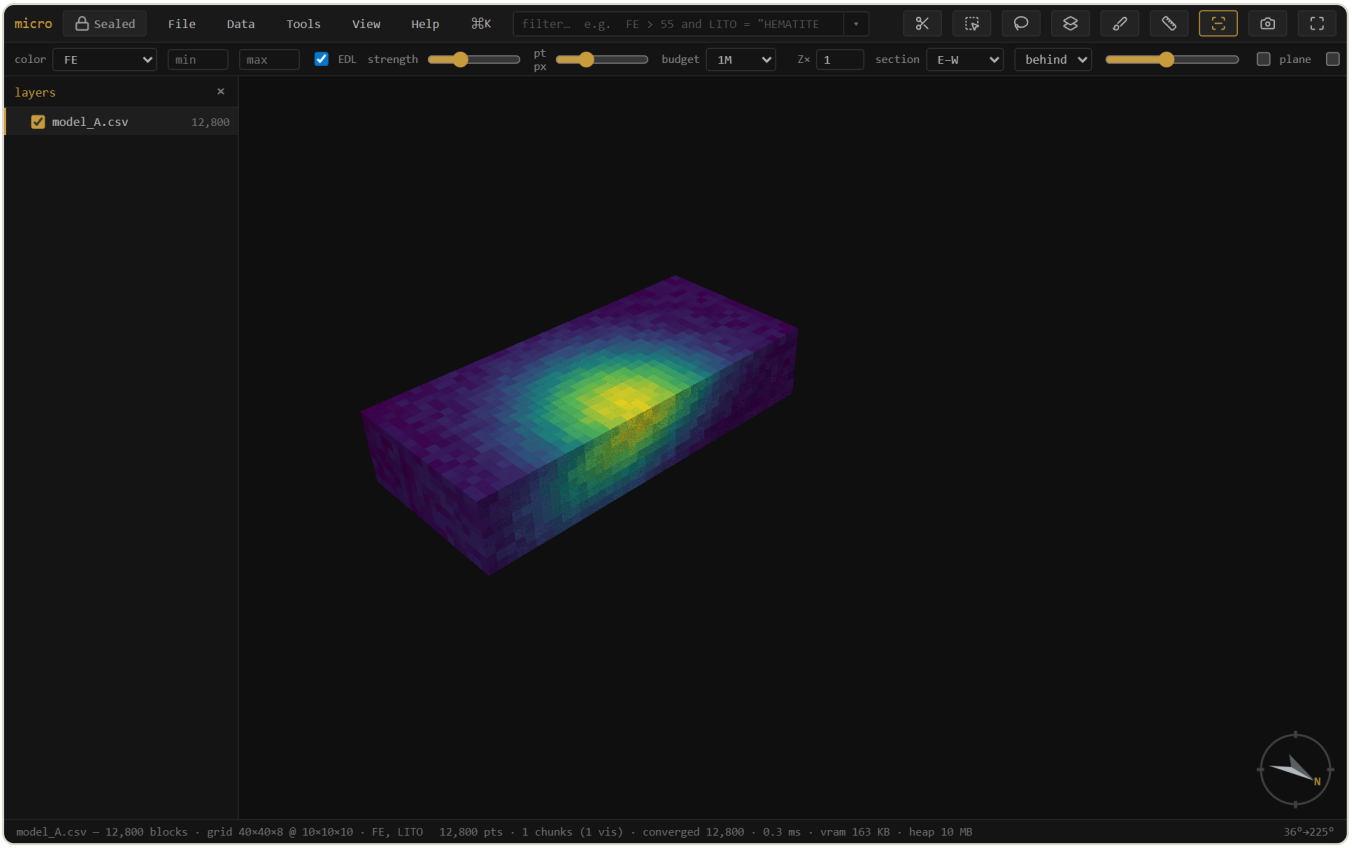
micro

A streaming, offline workbench for massive point clouds and block models.

🔒 **Sealed** — no network, nothing leaves your machine · single self-contained HTML file · download micro.html · open the app ↗

User documentation · Auditable / Geoscientific Chaos Union · gentropic.org/micro

micro opens a point cloud, block model, grid, or set of drillholes — however large — in about a second, with no import step. From there you view, query, derive, join, reconcile, validate, and produce report-ready figures. Every derived column and layer carries a provenance trail, and nothing ever touches the network.



A 12,800-block iron model colored by grade — the high-grade shoot in yellow. Loaded, inferred, and rendered from a plain CSV with no import step.

Contents

A Reference — keys & the expression language

B Formats

C Glossary

D Index

1 What micro is

micro is a single self-contained HTML file — a **viewer that grew into a workbench**. It streams massive spatial-element files (never holding them fully in memory), so the first look is near-instant at any size and the picture densifies while you orbit.

It reads:

- **Point clouds** — LAS, PLY, XYZ / PTS / DAT dumps
- **Block models** — CSV / TXT, Datamine `.dm`, and **Parquet**, sub-blocked included
- **Grids** — GeoTIFF, Surfer `.grd`, ESRI ASCII `.asc`
- **Meshes** — `.msh` (Leapfrog), OBJ, PLY-with-faces, LFM, Datamine wireframe pairs
- **Drillholes** — collar + survey + interval tables (desurveyed on load); a set can carry **several interval tables** (assays, lithology, geotech) sharing one geometry — see Drillholes

Beyond viewing, it does the everyday resource-geology work: filter and section; derive columns by expression, lookup, broadcast, or resampling; domain by a solid; join and reconcile model versions; grade–tonnage, swath, and validation analysis; figure export. All of it runs against the source file, offline, with the provenance kept.

> **What micro is for —** micro shows your data and checks it: view, query, section, pick, measure, join, reconcile, validate, convert. It is *not* an estimation or modelling suite — it doesn't interpolate grades, build geological models, classify resources, or write reports, and that boundary is deliberate. micro is the honest lens, not the resource factory.

How micro thinks

Four ideas explain almost everything else in this manual:

- **A dataset is an element.** A block model, a drillhole set, a grid — each is a table (or several, for drillholes) plus an interpretation of its geometry. The records are positional: row 173 of the file is block 173 everywhere, forever.
- **Columns are the atoms.** Source columns come from the file; derived columns — calculated, materialized, painted, joined, broadcast — behave identically once they exist. Everything downstream (filter, color, analysis, export) consumes columns and doesn't care where they came from.

- **Operations are recorded, not just run.** Every derivation is written down as data — the expression, the method and parameters, the inputs — in the project's element manifests (§16). What was done is always readable, and re-runnable.
- **Nothing is imported.** micro reads your files where they are, streams them, and keeps its own additions *beside* them — sidecar columns, manifests, caches. Delete micro's files and your data is exactly as it was.

Tools → Settings… holds the machine preferences — background, render budget, selection-tool stickiness, GPU/CPU compute, auto-optimize-on-save, the experimental command bar — stored in this browser only. View state (camera, sections, pinned windows) travels with the *project*; data decisions live in its manifests.

> **Tip —** The fastest way to reach anything is the **command palette**: `Ctrl/⌘K` (or the `⌘K` button). It searches every action and shows only what applies to the layer you have selected.

2 Getting started

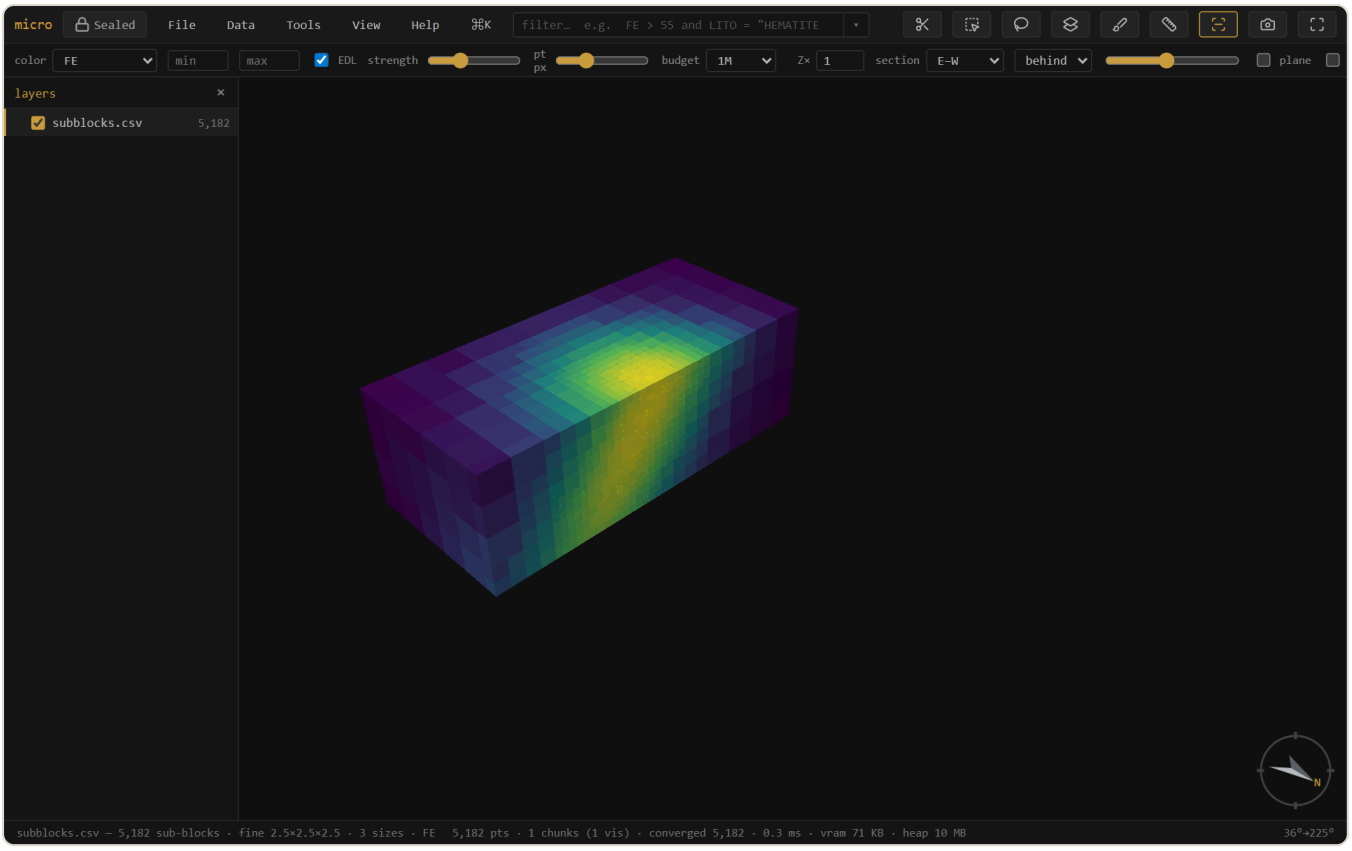
Opening data

Drag a file (or several) anywhere onto the window, or use **File** → **Open…**. Several files at once become separate **layers**. The open dialog shows each file's detected role — an **Open as** ▾ on every row lets you override it (a block model as points, a PLY mesh as points, a delimited file as collar / survey / intervals), and drillhole trios assemble themselves from those roles.

No data handy? The empty state offers ▸ **open the demo project** — one click generates a full scene locally: a 1.8M-block iron model with realistic bimodal grades, a drillhole set threading the ore, pre-mining **topography**, three **pit-phase shells**, and the **ore-shell** mesh the holes chase — sectioned open on the lens, with a grade–tonnage curve already computed. Every workflow in this manual can be tried on it, and **File** → **Save project as…** writes it to a real folder you can inspect: plain CSVs, JSON manifests, no black box.

Sub-blocked models

micro reads **sub-blocked** (octree) block models — where cells are refined into smaller boxes near a boundary or a high-grade zone — and renders **each block at its true size**, straight from the file. CSV/Parquet models with per-block `dX/dY/dZ` columns and Datamine `.dm` models with per-record `XINC/YINC/ZINC` are detected automatically; the model streams, picks, filters, and colors like any other. Join and reconcile handle them by **volume** — see Join & reconcile.



A sub-blocked (octree) model — coarse parent blocks away from the shoot, refined into progressively smaller boxes through the high-grade core, each drawn at its own size.

> **Note —** Detection assumes an **axis-aligned** model with power-of-two (octree) refinement — the standard export. A **rotated** model currently opens as its centroids (points); interpreting a rotated model as boxes (enter or measure its azimuth) is on the roadmap.

The layers panel

`p` toggles the layers panel. Each opened dataset is a layer with its own color, filter, and visibility. Layers can be grouped and reordered by dragging; `[` `]` step the active layer, `h` hides/shows it, `⇧H` **solos** it (press again to restore what was visible). Drillhole sets appear as their own group with collar and survey rows — see Drillholes.

Right-click a layer for its menu, organized top to bottom: **identity** (zoom, rename, reinterpret) · **inspect & analyze** (attribute table, filter, stats, grade–tonnage, swath, file info, lineage, export rows) · **derive & store** (materialize, store as Parquet, index) · **display** (opacity, edges, sections, hide) · **arrange** (move, copy into project, remove) · **Properties…**.

`F2` renames the active layer. **Rename layers…** (on a group, a multi-selection, or the palette) batch-renames: one-click **strip the common prefix/suffix**, or find→replace with optional **regex**, with a live before→after table.

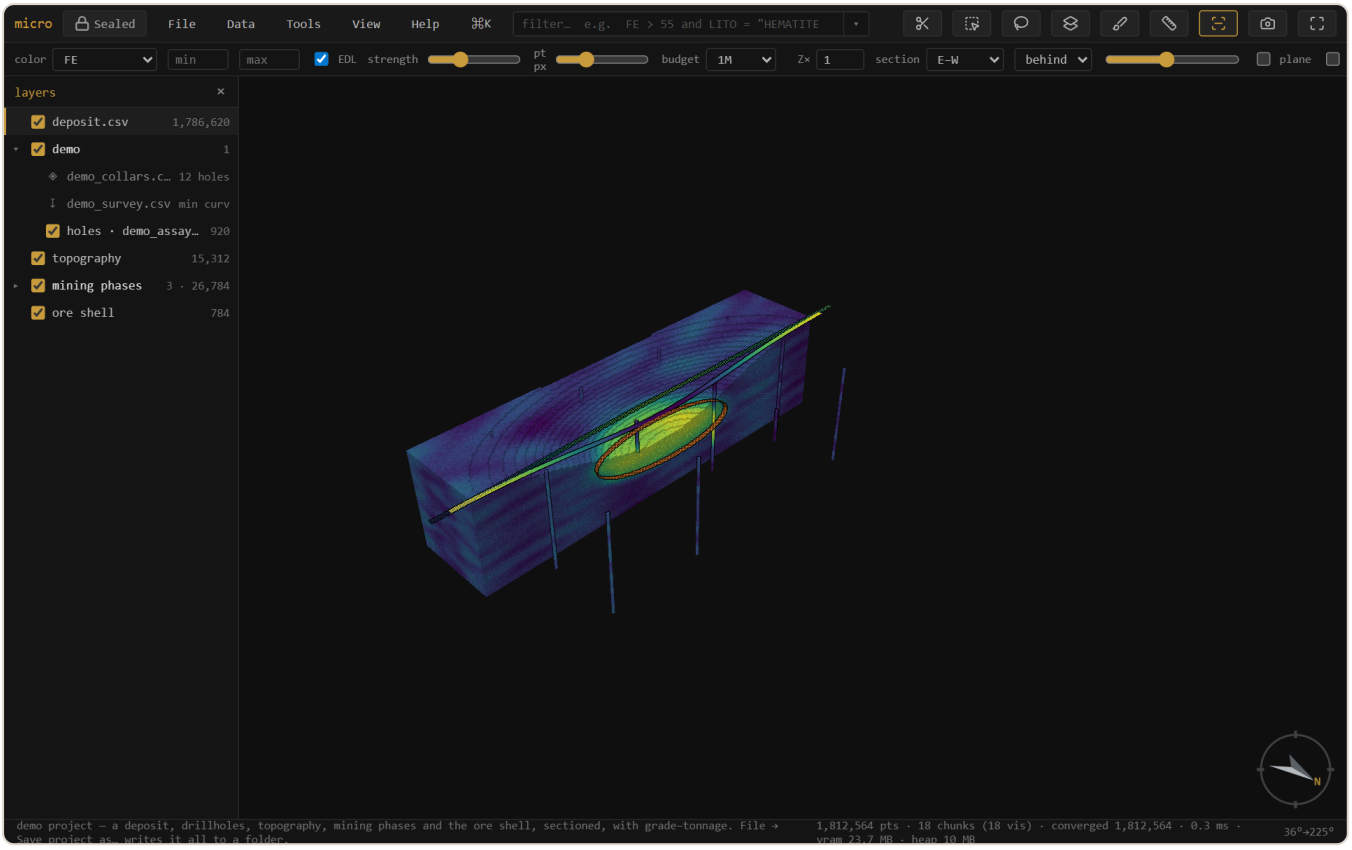
Getting around

Drag to orbit, right-drag (or shift-drag) to pan, wheel to zoom. `f` fits the data. `n s e w` look level that way, `d` is plan, `o` toggles orthographic (needed for a true scale bar). Click a block or point to **pick** it — the record panel shows its full source row, calculated and derived columns included, with a **fields** picker to trim it to the columns you care about.

