Skill Display Construction - Canvas

Purpose

Canvas displays are the paradigm for **pixel art that happens to show live data**. Process diagrams, P&IDs, equipment layouts, architectural overviews. Every element has an explicit `Left / Top / Width / Height` and you compose by placement, layering, and grouping (The Canvas displays is resolution independent vector graphics, pixel alt is just the analogy to explain).

Use Canvas when:

- The display shows equipment in a **spatial layout** (pipe A connects to pump B connects to tank C)
- The visual language is **P&ID / process flow** (operators scan it as a schematic)
- Custom shapes and line work matter
- You need the full set of dynamics (RotateDynamic, ScaleDynamic, MoveDragDynamic, SkewDynamic, BargraphDynamic are Canvas-only)

Use Dashboard (load **Skill Display Construction — Dashboard** instead) when the display is primarily data monitoring, grid-based cards, or there's no spatial relationship to preserve.

Prerequisite: load **Skill Display Construction — Basics** first. It covers theme-first thinking, write mechanics, the build loop, binding syntax, and spacing /typography tokens.

Section 1 — Canvas mental model

Choose Canvas or Dashboard by the question being asked

The choice isn't about aesthetics — it's about the question the operator is answering when they open the page.

Operator question	Canvas	Dashboard
"Where in the plant is this happening?"	x	
"Is flow going through path A or path B?"	x	
"What's the state of equipment X right now?"	x (if on the flow)	x (if ungrouped KPI)
"How is KPI X trending?"	x (possible)	x
"Which area has the most alarms?"	x (possible)	x
"Is the plant running within normal bounds today?"	x (possible)	x
"I need to click *on* a valve / pump / tank."	x (better here)	x (possible)
"I need a one-glance shift summary."	x	x

Rule of thumb: if removing the equipment layout and putting the same data in a grid of tiles would lose meaning, it's a Canvas. If the layout is indifferent to plant geometry, it's a Dashboard. Most plants need both — a Canvas per process area, and a Dashboard for the plant as a whole.

Think in zones, not elements

The worst Canvas displays are built element-first — you place a Rectangle, then another, then wonder why nothing aligns. The good ones are built zone-first.

Exception — a supplied drawing already defines the layout. When the task is to reproduce a customer's PFS, P&ID, or existing screen, do not re-zone it: the source geometry is the spec. See Section 1.2.

1. **Divide the canvas into zones.** Rectangular regions for each process section (Intake, Treatment, Distribution) or each info panel (Equipment Detail, Live Metrics, Identity).
2. **Lay zone background Rectangles first.** These are the visual scaffolding — operators read the display by scanning zones, not individual shapes.
3. **Place elements WITHIN zone coordinates.** Everything in a zone has its Left/Top relative to the zone's origin.
4. **Connect zones with flow indicators.** Arrows, pipe lines, direction markers.

Canvas sizes

Canvas	Width × Height	When
Standard HD	1366 × 728	Default, works everywhere
Wide	1600 × 900	Modern control-room monitors
Full HD	1920 × 1080	Dedicated operator displays

4K scaled	3840 × 2160	Rare — prefer 1920×1080 with StretchFill

Zone math (any canvas size, N zones)

```
margin = 20 gap = 15 titleBar = 60 // top strip for display title + status bottomPanel = 140 // for trend/alarm /summary zoneHeight = Height - titleBar - bottomPanel - margin zoneWidth = (Width - 2*margin - (N-1)*gap) / N zone [i].Left = margin + i * (zoneWidth + gap) zone[i].Top = titleBar zone[i].Width = zoneWidth zone[i].Height = zoneHeight
```

For a 1600×900 canvas with 3 zones: each zone is ~515 wide × 640 tall. For 4 zones: ~381 wide. For 2 zones: ~780 wide.

Z-order by Elements array order

Canvas has no z-index property. **The order of the Elements array IS the z-order** — earlier elements render behind, later elements in front. Always place:

1. Full-canvas background Rectangle first (if overriding root Background)
2. Zone background Rectangles next
3. Zone-internal shapes and symbols
4. Labels and text (on top of their containing zones)
5. Interactive overlays (click-zones, hover highlights) last

Section 1.2 — Working from a customer drawing (the fidelity contract)

When the user supplies a drawing of the screen — a PFS or P&ID, an SVG mockup from an integrator, a screenshot of the legacy HMI being replaced — settle one question **before authoring anything**: is that drawing the **specification**, or the **source material**?

The same input produces two completely different displays depending on the answer, and choosing wrong is the most common way a display-from-a-drawing task fails.

	Replica	**Derived**
The drawing is…	the spec. Its geometry *is* the requirement.	the source of truth for the *process*, not for the layout.
Acceptance test	Positional identity — "the FC is upper-left with stream 101 directly below it, same as the drawing."	Readability — can the operator answer their question quickly?
How elements are placed	**Extracted** from the source. Measured, not chosen.	**Composed** zone-first, per Section 1.
Pick it when	The customer hands you a drawing of their plant.	The request explicitly asks to modernise, simplify, or re-lay-out.

Default to Replica

When the user supplies a drawing, build a Replica and say so in one line. Ask which mode they want only when the request explicitly signals adaptation ("modernise this", "make it operator-friendly", "clean it up").

Default this way because the two failure modes are not symmetric:

- A faithful replica is never *rejected*. At worst the customer asks to improve it from there — and they still recognise their own plant on the screen.
- A re-composition gets "that is not my diagram." The customer concludes the tool cannot do the job, and the evaluation ends instead of iterating.

Default to the recoverable failure, not the prettier one.

Building a Replica — measure, do not infer

"Reproduce this drawing" is not achievable by eye. Placing elements where they *look* right produces a plausible, well-composed, wrong drawing — and because it looks fine, no review step catches it. The build loop will happily converge on a polished wrong answer, because every individual fix succeeds.

1. **Establish a coordinate frame in the source.** Render the PDF or image at a known scale and overlay a labelled grid, so positions can be read off numerically instead of estimated.
2. **Read positions into a table.** One row per object — equipment, instrument bubble, valve, stream label, off-page connector — recorded in the source document's own coordinate space.
3. **Map with a single transform.** `X = (x - x0) * s + margin`. One scale factor for the whole drawing preserves every relative position and every proportion at once.
4. **Emit the display from the table.** Generate the Elements array from the coordinate table rather than hand-placing elements one at a time.
5. **Keep the table.** It is the durable asset: the same table can drive a preview mockup and the FrameworX Canvas without the two drifting apart, and a later re-scale or re-target becomes a one-line change.

