The ArcGIS Compendium — Volume C: Data Creation and Management

ON THIS PAGE

The ArcGIS Compendium — Volume C: Data Creation and Management

Getting data in and keeping it healthy — every import path, schema design, editing workflows in web and Pro, quality assurance, and imagery.

One of eight volumes. Approximately 24162 words. Chapters cross-reference the whole set by number.

In this volume:

Creating Data: Every Path In

Every layer on every map you have ever seen started life somewhere else: as a spreadsheet on someone's laptop, a list of addresses in a billing system, a paper map in a filing cabinet, a GPS unit bouncing around in a truck, or a person tracing shapes on a screen. Getting that raw material into ArcGIS as usable geographic data is the subject of this chapter. Chapter 5 (Finding Data) covered the case where someone else already did this work and published the result. Here we handle the other case — the data does not exist yet as a GIS layer, and you have to bring it in yourself.

There are six broad paths in, and almost everything you will ever do is one of them:

PATH	YOU START WITH	YOU END WITH
Spreadsheet upload	A CSV or Excel file with coordinates or addresses	A point layer
Geocoding	Addresses or place names	Points placed by a matching engine
File import	A GIS file from another system (shapefile, GeoJSON, KML, GPX, file geodatabase)	The same data, hosted in ArcGIS
Digitizing	Nothing but your eyes and a basemap	Hand-drawn points, lines, or polygons
Georeferencing	A scanned paper map or an aerial photo	An image pinned to real-world coordinates, ready to trace
Derivation	Layers you already have	New layers extracted, dissolved, or merged from them

Each path has its own habits, and each has traps that are invisible until they bite. We will walk all six.

Spreadsheets: the most common front door

More GIS data begins as a spreadsheet than any other form. Someone has a CSV (a comma-separated values file — plain text where each row is a record and commas divide the columns) or an Excel workbook, and the rows describe things that exist somewhere: stores, inspections, complaints, wells, trees. If each row has a location, ArcGIS can turn the table into a point layer.

There are two ways a spreadsheet can carry location, and they lead to different machinery:

- **Coordinates**: columns holding longitude and latitude (or X and Y in some other coordinate system). ArcGIS just plots them. Fast, free, exact — assuming the numbers are right.
- **Addresses or place names**: columns like street, city, and postal code. ArcGIS must *geocode* them — look each one up against a reference database and estimate a point. Powerful, but probabilistic, and in ArcGIS Online it consumes credits (the platform's internal currency for services; see Chapter 34 for how credits work and who pays).

Uploading the file

In ArcGIS Online, the sturdiest route is through your content page: go to `Content > New item`, drag the file in, and choose to add it *and create a hosted feature layer*. That last part matters. You can also drag a small CSV directly onto a map in Map Viewer, but that creates a lightweight, map-embedded copy with a modest size cap and no independent existence — fine for a quick look, wrong for anything you intend to keep, share, or edit. A hosted feature layer (Chapter 10 covers these in depth) is a real, standalone dataset you can reuse across maps and apps.

Excel files bring a few quirks of their own: a workbook can hold many sheets, and you must pick the right one; merged cells, multi-row headers, and title rows above the real header confuse the reader, so the data should start at the top-left with one clean header row. When a workbook misbehaves, saving the one sheet you need as a CSV is often the fastest cure.

During the upload, ArcGIS shows you its guesses: which columns hold location, and what type each field is. Slow down here. This screen is where most spreadsheet problems are caught — or missed.

Field type inference, and why it goes wrong

ArcGIS reads the first chunk of your file and infers a type for each column: integer, decimal number, text, or date. It is guessing, and its guesses fail in predictable ways:

- **Leading zeros vanish.** A column of US ZIP codes like `02116` looks numeric, so it becomes the integer `2116`. The same happens to account numbers, tax parcel IDs, and phone numbers. Any code that merely *looks* like a number should be stored as text — override the type during upload, or in Excel format the column as text before saving.
- **Mixed columns collapse.** If a column is mostly numbers but one row says `N/A`, the inference may flip the whole column to text, or drop the offending values, depending on what it sampled. Clean sentinel values like `N/A`, `--`, and `unknown` out of numeric columns before upload; leave those cells genuinely empty instead.
- **Dates are a minefield.** `03/04/2026` is March 4th in the United States and April 3rd in most of the rest of the world. ArcGIS lets you specify the date format during upload — do it explicitly rather than trusting the guess. The unambiguous ISO format (`2026-03-04`) is the safest thing to put in a file you control.

> **Watch out:** The single most common spreadsheet casualty is the ZIP code column silently losing its leading zeros. Check any identifier column immediately after publishing — open the layer's table and eyeball a few rows. Fixing a field type after publishing usually means recalculating into a new field or republishing, so it is far cheaper to catch at upload.

Locale traps

If your file was made on a computer set to a European or Latin American locale, "CSV" may not mean what ArcGIS expects. Many locales use the comma as the *decimal* separator (`3,14` for pi) and therefore use semicolons to separate columns. A file like that, read as a plain comma-separated file, shatters into garbage — every decimal

comma becomes a column break. Coordinates suffer most: `48,8584` intended as a latitude becomes two meaningless fields.

The fixes are unglamorous but reliable: open the file in a text editor and look at it before uploading; convert decimal commas to periods and delimiters to true commas (a spreadsheet program's save-as options or a find-and-replace will do it); or upload as Excel format instead, which stores numbers as numbers and dodges the delimiter question entirely.

The coordinate path

If your table has longitude and latitude, tell the upload wizard which column is which. Two mistakes account for nearly all failures:

- **Swapped columns.** Longitude is the X (east–west) value, latitude is Y (north–south). A layer whose points land in the ocean off Antarctica, or form a mirror image of the expected pattern, has them reversed. Rule of thumb for sanity checks: latitude can never exceed 90 either direction; in the continental US, longitudes are negative.
- **Wrong coordinate system.** ArcGIS assumes plain longitude/latitude (the WGS84 system your phone's GPS speaks) unless you say otherwise. If your columns hold State Plane feet or UTM meters — big six- or seven-digit numbers — you must identify that coordinate system during upload, or your points will be plotted as if those huge numbers were degrees and land nowhere. Chapter 3 (Coordinate Systems) explains what these systems are and how to identify a mystery one.

Points that plot at exactly zero longitude, zero latitude — a spot in the Atlantic off West Africa that GIS people wryly call Null Island — mean some rows had empty or unreadable coordinates that were read as zeros. Filter them, fix the source rows, and re-upload or edit.

Geocoding in depth

Geocoding is the act of converting a text description of a place — usually a street address — into coordinates. It is done by a *locator*: a matching engine backed by an enormous reference database of streets, address points, postal areas, and place names. In ArcGIS Online the locator is a ready-made service — the ArcGIS World Geocoding Service, which covers most of the world. In ArcGIS Pro a locator can also be a local file,

and organizations with ArcGIS Enterprise (Chapter 35) often build their own locators from local address data, which can beat the world service on home turf.

It matters to understand that geocoding is *matching*, not *measuring*. The locator never knows where your building is; it knows where the reference database says addresses like yours should be, and it interpolates when it lacks an exact entry. The result carries a confidence level and a precision level, and honest work requires you to read both.

Single versus batch

Single geocoding is what the search box on any map does: type an address, get a candidate list, pick one. It is interactive, self-correcting (you see the candidates), and free in most contexts.

Batch geocoding processes a whole table at once — the address path of the spreadsheet upload above, or a standalone operation on an already-published table. It is the workhorse for real datasets and the place where quality discipline matters, because nobody is eyeballing each match.

> **Watch out:** Batch geocoding against the World Geocoding Service that *stores* the results consumes credits in proportion to the number of addresses. For a few hundred rows this is trivial; for hundreds of thousands it is a real cost that your administrator will notice. Know your row count before you press go, and see Chapter 34 for how to check your organization's credit balance.

Address data quality drives everything downstream. Before batch geocoding, spend ten minutes on the table: split or verify that address, city, region, and postal code sit in the fields you will map to the locator's inputs (one messy single-line column geocodes worse than clean components); strip unit numbers and "care of" text into separate columns; fix obvious typos in city names. Every minute here saves ten in rematching.

Reading the results: scores, status, and match type

Each geocoded row comes back with diagnostic fields. The names vary slightly by tool, but three concepts are universal:

- **Score** — a number from 0 to 100 expressing how well the input text matched the reference record. A perfect textual match scores at the top; a match that tolerated a misspelled street name or a missing directional ("N" vs "S") scores lower.
- **Status** — a single letter summarizing the outcome: matched, tied (two or more candidates scored equally well — the locator picked one, but you should not trust it blindly), or unmatched.
- **Match type** (often a field named along the lines of "address type") — *what kind of thing* was matched, which is really a precision statement. A rooftop or parcel-point match puts the dot on the actual property. A street-address match interpolates along the road segment: the reference data says the block runs from 100 to 198, your address is 150, so the point goes roughly halfway — usually close, sometimes not. A street-name match means the house number failed and the point sits at the street's midpoint. A postal-code or city match means only the area was recognized, and the point is the *centroid* — the geometric center — of that whole area, which can be miles from the true location.

That last case is the quiet killer. A geocoded layer can be 100 percent "matched" and still be junk, because a chunk of the rows matched only to a city centroid. Always tabulate the match type field after a batch job and decide, dataset by dataset, what precision your analysis can tolerate. Counting crimes by neighborhood can survive some postal-code-level matches; siting a delivery route cannot.

> **Tip:** Sort your geocoded layer's table by score ascending and by match type, and skim the worst fifty rows before you do anything else with the layer. Five minutes of skimming tells you more about the batch than any summary statistic.

Rematching

No batch job matches everything. The unmatched and suspicious rows go through *rematching* — an interactive review pass. In ArcGIS Pro, the Rematch Addresses pane walks you through unmatched records one at a time: it shows the input address, the candidate list with scores, and a map. You can pick a candidate, edit the address text and search again (fixing the typo the original data carried), or — the escape hatch — click the true location on the map yourself, which records a manual match. In ArcGIS Online the review is less interactive: unmatched records are flagged in the geocoding

results, and the practical remedy is to fix the offending source rows and geocode them again.

Treat rematching as data cleaning, not clicking. When you fix an address during rematch, fix it in the *source system* too if you can, or the same row will fail again next quarter. And when you place a point manually, you are asserting knowledge the locator did not have — make sure you actually have it.

File imports: data arriving from elsewhere

GIS is an old field with many file formats, and sooner or later each of them lands in your inbox. ArcGIS Online will ingest all of the common ones through the same `Content > New item` drag-and-drop, publishing a hosted feature layer; ArcGIS Pro reads most of them directly from disk. What differs is the fidelity of what survives the trip.

FORMAT	WHAT IT IS	WHAT TO WATCH
Shapefile	The decades-old vector standard: one dataset spread across several sibling files	Must be zipped for upload; short field names; missing `.prj` files
GeoJSON	A web-native text format beloved by developers	Assumed to be plain longitude/latitude; attribute types are loose
KML / KMZ	Google Earth's format, styling and all	Attributes often trapped in HTML description blobs
GPX	The GPS exchange format from handheld units and fitness apps	Waypoints, tracks, and routes import as separate things
File geodatabase	Esri's own container format, the highest-fidelity carrier	Must be zipped for upload; can contain many layers at once

Shapefiles

A shapefile is not one file but a family: `.shp` holds the shapes, `.dbf` the attribute table, `.shx` an index, `.prj` the coordinate system definition, and often several more.

To upload one to ArcGIS Online, zip the whole family together — the zip must contain all the siblings of exactly one shapefile, at the top level of the archive, not nested in folders. ArcGIS Pro just opens the `.shp` from a folder connection.

Two chronic shapefile ailments: first, the `.dbf` table format limits field names to about ten characters, so data exported to shapefile arrives with names like `POPULATN_2` and `AVG_HH_INC` — plan to set human-readable field aliases after import (Chapter 9 covers aliases). Second, a shapefile that traveled without its `.prj` file has lost its coordinate system label; ArcGIS will ask you what it is, and guessing wrong scatters the data. Chapter 3 explains how to deduce the right answer. Shapefiles also cannot store true dates-with-times, have a hard size ceiling of roughly two gigabytes, and cannot represent null numbers distinctly from zero — all reasons the format is fading, and all reasons to prefer a file geodatabase when you have the choice of what to receive.

GeoJSON

GeoJSON is plain text, human-readable, and the default currency of web developers and open-data portals. By specification it carries coordinates as plain longitude/latitude, so there is no coordinate system ambiguity — but also no way to carry data in anything else, so if someone hand-rolled a GeoJSON in a projected system it is simply broken. Attribute typing is loose (JSON does not distinguish integers from decimals, and dates are just strings), so double-check inferred field types after publishing, exactly as you would for a CSV.

KML and KMZ

KML is Google Earth's format (KMZ is just KML zipped, often with embedded images). It is a *presentation* format as much as a data format: it carries styling, folder organization, and pop-up content, but frequently stores what should be attribute fields as a single blob of HTML in each feature's description. When you import KML, expect geometry and names to arrive cleanly and expect structured attributes to need rescue — sometimes they survive as fields, sometimes you inherit one giant description column. If the KML is the only source, import it, inspect the table, and be prepared to parse what you find. If the sender has the original data in any other format, ask for that instead.

GPX

GPX files come from GPS receivers, fitness watches, and mobile tracking apps. One file can hold *waypoints* (individual marked points), *tracks* (breadcrumb trails of where the device actually went, as ordered point sequences that import as lines or points-with-timestamps), and *routes* (planned paths). ArcGIS imports these as separate layers or asks which you want. Coordinates are always plain longitude/latitude, and points usually carry timestamps and elevation — which makes GPX tracks lovely raw material for the space-time work in Chapter 20.

File geodatabases

A file geodatabase is Esri's native container: a folder (ending in `.gdb`) holding any number of layers, tables, and relationships, with full fidelity — long field names, true dates, domains and subtypes (Chapter 12), attachments, and no practical size ceiling for ordinary work. It is the best format in which to *receive* data from anyone in the Esri world. To upload one to ArcGIS Online, zip the `.gdb` folder; publishing can bring multiple layers into a single hosted feature layer. In Pro, a file geodatabase is simply the workspace you live in — every project gets one by default.

> **Tip:** When someone offers to send you data and asks what format you want, the ranking is: file geodatabase first, GeoJSON or shapefile second, KML a distant last, and "an Excel file with the addresses" only if nothing spatial exists yet. You will spend your cleanup time in direct proportion to how far down that list you land.

