 1, 2, ...), not the values — you look each value up with `array[i]` — while a `for...in` loop over a query result hands you the features themselves. Recipe 3 below shows the array form done correctly.

Writing your own functions

When a chunk of logic repeats inside an expression, wrap it in a function:

```
function tidy(value) {
  return Proper(Trim(value))
}
return tidy($feature.OWNER1) + ' & ' + tidy($feature.OWNER2)
```

Functions in Arcade are local to the expression they live in — there is no shared library across expressions, so if five pop-up expressions need the same helper, each carries its own copy. Mildly annoying, but it keeps every expression self-contained and portable.

Nulls: the quiet saboteur

Real data has holes. A field with no value is `null`, and nulls sabotage expressions quietly: a calculation or comparison built on a field that is sometimes null will misbehave on exactly those features, often without raising an error. The essential defenses are `IsEmpty(value)`, which returns true for null and for empty text, and `DefaultValue(value, fallback)`, which substitutes a fallback when the value is empty:

```
var pop = DefaultValue($feature.POPULATION, 0)
```

Watch out: Test every expression against at least one feature with missing data before you trust it. An expression that divides by a field will crash — or silently return nonsense — the first time that field is zero or null. Guard divisions explicitly: `IIf(area > 0, pop / area, null)`.

Working with Text

Half of everyday Arcade is making values presentable, and the `Text()` function does the heavy lifting. It converts numbers and dates into formatted text using a pattern string. For numbers, `#` marks an optional digit, `0` a required one, and `,` inserts grouping separators:

```
Text(1234567.891, '#,###')        // "1,234,568"
Text(1234567.891, '#,###.00')     // "1,234,567.89"
```

```
Text(0.347, '#%')                // "35%"
Text(19.5, '$#,###.00')          // "$19.50"
```

Around that core sits a toolbox of text functions: `Upper`, `Lower`, and `Proper` change case (`Proper('MAIN STREET')` gives "Main Street"); `Trim` strips stray spaces; `Left`, `Right`, and `Mid` slice out pieces; `Find` locates a substring; `Replace` swaps text; `Split` breaks a string into an array at a delimiter; and `Concatenate` joins an array of parts with a separator of your choice. You can also glue text with `+`, which is fine for short cases but gets unwieldy fast — prefer `Concatenate` when assembling more than two or three pieces.

One special citizen deserves mention: in labeling, `TextFormatting.NewLine` inserts a line break, which is how you build stacked, multi-line labels. See Chapter 9 (Pop-ups, Fields, and Labels) for where labels are configured and Chapter 23 for Pro's labeling engine; this chapter only supplies the expressions.

Working with Dates

Dates are where beginners lose the most time, so here is the honest version up front: a date field stores an exact moment, databases store those moments in universal time (UTC), and what you see is usually converted to some local time zone for display. Arcade gives you tools for all of it, and for most everyday work you only need five:

- `Now()` — the current date and time.
- `Today()` — the current date with the time zeroed out. Prefer this when comparing calendar days, so a 9 a.m. timestamp and a 5 p.m. timestamp count as the same day.
- `Date(year, month, day)` — build a date by hand. One trap: months count from zero, so `Date(2026, 0, 15)` is January 15. This inherits from JavaScript and bites everyone once.
- `DateDiff(laterDate, earlierDate, 'days')` — the gap between two dates, in units you choose ('years', 'months', 'days', 'hours', and so on). The first date minus the second, so put the later date first for a positive answer. The result can be fractional — two timestamps 36 hours apart differ by 1.5 days.
- `DateAdd(date, 30, 'days')` — a date shifted forward (or backward, with a negative number).

Pulling pieces out is direct: `Year(d)`, `Month(d)` (also zero-based), `Day(d)`, `Weekday(d)`, `Hour(d)`. And formatting a date for humans goes back through `Text()` with a date pattern:

```
Text($feature.INSPECT_DATE, 'MMMM D, Y')     // "March 4, 2026"
Text($feature.INSPECT_DATE, 'M/D/Y h:mm A')  // "3/4/2026 2:30 PM"
```

Watch out: If a date in a pop-up looks shifted by a few hours — or a "days since" calculation is off by one around midnight — you are looking at a time-zone conversion issue, not a math error. Compare calendar days with `Today()`-anchored logic, strip the time of day off stored timestamps before differencing them (recipe 7 shows how), and when precision matters, investigate how the layer stores time zones on its item page before blaming your expression.

Geometry Functions, at Cruising Altitude

Arcade can reach past a feature's attributes into its actual shape. `Geometry($feature)` returns the geometry — the point, line, or polygon itself — and a family of functions measures and compares shapes: `Area` and `Length` measure size; `Distance` measures separation between two geometries; `Centroid` finds a polygon's center point; `Buffer` inflates a geometry by a distance; and predicates like `Intersects`, `Contains`, `Within`, and `Overlaps` answer yes/no questions about how two shapes relate.

The one distinction worth internalizing now: most measurement functions come in two flavors, planar and geodetic. `Area($feature, 'acres')` measures on the flat projected map, which can be significantly distorted depending on the coordinate system; `AreaGeodetic($feature, 'acres')` measures on the curved earth and is almost always what you actually want for real-world figures. The same split applies to `Length` versus `LengthGeodetic` and `Distance` versus `DistanceGeodetic`. Why flat maps distort area at all is Chapter 3's territory (Coordinate Systems and Projections); the practical rule here is short: **when reporting real-world sizes and distances, reach for the geodetic version.**

Geometry access is not free. Styles and labels evaluate across every feature on screen, so heavy geometry work there can make a map feel sluggish. In pop-ups, where the expression runs once per click, you can be far more relaxed. And note that Arcade reads geometry; it is not the tool for wholesale geometry analysis across layers — that is what the analysis tools in Chapter 16 (Proximity and Overlay) are for.

> **Watch out:** In the styles and labeling profiles, the shape your expression sees may be a simplified, display-optimized copy of the real geometry, so measurements taken there can differ from the true figures. When the number itself matters — an acreage in a pop-up, a length stamped into a field — measure in the pop-up or field-calculation profile instead.

FeatureSets: Reaching Across Layers

Everything so far looked at one feature in isolation. **FeatureSets** break that boundary: a FeatureSet is a queryable collection of features — effectively a whole layer, or a filtered slice of one — that your expression can search, count, and summarize. This is the feature that elevates Arcade from a formatting tool to a genuine lookup engine.

You obtain a FeatureSet in one of three main ways. `FeatureSetByName($map, 'Layer Name')` grabs another layer in the same web map by its name. `FeatureSetByPortalItem(...)` reaches a hosted layer by its item ID even if it is not in the map. And `FeatureSetByRelationshipName($feature, 'RelationshipName')` follows a formal relationship (see Chapter 12, Schema Design) from the current feature to its related records — a hydrant to its inspections, a parcel to its permits.

Once you hold a FeatureSet, a consistent set of verbs applies:

```
var inspections = FeatureSetByName($map, 'Inspections')

// Filter with a where clause — the same SQL syntax
// (the standard database query language) that layer filters use
var failed = Filter(inspections, "RESULT = 'Fail'")

// Count, summarize, sort, take the first
```

```
var n = Count(failed)
var worst = First(OrderBy(failed, 'SCORE ASC'))
var avgScore = Average(failed, 'SCORE')
```

Spatial questions work too, and this is the signature move — the point-in-polygon lookup. Given a point feature and a polygon layer, "which district is this point in?" becomes:

```
var districts = FeatureSetByName($map, 'Council Districts')
var hit = First(Intersects(districts, $feature))
return IIf(IsEmpty(hit), 'No district', hit.DISTRICT_NAME)
```

`Intersects`, when handed a FeatureSet and a geometry, returns the subset of features that touch that geometry. `First` takes the first one (or null if there are none — always check). Suddenly your pop-up can report information the layer never stored.