Sanity check before writing: pick three objects that sit far apart on the source and confirm their relative order and spacing survive the transform.

Replica does not mean plain

Fidelity is about **position**, not about refusing to add value. A Replica still gets live bindings, colour by pipe service, a level fill in the vessel, and readouts beside the instrument they belong to — everything else in this skill still applies. What it does not get is **re-arrangement**: do not move equipment to balance the composition, do not substitute your own symbol vocabulary for the drawing's, and do not stub out a pipe run because routing it fully is inconvenient. If the drawing routes a line across the full sheet, route it.

When you must depart from the source — occasionally necessary for legibility at screen resolution — list every departure explicitly when handing the display back, so the customer can accept or reject each one. Silent departures are what turn a good display into "that is not my diagram."

Section 1.5 — Readouts are inline, not hero

The most common Canvas mistake is treating live values as Dashboard hero numbers. They're not. A readout on a canvas belongs **next to the piece of equipment it describes**, sized for scanning.

The inline readout pattern is three TextBlocks per row — label, value, unit — placed side-by-side (or a single TextBlock with a composite `LinkedValue` that concatenates them):

- **Label** — FontSize ~11, dim/secondary theme brush
- **Value** — FontSize 16–22, primary foreground brush, monospace for numeric readouts
- **Unit** — FontSize ~10, muted theme brush, monospace

Do not wrap individual equipment-attached values in `CenterValue`. CenterValue belongs on Dashboard tiles where the operator is answering "how is KPI X doing", not on Canvas where they're answering "how is *this specific piece of equipment* doing." When the value is attached to a visible vessel, pump, or pipe on the canvas, it's an inline readout.

Two Canvas-native exceptions to the inline-readout rule:

- **RadialGauge** when the absolute position on a dial is more meaningful than the digits — valve position, pressure relief, anything where headroom matters.
- **BargraphDynamic** on a Rectangle for level indication inside a tank body.

Otherwise: inline readout.

Section 1.6 — Responsive authoring (OnResize and the six locks)

A Canvas is authored at a fixed design size but renders into whatever region the Layout gives it — full screen, a docked band, a mobile viewport. **How a Canvas reflows is a display-level decision plus, optionally, per-element locks.** Pick the resize behavior deliberately; the wrong default either distorts your process geometry or pins it so hard it leaves dead space. The deep reference (RESS, mobile alternate layouts, the full property tour) lives in **Responsive Design and RESS**; this section is the *authoring* decision tree for Canvas pages.

Step 1 — Pick OnResize by what the page is

`OnResize` is set once per display (Draw Display Settings, or the `OnResize` JSON field). The five values and when each one is correct:

Page is…	OnResize	Why
A Canvas whose prominent element should grow — a DataGrid, TrendChart, or AlarmViewer that should consume the extra space	`Responsive`	Children keep their native size by default; you opt specific elements into stretching/tracking with the element locks below. Nothing distorts that you didn't ask to.
A process diagram / drawing — P&ID, equipment layout, anything where geometry carries meaning	`StretchUniform`	Scales the whole canvas by one factor, preserving aspect ratio. A round tank stays round; pipe angles stay true. The default for HMI process screens.
A viewer / text page — text-led pages where everything should scale together	`ResizeChildren`	Reflows children proportionally with the region so the whole composition grows as one.
A Header / Footer / Menu region docked in the Layout	`Responsive` + a lock (see Step 3)	The band stretches along its free axis but stays pinned to native pixels on its docked axis.
A Popup / Dialog / Form	`NoAction`	A form is authored at the size it should be shown. Resizing it just moves controls away from where the operator expects them.

(`StretchFill`, the fifth value, stretches each axis independently — it *will* distort process geometry, so it is rarely the right pick for a Canvas. Reserve it for full-bleed background imagery.)

Step 2 — Element locks (Responsive only)

When `OnResize=Responsive`, four per-element flags let you choose exactly which children move or stretch as the container changes size. They are **ignored under every other OnResize value** — they only do anything on a Responsive display. Set them as flat boolean properties on the element; the platform translates them to the underlying attached properties on write.

Flat property	Effect	Set it on…
`IsWidthAlign`	**Stretch** the element's width with the container	A DataGrid / TrendChart / AlarmViewer that should fill horizontally; a full-width band or row element
`IsHeightAlign`	**Stretch** the element's height with the container	The same prominent element when it should also fill vertically
`IsLeftAlign`	**Move** the element to track the container's right edge	A control that must stay glued to the right edge as the page widens
`IsTopAlign`	**Move** the element to track the container's bottom edge	A bottom-pinned input row that must ride the bottom edge as the page grows taller

Common combinations:

- **Prominent resizable element** (DataGrid / TrendChart / AlarmViewer) `IsWidthAlign` + `IsHeightAlign`.
- **Band / row element** that should widen but keep its height `IsWidthAlign` only.
- **Bottom-pinned input row** `IsTopAlign` + `IsWidthAlign` (rides the bottom, grows wide).
- **Right-edge control** `IsLeftAlign` (tracks the right edge).
- **Small top-left labels / nav icons** no locks (they stay put at the origin).

Authoring rule (do not bypass): always express these as the flat properties `IsWidthAlign` / `IsHeightAlign` / `IsLeftAlign` / `IsTopAlign`. Never hand-write raw attached-property XAML — the importer owns that translation, and a hand-written attached property silently does nothing.

Step 3 — Display locks for docked Header / Footer / Menu

A region docked into the Layout (Header, Footer, Menu, SubMenu) is still a Canvas with its own `OnResize`. Two display-level booleans pin a docked band to native pixels on its docked axis while it stays Responsive on the free axis:

Display property	Pins…	Use on
`LockedHeight`	The docked band's **height** to native pixels	A docked Header or Footer — it spans full width but keeps its exact pixel height
`LockedWidth`	The docked rail's **width** to native pixels	A docked Menu or SubMenu rail — it spans full height but keeps its exact pixel width

So a docked Header is **`OnResize=Responsive` + `LockedHeight=true`**: the band stretches across the full window width (its contents can use element locks to track edges) but never grows taller. A docked left Menu is **`OnResize=Responsive` + `LockedWidth=true`**.

