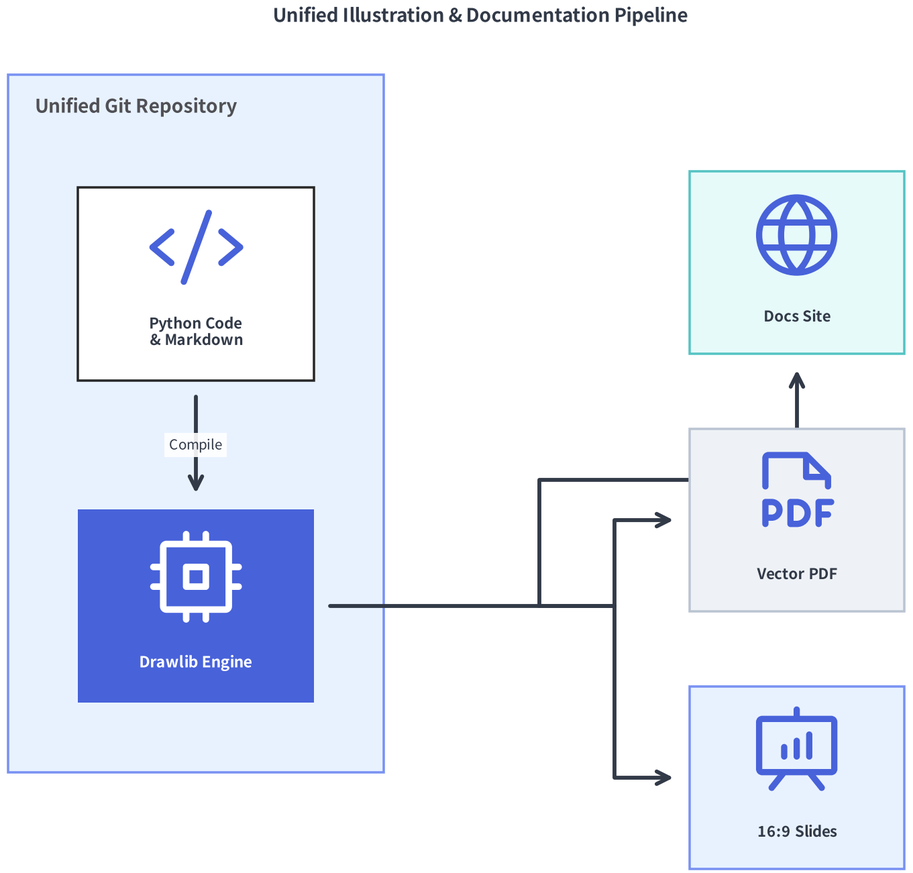
Drawlib

Illustration & Illustrated Documentation as Code

Unify **declarative Python diagrams**, **technical documentation**, **vector PDFs**, and **16:9 presentations** inside a single version-controlled Git repository.

- **Pure-Python Declarative API** — Zero GUI silos, raw SVG math, or rigid text DSLs
- **Single Source of Truth (SoT)** — Code, architecture diagrams, and docs evolve together in PRs
- **Autonomous AI Agent Ready** — Built-in rules catalog & multimodal visual self-healing loop



Drawlib Official Showcase Deck

Pure Python (`⩾3.11`) • Apache 2.0 License

What is Drawlib?

A declarative, pure-Python graphics and documentation engine that treats architectural diagrams, quantitative charts, animations, and technical documents as first-class software artifacts.

1. Pure-Python Declarative API

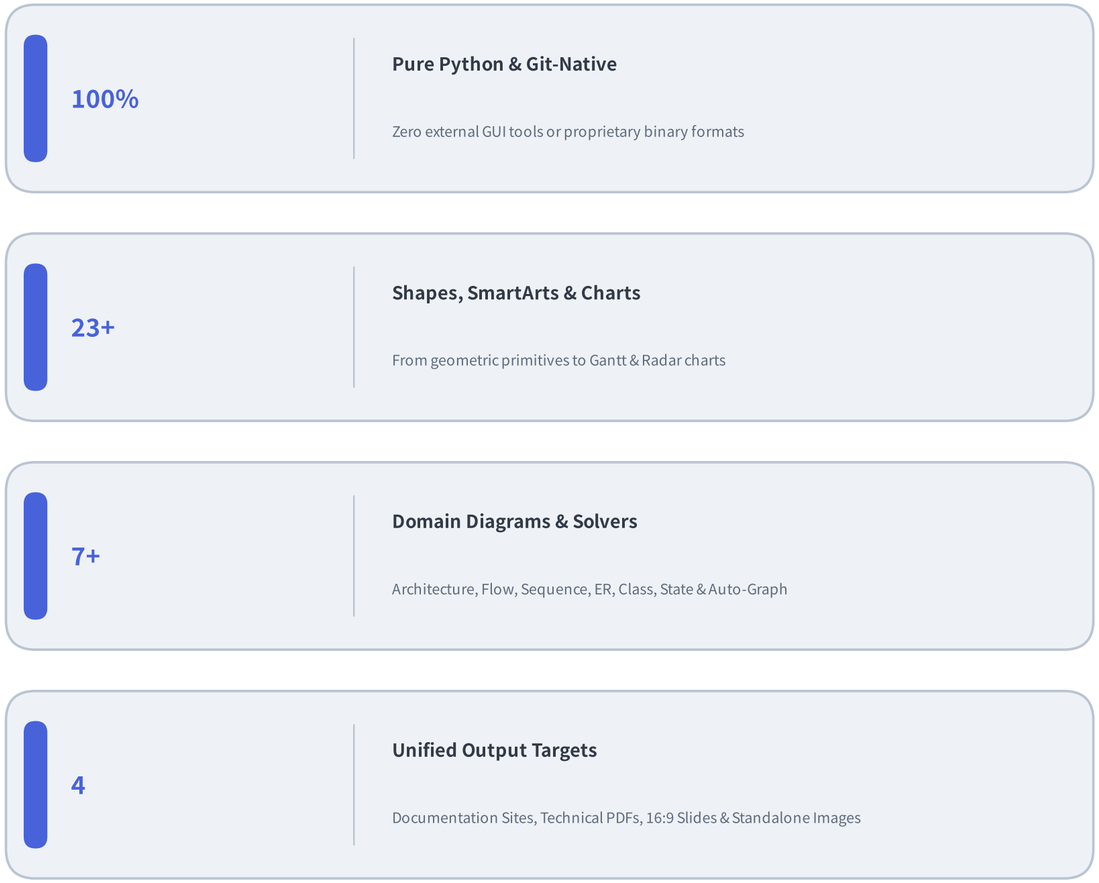
- Write expressive Python (`⩾3.11`) using variables, loops, type-checked models, and reusable project helpers (`styles.py` & `utils.py`).
- No opaque Graphviz/Java binaries, no fragile XML/SVG hand-editing, and no 200-line `matplotlib` patch boilerplate.

2. Four Publishing Archetypes

- Scaffold and build **Multi-Page Documentation Sites** (`site`), **Linear Technical Specs & Vector PDFs** (`doc`), **16:9 Widescreen Slide Decks** (`slide`), and **Standalone Images** (`image`) from one unified CLI.

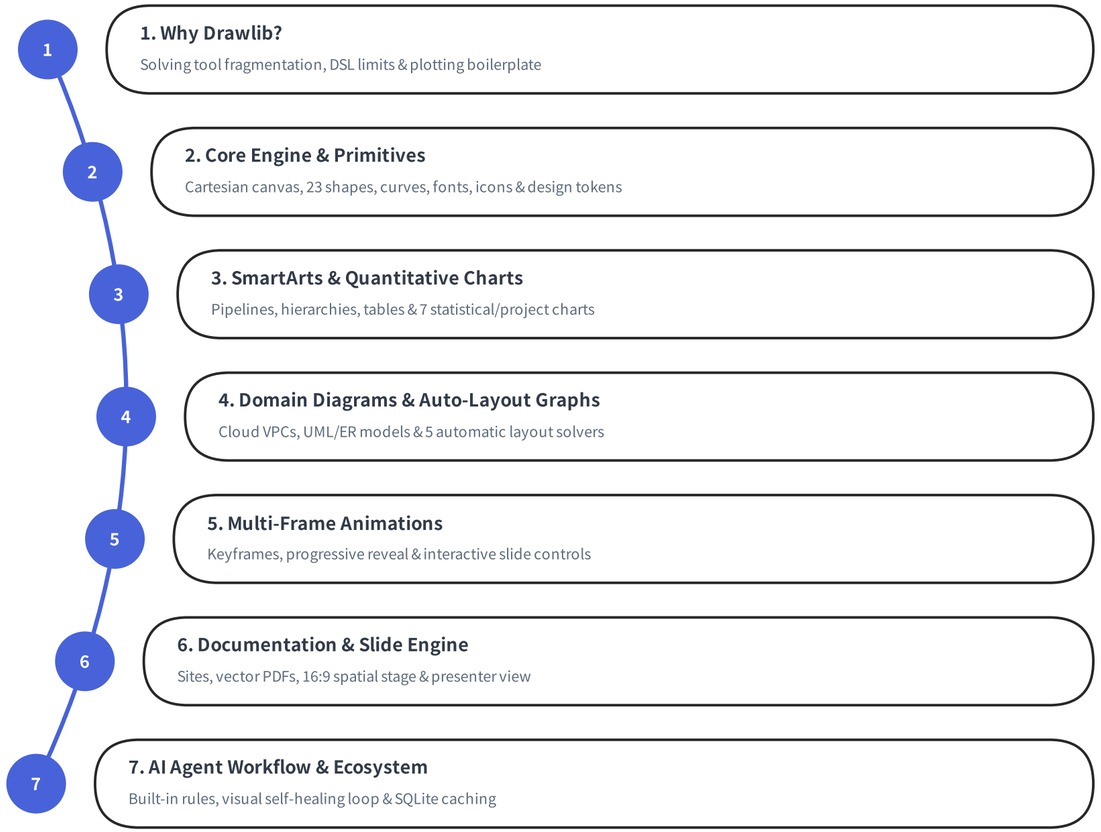
3. Autonomous AI Agent Ready

- Ships with an embedded **On-Demand Rules Catalog** (`drawlib rules show`) and headless coordinate-grid preview (`drawlib show -g`), enabling AI coding agents to



Presentation Roadmap

A comprehensive 7-chapter tour from the philosophy of Illustration as Code to high-level diagrams, animations, and AI workflows.



Live Dogfooding in Action

This 60-slide presentation is **100% authored and compiled with Drawlib**:

- **Source of Truth**: Markdown files and embedded `drawlib` Python blocks in `docs/slide_about_drawlib_src/`
- **Shared Theme & Macros**: Centralized palette in `styles.py` and reusable layout helpers in `utils.py`
- **Inline Vector SVGs**: Every static diagram renders as crisp inline `<svg>` with automatic `@font-face` bundling
- **Interactive `<canvas>` Animations**: Step-by-step animated diagrams controlled via `A` or Presenter View (`P`)
- **Dual Output**: Compiled into both an interactive HTML5 web deck and a 1920×1080 vector PDF

CHAPTER

01

Part 1 of 7

Why Drawlib?