Digitizing: drawing data from scratch

Sometimes there is no file. The data exists only in the world, or in your head, or visibly in a basemap's imagery — and the way in is to draw it. Digitizing means creating features by hand: clicking vertices to build points, lines, and polygons over a reference backdrop.

In Map Viewer

The lightweight route in ArcGIS Online: create an empty hosted feature layer, then draw into it. From your content page, `Content > New item > Feature layer` lets you define an empty layer — choose point, line, or polygon — and you can then add the fields you need from the layer's item page before drawing. Map Viewer also offers a

sketch layer you can draw on directly, but sketches are graphics saved inside one map, not a dataset — fine for annotation, wrong for data. For anything durable you want a true hosted feature layer, for all the reasons in Chapter 10.

With an editable layer in the map, open the `Edit` pane, pick a feature template (a preset that determines what kind of feature you are creating and can pre-fill its attributes), and click the map to place vertices; double-click (or use the finish control) to complete a line or polygon. Fill in the attribute form that appears. Map Viewer offers snapping — the magnetic pull that makes your new vertex grab exactly onto an existing vertex or edge nearby — and tolerable geometry tools, but it is honest to say web digitizing suits modest jobs: tens or hundreds of features, simple shapes. Chapter 13 (Editing Workflows) covers the full editing experience, including doing this work in the field with mobile apps (Chapter 30).

Before you draw your first feature, design the fields. It is *vastly* easier to add well-chosen fields, types, and pick-lists (domains) to an empty layer than to restructure one holding data. Read Chapter 12 (Schema Design) before any digitizing project bigger than an afternoon.

In ArcGIS Pro

Pro is the professional digitizing environment, and for any serious tracing job it earns its keep. The workflow: create a feature class (the geodatabase term for a layer) in your project geodatabase — right-click the geodatabase in the `Catalog` pane, `New > Feature Class`, and walk the wizard through name, geometry type, fields, and coordinate system. Add it to a map, then open the `Edit` ribbon and use `Create Features` to start placing geometry.

What Pro adds over the web experience:

- **Construction tools** beyond click-click-click: right angles, tangent curves, tracing along existing features, exact distances and directions typed at the keyboard.
- **Serious snapping** with per-layer control over what attracts your cursor — ends, vertices, edges, intersections.
- **Feature templates** that pre-fill attributes, so digitizing fifty "residential" parcels does not mean typing "residential" fifty times.
- **Undo that you can trust**, and explicit control over when edits are saved.

Digitizing quality is mostly about two disciplines: zoom in far enough that your clicks mean something (a vertex placed at neighborhood scale can be off by many meters), and use snapping religiously wherever your new features should touch existing ones — gaps and slivers between polygons that were "close enough" by eye become the topology errors that Chapter 14 exists to fix.

> **Tip:** Decide your target scale before you start and stay there. Data digitized at one scale mixed with data digitized at another looks fine zoomed out and ragged zoomed in. Consistency beats precision.

Georeferencing: pinning paper to the planet

A scanned map or an old aerial photograph is just a picture — thousands of pixels with no idea where on Earth they belong. Georeferencing is the process of telling ArcGIS where the picture goes: you identify points that appear both in the image and in the real world, and the software stretches and rotates the image until those points coincide. Once the image is pinned down, it becomes a backdrop you can digitize against — which is how historical maps, utility as-builts, and hand-annotated field sheets become GIS data.

This is ArcGIS Pro territory. The workflow:

1. Add the scanned image to a map that already contains trustworthy reference layers — a basemap, parcel lines, road centerlines. The image will land somewhere arbitrary (often at the map's origin, tiny or enormous).
2. Select the image layer and open the `Imagery > Georeference` tools.
3. Use `Fit to Display` to drag the image into the rough neighborhood, so you can see both it and the reference data.
4. Add **control points**, the heart of the job: click a recognizable spot on the *image* (a road intersection, a building corner, a survey monument), then click the same spot on the *reference layers*. Each pair is one control point. The image warps to honor them.
5. Repeat, spreading points across the whole image — corners and center, not clustered on one side. A handful of well-spread points beats a dozen bunched ones.

6. Watch the **residuals**: the georeference table reports how far each control point misses after the fit, and a summary figure (root-mean-square error) for the whole set. A single point with a residual far larger than its neighbors usually means you misidentified the location — delete and re-place it rather than adding more points to fight it.
7. **Save** the georeferencing when satisfied. This writes the transformation so the image loads in place from now on.

The transformation itself comes in escalating flavors — the simplest shifts, scales, and rotates the image rigidly; higher-order options bend it. Resist the temptation to use an aggressive transformation just to force residuals to zero: a bendy fit that nails your control points can distort everything *between* them. For a decently drawn map, the simple transformations are usually right; save the rubber-sheet-style local adjustments for wrinkled scans and truly warped source material.

Pick control features that do not move: road intersections, rail crossings, building corners, monuments. Do not use tree canopies, shorelines, or river bends — nature relocates them. And remember the source map has its own accuracy ceiling: georeferencing a sketch does not make it a survey. Chapter 15 (Imagery and Rasters) covers what images are as data; the work you do *after* georeferencing — tracing features off the pinned image — is exactly the digitizing described above.

> **Watch out:** A scanned map drawn in an old or local coordinate system may fit your reference data well in one area and drift badly elsewhere, because the *projection* differs, not just the alignment. If residuals grow steadily toward one edge no matter what you do, the mismatch may be the coordinate system itself — see Chapter 3 before adding a twentieth control point.

Deriving data: new layers from old ones

The final path in creates data from data you already have. These operations live in Map Viewer's analysis pane and in Pro's geoprocessing tools (Chapter 22 covers geoprocessing as a discipline; Chapter 16 covers the analytical overlay family). Here we care about the three derivations people reach for constantly when *building datasets*, as opposed to answering analytical questions.

Extract: carving out a subset

Extraction pulls part of a layer into a new layer. Three variants:

- **By attribute**: filter to the rows you want (all hydrants inspected before a date, all parcels zoned commercial) and export the selection to a new layer. In Map Viewer, apply a filter and run the analysis tool that extracts data, or export from the layer's item page; in Pro, select by attributes and right-click the layer, `Data > Export Features`, which honors the selection.
- **By geography (clip)**: cut a layer with a boundary polygon like a cookie cutter, keeping only what falls inside — and *trimming* features that cross the edge. A statewide roads layer clipped to one county becomes a county roads layer, with border-crossing roads cut at the line.
- **By selection on the map**: hand-pick features interactively and export just those.

Extraction is also how you take data *out* of ArcGIS Online: a hosted feature layer's item page offers exports to shapefile, file geodatabase, GeoJSON, and CSV — useful for backups, for sending data onward, and for the round trip into Pro.

Dissolve: merging neighbors into wholes

Dissolve removes internal boundaries between features that share a value. Feed it a counties layer and the state field, and you get a states layer — the county lines melt away. It is the standard way to derive coarser geography from finer, and it can simultaneously summarize attributes (total the population of the counties into each state, take the mean of a rate). Two things to know: dissolve on a field with messy values ("TX" vs "Texas") produces two Texases — clean the field first — and by default all features sharing a value merge into one *multipart* feature (a single record whose shape has several disconnected pieces) even if they are not touching, which you can toggle off when you want each contiguous blob kept separate.

Merge and append: combining like with like

Merge stacks multiple layers of the same geometry type into one new layer — twelve monthly incident layers become one annual layer. Append does the same job but loads records *into an existing layer* rather than creating a new one, which matters when apps, maps, and pop-ups already point at the target. Either way, the whole game is field matching: where the inputs use different field names or types for the same concept,

you must map them to each other, and unmapped fields come through empty. Merging layers whose schemas — their field structures — were designed independently is a data-cleaning project wearing a one-click disguise; reconcile the schemas deliberately (Chapter 12) rather than accepting whatever the default field mapping produces.

> **Tip:** Add a "source" text field before merging and stamp each input's origin into it. Six months later, when one subset of records turns out to be wrong, you will know exactly which contributor to go back to.

Choosing your path

When a new dataset need appears, run down this ladder:

1. **Does it already exist?** Search Living Atlas and open portals first (Chapter 5). The best data creation is none.
2. **Does it exist as a table?** Spreadsheet upload — coordinates if you have them, geocoding if you have addresses, with a quality review either way.
3. **Does it exist as a file?** Import it, preferring high-fidelity formats, and audit fields and coordinate systems on arrival.
4. **Does it exist only on paper or in imagery?** Georeference, then digitize.
5. **Does it exist only in the world?** Digitize from imagery if visible from above; send people with mobile collection apps if not (Chapter 30).
6. **Is it a reshaping of what you have?** Extract, dissolve, or merge.

Whichever door you come through, the arrival checklist is the same: open the attribute table and read it; confirm field types survived; confirm the coordinate system is declared and correct; spot-check features against a basemap; and write down where the data came from in the item's description, because future-you is the next person who will have to evaluate its quality — by the very standards of Chapter 5. Data creation is the one moment when you control everything about a dataset. Spend the control well, and every chapter after this one gets easier.

Schema Design: Fields, Domains, and Relationships

A schema is the blueprint of your data: the list of columns each record will have, what kind of value each column holds, which values are allowed, and how tables connect to each other. It is the least glamorous part of GIS and the part most likely to haunt you. A wobbly color choice takes thirty seconds to fix. A field that stored ZIP codes as numbers — silently amputating the leading zeros from every New England address — can take a weekend to repair after ten thousand records have piled up on top of the mistake.

This chapter is about making those decisions *before* you collect data, when every choice is free. Chapter 11 (Creating Data: Every Path In) covers the mechanics of actually creating layers and tables; Chapter 13 (Editing Workflows in Web and Pro) covers filling them with records. Here we stay one step earlier: deciding what the container should look like so that everything downstream — pop-ups, field apps, dashboards, analysis — clicks into place instead of fighting you.

Schema Is a Contract You Sign Early

Think of a schema as a contract between you and everyone who will ever touch this data: field crews entering records on phones, colleagues building dashboards, an analyst three years from now who inherits the layer with no documentation. The contract says "the `status` field will always contain one of these five values" or "every inspection will point back at exactly one hydrant." Every promise you encode into the schema is a promise nobody has to enforce by hand later.

The asymmetry to internalize: **schema changes are cheap before data exists and expensive after.** Before you collect a single record, renaming a field is a two-second edit. After six months of collection, that same rename can break every map, pop-up, label expression, dashboard widget, and integration that referenced the old name. The last section of this chapter deals with changing schemas after the fact — but the whole point of this chapter is to make that section one you rarely need.

A practical habit: sketch your schema in a spreadsheet or plain document first. One row per field, with columns for name, alias, type, domain, default, and a one-line note on why it exists. If you cannot explain why a field exists, delete it from the sketch. This

document later becomes your data dictionary, which Chapter 14 (Data Quality: Validation, Topology, and QA) will thank you for.

Field Types: Choosing the Right Container

Every field has a type, and the type is a promise about what the field can hold. Choose the wrong one and you either lose information (decimals truncated into whole numbers) or lose capability (dates stored as text cannot drive a time slider).

The core types

TYPE	HOLDS	REACH FOR IT WHEN
Text (string)	Letters, numbers, symbols — up to a maximum length you set	Names, descriptions, codes, IDs, anything you will never do math on
Short integer	Small whole numbers (roughly plus or minus thirty-two thousand)	Counts that stay small, category codes, subtype fields
Long integer	Whole numbers up to about plus or minus two billion	Counts, years, whole-number measurements, ID numbers you do math or joins on
Double	Decimal numbers with high precision	Measurements, money, coordinates, percentages — the default choice for decimals
Float	Decimal numbers with lower precision	Rarely; only when storage is genuinely scarce
Date	A calendar date and time of day together	Anything time-related: inspection dates, install dates, observation timestamps
GUID	A globally unique identifier you populate yourself	Foreign keys that point at another table's GlobalID (more on this below)

TYPE	HOLDS	REACH FOR IT WHEN
GlobalID	A globally unique identifier the system maintains for you	You don't choose this per record — you enable it per layer, and you should
ObjectID	The system's own row number	Automatic on every layer; you never create or edit it

A few types you may meet in ArcGIS Pro but will rarely need — blob (arbitrary binary data) and raster (an image stored per row) — exist mostly for specialized workflows inside a geodatabase (Esri's database container for layers and tables; Chapter 11 introduces the flavors). Recent versions have also added more precise date variants (date-only, time-only, and timestamp-with-offset types) for cases where storing a time of day with a date is wrong or ambiguous; if your project involves records spanning time zones, look into these when you build the layer.

Rules of thumb that prevent regret

If you will do math on it, store it as a number; otherwise store it as text. This sounds obvious until you meet identifiers that happen to look numeric. ZIP codes, phone numbers, parcel numbers, asset tags — these are labels, not quantities. You will never compute the average ZIP code. Store them as text.

> **Watch out:** Numeric fields silently discard leading zeros. Store the ZIP code 02114 in an integer field and it becomes 2114 — a different place. This is one of the most common irreversible schema mistakes, because by the time anyone notices, the original zeros are gone from every record. Identifiers go in text fields, always.

Store dates as the date type, never as text. A text field containing "3/7/2026" is ambiguous (March 7 or July 3?), unsortable in a meaningful way, and invisible to everything that understands time — filters like "last 30 days," time sliders, the space-time tools in Chapter 20, and date math in Arcade, ArcGIS's built-in expression language (Chapter 8). A real date field gets all of that for free.

Default to double for decimals. Float saves a little storage in exchange for rounding surprises — values like 3.7 quietly stored as 3.6999998. Unless you have millions of

records and a genuine storage problem, double is the safe choice.

There is no true yes/no type. ArcGIS has no checkbox field. The convention is a short integer or text field paired with a coded value domain — Yes/No, or 1/0 — which gives editors a two-item dropdown and gives you clean data. Domains are the next section.

Pick text lengths generously but deliberately. Every text field has a maximum length; a few hundred characters is a common default. Lengthening a field after data exists ranges from awkward to impossible depending on where the data lives, so lean roomy. But don't make every field enormous "just in case" — a 10,000-character `notes` field invites editors to bury important facts in prose that no filter or symbol can ever use. If a fact matters, it deserves its own field.

Field names and aliases

Every field has two labels. The **name** is the permanent internal identifier — what Arcade expressions, filters, integrations, and code refer to. The **alias** is the friendly display label that appears in pop-ups, tables, and editing forms. The name is nearly frozen the moment anything references it; the alias you can change anytime without breaking a thing.