Match the Header's native Width to the Layout Width. If the Header is authored narrower than the Layout, Responsive will compress its contents horizontally to fit; author the Header at the Layout's full width so it lays out clean with no horizontal compression.

> **A docked Header is NOT `NoAction`.** The demo-proven rule is `OnResize=Responsive` + `LockedHeight`, with the Header's native Width matched to the Layout Width. `NoAction` on a docked band leaves it pinned to its authored size and it will not span the window. (`NoAction` is correct for a *Popup / Dialog / Form*, not for a docked region.)

> **Two different knobs — don't conflate them.** Setting a Layout region's *HorizontalAlign* = *Stretch* (so the region spans the full window width) is a Layout setting and is completely separate from the display's `OnResize`. "Header spans full width" is the region's HorizontalAlign; how the Header's contents reflow inside that region is the display's `OnResize`.

Step 4 — Worked examples (Asset Monitor Demo)

The `Asset Monitor Demo` solution is the reference for every rule above. Each row pairs a display with the exact OnResize + lock settings that ship:

Display	Native size	OnResize + locks	Element locks inside
Header	1546 × 60	`Responsive` + `LockedHeight`; native Width matched to the Layout Width (1546)	— (clean, no horizontal compression)
HeaderPad	1366 × 40	`Responsive` + `LockedHeight`	—
HeaderMobile	400 × 40	`NoAction` + `LockedWidth` + `LockedHeight`	— (fixed mobile band)
MenuTree	180 × 628	`Responsive` + `LockedWidth`	TAssetsTree: `IsWidthAlign` + `IsHeightAlign`; the three bottom TextBoxes: `IsTopAlign` + `IsWidthAlign`
Area	Responsive Canvas	`Responsive`	TDataGrid: `IsWidthAlign` + `IsHeightAlign`; TFlowPanel: `IsWidthAlign`; top-left label TextBoxes: no locks
Home / OEEDashboard	Responsive Dashboards	`Responsive`	None — reflow comes from a TGrid (Star columns) + TGridSplitter + 2 ChildDisplays, not element locks

OEE_Left / OEE_Right	card panels	`StretchUniform`	— (fixed-aspect cards inside the grid cells)
SiteDallas / SiteDetroit / SiteHouston / SiteEurope	site views	`ResizeChildren`	—

Step 5 — Landing-page and mobile patterns

- **Home = Dashboard with two ChildDisplays.** The landing page is a Dashboard "Home" that hosts two ChildDisplays (in the demo, Home holds `Home_Network` and `Home_Features`). Reflow is a TGrid with Star columns + a TGridSplitter, not element locks — the two child displays each carry their own responsive behavior.
- **Mobile = split the Canvas into two child displays.** When a Canvas page must also render on mobile, don't try to make one Canvas serve both form factors. Split it into two child displays, each its own Canvas, and let the Layout pick the right one per viewport.

Section 2 — Shape primitives

Call `list_elements('Shapes')` for the authoritative shape catalog in the current release. Common primitives — all Canvas-only, all theme-aware (`Fill`/`FillTheme`, `Stroke`/`StrokeTheme`, `StrokeThickness`):

Shape	Required	Use when
Rectangle	—	Panels, status bars, tank bodies, bars, backgrounds. Has `RadiusX`/`RadiusY` for rounded corners.
Ellipse	—	Tank caps (top/bottom), status LEDs, sight glasses, pump bodies. Circle when Width=Height.
Polygon	`Points` 3	Arrows, funnels, hoppers, diamonds. **Auto-closes.** Set `Stretch: "None"` to preserve exact point coords.
Polyline	`Points` 2	Open curved lines. Doesn't close. Prefer Gridline for pipes.
Connector	`StartElement`, `EndElement`	From 10.1.6. A line attached to two elements. See *Connector between two elements* below.
Path	`Data`	Any curve, arc, complex shape. SVG mini-language (`M L H V C Q A Z`).
Gridline	`Points` 2	**Use for pipes and orthogonal connections.** Constrained to horizontal/vertical segments. The P&ID convention.
Spline	`Points` 2	Smooth curves through control points (Catmull-Rom). Rare — Path covers most cases.

Pipe segment pattern

```
{ "Type": "Gridline", "Left": 300, "Top": 200, "Points": "0,0 100,0 100,60 200,60", "StrokeTheme": "Water",
"StrokeThickness": 6, "StrokeLineJoin": "Round", "StrokeLineCap": "Round" }
```

Then a direction arrow (Polygon, no stretch):

```
{ "Type": "Polygon", "Left": 490, "Top": 253, "Width": 20, "Height": 14, "Points": "0,0 20,7 0,14", "Stretch":
"None", "FillTheme": "Water" }
```

Flow arrows: static markers, not animated dots

Place a small Polygon triangle where direction needs to be obvious — typically at each tee, each entry, each exit, and once in the middle of long runs. Add a `VisibilityDynamic` so the arrow only appears when the flow tag is non-zero. Animated flow dots (moving circles along a pipe) are expensive at Canvas scale — a 1920×1080 canvas with a dozen pipe runs degrades Runtime frame rate. Reserve animation for mixers, impellers, and the one or two pipes where flow activity is the operator's primary question. Static arrows are enough for the rest.

Reactor coil overlay (Path)

Zigzag heat-exchange coils over a reactor body:

```
{ "Type": "Path", "Left": 305, "Top": 220, "Width": 130, "Height": 240, "Data": "M0,0 Q65,20 130,0 M0,40 Q65,60
130,40 M0,80 Q65,100 130,80", "StrokeTheme": "StateRed", "StrokeThickness": 2, "Fill": "#00000000", "FillTheme":
"" }
```

Note the **stroked-but-unfilled** pattern: `Fill: "#00000000"` + `FillTheme: ""` gives a transparent fill so the stroke shows without a filled shape underneath.