The Paradigm Shift to Illustration & Documentation as Code

1.1 The Tool Fragmentation Problem in Engineering Teams

1.2 Why Text DSLs & Low-Level Plotting Fall Short

1.3 Comparison Matrix Across Diagramming Approaches

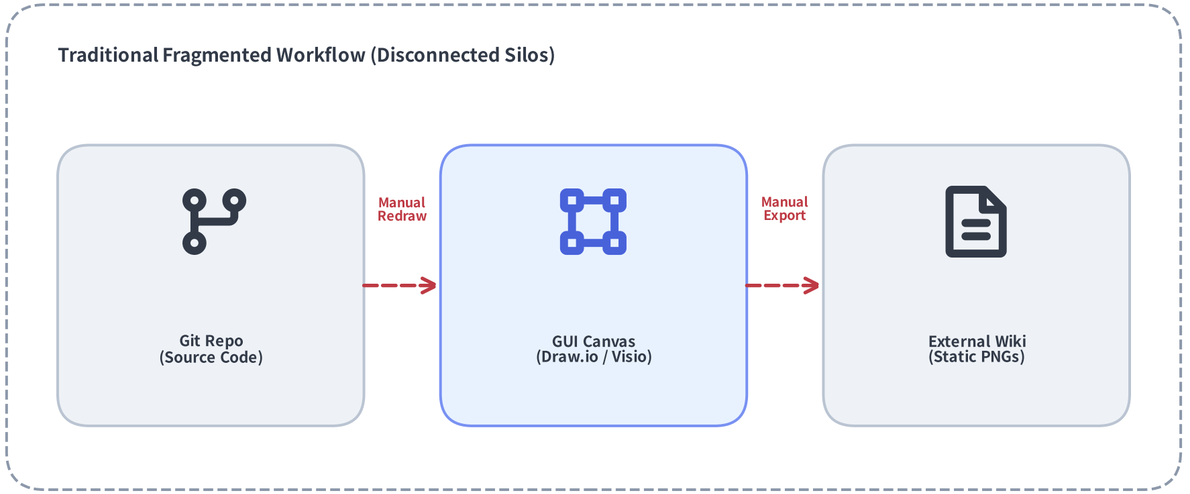
1.4 Single Source of Truth for Humans & AI Agents

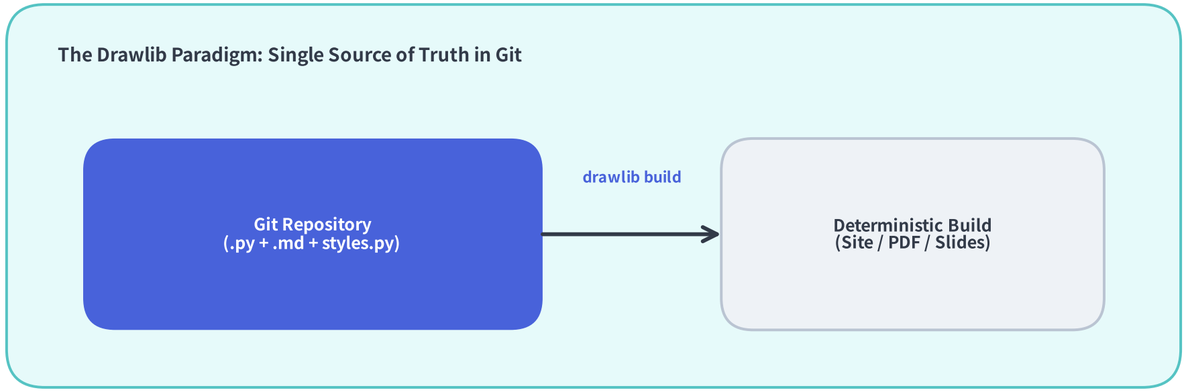
Pain Point 1: The Tool Fragmentation Problem

Why traditional technical documentation inevitably drifts out of sync with reality.

Three Disconnected Silos

1. **Source Code in Git (`src/`)**
   - Version-controlled, peer-reviewed in Pull Requests, and continuously tested in CI/CD.
2. **Diagrams in External GUI Tools (Draw.io, Visio, Figma)**
   - Drag-and-drop canvas state lives in proprietary cloud workspaces or opaque binary/XML blobs.
   - Impossible to review meaningful diffs in Git or refactor programmatically across 30 diagrams.
3. **Prose in External Wikis / Docs (Confluence, Word, Notion)**
   - Engineers manually export `.png` screenshots and paste them into external pages.





The Inevitable Result: Documentation Rot

- Updating a single microservice port requires **opening a GUI tool, re-aligning boxes, exporting a PNG, and re-uploading to a wiki**.

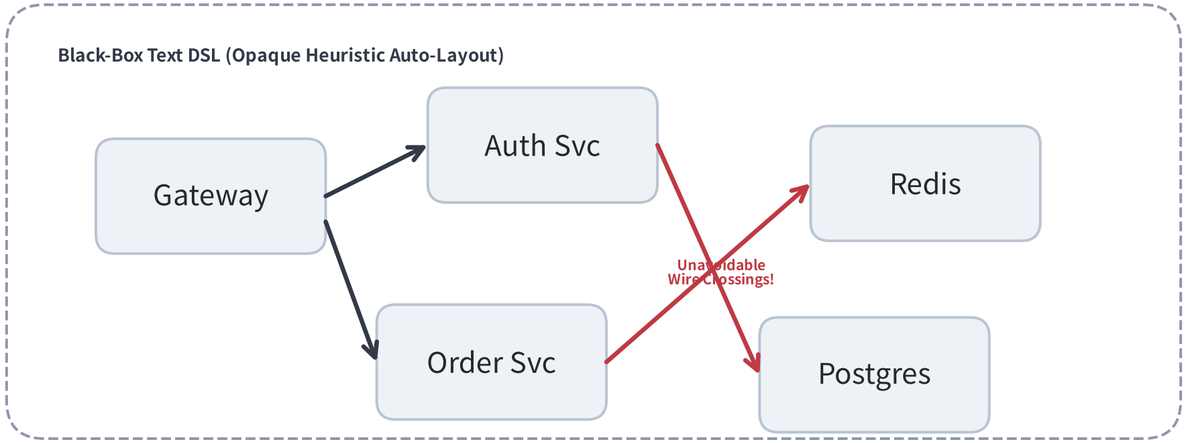
Pain Point 2: Text DSL Limitations (Mermaid / PlantUML / Graphviz)

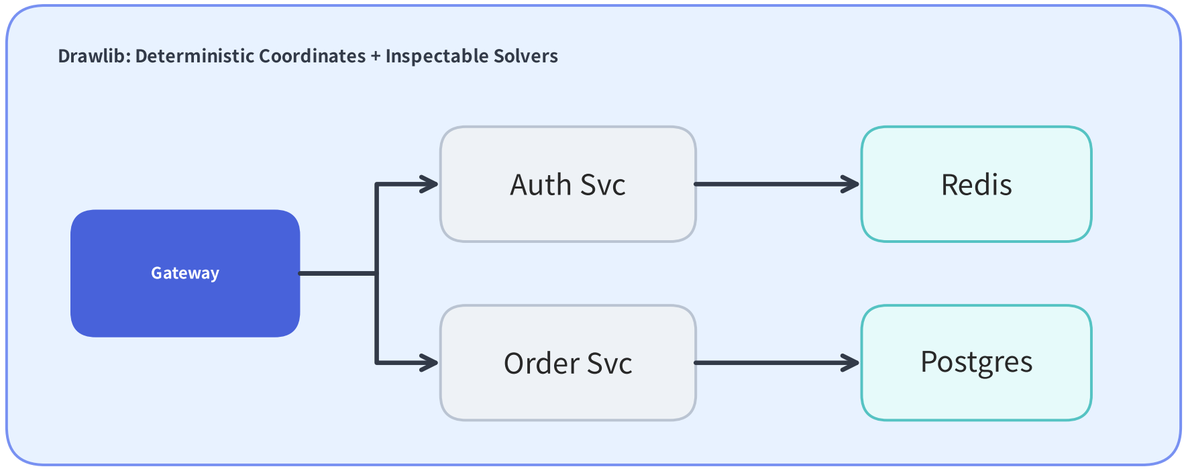
Why black-box heuristic layout DSLs hit a wall on real-world production architectures.

Great for 5 Nodes — Frustrating for 25 Nodes

Text-to-diagram DSLs (`Mermaid` , `PlantUML` , `DOT`) rely on rigid, opaque layout heuristics:

- **Unpredictable Layout Shifts**
  - Adding a single edge or renaming a label can cause the layout engine to reshuffle the entire diagram unpredictably.
- **No Fine-Grained Coordinate Control**
  - You cannot say *"align these 3 databases vertically at* `X=80` *"* or *"route this error path around the VPC border"*.
  - Engineers resort to hacks like invisible dummy nodes (`A ~~~ B`) and hidden links just to nudge boxes.
- **No Programming Constructs**
  - Static DSL syntax lacks variables, `for` loops, math expressions, conditional branches, or shared Python functions.
- **Limited Visual Expressiveness**





Pain Point 3: Raw SVG & Matplotlib Boilerplate

Why general-purpose plotting libraries and raw vector XML are too low-level for architecture diagrams.

The Low-Level Productivity Trap

When engineers outgrow text DSLs, they sometimes reach for raw `<svg>` templates or Python's `matplotlib.patches`:

- **Verbose Patch & Bezier Boilerplate**
  - Drawing a single rounded service box with an icon, centered two-line title, and orthogonal boundary-clipped arrow in raw `matplotlib` requires **40–60 lines** of `FancyBboxPatch`, `ConnectionPatch`, `Path`, and manual bounding-box math.
- **Inverted or Confusing Coordinate Defaults**
  - Raw SVG uses top-left origins with manual `<text dy="...">` baseline offsets and no automatic word/boundary clipping.
- **No Domain Semantics**
  - Plotting libraries know about scatter plots and histograms, not **VPC NodeGroups, UML Class relationships, Sequence lifelines, ER Crow's foot notation, or Phosphor/GCP icons**.



Landscape Comparison Matrix

How Drawlib bridges the gap between GUI canvases, text DSLs, and low-level plotting libraries.