That split dictates the strategy: make names terse, boring, and machine-friendly, and pour all your readability into the alias.

For names, follow the old-database conventions, because your data will eventually pass through something that enforces them:

- Lowercase letters, digits, and underscores only: `install_date`, `pipe_dia_mm`.
- Start with a letter, never a digit.
- No spaces, hyphens, or punctuation.
- Avoid words that databases reserve for their own grammar — `date`, `order`, `zone`, `desc` and the like can cause baffling errors in some systems. `install_date` is safer than `date`.
- Keep them reasonably short. If your data might ever be exported to a shapefile — an elderly format that still circulates widely — names get truncated to about ten characters, and `inspection_result_code` and `inspection_result_notes` may collide into gibberish. See Chapter 11 for more on format quirks.

- Put units in the name when a number has them: `length_m` and `depth_ft` prevent the classic disaster of metric and imperial values mixed in one column.

Then set aliases like "Installation Date" and "Pipe Diameter (mm)" so humans never see the underscores. Chapter 9 (Pop-ups, Fields, and Labels) covers how aliases flow into display.

> **Tip:** When you can't decide on a name, imagine typing it inside an Arcade expression fifty times. `$feature.install_date` is pleasant. `$feature.Date_of_Original_Installation__1` is a curse you cast on your future self.

Defaults and Required Fields

A **default value** is what a field starts with when someone creates a new record. Defaults are the cheapest data-quality tool you have, especially for field collection: if 90 percent of new hydrants are in "Active" status, defaulting `status` to Active means crews only touch the field for exceptions. Every tap you remove from a field form is a tap that can't be fumbled in the rain. Chapter 30 (Field Operations) leans heavily on this.

Set defaults where a value is genuinely typical — condition ratings, ownership, status. Do *not* set defaults where the value must be observed — defaulting `pipe_material` to PVC guarantees that lazy or rushed edits produce plausible-looking lies. A blank field is honest; a defaulted wrong value is camouflage.

Closely related is whether a field may be left empty. A field that **allows null values** can hold "no answer"; a **required** (non-nullable) field must have a value before the record saves. Null is not zero and not an empty string — it is the explicit absence of information, and it's usually what you want for "we don't know yet." Reserve required fields for the small set of facts without which a record is meaningless — an inspection without a date is not an inspection. Make too many fields required and field crews will satisfy the form with junk values just to save the record, which is worse than blanks. You can also enforce "please fill this in" more gently at the form level in Field Maps and Survey123, the mobile data-collection apps Chapter 30 covers.

Domains: Replacing Free Text with Menus

Left to type freely, ten editors will spell one concept ten ways: "Good", "good", "GOOD", "Gd", "Good condition", "goood". Every variant is a different value to a filter, a different slice in a chart, a different color in a map. A **domain** is a rule attached to a field that says "only these values are allowed," and it converts that free-text chaos into a dropdown menu.

Coded value domains

A **coded value domain** is a fixed list of allowed values, each with two parts: a **code** (what is actually stored in the database) and a **description** (what humans see in forms, pop-ups, and legends). A pipe-material domain might store the code `PVC` and display "PVC — Polyvinyl Chloride"; a condition domain might store `1` and display "Good". Editors pick from a menu; the database stores the code; every map and app translates it back to the description automatically.

Codes can be text or numbers — the one constraint is that the code's type must match the field's type, so text codes attach to text fields and numeric codes to numeric fields. Short text codes (`PVC`, `DI`, `CU`) have the advantage of being legible when you meet the raw data in an export or a script. The critical property is that **codes are forever and descriptions are editable**. You can rewrite a description ("Fair" becomes "Fair — minor defects") anytime without touching stored data, because the stored value is the code. Changing a *code* after data exists, though, means the old stored values no longer match anything in the list — so choose codes you can live with.

> **Tip:** Design every coded domain to be exhaustive, including the awkward cases. Add an `UNK` (Unknown) and, sparingly, an `OTH` (Other) option. If the menu has no honest choice, editors will pick a dishonest one, and you'll never be able to tell which records are real. If "Other" starts winning elections, that's your signal the list needs new members.

Range domains

A **range domain** applies to numeric fields and sets a minimum and maximum: pipe diameter between 50 and 600, condition score between 0 and 100, depth that cannot be negative. Range domains catch the typo class of error — the extra zero that turns a

40-millimeter pipe into a 400-millimeter one. They cannot catch a wrong-but-plausible value; that's what the validation rules in Chapter 14 are for.

Where domains live, and designing ones that age well

In a geodatabase managed through ArcGIS Pro, domains are defined once at the geodatabase level and can be assigned to any number of fields across any number of feature classes (geodatabase-speak for layers) — define "Condition" once, reuse it everywhere, and all of them stay consistent. With a layer from that geodatabase selected, the editor lives in the ribbon's data design views (`Data > Design > Domains`; the same group holds the Fields and Subtypes views). For a **hosted feature layer** — a layer that lives in ArcGIS Online or Enterprise rather than a local geodatabase (Chapter 10) — each field carries its own list of allowed values, editable on the layer's item page (open the `Data` tab, switch to `Fields`, and pick the field). Layers published from Pro carry their domains with them.

Three habits make domains age well. First, one fact per domain: don't create values like "PVC — Poor Condition" that fuse material and condition into one field; that combination belongs in two fields. Second, keep lists short enough to scan on a phone — beyond a couple dozen entries, consider whether you're really encoding two facts (a category and a subcategory) that deserve two fields. Third, write down what each code means in your data dictionary, because "condition 3" will spark a meeting-room argument within a year if you don't.

Subtypes: One Layer with Several Personalities

Sometimes the records in a single layer fall into a few major flavors that behave differently. A water-mains layer contains transmission mains and distribution mains; a signs layer contains stop signs, speed-limit signs, and street-name signs. They belong in one layer — same geometry, same core fields, mapped and analyzed together — but the flavors want different rules: a transmission main has different typical diameters and a different set of plausible materials than a distribution main.

A **subtype** handles exactly this. You designate one integer field as the subtype field, define a short list of subtypes (each is a code plus a name), and then — this is the payoff — each subtype can carry its **own default values and its own domains for the other fields**. Set the subtype to "Distribution," and the diameter default and the

material dropdown are the distribution versions; switch to "Transmission" and both change. Subtypes also give you automatic symbology grouping and separate editing templates per flavor, which Chapter 13 puts to work.

Subtypes are created in ArcGIS Pro's data design views (`Data > Design > Subtypes`), and layers published with subtypes keep them in web maps and Field Maps. They are the middle option on a spectrum:

APPROACH	USE WHEN	COST
One field with a coded domain	Categories are just labels — every record behaves the same otherwise	Nearly free; start here
Subtypes	Categories need different defaults or different allowed values in *other* fields	Moderate setup in Pro; must be designed before heavy collection
Separate layers	"Categories" have different geometry types or mostly different fields	Most flexible, but everything is now duplicated: symbology, forms, permissions

The honest guidance: most projects never need subtypes, and reaching for them prematurely adds complexity you'll pay for every time you touch the schema. Use a plain domain field until you catch yourself wishing the dropdowns changed depending on the category — that wish is the precise symptom subtypes cure.

Related Tables and Relationship Classes

Every schema question so far has lived inside one table. The most important structural decision in schema design is recognizing when one table isn't enough.

The parent-child pattern

Consider hydrant inspections. Each hydrant is inspected roughly yearly, forever. The beginner's instinct is to widen the hydrant layer: `last_inspection_date` , `last_inspection_result` , maybe `insp_2025_date` , `insp_2026_date` ... This collapses

immediately. You either overwrite history every year or grow an unbounded parade of columns, most empty, none filterable.

The durable answer is two tables. The **parent** is the hydrants layer: one record per physical hydrant, holding the facts that describe the hydrant itself. The **child** is an inspections table — not a map layer at all, just a table — holding one record per inspection event: the date, the result, the inspector, and a pointer back to which hydrant it belongs to. One hydrant, many inspections, and the count grows without ever touching the schema again. Any "one thing, many events" situation fits this pattern: work orders per asset, water samples per well, sightings per nest, service calls per address.

Keys: how records find each other

The pointer works through a pair of fields called keys. The parent has a **primary key** — a field whose value uniquely identifies each record. The child has a **foreign key** — a field that stores the parent's key value, declaring "I belong to that one." A **relationship class** (the geodatabase term; hosted feature layers just call the result a *relationship*) is the formal object that records this pairing so that ArcGIS treats it as structure rather than coincidence.

The best practice, stated bluntly: **use the parent's GlobalID as the primary key and a GUID field on the child as the foreign key.** The reasons live in the GlobalID section below, but the short version is that GlobalIDs are the only identifiers guaranteed to be unique and permanent. Whatever you do, never build a relationship on ObjectID — those numbers get reshuffled when data is copied or exported, which orphans every child record.

Cardinality and delete behavior

Two settings define a relationship's personality. **Cardinality** is how many records can pair up: one-to-one (each parcel has one deed record), one-to-many (each hydrant has many inspections — by far the most common), or many-to-many (each inspector covers many hydrants and each hydrant sees many inspectors — possible but rarely worth the complexity in a hosted setting).

The second setting is what happens when a parent is deleted. In a **simple** relationship, deleting the hydrant leaves its inspection records behind as orphans. In a **composite**

relationship, deleting the parent deletes its children too. Composite sounds tidier, and for attachments-style data it is — but think hard before making inspection history composite, since "we removed the hydrant" and "we never inspected anything at that location" are very different claims. Regulatory and audit data usually argues for keeping history.

Building and using relationships

You create relationship classes in ArcGIS Pro (in the Catalog pane, right-click the geodatabase: `New > Relationship Class`, then walk through the wizard choosing parent, child, keys, and cardinality). When you publish the layer and table together as one hosted feature layer, the relationship comes along. From there the payoff spreads through the whole platform: Map Viewer pop-ups can show a hydrant's related inspections and let you browse them (Chapter 9); Field Maps lets a crew tap a hydrant and add a new inspection record on the spot (Chapter 30); Survey123 surveys with repeating sections publish as exactly this parent-child shape. Chapter 10 covers what relationships mean for layer views and performance.

Attachments

An **attachment** is a file — almost always a photo, sometimes a PDF or short video — stored with a specific feature. Enable attachments on the inspections table and every inspection can carry the pictures that prove its findings.

Under the hood, attachments are the relationship pattern you just learned, prebuilt: enabling attachments creates a hidden child table keyed to your layer's GlobalIDs, one row per file — which is why GlobalIDs must exist first (the tools add them if needed). You can enable attachments when creating the layer in Pro (right-click the feature class in the Catalog pane and look under `Manage`, or run the Enable Attachments geoprocessing tool) or afterward on a hosted feature layer's item page settings. Two practical notes. First, decide *which* table gets attachments deliberately — photos of an inspection belong on the inspection record, not the hydrant, so each photo is tied to the visit that produced it. Second, photos dominate storage: a hosted layer whose attribute data is trivially small can carry gigabytes of images, and hosted storage consumes credits (Chapter 34 covers the credit model). It's worth configuring your field apps to capture reasonably sized photos rather than full-resolution originals.

GlobalIDs and Editor Tracking

Two switches deserve to be flipped on essentially every serious layer, ideally at creation.

ObjectID versus GlobalID

Every layer has an **ObjectID**: a plain integer the system assigns each row. It's fine for casual reference, but it is only unique within that one table, and — the trap — it is *not stable*. Export the layer, copy it to another geodatabase, or rebuild it, and the ObjectIDs can be reassigned. Anything that depended on them now points at the wrong records.

A **GlobalID** is a system-maintained identifier of a different breed: a long generated string (a GUID, or globally unique identifier — something like `{7A9C0E51-...}`) that is unique across every table everywhere and never changes for the life of the record, surviving exports and copies. GlobalIDs cost you nothing to enable (in Pro: right-click the feature class in the Catalog pane, `Manage > Add Global IDs`; hosted layers created in ArcGIS Online generally have them from birth) and they quietly unlock the grown-up workflows: attachments, relationships built on stable keys, and offline sync in Field Maps, which needs a reliable way to match records edited on a disconnected phone back to the master copy.

Editor tracking

Editor tracking adds four system-maintained fields to a layer — who created each record and when, and who last edited it and when — and keeps them updated automatically. Nobody types into them; the platform stamps them on every save. Enable it in a hosted layer's item page settings (the option about keeping track of who created and updated features) or in Pro under the feature class's `Manage` menu.

The payoff is accountability and operational awareness: filter to "records created this week" for a QA sweep, build a dashboard of inspections per crew per day (Chapter 28), or answer "who changed this value?" without an interrogation. Editor tracking is also the raw material for the ownership-based editing controls in Chapter 34, where a layer can be configured so field users see and edit only their own records.

> **Tip:** Enable GlobalIDs and editor tracking on day one even if you have no immediate use for them. Both are free, invisible until needed, and dramatically

easier to have from birth than to wish for after a year of collection — editor tracking can only stamp edits it was awake for.

Changing a Schema After Data Exists

Despite your best design, reality will eventually demand a change. The changes divide sharply into safe and hazardous.

The safe list

These can be done almost anytime with little risk:

- **Adding a new field.** Existing records get null in the new field; nothing downstream breaks. This is the workhorse of schema evolution.
- **Adding new codes to a coded value domain.** The menu grows; existing data is untouched.
- **Editing domain descriptions and field aliases.** Pure display changes.
- **Adding a new related table**, enabling attachments, or turning on editor tracking.

The hazardous list

- **Renaming a field.** The name is what every dependent thing references: Arcade expressions, label classes, filters, dashboard widgets, Experience Builder configurations, Python scripts, outside integrations. Rename `status` to `op_status` and each of those references silently fails or errors. In hosted feature layers a true in-place rename often isn't even offered — which is the platform telling you something.
- **Changing a field's type.** Effectively impossible in place. Text cannot become a date by decree; the cure is the migration pattern below.
- **Deleting a field.** Instant and irreversible — the data in it is gone, and anything referencing it breaks exactly as with a rename. Export a backup first, every time.
- **Removing or repurposing domain codes.** Removing a code from a domain does *not* touch records already storing it; those records now hold an orphaned value that displays as a raw code and fails validation. Worse is quietly changing what a code *means* — records stored under the old meaning become retroactive lies. Retire codes; never recycle them.

- **Shortening a text field or tightening a range domain** when existing values exceed the new limit.