The honest performance notes

FeatureSets are powerful enough that people over-use them, so here is the truthful fine print.

Every FeatureSet operation is, underneath, a query — and for hosted layers, that usually means a round trip to a server. One lookup in a pop-up is imperceptible. Five FeatureSet expressions in one pop-up means five-plus queries per click, and users will feel it. Dashboards data expressions that chain filters and group-bys over large layers can take seconds to refresh. The costs multiply quietly.

Some concrete habits keep FeatureSets fast:

- **Request only what you need.** `FeatureSetByName` accepts optional arguments listing which fields to fetch and whether to include geometry. `FeatureSetByName($map, 'Parcels', ['STATUS'], false)` pulls one field and no shapes — dramatically lighter than the default.
- **Filter before you count or loop.** Push the work into `Filter` (which the server executes efficiently) instead of looping over every feature and testing conditions in Arcade.

- **Remember which profiles allow it.** FeatureSet functions are available where evaluation happens per event — pop-ups, field calculation, attribute rules, forms, dashboards — and not in the styles and labeling profiles, which must stay fast enough to run per feature per redraw. If you need a cross-layer value to drive symbology, calculate it into a real field first (see Chapter 10 on hosted feature layers, and Chapter 16 for spatial joins at analysis time).
- **Mind the editing context.** In forms, a FeatureSet lookup runs while someone is standing in a field with a tablet; a slow or offline connection makes that lookup hang or fail. Keep form calculations lean.

> **Tip:** A field calculation is the pressure-release valve for FeatureSet performance. If a lookup value rarely changes — the district a parcel sits in, say — run the FeatureSet expression once as a field calculation to stamp it into a real field, then style, label, and filter on that field at zero runtime cost. Recalculate when boundaries change.

Recipes: Fifteen Expressions Worth Stealing

Each recipe below names its intended profile. Adapt field names to your data; everything else should run as written.

1. Proper-case a shouting name field (pop-up or label)

```
return Proper($feature.OWNER_NAME)
```

`Proper` also accepts a second argument, `'firstword'`, if you only want the first word capitalized.

2. Format a number as currency (pop-up)

```
return Text($feature.ASSESSED_VALUE, '$#,###')
```

Add `.00` inside the pattern if cents matter.

3. Assemble an address, skipping blanks (pop-up or field calculation)

```
var fields = [$feature.UNIT, $feature.HOUSE_NO, $feature.STREET, $feature.CITY]
var parts = []
for (var i in fields) {   // for-in over an array yields index positions
  if (!IsEmpty(fields[i])) Push(parts, Trim(Text(fields[i])))
}
return Concatenate(parts, ' ')
```

`Push` appends to an array; only non-empty pieces make the cut, so you never get double spaces or dangling commas from missing units. Note the `fields[i]` lookups — the loop variable is an index, not the value.

4. Show the domain description instead of the stored code (pop-up or label)

```
return DomainName($feature, 'ZONING_CODE')
```

If the field uses a coded-value domain (Chapter 12), this returns the human-readable description — no `Decode` table to maintain.

5. Bin a continuous number into named classes (visualization)

```
var v = $feature.PCT_VACANT
return When(v < 5, 'Low', v < 15, 'Moderate', v < 30, 'High', 'Severe')
```

Return the class name, then style with unique values — Chapter 7 covers wiring an expression into a style.

6. Safe percentage of two fields (pop-up or visualization)

```
var num = DefaultValue($feature.RENEWED, 0)
var den = DefaultValue($feature.TOTAL, 0)
return IIf(den > 0, Round(num / den * 100, 1), null)
```

Returning `null` on a zero denominator lets the map's "no data" handling take over instead of showing a bogus zero.

7. "Days since" with friendly wording (pop-up)

```
var d = $feature.LAST_INSPECTION
if (IsEmpty(d)) return 'Never inspected'
var dDay = Date(Year(d), Month(d), Day(d))  // strip the time of day
var days = Round(DateDiff(Today(), dDay, 'days'))
return When(days == 0, 'Inspected today',
            days == 1, 'Inspected yesterday',
            'Inspected ' + Text(days, '#,###') + ' days ago')
```

Stripping the time first keeps the count stable around midnight — remember that `DateDiff` returns fractional days when a timestamp carries a time of day. (`Month()` and `Date()` are both zero-based, so they agree.)

8. Overdue flag from an expiration date (visualization or pop-up)

```
var exp = $feature.PERMIT_EXPIRES
if (IsEmpty(exp)) return 'No permit'
return IIf(exp < Today(), 'EXPIRED', 'Current')
```

Two return values, two symbols — an instant compliance map.

9. Age in years from an installation date (pop-up, or field calculation for reuse)

```
return Floor(DateDiff(Today(), $feature.INSTALL_DATE, 'years'))
```

`Floor` rounds down, which is how ages work: a pipe installed 11.8 years ago is 11 years old.

10. Two-line label with a graceful fallback (labeling)

```
var name = DefaultValue($feature.SITE_NAME, 'Unnamed site')
return name + TextFormatting.NewLine + Text($feature.CAPACITY, '#,### units')
```

11. Real-world acreage from the polygon itself (pop-up or field calculation)

```
return Round(AreaGeodetic($feature, 'acres'), 2)
```

No area field required — the shape carries the answer, measured on the curved earth.

12. Which polygon contains me? (pop-up)

```
var zones = FeatureSetByName($map, 'Flood Zones', ['ZONE_CLASS'], true)
var z = First(Intersects(zones, $feature))
return IIf(IsEmpty(z), 'Outside mapped flood zones', 'Flood zone: ' +
z.ZONE_CLASS)
```

The classic cross-layer lookup. Geometry must be included (the final `true`) for the spatial test to work.

13. Count related records through a relationship (pop-up)

```
var visits = FeatureSetByRelationshipName($feature, 'Inspections')
return Count(visits) + ' inspection(s) on record'
```

14. Sum a field across a filtered slice of another layer (pop-up)

```
var sales = FeatureSetByName($map, 'Sales', ['AMOUNT', 'REGION'], false)
var mine = Filter(sales, "REGION = '" + $feature.REGION + "'")
return Text(Sum(mine, 'AMOUNT'), '$#,###')
```

Note how the current feature's own attribute is spliced into the filter — the pop-up feature drives the query. The same pattern works in a dashboard data expression, but

there is no `$map` there: fetch the layer with `FeatureSetByPortalItem` instead.

15. Extract text before a delimiter (field calculation)

```
// 'SMITH, JOHN' -> 'SMITH'
return Trim(Split($feature.FULL_NAME, ',')[0])
```

`Split` returns an array; `[0]` takes the first element. Run on a copy first — field calculations are permanent (see Chapter 11 for safe data-editing habits).

Debugging Expressions

Every Arcade editor across the platform shares the same basic anatomy: a code panel, a searchable list of available functions and globals for the current profile, and a way to test-run the expression against a sample feature. Fluency in debugging means using all three deliberately.

Run early, run often. Do not write twenty lines and then test. Write two lines, run, confirm, extend. The test run evaluates your expression against one sample feature and shows the result or the error. In Map Viewer the editor picks a feature for you; be aware the sample may be unusually clean — a feature with no nulls proves nothing about the features with nulls.