Evaluation Criteria	GUI Tools (Visio / Draw.io / Figma)	Text DSLs (Mermaid / PlantUML)	Raw Plotting (Matplotlib / Raw SVG)	**Drawlib (Illustration as Code)**
Git Diffability & PR Review	✗ Opaque XML / Binary	✓ Clean Text Diffs	△ Verbose Code Diffs	✓ **Concise Python + Markdown**
Layout Control	✓ Manual Pixel Dragging	✗ Rigid Black-Box Heuristics	△ Manual Math Only	✓ **Exact Coords + 5 Solvers**
Reusable Python Logic	✗ None (Copy-Paste)	✗ Static DSL Syntax	✓ Full Python	✓ **Functions, Loops & styles.py**
Unified Charts + Diagrams	✗ Diagrams Only	△ Limited / Separate	✗ Plots Only (No VPC/UML)	✓ **7 Diagrams + 7 Charts + 23 Shape**
Multi-Frame Animation	✗ Static Exports	✗ Static Exports	△ Complex FuncAnimation	✓ **Native APNG/WebP & Slide Contr**
AI Agent Self-Healing	✗ Requires GUI Mouse	△ Cannot Fix Wire Crossings	✗ Too Verbose for Context	✓ **Rules CLI + Grid (-g) Loop**

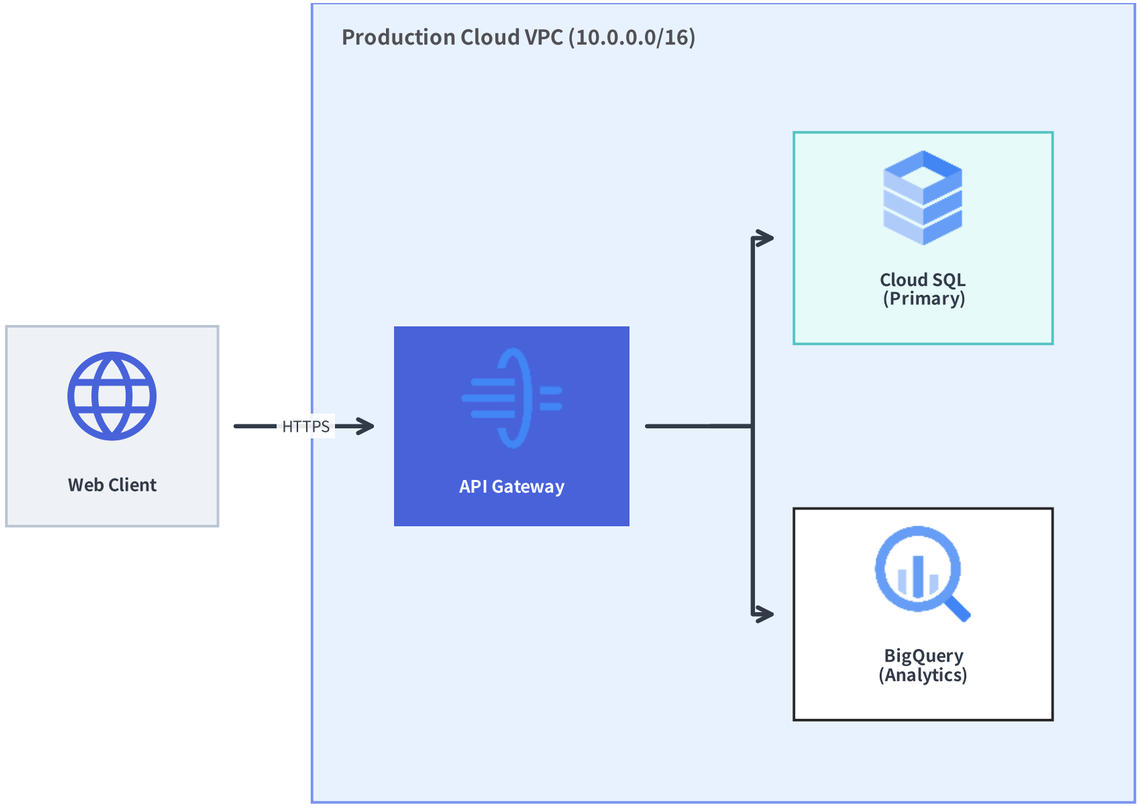
The Drawlib Solution: Declarative Python Architecture

Write clean, type-checked Python on the left—get publication-grade vector diagrams on the right.

15 Lines of Declarative Python

```python
from drawlib.diagrams.architecture import (
    ArchitectureDiagram, GcpIcon, Node, NodeGroup, Phospho
)
from drawlib.styles import Styles

d = ArchitectureDiagram(
    node_style=Styles.Primary,
    node_text_style=Styles.DarkBold,
    edge_style=Styles.DarkBold,
    edge_text_style=Styles.Dark,
    node_card_style=Styles.Neutral,
)
vpc = d.add(NodeGroup("Production Cloud VPC", padding=7),
client = d.add(Node("Web Client", width=18, height=16, ico
gw = vpc.add(
```



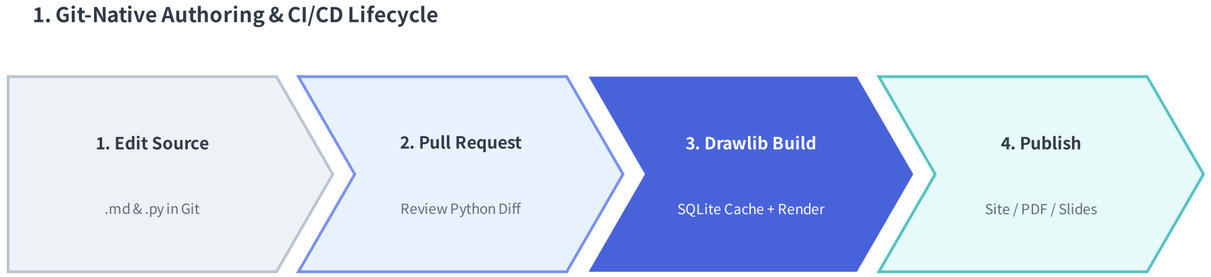
- **Auto-Bounding Containers**: `NodeGroup` wraps child nodes automatically.
- **Smart Orthogonal Edges**: `.connect()` and `.fork()` clip cleanly at node boundaries.

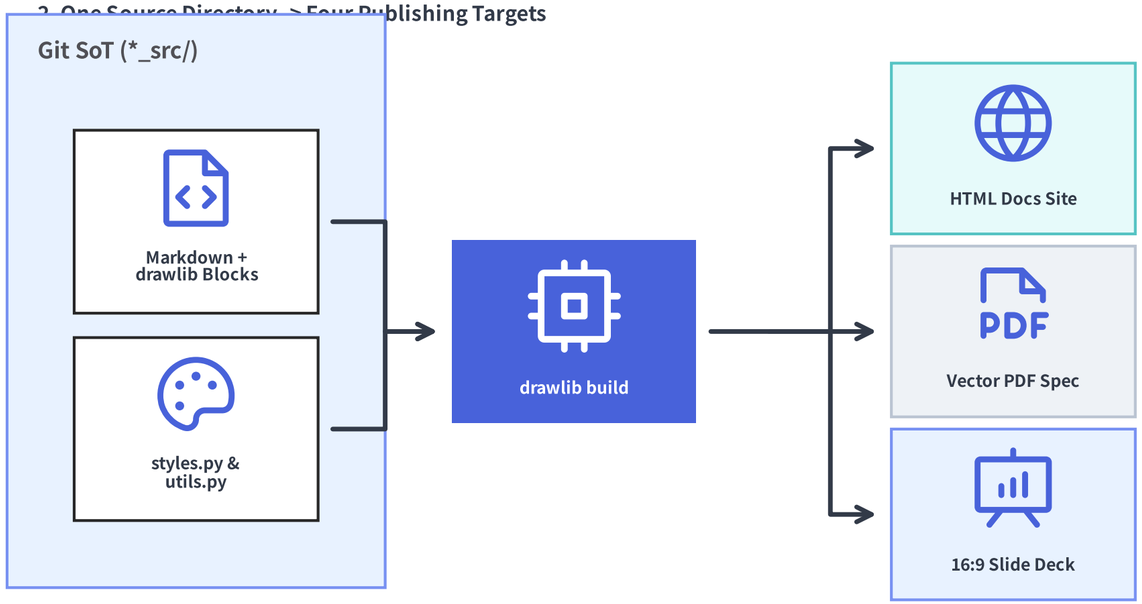
Single Source of Truth (SoT) in Git

Code, diagrams, and prose evolve together in atomic Pull Requests and compile deterministically.

Software Engineering Rigor for Documentation

1. **Atomic Pull Requests**
   - When an engineer adds a new gRPC service or database table in `src/`, they update the `drawlib` diagram block in `docs_src/` in the **exact same commit**.
2. **Code Review & Static Verification**
   - Reviewers inspect clean Python diffs instead of binary `.png` blobs.
   - CI validates every diagram block, checks internal hyperlinks (`drawlib check`), and verifies zero syntax or type errors.
3. **Deterministic Multi-Target Compilation**
   - One command (`drawlib build`) compiles `.md` + `.py` sources into HTML documentation sites, GitHub-flavored Markdown, 1920×1080 slides, and vector PDFs.
4. **Content-Addressable SQLite Caching**





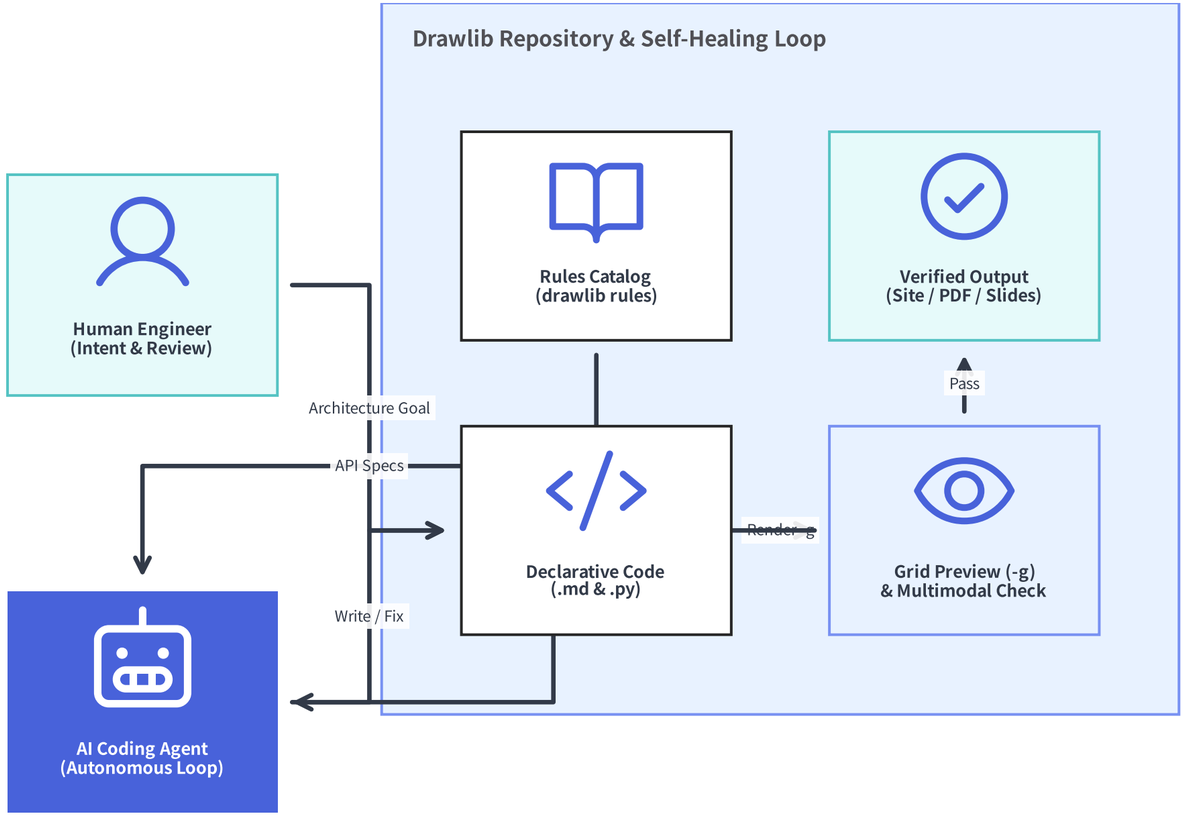
Designed for Both Humans and AI Coding Agents

Why pure Python + multimodal visual inspection unlocks autonomous documentation engineering.