Vertical exaggeration — the **Z×** box on the instrument rail (or View → Vertical exaggeration) stretches the whole scene vertically so subtle relief or a flat-lying ore lens reads. It's a property of the *view*, not one layer: every layer stretches together and stays registered, and it's display-only — picking, measuring, and section cutting all stay in true coordinates. It travels with bookmarks and scenes.

3 A guided tour

Twenty minutes, no data needed, every major idea once. Open micro and click ▸ **open the demo project** on the empty state — it generates a small mining scene locally: a 1.8M-block iron deposit (cut open on an E–W section), a twelve-hole drilling program threading it, pre-mining topography, three pit-phase shells, and the orange ore-shell mesh. A grade–tonnage window is already computed on the right.



The demo project as it opens: the deposit cut on an E–W section, drillholes threading it whole, the ore shell tracing an orange ring around the high-grade zone.

1 · Look around

Drag to orbit, wheel to zoom, `f` to fit. The deposit is colored by iron grade — the yellow core is the lens. Hold **Ctrl** and roll the wheel: the **section scrubs** through the deposit, and watch the orange ring on the cut face — that is the ore shell's **trace**, the mesh drawn where the plane cuts it, with the topography a thin line above. Press `1` to face the section square-on. In the layers panel (`p`), open **mining phases** and toggle the phase shells; every layer has its own eye.

2 · Ask a question — the filter

Press `/` and type `FE > 55`, then `Enter`. The deposit reduces to its high-grade core, live, and the status bar counts the matching blocks. Now click the ▾ beside the filter box: the **drawer** opens with your expression as a slider. Drag it — the threshold, the expression text, and the model all move together. Use **add condition…** to bring `LITO` in, click a chip to switch lithology, then remove the row with its ×. Nothing you do in the drawer is different from typing — it *is* typing, done for you.

3 · Meet the drillholes

In the layers panel, the **demo** group is the drillhole set: a ◈ **collars** row, a ↧ **survey** row, and the assay intervals. Click ◈ **collars** → **Attribute table…** — twelve holes, and notice `EOH` and `COMPANY`: PIONEER's early scout holes stopped at 150 m, MERIDIAN drilled to the floor. Now ◈ **collars** → **Filter holes…** and type `COMPANY = "MERIDIAN"` — the scout holes vanish from every interval table at once, because the filter lives on the *collars* and flows down the hole relation. Clear it (empty expression), then try **Broadcast column to intervals** → **EOH**: every interval now carries its hole's depth as a real column — color by it, filter `EOH > 200`, export it.

4 · Derive something — the *ƒ* workbench

Select the deposit layer, then **Tools** → **Calculated columns…**. Name a column `CLASS` and type:

```
if(FE > 58, 2, if(FE > 45, 1, 0))   # 2 = high grade, 1 = mid, 0 = waste
```

Below the editor the ladder appears as a **case table** — each condition a row of widgets, each value a slider. Save it, then pick ***ƒ* CLASS** in the color dropdown: the deposit repaints as three grade classes. Go back to the window and drag the `58` slider — the model re-classes as you drag. That loop — define, color, tune — works for any expression over any columns, including the ones you just broadcast.

5 · The curve — grade–tonnage

In the grade–tonnage window, set **density expr** to `SG` (the model carries specific gravity), **cutoffs** to 40, and **Run**. Real tonnes, forty cutoffs, both curves. Hover the plot to read values; ⧉ **table** puts cutoff · tonnes · grade on the clipboard for a spreadsheet. Try **scope: filt** with your `FE > 55` filter still on — the curve recomputes for the filtered core only.

6 · Make the figure

View → **Decorations & figure…**: turn on the scale bar, north arrow, and legend, give it a title, set the background to **white**, and **Export figure**. The PNG on your disk is exactly what you saw — report-ready.

7 · Keep it

File → **Save project as…** (Chromium) writes everything to a folder. Open that folder outside micro: `models/` and `drillholes/` hold plain CSVs, `project.json` the view state — and beside the deposit, `deposit.csv.element.json`. Open it in a text editor: your `CLASS` expression is recorded there as an operation, inputs and all. That file is the audit trail — everything you derived, readable without micro. Reopen the project any time; it comes back exactly as you left it, windows included.

> **Where next —** deeper on each idea: the filter · the *ƒ* workbench · drillholes · join & reconcile · the project store. Your own data works the same way — drag it in.

4 Viewing & sections

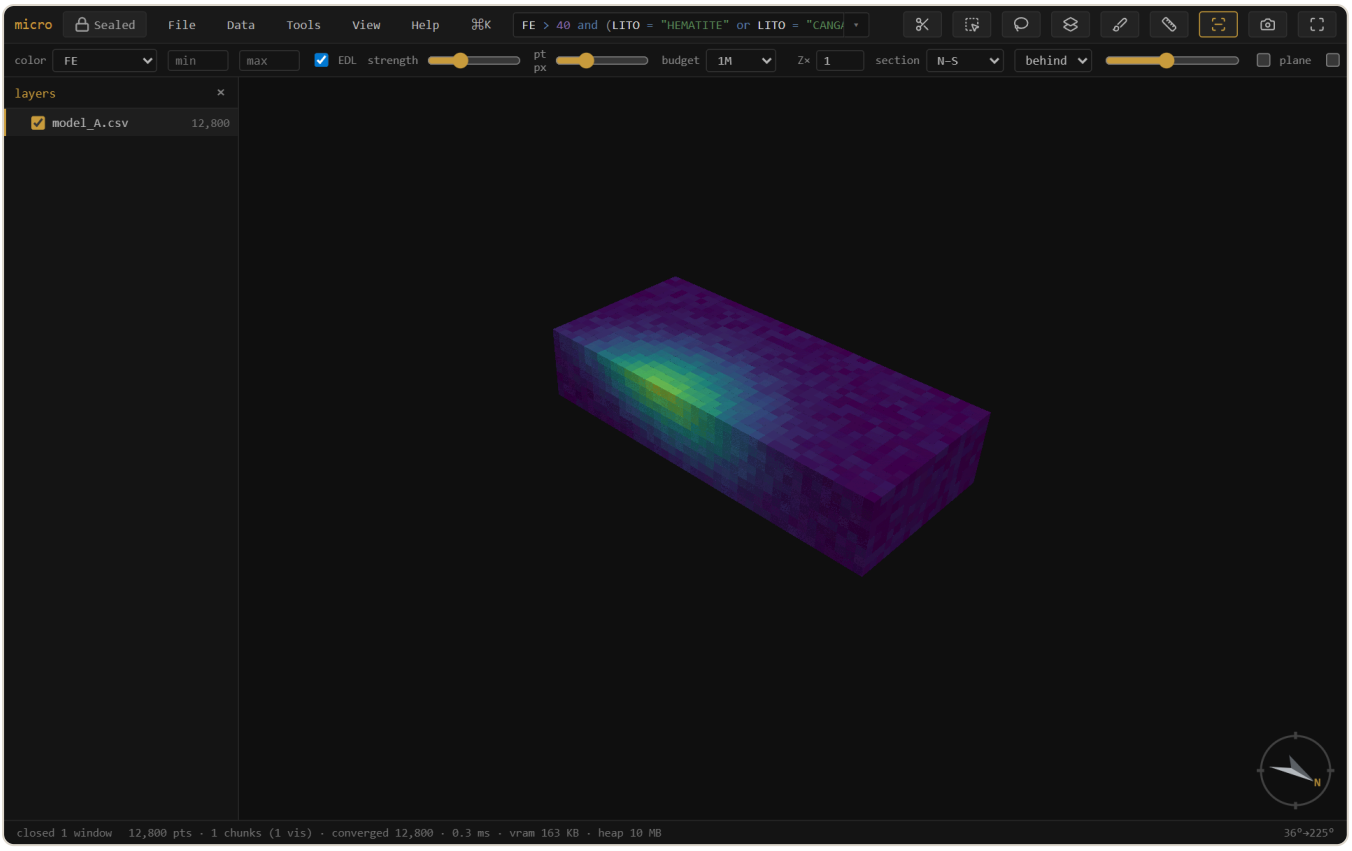
Color

The **color** dropdown drives the active layer: by elevation, a numeric **channel** (a grade), a **category** (a rock type), a painted column (🖌), a materialized column (%), or a calculated column (ƒ — see the ƒ workbench). Pick a ramp per layer (viridis · spectral · magma · turbo · greys) in **Properties** → **style**; the **min/max** boxes clip the scale so outliers stop eating the ramp. On a grade channel, double-click the histogram to add **breakpoints** — **classed** turns them into a solid-color grade-shell legend.

Sections & measure

A section is a live cut, and on block models it is a **true** cut: each block is clipped analytically against the slab faces, so the cut paints a continuous wall — with the block's color — instead of a ragged centroid cull, and the wall itself is pickable.

Meshes and surfaces draw their trace on the wall. A sectioned mesh doesn't hide the cut face: it renders as its **intersection trace** — a closed ore shell becomes an outline ring around the grade zone it encloses, a topo surface a thin line across the top — flattened onto the section face like a drafted section plot. The trace width follows the section-width control. A layer that should stay whole (topo for context, the drillholes threading a cut model) sets right-click → **Cut by sections** → **don't cut**.



A thin N–S slab. The cut face is a continuous colored wall — blocks are clipped against the plane, not dropped by centroid — and clicking the wall picks the block behind it.

Pick a preset (plan · N–S · E–W), press `k` and drag a line (⇧ snaps 15°, ⇧⌥ 5°), or type a plane's dip-direction + dip. `x` toggles it, `l` faces it, `⇧L` faces its back. `,` `.` step one slab at a time, and **Ctrl+wheel** scrubs the position continuously; **Alt+wheel** thickens or thins the slab. **Follow** makes the camera ride along as you scrub; **2D** locks the view flat onto the section — drags pan instead of orbit — for section-by-section interpretation. `m` then two clicks **measures** 3D · plan · Δz between records. Per layer, **Cut by sections** picks how the section treats it: **follow** (the slab), **keep front only** / **keep behind only** (a half-space split at the plane), or **don't cut**.

Picking & the record panel

Click any block, point, or interval to **pick** it: the record panel (right) shows the full source row — every column, including calculated, estimated, painted, and broadcast ones, each labeled by kind. The **fields** picker (search + all/none) trims the panel to the columns you care about, per layer, and the choice persists with the project. Picking works on section walls too — the block behind the cut face answers. In the attribute table the link runs both ways: click a row to pick it in the scene, pick in the scene to highlight the row.

Measure

`m` arms the tape: click two records and read **3D distance · plan distance · Δz** in the status bar — bench heights, hole spacing, the plan width of a zone. The measurement anchors to picked records (not screen pixels), so it is exact in world coordinates and unaffected by vertical exaggeration. Press `m` again or `Esc` to put the tape away.

Block edges

`⇧B` (or View → block edges) outlines every block — the classic wireframe-over-solid look for checking sub-blocking or lattice alignment. Each layer can override the global toggle from its context menu (always / never / follow).

The attribute table & the scan

`t` opens the layer's rows as a draggable, layer-pinned window — windowed reads, so a 1.8M-row model scrolls cold. **matches** shows only the filtered rows; click a row to pick it in the scene. In **Properties → columns**, **scan columns** streams the whole table once to fill n · mean · std · quantiles · histogram per column — the scan sees calculated and derived columns too.

The stats table

Stats… (right-click a layer, or the palette) is the grouped summary report: pick columns, an optional **group by** (the category channel or any painted column — *assign domains, then stats by domain* is the classic resource table), and an optional **weight expression** (`SG` for density-weighted means — quantiles stay unweighted and the window says so). Same chassis as grade–tonnage and swath: scope (all / filt / sel), Run, ⧉ **table** TSV copy, **save as recipe…**, and the setup persists with the project.

5 The filter & its widgets

`/` focuses the filter box. It speaks a familiar expression language over **every** column — source, calculated, painted, materialized, joined:

```
FE > 55 and LITO = "HEMATITE"
AU_OK between 0.5 and 3  or  DOMAIN in ("ORE", "HALO")
SIO2 > 10 and not (LITO = "CANGA")   # comments are fine
```

Matching elements isolate; everything else drops out of the render, the table, and any analysis. `Enter` applies, `Esc` clears. The box is **syntax-highlighted** as you type, autocomplete offers columns, functions, and category values, and unknown names get a *did-you-mean*. The full language — operators, functions, quoting, comments — is in the reference.

On big files the filter stays fast for three reasons: Parquet models skip whole row-groups their footers prove can't match; CSV and `.dm` files skip stretches via the project's band index; and after one complete scan the touched columns stay **pinned in memory** (see the project store). The result is that dragging a widget re-filters millions of blocks in milliseconds.

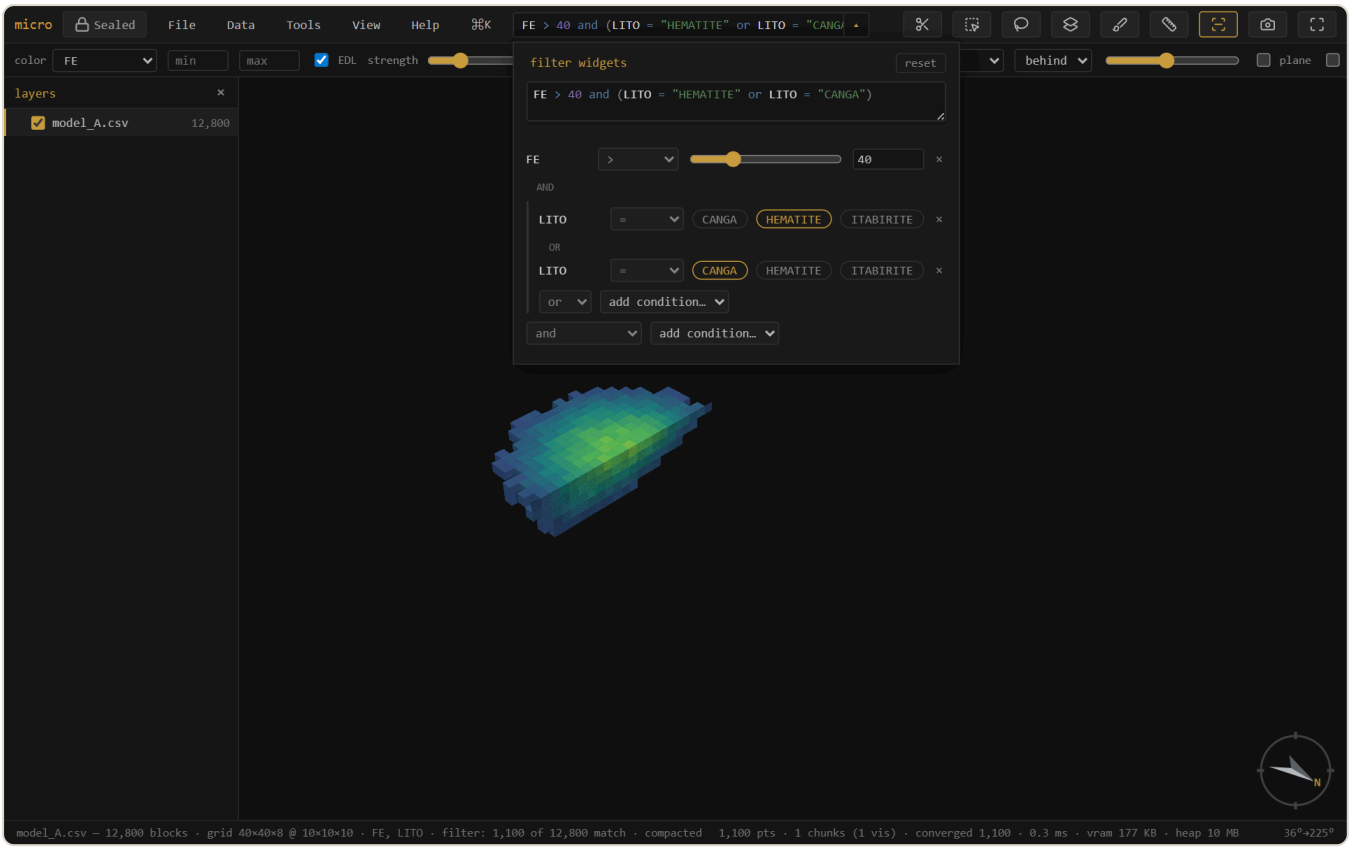
The filter drawer — the expression as live controls

The ▾ button glued to the filter box opens the **drawer**: your current expression projected as widgets. The expression stays the single source of truth — every control rewrites its own span of the text, so the box updates as you drag and your formatting survives.