Use Console for X-ray vision. The `Console()` function prints anything you hand it to a messages panel in the editor without affecting the returned value:

```
var d = $feature.LAST_INSPECTION
Console('raw date: ' + d)
var days = DateDiff(Today(), d, 'days')
Console('days computed: ' + days)
return days
```

When an expression returns something baffling, sprinkle `Console` calls after each step and read the story of where the value went wrong. Remove them when you are done;

they cost a little performance and clutter nothing but your own view, since end users never see console output.

Read error messages literally. Arcade's errors are terse but honest. "Field not found" means exactly that — check spelling against the fields list in the editor's side panel rather than typing field names from memory, and remember that a layer configured to expose only some fields will hide the rest from your expression. "Function not available" almost always means the function exists but not in this profile. A type complaint ("cannot compare text and number") means a field holds text that merely looks numeric — convert deliberately with `Number()` before doing math.

When the result is empty rather than an error, suspect nulls first, then the sample feature, then time zones (for dates), and only then your logic. `Console(IsEmpty($feature.THE_FIELD))` settles the null question in one run.

When a FeatureSet expression is slow or silently fails, test the layer independently: can you see it in the map, does its name exactly match the string in `FeatureSetByName`, and does your account have access to it? An expression can only query what the signed-in user is allowed to see — an expression that works for you and shows blanks for the public usually means the referenced layer is not shared as widely as the map (sharing is Chapter 34's territory).

> **Watch out:** Renaming a layer in the map breaks every `FeatureSetByName` expression that referenced the old name, quietly. If you rename layers, re-open your expressions and update the strings. This is one of the most common "the pop-up just stopped working" causes in the wild.

> **Tip:** Keep a personal recipe file. Arcade has no shared library, so fluent users keep their proven expressions in a plain text document and paste-and-adapt. The fifteen recipes above are a starter kit; yours will grow to fit your data.

Where Arcade Ends

Arcade is deliberately bounded. It cannot call external web services or run scheduled jobs — it computes values when the platform asks and does nothing in between. Nor

does it change your data unbidden: only the field-calculation and attribute-rule profiles write anything, and only when an edit or calculation actually happens. When you outgrow those bounds — batch automation, heavy analysis, custom applications — the escalation path is Python (Chapter 31) for automation and analysis, and the Maps SDK for JavaScript (Chapter 33) for custom apps, where, satisfyingly, your Arcade expressions still work, because the SDK evaluates the same language. Learn Arcade once and it pays rent everywhere: smarter styles in Chapter 7, richer pop-ups and labels in Chapter 9, smarter forms in Chapters 13 and 30, and dashboard indicators in Chapter 28 all speak it fluently — and now, so do you.

Pop-ups, Fields, and Labels, Complete

A map earns a glance. A pop-up earns a conversation. When someone clicks a feature on your map — a hydrant, a parcel, a hiking trail — the pop-up is where the map stops being a picture and starts being an answer. Labels do the same job one step earlier: they answer the most common question ("what is this?") before anyone has to click at all. And underneath both sits the humble field — the column of data whose name and formatting decide whether your pop-up reads like a report or like a database dump.

This chapter covers all three in depth, because they are really one subject: how the attributes in your data become words on the screen. Everything here happens in Map Viewer, the web map editor covered end to end in Chapter 6 (Map Viewer: The Complete Reference). Where the Arcade scripting language comes up — and it comes up a lot — the language itself is taught in Chapter 8 (Arcade: From Zero to Fluent); here you'll see what it's *for*.

Fields First: Display Names and Formatting

Before you touch a single pop-up setting, fix your fields. Every improvement you make here flows automatically into pop-ups, labels, tables, and charts, which makes it the highest-leverage ten minutes in this entire chapter.

A quick refresher on vocabulary. Every layer stores its data in a table, and each column in that table is a **field**. A field has two names: the **field name**, which is the machine name baked into the data (often something like `INSP_DT_LAST` — short, cryptic, and unchangeable without editing the schema, the layer's underlying table structure), and the **display name** (Esri also calls this an **alias**), which is the human-readable label shown to your readers. The field name is permanent plumbing; the display name is decoration you can change any time.

To edit display names and formatting in Map Viewer, select your layer and open `Fields` from the settings toolbar. Click any field to edit it.

Display names worth reading

Rewrite every display name a reader will ever see. `INSP_DT_LAST` becomes "Last inspection". `POP2020` becomes "Population (2020)". Three habits pay off:

- **Write for the reader, not the database.** "Owner mailing address" beats "OWN_ADDR_M" every time, and there is no cost — the underlying field name is untouched, so nothing downstream breaks.
- **Put units in the name.** "Pipe diameter (inches)" prevents the single most common misreading of numeric data. A bare "Diameter: 8" forces every reader to guess.
- **Keep names short enough to scan.** Pop-up field lists render as a two-column table; a forty-character display name squeezes the value into a sliver.

Number formatting

For each numeric field, the same panel lets you set how many decimal places to show and whether to use a thousands separator. Two rules cover almost every case:

- **Round to the precision you actually trust.** A calculated area of 12,847.29471 square meters should display as 12,847. Extra decimals don't communicate accuracy — they fake it, and they make numbers harder to compare at a glance.
- **Turn on the thousands separator for anything that can exceed four digits** — populations, areas, dollar amounts. Leave it off for things that are codes rather than quantities: years, ZIP codes, asset IDs stored as numbers. "Installed: 2,014" is a genuinely embarrassing pop-up, and it happens constantly.

Date formatting

Date fields get a format picker: short numeric dates, long spelled-out dates, and variants with or without time of day. Choose based on what the date means. An inspection that happens once a year does not need "3:47 PM" appended to it; a 911 call absolutely does. Spelled-out months ("March 4, 2026") also sidestep the international ambiguity of 03/04/2026, which is March 4th to an American and April 3rd to nearly everyone else.

> **Tip:** Field formatting set this way is saved in the *web map*, not in the layer itself. If you want the same display names and formats to appear in every map that uses the layer, set them on the layer's item page instead (open the layer item and look for the fields or visualization settings there). Chapter 10 (Hosted Feature Layers) explains this layer-versus-map settings split in detail — it is one of the most common sources of "why did my formatting disappear?" confusion.

The fields you shouldn't show at all

Every layer carries fields nobody needs to see: internal object IDs, editor-tracking columns, GlobalIDs (long unique codes used for syncing), leftover join keys. You'll hide them from the pop-up shortly — but flag them now, while you're already looking at the field list. A good pop-up shows five to twelve fields. If your layer has forty, the real work is choosing which thirty of them the reader never sees.

Anatomy of a Pop-up

Select a layer in Map Viewer and open `Pop-ups` from the settings toolbar. The pane shows a toggle to enable or disable pop-ups for that layer, a title, and a stack of **content blocks** — independent pieces of content that render top to bottom when a feature is clicked. You add blocks with an `Add content` button, drag them to reorder, and configure each one individually.

Turning pop-ups *off* is a legitimate design choice, by the way. Basemap-like reference layers — say, a county boundary that exists only for context — should usually have pop-ups disabled so they don't intercept clicks meant for the layer that matters.

The title

The title is the headline, and the default is usually terrible: the layer name, or the first text field, or a raw ID. You can type static text, insert field values using curly braces — `{SchoolName}` — or mix the two: `{SchoolName} – {District}`. The braces are placeholders; when a reader clicks a school, `{SchoolName}` is replaced with that school's actual name.