> **Watch out:** With hosted feature layers, the blast radius of a schema change extends past the layer itself. Views built on the layer inherit its schema (Chapter 10), and every map and app configured against it holds references to field names. ArcGIS gives you no complete built-in listing of everything that uses a layer, so before any hazardous change, inventory the dependents yourself — check the web maps you know about, search your organization's content, ask around (admin scripts can automate the hunt) — and schedule the change when editors are offline.

The add-calculate-verify-retire pattern

When you truly must rename or retype a field with data in it, don't fight the platform — migrate:

1. **Export a full backup** of the layer (a file geodatabase export from the item page or Pro) so no mistake is fatal.
2. **Add** a new field with the correct name and type.
3. **Calculate** the new field from the old one — Calculate Field in Pro or the field calculator on the item page's table, converting values as needed (parsing text into real dates, for example).
4. **Verify**: sort by the new field, check nulls, spot-check rows against the old values, confirm counts match.
5. **Repoint** everything that referenced the old field — pop-ups, labels, symbology, filters, app widgets, scripts.
6. **Retire** the old field: rename its alias to something like "OLD — do not use" for a probation period, and delete it only when nothing has missed it for a few weeks.

Slower than a rename would be, but it converts a silent, spread-out breakage into one controlled, reversible operation.

A Worked Miniature: Hydrants and Their Inspections

Here is the whole chapter compressed into one small, sound design — a parent layer, a child table, and every concept above earning its keep.

Hydrants (point layer — the parent). GlobalIDs enabled, editor tracking on, attachments off (photos belong to inspections):

FIELD NAME	ALIAS	TYPE	RULES
`asset_id`	Hydrant ID	Text	Required; the ID painted on the hydrant — text because it's a label
`status`	Status	Text	Coded domain: Active / Inactive / Removed; default Active
`hyd_type`	Hydrant Type	Text	Coded domain of models in use, plus Unknown
`install_date`	Installation Date	Date	Null allowed — often genuinely unknown
`pressure_zone`	Pressure Zone	Text	Coded domain; note `zone` alone is a risky reserved-ish name

Hydrant Inspections (standalone table — the child). GlobalIDs and editor tracking on, attachments **on**; related to Hydrants one-to-many via Hydrants' GlobalID → `hydrant_guid`:

FIELD NAME	ALIAS	TYPE	RULES
`hydrant_guid`	Hydrant	GUID	Foreign key; Field Maps fills it automatically when a crew adds an inspection from the hydrant
`insp_date`	Inspection Date	Date	Required — an undated inspection is meaningless
`result`	Result	Text	Coded domain: Pass / Fail / Needs Follow-up; **no default** — must be observed

FIELD NAME	ALIAS	TYPE	RULES
`flow_lpm`	Flow (L/min)	Double	Range domain keeping values physically plausible; units in the name
`notes`	Notes	Text	A few hundred characters; the escape valve for everything the domains can't say

Notice the pattern of the choices: identifiers as text, observations without defaults, one fact per field, units in names, history in a child table that can grow forever without a schema change. A crew member taps a hydrant in Field Maps, adds an inspection, snaps two photos, and every design decision above is silently doing its job.

Where the Schema Tools Live

In **ArcGIS Pro**, the data design views are the professional's schema workbench: with a layer selected, the ribbon's `Data > Design` group opens the `Fields`, `Domains`, and `Subtypes` views, and relationship classes are created from the geodatabase in the Catalog pane. Design in a local file geodatabase, then publish (Chapter 10). For **hosted feature layers**, the item page's `Data` tab handles fields and value lists, and the settings page handles attachments and editor tracking — sufficient for simple schemas, but subtypes and relationships still want Pro. **Survey123** deserves a mention as schema design in disguise: every question you add to a form becomes a field, every choice list becomes a domain, and every repeating section becomes a related table (Chapter 30) — the same design principles apply, just wearing a form-builder costume.

However you build it, spend the extra afternoon on the sketch before the first record lands. Schema design is the rare part of GIS where an hour of thinking reliably saves a week of repair.

Editing Workflows in Web and Pro

Every map you have ever admired was edited into existence. Someone drew those parcel boundaries, corrected those street names, split that field into two when the farm changed hands. Editing is the verb of GIS — and it is also where most real damage happens, because unlike styling a map (which you can always restyle) or running an analysis (which you can always rerun), editing changes the data itself. Get it wrong with three people editing at once and you can quietly destroy weeks of work.

This chapter covers the two places you will edit in the ArcGIS world: the web browser, using Map Viewer against hosted feature layers, and ArcGIS Pro, the desktop application, against layers of nearly any origin. The two environments share concepts — templates, snapping, attribute forms — but they have very different ideas about what "saved" means and very different tools for reshaping geometry. Then we tackle the harder problem: what happens when you are not the only editor, and how organizations keep multiple people from stepping on each other's changes.

If you have not yet created a layer to edit, Chapter 11 (Creating Data: Every Path In) covers publishing your first editable layer, and Chapter 12 (Schema Design) covers designing the fields and rules that make editing safe. This chapter assumes an editable layer already exists and focuses on the act of editing it well.

Two Editing Worlds, One Set of Data

A quick orientation before the details. **Web editing** happens in Map Viewer (or in apps built on top of it — Field Maps, Experience Builder, and friends) and works against *hosted feature layers*: data that lives in ArcGIS Online or ArcGIS Enterprise and is served over the web. Every edit you make travels over the internet and is applied to the layer immediately. **Pro editing** happens in ArcGIS Pro on your desktop and can target hosted layers, files on your machine, or enterprise databases. Pro holds your edits in a pending state until you deliberately save them.

QUESTION	WEB (MAP VIEWER)	ARCGIS PRO
What can it edit?	Hosted feature layers with editing enabled	Almost anything: geodatabases, shapefiles, hosted layers, enterprise databases

QUESTION	WEB (MAP VIEWER)	ARCGIS PRO
When do edits become real?	Immediately, one feature at a time	When you click Save; until then they are pending
Geometry tools	Basic: draw, move vertices, reshape in recent versions	Deep: reshape, split, merge, align, trace, topology-aware editing
Attribute entry	Configurable forms with validation and logic	Attribute pane, tables, forms
Best for	Simple updates, distributed teams, non-GIS staff	Heavy geometry work, bulk edits, quality-controlled production
Undo depth	Shallow — the current sketch and recent actions	Full undo stack until you save

The practical rule: use the web for *many people making small edits*, and Pro for *few people making serious edits*. Most organizations use both, often against the same layers.

Editing in the Browser

Web editing is deceptively simple on the surface — click, draw, type, done — but there is a configuration layer underneath that determines who can do what, and an important mental model about saving that trips up almost everyone coming from desktop software.

Turning editing on

A hosted feature layer is not editable until its owner says so. Editing is a property of the *layer item*, not of any particular map. You will find the switch in the layer's item page settings — open the layer's item in your content, go to its settings, and look for the feature layer editing section. There you control not just whether editing is allowed, but what kind:

- **What editors can do**: add features only, update only, delete, or full add/update/delete.

- **What editors can see and touch**: everyone's features, or only features they themselves created. That second option — often called ownership-based access — is the backbone of many field data collection workflows, and it means two field workers can share one layer without ever being able to damage each other's records.
- **Whether attributes only or geometry too**: you can allow people to fix a typo in a name field while forbidding them from dragging the point somewhere else.

These settings apply to the layer everywhere it appears. If you need one audience to edit and another to only view, do not make two copies of the data — make a *view layer* with different settings, which is exactly what Chapter 10 (Hosted Feature Layers) is about.

> **Tip:** Before enabling editing on any layer that matters, turn on the option to keep track of who created and updated features (editor tracking), and consider enabling the setting that lets you export data. Editing without editor tracking is like lending your car without asking who is driving.

The Edit panel

In Map Viewer, open the map containing your editable layer and look for `Edit` in the toolbar (on the right side, with the map settings tools). This opens the Editor panel, which has two personalities:

1. **Create**: a gallery of *feature templates* — one entry per layer (or per category within a layer) that you can add to. Click a template, then click or sketch on the map to place a new feature.
2. **Select and edit**: click an existing feature on the map and the panel switches to showing its attributes and geometry handles so you can change it, move it, or delete it.

If the Edit button is missing or greyed out, the diagnosis is almost always one of three things: the layer does not have editing enabled (fix it in the item settings), you do not have edit privileges in your organization role (an administrator question — see Chapter 34), or the layer in this map has been set to read-only in the map itself. Map Viewer lets a map author disable editing per layer within the map even when the underlying layer allows it — a useful safety for maps meant for browsing.

Feature templates

A template is a pre-packaged answer to the question "what kind of thing are you about to create?" For a simple layer, there is one template per layer. For a layer styled by category — say, a hydrant layer symbolized by hydrant type — Map Viewer typically offers one template per category, and choosing the "dry barrel" template pre-fills the type field for you.

This matters more than it looks. Every attribute a template pre-fills is an attribute nobody has to remember to type, and therefore an attribute that cannot be typo'd. If your editors constantly create features and then set the same three fields to the same three values, that is a sign your templates (or your schema's default values — Chapter 12) should be doing that work.

Snapping

Snapping means the cursor gravitates to existing geometry — vertices (the corner points that define a shape), edges, endpoints — when you get close, so the point you place lands *exactly* on the line you meant, not a hair's width away. That hair's width is invisible at your current zoom and glaring when someone zooms in, and it wrecks any analysis that depends on things actually touching (see Chapter 16 on overlay).

Map Viewer has snapping controls in the editing interface — look for a snapping settings control in the Editor panel. You can toggle snapping on and off and choose which layers act as snapping targets. When it is on, watch for the small cue that appears as your cursor locks onto geometry: that is your confirmation the connection will be exact.

> **Tip:** Turn snapping *off* when digitizing features that genuinely should not connect to anything — scattered tree points, for example. Snapping that is helpfully grabbing a nearby sidewalk edge while you are trying to place trees is worse than no snapping at all.

Forms: guided attribute entry

When an editor creates or selects a feature, the attribute entry experience is controlled by the layer's *form* — a configurable arrangement of fields that the map author designs

in Map Viewer (look for the forms configuration when a layer is selected in the map). A well-built form is the difference between clean data and chaos:

- **Order and grouping**: put the fields people fill in first at the top; collapse rarely-used fields into groups.
- **Required fields**: mark a field required and the form will not accept the submission without it.
- **Choice lists**: if a field has a domain (a fixed list of allowed values, from Chapter 12), the form presents a dropdown instead of a free text box.
- **Conditional visibility and calculated values**: using expressions written in Arcade, ArcGIS's built-in expression language (Chapter 8), you can hide the "date closed" field until status is set to "Closed," or automatically calculate a field from others.

The form you configure here follows the layer into other apps — most importantly Field Maps (Chapter 30), where the same form appears on a phone. Design it once, carefully.

What "saved" means on the web — and undo semantics

Here is the mental model shift: **web editing has no Save button.** The moment you finish creating a feature — you complete the sketch and confirm — that feature is written to the hosted layer. The moment you press delete and confirm, the feature is gone from the layer, for everyone, everywhere that layer appears. There is no pending state, no "discard my session," no going home for the night and deciding tomorrow.

Undo exists, but it is shallow and local. While you are actively sketching a geometry, you can undo vertex by vertex, and the editor interface offers undo/redo for recent actions within your current editing flow. But this is not the deep, session-long undo stack that desktop software trains you to expect. Once you have moved on — closed the panel, refreshed the page, edited something else — reversing a change means editing it back by hand.

> **Watch out:** Deleting a feature in web editing is effectively permanent. There is no recycle bin for deleted features. Recovery depends on preparation someone did earlier — an export made before the damage, or backups an administrator keeps — so as an editor you should treat every delete as final. If you are unsure, don't

delete: flag the feature with a status field and let someone with authority do the deleting.

This immediacy is also web editing's superpower: the instant a field worker submits a repair record, the office dashboard updates. There is no sync step, no end-of-day upload. You are trading the safety of a pending state for the immediacy of a live one. Just make sure everyone editing understands the trade.

Editing in ArcGIS Pro

Pro is where editing grows teeth. The geometry tools are an order of magnitude deeper, the undo stack is real, and edits are held in a pending state until you commit them. If web editing is a pen writing directly on the record, Pro editing is a draft you review before filing.

Chapter 21 covers Pro's interface in general; here we assume you can open a project and add layers, and we focus on the editing machinery.

Edit sessions: pending until saved

In Pro, when you modify a feature — move it, reshape it, change an attribute — the change exists only in your current editing state. The `Edit` ribbon tab shows `Save` and `Discard` buttons: `Save` commits everything you have done to the underlying data source; `Discard` throws all of it away and returns the data to its last saved state. Until you save, you also have a full undo/redo stack (the undo button in the quick access area, or the standard keyboard shortcut), letting you step backward through individual edits.

By default, recent versions of Pro let you simply start editing — click a tool and go — without a ceremonial "begin edit session" step. If you prefer a deliberate gate (many production shops do), Pro's editing options include a setting to require explicitly enabling and disabling editing from the ribbon, which restores the classic discipline of *choosing* to enter an editing state. You can also make individual layers non-editable within your map via the layer list's editing status controls, which is cheap insurance against fat-fingering a layer you only meant to look at.

Tip: Save your edits at meaningful milestones — after finishing a block of parcels, after completing one street's worth of addresses — not reflexively every thirty seconds. Each save is a commit point you can no longer undo past. Frequent tiny saves throw away your safety net; hour-long marathons risk losing work to a crash. Think in paragraphs, not keystrokes.

The Create Features pane

To make new features, go to `Edit > Create` (in the Features group of the Edit ribbon). The Create Features pane appears, listing feature templates for every editable layer in the map — the same template concept as the web, but richer. Each template carries default attribute values and a default construction tool, and clicking a template reveals a small palette of construction tools suited to the geometry: for lines and polygons you get straight segments, curves, right-angle modes, freehand, and often a trace tool that follows existing geometry; for points, click-to-place or place-at-coordinates.

Two habits worth building from day one:

- **Fill attributes at creation time.** The active template panel lets you set attribute values *before* placing the feature. Set them there and every feature you place inherits them — far better than creating twenty features and then attributing them one by one.
- **Learn the trace tool.** When your new polygon must share an edge with an existing one — a new parcel carved from an old block, a district that follows a road — tracing the existing geometry guarantees a perfect shared boundary. Digitizing it by eye guarantees slivers — the skinny accidental gap-and-overlap polygons that appear where two boundaries almost, but not quite, coincide.