- **Numeric comparisons** become sliders with a number box; `between` becomes a two-slider range row. The **operator is a dropdown** (`> ≥ < ≤ = ≠ between`) — switching to `between` converts the clause in place.
- **Category comparisons** become chips — click a value to swap it, and with `in (…)` chips toggle membership. The op dropdown offers `= ≠ in not in`.
- The **AND / OR pills** between rows are clickable — each one switches its own joiner in the text.
- **Brackets are real**: parenthesized groups render as indented blocks with their own *add condition* row (inserting inside the brackets), and the top-level *add* offers `and ( … )` to seed a new group or `( all ) or …` to wrap everything so far and start an alternative branch.

- Every row has a **× to remove it** — the clause and its joiner go together, removing the last condition inside brackets removes the whole group, and removing the only condition clears the filter.
- The drawer includes a **multiline editor** (with `#` comments) for the expression itself, live-validated; **reset** restores the filter as it was when the drawer opened.

Add a condition from the dropdown, then reshape it entirely with the widgets — nothing is locked in at add time.



The drawer projecting `FE > 40 and (LITO = "HEMATITE" or LITO = "CANGA")` — a slider with its operator dropdown, a bracketed group with clickable chips and an OR pill, a remove × on every row. Dragging the slider rewrites the expression above, live.

6 The ƒ workbench — derived columns

A derived column behaves exactly like a source column: the filter, color ramps, stats, grade–tonnage, swath, the record panel, and every export see it. micro derives columns five ways, and each records *how* in the layer's provenance:

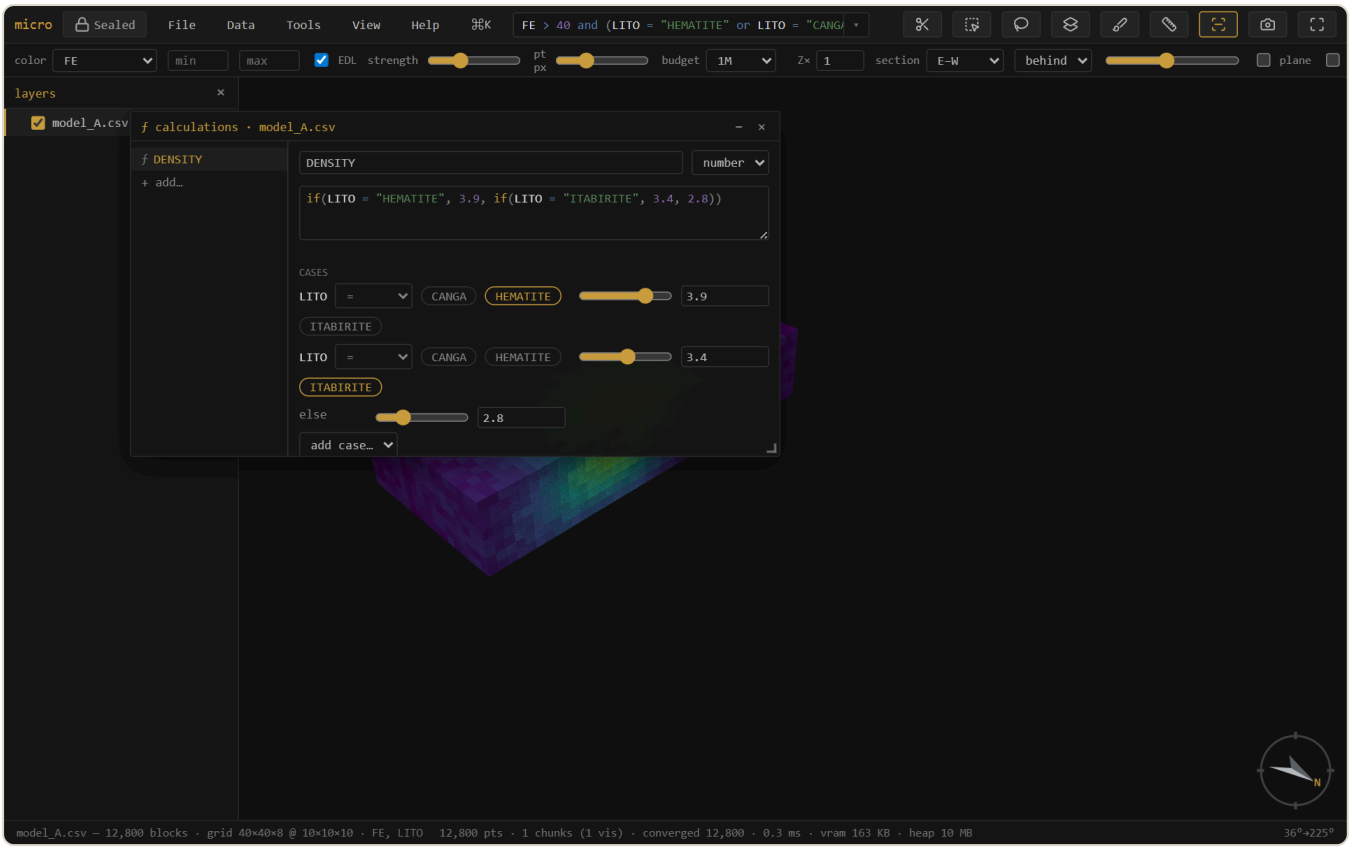
Kind	Made by	Stored as
ƒ calculated	an expression (`FE + 0.4 * MN`)	a definition — computed on the fly
materialized	freezing a ƒ into a stored column (**materialize** → **stored**)	a Parquet column beside the source
painted ✎	brushing over the scene (`b`), or flag / assign-domains by a solid	a category column
joined	a key join or an own-lattice spatial join — see Join	materialized
broadcast	a collar column landed on drillhole intervals — see Drillholes	materialized

The Calculations window

Tools → Calculated columns… (or the **ƒ** button in Properties → columns) opens the calculations workbench: your ƒ columns listed on a rail, one editor on the right. The editor is multiline and syntax-highlighted, validates live against the layer's **full** schema — so a calculation can reference source columns, painted columns, materialized columns, **and other ƒ columns** (`FE_EQ` then `DBL = FE_EQ * 2`) — and the type is a visible choice (auto / number / text). Saving a definition immediately updates any live filter or color that uses it.

Typing the expression is the primary interface, but below the editor the definition also **projects as widgets**, the same contract as the filter drawer:

- An `if(…)` ladder renders as a **case table** — condition | value, row per case, plus *else* and *add case*. Density by lithology (`if(LITO = "HEM", 3.9, if(LITO = "ITA", 3.4, 2.8))`) becomes a table you edit with chips and sliders.
- Any other expression lists its **numeric constants as sliders** — tune the coefficient in an equivalence formula and, if a filter or color uses the column, watch the model respond live.



The ƒ workbench: a density-by-lithology `if()` ladder — the highlighted expression on top is the truth; below it the same definition as a case table of chips and value sliders.

Widget edits auto-save an existing column; typed edits wait for the explicit save. **materialize → stored** freezes a ƒ into a real Parquet column (it survives even if its inputs are later removed, and scans get faster); **save as recipe…** writes the whole column set as a re-runnable YAML — apply your standard derived columns to next month's model in one click.

Color by ƒ

Pick `ƒ NAME` in the color dropdown and micro computes the column once for display — a cached realization, recomputed automatically when the definition or anything it references changes, never written to disk (the definition is the provenance). With the constant sliders this closes the loop: drag the coefficient, the model recolors.

7 Drillholes

A drillhole set is **three tables bound by the hole id** — collars (one row per hole), survey (deviation stations), intervals (from/to samples) — and micro keeps that structure first-class rather than flattening it away.

The set in the layers panel

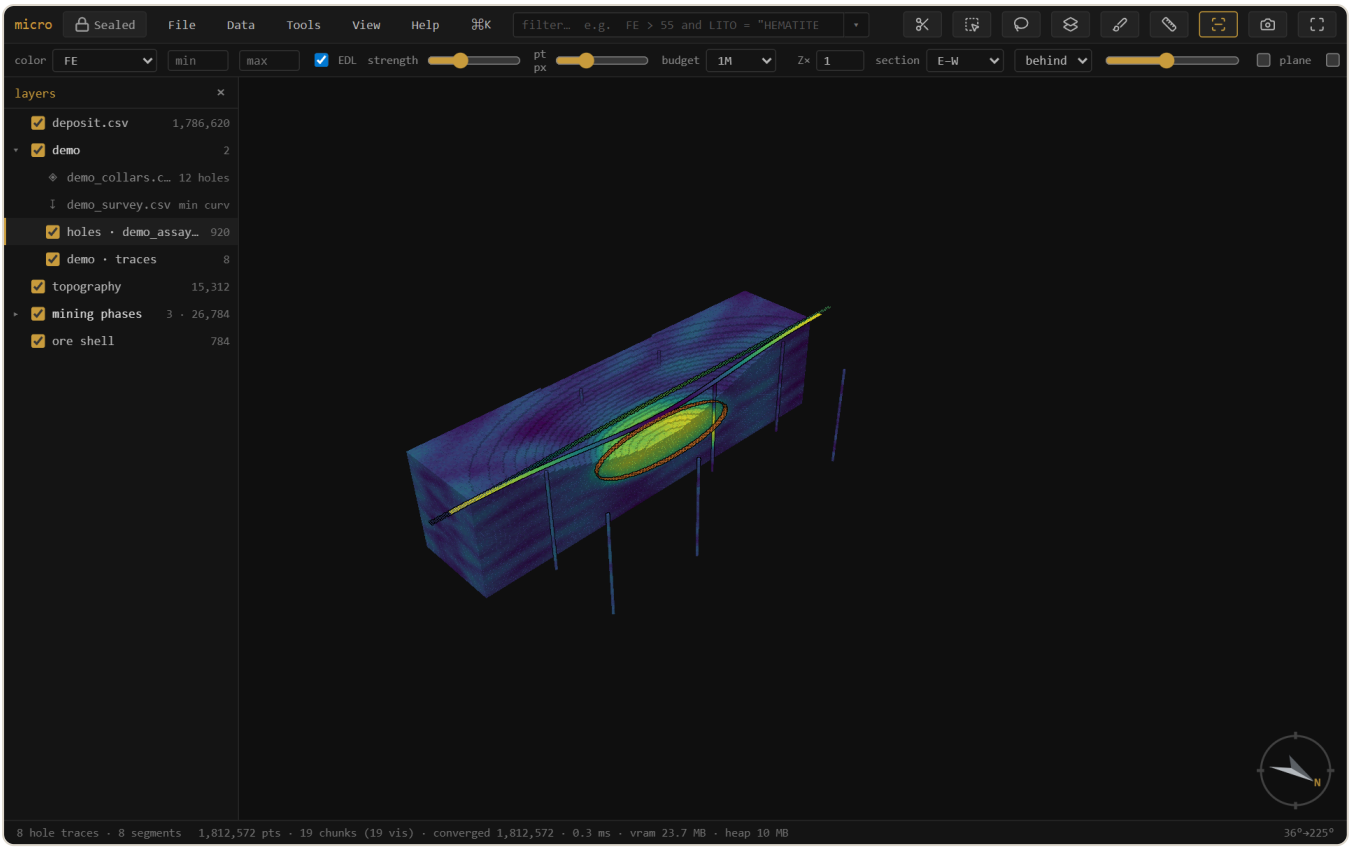
Every load lands as a **set group**: the group is the drillhole set, with a ◈ **collars** row (hole count), a ↧ **survey** row (desurvey method), and one interval layer per interval table. The rows are live — click one for its menu:

- ◈ **collars** — **Attribute table…** (the collar file as a table), **Filter holes…**, **Broadcast column to intervals**.
- ↧ **survey** — **Attribute table…**, **Re-desurvey…** (method: minimum curvature / balanced tangential / tangential; dip convention), **Show hole traces** (the full collar-to-EOH paths as thin grey sticks, filter-aware).
- The **set group** itself — hole traces, **Add interval table…** (assays + lithology + geotech share one geometry as sibling layers).

A drillhole set in the layers panel: the group is the set, with ◈ collars (hole count) and ↧ survey (desurvey method) rows above the interval layer. Click a row for its actions.

The hole filter — filter by collar data

Filter holes… takes an expression over the **collar** columns — `EOH > 200`, `COMPANY = "ACME"`, a campaign year — and applies the verdict through the whole set: only the matching holes' intervals stay visible (in every interval table at once), the traces redraw for the matching holes only, and the interval filter box then works *within* the kept holes. Nothing is copied or materialized — it's a live mask driven by the hole relation, and it persists with the project.



The hole filter at work: `COMPANY = "MERIDIAN"` on the collars keeps that campaign's holes — the interval count drops and the traces redraw for the matching holes only, across the whole set.

Broadcast — collar data on the intervals

Broadcast column to intervals lands a collar column on every interval, matched by hole id — so `EOH`, the logging company, or a campaign field becomes a real interval column you can filter, color, and export by. Numeric columns materialize to the project's column store; text columns become category columns with a legend. The operation is recorded with its inputs, like every derivation.

Display

Intervals render as true cylinders (**Stick radius…** sets the world-space radius). Color by any interval column; sections cut the sticks true, and the set usually reads best with the model sectioned and the holes set to **don't cut**, threading whole through the opened deposit — which is exactly how the demo project arranges itself.

8 Selection & domaining

Select with `r` (rectangle) or `v` (lasso): what you *see* gets selected (occlusion-honest, all visible layers), tinted gold — **shift** adds, **alt** subtracts, `Esc` clears. A plain release puts the tool down (left-drag orbits again); release **with shift or alt held** and the tool stays armed for the next gesture — you're clearly accumulating. Middle-drag orbits and right-drag pans either way, so navigation never leaves; Settings → interaction offers one-shot and always-armed instead. The **select-through** toolbar toggle flips to the extruded tube: everything whose position falls inside the shape, occluded included.

Select by solid / surface (Tools menu, or right-click a mesh) classifies records against a closed mesh by winding number: **select** inside/outside, write a **flag column**, or a **fraction-inside** 0–1 column. Open surfaces (topo, weathering horizons) classify **above / below / between**. **Assign domains** runs the whole coded-model loop — pick the domain solids and a column name, and every block goes to the solid holding its largest fraction.

The whole evaluation is **recipe-able**: **save as recipe…** in the dialog captures geometry, mode, engine, targets, and the column — so *"flag the oxide onto this month's model"* is a saved file that re-runs from **Tools** → **Recipes** or as a routine step, layers resolved by name.

Paint a column (`b`) is interactive domaining by hand: a category column you fill by brushing over the scene. Painted values are real columns the filter, table, and export all see.

Selection → column… (right-click the layer, or the palette) materializes the current selection as an explicit **0/1 column** — every row gets a value, so it filters (`SEL = 1`), edits like any painted column, and rides every export. Selections themselves also persist in the project.