A good title passes one test: if the reader saw *only* the title, would they know what they clicked? "Lincoln Elementary — Ward 4" passes. "Feature 20871" does not. For titles that need logic — say, showing a facility's name if it has one and its address if it doesn't — you can drive the title with an Arcade expression instead of a simple field placeholder.

Content blocks at a glance

BLOCK	WHAT IT SHOWS	BEST FOR
Fields list	A two-column table of chosen fields and values	The factual core of most pop-ups
Text	Formatted sentences with field values woven in	Narrative context, plain-English summaries
Image	A photo or graphic, from a URL or an attachment	Site photos, logos, diagrams
Chart	A small bar, line, or pie chart of numeric fields	Comparing several numbers within one feature
Arcade	Content generated entirely by a script	Anything conditional, computed, or fetched from other layers
Attachments	Files stored with the feature	Inspection photos, PDFs, documents
Related records	Rows from a related table	Repeat data like inspection histories (see Chapter 12)

The craft is in combining them. A strong pop-up often reads: text block (one summary sentence), fields list (the key facts), image (the site photo), attachments (the

paperwork). Let's take each block in depth.

Content Blocks in Depth

The fields list

The workhorse. It renders selected fields as a tidy label-and-value table, using the display names and formatting you set earlier — which is why fields came first in this chapter. Click into the block to choose which fields appear and in what order.

Order fields by the reader's priorities, not the table's column order: identity first (name, type), then status, then measurements, then administrative trivia — if the trivia makes the cut at all. Fields that are empty for a given feature show up as blank rows, which looks unfinished; if a field is empty for *most* features, consider dropping it from the list and surfacing it another way (the Arcade section below shows how to display a field only when it has a value).

The text block

The text block is a small rich-text editor: paragraphs, bold and italics, bulleted lists, hyperlinks, and — crucially — field placeholders in curly braces. This is how you turn data into sentences:

This {ZoneType} zone covers {Acres} acres and was last reviewed on {ReviewDate}.

Each reader sees the sentence completed with their clicked feature's values. Text blocks shine as one-line summaries above a fields list, or as the entire pop-up for simple layers where a sentence beats a table.

A few cautions. Placeholders substitute raw-ish values, so grammar can break in ways you won't notice until the wrong feature is clicked — "covers 1 acres," or a sentence with an awkward hole where an empty field should be. When wording must adapt to the data (singular versus plural, present versus absent), that's a job for an Arcade element, not a static sentence. And resist long paragraphs: pop-ups are read in a small floating window, often on a phone. Two short sentences beat five long ones.

You can also make links dynamic: a hyperlink's address can contain a placeholder, such as `https://example.gov/permits/{PermitID}`, so every feature links to its own record in

some external system. This one trick — pop-up as doorway into another system — is often the most valuable thing a pop-up does.

The image block

An image block displays one picture, with optional caption, alt text, and a link to open when clicked. The image comes from a URL. There are three common sources:

- **A URL stored in a field.** If your layer has a `PhotoURL` field, set the image source to `{PhotoURL}` and every feature shows its own photo. This is the most flexible pattern and the easiest to maintain.
- **A static URL.** Same image for every feature — occasionally right for a legend graphic or logo, usually a waste of pop-up space.
- **Attachments.** If the layer has attachments enabled, the pop-up can display attached images directly; more on attachments below.

The image must be hosted somewhere your readers can reach. A URL that works on your office network but nowhere else will produce broken-image icons for the public — test while signed out, on a phone, off the office Wi-Fi.

> **Watch out:** Images served over plain `http://` rather than secure `https://` are blocked by modern browsers when the map itself is served securely — which it always is on ArcGIS Online. The pop-up will simply show nothing where the photo should be, with no error message. If a photo mysteriously fails to appear, check the URL scheme first.

The chart block

A chart block draws a small chart *from the numeric fields of the clicked feature*. That italicized part is the concept people miss: a pop-up chart does not show trends across many features (that's a Dashboard's job — see Chapter 28). It compares several numbers belonging to *one* feature. A census tract with fields for five age brackets becomes a little bar chart of that tract's age structure. A monitoring station with twelve monthly rainfall fields becomes a line chart of that station's year.

You pick the chart type (column, bar, line, or pie) and the set of numeric fields to include. Guidance:

- Charts need **several comparable fields in the same units**. If your data stores one number per feature, there is nothing to chart — don't force it.
- Use **column or bar** for comparing categories, **line** only when the fields form a genuine sequence (months, years), and treat **pie** with suspicion: more than four or five slices is unreadable at pop-up size.
- The field display names become the chart labels — one more dividend from the aliasing work you did earlier.

The Arcade element

The Arcade content block is the escape hatch: instead of configuring content, you write a short script that *builds* content — text, a field list, or media — and the pop-up renders whatever the script returns. Chapter 8 teaches the language; here is what it unlocks that no other block can do:

- **Conditional content.** Show a red overdue warning only when the inspection date is past due; show "No hazards reported" when the hazards field is empty rather than an awkward blank.
- **Computed values.** Display density from population and area, age from a birth date, a letter grade from a numeric score — without adding a single field to your data.
- **Cross-layer lookups.** An Arcade expression can query *other layers in the map*: a clicked parcel's pop-up can report which flood zone it falls inside, or count the permits within it, even though the parcel layer stores neither fact. This borders on magic and is the single best reason to learn Arcade.

Arcade also works in smaller doses throughout the pop-up configuration: expressions can drive the title, individual list entries, and image URLs. A common pattern is a mostly ordinary pop-up with one expression-driven line doing something clever.

> **Tip:** Keep a "no data" branch in every Arcade expression. Real data has nulls, and an expression that assumes a value exists will render an error or an empty block for exactly the features your readers ask about. A simple check — if the field is empty, return a friendly fallback string — is the difference between polished and broken.

The attachments block

If (and only if) the layer has attachments enabled, this block surfaces the files stored with each feature. You typically choose between a compact list of file links and a gallery-style preview for images. Attachments deserve their own section, next.

Attachments in Practice

An **attachment** is a file — photo, PDF, spreadsheet, short video — stored *inside* the feature layer, tied to a specific feature. Unlike an image URL, which points at a picture hosted elsewhere, an attachment travels with the data: copy the layer, and the files come along.

Attachments must be enabled on the layer itself, which is a layer-item setting rather than a map setting (find it on the layer's item page; Chapter 10 covers layer settings, and Chapter 12 discusses when attachments belong in your schema at all). Once enabled, files get attached in three main ways: field crews snap photos in Field Maps or Survey123 (Chapter 30 — this is by far the most common source), editors add files while editing in Map Viewer or ArcGIS Pro, or a script loads them in bulk (Chapter 31).

Practical guidance that saves grief later:

- **Photos are the sweet spot.** Image attachments preview beautifully in pop-ups. PDFs and other documents show as download links — useful, but readers must leave the map to view them.
- **Size adds up fast.** Attached files count against your organization's storage. A field operation collecting several photos per feature can quietly become the largest storage item in your organization. Downsizing photos on capture — most field apps can do this — is almost always worth it.
- **Attachments have no captions or guaranteed order.** If which-photo-is-which matters ("before" versus "after"), encode it in the filename or add a field describing the photos, because the pop-up gallery won't tell your reader.
- **Sharing includes attachments.** If a layer is public, its attachments are public. Field photos have a way of capturing license plates, faces, house interiors, and handwritten notes. Review before you share widely.

Watch out: Deleting a feature deletes its attachments with it — permanently, with no recycle bin. If attached photos are records you're legally required to keep, the layer is not an archive; export attachments somewhere durable before any bulk cleanup.