Connector between two elements (from 10.1.6)

From FrameworX 10.1.6, write `Type: "Connector"` to draw a line attached to two elements. Name the two elements by `Uid`. The platform derives the anchors, the geometry and the attachment at both ends.

```
{ "Type": "Connector", "Uid": "conn1", "StartElement": "pump1", "EndElement": "tank1", "StartSide": "Auto",
"EndSide": "Auto", "Stroke": "#FF000000", "StrokeThickness": 2 }
```

Property	Rule
`StartElement` / `EndElement`	Required. The `Uid` of each element, not its name. Two different elements in the same display.
`StartSide` / `EndSide`	`Auto` (default, the side facing the other element), `Left`, `Right`, `Top`, `Bottom` or `Center`.
`StartTerminal` / `EndTerminal`	1-based terminal on a symbol with more than one. The end is drawn to this terminal and follows the symbol's size and rotation. A terminal takes precedence over the side. Omitted: the terminal nearest the side. A terminal the symbol does not have: a warning, and the nearest terminal instead.
`Points`	Optional path, space-separated `x,y` pairs. The first and last points are the connection points. Omitted: a straight line.

- Write the connector in the same `Elements` array as its endpoints, in any position. A connector used as an endpoint of another connector comes first in the array.
- Do not set `Left`, `Top`, `Width`, `Height` or `Connections`. The platform derives them.
- Create-only. A connector `Uid` already in the display is an error: delete the connector and write it again.
- Read-back: `get_objects(detail='full')` returns a `Polyline` with `IsConnector` and `Connections`, never `Type: "Connector"`.
- Canvas only. Dashboard displays are not supported in 10.1.6.
- A solution built from an AI Console workspace without the Designer draws a terminal end to the symbol side instead, records the terminal and returns a warning for each such end. Importing the same workspace through the AI Designer draws it to the terminal.

Section 3 — First-class auto-shapes

These are **first-class shape primitives** that write with just `Type` + geometry + colors. The platform auto-injects the underlying Path/Polygon geometry on save. Massive productivity win over composing from primitives. Call `list_elements('Shapes')` for the authoritative list in the current release. Common auto-shapes:

Type	Default geometry	Use for
Cylinder	Vertical cylinder with elliptical caps	Tanks, vessels, drums, silos
Gear	8-tooth gear	Machinery icons, manual/auto toggles
Arrow	Right-pointing arrow	Flow direction, callouts (rotate via RotateDynamic for other directions)
Cloud	Puffy cloud outline	MQTT broker, cloud service, weather
Star	5-pointed star	Favorites, highlights, quality marker
Hexagon	Regular hexagon	Node diagrams, honeycomb layouts
Pentagon	Regular pentagon	Rare — use for ANSI warning signs
Trapezoid	Isosceles trapezoid	Hoppers, funnels, cone-bottom tank sections

```
{
  "Type": "Cylinder",
  "Left": 300, "Top": 200,
  "Width": 80, "Height": 180,
  "FillTheme": "ElementBlue"
}
```

That's an entire vessel. No Points, no Data, no compositing.

Writer normalization — read-back shows expanded form

When you read a display back after writing a `Cylinder`, you'll see a `Path` with auto-generated Data. This is expected — the shortcut is a write-time macro. For edits, either add new Cylinders (which normalize the same way) or edit the Path directly.

Runtime discovery over hardcoded lists

`list_elements()` is the authoritative runtime catalog. Any shape entry not returned by `list_elements()` in the current release should be treated as non-existent. For shapes beyond the standard set (e.g., triangle, octagon, custom forms), compose from primitives: `Polygon` with explicit `Points` handles most custom 2D shapes, and `Path` handles curves.

Section 4 — Containers (ShapeGroup, SvgGroup, Group)

Container	Children	Dynamics apply to	Use for

ShapeGroup	Shapes only	ALL children uniformly	"The whole vessel turns alarm red when Running=0"
SvgGroup	Auto-parsed from inline SVG string	ALL children (normalized to ShapeGroup on write)	"I have an SVG, I want dynamics per element"
Group	Any element type	Each child has independent dynamics	"Interactive panel with chart+buttons that moves as a unit"

ShapeGroup — compose equipment with unified state

The **killer feature**: a FillColorDynamic on the ShapeGroup changes the fill of ALL children at once. Build a vessel from primitives, and the entire vessel can turn red on alarm:

```
{ "Type": "ShapeGroup", "Left": 300, "Top": 200, "Width": 180, "Height": 220, "Stretch": "None", "Children": [ { "Type": "Ellipse", "Left": 40, "Top": 0, "Width": 100, "Height": 20 }, { "Type": "Rectangle", "Left": 40, "Top": 10, "Width": 100, "Height": 180 }, { "Type": "Ellipse", "Left": 40, "Top": 180, "Width": 100, "Height": 20 } ], "FillTheme": "ElementBlue", "Dynamics": [ { "Type": "FillColorDynamic", "LinkedValue": "@Tag.Reactor/Alarm", "ChangeColorItems": [ { "Type": "ChangeColorItem", "ChangeLimit": 0, "LimitColor": "#FF1E3A8A" }, { "Type": "ChangeColorItem", "ChangeLimit": 1, "LimitColor": "#FFEF4444" } ] } ] }
```

SvgGroup — when you'd rather author in SVG

Any inline SVG string becomes a ShapeGroup with native WPF shapes. **Supported SVG elements:** `rect`, `circle`, `ellipse`, `line`, `path`, `polyline`, `polygon`, `g`. **Trade-off:** SvgContent hex colors are opaque to the theme system. Fine for process-meaning colors, wrong for UI chrome. If you need theme-reactive composed equipment, use ShapeGroup directly.

Group — for interactive panels

Use `Group` when children need INDEPENDENT dynamics (a panel with its own chart, buttons, and labels that moves as a unit but where each child has its own behavior).

Section 4.5 — Choosing a symbol source

Before composing equipment from primitives, look for an existing symbol. The lookup order is **Solution Library/HPG Library/HMI Wizard primitives** — using the wrong source creates inconsistency across a solution's displays.

The full lookup, a per-equipment-class recommended-path table (10 equipment families), sizing tables (Min / Compact / Standard / Hero per family), SymbolLabels keys per Wizard, discovery commands, and the symbol-choice pitfalls live in **Skill Display Construction - HMI Symbol Choice**. Load that skill alongside this one whenever the task is "which symbol do I use for this piece of equipment?"