Python is the Native Lingua Franca of LLMs

- **Zero Proprietary Syntax Friction**
  - LLMs are pre-trained on billions of lines of Python—they naturally understand functions, loops, arithmetic coordinate layout, and Pydantic type hints.
- **Self-Contained Rules Catalog (`drawlib rules`)**
  - Agents query `uv run drawlib rules show <topic>` on demand to inspect exact function signatures and style guidelines without bloating context windows.
- **Autonomous Visual Self-Healing Loop**

1. **Author**: Write or update `drawlib` Python code in `*_src/`.
2. **Render with Grid**: Export a preview with `-g` (`drawlib show ... -g -o preview.png`).
3. **Inspect Multimodally**: View the rendered image to check alignment, margins, and contrast.
4. **Self-Repair**: Adjust coordinates or styles until the visual output is publication-grade.



CHAPTER

02

Part 2 of 7

Core Engine & Primitives

Cartesian Geometry, Shapes, Connectors, Typography, Icons & Design Tokens

2.1 Cartesian Coordinate Space (0,0 at Bottom-Left) & Lifecycle

2.2 23 Geometric Primitives & Specialized Block Arrows

2.3 Straight, Orthogonal, Curved & Bezier Connectors

2.4 Universal Typography, Vector Icons & 50%+ Neutral Design Tokens

Cartesian Canvas & Coordinate System (`drawlib.canvas`)

An intuitive, resolution-independent 2D mathematical coordinate space with `(0, 0)` at the bottom-left.

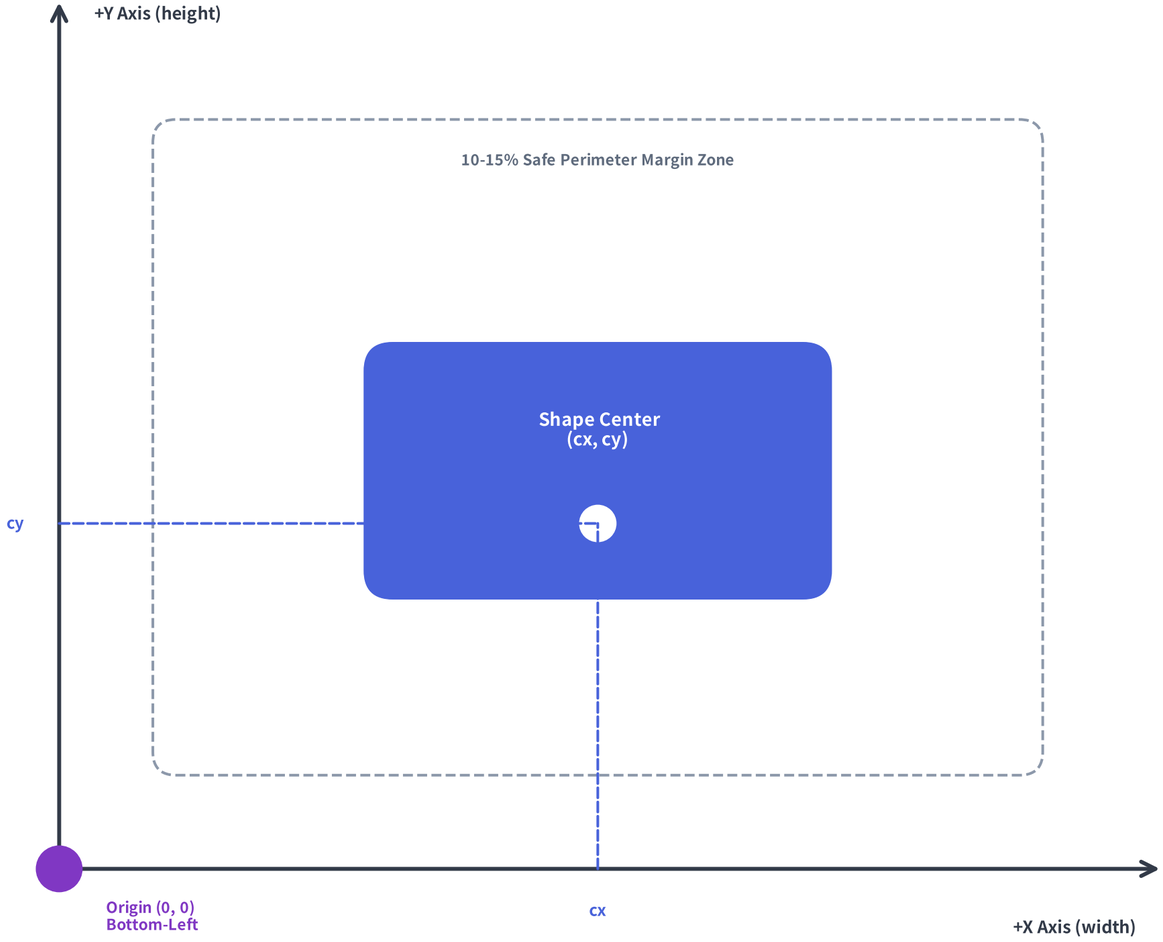
1. Canvas Lifecycle

```
from drawlib.canvas import clear, save, setup
from drawlib.shapes import rectangle
from drawlib.styles import Styles

clear()
setup(width=100, height=70)
```

2. Core Spatial Rules

- **Bottom-Left Origin** `(0, 0)`: `X` increases rightward (`0 → width`); `Y` increases upward (`0 → height`).
- **Center-Anchored Primitives** `(cx, cy)`: Shapes (`rectangle` , `circle` , `cylinder` , `rhombus` , etc.) and icons are anchored at their geometric center `(cx, cy)` , making horizontal (`cy` equal) and vertical (`cx` equal) alignment trivial.
- **Z-Order Layering Discipline**: Draw background containers first, entity shapes second, connector lines third, and foreground badges/text last.
- **10–15% Safe Perimeter Margin**: Keep outer margins clear so labels and arrowheads never clip.



Visual Debugging with Coordinate Grid (`grid=True` / `-g`)

Inspect exact coordinates and verify margins without touching production code.

Two Ways to Enable the Coordinate Grid

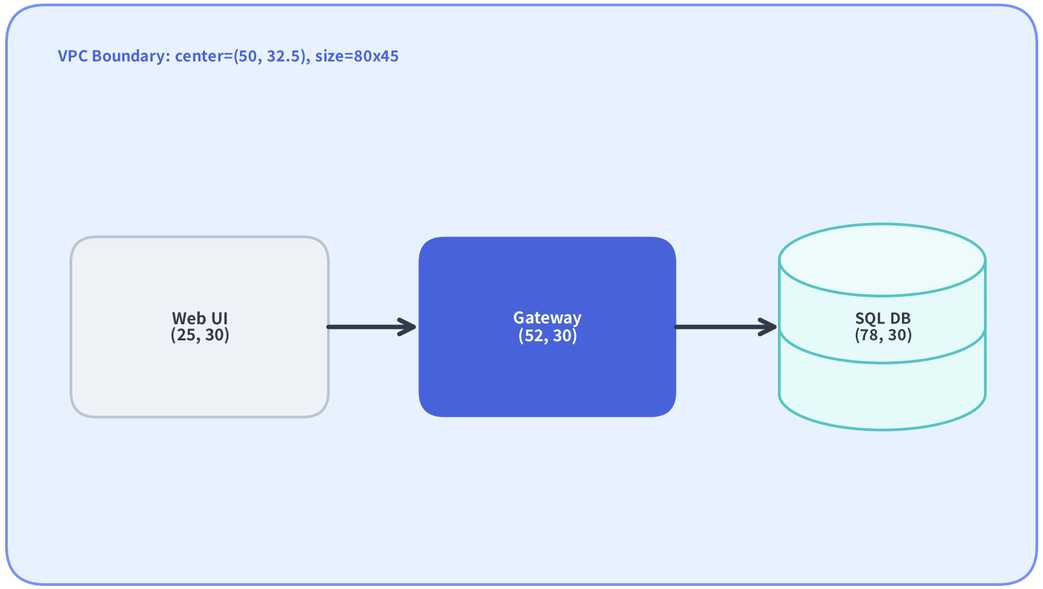
1. **CLI Flag (`-g` / `--grid`) — Recommended**
   - Render any Python script or embedded Markdown diagram with a temporary coordinate ruler overlay:

```
# Preview a standalone script with grid
uv run drawlib show arch.py -g -o preview.png

# Preview an embedded Markdown diagram with grid
uv run drawlib show docs_src/index.md arch.png -g -o pr
```

2. **Programmatic `setup(..., grid=True)`**
   - Turn on the grid directly inside `setup()` during interactive layout design:

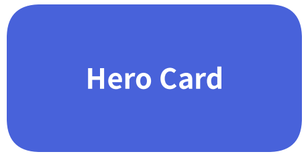
```
setup(width=100, height=65, grid=True, grid_xpitch=10,
```



Why the Grid Matters

40+ Geometric Primitives (`drawlib.shapes`)

A rich vocabulary of vector shapes—all supporting `style` (including rotation `style.angle`), `text`, and `text_style`.



rectangle(shape_r=2.5)



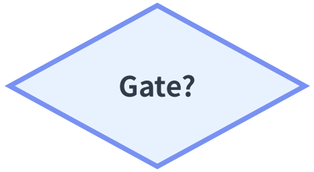
circle()



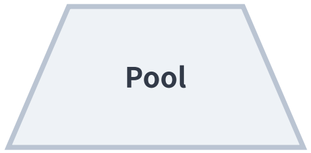
ellipse()



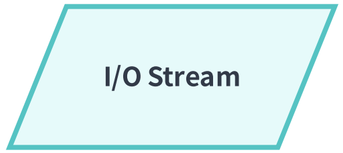
cylinder(disks=3)



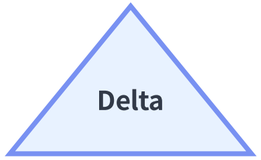
rhombus()



trapezoid()



parallelogram()



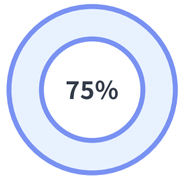
triangle()



regularpolygon(6)



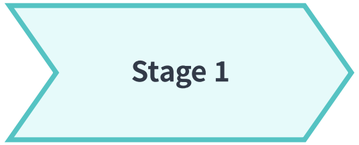
star(5)



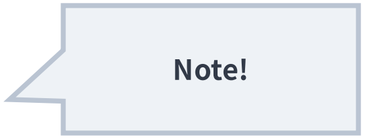
circle(hole_ratio=0.6)



wedge()



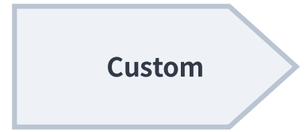
chevron()



bubblespeech()



face(variant="smile")



polygon(xys)

Block Arrows & Callouts (`arrow`, `arrow_l`, `arrow_u`, `arrow_arc`)