Snapping in Pro

Pro's snapping is more configurable than the web's. The snapping controls live on the Edit ribbon (a snapping dropdown) and let you toggle individual *snap agents* — vertex, edge, endpoint, intersection, midpoint, and more — plus set the snap tolerance (how close, in screen pixels, the cursor must get before it snaps). You can also control which layers participate through the snapping settings.

The skill is not turning snapping on; it is knowing which agents you need for the task. Digitizing water lines that must connect end-to-end? Endpoint snapping is essential; midpoint snapping is noise. Splitting parcels along an existing lot line? Edge and vertex. When your cursor keeps jumping somewhere unhelpful, do not fight it — open the snapping dropdown and turn off the agent that is grabbing you.

The Modify Features toolset

`Edit > Modify` opens the Modify Features pane: a long, categorized catalog of everything Pro can do to existing geometry. You will use a small subset constantly. The essential verbs:

TOOL	WHAT IT DOES	TYPICAL USE
Move / Rotate / Scale	Rigid transforms of whole features	Nudging a misplaced building footprint
Vertices	Edit individual vertices: add, delete, drag	Fixing one bad corner of a polygon
Reshape	Sketch a new line across a feature; the crossed portion is replaced by your sketch	Correcting a stretch of stream centerline
Split	Cut one feature into multiple along a sketched line or at a point	Dividing a parcel; breaking a road at an intersection
Merge	Combine multiple selected features into one	Reuniting split parcels; dissolving accidental duplicates
Align Edge / Replace Geometry	Conform one feature's edge or entire shape to another's	Snapping a zoning boundary flush to parcel lines

A few of these deserve narration beyond the table.

Reshape is the tool nobody discovers on their own and everybody loves once shown. Select a feature, activate reshape, and draw a line that crosses the feature's boundary at two points (or crosses a line feature twice). Pro replaces the portion between your crossings with your sketch. It is the natural way to fix "this boundary is right except for this bulge."

Split and **Merge** are inverses, and both have an attribute problem you must think about. When you split a polygon, what happens to its attributes? Both children typically inherit copies of the parent's values — which is right for "zoning district" and wrong for "area in acres" or "assessed value." When you merge, several features' attributes must collapse into one: Pro asks which feature's values survive. Neither tool can know your intent, so *you* must know which fields need manual correction afterward. Chapter 12's discussion of schema design — especially fields that should be calculated rather than stored — is the long-term fix.

> **Watch out:** Merge keeps the attributes you tell it to keep and silently discards the rest. If two parcels with different owner names are merged and you pick the wrong survivor, the losing owner's name is gone from the record. Before any merge, glance at the attributes of everything selected and decide deliberately which record is the keeper.

Align Edge and its relatives exist because "these two boundaries should be the same line but aren't" is one of the most common defects in real data. Rather than dragging vertices one at a time, alignment tools conform one edge to another in a single operation.

Topology-aware editing

Now the deeper idea. In many datasets, features *share* geometry conceptually: two adjacent parcels share a boundary line, a county boundary coincides with the state boundary along one side. But in an ordinary feature layer, each polygon stores its own complete boundary — the shared edge exists twice, as two independent copies that merely happen to coincide. Move one and the other stays put, opening a gap or an overlap.

Topology-aware editing fixes this by treating coincident geometry as shared while you edit. In Pro, the lightweight version is called **map topology**: when you activate it (there is a topology toggle on the Edit ribbon — switching from no topology to map topology), the vertex and reshape tools begin operating on shared edges. Drag the boundary between two parcels and *both* parcels update together; no gap, no overlap. The edge behaves as the single line your mental model always said it was.

Map topology is transient — a behavior of your editing session, not a property stored in the data. The heavyweight version, **geodatabase topology**, stores explicit rules ("parcels must not overlap," "manholes must sit on sewer lines") and can validate a whole dataset against them, flagging every violation for review. That belongs to quality assurance and is covered properly in Chapter 14 (Data Quality: Validation, Topology, and QA). For day-to-day editing, the takeaway is simpler: whenever you edit boundaries that neighbors share, turn map topology on first. Editing shared edges without it is how sliver polygons are born.

Versioned and Non-Versioned Editing

So far we have quietly assumed you are the only editor. Drop that assumption and a hard question appears: if two people edit the same data at the same time, whose changes win? This section is conceptual — you may never administer the systems described here, but you will almost certainly *work within* one, and understanding the model keeps you out of trouble.

The default: last writer wins

For hosted feature layers and non-versioned database editing, the answer is brutally simple: edits apply in the order they arrive. If Ana moves a hydrant at 2:01 and Ben moves the same hydrant at 2:03, the hydrant ends up where Ben put it, and Ana's edit is simply gone — no warning, no ceremony. Each individual edit is atomic (you will never get half of Ben's change), but there is no memory that Ana's version ever existed unless editor tracking or history is recording it.

This sounds alarming, and occasionally it is, but notice what makes it tolerable in practice: *real editors rarely touch the same feature within the same window of time.* A hundred field workers updating a million hydrants collide almost never. The last-

writer-wins model works fine when edits are small, fast, and spread out. It breaks down when edits are big, slow, and concentrated.

Versioning: everyone gets a draft

Enterprise geodatabases (the multi-user databases behind ArcGIS Enterprise — Chapter 35) offer a richer model called **versioning**. The idea: instead of everyone editing the one true copy, each editor (or project, or crew) works in a *version* — a personal view of the data that starts identical to the main version (usually called *default*) and accumulates that editor's changes privately. You can edit in your version for hours or weeks; nobody else sees your work, and you do not see theirs.

When you are ready, you **reconcile**: pull the latest state of the parent version into yours and let the system compare. Most changes slide together automatically because they touched different features. Where both you and someone else changed the *same* feature, you have a **conflict**, and the software presents each one for a human decision: keep your change, keep theirs, or inspect and choose field by field. Once conflicts are resolved, you **post** — push your reconciled changes up into the parent, where they become part of the shared truth.

If this sounds like branching and merging in a version control system for code, that is exactly the right analogy. There are two implementations you will hear named — an older one tied to direct database connections and a newer one designed around web services, commonly called traditional and branch versioning — and their operational details differ, but the editor's mental model (branch, work privately, reconcile, resolve conflicts, post) is the same in both. Which one your organization uses is an architecture decision for Chapter 35; whether *you* need versioning at all comes down to one question: do your edits take long enough, and overlap enough, that last-writer-wins would hurt? Utility networks and parcel fabrics — Esri's specialized datasets for utility systems and land records, both built on long, interdependent edit transactions — answer yes. A volunteer-maintained bench inventory answers no.

Where hosted layers fit

Hosted feature layers in ArcGIS Online are non-versioned in this sense: edits apply immediately, last writer wins. What they offer instead are complementary safeguards — ownership-based editing (people can only touch their own features), editor tracking, view layers that expose read-only or filtered slices, and offline editing with sync for

field work (where the sync process handles the merge automatically, favoring the most recent edit). For the large majority of web-centric workflows, these are enough. When they stop being enough, that is the signal you have grown into Enterprise territory.

Many Hands on One Map

Technology handles simultaneous edits; it cannot handle disorganized teams. The most reliable multi-editor deployments succeed on coordination patterns, not features. Here is the playbook, roughly in the order you should reach for each piece.

Divide the territory

The cheapest conflict is the one that cannot happen. Partition the editing work so no two people have a reason to touch the same features:

- **Geographically**: Ana edits the north district, Ben the south. Make it visible — a boundaries layer in the editing map so everyone knows whose side they are on.
- **Thematically**: one person owns roads, another owns addresses, even where they overlap spatially.
- **By lifecycle**: field staff create features with status "unverified"; only the QA editor may change status to "approved" or delete anything.

That last pattern — a status field plus social rules about who advances it — is the single highest-value trick in this chapter. It converts destructive operations (delete, overwrite) into reviewable proposals, using nothing but a domain field from Chapter 12.

Turn on editor tracking

Editor tracking adds four automatically-maintained fields to a layer: who created each feature and when, and who last edited it and when. You enable it in the hosted layer's item settings (or, for geodatabase data in Pro, through the dataset's properties). Once on, every record carries its own provenance. The costs are near zero; the benefits compound:

- **Accountability**: "who moved this hydrant?" becomes a query, not an investigation.
- **Filtering**: dashboards showing "features edited this week" or "Ana's captures today" come free.

- **Ownership-based access control depends on it**: the platform needs to know who created a feature to enforce "editors can only edit their own."

> **Watch out:** Editor tracking timestamps are stored in universal time (UTC), not your local time zone. A feature edited at 5 p.m. local may show a stored time that looks like the middle of the night. Configure pop-ups and exports to convert to local time before anyone panics about after-hours editing that never happened.

Design for conflict avoidance, then resolution

With territory divided and tracking on, add friction only where risk remains:

- **Give editors the narrowest capability that does the job.** If field crews only ever add inspections, grant add-only — then a bad day in the field cannot delete anything.
- **Route different audiences through different view layers** (Chapter 10): the public gets read-only, the field crew gets add-and-update-own, the QA lead gets full capability on the source.
- **Keep editing maps separate from viewing maps.** An editing map contains just the editable layers, the reference layers editors need, and nothing tempting.
- **For long-running, high-overlap work in Enterprise**, use versioning and make reconcile-and-post a scheduled ritual — end of day or end of shift — so versions never drift far from default and conflicts stay small and fresh.

Audit patterns

Finally, assume something will eventually go wrong anyway, and arrange to notice quickly and recover cheaply:

- **Snapshot before campaigns.** Before a big editing push, export the layer (to a file geodatabase or a copy of the hosted layer). A snapshot turns "we ruined the data" into "restore from Tuesday."
- **Build a QA view.** A simple web map filtered to recently edited features (editor tracking makes this trivial) lets a reviewer eyeball each day's changes in minutes. Pair it with the status-field lifecycle and you have a lightweight approval pipeline.

- **Watch the counts.** A feature count that drops unexpectedly is the loudest possible alarm that someone deleted in bulk. A dashboard tile (Chapter 28) showing total records costs nothing.
- **In Enterprise, consider archiving** — a geodatabase option that preserves history so past states can be reconstructed. Where available, it is the difference between an audit trail and an audit guess.

> **Tip:** Run a five-minute pilot before any multi-editor rollout: two people, same layer, deliberately edit the same feature and delete each other's test records. Whatever confusion surfaces in the pilot would have surfaced in production with real data. Every setting mentioned in this chapter is easier to change before launch than after.

Choosing Your Workflow

Strip away the detail and the decisions are few. Edit in the browser when the edits are small, the editors are many or non-technical, and immediacy matters; configure the layer's item settings and forms carefully, because they are your only guardrails there. Edit in Pro when geometry work is serious, when you want a pending state and a deep undo stack, and when topology-aware tools will save you from slivers. Stay with last-writer-wins hosted layers until long, overlapping edit transactions force you toward versioning — and reach for coordination patterns (territory, status lifecycles, editor tracking, narrow capabilities, QA views) long before you reach for heavier technology.

Editing is also where data quality is won or lost, which is why the next chapter, Chapter 14 (Data Quality: Validation, Topology, and QA), picks up exactly where topology-aware editing left off — turning "please digitize carefully" into rules the software enforces for you.

Data Quality: Validation, Topology, and QA

A map made from bad data is worse than no map at all, because it looks authoritative. The parcel boundary that overlaps its neighbor, the hydrant recorded twice, the survey point sitting in the Gulf of Guinea because someone's GPS reported zeros — none of these announce themselves. They hide inside a polished map until the moment someone makes a decision based on them. This chapter is about finding those problems before they find you: what "clean" actually means, the tools ArcGIS gives you to define and enforce cleanliness, and how to build a repeatable review process so quality is a habit rather than a heroic one-time cleanup.

You already know how data gets created (Chapter 11) and how a good schema prevents many problems before they start (Chapter 12). This chapter picks up where those leave off: the data exists, and now you need to trust it.

What Clean Data Actually Means

"Clean" is not one property. It is several independent properties, and data can pass some while failing others. A dataset can have perfect geometry and garbage attributes, or flawless attributes attached to shapes that overlap where they shouldn't. When you assess a dataset — yours or someone else's — walk through each dimension separately.

DIMENSION	THE QUESTION IT ANSWERS	TYPICAL FAILURE
Geometric validity	Is each individual shape well-formed?	A polygon whose boundary crosses itself
Topological integrity	Do features relate to each other correctly?	Two parcels that overlap; a road that doesn't connect
Attribute validity	Are the values in each field legal and sensible?	A status field containing "actve", "Active", and "yes"
Completeness	Is everything that should be there, there?	Half the county digitized, no note saying so
Uniqueness	Is each real-world thing represented exactly once?	The same inspection submitted twice from the field

DIMENSION	THE QUESTION IT ANSWERS	TYPICAL FAILURE
Currency	Is the data recent enough for its purpose?	A "current" zoning layer last edited years ago
Lineage	Do we know where it came from and how it was made?	No metadata at all

The first three dimensions have dedicated tooling in ArcGIS, and most of this chapter covers them. The last four are mostly process and documentation — which is why the chapter ends with review loops and metadata rather than more buttons.

One framing worth internalizing early: **the cheapest place to fix bad data is before it exists.** A domain that prevents a typo costs nothing; finding and fixing that typo across ten thousand records six months later costs an afternoon. Every technique below gets more expensive as you move from schema design to field capture to post-hoc cleanup, so push your quality effort as far upstream as you can.

Geometry Validity: When Shapes Themselves Are Broken

Before you worry about how features relate to each other, each feature has to be well-formed on its own. A *valid geometry* is a shape that obeys the structural rules of its type: a polygon's boundary must close and must not cross itself, a line must have at least two distinct vertices, and every feature must actually contain coordinates rather than being an empty placeholder.

Invalid geometry creeps in from many directions: data converted from CAD drawings, hand-digitizing accidents, buggy exports from other software, or geoprocessing operations that produced degenerate leftovers. The symptoms are maddeningly indirect — a tool fails with a cryptic error, an area calculation returns a negative number, a feature refuses to draw or select properly. When a geoprocessing tool fails on data that "looks fine," suspect geometry validity first (Chapter 39, the Troubleshooting Encyclopedia, returns to this).

Checking and repairing

ArcGIS Pro ships two companion geoprocessing tools, and their names say it all: **Check Geometry** and **Repair Geometry**. (Geoprocessing tools are covered in depth in Chapter 22; for now, open the Geoprocessing pane from `Analysis > Tools` and search for them by name.)