Rule: **when writing to an existing solution, run `list_elements('Solution')` before reaching for a Library symbol or a Wizard.** Match what's already there.

Section 5 — The equipment cookbook

Recipe 1 — Vessel with jacket

The archetype: a cylindrical vessel with a heating jacket that changes color when the heater is running.

```
{ "Type": "Ellipse", "Left": 300, "Top": 180, "Width": 96, "Height": 24, "FillTheme": "ElementGray" }, { "Type": "Rectangle", "Left": 328, "Top": 150, "Width": 40, "Height": 36, "FillTheme": "ElementGray" }, { "Type": "Cylinder", "Left": 276, "Top": 200, "Width": 144, "Height": 300, "FillTheme": "ElementBlue" }, { "Type": "Rectangle", "Left": 256, "Top": 280, "Width": 22, "Height": 180, "FillTheme": "OffFill", "Dynamics": [ { "Type": "FillColorDynamic", "LinkedValue": "@Tag.Reactor/Heater/Running", "ChangeColorItems": [ { "Type": "ChangeColorItem", "ChangeLimit": 0, "LimitColor": "#FF7F1D1D" }, { "Type": "ChangeColorItem", "ChangeLimit": 1, "LimitColor": "#FFEF4444" } ] } ] }, { "Type": "Rectangle", "Left": 418, "Top": 280, "Width": 22, "Height": 180, "FillTheme": "OffFill", "Dynamics": [ /* same FillColorDynamic */ ] }
```

Elements top to bottom: drive cap (Ellipse), motor housing (Rectangle), vessel body (Cylinder with ElementBlue), left jacket rail (Rectangle with heater-gated FillColorDynamic), right jacket rail (same).

For an impeller shaft: add a thin vertical Rectangle, apply RotateDynamic gated by `Running`:

```
{ "Type": "Rectangle", "Left": 344, "Top": 230, "Width": 8, "Height": 240, "FillTheme": "DefaultStroke", "Dynamics": [ { "Type": "RotateDynamic", "LinkedValue": "30", "IsRpm": true, "OnOffLink": "@Tag.Reactor/Agitator/Running" } ] }
```

`LinkedValue: "30"` and `IsRpm: true` means "rotate at 30 rpm." `OnOffLink` gates the rotation — the shaft only spins when Running=1.

Recipe 2 — Wizard symbol (the fast path for canonical equipment)

Five Wizard symbols — `Wizard/BLOWER`, `Wizard/MOTOR`, `Wizard/PUMP`, `Wizard/TANK`, `Wizard/VALVE` — are pre-wired with common state dynamics. When one of these five fits the equipment, it's almost always the fastest path.

The AI's job when dropping a Wizard:

1. Place it (`Left`, `Top`, `Width`, `Height`)
2. Wire its `SymbolLabels` (State, RPM, whatever the symbol exposes) to tags
3. Stop there

Visual customization — orientation, style variant, pipe-connection direction, foot details — is handled by the user in Designer via the Wizard configuration button (double-click the symbol in the editor). Do not attempt to reproduce these variants via raw geometry or extra elements; the Wizard holds them.

Equipment vocabulary which Wizard. Map the operator's words to the right `SymbolName` before reaching for a Library or composing from primitives. `Wizard/BLOWER` is for fans, blowers, and forced-draft units — anything that moves air. `Wizard/MOTOR` is for the rotating drive itself when it's the focal element, decoupled from a pump or fan. `Wizard/PUMP` covers any liquid mover (centrifugal, positive-displacement, dosing). `Wizard/TANK` is for vessels with level dynamics — storage tanks, drums, day tanks, surge vessels. `Wizard/VALVE` is the universal flow-control element — manual, motorized, on/off, modulating. When the equipment in the spec doesn't fall cleanly into one of these five categories, fall through to the §4.5 source order (Solution Library /HPG Library/HMI primitives).

Rename and SymbolLabels. Wizards drop with a generic prefix (`PUMP1`, `PUMP2`, `TANK1`) — that's just the placed-instance name on the canvas, not the bound tag. The actual data wiring is in `SymbolLabels`: each label `Key` matches a slot the symbol declares internally (`State`, `RPM`, `Level`, `Position`), and `LabelValue` is the `@Tag.<path>` that drives it. The same `Wizard/PUMP` placed twice with different SymbolLabels payloads becomes two independently-bound pumps. Never bind via direct properties on the Symbol element itself — SymbolLabels is the only path data takes into a Wizard.

```
{
  "Type": "Symbol",
  "SymbolName": "Wizard/PUMP",
  "Left": 500, "Top": 300,
  "Width": 80, "Height": 80,
  "SymbolLabels": [
    { "Type": "SymbolLabel", "Key": "State", "LabelName": "State", "LabelValue": "@Tag.Pump1/Running",
"FieldType": "Expression" },
    { "Type": "SymbolLabel", "Key": "RPM", "LabelName": "RPM", "LabelValue": "@Tag.Pump1/Speed", "FieldType":
"Expression" }
  ]
}
```

The symbol knows how to change its fill based on `State` — no extra FillColorDynamic needed. To add a click-to-open-detail action, put the ActionDynamic directly on the Symbol:

```
"Dynamics": [ { "Type": "ActionDynamic", "MouseLeftButtonDown": { "Type": "DynamicActionInfo", "ActionType":
"OpenDisplay", "ObjectLink": "PumpDetail" } } ]
```

For equipment outside the five Wizard types, follow the symbol source order in §4.5 — Solution first, then HPG Library, then HMI Library, then primitives.

Recipe 3 — Pipe segment with flow direction

```
{
  "Type": "Gridline",
  "Left": 200, "Top": 400,
  "Width": 300, "Height": 60,
  "Points": "0,30 100,30 100,0 200,0 200,60 300,60",
  "StrokeTheme": "Water",
  "StrokeThickness": 6,
  "StrokeLineJoin": "Round",
  "StrokeLineCap": "Round"
}
```

Then a direction arrow (Polygon, no stretch) with a VisibilityDynamic gated on flow:

```
{
  "Type": "Polygon",
  "Left": 490, "Top": 53,
  "Width": 20, "Height": 14,
  "Points": "0,0 20,7 0,14",
  "Stretch": "None",
  "FillTheme": "Water",
  "Dynamics": [
    { "Type": "VisibilityDynamic", "LinkedValue": "@Tag.Pipe1/FlowRate" }
  ]
}
```

The arrow only appears when FlowRate is non-zero — a simple, strong visual cue.