9 Join & reconcile

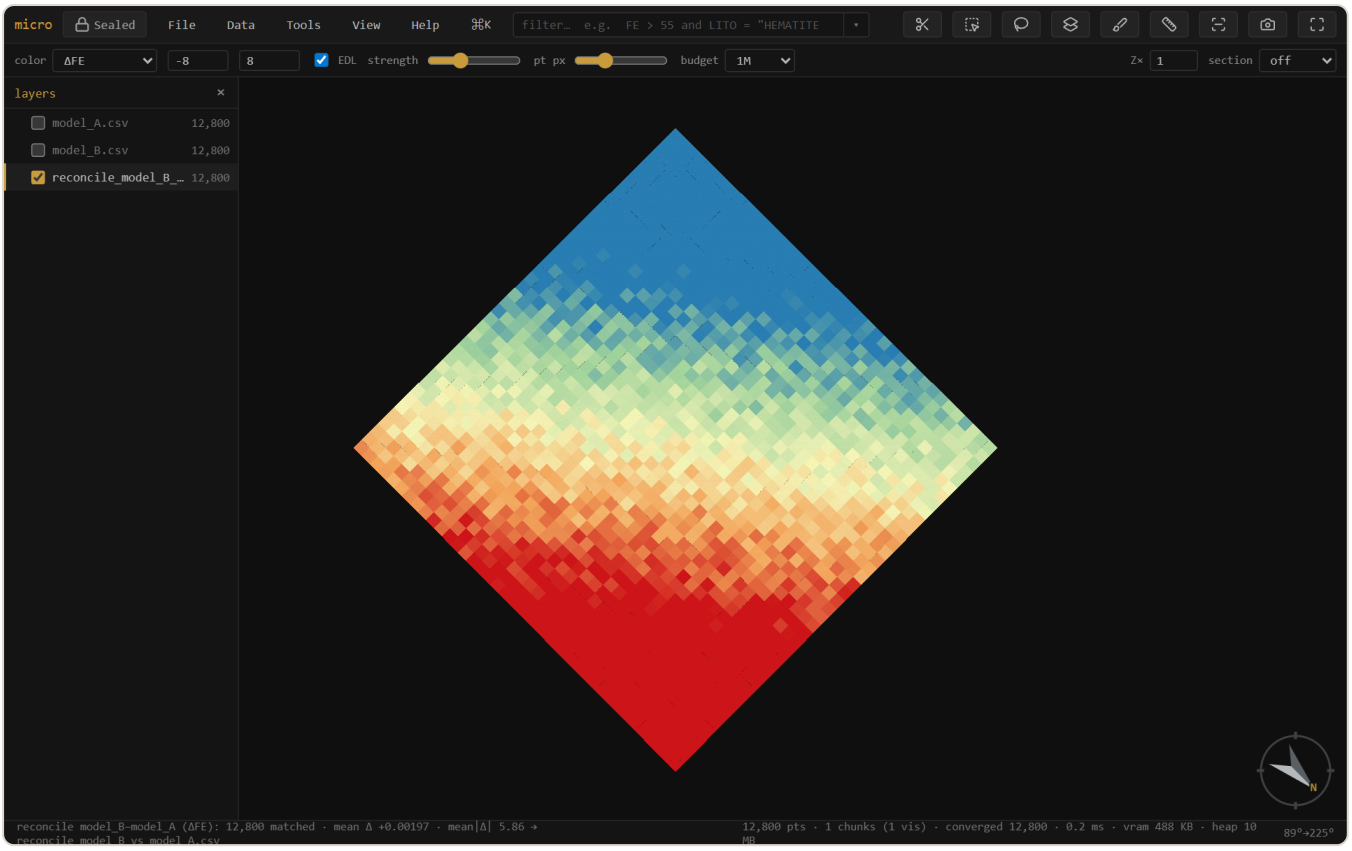
Two model versions, or a lookup table? The **join tab** (Properties → **join**) brings data across — and it follows one rule: **if the operation keeps this layer's records, the result lands as columns on it; only an operation that changes the record set makes a new layer.**

Only the 3D selection — a toggle above the run buttons scopes any join or reconcile to the rubber-banded zone: lasso the pit area, reconcile *just it*. Each side uses its own selection where it has one, and a scoped run is stamped `selection: true` in the lineage.

- **Key join → columns here** — match rows by a key column (a block id, a domain code) and the brought columns land on this layer: numeric as materialized columns, text as categories, name collisions prefixed. The classic domain→density lookup is one join, no new layer.
- **Spatial join → columns here** — with the target grid set to *this model's own lattice*, the other model's columns resample onto your existing cells (mean / sum / count, optionally weighted; categories by majority) and land as columns. A compatibility banner tells you whether the grids share a lattice, or why not.
- **→ new layer** — the secondary, export-shaped verb: a joined copy as a new layer/file. Spatial joins onto a *third* lattice (common-finest / coarsest) stay layers — they genuinely change the record set.

Reconcile is the preset: it produces a colored **Δ map** (this model − reference) on a common lattice, with a diverging ramp centered on zero — blue where the new estimate is lower, red where it's higher. The summary reads **matched · mean Δ · mean|Δ| · added · removed**.

Sub-blocked models reconcile by **volume**: each variable-size block is aggregated onto the reference (parent) grid weighted by the space it fills, so a sub-blocked estimate compares cleanly against a regular one at the parent resolution.



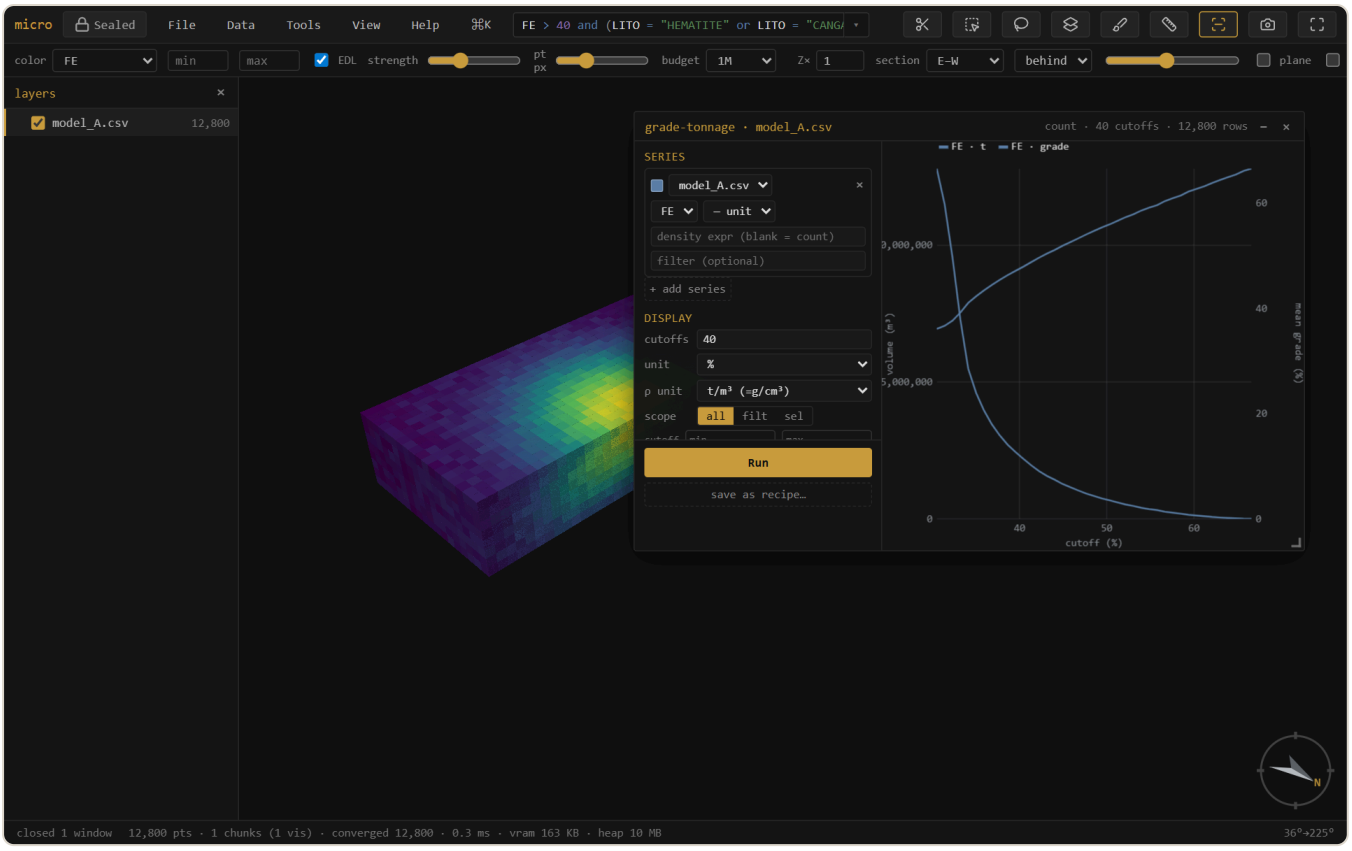
Reconciling two model versions. Global mean Δ is ≈ 0, but the Δ map exposes a clear **conditional bias** — the new estimate runs low in the north (blue) and high in the south (red). A global check would have missed it entirely.

> **Tip —** The Δ ramp auto-clips to the full range, which extreme blocks can widen. Tighten the **min/max** boxes (e.g. ±2× the mean|Δ|) to make the local pattern read, as above.

10 Grade–tonnage

Grade-tonnage… (right-click a model, or the palette) draws tonnage and mean grade above cutoff. It shares the swath's interface: a config rail of **series**, each one a **layer + grade column + declared unit + density expression + optional row filter** — configure, then **Run**. Because series pick their own layer, model-vs-model curves (this estimate against the last one) are one window; on a reconcile layer the reference/new pair is pre-seeded. Derived columns — calculated, materialized, joined — are all offerable as the grade or the density.

- **Density** is a free expression (`SG`, `2.7`, `SG * PARTIAL`) — tonnage = density × block volume, with a declared density unit converted to t/m³ so tonnes are tonnes. Blank = count/volume.
- **Units are declarations**, not display toggles: declare what each series' grade *is* (%, g/t, ppm, oz/t) and a **plot unit** — series convert onto the shared axis honestly, so a %-model overlays a ppm-sample set without lying.
- One streamed scan per Run caches each series' cumulative curve, so **cutoff count, plot unit, and manual axis ranges are live afterwards** — no re-scan.
- ↓ **png** saves the plot, ⧉ **copy** puts it on the clipboard, and ⧉ **table** copies the numbers as TSV — cutoff · tonnes · grade · contained metal per series, ready for a spreadsheet.



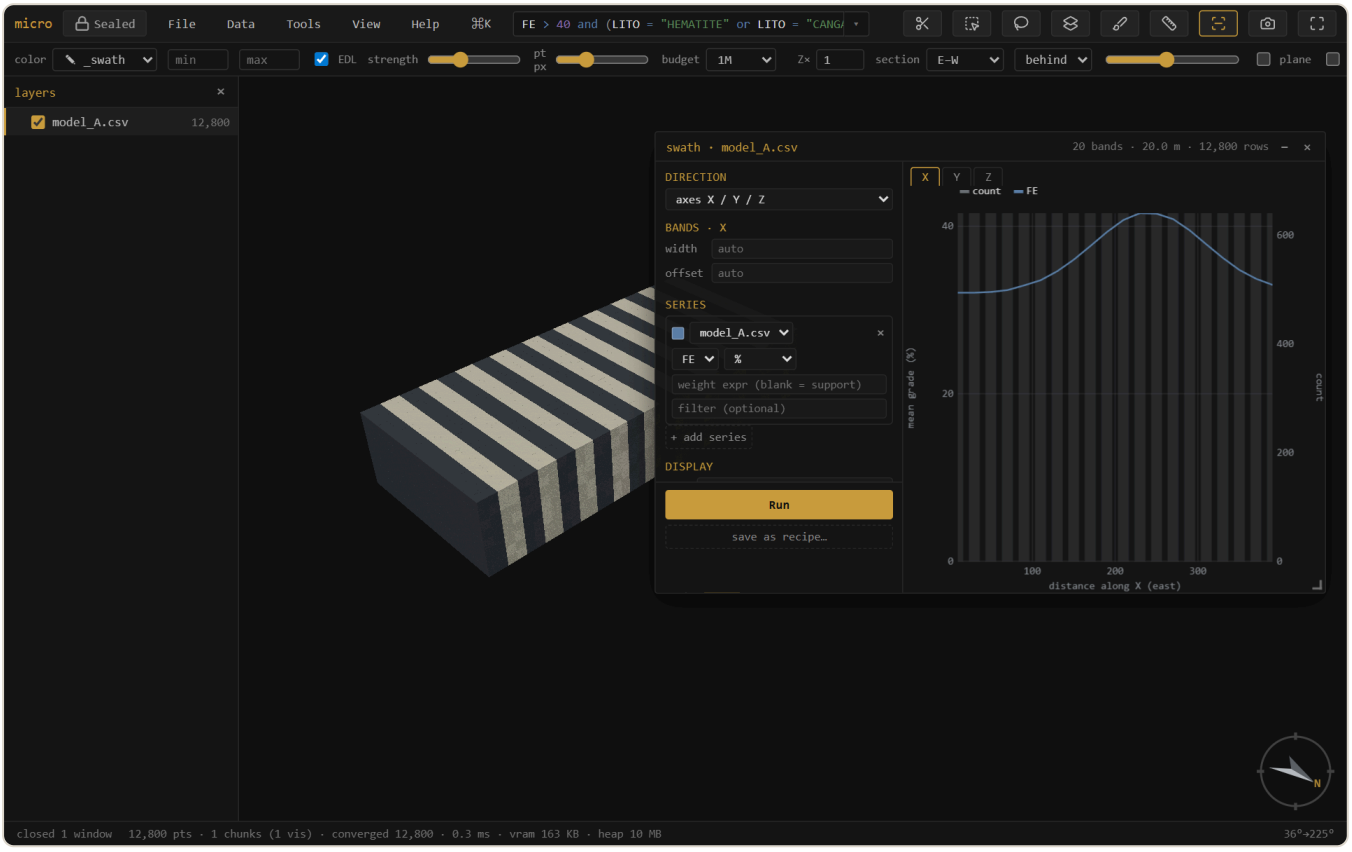
Grade–tonnage: tonnage (left axis) falls and mean grade (right axis) rises as the cutoff climbs — the recoverable-resource trade-off at a glance. The rail holds the series, units, density expression, scope, and manual ranges.

11 Swath / drift

Swath plot… shows mean grade per band along a direction — the classic drift check for local bias. The direction can be the grid **axes** X/Y/Z, an oriented trio from **dip-direction / dip / rake** (across-structure · along-strike · down-dip), or a single **trend / plunge**. Series work exactly like grade–tonnage's: layer + column + declared unit + a free **weight expression** (blank = block volume) + an optional per-series row filter — so kriging vs nearest-neighbour vs declustered samples overlay in one plot.

One **Run** streams everything once into fine bins; after that, **band width and offset re-bin instantly** — and they are **per direction**, so your E–W drift keeps its 25 m bands while the bench-by-bench Z profile keeps 10 m. Manual axis ranges, ⭳ png / ⧉ copy / ⧉ **table** (band · count · weight · per-series mean, TSV) match grade–tonnage.