Labels: The Pop-up You Don't Have to Click

A **label** is text drawn on the map next to a feature, straight from the data. Labels answer "what is this?" for every feature simultaneously — no clicking — which makes them the most efficient communication tool you have. They are also the easiest thing to overdo: a map where everything is labeled is a map where nothing is readable.

Select a layer and open `Labels` from the settings toolbar. Enable labels, then work through four decisions: what text, at which scales, in what style, and placed where.

What text: fields and label expressions

The simple case is a single field — label parcels with `ParcelID`, streets with `StreetName`. The label picker offers your fields, shown by their display names.

For anything fancier, labels accept Arcade expressions, and three patterns cover most needs:

- **Combining fields:** name on top, capacity beneath — an expression can join two fields with a line break, giving compact two-line labels.
- **Reformatting:** data stored in shouting uppercase (`MAPLE AVE`) can be converted to title case (`Maple Ave`) at label time, with no edits to the data. Rounding numbers and adding units ("2.4 mi") works the same way.
- **Selective labeling:** an expression that returns empty text for features below some threshold labels only the features that matter — only cities above a population cutoff, only assets in "Failed" status. This is the most powerful de-cluttering trick in web labeling.

Label classes

A **label class** is one complete set of labeling rules — its own text, style, scale range, and optionally its own filter so it applies only to some features. Map Viewer lets you add

several label classes to one layer, and this is how you build grown-up labeling:

- Cities layer: one class labels state capitals in a larger, bolder style; a second class labels everything else smaller. The filter on each class (by a `Capital` yes/no field) keeps them apart.
- Roads layer: highways get labels visible even when zoomed far out; residential streets get a class that only appears when zoomed way in.

If you find yourself wanting one class's rules to vary by feature type, that's the signal to split it into two classes.

Multi-scale labeling: the visible range

Every label class has a **visible range** — the span of zoom levels at which it draws. This one slider does more for map readability than any font choice. The principle: a label should appear only at scales where its feature is individually distinguishable. Parcel numbers at a citywide zoom are pixel soup; at neighborhood zoom they're useful. Set each class's range by zooming to where the labels start to help, noting the level, and clamping the range there.

Layer visibility has its own separate range setting (Chapter 6), and the two interact: features can be set to appear zoomed out while their labels wait until closer in. That staging — shapes first, names later — is exactly how professional basemaps behave.

Style: halos, fonts, and restraint

Label style options include font, size, color, boldness, and the **halo** — a thin contrasting outline around each letter. The halo is not decorative; it is what keeps text legible when it crosses a busy background, which on a real map is always. A white halo around dark text (or dark around light) is effectively mandatory. Keep it thin — a couple of pixels reads cleanly, while a thick halo turns text into puffy blobs.

Beyond halos: prefer one font family across the whole map and vary size and weight instead; keep label colors near-neutral (dark gray on light basemaps) unless color is carrying meaning; and if you want labels to visually pair with a layer's symbols, tint them a darker shade of the symbol color rather than the symbol color itself — saturated text is hard to read. Chapter 4 (Cartographic Design) covers the typographic reasoning;

the ten-second version is that labels should feel like they were printed on the basemap, not stuck on top of it.

Placement

Placement options depend on geometry. Point labels can sit above, below, or beside the point — pick one convention per map and let the engine flip sides when it must to avoid collisions. Line labels can run along the line, which is right for streets and rivers even though curved text renders a bit less crisply. Polygon labels sit at the polygon's visual center; odd-shaped polygons (a C-shaped park) sometimes get a center that falls outside themselves, and web labeling gives you little recourse beyond simplifying the label or living with it.

One honest limitation to plan around: the labeling engine in web maps automatically drops labels that would overlap, and *which* labels it drops is not fully under your control. In dense areas, some features simply won't be labeled. That's usually the right trade — overlapping text is worse — but if you need publication-grade control over every label's exact position, that is ArcGIS Pro territory: see Chapter 23 (Symbology and Labeling: The Pro Engine) and Chapter 24 for print layouts. In the web, the winning move is fewer, better labels: tighter visible ranges, selective expressions, and multiple classes.

> **Tip:** After configuring labels, pan slowly across the densest part of your data at each zoom level you expect readers to use. Label problems live in the crowded corners you didn't test, not in the tidy area you configured while looking at.

Accessibility and Readability

Everything above determines whether your map merely displays information or actually delivers it — including to readers using screen readers, readers with low vision or color-vision deficiency, and readers on a phone in sunlight. A short checklist that costs little and helps everyone:

- **Write alt text for pop-up images.** Alt text is the short written description a screen reader speaks in place of a picture; the image block has a spot for it. Describe what

the photo shows ("Cracked sidewalk panel with exposed rebar"), not what it is ("photo").

- **Never let color carry meaning alone.** If the pop-up says a status in red text, also say it in words: "Status: Overdue." Labels styled by color need the same backup. Roughly one in twelve men can't reliably distinguish red from green.
- **Check contrast on labels and pop-up text.** Dark-on-light or light-on-dark, with halos on labels. Mid-gray text on a mid-toned basemap fails for everyone, not just readers with low vision.
- **Structure pop-ups for linear reading.** Screen readers walk the pop-up top to bottom, block by block. Put the summary first and order fields by importance — which, conveniently, is identical to good design for sighted readers.
- **Keep sentences and display names plain.** A pop-up is read in five seconds by someone who did not study your data model. "Last inspection" beats "Most recent inspection event date-time."
- **Test at phone size.** Long titles wrap, wide charts squeeze, and giant images push the fields list below the fold. Most public map traffic is mobile; design for the small screen and the desktop takes care of itself.

A Working Method

When you configure a new layer, run this order and each step makes the next one easier: fix field display names and formatting; hide the fields nobody needs; write a title that identifies the feature; build the pop-up from blocks — summary text, trimmed fields list, image or attachments if you have them, an Arcade element for anything conditional; then add labels with a deliberate visible range, a halo, and an expression if the raw field isn't display-ready. Finish by clicking around the map as a stranger would — signed out, on a phone — and read what they'd read.

Pop-ups and labels are also where later chapters compound. The same Arcade skills deepen in Chapter 8; the pop-ups you configure here carry straight into Instant Apps (Chapter 26), StoryMaps (Chapter 27), and Dashboards (Chapter 28) — most app builders inherit the web map's pop-up configuration rather than defining their own, so effort invested here pays out everywhere the map travels.

Hosted Feature Layers: Publishing, Views, and Performance

Every web map you have built so far rests on a piece of infrastructure you have probably never inspected: the hosted feature layer. When you uploaded that spreadsheet of park locations and it appeared as dots on a map, ArcGIS Online quietly did three things. It parsed your file, loaded the rows into a cloud database that Esri operates on your behalf, and stood up a small web service that hands out those rows to any map that asks. The dots you see are not your spreadsheet. They are the answers to questions your map keeps asking that service: "what features are inside this rectangle, at this zoom level, with these fields?"

That arrangement — your data living in Esri's cloud, served on demand through a web endpoint — is what "hosted" means. A **feature layer** is a collection of geographic features (points, lines, or polygons) with a table of attributes attached, as described in Chapter 1 (How GIS Thinks). A **hosted feature layer** is one where ArcGIS Online stores the data and does the serving, so you never touch a server. The alternative — running your own server infrastructure — is the subject of Chapter 35 (ArcGIS Enterprise and When You Need It), and for most people it is unnecessary.