Check Geometry scans a *feature class* — a geodatabase's table of features sharing one geometry type (Chapter 12) — and writes a report table listing every problem it finds: self-intersections, unclosed rings, null (empty) geometries, and similar defects, with the object ID of each offender, so you can inspect them yourself. Repair Geometry actually fixes the problems, using standard corrections: it closes open rings, splits self-intersecting boundaries, and deletes features whose geometry is entirely empty.

Both tools let you choose a validation standard. Esri's own definition of "valid" differs slightly from the OGC definition (the Open Geospatial Consortium, the industry standards body). If your data will travel to non-Esri software — PostGIS, QGIS, web mapping libraries — validate against the OGC standard, since that is what the rest of the world checks against.

> **Watch out:** Repair Geometry edits your data in place, and some repairs delete features outright (an empty geometry has nothing to save). Always run it on a copy, or at minimum run Check Geometry first and read the report so you know what is about to change. There is no "preview repairs" mode.

A related but distinct problem: geometry can be *valid* but *wrong* — a perfectly well-formed parcel in the wrong place. Validity checks can't catch that. Positional accuracy is a completeness-and-lineage question, which is why field verification and metadata (later in this chapter) matter even when every automated check passes.

Topology: Rules About How Features Relate

Topology is the study of how shapes connect, touch, and contain one another — independent of exactly where they sit. In GIS, topology is how you express statements like "parcels must not overlap," "there must be no gaps between counties," and "every water valve must sit on a water main." Individually valid features can still violate these

collective rules, and no amount of squinting at a map will reliably find every violation. Topology tooling finds them for you.

Two kinds of topology in Pro

ArcGIS Pro offers topology in two flavors, and confusing them wastes time:

- **Map topology** is a lightweight editing aid. Turn it on during an edit session and Pro treats coincident edges and vertices as shared — drag a boundary between two parcels and both parcels update together. It enforces nothing and stores nothing; it just makes coordinated editing possible. Chapter 13 covers it as part of editing workflows.
- **Geodatabase topology** is the real enforcement machine: a stored set of rules attached to feature classes, plus machinery to validate the data against those rules and track every violation. This is what the rest of this section is about.

Setting up a geodatabase topology

Geodatabase topology lives inside a *feature dataset* — a container in a geodatabase that groups feature classes sharing one coordinate system (Chapter 12 introduced these). If your layers are loose in the geodatabase, move them into a feature dataset first. In the Catalog pane, right-click the feature dataset and choose the option to create a new topology; a wizard walks you through the pieces:

Cluster tolerance. The distance within which two vertices are considered "the same place." Coordinates closer together than this snap together during validation. The default, derived from the data's coordinate precision, is deliberately tiny — and you should almost always keep it tiny.

> **Watch out:** A generous cluster tolerance feels helpful ("it'll snap all my sloppy edges!") but it is a blunt instrument: validation will move any vertex within that distance of another, including vertices that were correct. Set it large and you can visibly distort careful data, and the distortion is permanent. Fix sloppy digitizing with editing tools and snapping (Chapter 13), not with cluster tolerance.

Ranks. When vertices from two layers snap together, rank decides which one moves. Give your most trusted layer (say, surveyed parcel boundaries) the best rank and the

less trusted layer (a digitized soils map) a worse one; the soils vertices will move to meet the parcels, never the reverse.

Rules. The heart of the topology. You pick from a catalog of a few dozen predefined rules, each binding one or two feature classes.

The rules you'll actually use

The full rule catalog is long, but a handful of rules do most of the real-world work:

RULE (PARAPHRASED)	APPLIES TO	WHAT IT CATCHES
Must Not Overlap	Polygons	Two parcels claiming the same ground
Must Not Have Gaps	Polygons	Slivers of no-man's-land between adjacent polygons
Must Be Covered By Feature Class Of	Polygons	A city block not covered by any census tract
Must Not Have Dangles	Lines	A road segment whose end connects to nothing
Must Not Self-Overlap / Self-Intersect	Lines	A stream that doubles back over itself
Must Be Covered By	Points on lines	A valve floating off its water main
Must Be Properly Inside	Points in polygons	An address point outside every parcel

Notice the pattern: polygon rules are mostly about *coverage* (no gaps, no overlaps, everything accounted for), line rules are mostly about *connectivity* (things that should join actually join), and point rules are mostly about *containment or coincidence* (points sit where they belong). When you design a topology, translate your data's real-world logic into these terms: what must touch, what must never touch, what must be inside what.

A note on "Must Not Have Gaps": this rule flags the outer boundary of your whole dataset too, since technically the world outside your study area is a "gap." That outer ring is legitimate, and topology has a concept for legitimacy — read on.

Validate, inspect, fix

Creating the topology does nothing by itself. You must **validate** — run the rules against the data. Right-click the topology in the Catalog pane to validate everything, or validate just the visible extent from the editing ribbon while working. After the first validation, the topology tracks *dirty areas*: regions where features have been edited since the last validation. Only dirty areas need re-checking, which keeps validation fast even on large datasets.

Violations become *error features* — points, lines, or polygons marking exactly where each rule failed. Add the topology to a map and errors draw in an unmissable style. The **Error Inspector** pane (available from the editing ribbon when a topology layer is in your map) lists every error in a table you can sort, filter by rule, and work through row by row.

For each error you have three honest options:

1. **Fix it.** The Error Inspector offers predefined fixes appropriate to each rule — merge a sliver gap into a neighboring polygon, trim or extend a dangling line, snap a point to the line it should sit on. For messier cases you fix the geometry by hand with the editing tools from Chapter 13, then re-validate.
2. **Mark it as an exception.** Some violations are correct. A cul-de-sac genuinely dangles. Your dataset's outer boundary genuinely has a "gap" beyond it. Marking these as exceptions records a human judgment: *seen, considered, legitimate.* Exceptions stop appearing as errors but remain stored, so the decision is auditable.
3. **Leave it as a known error.** Sometimes the fix needs information you don't have yet. An unresolved error is honest; a hasty wrong fix is not.

> **Tip:** The error count after your first validation on inherited data can be demoralizing — thousands of errors is normal. Sort the Error Inspector by rule and fix in bulk, one rule at a time, starting with the rule that matters most for your

use. Most datasets have a few systematic problems repeated many times, not thousands of unique ones.

One important boundary to understand: geodatabase topology is a Pro-and-geodatabase capability. A hosted feature layer in ArcGIS Online (Chapter 10) does not enforce topology rules as people edit it over the web. The standard pattern is to treat the geodatabase in Pro as the authoritative, topology-checked home of the data, and publish outward from there — or to pull web-edited data back into Pro periodically for validation, which is exactly the review-loop pattern described later in this chapter.

Attribute Validation: Keeping Fields Honest

Geometry can be perfect while the attribute table rots. Attribute quality lives or dies on one principle: **constrain input so bad values are impossible, rather than cleaning bad values after the fact.**

Domains as guardrails

Chapter 12 covered *domains* from the design side; here is the quality angle. A **coded value domain** restricts a field to a fixed list of legal values ("Active," "Abandoned," "Proposed" — and nothing else); a **range domain** restricts a numeric or date field to a legal interval (pipe diameter between plausible bounds). Once a domain is attached to a field, every editing surface that respects it — Pro's attribute pane, Map Viewer editing, Field Maps forms — replaces free typing with a pick list or a range check. The typo "actve" simply cannot be entered.

Domains are the single highest-leverage data quality tool in the entire platform. If a field's plausible values can be listed or bounded, it should have a domain. When you inherit undomained data, the cleanup sequence is: summarize the field's existing values (right-click the field in the attribute table and choose the summarize option, or run the Summary Statistics tool with that field as the case field), standardize the variants to a canonical set with field calculations, then attach a domain so the mess never returns.

Contingent values: when one field constrains another

Domains treat each field independently, but real attributes often depend on each other. If `TreeType` is "Conifer," then `Species` should offer pines and firs — not maples. **Contingent values** encode exactly this: you group related fields into a *field group* and define which combinations of domain values are valid. Editing surfaces that support contingent values (Pro, and web and field editing in recent versions) then filter each pick list based on what's already chosen, so editors are steered toward valid combinations instead of being scolded afterward.

Contingent values are defined on the feature class in Pro (look in the feature class's design views alongside fields and domains). They require the fields involved to already have domains — the contingency picks from domain values, it doesn't invent new ones.

Attribute rules: validation with logic

When validity depends on logic rather than lists — "an inspection's follow-up date must be after its inspection date," "a service line's material must match its parent main unless the install year is recent" — you graduate to **attribute rules**: small scripts written in Arcade, Esri's expression language (Chapter 8), attached to a feature class. They come in three flavors: *calculation* rules that fill in or correct values automatically as edits happen, *constraint* rules that block an edit outright when it violates a condition, and *validation* rules that run as a batch check and flag violations for review rather than blocking anything.

Attribute rules are primarily a geodatabase capability, with support on hosted feature layers growing in recent versions. For deep, standards-driven QA programs, Esri also offers **ArcGIS Data Reviewer**, an extension that adds a large library of ready-made automated checks and review-management tooling on top of the attribute rule machinery. If your organization runs formal QA on large datasets, it is worth knowing Data Reviewer exists; for most projects, domains, contingent values, and a few attribute rules cover the need.

> **Tip:** The very cheapest validation happens on the capture form itself. If your data arrives via Survey123, its form logic — required questions, input masks, constraint expressions, and questions that appear only when relevant — rejects bad values on the phone, before they ever travel to the server. Chapter 30 covers form design; think of it as attribute validation's front line.

Duplicates: Each Thing Exactly Once

Duplicate features are among the most common and most damaging quality problems, because they silently corrupt counts, sums, and analysis results. They arise from predictable causes: a field worker's submission uploads twice on a flaky connection, two datasets covering overlapping areas get merged, an import runs twice, or two field crews independently record the same asset.

Exact duplicates

For features that are identical — same geometry, same attributes, or both — Pro has a matched pair of geoprocessing tools. **Find Identical** reports duplicates without touching anything: you choose which fields to compare (include the shape field to compare geometry), and it writes a table grouping identical records together. **Delete Identical** takes the same parameters and removes all but one record from each group.

Both tools accept a spatial tolerance, so "identical" can mean "within a meter of each other" rather than "bit-for-bit the same coordinates" — essential for GPS-collected points, where two records of the same hydrant will never match exactly.

> **Watch out:** Delete Identical keeps one record from each duplicate group, but *you don't control which one*. If your duplicates differ in ways you care about — one copy has a filled-in comment field, the other doesn't — Delete Identical may keep the worse copy. For anything but trivially identical records, run Find Identical, review the groups, and decide survivorship yourself before deleting.

Near-duplicates

The harder problem is records that are duplicates in reality but not in the database: the same business entered as "Main St Café" and "Main Street Cafe," two points forty meters apart that are the same well. No tool fully automates this, but a practical workflow narrows it fast: use spatial proximity to generate candidate pairs (points within some distance of each other — Chapter 16's proximity tools do this), then compare attributes across each pair, flagging pairs whose names or identifiers are similar. Sort the candidates so a human reviews the ambiguous ones. The goal of automation here is not to decide, but to shrink ten thousand comparisons down to the eighty that need eyes.

The durable fix, as always, is upstream: give features a real-world unique identifier (an asset tag, a permit number) and enforce it at capture, so the second entry of the same thing is caught at the door. Chapter 12's discussion of keys is the foundation.

Spatial QA Patterns: The Usual Suspects

Beyond formal topology, a handful of spatial defects recur in almost every dataset. Learn to hunt them by habit.

Points at nowhere: Null Island and friends

When a GPS fails or a geocoder gives up (a *geocoder* is the service that converts addresses into coordinates — Chapter 11), software often records coordinates of latitude zero, longitude zero — a spot in the ocean off West Africa that GIS people wryly call *Null Island*. Any respectable point dataset check starts with: zoom to the full extent of the layer. If you see a lone cluster far from your study area, investigate. Common causes: zero-zero coordinates, latitude and longitude swapped (points land in the wrong part of the world, or fail entirely, since a longitude like 122 is not a legal latitude), or a coordinate system misassignment that lands data thousands of kilometers off (Chapter 3 explains why and how to fix the definition rather than the data).

The systematic version of this check: select by location. Build a boundary polygon for your legitimate study area, then use `Map > Selection > Select By Location` in Pro (or the equivalent in Map Viewer's analysis tools) to select features that do *not* fall within it. Everything selected needs an explanation.

Orphan points

An *orphan* is a feature that should be attached to something but isn't: an inspection record whose asset ID matches no asset, a tree point inside no park polygon in a parks-tree dataset, a related-table row whose parent was deleted. Orphans are the spatial-relational cousin of the dangling line. Hunt them two ways: relationally, by joining child to parent on the key field and selecting records where the join failed to match; and spatially, with Select By Location for points that fall inside no parent polygon. If your schema uses relationship classes (Chapter 12), honor them during deletes so orphans don't get created in the first place.

Sliver polygons

Slivers are the tiny, threadlike polygons produced when two boundaries that should coincide almost — but don't quite — coincide. Overlay operations (Chapter 16) manufacture them wholesale: intersect a parcel layer with a zoning layer whose shared boundaries were digitized separately, and the output sprouts hundreds of skinny fragments along every almost-shared edge.

Detection exploits their shape: slivers have tiny area and, relative to that area, absurdly long perimeter. Add area and perimeter fields (geodatabase feature classes maintain shape area and length automatically), sort ascending by area, and inspect the small end of the table; a calculated area-to-perimeter ratio flags thin shapes even when they aren't the smallest. For fixing, small numbers of slivers are merged into neighbors by hand during editing; large numbers can be handled with the Eliminate geoprocessing tool, which merges selected polygons into the neighbor they share the longest border with (note that Eliminate requires a higher Pro license level). Prevention beats cure: a "Must Not Have Gaps"/"Must Not Overlap" topology on the inputs, or snapping the layers' shared boundaries before overlay, stops slivers at the source.

Dangles, overshoots, and undershoots

For line networks, the classic defects are the *undershoot* (a line that stops just short of the line it should connect to) and the *overshoot* (a line that crosses slightly past it). Both produce dangles, and both silently break any analysis that depends on connectivity — a route can't cross a gap of two centimeters any more than it can cross a canyon (Chapter 18 depends utterly on connected networks). The "Must Not Have Dangles" topology rule finds them; the Error Inspector's extend and trim fixes handle most of them mechanically, with legitimate dead-ends marked as exceptions.

A QA Review Loop for Field-Collected Data

Field collection (Chapter 30) mass-produces data, and mass production needs quality control. The mistake teams make is treating QA as a cleanup phase after collection ends. The better model is a *loop* that runs continuously while collection is happening — problems get caught while the crew is still nearby and memory is fresh.