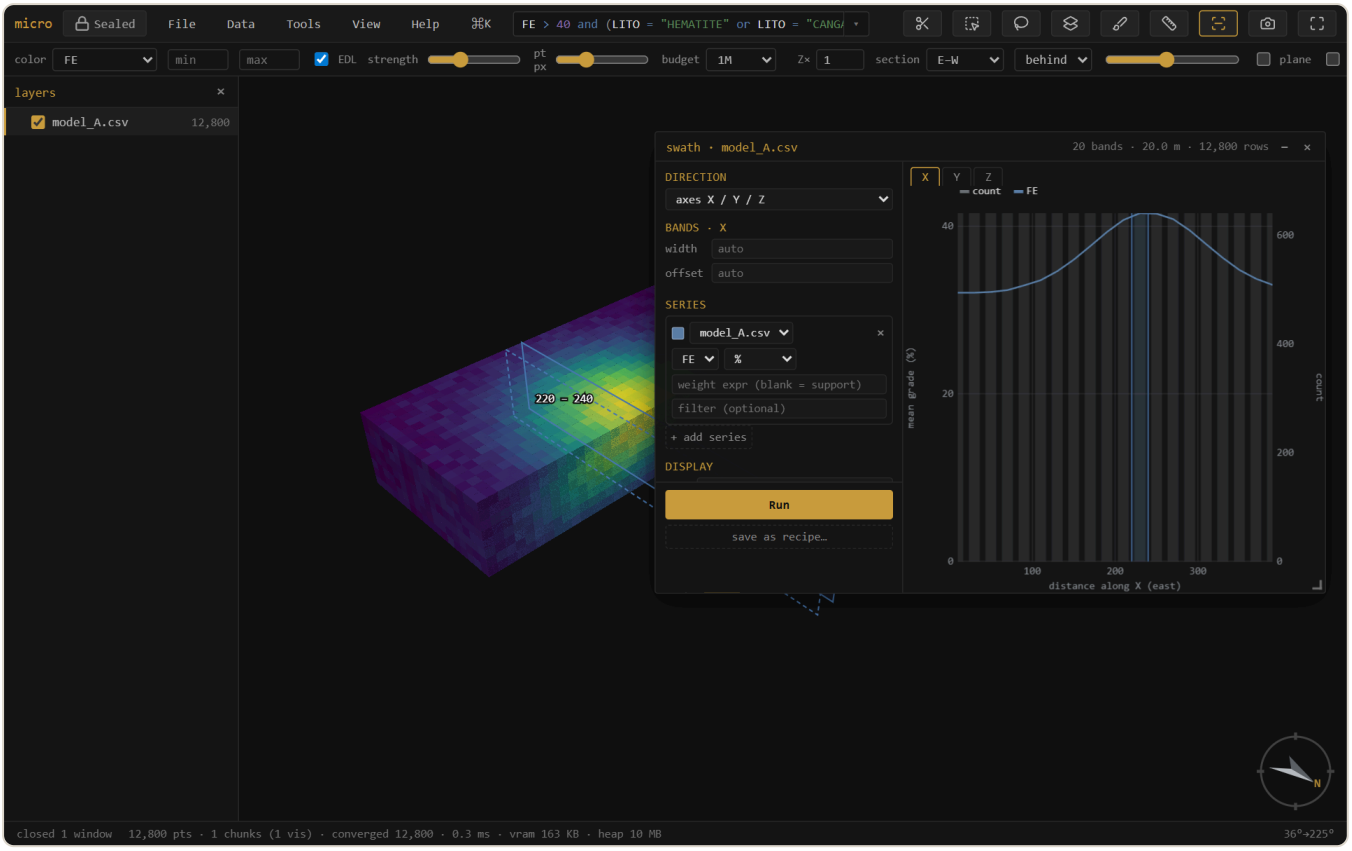
Zebra paints the alternate bands onto the 3D, so the slabs you see in space *are* the bins in the plot.



A swath along easting, with the bands zebra-striped onto the model. Because the plot and the stripes share one direction / width / offset, you read the profile against the deposit it came from.

The band ghost — hover the plot, see the slab

With **locate on 3D** on (it is by default), hovering the swath plot lights that band on the 3D as a translucent slab outline — so a dip in the profile points straight at the benches or the zone that caused it. **Drag** across the plot to hold a band *range* (violet), then turn it into a real 3D **selection** right from the menu that appears — and from there it's linked brushing, a 0/1 column, or an export like any other selection.

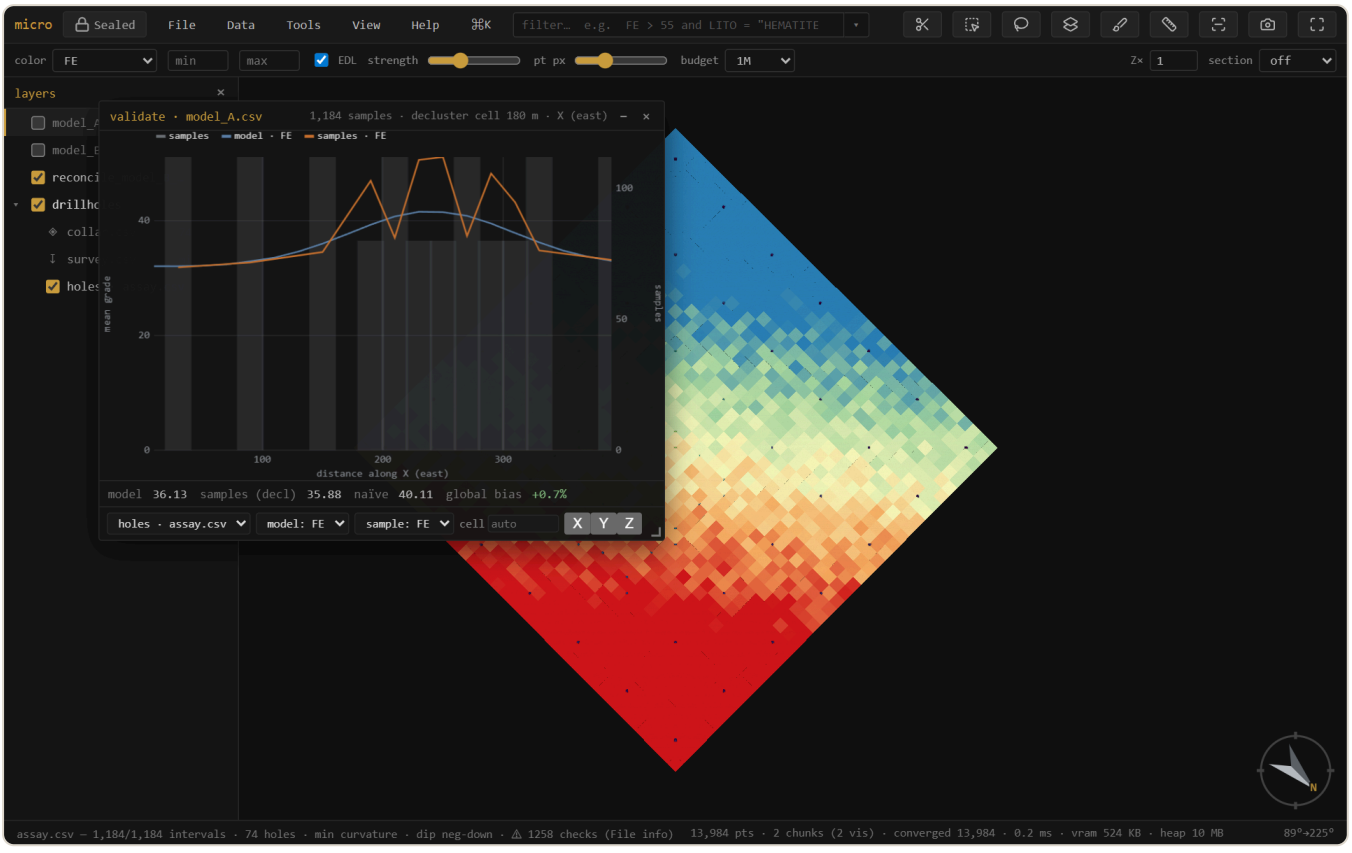


The band ghost: the bin under the cursor (shaded in the plot) is the slab outlined on the model. Drag a range and the menu offers *Select rows in…* — the plot becomes a selection tool.

12 Validate vs drillholes

Is the model consistent with the samples it was built from? **Validate vs drillholes…** takes a block model and a drillhole layer, **declusters** the sample grades (spatial sampling over-drills high grade, so a naïve sample mean is inflated), and reports the **global bias** (model mean vs declustered sample mean) plus a **model-vs-sample swath** along X/Y/Z.

> **Note —** The validation window is **temporarily hidden from the menus** while it's rebuilt on the new analysis chassis (per-series units, weight expressions, scopes). The machinery it uses — declustering, sample swaths — is unchanged; meanwhile the swath window already overlays model vs drillhole-sample series directly.



Validation. The naïve sample mean (40.1) is inflated by clustered infill drilling; **declustered** it's 35.9, essentially matching the model (36.1) — a **+0.7% global bias**. The swath compares model (blue) and sample (orange) grade along easting, slab by slab.

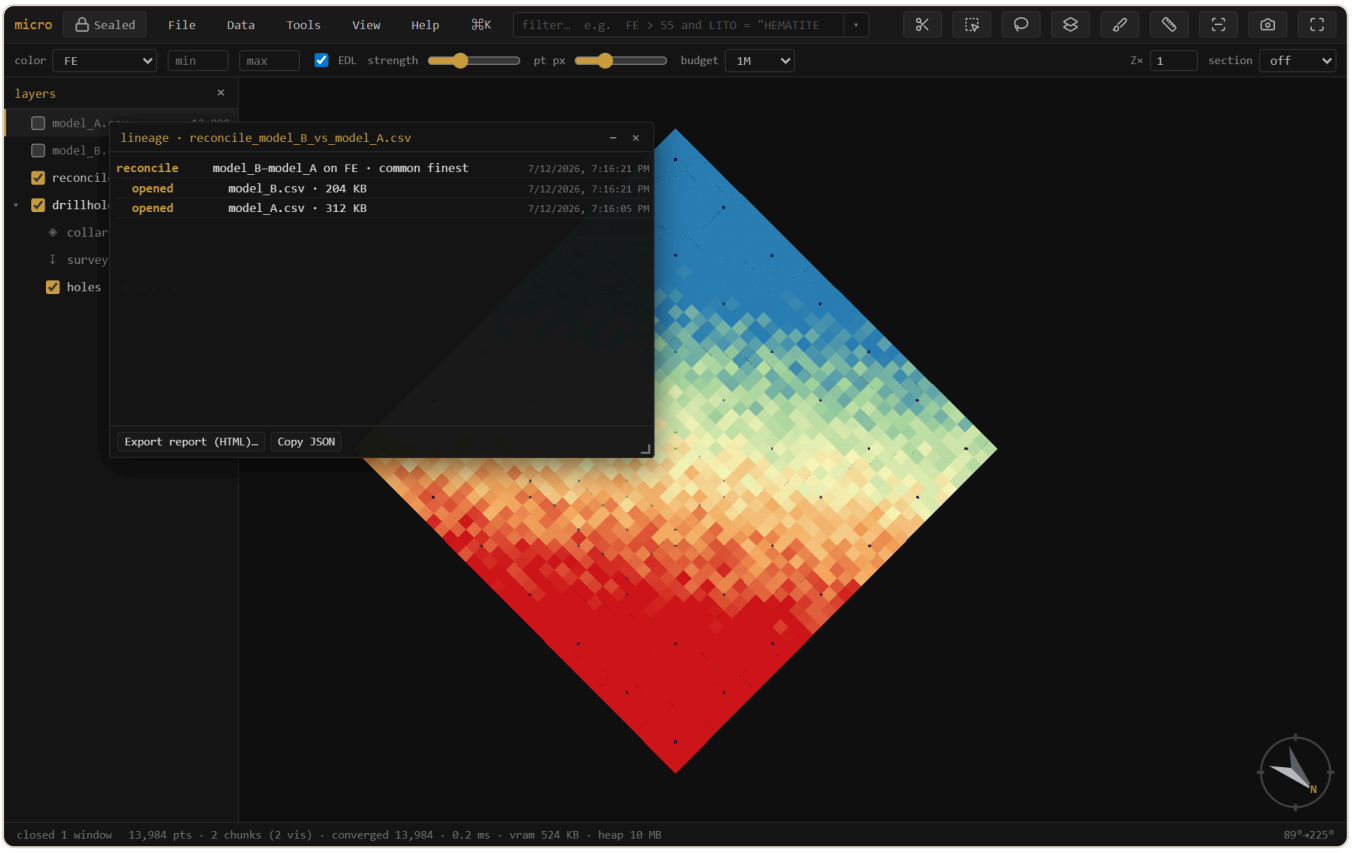
> **Note —** Samples are point support, blocks are block support, so a *global* grade offset between them can be real (change of support), not an error. The swaths are what flag *local*, conditional bias.

13 Linked brushing

The grade-tonnage and swath windows carry an **all / filt / sel** scope. **sel** follows the 3D selection: rubber-band a bench, a domain, or a suspect zone and the plot recomputes for just those blocks, live, as you brush. **filt** follows the **filter box** instead — type `LITO = "HEMATITE"` and every open analysis window scoped to it re-runs with each new expression. The analysis becomes interactive regional exploration, not a whole-model summary.

14 Lineage & provenance

Every derived layer *and every derived column* remembers *how it was made*. Opening a file is a leaf; a join, reconcile, or optimize wraps its source layers with the operation, its parameters, the build, and a timestamp — a tree. An estimated, broadcast, or joined column records its operation and inputs the same way, in the project's element manifests (see the project store). **Lineage…** shows the layer tree and exports a readable HTML **report** you can attach to a sign-off. Parquet exports carry the lineage embedded in the file, so the provenance travels *inside* the data.



The provenance tree for a reconciliation: the result wraps the two opened models. Export it as an HTML report, or read it back from any Parquet micro wrote.

15 Grids & surfaces

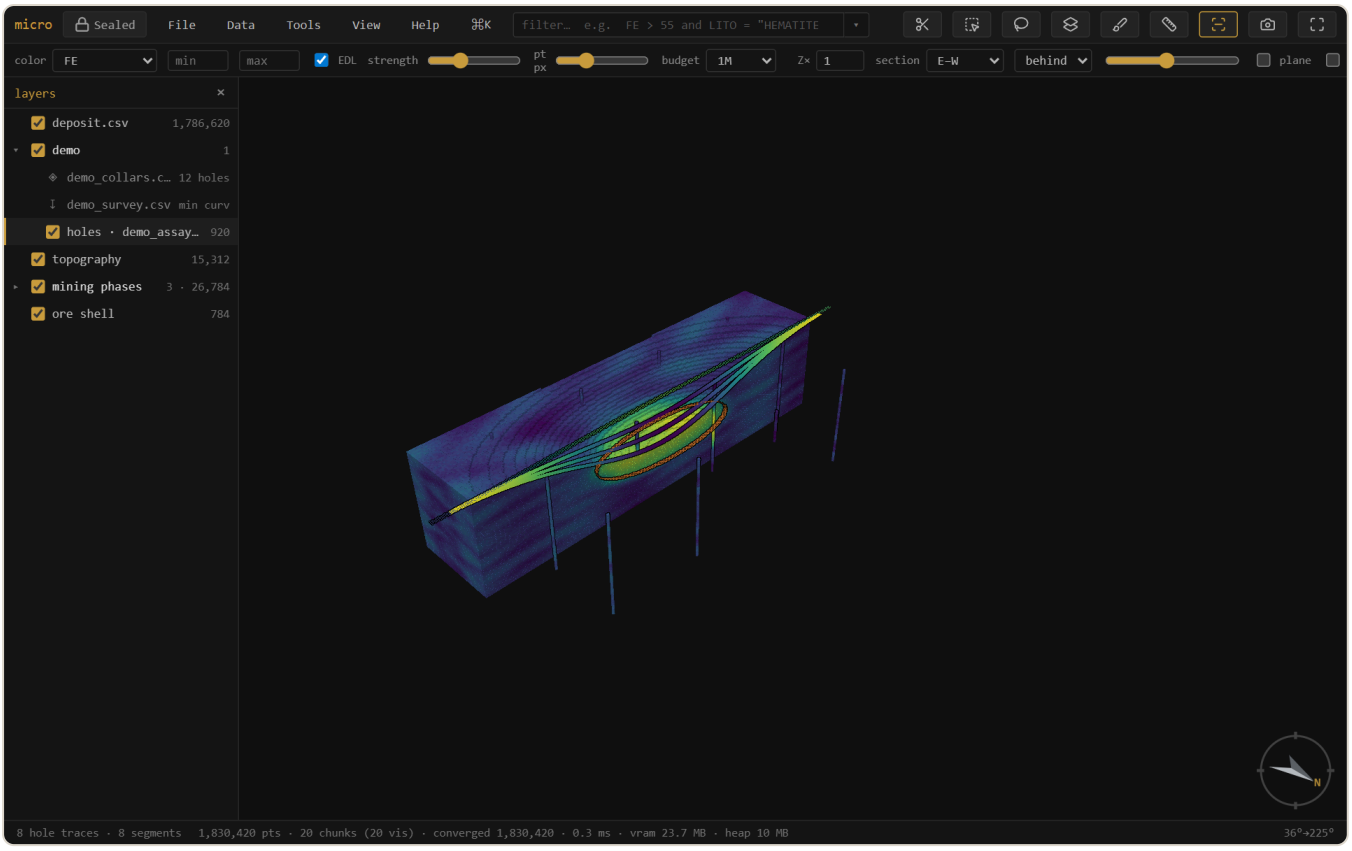
GeoTIFF / Surfer `.grd` / ESRI `.asc` open as heightfield layers — an elevation point cloud where cells are pickable records (huge DEMs decimate for display while the full grid stays for evaluation). A mesh right-click → **Rasterize to grid…** renders it onto a DEM (the surface↔grid converter, with fill / extrapolation for gaps). **New… → grid / block model** creates empty grids from parameters, covering a layer's bounds, or — **cover compatible models** — on a lattice that covers and is compatible with several models at once (the reconcile target). Grid layers export as ESRI ASCII.