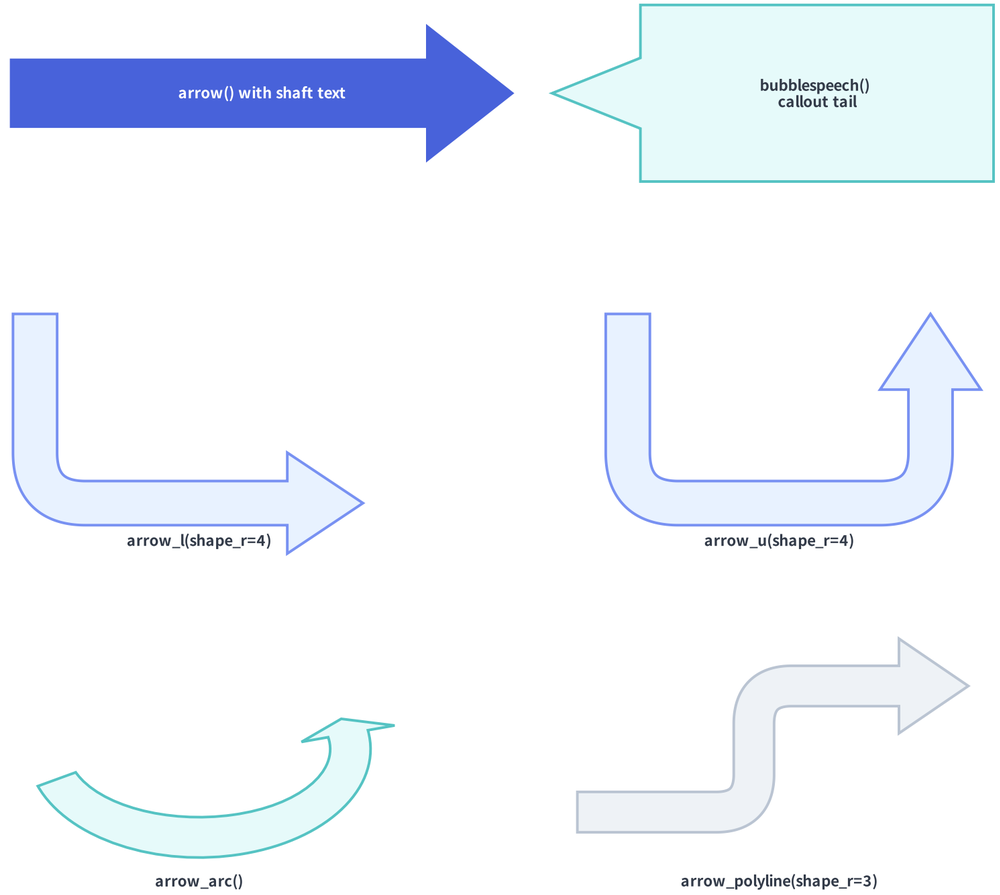
2D filled planar block arrows and speech callouts for prominent phase transitions and annotations.

5 Directed Block Arrows + Speech Callouts

```
from drawlib.shapes import (
    arrow, arrow_arc, arrow_l, arrow_polyline, arrow_u, b
)
from drawlib.styles import Styles

# 1. Straight block arrow (supports embedded shaft text)
arrow((10, 66), (48, 66), tail_width=4, head_width=9,
      head_length=6, style=Styles.PrimaryFlat,
      text="Data Transfer", text_style=Styles.WhiteBold)

# 2. L-shaped & U-turn block arrows (with rounded elbow
arrow_l((25, 42), width=26, height=18, tail_width=3.5,
        head_width=8, head_length=6, style=Styles.Seconda
arrow_u((68, 42), width=24, height=20, tail_width=3.5,
        head_width=8, head_length=6, style=Styles.BlueNeu
```



- **Note**: `arrow()` supports embedded `text`; for `arrow_l`, `arrow_u`, `arrow_arc`, and `arrow_polyline`, place external `text()` labels alongside the shaft.

Lines, Polylines & Bezier Curves (`drawlib.lines`)

Precision 1D vector connectors with straight, orthogonal, filleted, arc, and polynomial Bezier routing.

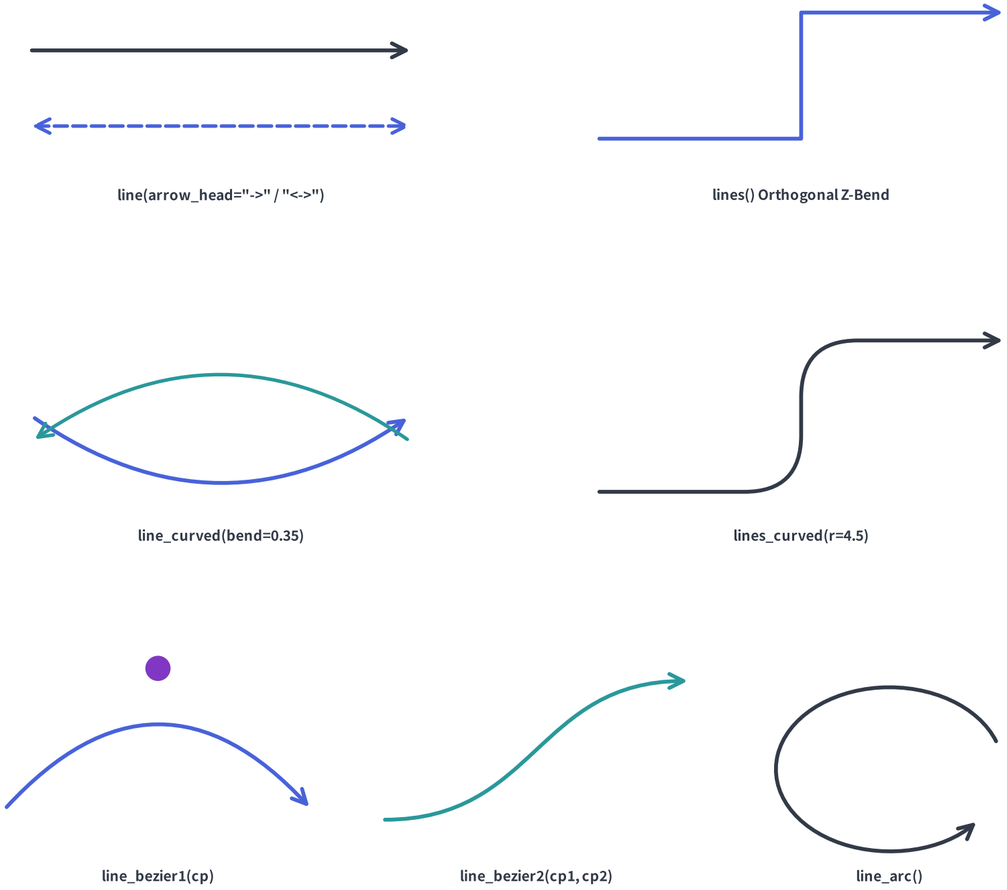
7 Connector Primitives & Arrowheads

```
from drawlib.lines import (
    line, line_arc, line_bezier1, line_bezier2,
    line_curved, lines, lines_curved,
)
from drawlib.styles import Styles

# 1. Straight line & orthogonal polyline
line((10, 70), (45, 70), arrow_head="→", style=Styles.Da
lines([(55, 65), (72, 65), (72, 75), (90, 75)],
      arrow_head="→", style=Styles.PrimaryBold)

# 2. Curved arc spline (bend) & filleted polyline (r)
line_curved((10, 44), (45, 44), bend=0.35,
            arrow_head="⟷", style=Styles.PrimaryBold)
```

- **Arrowhead Options**: `""` (none), `"→"` (forward), `"←"` (reverse), `"⟷"` (bidirectional).
- **Filled vs. Stick Heads**: Customize via `style.patch(line_arrow_head_fill=True, line_arrow_head_scale=22)`.



Typography, Alignments & Embedded Fonts (`drawlib.text`, `drawlib.fonts`)

Universal multilingual fonts, explicit anchor alignments, and automa

1. Horizontal & Vertical Text Primitives

```python
from drawlib.fonts import Font, FontFile, FontMonoSpace,
from drawlib.styles import Styles
from drawlib.text import text, text_vertical

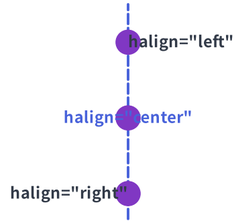
# Anchor alignment via halign & valign
text((20, 60), "Left Anchor",
     style=Styles.DarkBold.patch(halign="left"))
text((80, 60), "Right Anchor",
     style=Styles.DarkBold.patch(halign="right"))

# Built-in font families & custom TTF/OTF files
text((50, 40), "Roboto Bold",
     style=Styles.PrimaryBold.patch(text_font=FontRoboto
```

2. Automatic SVG `@font-face` Bundling

- When compiling slides or docs with `.svg` diagrams, Drawlib automatically copies required `.ttf` / `.otf` font files into `_assets/fonts/` and injects `@font-face` rules so browsers and PDF exports render identically anywhere.

1. Explicit Anchor Alignment (halign)



FONTS

Font.SANSSERIF_BOLD — Universal Latin + CJK

FontRoboto.ROBOTO_REGULAR — Clean Western UI

FontMonoSpace.ROBOTO_MONO_REGULAR — 10.0.0.1:443

FontSourceCode.SOURCECODEPRO — def draw():

Line Spacing & Rotation (angle=0..360, text_line_spacing)

Standardized Icons & Image Processing (`drawlib.icons`, `drawlib.images`)

1,500+ Phosphor vect[illegible] 250[illegible]fficial GCP cloud icons, and

1. Dual Icon Engine (`phosphor` & `gcp`)

```
from drawlib.icons import gcp, phosphor
from drawlib.images import Dimage, image
from drawlib.styles import Styles

# Vector Phosphor icons (5 weights: thin, light, regular
phosphor.cpu((20, 65), width=12, style=Styles.PrimaryBold
phosphor.database((45, 65), width=12, style=Styles.Primar
```

2. Bitmap Manipulation with `Dimage`

- Load external `.png` / `.jpg` / `.webp` assets or capture sub-diagrams in memory via `get_dimage_from_code()`.
- Chain non-destructive effects: `.grayscale()`, `.sepia()`, `.invert()`, `.brightness()`, `.blur()`, `.mosaic()`, `.trim()`, and `.make_transparent()`.

```
image((30, 15), width=16, image="_assets/linux.png")
image((65, 15), width=16, image=Dimage("_assets/linux.png
```

1. Phosphor Vector Icons (1,500+ Icons × 5 Weights)



2. Official Google Cloud Architecture Icons (250+ Icons)



3. Dimage Raster Image Processing (_assets/linux.png)



Design Token Discipline & The 50%+ Neutral Rule (`drawlib.styles`)

How systematic PascalCase tokens and neutral grounding prevent "rainbow chaos" in technical diagrams.

1. Systematic `Styles.<Color><Variant>` Matrix

```
from drawlib.styles import Colors, Styles, get_intermedia

# Always import PascalCase Styles & Colors!
hero = Styles.PrimaryFlat                # Saturated focal
card = Styles.PrimaryNeutral             # Soft tinted-neu
base = Styles.Neutral                    # Calm slate/whit
zone = Styles.MutedDashed                # Container bound

# Immutable overrides via .patch()
```

2. The 50%+ Neutral Baseline Rule

- **Avoid Rainbow Chaos**: Never fill every node with competing saturated colors (`PrimaryFlat` , `WarningFlat` , `DangerFlat` , `PurpleFlat`).
- **Ground 50%+ in Calm Neutrals**: Use `Styles.Neutral` , `Styles.PrimaryNeutral` , `Styles.SecondaryNeutral` , `Styles.BlueNeutral` for standard components, reserving `Styles.PrimaryFlat` / `Styles.AccentFlat` (`text_style=Styles.WhiteBold`) strictly for **1–2 focal hero nodes**.