Design the loop before anyone collects anything

Three schema decisions, made up front, make the whole loop possible:

1. **A review-status field with a domain.** Something like `qa_status` with values *Needs Review*, *Approved*, *Rejected*. Default every new record to *Needs Review*.
2. **A reviewer-comment field**, so rejection always carries a reason the field crew can act on.
3. **Editor tracking enabled.** Hosted feature layers can automatically stamp who created and last edited each record and when — turn this on in the layer's item page settings. It costs nothing and answers "who do I ask about this weird point?" forever.

Route the data with views

Hosted feature layer *views* (Chapter 10) let one dataset present different faces to different audiences, which is exactly what a review loop needs:

- **The field view**: editable, used by collection crews in Field Maps, ideally restricted so crews see and edit only their own submissions.
- **The reviewer view**: editable, filtered to `qa_status = Needs Review`, used by whoever performs QA. As records get approved they vanish from this view automatically — the reviewer's queue is self-maintaining.
- **The published view**: read-only, filtered to `qa_status = Approved`. This is the only version stakeholders, public maps, and downstream analysis ever see. Unreviewed data is simply invisible to the world.

A Dashboard (Chapter 28) pointed at these layers gives the whole operation a heartbeat: submissions per day, counts by status, rejection rate by crew. When the rejection rate for one crew spikes, that's a training conversation, not a data problem.

What the reviewer actually does

An effective review session is a checklist, not a vibe. For each record in the queue:

- **Location plausible?** Point falls inside the study area, on or near the asset it claims to describe, not stacked on top of an existing approved record (duplicate check).
- **Attributes complete and consistent?** Required fields filled, values consistent with each other and with the attached photo — the photo is the reviewer's ground truth. A record claiming a two-meter culvert whose photo shows a drainage ditch gets rejected with a comment.

- **Then set the status.** Approve, or reject with a written reason. Never silently fix substantive field errors yourself: fixing without feedback guarantees the same error arrives tomorrow. Trivial normalization (capitalization, spacing) is fine to fix directly.

Rejected records flow back to the field view, where crews see their own rejects and the reasons, fix them on-site, and resubmit — status back to *Needs Review*, and around the loop again.

> **Tip:** Run the loop daily during active collection, even for ten minutes. QA feedback is fresh produce: a rejection delivered same-day gets fixed from memory; a rejection delivered three weeks later requires a return trip to the site.

Periodically — weekly, or at collection milestones — pull the accumulated approved data into Pro and run the heavier machinery from earlier in this chapter over it: geometry checks, topology validation, Find Identical. Web-based review catches record-level problems; the geodatabase pass catches the collective ones.

Documenting Quality with Metadata

Every judgment this chapter has taught you to make — what was validated, what was declared an exception, how complete the data is, what it should and should not be used for — evaporates unless you write it down. *Metadata* is where it gets written down: structured documentation attached to the data itself, traveling with it wherever it goes.

Chapter 5 taught you to read other people's metadata skeptically when evaluating data you find. This is the other side of that transaction. Someone — quite possibly you, eighteen months from now, with no memory of any of this — will evaluate *your* data the same way, and the metadata you write today is the only testimony they'll have.

At minimum, record:

- **Lineage**: where the data came from (source materials, collection method, instruments) and what was done to it (projections, edits, merges, cleanups).
- **Currency**: when it was collected and when last updated — and, more usefully, whether it is maintained or a snapshot.

- **Completeness**: what's covered and what's deliberately or accidentally missing. "Sidewalk data complete for the downtown district only" is one sentence that prevents a hundred wrong conclusions.
- **Accuracy**: how positions were captured (survey-grade receiver versus phone GPS implies very different trust) and what validation was performed — including your topology rules and known unfixed errors.
- **Constraints**: license, attribution requirements, and any use warnings ("planning purposes only; not a survey product").

Where it lives: in ArcGIS Online, every item has an item page — fill in the summary, description, terms of use, and credits there, because that page is what data browsers actually read. In Pro, every dataset carries a full metadata document editable from the Catalog pane (right-click the item and look for the metadata view), and Pro can present the editor in the shape of well-known formal standards such as FGDC and ISO metadata if your organization requires them. Formal standards matter for government and inter-agency exchange; for everything else, honest prose beats an empty standards-compliant template every time.

> **Tip:** Write metadata the day the work happens, in plain sentences, even if it's three lines. The perfect metadata record you'll write "when the project wraps up" has never once been written in the history of GIS.

The Working Checklist

Quality is not a phase; it is a posture. Compressed to a card you could tape to your monitor:

1. Constrain first — domains, contingent values, and form logic before anyone types a value.
2. Trust nothing inherited — run Check Geometry and a full-extent zoom on every dataset that enters your world.
3. Encode relationships as topology rules — validate, fix, and mark true exceptions honestly.

4. Hunt the usual suspects — Null Island, orphans, slivers, dangles, duplicates — on a schedule, not on suspicion.
5. Review field data in a daily loop — status field, filtered views, feedback with reasons.
6. Write down what you did and what you didn't — metadata is QA's memory.

Do these habitually and something quietly changes: you stop asking "is this data any good?" because you already know — and you can prove it to anyone who asks.

Imagery and Rasters in Practice

A raster is a grid of cells, each holding a number. That is the entire idea. A satellite photo is a raster where the numbers describe brightness. An elevation model is a raster where the numbers are heights. A rainfall surface is a raster where the numbers are millimeters. Chapter 1 (How GIS Thinks) laid out the theory of vector versus raster; this chapter is about what you actually do with rasters — where imagery comes from, how you get it into a map, how to read and combine its bands, how to process it without wrecking the original, and how to keep it from swallowing your storage budget. Rasters are the heaviest data you will ever handle in ArcGIS, and most of the pain people experience with them comes from not knowing a handful of practical realities up front. By the end of this chapter you will know all of them.

What a Raster Really Is When You Are Working With One

Before touching sources and tools, four working terms, because everything else in the chapter leans on them.

Cells and resolution. Every raster is a rectangle of cells (also called pixels, from "picture elements"). Each cell covers a square patch of ground, and the size of that patch is the raster's *resolution*. A 10-meter raster has cells that each represent a 10-by-10-meter square of the Earth. Smaller cells mean finer detail — and dramatically bigger files, a point we will return to at the end.

Bands. A raster can store more than one number per cell. Each layer of numbers is called a *band*. A simple elevation raster has one band (the height). A typical color photo has three (red, green, blue). Satellite sensors often record many more, including kinds of light your eyes cannot see. Bands are the key to almost everything interesting you can do with imagery, so they get their own section shortly.

Bit depth. Each cell's number has to be stored somehow, and *bit depth* is how much room each number gets. Low bit depth can only hold small whole numbers; higher bit depth can hold big numbers or decimals. You rarely choose this yourself, but it explains why two rasters covering the same area can differ wildly in file size.

NoData. Rasters are rectangles, but the world is not. The corners of a satellite scene, the area outside a study boundary, cloud-masked pixels — these cells hold a special marker called *NoData*, meaning "nothing was measured here." NoData is not zero. Zero is a measurement (zero elevation, zero rainfall); NoData is an absence.

> **Watch out:** Confusing NoData with zero is one of the most common raster mistakes. If a "fix" fills missing cells with 0, every later calculation — averages, slopes, vegetation indices — will be silently poisoned by fake zeros. When a raster looks like it has a suspicious black or flat border, check whether those cells are NoData or actual zeros before you analyze anything.

Where Imagery Comes From

"Imagery" means rasters captured by a camera or sensor, as opposed to rasters computed from other data. It arrives from four main directions, and knowing which one you are dealing with tells you what to expect about detail, freshness, and cost.

Satellites

Satellites give you the widest coverage and the most regular repeat visits. Two families matter for most readers:

Free public programs. Government-operated satellites such as the United States' Landsat series and Europe's Sentinel series (part of the EU's Copernicus program) photograph the world's land surfaces on a repeating schedule, and the imagery is free to use. The trade-off is resolution: pixels are tens of meters on a side (roughly 30

meters for Landsat, roughly 10 meters for Sentinel-2's sharpest bands). You cannot see a car or a house footprint clearly at that scale, but you can see fields, forests, lakes, burn scars, and urban growth beautifully — and because the archive goes back decades, you can watch them change. These programs also capture invisible-light bands, which is where the analysis magic lives.

Commercial high-resolution satellites. Private operators fly satellites whose pixels are well under a meter — sharp enough to count vehicles. This imagery is licensed, not free, and you typically buy it per area or access it through subscriptions. The crisp global basemap you see in ArcGIS is stitched largely from commercial sources like these.

Aerial photography

Aircraft flying survey patterns produce imagery sharper than most satellites, usually with pixels around a meter or better. Many governments run recurring aerial programs — in the United States, an agriculture-department program photographs the whole country every few years, and the results are free. Aerial imagery you get from reputable programs is almost always delivered as an *orthophoto*: a photo that has been geometrically corrected so that every pixel sits in its true map position, as if the camera had been looking straight down everywhere at once. Raw air photos, by contrast, are distorted by camera tilt and terrain — a mountainside leans, a tall building smears outward. Orthorectification (the correction process) is what turns a picture *of* a place into a picture you can measure distances on.

Drones

Drones are aerial photography you control. You choose the day, the height, and the area, and you get pixels measured in centimeters. The catch is that a drone produces hundreds of overlapping snapshots, not one clean map-ready image. Turning them into an orthophoto (and, as a bonus, a surface model) is the job of *photogrammetry* software — Esri's offerings in this space process drone photo sets into orthomosaics (a single seamless orthophoto stitched from all the overlapping shots) and 3D surfaces (see Chapter 2 for the product landscape). For small sites — a construction lot, a farm field, a stretch of shoreline — a drone is often the cheapest way to get imagery that is both very recent and very sharp.

Scanned maps and historical documents

The fourth source is paper: old survey maps, historical aerial prints, hand-drawn plans, geology sheets. Scan them and you have a raster — but a raster that knows nothing about where it belongs on the Earth. *Georeferencing* fixes that: you identify points on the scan whose real-world locations you know (a road intersection, a survey monument), match them to those locations on a modern basemap, and the software stretches the scan into place. ArcGIS Pro has dedicated georeferencing tools (with the scanned raster selected, look for **Georeference** on the **Imagery** ribbon tab), and Chapter 11 (Creating Data: Every Path In) covers the workflow among the other ways of bringing data to life.

Getting Imagery Into Your Map

You have two broad routes: use imagery someone else already hosts, or bring your own files.

The ready-made route: Living Atlas

Before you download or upload anything, check ArcGIS Living Atlas of the World, Esri's curated catalog of ready-to-use layers (Chapter 5 covers it in depth). It carries live, regularly updated imagery layers — Sentinel-2 and Landsat archives you can filter by date and cloud cover, national aerial programs, global elevation, and more. In Map Viewer, `Add > Browse layers` and switching the source to Living Atlas gets you there; searching for "Sentinel-2" or "imagery" surfaces the main candidates. These layers stream from Esri's servers, cost you nothing to store, and are maintained by people whose whole job is keeping them current.

> **Tip:** Make Living Atlas your first stop every time. The most expensive raster is the one you host yourself when an identical, better-maintained copy already exists as a free streaming layer. Host your own imagery only when it is genuinely yours — drone flights, purchased scenes, scanned documents.

The bring-your-own route

Your own imagery usually arrives as files — most often GeoTIFF, a widespread raster format that carries its map coordinates inside the file. What you do next depends on where you work:

In ArcGIS Pro, adding a raster file is as simple as adding any other data: `Map > Add Data` and browse to the file. Pro reads it directly from disk; nothing is uploaded anywhere. This is where heavy raster work should happen, for performance reasons discussed later.

In ArcGIS Online, sharing imagery means publishing it as a hosted layer. There are two fundamentally different flavors, and choosing wrong causes grief:

- A **tile layer** is a pre-rendered picture of your raster: the system bakes it into a pyramid of small image tiles, like the basemap. It draws fast anywhere and is cheap to serve, but the pixel *values* are gone — viewers see colors, not numbers. Good for "look at this map." Useless for analysis.
- A **hosted imagery layer** keeps the actual pixel values, bands and all. Viewers can query values, change band combinations, and run processing on it. This is the analytical option, and it requires the right license level and consumes storage credits in proportion to the data size (Chapter 34 covers user types and credits).

The publishing flow lives in your content pages — from `Content > New item` you can upload a raster and choose how it becomes a layer — and the options you see depend on your organization's licensing.

Imagery layers versus the imagery basemap

A confusion worth killing early: the satellite basemap under your map is not the same thing as an imagery layer, even though both look like photos of the Earth.

	IMAGERY BASEMAP	IMAGERY LAYER
What it is	Pre-rendered picture tiles stitched from many sources	Actual raster data with pixel values
Pixel values	Gone — just colors	Present — every band, every number
Capture date	Mixed; varies patch by patch, often unlabeled	Known per image; often filterable by date

	IMAGERY BASEMAP	IMAGERY LAYER
Band combinations	Fixed true color	Switchable (false color, indices, custom)
Analysis	Not possible	Query, process, compute
Cost to you	Free to view	Free if Living Atlas; storage costs if you host it

The basemap is wallpaper — gorgeous, useful wallpaper, but wallpaper. If your project needs to *measure* anything from imagery, or needs to know exactly when a picture was taken, you need an imagery layer.

Watch out: The imagery basemap is a patchwork of different sources and dates. Two neighboring towns may have been photographed years apart, and there is generally no reliable way to pin down the date of a given patch. Never use the basemap as evidence that something existed "as of" some date — use a dated imagery layer instead.

Bands and Band Combinations

Here is the single most mind-expanding fact about satellite and aerial sensors: they see light your eyes cannot. Understanding this turns imagery from pictures into measurements.

Visible light — the rainbow from violet to red — is a narrow slice of a much wider spectrum. Just past red lies *near-infrared* (NIR), invisible to your eyes but perfectly visible to a sensor. Further along lies *shortwave infrared* (SWIR), and beyond that thermal infrared, the heat glow of the surface itself. A *multispectral* sensor — one that records several slices of the spectrum at once — stores each slice as a separate band: one grid of numbers for blue light, one for green, one for red, one for near-infrared, and so on.

Why care? Because materials on the ground have signatures. Healthy vegetation absorbs red light (chlorophyll eats it for photosynthesis) but reflects near-infrared

intensely — living leaves are practically mirrors in NIR. Water absorbs NIR almost completely, so water goes near-black in that band. Freshly burned ground reflects strongly in SWIR. None of this is visible to your eyes, but it is sitting right there in the numbers.