Show a grid as a surface — relief, flat, and draped

Right-click a grid → **Reinterpret as** → **surface** renders it as a solid, smooth-shaded **relief** mesh — a DEM reads as terrain, not a gappy point cloud. In **Properties** → **Style** a surface then offers:

- **Drape** — color it by its own *elevation/value*, or by **another loaded grid's values** (grade / geochem / slope over topo), sampled at each point. Paint the terrain with whatever channel you like. (The drape source is a 2D grid, sampled by position.)
- **Shape → flatten @ z** — turn a 2D data grid that has *no* elevation (a grade or geochem grid) into a **horizontal colored sheet** at a chosen level, colored by its value. A plan sheet, floating where you put it.
- **Band** — for a **multi-band GeoTIFF**, a picker to choose which band is shown (a GeoTIFF's bands are like a grid's columns). Switch it anytime; it re-reads that sample.

Surface, drape, flat, and band all **persist in the project** — reopen and the terrain comes back exactly as you left it. Sectioned surfaces draw their trace on the cut face (see Sections), and layer opacity applies — a translucent topo over an opened pit.



Relief surfaces in the demo: the pre-mining topography and the pit-phase shells over the opened deposit, each an ordinary layer with its own eye, opacity, and section behavior.

Surface tools: extend & combine

Extend to footprint… (right-click a surface — mesh or grid) grows it to cover a chosen extent: **nearest** holds the boundary z flat (the "extend the outer ring" move), **trend** continues the fitted dip. A topo that stops short of the model no longer leaves blocks unclassified — extend it, then flag above/below covers everything. **Combine surfaces…** rasterizes two or more surfaces onto one lattice and resolves the overlap by a rule: **min** (topo ∧ weathering base — take the lower), **max**, **first** (list order is priority — patch an end-of-month survey into the original topo), or **mean**. Both produce **grid layers** — exact and fast for flags and reconcile — with the rule recorded in the layer name and its lineage; export .asc anytime.

16 The project store, inside

A micro project is a plain folder, readable without micro. This section is its anatomy, because auditability means being able to open the hood:

```
my-project/
├─ project.json                  view state — layers, styling, camera, windows
├─ models/
│  ├─ deposit.parquet            the model, Morton-sorted (spatially indexed)
│  ├─ deposit.parquet.element.json   the DATA MODEL — columns, ops, provenance
│  ├─ deposit.parquet.meta.json  cache: discovery + stats + band index
│  └─ deposit.parquet.cols/
│     ├─ AU_OK.parquet           an estimate — one Parquet file per column
│     └─ DOMAIN.parquet          a category column, strings
├─ drillholes/
│  ├─ collars.csv · survey.csv · assay.csv
│  └─ dh.element.json            the SET: locations, hole relation, desurvey op
├─ grids/  ·  meshes/  ·  clouds/
├─ bookmarks/  ·  scenes/        one small JSON per saved view
└─ recipes/                      analysis setups as commented YAML
```

Parquet — the model store

Store as Parquet… (right-click a CSV/ `.dm` model) converts it Morton-sorted, so each row-group's footer becomes a tight spatial index and queries skip whole row-groups they can't match. The dialog says exactly what will happen — per-layer rows, sizes, and a **keep original** option — and the same conversion runs **when you save a project** (a one-time, remembered choice: always / ask / off). Painting stays editable through the reorder, the file self-describes (grid, extents, categories in its metadata, so it reopens with no discovery pass), and the whole thing round-trips. Parquet block models also open **streamed**, holding nothing decoded, with predicate and spatial push-down on the filter.

The element manifest — the data model on disk

Beside each data file the project writes `<file>.element.json`: a readable description of the element — its geometry interpretation, its record locations (a drillhole set lists *collars*, *vertices*, and one location per interval table, related by the hole id; a grid lists its lattice with coordinates implicit), its columns, and an **ops list**: every derivation (a calculation's expression, an estimate's method and parameters, a broadcast's source column, a join's inputs) recorded as data. The

ops list is the audit trail — open the JSON and read exactly how every derived column came to be. It's also how the project restores: definitions reload from the manifest, and derived data that's missing or stale is simply re-derivable.

Column sidecars

Derived columns that carry data — estimates, materialized calculations, joins, broadcasts, painted domains — live in a `<file>.cols/` folder as **one Parquet file per column** (numeric as floats, categories as strings). Small, typed, self-describing, and readable by anything that reads Parquet.

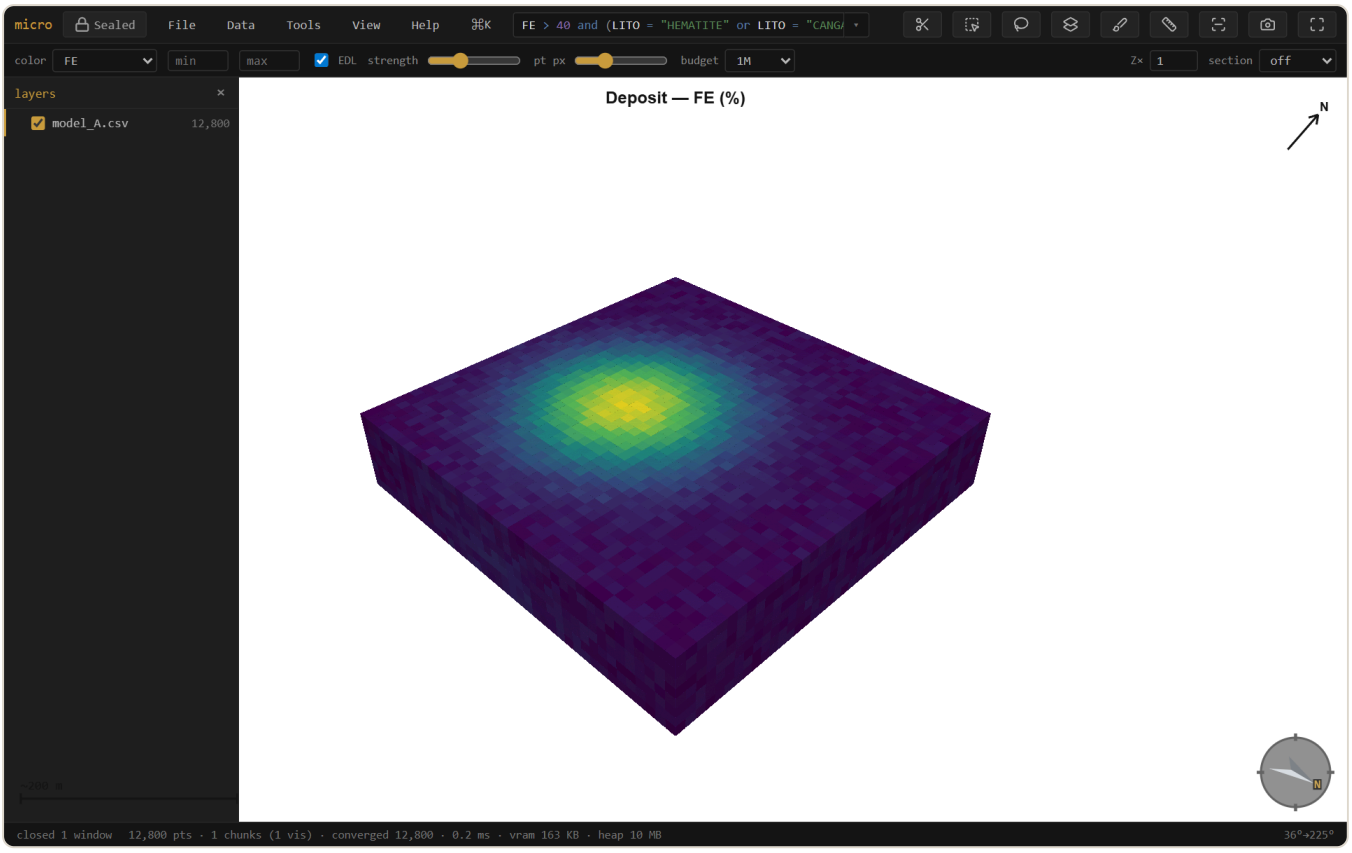
Sidecars & the memory budget

Files in a project also get a small `.meta.json` **sidecar** at save time: the discovery result (grid, extents, sub-blocking, categories — reopening skips the sweep entirely, even on a 13 GB `.dm`), column statistics, and a **band index** that lets filters skip stretches of CSV/`.dm` files that provably can't match. Sidecars are a cache: delete one and micro re-scans.

In memory, scanned columns **pin** into a budgeted cache (View → **Filter column cache**: off / auto / a size in MB — auto scales with your machine) so repeat filters and widget drags run with no file reads; derived-column values share the same budget and quietly reload from their sidecars when evicted. The memory panel (Help → memory) shows what's held.

17 Figures & decorations

View → Decorations & figure… toggles a **scale bar**, **north arrow**, color **legend**, and **title** onto the viewport — they show live and are captured verbatim by **Export figure** (at 1× / 2× / 3× scale for print). Legends are **per layer** — stack one per model when several share the view, each with its own title. Set a **background** (white or paper for print; the decorations flip to dark ink to stay readable). The exported PNG is what you see, ready for a report — and it pairs with the lineage report for a fully auditable figure. The analysis plots have their own **↓ png / ⧉ copy / ⧉ table** buttons.



A report-ready figure: white background, scale bar, north arrow, grade legend, and title — composited onto the export exactly as shown.

18 Projects & export

In a Chromium browser, **File** → **Open project folder** keeps your layers, styling, filters, selections, camera, section, bookmarks & scenes — and your open **analysis windows** (see Windows) — in a plain folder (`project.json` + the data files). `Ctrl+S` saves.

The folder is organized **by data kind** — `models/` `drillholes/` `clouds/` `meshes/` `grids/` `tables/` — with each file's manifest and sidecars beside it (see the project store). **Tidy project folder** migrates an older flat project in place, and **Scan folder for new data** treats the folder as an inbox: drop a file in by hand (or from another tool) and micro picks it up. A loose layer (opened from outside the folder) offers **Copy into project** from its context menu — always with a sized consent step, never silently.

The inbox handles *new* files; **Reload from disk** (a layer's context menu) handles *changed* ones. If a file was edited outside micro — a model re-run, a parameter table updated in another tool — reload re-reads the fresh bytes through the layer's existing setup: column mapping, styling, calc columns, filter, and surface state all ride through. Edit a routine's parameter CSV, reload it, run the routine again — the whole loop without reopening anything. Drag-dropped files can't reload (they're snapshots with no path), and if the file's size changed, micro notes that painted and materialized columns — which attach by row position — may be stale.

Export rows… writes any layer back out — scope **all** / **filter matches** / **selection**, every column including calculated, painted, estimated, and joined ones, as CSV or Parquet. Opening a `.dm` and exporting *is* the `.dm` →CSV (or →Parquet) converter.

Tables — data without geometry

Not everything in a project is spatial. Drop a CSV with **no X/Y/Z columns** — a cut-off table, a density lookup, a price deck, a routine's parameter table, a report you exported earlier — and micro opens it as a **table layer**: rows and columns, no geometry. It sits in the layers panel with a ▦ mark instead of an eye (there is nothing to show or hide), and it never enters the 3D budget.

Everything that is about *data* works on it: the **attribute table**, the **filter** (the same expression language — the matches drive the table view and the export), **ƒ calc columns**, **stats**, and **Export rows**. Everything that is about *space* is simply not offered — no grade–tonnage, no swath — because there are no coordinates to work with, not because a rule forbids it. Table files live in `tables/` in the project folder and reopen as tables.

A spatial layer can be **reinterpreted as a table** (drop the geometry) from its context menu, and a table whose file *does* carry coordinates can go the other way. Before this, a coordinate-less CSV was simply an error.

Excel workbooks

Open an **.xlsx** and micro reads it with no import step: pick which sheets you want (or take the whole workbook, which arrives as a group named after the file) and **each sheet becomes a table layer**. Column types come from Excel itself — a number is a number, a date is a date — so nothing is re-guessed from text.

A workbook is not a coordinate system. A sheet whose columns happen to be called X, Y and Z might be a block model, or it might be a table about something else entirely, so micro never assumes: every sheet opens as a table, and promoting one to geometry is your explicit act — **Reinterpret as → block model / points**, offered only when the columns actually support it. The workbook stays the layer's source file, so the project remembers which sheet a layer came from and re-reads it on load.

This makes the spreadsheet a first-class citizen of a project: keep the quarter's run matrix, cut-off tables, or density lookups where they were already maintained. A routine can read its parameter table straight from one — `from: "params.xlsx#Bench targets"` — so you can edit the workbook in Excel, **Reload from disk**, and run the routine again without leaving micro.

Bookmarks & scenes — save where you were looking

Two ways to save a view, under **View → Bookmarks** and **View → Scenes**. A **bookmark** is a *viewpoint* — camera + section — the quick nav aid: drop a few (**Save bookmark…**), or press `g` then `Shift`+`1`–`9` to save the current view into a numbered **slot** — and `g` then the digit jumps back to it (a bare digit deliberately does nothing, so a stray keystroke can't teleport the camera). A **scene** is the full *working state* — camera + section *plus* each layer's visibility, color-by, and filter, so applying one puts the whole scene back the way it was (**Save scene…**, or **Save scene (with selection)…** to also pin the 3D selection). Applying a scene names any layer it wanted that isn't loaded, and skips it. Both live as small JSON files in the project folder — `bookmarks/` and `scenes/`, one per entry — so they travel with the project, are hand-editable, and get picked up by **Scan folder for new data**. There is no fly-through: applying a view snaps straight there.