1. Anti-Pattern (Rainbow Chaos) vs. 50%+ Neutral Discipline



2. Orthogonal Style Variants (Styles.<Role><Variant>)



3. Color Interpolation: get_intermediate_colors(Primary, Secondary)



CHAPTER

03

Part 3 of 7

SmartArts & Quantitative Charts

Structured Business Visuals & Vector Data Plotting on a Single Canvas

3.1 10 Automated SmartArt Components with Unified Lifecycle

3.2 Process Pipelines (ChevronProcess, Cycle, BoxList)

3.3 Hierarchies, Matrices & Tables (TreeNode, MindMap, Grid, Pyramid)

3.4 7 Statistical & Project Charts (Bar, Line, Area, Pie, Radar, Scatter, Gan

SmartArts Architecture & Unified Lifecycle

10 High-Level Structured Components (`drawlib.smartarts`)

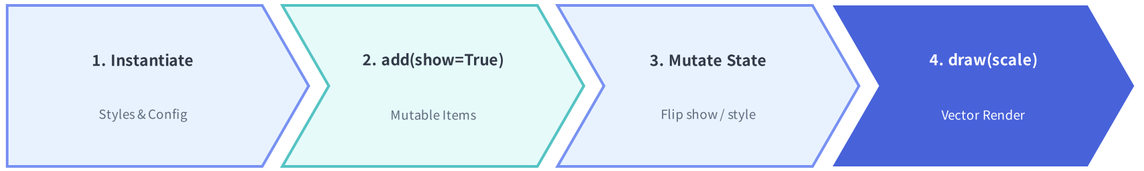
Instead of manually calculating polygon vertices and text offsets, `drawlib.smartarts` automates layout geometry for **10 essential diagram patterns**:

Anchor Convention	SmartArt Components	Flow Direction
Top-Left `(x, y)`	`Table`, `TreeNode`, `BulletPoints`, `SourceCode`	Rightward (`+x`) & Downward (`-y`)
Bottom-Left `(x, y)`	`ChevronProcess`, `GridLayout`, `Pyramid`	Rightward (`+x`) & Upward (`+y`)
Center `(x, y)`	`MindMapNode`, `Cycle` *(default)*	Radial outward from center
Directional `(x, y)`	`BoxList` (`"left"`, `"right"`, `"bottom"`, `"top"`)	Along chosen axis

Unified 4-Phase Component Lifecycle

1. **Instantiate**: Configure default `style` and `text_style` tokens.
2. **Register (`add(..., show=True)`)**: Returns a mutable item reference.

Unified 4-Phase Lifecycle (ChevronProcess)



Key Architectural Benefits

- Zero manual vertex math
- Layout-preserving visibility:
  - item.show = False hides item
  - Siblings never shift or jump
- Lazy style & text evaluation
- Proportional scale=1.0 transform

Declarative Python API (SourceCode)

```python
from drawlib.smartarts import ChevronProcess
from drawlib.styles import Styles

proc = ChevronProcess(
    style=Styles.Neutral,
    text_style=Styles.DarkBold,
    description_style=Styles.Muted,
    flat_left_end=True,
)
s1 = proc.add("Build", description="CI")
s2 = proc.add("Deploy", style=Styles.PrimaryFlat,
              text_style=Styles.WhiteBold)

# Hide step 2 without shifting step 1:
s2.show = False
proc.draw(xy=(10, 20), width=80, scale=1.0)
```

Pipelines & Feedback Loops (`ChevronProcess` & `Cycle`)

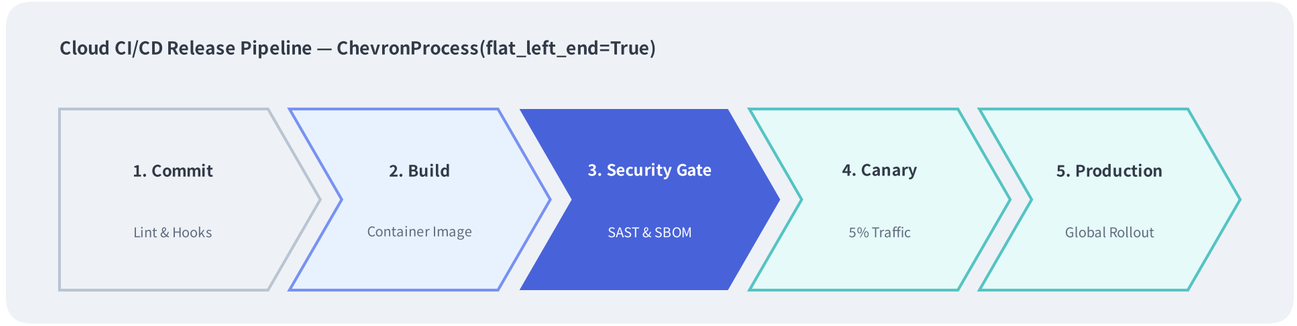
Linear Stages vs. Continuous Iterations

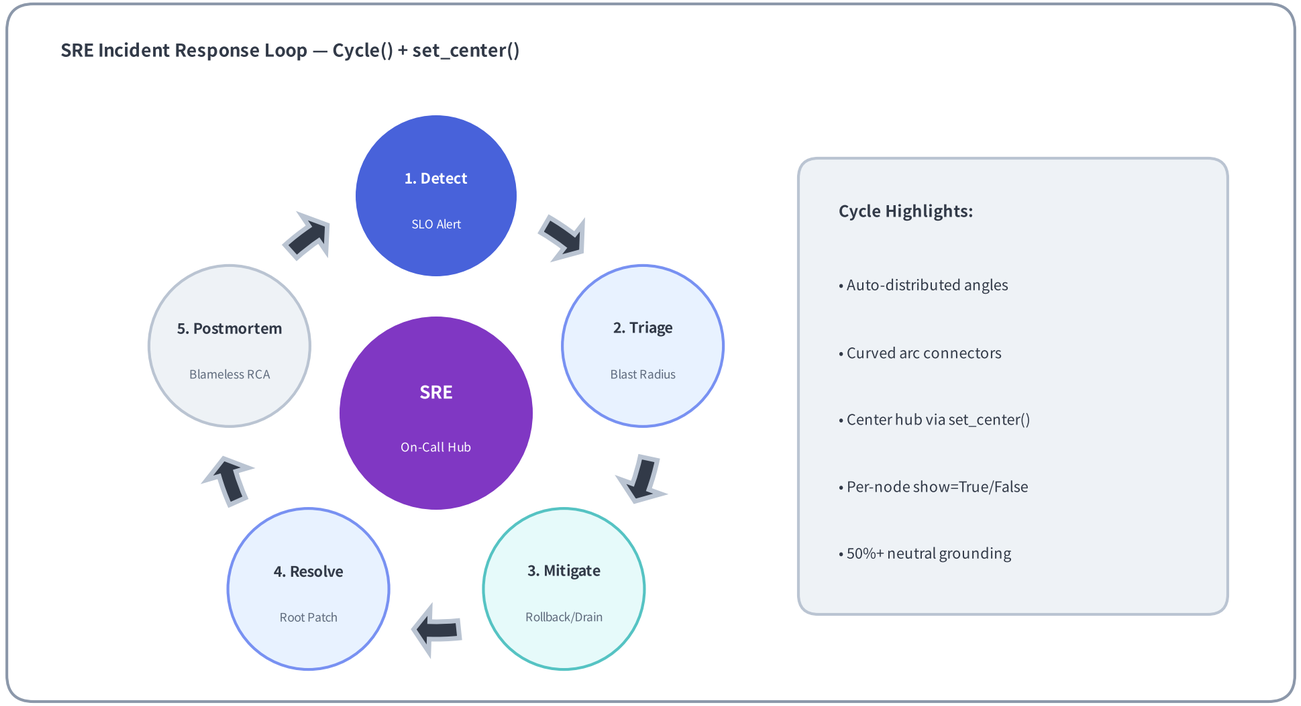
1. `ChevronProcess` — Phased Pipelines

- **Interlocking Geometry**: Automatically computes chevron tip angles (`corner_angle=60.0`) and uniform item widths across total `width` .
- **Clean Entry Edge**: `flat_left_end=True` renders the first stage as a flat-backed pentagon while subsequent stages interlock seamlessly.
- **Dual-Line Labels**: Each stage supports a primary `text` heading and a secondary `description` subtitle with independent style overrides.

2. `Cycle` — Circular & Radial Workflows

- **Trigonometric Orbit**: Distributes `N` nodes (`"circle"` , `"rectangle"` , or `"none"`) evenly around orbit `radius` starting from `start_angle=90.0` (12 o'clock).
- **Curved Arc Connectors**: Connects steps with circular arc arrows (`arrow_type="arc"`) and configurable `arrow_color_mode` (`"monochrome"` , `"match_source"` , `"match_target"`).
- **Central Command Hub**: Calling `cycle.set_center(...)` places a central hub node inside the orbit for radial governance or SRE command loops.





Stacks, Matrices & Tiered Layers (`BoxList`, `GridLayout`, `Pyramid`)

Structured Card & Layer Components

1. `BoxList` — Directional Card Sequences

- Aligns sequential cards along `"left"`, `"right"`, `"bottom"`, or `"top"` orientations.
- Ideal for status badges, KPI ribbons, and linear step stacks.

2. `GridLayout` — Multi-Span Architecture Matrices

- **Bottom-Left Indexed**: `(col=0, row=0)` starts at the bottom-left cell; rows increment upward (`+y`) and columns increment rightward (`+x`).
- **Cell Spanning**: Any item can span multiple columns and rows (`position=(col, row), width=w, height=h`) to model full-width ingress/platform layers alongside multi-column microservice cells.
- **Outer Container**: Optional `outer_style` (including `shape_r`) wraps the entire matrix in a dashed or solid boundary.

3. `Pyramid` — Proportional Tiered Hierarchies

- Slices a triangular pyramid into an apex triangle and progressive trapezoidal tiers.