Common band combinations

A screen can only show three colors at once — red, green, and blue channels. A *band combination* is a choice of which sensor bands to pour into those three channels:

True color puts the red band in the red channel, green in green, blue in blue. The result looks like what a passenger in a window seat would see. It is the default and the most intuitive, and it hides everything interesting about infrared.

Color infrared, the classic *false color* combination, pours near-infrared into the red channel, red into green, and green into blue. Suddenly healthy vegetation glows vivid red (because it is so bright in NIR), water turns black, and bare soil goes gray-brown. This combination has been the workhorse of vegetation mapping since film cameras carried infrared-sensitive stock decades ago. The first time you flip a familiar landscape into color infrared and watch the golf courses ignite, band combinations stop being abstract.

Other combinations highlight other things: SWIR-based combinations make burn scars and moisture differences pop, and agriculture-tuned combinations separate crop types. You do not need to memorize recipes — imagery layers in ArcGIS typically ship with a menu of named renderings.

Where to switch: with an imagery layer selected in Map Viewer, the layer's settings offer a choice of processing templates or renderings — look for options named things like "Color Infrared" or "Natural Color." In Pro, the raster's **Appearance** or symbology controls let you assign any band to any channel directly.

NDVI: math on bands

Band combinations are for your eyes; *indices* are for computation. The most famous is NDVI — the Normalized Difference Vegetation Index — and the idea is simple enough to say in a sentence: for each cell, compare how bright it is in near-infrared versus red, and score the difference on a scale from -1 to +1. The actual arithmetic is (NIR − Red) ÷

(NIR + Red), computed cell by cell. Because healthy vegetation is bright in NIR and dark in red, it scores high — typically well above zero and climbing toward the top of the scale as the canopy gets denser and healthier. Bare soil hovers near the low positives, and water usually lands below zero.

What makes NDVI powerful is not one map but comparison: this field versus that field, this June versus last June. Drought stress, irrigation failures, crop emergence, deforestation — all show up as NDVI shifts before your eyes would catch them in true color. Computing NDVI is a one-click operation on most multispectral imagery layers (it is a standard processing template) and a basic raster function in Pro. Chapter 19 (Terrain and Raster Analysis) takes index-based analysis much further.

> **Tip:** When someone asks "can we tell how the vegetation is doing from imagery?", the honest plain-English answer is: yes, because plants reflect invisible infrared light in proportion to their health, and we can compute a per-pixel health score from it. That one sentence justifies half the multispectral imagery ever collected.

Raster Functions: Processing Without Making Copies

Traditional geoprocessing (Chapter 22) works like a kitchen: ingredients in, new dish out, and the new dish takes up space in the fridge. Run a tool on a large raster and you wait while a full-size output raster gets written to disk. Do that five times in a row and you have five enormous intermediate files.

Raster functions work differently: they process pixels on the fly, at display time, only for the pixels currently on your screen. Apply an NDVI function to a continent-sized imagery layer and the result appears in seconds — not because the whole continent was computed, but because only your current view was. Pan the map, and the newly visible pixels are computed as they arrive. Nothing is written to disk; the original raster is untouched.

Common raster functions include:

- **Stretch** — adjusts brightness and contrast so the raster's value range maps well to screen colors (imagery often looks murky until stretched).
- **Clip** — shows only the portion inside a boundary.

- **Band arithmetic** — NDVI and its many index cousins.
- **Hillshade** — turns elevation into shaded terrain (next section).
- **Remap** — reclassifies values into categories, such as turning continuous slope into "gentle / moderate / steep."

Functions chain. Clip to a county, compute NDVI, remap into three health classes — that is a three-link chain, evaluated fresh every time the display draws, always from the original pixels. In Pro, the **Raster Functions** pane (on the **Imagery** ribbon tab) is the workshop, and a saved chain is called a raster function template — a reusable recipe you can apply to any compatible raster. In ArcGIS Online, hosted imagery layers expose the same idea as the processing templates mentioned earlier.

When do you still want old-style geoprocessing? When you need a permanent product: a file to deliver to a client, an input for a long analysis pipeline, a dataset other tools will read. The rule of thumb: raster functions for exploring and displaying, geoprocessing for producing.

Elevation: DEMs, DSMs, and Hillshades

Not all rasters are photos. The other giant family is elevation, where each cell's number is a height.

The vocabulary

A **DEM** — digital elevation model — is the umbrella term for any raster of heights. Within it live two siblings that people constantly mix up:

- A **DTM** (digital terrain model) records the height of the *bare ground*, as if every building and tree were erased.
- A **DSM** (digital surface model) records the height of whatever is on top — rooftops, tree canopy, bridge decks.

The difference matters enormously. Model flooding with a DSM and water will pile up absurdly against "walls" that are actually tree canopies. Estimate solar panel potential with a DTM and every rooftop vanishes. When someone hands you elevation data, your first question is always: ground or surface? And do not trust the label alone — in

everyday use, "DEM" is often applied loosely to mean the bare-earth model specifically, so check the metadata or ask rather than assume.

Elevation data comes from several places: satellite-derived global models (moderate resolution, whole-planet coverage), lidar surveys (aircraft firing laser pulses and timing the echoes — the gold standard, capable of producing both DTM and DSM from the same flight), and photogrammetric surfaces computed from overlapping drone or aerial photos (which are inherently DSMs, since a camera cannot see the ground under a tree). Living Atlas hosts ready-to-use world elevation layers, so for most projects you never need to host terrain yourself.

Hillshade: making terrain legible

A raw DEM displays as a murky gray gradient — technically accurate, visually dead. A *hillshade* fixes that by simulating sunlight: pick a direction and angle for an imaginary sun, and compute how brightly each cell would be lit given its slope and orientation. The result is the familiar, instantly readable shaded-relief look where ridges gleam and valleys fall into shadow.

Two knobs control it: *azimuth* (compass direction of the light, conventionally from the upper-left, because human perception reads relief correctly with light from that side) and *altitude* (sun angle above the horizon — lower angles cast longer, more dramatic shadows). A refinement called *multidirectional hillshade* blends light from several directions at once, so slopes facing away from a single sun do not drown in shadow; it is generally the better-looking default, and Living Atlas offers a world multidirectional hillshade ready to drop under your layers.

Hillshade is display, not analysis — a portrait of the terrain, not a measurement of it. The measurable derivatives of a DEM — slope steepness, facing direction, visibility, watershed boundaries, flow paths — are the subject of Chapter 19.

> **Tip:** The fastest cartographic upgrade in the business: put a hillshade layer under your data with some transparency, or use a blend mode to merge it with land cover or a light basemap. Flat maps gain physical presence immediately. Chapter 4 (Cartographic Design) covers the aesthetics; the ingredient is just a hillshade from Living Atlas.

Mosaics: Many Rasters Acting as One

Imagery rarely arrives as one tidy file. A satellite archive is thousands of overlapping scenes shot on different days. A county aerial survey is delivered as hundreds of tiles. A drone flight yields a stack of processed strips. Managing a pile of rasters as separate layers is misery: seams everywhere, no way to search by date, duplicated styling.

The answer is a *mosaic* — many rasters presented and managed as one seamless layer. Two implementations matter:

In ArcGIS Pro, the mosaic dataset. This is a catalog, not a copy: it stores *references* to your raster files plus rules for how to blend them — which image wins where two overlap (the newest? the least cloudy? the one whose center is nearest?), where the invisible seamlines between images run, and how colors get balanced so adjacent images do not visibly clash. Add ten thousand scenes to a mosaic dataset and your disk usage barely moves, because the pixels stay in the original files. The mosaic assembles the requested view on the fly. Crucially, the individual images keep their identities and attributes — capture date, cloud cover, sensor — so you can query the mosaic like a table: "show me only imagery from last spring."

In ArcGIS Online, imagery layer collections. When you publish multiple images as a hosted imagery layer, you can choose to publish them as a single merged picture or as a *collection* that preserves each image's identity and metadata — the hosted cousin of the mosaic dataset. Collections are what enable date filtering and per-image queries in web maps.

When do you actually need mosaic machinery? If you have a handful of tiles that never change, a one-time merge into a single raster is simpler. Reach for a real mosaic when the imagery is numerous, overlapping, multi-date, or still growing — that is, when the *management* of imagery is itself the problem. Serving large mosaics to many users at scale is one of the classic jobs of ArcGIS Enterprise's imagery capabilities (Chapter 35).

Raster or Vector: Choosing the Right Form

You now know both worlds — Chapter 1 introduced them, Chapters 10 through 14 have lived in vector, and this chapter lives in raster. So which do you use when the choice is genuinely yours?

The deciding question: is the thing you are modeling *continuous* or *discrete*? Continuous phenomena — elevation, temperature, rainfall, noise, imagery brightness — have a value *everywhere*, and a grid of cells is the natural way to say so. Discrete things — parcels, roads, hydrants, customers, boundaries — exist at specific places with hard edges and rich attributes, and geometry-plus-table is the natural way to say that.

SITUATION	BETTER MODEL	WHY
Elevation, temperature, rainfall surfaces	Raster	A value exists at every point; grids model that directly
Photographic imagery	Raster	It is born as pixels
Parcels, buildings, utility assets	Vector	Crisp boundaries, many attributes per feature, editable one at a time
Roads and networks for routing	Vector	Connectivity between features is the whole point (Chapter 18)
Land cover classification	Either	Raster from imagery is cheaper; vector polygons are cleaner for reporting
Modeling spread (fire, flood, pollution)	Raster	Cell-by-cell math over a surface is what these models do
Sparse points over a large area	Vector	Storing millions of empty cells to represent a few hundred wells is waste
Overlaying many criteria (site suitability)	Raster	Weighted cell math across layers is fast and simple (Chapter 19)

Two practical notes soften the dichotomy. First, conversion runs both directions — tools exist to trace raster categories into polygons and to burn polygons into cell grids — so a wrong initial format is an inconvenience, not a dead end. Second, real projects almost always mix the two: vector parcels *over* raster imagery *over* raster hillshade is the everyday sandwich of applied GIS. The skill is not picking a side; it is knowing which layer of the problem is continuous and which is discrete.

Storage and Performance Realities

Everything above eventually collides with physics. Rasters are big, and the ways they get big are predictable.

Why rasters are big

A raster's raw size is cells × bands × bytes per cell. The vicious part is the first term: cut the cell size in half and you have four times as many cells; cut it to a quarter and you have sixteen times. This is why "just get sharper imagery" is never a small request — a county at drone resolution is not somewhat bigger than the same county at satellite resolution, it is orders of magnitude bigger. Multiply again by band count and a multispectral survey becomes serious storage no matter where it lives.

Compression: lossless versus lossy

Compression shrinks files two ways. *Lossless* compression packs the numbers more cleverly and gives back exactly the original values when read — always safe. *Lossy* compression (the JPEG family) discards detail the eye will not miss, achieving far smaller files at the price of slightly altered pixel values.

The rule follows directly from the difference: for anything you will *analyze* — elevation, classified data, imagery destined for NDVI or measurement — use lossless, because analysis reads the numbers and lossy compression quietly changed them. For pure visual backdrops, lossy is fine and the savings are enormous.

> **Watch out:** Lossy compression is a one-way door. Once imagery has been saved with lossy compression, the original pixel values are gone forever, and every index computed from it inherits subtle artifacts — often invisible on screen, but present in the math. Keep an untouched archive copy of any imagery you might ever analyze, and compress derivatives, not originals.

Pyramids: why big rasters can still draw fast

Here is the trick that makes continent-sized rasters usable: *pyramids* (also called overviews) are pre-computed, progressively coarser copies of a raster stored alongside the full-resolution version — the original, a half-resolution copy, a quarter-resolution copy, and so on up the pyramid. When you are zoomed out looking at a whole state, the

software draws from a coarse level and reads a tiny amount of data; only when you zoom to a neighborhood does it touch full-resolution pixels. Without pyramids, viewing a giant raster zoomed out means reading *every* pixel just to average them down to your screen — the classic cause of the several-minute redraw. Pro offers to build pyramids when you add a large raster that lacks them; say yes. The build takes a while once and pays off every draw thereafter.

Modern cloud-friendly raster formats bake this idea in: they arrange pyramids and tiles internally so a viewer can fetch just the fragment it needs over a network. This is what lets hosted imagery layers and Living Atlas stream planet-scale data to your browser without your machine ever holding more than the pixels on screen.

Where the work should happen

A few placement rules save real money and time:

- **Stream what exists.** Public satellite archives, world elevation, national aerial programs — consume them as Living Atlas layers. Hosting your own copy duplicates terabytes that Esri already serves free.
- **Process heavy, publish light.** Do photogrammetry, mosaicking, and bulk processing on a desktop or server near the raw files. Publish the finished product — not the raw inputs — to ArcGIS Online. Uploading a drone flight's raw photo set to the cloud so you can process it there is usually the slowest and most expensive ordering of events.
- **Match the layer type to the need.** If viewers only need to *see* the imagery, publish a tile layer and keep the analyzable original offline; hosted tile storage is much lighter than hosted pixel data. Publish a full imagery layer only when web users genuinely need pixel values.
- **Watch hosted storage.** Hosted raster content consumes credits in proportion to size, and imagery is the easiest way in the platform to consume a lot of them quietly. Chapter 34 covers monitoring; the habit that matters is checking what you are storing a few times a year and deleting superseded uploads.

Tip: Before publishing any raster, ask three questions in order: Does a streaming version already exist in Living Atlas? Do viewers need pixel values, or just a

picture? Is this the finished product or an intermediate? The answers route you to — respectively — no upload at all, a lightweight tile layer, or waiting until the real product is ready.

Where This Leads

You can now read a raster the way a practitioner does: cells and bands rather than "a picture," NoData rather than mysterious black borders, pyramid levels rather than inexplicable slowness. You know the four doors imagery arrives through, the difference between the basemap's wallpaper and an imagery layer's measurements, and why a false-color rendering or an NDVI score extracts truth that true color hides. From here, two chapters carry the thread forward: Chapter 19 (Terrain and Raster Analysis) turns these rasters into answers — slope, visibility, suitability, watersheds — and Chapter 22 (Geoprocessing in Depth) covers the permanent-output tools that complement the on-the-fly raster functions you met here. And when a project finally hands you a folder of GeoTIFFs and a deadline, remember the quiet rule that organizes this whole chapter: the numbers in the cells are the data; everything on the screen is just one of many ways to look at them.