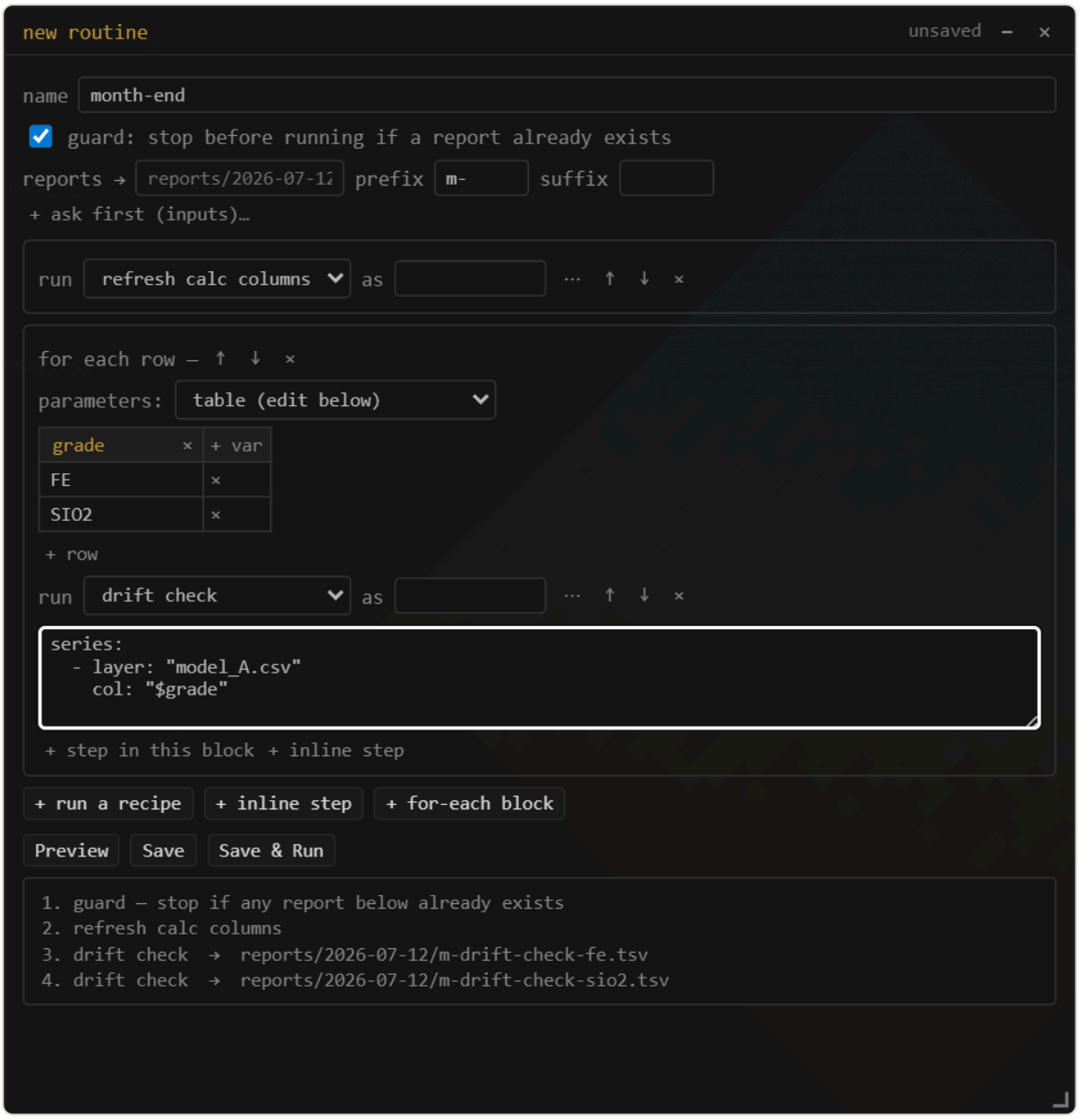
Recipes — save an analysis to run again

save as recipe… (next to Run in the grade–tonnage, swath, and stats windows, in the Calculations window, in the **select by solid / surface** dialog, and in the decorations panel) writes the current setup as a named YAML file in `recipes/` — every series, unit, weight expression, band, scope, and range; for calculations, the whole column set. A **figure recipe** captures the view + **section** + decorations + background + export scale; running it re-renders, exports the PNG, and restores everything it touched — the monthly plan-view figure regenerates itself. **Tools** → **Recipes** lists them; picking one opens the tool configured **and runs it**. Recipes are plain, commented, hand-editable files — copy one between projects, keep an office standard set, drop one into the folder and **Scan folder** picks it up. Series reference layers **by name**; if the named layer isn't loaded, one question asks which loaded model to use instead — that's the whole parameter system. A recipe is data, never code: the strict YAML subset can't carry a script.

```
# micro recipe — grade-tonnage. Hand-edit freely; series reference layers by NAME
name: "Monthly GT"
tool: "gt"
params:
  nCut: 15
  gradeUnit: "pct"
  series:
    - layer: "model.parquet"
      col: "FE"
      weight: "SG"
```

Routines — recipes that run recipes

A **routine** chains saved recipes into one runnable document: refresh the calc columns, then a drift check for each grade of interest, every table filed under `reports/`. It is the month-end close as a file — and you don't have to write it. **Tools** → **Recipes** → **New routine…** opens the routine editor; what it saves is a small YAML file you can read back in one glance.



The routine editor. The **for each row** table is the heart of it — variables are columns, runs are rows. **Preview** lists every step and every report path before anything runs.

Steps come in two shapes. A plain step runs one recipe, optionally layering **overrides** on top (a different cutoff count, a series pointed at another column — the ⋯ button on a step). A **for-each block** runs its steps once per **row of a parameter table**: add a variable for each thing that changes, a row for each run, and write `"$grade"` in a step wherever the value should land. **+ row** duplicates the last run so you only edit what differs. There is no scripting here — expansion is plain substitution, so the whole run is knowable before anything executes. That is exactly what **Preview** shows, and what the optional **guard** checks: stop before step one if any report it would write already exists.

The preview is not just a listing — each line is **clickable**. Every for-each row expands to its own line, so clicking one opens that step's tool window *configured* with exactly that row's values baked in, and nothing runs: tune a drift check against one grade, save it back under the same

recipe name, and the whole routine picks up the change. (Calc-column and figure steps act immediately, so those lines don't open.)

A step doesn't have to point at a saved recipe at all. Give it `tool:` instead (**+ inline step** in the editor: `gt`, `swath`, `stats`, `figure`, `calccols`, or `solid`) and its whole setup lives right in the step — so a batch can be written from scratch, with nothing recorded first. Name a recipe when you'll reuse the setup elsewhere; inline it when it belongs to this routine alone.

The editor writes a file like this (block style, strings quoted — hand-edit or write them from scratch if you prefer):

```
# micro recipe — routine. Hand-edit freely; series reference layers by NAME.
name: "month-end"
tool: "routine"
params:
  output:
    prefix: "m-"
  steps:
    - guard: "no-overwrites"
    - recipe: "refresh calc columns"
    - each:
        - grade: "FE"
        - grade: "SIO2"
      steps:
        - recipe: "drift check"
          series:
            - layer: "model_A.csv"
              col: "$grade"
```

Which rows run — `filter`

A parameter table usually outlives any one run: benches that are finished, domains you're not reporting this month, a scratch row someone left behind. Rather than maintaining a second copy of the table, give the `each` block a **filter** — the same expression language again, deciding which rows fan out:

```
- each: { from: "params.xlsx#Benches" }
  as: b
  filter: "active = true and bench >= g.floor"   # which rows run
  steps: [ ... ]
```

The row's own columns are plain names, and any scalars from `params` are visible too, so a row can be compared against a global. This is *data selection*, not an `if`: the rows that run are still

decided entirely by the files you can read, never by the result of a step — so **Preview** still lists every run and the **guard** still checks every output, exactly as before. A mistyped column in the filter says so out loud instead of quietly matching nothing.

Computed values — `${…}`

A `$name` substitutes a value as it stands. Wrap it in `${…}` and you get an **expression** — the *same* language as the filter box and the *ƒ* columns, so there is nothing new to learn:

```
- recipe: "grade tonnage"
  nCut: ${g.cutoff * 1.1}                    # arithmetic
  floor: ${if(b.grade = "FE", 55, 45)}       # if(), the same one you use in ƒ
  as: "bench ${b.bench} · ${b.grade}"        # inside text, it interpolates
```

Expressions compute **values**, never **structure**. They can say what a cutoff *is*; they cannot decide how many runs there are, which recipes exist, or what a report is called. That is deliberate, and it is what keeps a routine honest: the step list is still fully enumerable before anything executes, so **Preview** still lists every step and the **guard** still checks every output. A routine remains a document you can read, not a program you have to run to understand.

A mistyped name fails loudly rather than quietly becoming empty — `${b.bnch}` reports *unknown column: b.bnch — did you mean b.bench?* — and dotted names are just names, so `$g.cutoff` and a real assay column called `Fe.pct` are spelled the same way.

Nesting — every bench × every grade

An **each** block is just a step, and its body is a list of steps — so an `each` inside an `each` simply **nests**, and nesting is a **cross product**: every row of the outer table against every row of the inner one. Give each table an `as` and its columns get a namespace, which is what keeps two tables apart when they both have a column called *cutoff*.

```
steps:
  - each: { from: "params.xlsx#Benches" }
    as: b                                  # → $b.bench
    steps:
      - each: { from: "params.xlsx#Grades" }
        as: g                              # → $g.grade
        steps:
          - recipe: "grade tonnage"        # runs once per (bench × grade)
            series: [{ layer: "model.parquet", col: $g.grade }]
            bench: $b.bench
```

Two benches and two grades give **four** runs, and each report's name carries both values (`gt-1040-fe`). Contrast that with a *single* `each` whose columns are two equal-length lists: that is a **zip** — the columns are paired, giving **two** runs, not four. Both are useful and they are easy to confuse, so **Preview** lists the expansion either way: count the lines and you know which one you wrote.

Parameters that don't fan out

A real parameter workbook holds two kinds of sheet: **run tables** (one row per run — that's `each`) and a **key→value sheet**, usually called *General*, with a `parameter` column and a `value` column. That second kind is a *series*, not a fan-out: you want its numbers available to every step, not a run per row. That is what **params** is.

```
name: "Month-end"
tool: "routine"
params:
  params:
    - from: "params.xlsx#General"    # parameter | value  →  $g.cutoff, $g.density
      as: g                          # namespace (default: the sheet name; "" merg
      index: parameter               # key column   (default: the first)
      value: value                   # value column (default: the second, when the
      fill: 0                        # what an empty cell means
  steps:
    - recipe: "drift check"
      nCut: $g.cutoff                # a scalar, everywhere
    - each: { from: "params.xlsx#Benches" }
      steps:
        - recipe: "grade tonnage"
          series: [{ layer: "model.parquet", col: $grade }]
          floor: $g.cutoff           # scalars reach inside each blocks too
```

`params` is also a **step**, not only a header: written in the step list it binds its names *forward*, for the steps that follow — and written *inside* an each block it is scoped to that block. Same for the report names: `as` on a `recipe` or `tool` step names its report, while `as` on an `each` or `params` step names a scope.

If the sheet has **more than two columns** and you don't name a `value` , the whole row stays reachable: a *Domains* sheet keyed by domain gives you `$d.hem.cutoff` and `$d.hem.price` . Keys are made safe for use as names — *dilution factor* becomes `$g.dilution_factor` . A variable from an `each` row **shadows** a scalar of the same name, so a run table can override a global. And **Preview** lists what the params resolved to, so the scope is never invisible.

One Excel quirk worth knowing: a value column with a single blank cell is typed as *text* by Excel, so micro coerces numeric-looking parameters back to numbers when it reads a routine's tables — `55` stays a number even if the sheet has a gap.

The parameter table has two more spellings for hand-writers: **equal-length columns** (`each:` holding one list per variable, zipped row by row — never a cross-product; unequal lengths are an error, not a guess) and `from: "params.csv"`, which reads the rows from a table file in the project — keep the quarter's run matrix as a CSV and the routine stays two lines.

Keeping a result — ⇥ layer

Every analysis window has a ⇥ **layer** button beside its export buttons. It keeps the result — the grade–tonnage curve, the swath profile, the stats table — as a **table layer**: something you can filter, join against another run, and export like any other data, instead of a window you look at and close. In a routine, a step with `emit: layer` does the same automatically, so a month-end run ends with layers you can read, not only files you have to open elsewhere.

Filing layers — a name can be a path

A layer name with slashes in it is a **path**: name a result `swaths/FE drift` and it lands in a **swaths** group (created if it isn't there yet), labelled *FE drift*. It nests — `month-end/gt/FE` makes a *gt* group inside a *month-end* group — and it works on any layer, not just results: **rename** a model to `models/deposit` and it files itself. A plain name with no slash just renames, as always.

This is sugar at naming time, not a new kind of group: once filed, the layer tree is the truth again — drag layers between groups, collapse them, reorder them exactly as before. It means a routine's outputs organise themselves: a step with `emit: layer` files its result under the routine's own name, and `emit: { layer: "swaths/$b.grade" }` puts it exactly where you say.

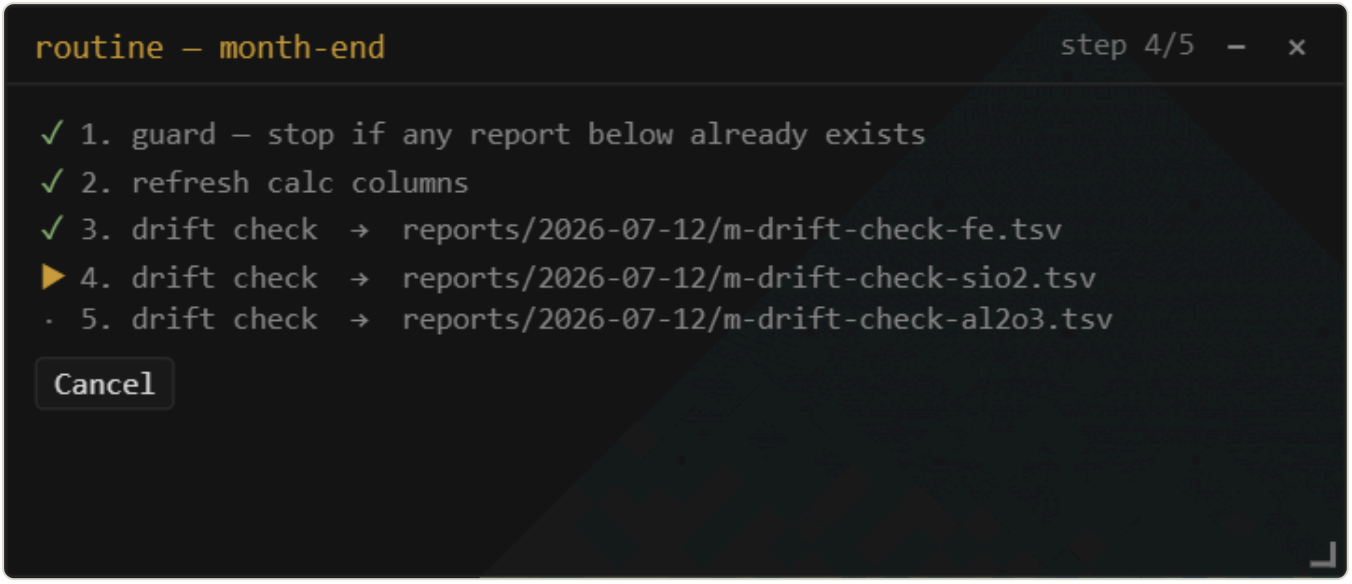
What a step writes, and what stops it

Preview now says what every step *writes* — a report path, a **column** on a layer (marked *(replaces)* if it is already there), a figure, or *(opens only)* if it writes nothing. The **guard** checks all of it: a calc-column step that would overwrite an existing column stops the routine before anything runs, exactly as a report file does. Producing operations replace by name rather than refusing, so running a routine twice leaves the same project as running it once.

When a run stops

A long routine that fails at step nine should not cost you the first eight. A stopped run offers **Retry step** and **Skip & continue**; a cancelled one offers **Resume**; and right-clicking any line in **Preview** runs the routine *from* that step. Steps that were not run are marked with a dash in the tracker, and the summary says *"1 step run (2 skipped)"* rather than claiming credit for the whole thing.

Analysis steps (grade–tonnage, swath, stats) write their tables to `reports/<date>/<step>.tsv` — dated folders diff month over month — and `output:` takes `dir`, `prefix`, and `suffix` to steer the naming. Calc-column and figure steps run their engines directly. Routines never prompt mid-run: every referenced recipe and layer is checked up front and missing ones are named before anything starts; a failed step stops the routine visibly, its window left open for inspection.



The run tracker — the same expanded list, live. ✓ done, ▶ running, ✗ failed with its message in place; **Cancel** stops cleanly between steps.

A running routine opens the **run tracker**: the expanded step list with a live state per line and *step n of N* in the title. **Cancel** stops after the current step finishes — reports already written stay written, nothing is left half-done. The window stays open when the run ends (or stops, or is cancelled) as the run's record: what ran, what it wrote, where it stopped.

Ask-me-first recipes — `inputs:`

A recipe with `inputs:` asks its questions first. The column picker follows whichever layer is chosen above it.

Any recipe or routine can declare `inputs:` — a short list of questions. Running it then opens a small generated form instead of executing immediately: pick the model, pick the column, accept or change the defaults, press **Run**. The answers flow into the recipe through the same `$name` markers the for-each table uses, so one saved setup serves every model and grade in the project.

```
name: "GT — any model"
tool: "gt"
inputs:
  - name: "model"
    label: "Block model"
    type: "layer"
  - name: "grade"
    label: "Grade column"
    type: "column"
    of: "model"
  - name: "ncut"
    label: "How many cutoffs"
    type: "number"
    default: 15
params:
  nCut: "$ncut"
  series:
    - layer: "$model"
      col: "$grade"
```

Five input types: `text` (the default), `number`, `choice` (with `options:`), `layer` (a picker over the loaded layers), and `column` (a picker over the columns of the layer named by `of:` — it re-fills live when that layer changes). A marker that is exactly `"$name"` lands **typed** — a number input arrives as a number — while markers embedded in longer text become text. Answers are remembered for the session, so the next **Run…** starts from where you left it.