For multi-colored pipe based on flow threshold:

```
"Dynamics": [
  {
    "Type": "LineColorDynamic",
    "LinkedValue": "@Tag.Pipe1/FlowRate",
    "ChangeColorItems": [
      { "Type": "ChangeColorItem", "ChangeLimit": 0, "LimitColor": "#FF64748B" },
      { "Type": "ChangeColorItem", "ChangeLimit": 10, "LimitColor": "#FF38BDF8" },
      { "Type": "ChangeColorItem", "ChangeLimit": 50, "LimitColor": "#FF0369A1" }
    ]
  }
]
```

Recipe 4 — Reactor with heating coils

Uses Recipe 1 (vessel) + Path overlay for coils:

```
{ /* Recipe 1 elements for vessel body with jacket */ },
{
  "Type": "Path",
  "Left": 305, "Top": 220,
  "Width": 130, "Height": 240,
  "Data": "M0,0 Q65,20 130,0 M0,40 Q65,60 130,40 M0,80 Q65,100 130,80 M0,120 Q65,140 130,120 M0,160 Q65,180
130,160 M0,200 Q65,220 130,200",
  "StrokeTheme": "StateRed",
  "StrokeThickness": 2,
  "Fill": "#00000000",
  "FillTheme": "",
  "Dynamics": [
    { "Type": "VisibilityDynamic", "LinkedValue": "@Tag.Reactor/Heater/Running" }
  ]
}
```

The coils are only visible when the heater is active — a subtle but unmistakable visual cue for operators.

Recipe 5 — Reference displays to study

For a worked canvas at scale, read displays from the `Industrial_Ontology_Enhanced_Demo` solution (solution_id `J8P7LY`). The display `ProcessAreaOverview` is a good starting anchor — it demonstrates zone decomposition, inline readouts attached to equipment, pipe/gridline composition, and the use of Library symbols alongside primitives. Read with `get_objects('DisplaysList', names=['ProcessAreaOverview'], detail='full')` to see how the pieces compose at scale.

Section 6 — Dynamics reference

Call `list_dynamics()` for the full list. The 14 types grouped by what they do:

Action category

Dynamic	What it does	Use for
`ActionDynamic`	Run an action on mouse event	Navigation, tag writes, scripts, toggles
`CodeBehindDynamic`	Run code on display lifecycle events	Init, cleanup, periodic updates
`HyperlinkDynamic`	Open external URL on click	Documentation links, external dashboards

Color category

Dynamic	What it does	Use for
`FillColorDynamic`	Change element Fill based on thresholds	Status indicators, process meaning
`LineColorDynamic`	Change element Stroke based on thresholds	Pipe color-by-flow, boundary-alarm borders
`TextColorDynamic`	Change text Foreground based on thresholds	Alert text, highlighted values

Animation category (Canvas-only)

Dynamic	What it does	Use for
`RotateDynamic`	Rotate element by angle or rpm	Pointers, impeller shafts, motors, compass needles
`ScaleDynamic`	Scale element by tag value	Growth/shrink animations, level indicators
`MoveDragDynamic`	Move element by tag value OR let user drag	Sliding valves, operator drag-to-set
`SkewDynamic`	Skew element	Perspective effects, rare

Data category (Canvas-only)

Dynamic	What it does	Use for

`BargraphDynamic`	Fill a Rectangle proportionally based on value	Tank level fill, progress fills, power meter strips

Visibility, Security, Feedback

Dynamic	What it does	Use for
`VisibilityDynamic`	Show/hide based on expression	Conditional panels, state-dependent overlays, coils-only-when-heating
`SecurityDynamic`	Hide/disable based on user permission	Admin-only controls, role-based HMI
`ShineDynamic`	Mouse-over highlight / glow effect	Hover feedback on clickable shapes

Copy-paste patterns

status_indicator (shape that changes color on running/stopped):

```
{
  "Type": "Ellipse",
  "Left": 100, "Top": 100,
  "Width": 24, "Height": 24,
  "Dynamics": [
    {
      "Type": "FillColorDynamic",
      "LinkedValue": "@Tag.Equipment/Running",
      "ChangeColorItems": [
        { "Type": "ChangeColorItem", "ChangeLimit": 0, "LimitColor": "#FF808080" },
        { "Type": "ChangeColorItem", "ChangeLimit": 1, "LimitColor": "#FF34D399" }
      ]
    }
  ]
}
```

toggle_button (click toggles a boolean, color reflects state):

```
{
  "Type": "Rectangle",
  "Left": 100, "Top": 200,
  "Width": 120, "Height": 40,
  "Dynamics": [
    {
      "Type": "ActionDynamic",
      "MouseLeftButtonDown": {
        "Type": "DynamicActionInfo",
        "ActionType": "ToggleValue",
        "ObjectLink": "@Tag.Motor/Start"
      }
    },
    {
      "Type": "FillColorDynamic",
      "LinkedValue": "@Tag.Motor/Start",
      "ChangeColorItems": [
        { "Type": "ChangeColorItem", "ChangeLimit": 0, "LimitColor": "#FFEF4444" },
        { "Type": "ChangeColorItem", "ChangeLimit": 1, "LimitColor": "#FF34D399" }
      ]
    },
    { "Type": "ShineDynamic" }
  ]
}
```

animated_motor (continuous rotation gated by boolean):

```
{
  "Type": "Symbol",
  "SymbolName": "Wizard/MOTOR",
  "Left": 400, "Top": 300,
  "Width": 80, "Height": 80,
  "Dynamics": [
    { "Type": "RotateDynamic", "LinkedValue": "30", "IsRpm": true, "OnOffLink": "@Tag.Motor/Running" }
  ]
}
```

level_bar (tank-fill effect):

```
{
  "Type": "Rectangle",
  "Left": 200, "Top": 200,
  "Width": 120, "Height": 240,
  "Dynamics": [
    {
      "Type": "BargraphDynamic",
      "LinkedValue": "@Tag.Tank/Level",
      "MinValueLink": "0",
      "MaxValueLink": "100",
      "BarColor": "#FF38BDF8",
      "Orientation": "VerticalUp"
    }
  ]
}
```