Multi-Tier Platform Matrix (GridLayout)



Software Testing Pyramid (Pyramid)

Hierarchies & Brainstorming (`TreeNode` & `MindMapNode`)

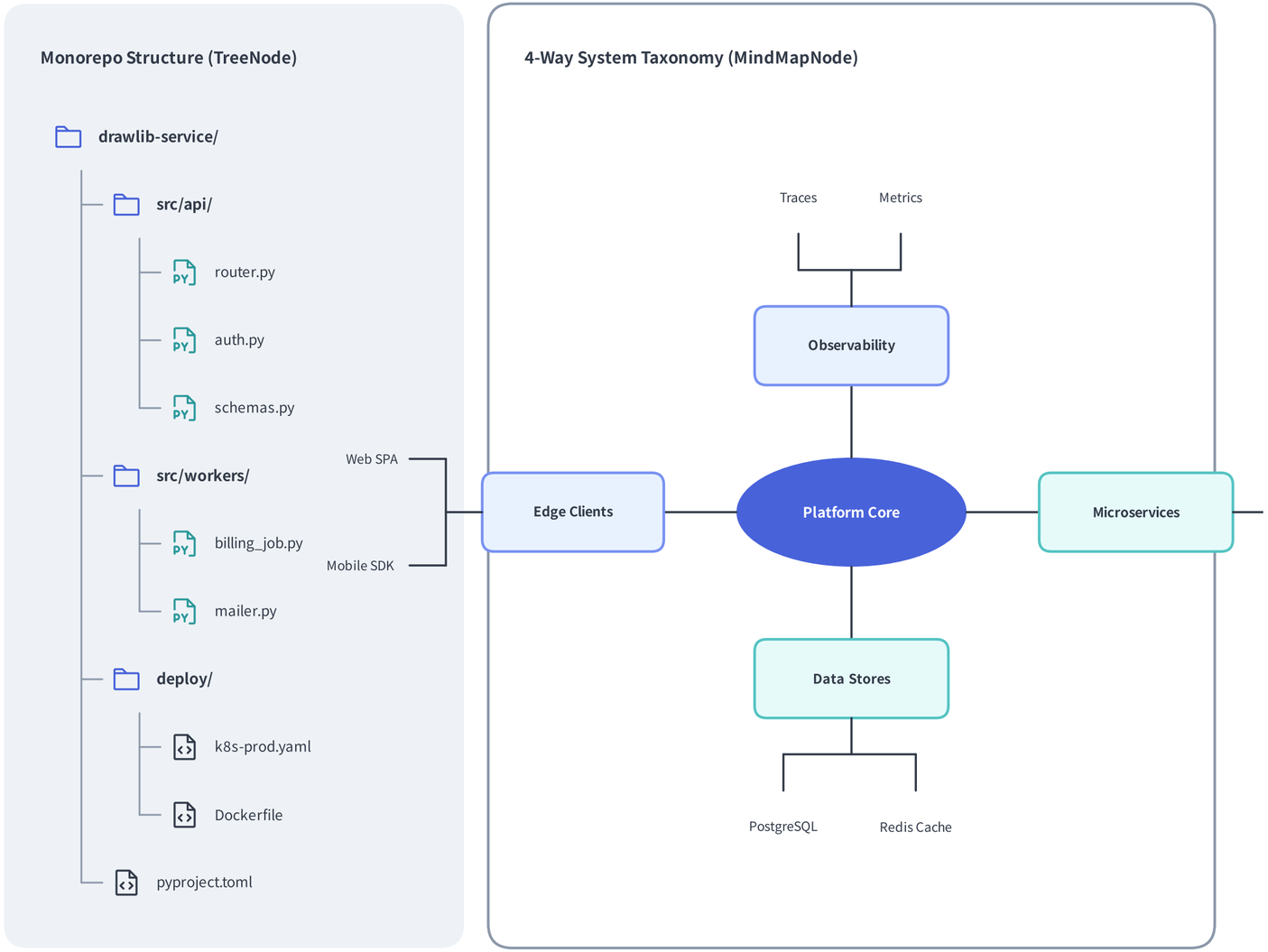
Two Complementary Tree Engines

1. `TreeNode` — Directory & Package Hierarchies

- **Top-Left Anchored**: Steps downward row-by-row with crisp orthogonal `├` and `└` connector lines.
- **Cascading Root Styles**: Set `text_style`, `line_style`, and margins once on the root `TreeNode`; all descendants inherit them automatically.
- **Custom Icon Attachments**: Use `TreeNode.register_drawing_item(...)` and `.set_drawing_item(name)` to prepend or append Phosphor icons (`phosphor.folder`, `phosphor.file_py`, `phosphor.file_code`).

2. `MindMapNode` — Two-Pass Radial & Org Trees

- **Pass 1 (Bottom-Up Measurement)**: Recursively measures exact subtree bounding heights and widths so branches never collide.
- **Pass 2 (Orthogonal Routing)**: Routes right-angled junction lines from the center root node across all 4 directions (`branch="left"`, `"right"`, `"top"`, `"bottom"`).
- **Flexible Node Shapes**: Mix `"ellipse"`, `"rectangle"`, and borderless `"none"` leaf labels.



Data Matrices & Embedded Code (`Table` & `SourceCode`)

Tabular Precision & Vector Syntax Highlighting

1. `Table` — Fine-Grained Matrix Styling

- **Flexible Sizing**: `draw(xy, width, height, data)` distributes columns evenly, while `draw_flexible(xy, column_widths, row_heights, data)` gives exact per-column and per-row dimensions.
- **Zebra & Cell Highlighting**:
  - `set_style_cell_evenodd(even_color, even_text_style, odd_color, odd_text_style)` applies alternating row stripes.
  - `set_style_cell(background_color, text_style, rows=[...], columns=[...])` highlights specific SLA breaches or target cells.
- **Selective Borders**: `set_style_border(top=..., top2=..., bottom=..., between_rows=...)` controls perimeter and inner rules independently.

2. `SourceCode` — Vector Code Containers

- Renders Pygments-tokenized code directly onto the canvas as crisp vector glyphs and rounded containers (`show_linenum=True`).
- Built-in themes (`"dark"` , `"monokai"` , `"default"` , `"google"` , `"monochrome"`) and CJK-aware monospace fonts (`font_lang="en" | "ja"`).

Production Microservice SLA Matrix — Table.draw_flexible()

Service Name	Protocol	P99 Latency	Availability SLA	Tier
Edge Auth Gateway	gRPC / HTTP3	12 ms	99.99 %	Tier 0
Ledger Core Engine	gRPC (mTLS)	6 ms	99.999 %	Tier 0
Event Dispatcher	Kafka Stream	45 ms	99.95 %	Tier 1
Batch Analytics	REST / Parquet	380 ms	99.50 %	Tier 2

Embedded Vector Code Container — SourceCode.draw(show_linenum=True)

```yaml
# Envoy Rate-Limit & Circuit-Breaker Filter Config
apiVersion: networking.istio.io/v1alpha3
kind: EnvoyFilter
metadata:
  name: ledger-core-circuit-breaker
  namespace: prod-mesh
spec:
  workloadSelector:
    labels:
      app: ledger-core
      tier: tier-0
  configPatches:
    - applyTo: CLUSTER
      patch:
        operation: MERGE
        value:
          circuit_breakers:
```

Quantitative Charts Overview (`drawlib.charts`)

Why Built-In Vector Charts Beat External Plotting Tools

Traditional workflows force you to plot charts in Matplotlib/Excel, export PNGs, and paste them next to architecture diagrams—resulting in mismatched fonts, blurry raster scaling, and broken visual alignment.

4 Pillars of `drawlib.charts`

1. **First-Class Canvas Coexistence**: Charts render directly onto the Drawlib canvas at any `(x, y)` coordinate. You can draw connector arrows directly from an architecture microservice node to a live telemetry `BarChart` or `PieChart`.
2. **Deterministic Bounding Layout**: Explicit `width`, `height`, and `radius` parameters guarantee predictable margins without layout drift.
3. **Nice Numbers & Logarithmic Axes**: `Axis(scale="linear" | "log")` automatically computes human-readable tick steps ($1, 2, 5 \times 10^k$) or base-10 powers (10^k) with custom formatters (`format="{:.1f}M"` , `unit="ms"`).
4. **Animation-Ready (`show` & `draw_ratio`)**: Toggle `series.show = False` without rescaling axes, or animate `draw_ratio` (`0.0 → 1.0`) along `"bottom_to_top"` or `"left_to_right"` .



Categorical & Trend Charts (`BarChart`, `LineChart`, `AreaChart`)

Cartesian Data Visualization

1. `BarChart` — Grouped, Stacked & Log Scales

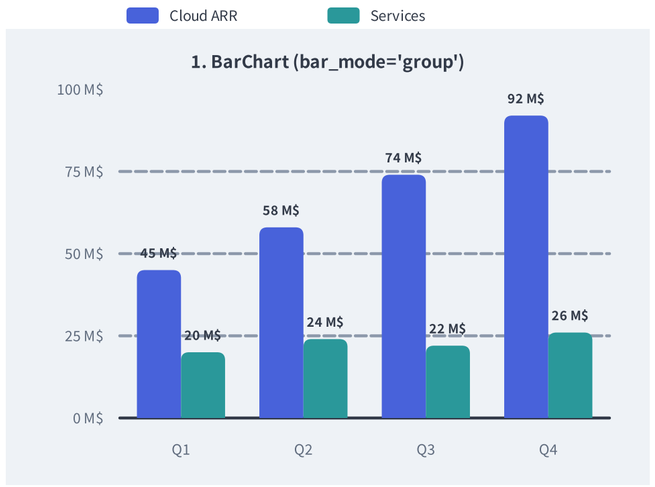
- Supports `orientation="vertical" | "horizontal"` and `bar_mode="group" | "stack"`.
- Rounded bar tip corners (`bar_r=0.8`), direct value labels (`value_format="{:.0f}M"`), and linear or logarithmic (`scale="log"`) axes.

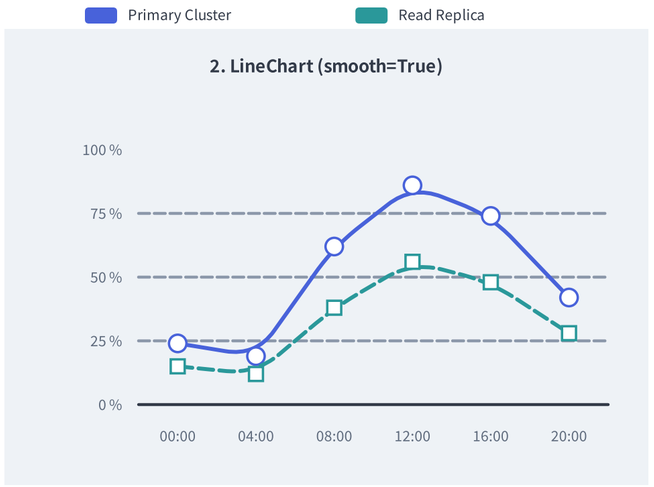
2. `LineChart` — Linear Polylines & Smooth Splines

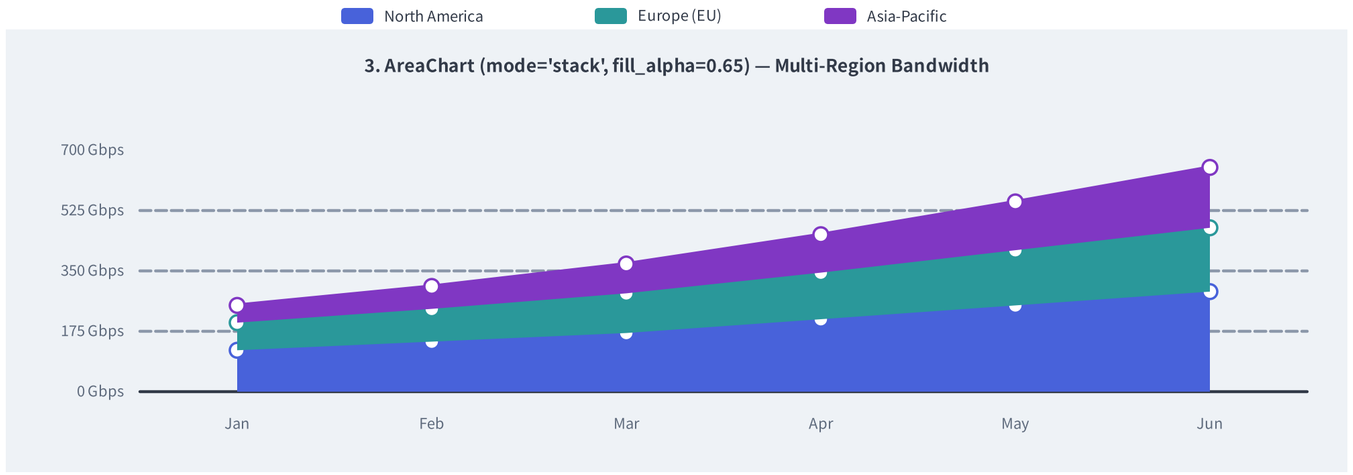
- Set `smooth=True` for cubic-like spline interpolation or `smooth=False` for exact vertex segments.
- Per-series `line_style` (`"solid"`, `"dashed"`, `"dotted"`), `line_width`, and `point_shape` (`"circle"`, `"square"`, `"none"`).