Routines ask too: declare `inputs:` on the routine itself — the editor's **+ ask first** section adds them without touching YAML — and the answers substitute everywhere in the steps *before* the parameter tables expand. Keep input names distinct from the tables' variable names; the questions come first, the table fans out after.

Mesh export

Export mesh… (right-click a mesh layer; multi-selections and groups get **Export N meshes…**) writes **OBJ**, **PLY**, **MSH** (Leapfrog/ARANZ), or **LFM**. OBJ and LFM are multi-mesh — one named object per layer, with each layer's tint carried as the LFM color — while PLY and MSH merge multiple layers into one mesh. Exports re-read the source geometry, so coordinates leave in **full double precision** (the PLY writer uses `double` — at UTM northings, float32 would cost half a meter). A Datamine wireframe pair in, an LFM out: micro is also a mesh-format converter.

19 Windows

The analysis, table, calculations, and paint windows are all draggable and pinned to their layer (open several at once). **View** → **Windows…** lists every open window — click to focus, or **Cascade** and **Close all** to tidy up. The window on top carries an amber accent in its title bar; `Ctrl+`` cycles through the open windows.

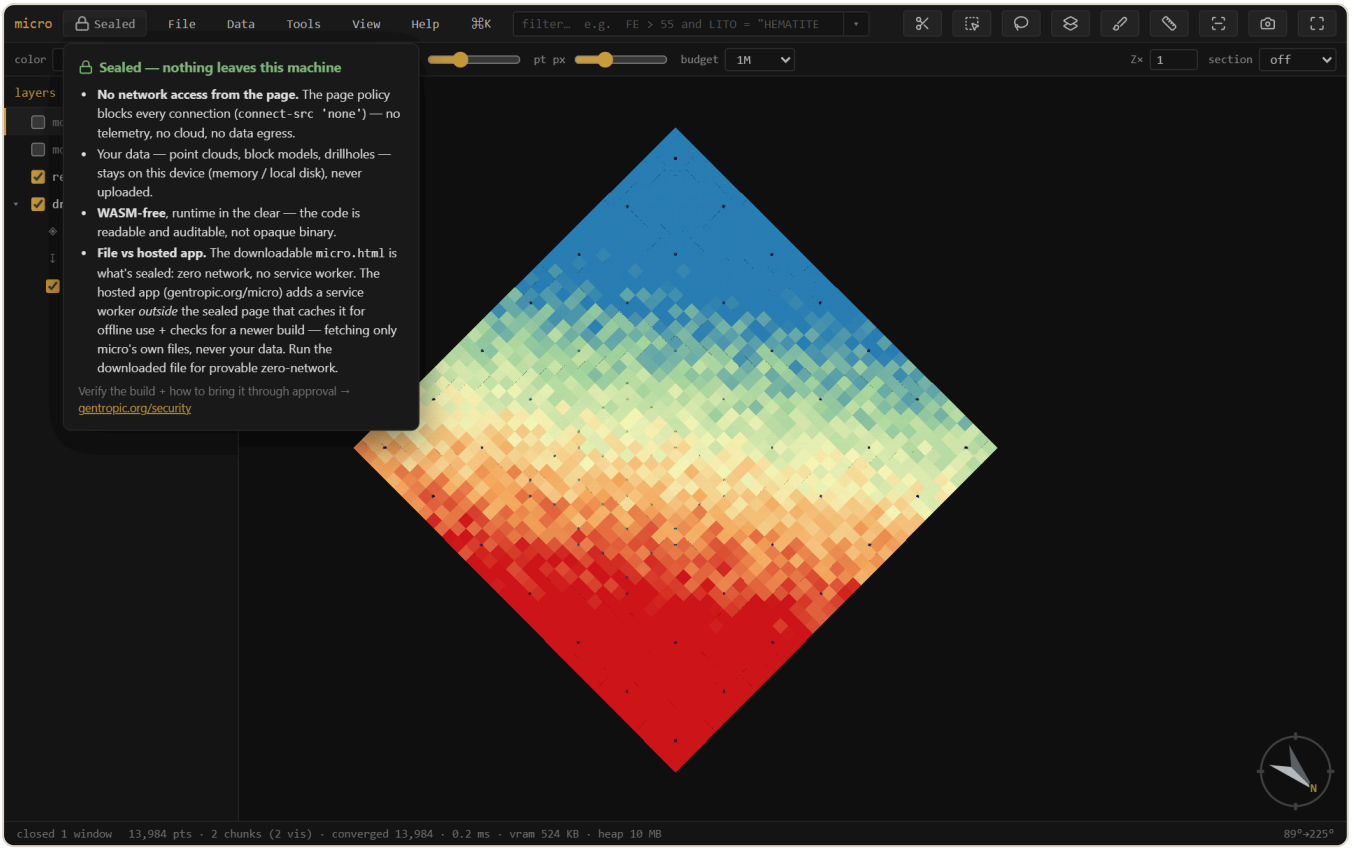
Minimize (the **–** button) sends a window to a slim strip of chips above the status bar — click its chip to bring it back exactly where it was. **Drag a window to the left or right screen edge** to snap it to that half of the screen (a preview shows before you release). Every window kind remembers its size — and its position, when it's the only one of its kind — so your grade–tonnage window opens where you like it, every time.

Every analysis window shares one **results row** in its footer, under Run: ⭳ **png** and ⧉ **copy** for the plot, ⇥ **layer** to keep the result as a table layer, ⧉ **table** to copy the numbers as TSV — the same actions in the same place, in every window.

Grade–tonnage and swath setups **persist in the project**: every series, unit, weight expression, band width, scope, and manual range is snapshotted on each Run and on close, and a window left open when you save is **pinned** — reopening the project brings it back in place, configured, reading *restored — Run to compute*. Run stays manual, so a huge project never surprise-scans on open.

20 Security: Sealed

micro is **Sealed**: it makes **no network requests** — a browser security policy (`connect-src 'none'`) blocks every connection, so there is no telemetry, no cloud, and no auto-update. Your data — point clouds, block models, drillholes — stays on your device. The build is **WASM-free** with the runtime in the clear, so the code is readable and auditable rather than opaque binary.



The **Sealed** badge (top-left) explains the posture and links to the verifier. For confidential resource data, this is the point: nothing leaves the machine.

Verify the build and read how to bring it through approval at gentropic.org/security.

21 Install & offline

micro runs at gentropic.org/micro — install it as an app from the browser, or download `micro.html`: one file, no installer, that you can keep, e-mail, drop on a shared drive, or run fully offline by double-clicking. There is nothing to set up and nothing that phones home. Works in current Chrome / Edge / Firefox; project folders and mounts need a Chromium browser.

22 Questions & troubleshooting

Which browsers work? Current Chrome, Edge, and Firefox all run micro. Project folders, disk mounts, and Save-as need a **Chromium** browser (Chrome/Edge) — Firefox opens and analyzes everything but can't hold a folder connection.

How big is too big? There isn't a hard limit — micro streams, so opening is near-instant at any size and memory stays bounded. The practical costs are per-*scan*: the first filter or analysis over a huge CSV reads the file once (a 13 GB Datamine file is a few tens of seconds), after which sidecars, push-down, and the pin cache make repeats fast. For models you'll work with daily, **Store as Parquet…** is the single biggest speedup.

My filter is slow on a CSV / .dm. Run **Build index** (right-click the layer) or just save the project — the band index lets filters skip file regions that can't match. Or convert to Parquet, which also gets spatial push-down.

micro looks memory-hungry / I want it leaner. View → **Filter column cache** sets the budget (off / auto / a size). Cached columns and derived values evict under the budget and reload from disk when needed; **off** disables caching entirely. Help → memory shows what's held.

A numeric column isn't offered for color / grade–tonnage. The type sniff read it as text (a stray unit string or thousands separator will do it). Right-click the column in **Properties** → **columns** → **Number**.

LAZ files? Not yet — micro is deliberately free of compiled decoders (see Sealed), and LAZ needs one. Convert to LAS, or export XYZ.

Rotated block models? They open as centroids (points) today; box rendering assumes an axis-aligned lattice. On the roadmap.

Sub-blocks come up as points. Detection needs per-block dimension columns (`dX/dY/dZ` or `XINC/YINC/ZINC`) and power-of-two refinement. Odd ratios fall back to centroids — everything still works, blocks just draw as points.

Can I run it fully offline / air-gapped? Yes — that's the design. Download `micro.html`, copy it anywhere, double-click. Nothing phones home either way; the offline copy just makes it visible.

Something looks wrong / a file won't open. `i` (file info) shows what micro detected — column mapping, grid inference, parse warnings. Most import surprises are visible there, and the mapping can be corrected in place.

A Reference — keys & the expression language

Keyboard

Key	Action	Key	Action
`⌘/Ctrl K`	command palette	`/`	focus the filter
`f`	fit the data	`p`	layers panel
`n s e w`	look level that way	`d` / `u`	plan / from below
`o`	ortho / perspective	`x`	section on/off
`k` / `c`	knife / cut — drag a section (⇧ snap 15°, ⇧⌥ 5°)	`l` / `⇧L`	face the section / its back
`,` `.`	step one section slab	`Ctrl wheel`	scrub the section
`Alt wheel`	slab thickness (thicker / thinner)	`1`–`9`	jump to bookmark
`m`	measure	`i`	file info
`r` / `v`	select rectangle / lasso	`b`	paint a column
`g` `1`–`9`	go to view slot (Shift+digit saves it)	`F1`	keyboard & mouse help
`t`	attribute table	`[` `]`	step the active layer
`h` / `⇧H`	hide–show / solo the active layer	`F2`	rename the active layer
`⇧B`	block edges	`⇧P`	properties
`Ctrl S`	save project	`Ctrl Z`	undo paint
`Esc`	clear selection / cancel		

The expression language

One language everywhere — the filter, calculated columns, density and weight expressions, per-series filters, the hole filter. It is **total**: an expression can compare and compute but never loop, call out, or crash a scan — a blank input gives a blank result.

Comparisons	`> >= < <= = !=` · `between a and b` · `in (a, b, c)` · `contains "txt"` · `like "A%_1"` (SQL wildcards) · `matches "regex"` · `is blank` / `is filled`
Boolean	`and` · `or` · `not` — with parentheses for grouping
Arithmetic	`+ - * /` · `^` (power, Python-style precedence)
Numeric fns	`round int abs floor ceil mod log log10 exp sqrt pow min max clamp bin`
Blank handling	`coalesce(a, b, …)` (first filled) · `nullif(x, v)` (sentinel → blank: `nullif(AU, -99)`) · `ifnum isnum isblank isfilled` · the `blank` literal
Text fns	`upper lower trim len left right substr replace concat`
Conditional	`if(cond, then, else)` — nests into case ladders
Names	plain column names as-is (case-insensitive); anything with spaces or symbols in backticks: `` `Assay Au ppm` `` (the pandas convention)
Text values	in double quotes: `LITO = "HEMATITE"`
Comments	`# to end of line` — expressions may span multiple lines

`not` is a true complement: `not (AU > 5)` keeps blanks (they fail `AU > 5`), so it is *not* the same as `AU <= 5` — a deliberate, documented choice that makes filter + inverted-filter always cover the whole table. Autocomplete offers columns, functions, and category values — `Tab` / `Enter` accepts.

Expression cookbook

Task	Expression
Sentinel values (−99) to blank	`nullif(AU, -99)`
Cap grades for a resource case	`clamp(AU, 0, 30)`
Merge two estimates, first filled wins	`coalesce(AU_OK, AU_ID2)`
Grade classes as a numeric code	`if(FE > 58, 2, if(FE > 45, 1, 0))`
Density by lithology	`if(LITO = "HEM", 3.9, if(LITO = "ITA", 3.4, 2.8))`
Iron equivalence	`FE + 0.4 * MN`
Tonnes per block (10×10×5 m)	`SG * 500`
Contained metal (%, per block)	`FE / 100 * SG * 500`
Normalize messy hole ids	`upper(trim(BHID))`
Everything except two domains	`not (DOMAIN in ("WASTE", "AIR"))`
Filter to assayed rows only	`AU is filled and AU > 0`
Log-scale color for a skewed grade	`log10(clamp(AU, 0.01, 100))`
Flag suspicious duplicates of a cap value	`FE = 68 # exactly at the clamp`
Bench index from elevation (10 m benches)	`floor(ZC / 10)`

Each of these works anywhere an expression is accepted: the filter box, a f column, a density or weight slot, a per-series filter, the hole filter (over collar columns).

B Formats

Format	Read	Write	Notes
CSV / TXT / DAT	✓	✓	delimiter sniffed (, ; tab); block models need coordinate columns; per-block `dX/dY/dZ` → sub-blocked
Parquet	✓	✓	streamed with predicate + spatial push-down; micro's own store (Morton-sorted, self-describing metadata)
Datamine `.dm`	✓	—	block models + wireframe pairs (`…pt.dm` / `…tr.dm`); per-record `XINC/YINC/ZINC` → sub-blocked; export via CSV/Parquet
LAS	✓	—	point clouds; LAZ is not read (needs a compiled decoder — see §20)
PLY	✓	✓	points or mesh; writes `double` coordinates
XYZ / PTS	✓	—	plain point dumps
OBJ	✓	✓	meshes; multi-object on export
MSH (Leapfrog/ARANZ)	✓	✓	single mesh
LFM	✓	✓	multi-mesh with colors
GeoTIFF	✓	—	grids incl. multi-band + overviews (COG-friendly); no compiled codecs — LZW/deflate/PackBits
Surfer `.grd`	✓	—	ASCII + binary
ESRI ASCII `.asc`	✓	✓	the grid export

Column conventions the sniffers look for — coordinates: `XC/YC/ZC` , `X/Y/Z` , `EAST/NORTH/ELEV` -style names; sub-block sizes: `dX/dY/dZ` , `XINC/YINC/ZINC` , `DIMX/DIMY/DIMZ` ; drillholes: a hole id (`BHID/HOLEID` -style), `FROM/TO` on intervals, `DEPTH/AT · AZ/BRG · DIP` on surveys. Everything the sniff decides is shown in **file info** (`i`) and correctable there.

C Glossary

element	one dataset as micro models it: geometry interpretation + record locations + columns + recorded operations
location	a record space within an element — a block model's cells; a drillhole set's collars, trajectory vertices, and intervals
ƒ (calculated) column	a column defined by an expression, computed on the fly; nothing stored
materialized column	a derived column whose values are stored (a Parquet file in `.cols/`) — estimates, frozen ƒ, joins, broadcasts
painted column	a category column filled by hand (brush) or by a solid/surface classification
broadcast	landing a collar column on drillhole intervals by the hole id
op	a recorded, re-runnable derivation — the expression / method / inputs, stored as data in the element manifest
element manifest	`<file>.element.json` — the readable description of an element and the audit trail of its ops
sidecar	a small file micro keeps beside yours: `.meta.json` caches, `.cols/` columns, display grids
pinned column	a column held in memory (under the cache budget) after a scan, so repeat filters skip the file
section / slab	the live cut: a plane with thickness; blocks clip against it analytically
trace	a sectioned mesh's intersection outline, drawn flattened on the cut face
isolate	what the filter does: matching records stay, everything else leaves render, table, and analysis
declustering	down-weighting spatially clustered samples so a mean isn't dragged by infill drilling
lineage	the derivation tree of a layer — what it was made from, with parameters and timestamps
Sealed	the security posture: no network requests possible, no compiled code, runtime readable

recipe	an analysis setup saved as commented YAML — re-runnable, hand-editable, data not code

D Index

micro — part of the Auditable project, Geoscientific Chaos Union. Engine: `@gcu/condenser`; expressions: `@gcu/expr`; declustering: `@gcu/gsjs`. MIT.
Screenshots are generated from the app by an automated harness, so they track the current build. Docs for micro — see the in-app **Help** menu for the terse quick reference.