This chapter follows the full life of a hosted feature layer: how it gets born, the settings that control what people can do to it, how to create filtered "views" of it for different audiences, how to refresh its data without breaking every map built on top of it, how to keep it fast as it grows, and how to back it up and keep its storage bill sane. Styling and pop-ups are decoration; the hosted layer is the building.

The Anatomy: Item, Layer, and Service

Before publishing anything, it helps to untangle three words Esri uses almost interchangeably.

The **service** is the actual web endpoint — a URL that speaks REST, a web convention for asking servers questions through structured web addresses (covered properly in Chapter 32). Every hosted feature layer has one. The layer's item page displays this URL; paste it into a browser and you will see a plain, unstyled directory of what the service offers. Maps, apps, and scripts all talk to this URL.

The **layer** is one dataset inside that service. A service can contain several layers and tables — a "Schools" service might hold a points layer of campuses, a polygons layer of attendance zones, and a plain table of enrollment figures, all traveling together.

The **item** is the catalog entry in your content — the card with a title, thumbnail, summary, and settings. The item is what you share, protect, and manage. It points at the service.

Why care? Because the failure modes of hosted layers almost always come down to confusing these three. You can delete an item and orphan the maps pointing at its service. You can share an item but forget that a view of the same service has different sharing. And when you "overwrite" a layer, what you are really doing is replacing the data behind the service while keeping the item — which is exactly why overwriting, done correctly, does not break your maps.

Publishing: Every Path In

There are more ways to create data than to publish it, and Chapter 11 (Creating Data: Every Path In) covers the creation side — digitizing, importing, geocoding, field collection. Here we care about the moment data becomes a hosted feature layer. The main routes:

PUBLISH PATH	WHERE IT HAPPENS	WHAT YOU GET	BEST FOR
Upload a file	`Content > New item`, then drag in a CSV, Excel file, shapefile, GeoJSON, or file geodatabase	The source file as one item, plus a hosted feature layer as a second item	One-time or occasional data loads
Sketch in Map Viewer	Create a new editable layer while making a map	A hosted feature layer with the schema you define	Small datasets you draw by hand

PUBLISH PATH	WHERE IT HAPPENS	WHAT YOU GET	BEST FOR
Publish from ArcGIS Pro	`Share > Web Layer` in Pro	A hosted layer mirroring your Pro data, with full control over schema and styling	Curated datasets prepared on the desktop
A form or field app creates it	Survey123, Field Maps, or QuickCapture generates the layer behind a survey or collection project	A hosted layer wired to a data-collection workflow	Field operations (Chapter 30)
Scripted publishing	The ArcGIS API for Python	A hosted layer created and updated by code	Recurring, automated data feeds (Chapter 31)

Two publishing details matter for the rest of the layer's life.

First, **the file-upload path creates two items**, and the pairing is not cosmetic. The hosted feature layer remembers which file it came from, and that memory is what powers the easiest update workflow later: replacing the layer's contents by uploading a fresh copy of the same file. If you delete the source file item to tidy up, you lose that convenience. Keep both, and consider putting them in the same folder so their relationship stays obvious.

Second, **publishing is when your schema gets locked in the ways that matter**. The schema is the layer's structure — its fields, their types, and the rules around them. Field names, field types, and whether a field allows empty values are all far easier to get right before publishing than to change after. A field published as text cannot casually become a number later; the practical fix usually involves adding a new field and recalculating, or republishing. Chapter 12 (Schema Design) is the pre-flight checklist worth reading before you publish anything you intend to keep.

Tip: Before publishing a spreadsheet, open it and ruthlessly clean the header row. Column names become field names, and cryptic or duplicated headers become

cryptic field names that you, your pop-ups, and your Arcade expressions (Chapter 8) will live with indefinitely. A minute of renaming now saves an hour of aliasing later.

When you publish, ArcGIS Online also asks (or decides) how to locate your rows in space. Files with explicit coordinates or geometry — shapefiles, GeoJSON, geodatabases — carry their own spatial reference, which Chapter 3 (Coordinate Systems) explains. Spreadsheets with only street addresses must be geocoded, which converts addresses to coordinates and consumes credits — the metered currency of ArcGIS Online, covered in Chapter 34 (Administration). Coordinates are free; addresses are not. If you can get latitude and longitude columns into your spreadsheet before publishing, do.

The Item Page: Settings That Control Everything

Open any hosted feature layer from your content and you land on its **item page** — tabs for overview, data, visualization, usage, and settings. The `Settings` tab is where the layer's rules of engagement live. Four clusters of settings do most of the work.

Editing

By default, a newly published layer is usually not editable by others — only you, the owner, can change it. Turning on editing opens the data to whoever the layer is shared with, and the sub-options matter enormously:

- **What editors can do**: add features only, update only, delete only, or any combination. A public suggestion box might allow add-only, so contributors can submit points but never tamper with anyone else's.
- **What editors can see and edit**: everyone's features, or only their own. "Only their own" is the backbone of crowdsourcing and multi-crew field work — each editor operates in a private slice of a shared layer.
- **Attribute-only editing**: allow changing the table values but not the geometry, useful when locations are authoritative but statuses change.

Alongside editing sits **editor tracking** — an option to record who created each feature and when, and who last edited it and when. Turn this on for anything multiple people

touch. It costs you four system-maintained fields and buys you an audit trail; Chapter 13 (Editing Workflows) leans on it heavily.

> **Watch out:** Editing permissions travel with sharing. An editable layer shared with "Everyone" is editable by everyone — including anonymous strangers, including bots. If a layer must be both public and editable (a genuine public survey, say), restrict editors to add-only, hide other people's features, and treat everything submitted as unverified until reviewed. Better still, keep the master layer private and expose an editable view, a pattern we will build shortly.

Sync

Sync lets copies of the layer be taken offline — onto a phone in a no-signal canyon, for instance — and reconciled with the master copy when connectivity returns. It is the machinery behind offline field work in Field Maps, and Chapter 30 covers the workflow. Here, know two things. Enabling sync is a settings checkbox, and it changes the layer's internal bookkeeping: the service starts maintaining **replicas**, registered offline copies it must track so changes can merge back. That bookkeeping constrains other operations — most notably, a layer with active offline copies will resist being overwritten, because destroying data out from under an unreturned replica would make reconciliation impossible. Enable sync when you need offline; leave it off otherwise.

Attachments

Attachments let each feature carry files — most often photos, sometimes PDFs or documents. An inspector photographs a cracked hydrant; the photo rides along with that hydrant's record and appears in its pop-up (Chapter 9). Enable attachments in the layer's settings before the field crew heads out, because retrofitting them onto a collection already in progress is awkward. Attachments are wonderful and heavy: a layer of ten thousand inspection points might occupy a trivial amount of storage while its photos occupy orders of magnitude more. We will return to that in the storage section.

Delete protection and the safety rail

On the settings tab sits an unassuming checkbox: **delete protection**. When on, the item cannot be deleted until someone deliberately switches the protection off first — a two-step that turns "oops" into "are you absolutely sure, twice." Deleting a hosted feature layer is close to irreversible: the maps and apps that referenced it do not get copies of the data; they get broken layers and error messages. Turn delete protection on for any layer that a map, app, dashboard, or colleague depends on. This should be a reflex, like locking your car.

The same tab holds the switch that allows others to **export the data** — download it in various formats. Leave it off for layers where you want to control distribution; turn it on for open-data offerings. And note **sharing** itself — private, organization, groups, or everyone — is managed from the item as well; the mechanics of groups and sharing levels belong to Chapter 34.

