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.

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)

text 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 1.5 — 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 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.
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

json { "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):

json { "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:

json { "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.

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

<ac:structured-macro ac:name="code"> <ac:parameter ac:name="language">json</ac:parameter> ac:plain-text-body{ "Type": "Cylinder", "Left": 300, "Top": 200, "Width": 80, "Height": 180, "FillTheme": "ElementBlue" }</ac:plain-text-body> </ac:structured-macro>

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:

json { "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 search order matters — using the wrong source creates inconsistency across a solution's displays.

1. **Solution symbols first.** When working in an existing solution, `list_elements('Solution')` shows what the plant has already standardized on. If a `Solution/Tanks/Tank_Vertical` exists, use it — other displays in this solution already do. Mixing Solution symbols with Library symbols or Wizards for the same equipment type produces a display that visually disagrees with the rest of the plant's screens.
2. **HPG Library next.** `list_elements('Library/HPG')` for High Performance Graphics — flat, theme-aware symbols that respond to the standard state convention (`0`=Off/Stopped/Closed, `1`=On/Running/Open, `2`=Disabled/Out-of-Service) via the `HPOffFill`/`HPOnFill`/`HPDisableFill` theme brushes. This is the right default for operator control screens.
3. **HMI Library for detailed/realistic symbols.** `list_elements('Library/HMI')` — traditional detailed symbols with shading and depth. Right for training material, technical diagrams, and mechanical documentation. Wrong for live operator screens (too much visual noise competes with state communication).
4. **Wizard symbols for fast-path canonical equipment.** Five pre-wired Wizards cover the most common cases: `Wizard/BLOWER`, `Wizard/MOTOR`, `Wizard/PUMP`, `Wizard/TANK`, `Wizard/VALVE`. See Recipe 2.
5. **Compose from primitives (ShapeGroup)** only when none of the above fits. This is the slowest path and the one most likely to drift across displays.

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.

json { "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`:

json { "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.

<ac:structured-macro ac:name="code"> <ac:parameter ac:name="language">json</ac:parameter> ac:plain-text-body{ "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" }] }</ac:plain-text-body> </ac:structured-macro>

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:

json "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

json { "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" }, { "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" }] }]]></ac:plain-text-body> </ac:structured-macro> The arrow only appears when FlowRate is non-zero — a simple, strong visual cue. For multi-colored pipe based on flow threshold: <ac:structured-macro ac:name="code"> <ac:parameter ac:name="language">json</ac:parameter> <ac:plain-text-body><![CDATA["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:

json { /* 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" }] }]]></ac:plain-text-body> </ac:structured-macro> The coils are only visible when the heater is active — a subtle but unmistakable visual cue for operators. <h3>Recipe 5 — Reference displays to study</h3> For a worked canvas at scale, read displays from the <code>Industrial_Ontology_Enhanced_Demo</code> solution (solution_id <code>J8P7LY</code>). The display <code>ProcessAreaOverview</code> 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 <code>get_objects ('DisplaysList', names=['ProcessAreaOverview'], detail='full')</code> to see how the pieces compose at scale. <h2>Section 6 — Dynamics reference</h2> Call <code>list_dynamics()</code> for the full list. The 14 types grouped by what they do: <h3>Action category</h3> <table> <tbody> <tr><th>Dynamic</th><th>What it does</th><th>Use for</th></tr> <tr><td><code>ActionDynamic</code></td><td>Run an action on mouse event</td><td>Navigation, tag writes, scripts, toggles</td></tr> <tr><td><code>CodeBehindDynamic</code></td><td>Run code on display lifecycle events</td><td>Init, cleanup, periodic updates</td></tr> <tr><td><code>HyperlinkDynamic</code></td><td>Open external URL on click</td><td>Documentation links, external dashboards</td></tr> </tbody> </table> <h3>Color category</h3> <table> <tbody> <tr><th>Dynamic</th><th>What it does</th><th>Use for</th></tr> <tr><td><code>FillColorDynamic</code></td><td>Change element Fill based on thresholds</td><td>Status indicators, process meaning</td></tr> <tr><td><code>LineColorDynamic</code></td><td>Change element Stroke based on thresholds</td><td>Pipe color-by-flow, boundary-alarm borders</td></tr> <tr><td><code>TextColorDynamic</code></td><td>Change text Foreground based on thresholds</td><td>Alert text, highlighted values</td></tr> </tbody> </table> <h3>Animation category (Canvas-only)</h3> <table> <tbody> <tr><th>Dynamic</th><th>What it does</th><th>Use for</th></tr> <tr><td><code>RotateDynamic</code></td><td>Rotate element by angle or rpm</td><td>Pointers, impeller shafts, motors, compass needles</td></tr> <tr><td><code>ScaleDynamic</code></td><td>Scale element by tag value</td><td>Growth/shrink animations, level indicators</td></tr> <tr><td><code>MoveDragDynamic</code></td><td>Move element by tag value OR let user drag</td><td>Sliding valves, operator drag-to-set</td></tr> <tr><td><code>SkewDynamic</code></td><td>Skew element</td><td>Perspective effects, rare</td></tr> </tbody> </table> <h3>Data category (Canvas-only)</h3> <table> <tbody> <tr><th>Dynamic</th><th>What it does</th><th>Use for</th></tr> <tr><td><code>BargraphDynamic</code></td><td>Fill a Rectangle proportionally based on value</td><td>Tank level fill, progress fills, power meter strips</td></tr> </tbody> </table> <h3>Visibility, Security, Feedback</h3> <table> <tbody> <tr><th>Dynamic</th><th>What it does</th><th>Use for</th></tr> <tr><td><code>VisibilityDynamic</code></td><td>Show/hide based on expression</td><td>Conditional panels, state-dependent overlays, coils-only-when-heating</td></tr> <tr><td><code>SecurityDynamic</code></td><td>Hide/disable based on user permission</td><td>Admin-only controls, role-based HMI</td></tr> <tr><td><code>ShineDynamic</code></td><td>Mouse-over highlight / glow effect</td><td>Hover feedback on clickable shapes</td></tr> </tbody> </table> <h3>Copy-paste patterns</h3> status_indicator (shape that changes color on running/stopped): <ac:structured-macro ac:name="code"> <ac:parameter ac:name="language">json</ac:parameter> <ac:plain-text-body><![CDATA[{ "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):

json { "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):

json { "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):

json { "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: value`	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

json { "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 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.

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.
- **Background default #FFFAFAFA baked in.** Explicitly set `Background: "theme:PageBackground"`.
- **Polygon without Points silently skipped.** Always include `Points`.

Section 10 — Canvas checklist

- ? `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
- ? 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

Section 11 — Quick reference

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

text 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

Tool-call recipes