ActionDynamic action types

ActionType	Extra fields	What it does
`OpenDisplay`	`ObjectLink: "DisplayName"`	Navigate to another display
`ToggleValue`	`ObjectLink: "@Tag.X"`	Flip a boolean tag
`SetValue`	`ObjectLink: "@Tag.X"`, `ObjectValueLink` per the four-family rule: numeric `"100"`, boolean `"true"`, tag /expression `"@Tag.Setpoint"`, or string-literal `"\"Dark\""` (escape-wrapped quotes; a bare `"Dark"` silent no-ops)	Write a specific value
`RunScript`	`ActionScript: "MethodName"`	Run a CodeBehind method

Mouse events beyond `MouseLeftButtonDown`: `MouseRightButtonDown`, `MouseDoubleClick`, `MouseEnter`, `MouseLeave`. Each is a top-level key on the ActionDynamic.

Format rules

- **Dynamics go in the `Dynamics` array.** Never as direct properties on the element.
- **`ChangeColorItems` is a flat array.** Not `{"Type": "ColorChangeList", "Children": [...]}`.
- An element can have MULTIPLE dynamics of different types in the same `Dynamics` array.

Section 7 — Navigation: headers own page links

Page-to-page navigation belongs in the **Header display**, not on content pages. Content pages only get in-page actions (start/stop, open popup, acknowledge).

Workflow

1. Build all your content pages first.
2. Read the Startup layout: `get_objects('DisplaysLayouts', names=['Startup'], detail='full')`.
3. Read the Header display: `get_objects('DisplaysList', names=['Header'], detail='full')`.
4. Add navigation buttons right-aligned in the header (100–130 × 30–35, 10px gap).
5. Write the modified Header display back.

Back-navigation on detail pages

```
{ "Type": "TextBlock", "Left": 48, "Top": 32, "Width": 400, "Height": 20, "LinkedValue": " Process Area Overview", "ForegroundTheme": "AccentBrush", "Dynamics": [ { "Type": "ActionDynamic", "MouseLeftButtonDown": { "Type": "DynamicActionInfo", "ActionType": "OpenDisplay", "ObjectLink": "ProcessAreaOverview" } } ] }
```

Section 7.5 — Embedded vs separate display vs popup

Side-rail and inspector panels: when the secondary view is dependent on the main view's selection or context, embed it as a right-rail zone in the same Canvas (reserve 360–560 px on the right) or drop a `ChildDisplay` cell. Separate-display + ASK button is correct ONLY when the secondary view is independent and full-screen worthy. Default to embedded for chat, inspectors, and asset detail; reserve `OpenDisplay` for navigation between pages of equal weight.

This is the same zone-first reasoning from §1: a context-dependent secondary view is just another zone on the current canvas, not a separate page. `OpenDisplay` replaces the current canvas entirely, and the operator loses the selection context that the secondary view depends on. Reserve it for peer pages where the destination has its own selection state and is full-screen worthy in its own right.

Section 7.6 — First Page polish

When the display is the operator's landing page or the demo's headline, lift visual quality:

1. **Prefer Library/HMI symbols over primitives for every vessel.** The first page is the one customers screenshot; composed primitives read as a sketch, not a product. Follow the §4.5 source order strictly here.
2. **Reserve a 1/3 right-rail or bottom band** for live KPIs, chat, or asset tree. Use the §7.5 embedded pattern; never split the headline view to a separate page.
3. **Saturated color only on state changes.** Nominal state stays in desaturated theme brushes. The eye should be drawn to anomalies, not to chrome.
4. **Show three things at once:** where you are (asset path), what's running (active operation + batch), what's wrong (alarm summary or all-clear). These three answers belong in the title strip and the right-rail zone; the central canvas is for the spatial layout itself.

The landing page is a Home Dashboard

The headline / landing page itself is best built as a **Dashboard "Home"** hosting two ChildDisplays (see §1.6 Step 5) — one for navigation / network context, one for the feature or KPI summary — with a TGrid + TGridSplitter for reflow. The two ChildDisplays can each be a Canvas. Keep the headline Canvas content inside those child displays rather than authoring one monolithic page; it keeps the landing page responsive and lets the mobile split (§1.6 Step 5) reuse the same child Canvases.

Keep MainPage in place — never rename to "consolidate." The landing page in this pattern is named `Home`; the template's `MainPage` (Display ID 0) is a *separate*, reserved page — the Solution Center preview anchor. Build `Home`, point the Startup layout's Content region at it, and leave `MainPage` in place (a static placard is enough for the thumbnail). Do **not** rename `MainPage` out of the way, or rename `Home` onto it: renaming the ID-0 page builds green but silently detaches the preview anchor. An orphaned-looking `MainPage` beside a wired `Home` is the intended state — not a cleanup target. See Displays List Reference § Special Pages.

Section 8 — Anti-patterns (what Canvas is not)

The following patterns are common when a Canvas is built by someone whose background is SaaS dashboards. Each breaks the operator's scan pattern.

1. **Rounded cards around equipment.** Wrapping each tank or pump in a Rectangle with `RadiusX:8 RadiusY:8` and a shadow turns the canvas into a dashboard. The zone border is the only "card" the operator should see; equipment lives inside the zone by geometry, not inside a card.
2. **Hero numbers on equipment-attached values.** Live values attached to a vessel, pump, or pipe are inline readouts (FontSize ~14–22), not Dashboard CenterValue tiles. The only exception is a single top-line header value intended to be read from across a control room.
3. **Decorative icons next to text labels.** A pump already has a tag (`P-101A`) and a state word (`RUN`). An additional pump icon next to the label costs scan time without adding information. On Canvas the *pictogram itself is the equipment*; an additional decorative icon is redundant.
4. **Gradient or textured backgrounds.** Any non-flat background adds a non-data variable to the scene that the operator's eye has to track. Use `theme:PageBackground` directly; leave gradients off.
5. **Lowering `IsOpenInterval` below the 1000 ms default.** Leave `IsOpenInterval` at the default 1000 ms unless `DisplayIsOpen()` has measurable per-frame work. Lower values cost CPU on every connected client without observable benefit when CodeBehind is empty.
6. **Setting a docked Header / Footer / Menu to `NoAction`.** A docked band must be `OnResize=Responsive` + the matching lock (`LockedHeight` for Header/Footer, `LockedWidth` for Menu), with the Header's native Width matched to the Layout Width (§1.6 Step 3). `NoAction` leaves the band pinned to its authored size so it will not span the window; that value is for Popups / Dialogs / Forms only.
7. **Hand-writing raw attached-property XAML for element locks.** Express width/height stretch and edge-tracking through the flat properties `IsWidthAlign` / `IsHeightAlign` / `IsLeftAlign` / `IsTopAlign` (§1.6 Step 2). Raw attached-property markup is silently dropped on import.