View Layers: One Dataset, Many Faces

Sooner or later you will face this problem: one dataset, several audiences with different needs. The field crew needs to edit. The public should see most of it, read-only, but not the columns holding phone numbers. The regional office only cares about its own region. The wrong answer is to make three copies of the data, because copies immediately begin to drift apart and you now maintain three of everything. The right answer is the **hosted feature layer view**.

A view is a second item that presents the *same underlying data* as its source layer, through a lens you configure. No data is duplicated. Edit a feature through the source, and every view reflects the change instantly, because there is only one set of rows. A view has its own item page, its own sharing settings, its own editing settings, its own pop-up and styling defaults — everything about the *presentation and permissions* is independent, while the *data* stays single.

You create a view from the source layer's item page — look for a `Create View Layer` action on the overview. During creation (or afterward, in the view's settings) you shape the lens three ways:

- **Limit the fields.** Choose which columns the view exposes. The phone-number and internal-notes fields simply do not exist as far as the view's consumers are

concerned — they are not hidden by a pop-up setting someone could undo; they are absent from the service the view presents.

- **Limit the features.** Apply a filter — a condition on attribute values, called a definition in some parts of the interface — so the view serves only matching rows. `status = 'Approved'` gives the public a view containing only vetted records while drafts stay invisible.
- **Limit the area.** Restrict the view to a geographic extent, so only features inside that boundary are served.

These limits are enforced by the service itself, which is what makes views genuinely trustworthy for security-adjacent purposes in a way that map-level filters are not. A filter set inside a web map (Chapter 6) is a polite request that a determined user can strip away by opening the layer directly. A view's limits are the layer.

The master-and-views pattern

Put the pieces together and you get the single most useful architecture in ArcGIS Online:

The **master layer** holds everything: all fields, all rows, full editing enabled, editor tracking on, delete protection on — and it is shared with almost no one. Perhaps only you, or a small data-management group. Nobody builds maps directly on the master.

Around it, you create views, each tuned to an audience:

VIEW	FIELDS	FEATURES	EDITING	SHARED WITH
Public view	Non-sensitive fields only	Approved records only	None (read-only)	Everyone
Field crew view	Fields the crew fills in	Their assigned area	Add and update, see own features	The crew's group
Manager view	All fields	All records	Read-only	Managers' group

Maps and apps are built on the views, never the master. The public web map consumes the public view; the Field Maps project consumes the crew view; the dashboard (Chapter 28) consumes the manager view. Everyone sees one consistent, live dataset, sliced appropriately — and when policy changes ("actually, hide the cost field from the public"), you adjust one view and every public map updates, because they all drink from the same tap.

Views can be editable, and an editable view is often *safer* than an editable source: you can grant the crew add-and-update rights through their view while the master's own sharing stays locked down, and the view's field list keeps them from touching columns that are not their business.

> **Tip:** Name views for their audience and purpose, not just the dataset — `Hydrants (public, read-only)` beats `Hydrants_view_2_final`. Six months from now, when you are staring at five similarly named items, the names are the documentation.

Two limits keep views honest. A view cannot contain anything its source lacks — it is a lens, not a new dataset, so schema changes (new fields, domains — the pick-list field rules from Chapter 12) happen on the source and flow outward to every view. And a view cannot exist apart from its source: ArcGIS Online refuses to delete a source layer while views still depend on it, and once the views are gone, deleting the master takes the data out from under everything that used them — one more reason delete protection belongs on the master. Views also come only from source layers: you cannot create a view of a view, so the architecture stays one layer deep by design — a master and its views.

Updating Data Without Breaking Maps

Data goes stale. New inspections happen, boundaries get redrawn, this year's numbers replace last year's. The naive update — delete the old layer, publish a new one — is also the destructive one, and understanding why teaches you most of what matters here.

Every item in ArcGIS Online has a permanent ID, and every map or app that uses a layer records that ID (and the service URL behind it). Delete the item and publish a replacement, and the replacement gets a *new* ID and a *new* URL. Every map pointing at

the old one is now pointing at nothing. Your styling, pop-up configurations, filters, and view layers all evaporate too, because they belonged to the old item. The cardinal rule of hosted-layer maintenance: **keep the item, replace the data inside it.** There are three honest ways to do that.

Manual and scripted edits

For small, ongoing changes, just edit: through Map Viewer's editing tools, through the item page's data tab, through Field Maps, or through Python (Chapter 31). The layer is a live database; you do not need a ceremony to change ten rows. Chapter 13 covers editing workflows in depth.

Overwrite: replace everything

Overwrite throws away all the layer's current rows and reloads it from a fresh source, while keeping the item, its ID, its URL, its settings, and its dependent views intact. If the layer was published from an uploaded file, look for an `Update Data` action on the item page with an option to overwrite the entire layer; you supply a fresh copy of the file — expect ArcGIS Online to insist it match the original's file name and format, not just its structure — and the service swallows it whole. If the layer was published from ArcGIS Pro, the overwrite runs from Pro instead — and Pro expects to find the original project and source dataset it published from, which is a strong reason to keep that project safe and backed up rather than treating it as disposable scaffolding.

Overwrite is the right tool when the new data is a complete replacement — this year's full parcel roll, the refreshed station list. Its sharp edges:

- **Schema drift breaks things downstream.** Overwriting with a file whose columns have been renamed or removed will ripple outward: pop-ups reference fields that no longer exist, Arcade expressions error, views that exposed a vanished field misbehave. Keep the incoming file's structure identical to the original; change data, not columns.
- **Sync blocks it.** As noted earlier, a sync-enabled layer with outstanding offline copies will refuse to be overwritten. Reconcile or remove the offline copies first.
- **There is no undo.** The old rows are gone the moment the overwrite commits. Export a backup first — the habit described two sections from now.

Watch out: Overwrite replaces *data*, but feature IDs are not guaranteed to survive it. Anything that referenced individual features by their internal ID — a bookmarked selection, a link to a specific feature in an app — may point at a different feature afterward. If stable identity matters, maintain your own ID field in the data and use it, not the system-generated one.

Append: add or update selectively

Append loads new records into the layer without touching what is already there — and, in its more powerful form, can *update* existing records by matching them against incoming rows on a field you designate. That match-and-update behavior is often called **upsert** (update-or-insert): rows that match an existing record update it; rows that match nothing are added. Look for append under the same `Update Data` action on the item page, and expect a field-mapping step where you tell ArcGIS Online which incoming column feeds which layer field.

Append shines for incremental feeds: each week's new inspections added to the running history, or a nightly refresh where records carry a reliable unique ID to upsert against. That unique ID is the linchpin — upsert needs a field whose values genuinely identify one record apiece, which is a schema-design decision (Chapter 12) best made before the first publish.

Choosing among them

SITUATION	RIGHT TOOL	WHY
A handful of corrections	Manual editing	No ceremony needed for small change
Complete periodic replacement, same schema	Overwrite	One step, views and maps survive
New records arriving on a schedule	Append	Existing data untouched

SITUATION	RIGHT TOOL	WHY
Mixed new-and-changed records with a unique ID	Append with upsert	Surgical updates, no data loss
The schema itself must change	Field edits on the source, or careful republish	Data operations cannot restructure a layer

Whatever the path, the discipline is the same: same item, same ID, same URL, forever. A hosted layer that keeps its identity for years while its data turns over weekly is a hosted layer doing its job.