Section 9 — Canvas quirks and write-time normalizations

- **Cylinder / Gear / Arrow / Cloud / Star / Hexagon / Pentagon / Trapezoid** become `Path` or `Polygon` on disk. Auto-geometry expanded.
- **SvgGroup ShapeGroup** on save. SVG parsed into native WPF shapes.
- **Theme brushes** stored with `theme:` prefix: `{"FillTheme": "ElementBlue"}` `{"Fill": "theme:ElementBlue"}`.
- **`Stretch: "Fill"` default silently dropped** on Polygon/Polyline/Path/Spline. If you want `Stretch: "None"`, it IS stored.
- **Element locks (`IsWidthAlign` / `IsHeightAlign` / `IsLeftAlign` / `IsTopAlign`) and display locks (`LockedWidth` / `LockedHeight`)** are flat properties on write; the platform stores them in the attached-property / display-body form. Read-back may show the translated form — this is expected. Keep authoring with the flat properties.
- **Background default #FFFAFAFA baked in.** Explicitly set `Background: "theme:PageBackground"`.
- **Polygon without Points silently skipped.** Always include `Points`.

Section 10 — Canvas checklist

- ? Fidelity mode settled when a source drawing was supplied — Replica (positions extracted from the source) or Derived (re-composed), stated back to the user (§1.2)
- ? `Background` set explicitly to `theme:PageBackground` (not the #FFFAFAFA default)
- ? Zones calculated to fill the full canvas width and height
- ? Every zone has a background Rectangle with `FillTheme: "PanelBackground"` — FIRST in Elements
- ? Symbol source lookup order followed: Solution Library/HPG Library/HMI Wizard primitives (§4.5)
- ? Symbols 60×60 (Wizard), 80×80 recommended on process overview displays

- ? Library symbols scaled proportionally — maintain aspect ratio
- ? No equipment symbols overlap unless intentionally layered
- ? Zone titles FontSize 14, value text FontSize 12
- ? Value and Unit in the SAME TextBlock with composite LinkedValue, OR adjacent TextBlocks (inline readout pattern)
- ? Live values attached to equipment use inline TextBlocks, not `CenterValue`
- ? No hero numbers on live values (FontSize 22, except one wall-display header KPI)
- ? Nominal-state elements use desaturated theme brushes; saturated color reserved for warning/alarm
- ? No element extends beyond the display's Width/Height
- ? `OnResize` chosen by page type: `Responsive` for a Canvas whose prominent element should resize; `StretchUniform` for process diagrams / drawings; `ResizeChildren` for viewer/text pages; `NoAction` for Popups / Dialogs / Forms (§1.6 Step 1)
- ? Docked Header / Footer / Menu set to `Responsive` + the matching lock (`LockedHeight` for Header/Footer, `LockedWidth` for Menu) — never `NoAction`; Header native Width matched to the Layout Width (§1.6 Step 3)
- ? On a `Responsive` Canvas, element locks set deliberately — prominent resizable element gets `IsWidthAlign` + `IsHeightAlign`; bottom-pinned input row gets `IsTopAlign` + `IsWidthAlign`; right-edge control gets `IsLeftAlign`; small top-left labels/nav icons get no locks (§1.6 Step 2). Authored as flat properties, never raw attached-property XAML
- ? Page navigation is in the Header, not on content pages
- ? All dynamics inside `Dynamics` array (never as direct properties)
- ? `ChangeColorItems` is a flat array (no ColorChangeList wrapper)
- ? All shape / symbol types verified via `list_elements()` before use
- ? `get_state` after write shows `errorList` empty
- ? `get_display_tree(element=<DisplayName>)` returns — for each interactive control carrying a `LabelLink`, `labelResolved` matches the authored intent (no default caption left in place, no unexpected empty text)
- ? Resolved fill and foreground colors on each themed element fall in the theme palette — no hard-coded hex bleed-through under the active theme
- ? Every shape, symbol, and equipment instance the recognition map referenced appears in the resolved tree at the expected location (no silently-skipped elements)

Section 11 — Quick reference

The 10 element types you'll actually use on most Canvas displays

```
Rectangle // backgrounds, bars, pipes (when rectangular) Ellipse // caps, LEDs, pump bodies Polygon // arrows, funnels, custom closed shapes Gridline // pipes (always prefer over Polyline) Path // curves, coils, complex shapes Cylinder // vessels, tanks (first-class shortcut) TextBlock // all text (labels, values, composite) Symbol // Solution / Library / Wizard ShapeGroup // composed equipment with unified dynamics RadialGauge // temperature, pressure, any circular gauge
```

OnResize at a glance

Page type	OnResize	Locks
Canvas, prominent element resizes (DataGrid / TrendChart / AlarmViewer)	`Responsive`	element: `IsWidthAlign` + `IsHeightAlign` on that element
Process diagram / drawing (P&ID, equipment layout)	`StretchUniform`	—
Viewer / text page	`ResizeChildren`	—
Header / Footer	`Responsive`	display: `LockedHeight`; match Header Width to Layout Width
Menu / SubMenu rail	`Responsive`	display: `LockedWidth`
Popup / Dialog / Form	`NoAction`	—

Tool-call recipes

```
# Before first use of an element
list_elements('RadialGauge')

# Before first use of a dynamic
list_dynamics('FillColorDynamic')

# Pattern search in library
list_elements('Library/HMI/Pumps')

# Wizard catalog (always 5 symbols)
list_elements('Wizard')

# Theme brushes
list_elements('ThemeColors')
```