Performance: Keeping Layers Fast

A hosted feature layer with a few hundred features is fast no matter what you do. Growth is what exposes the difference between a layer that was set up thoughtfully and one that was not. When a map feels sluggish, the layer is usually doing one of two expensive things: sending too many features, or sending too much *per* feature. The remedies target both.

Indexes: helping the database find things

Every query your map fires — "features in this box," "features where status is Open" — makes the cloud database search. An **index** is a pre-built lookup structure, like the index at the back of a book, that lets the database jump to matching rows instead of reading every one. Hosted layers maintain a **spatial index** (for the where-in-space questions) automatically. **Attribute indexes** — for the where-status-is-Open questions — are the ones you influence. Fields that get hammered by filters, view definitions, joins, or upsert matching are candidates; some indexes appear automatically when you designate a field as unique or enable certain options, and others can be added through the layer's settings or its administrative interface. If a filtered layer is slow and the filter field is a plausible index candidate, this is the first dial to turn. After massive data changes, rebuilding indexes — surfaced in the layer's settings or through that same administrative interface — can restore speed that has quietly degraded.

Sending less: filters, fields, and scale

The cheapest feature to draw is the one you never request. Three habits, all free:

- **Filter early.** If a map only ever shows active records, put that filter on (or better, consume a view limited to active records) so inactive rows never cross the wire.
- **Trim fields.** Every field you exclude — via a view's field list or the map's configuration — shrinks every response. Layers accumulate fields like drawers accumulate cables; views let you present only the live ones.
- **Set visible scale ranges.** A nationwide dataset has no business drawing all at once when someone is zoomed to the whole country. Set the layer to appear only past a sensible zoom level, and give people a summarizing layer at small scales. Chapter 7 (Styling and Smart Mapping) and Chapter 4 (Cartographic Design) both bear on what to show when.

Optimized drawing and generalization

Polygon layers with intricate boundaries — coastlines, parcels, watersheds — carry enormous geometric detail that is invisible when zoomed out. **Generalization** is the simplification of geometry for display: fewer vertices at small scales, full detail up close. Hosted feature layers offer an option, typically labeled along the lines of *optimize layer drawing* in the layer's settings, that pre-computes generalized versions of large polygon layers so small-scale drawing stops choking on detail nobody can see. It takes a while to build and is worth it for any big, complex polygon layer meant for interactive maps.

Tile layers: the pre-rendered escape hatch

When a layer is very large, essentially static, and mostly serves as visual context rather than something to query, stop serving it as live features and serve it as **tiles** — pre-rendered images (or pre-packaged vector bundles, for a **vector tile layer**) cut into a pyramid of zoom levels, drawn once and cached. Tiles draw almost instantly at any scale because the hard work already happened. You can publish a tile layer *from* a hosted feature layer via the item page's publishing options. The trade-offs: tiles are frozen until you rebuild them, raster tiles lose pop-ups and per-feature interactivity, and building and storing tile caches can consume credits. A common architecture pairs the two — a tile layer for fast display, with the feature layer layered invisibly on top to answer clicks.

Tip: Diagnose before optimizing. The item page's usage tab shows how much the layer is actually being hit, and your browser's developer tools show which requests are slow. A map can also be slow for reasons that have nothing to do with the layer — a heavyweight basemap, a dozen unneeded layers, labels on everything. Measure first; Chapter 39 (The Troubleshooting Encyclopedia) has a slow-map diagnostic sequence.

Export and Backup: Your Data's Safety Net

Esri's cloud is professionally run, but no platform protects you from yourself. The overwrite you ran with the wrong file, the field a colleague deleted, the intern's bulk edit — these are the realistic disasters, and none of them come with an undo button. A deleted *item* may be recoverable for a short window if your organization has a recycle bin available, but data replaced *inside* a live layer never passes through it. Your backup strategy is exports.

From a layer's item page, `Export Data` produces a downloadable copy in your choice of format. For backup purposes, prefer the **file geodatabase** — Esri's native container format — because it preserves everything faithfully: geometry, field types, attachments, domains. Shapefiles, an older format, truncate field names and mangle types; CSV drops geometry beyond simple points; GeoJSON is fine for interchange but looser on types. Export to formats like CSV when a human or another system needs the data; export to file geodatabase when *future you* needs to restore it.

A workable backup discipline for a small operation:

- **Before every overwrite or bulk operation**, export a file geodatabase. Name it with the date. This is the backup you will actually use one day.
- **On a schedule** proportional to how fast the data changes — weekly for active collection layers, quarterly for slow-moving reference data — export and stash the copy somewhere outside ArcGIS Online.
- **For anything mission-critical**, automate it: a short Python script using the ArcGIS API for Python can export every important layer on a timer without a human remembering to. Chapter 31 shows the pattern.

Remember also that the settings around a layer — pop-ups configured in maps, view definitions, symbology — are not in the exported data. They live in the item and the maps. Losing a layer means losing data *plus* configuration, which is one more reason the keep-the-item rule from the update section doubles as a backup strategy.

> **Watch out:** An export permission is not a backup plan. "Someone in the org probably has a copy" fails exactly when you need it. If you cannot point to the file that would restore a layer, and say when it was made, the layer is not backed up.

Storage Costs: What the Meter Is Measuring

ArcGIS Online bills storage in credits, and hosted feature layers are the line item that surprises people — not because any single layer is expensive, but because feature storage is metered continuously and priced differently from ordinary file storage. The full credit system belongs to Chapter 34; here is the layer-owner's working knowledge.

Feature storage costs more per unit than file storage. The same data sitting as an uploaded zip file in your content is cheap; loaded into the live database behind a hosted layer, it costs meaningfully more, because you are paying for a database that answers queries around the clock, not a file at rest. This is fair — the layer is doing work — but it means the tidy instinct to "publish everything, just in case" has a monthly price.

Attachments dominate. Attribute rows and geometries are small; photos are not. A field-collection layer whose features average a couple of photos each will find its storage bill is overwhelmingly a photo bill. Configure field apps to capture reasonable image sizes rather than full-resolution originals (Chapter 30), and periodically archive attachments from closed-out records if the layer is long-lived.

Views are nearly free; copies are not. Because a view stores no data, the master-and-views pattern costs roughly one layer's storage no matter how many views surround it. Duplicated layers, forgotten test publishes, and "backup copies" left as live hosted layers all pay full freight. Your organization's administrative reports can list hosted layers by size — reviewing that list is the fastest housekeeping win available.

A twice-a-year sweep keeps the bill honest: sort your hosted layers by size and by last-used, export-and-delete the ones nothing references, convert big static display layers to

tiles where that fits, and check that attachment-heavy layers still need every photo they carry. Storage is the rare GIS problem that is genuinely boring and genuinely solvable in an afternoon.

The Lifecycle, End to End

The whole chapter compresses to a lifecycle you can run from memory. Publish deliberately: clean schema, sensible field names, coordinates over addresses, source file retained. Configure immediately: editing scoped to what editors truly need, editor tracking on, attachments only if needed, delete protection on before anything depends on the layer. Architect with views: a locked-down master, purpose-built views per audience, maps built on views. Update in place: manual edits, append, or overwrite — never delete-and-republish — with an export taken before anything destructive. Tune as it grows: indexes on filtered fields, trimmed fields, scale ranges, optimized drawing, tiles for the big static stuff. Back up like you mean it, and audit storage twice a year.

Do these things and your hosted layers become what infrastructure should be: invisible. The maps in Chapter 6, the styling in Chapter 7, the pop-ups in Chapter 9, and every app in Volume F all stand on this foundation — and the finest dashboard in the world is only as good as the layer feeding it.