3. `AreaChart` — Cumulative & Overlapping Streams

- `mode="stack"` layers series cumulatively to show total volume breakdown; `mode="overlap"` renders semi-transparent overlapping polygons (`fill_alpha=0.35`).
- Shared fluent helpers across all three: `configure_x_axis(...)`, `configure_y_axis(...)`, and standalone `draw_legend(xy, ...)`.







Radial & Relational Charts (`PieChart`, `RadarChart`, `ScatterChart`)

Proportions, Trade-Offs & Correlations

1. `PieChart` — Solid Pies & KPI Donuts

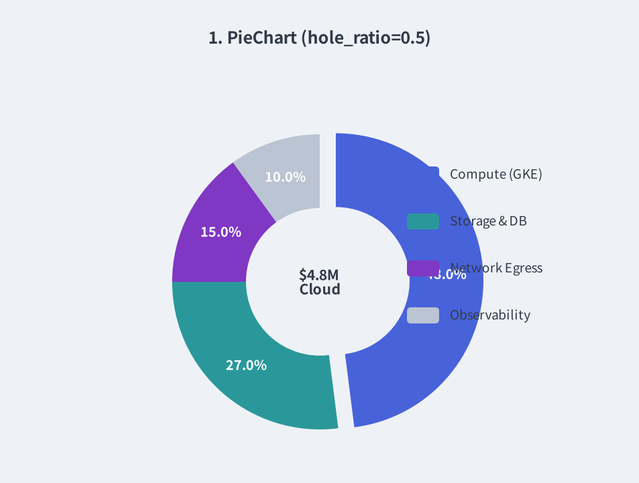
- Set `hole_ratio=0.5` (up to `0.9`) with `center_text` to render executive KPI donut rings.
- Supports per-slice outward emphasis (`explode=2.5`) and automatic label suppression on tiny wedges ($<4\%$).

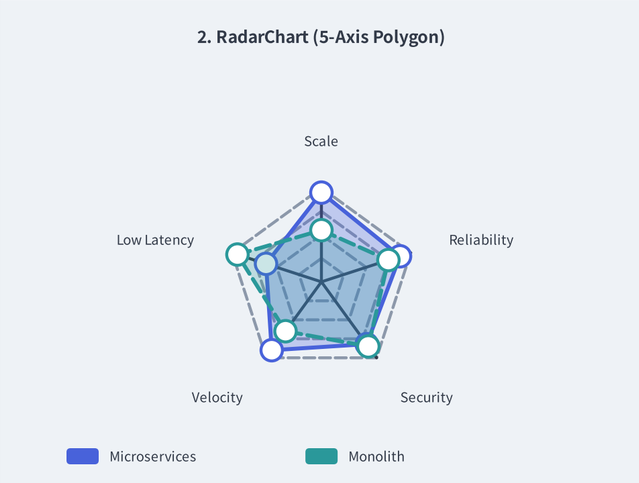
2. `RadarChart` — Multi-Axis Trade-Off Polygons

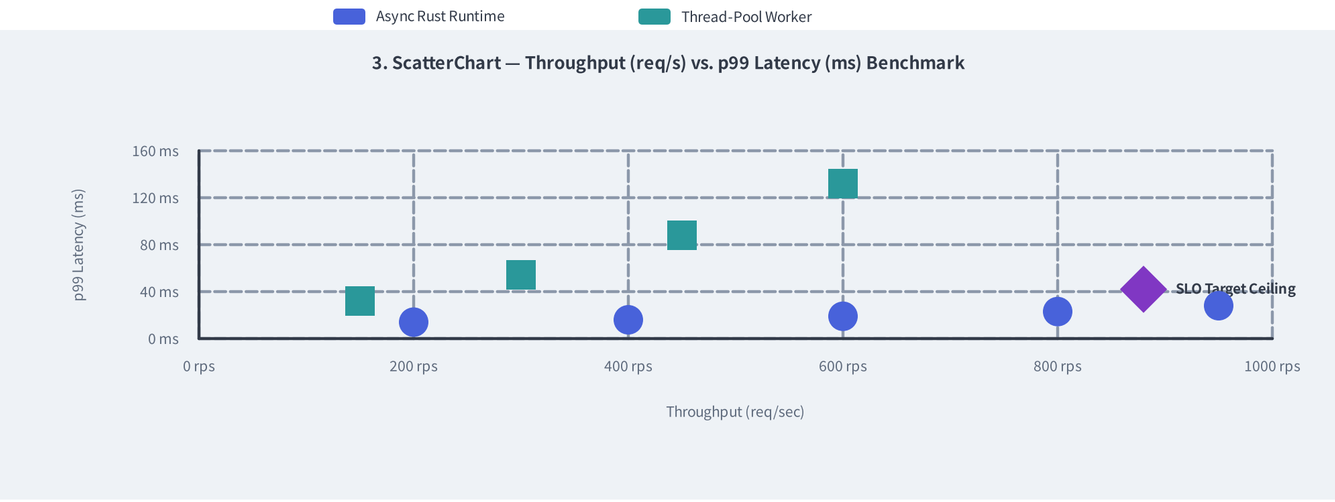
- Evaluates 3+ symmetric dimensions radiating from a central origin (`grid_shape="polygon" | "circle"`).
- Ideal for comparing architectural trade-offs (e.g., Microservices vs. Monolith across Scalability, Latency, Reliability, Operability, and Velocity).

3. `ScatterChart` — XY Plots & 3D Bubbles

- Plots continuous `(x, y)` coordinates or 3-tuple `(x, y, radius)` bubbles across dual independent `linear` or `log` axes.
- Combine named `add_series(...)` clusters with individually labeled benchmark points via `chart.add(xy, label=...)`.







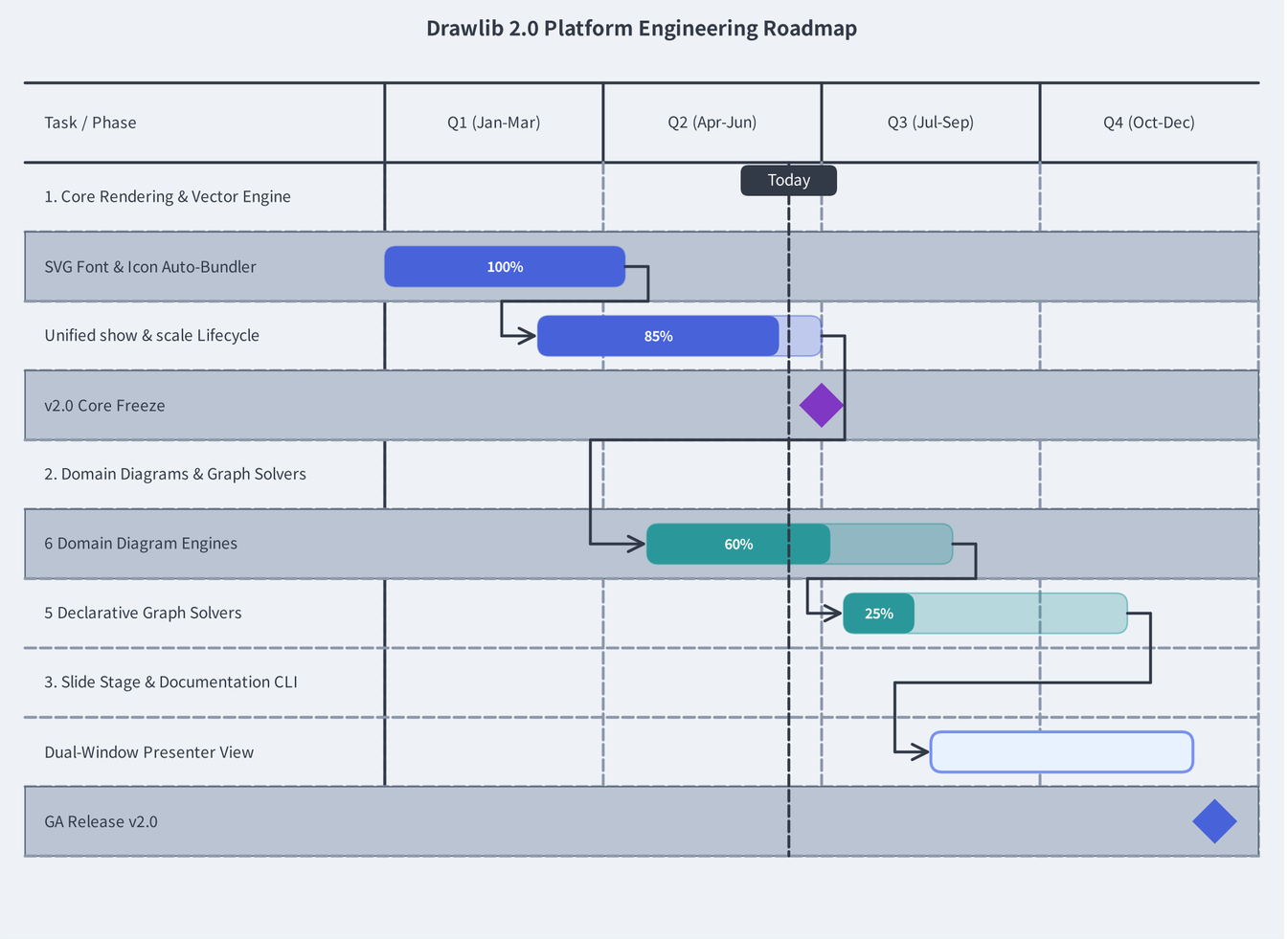
Engineering Roadmaps & Timelines (`GanttChart`)

Declarative Project Schedules as Code

Maintaining engineering roadmaps in external spreadsheet or GUI tools makes version control diffs impossible. `GanttChart` models schedules directly in Python:

Core Schedule Building Blocks

- `add_section(name)`: Full-width horizontal divider bands that group tasks by workstream or team.
- `add_task(name, start, end, style, progress=...)`:
  - `start` and `end` accept column names (`"Apr"`) or continuous floating-point column indices (`1.5` = middle of 2nd column).
  - `progress` (`0.0` to `1.0`) renders a two-tone completion fill bar with percentage text (`progress_text_style`).
- `add_milestone(name, at, style)`: Renders a diamond milestone marker at exact timeline coordinate `at`.
- `add_dependency(from_task, to_task)`: Routes orthogonal finish-to-start dependency arrows between tasks.
- `add_marker(at, style, label="Today")`: Draws a vertical status line across all rows.



CHAPTER

04

Part 4 of 7

Domain Diagrams & Auto-Layout Graph

6 Purpose-Built Software Diagram Engines & 5 Declarative Graph Solvers

4.1 Cloud & Microservice Topologies (ArchitectureDiagram)

4.2 Workflows & API Lifelines (FlowDiagram, SequenceDiagram)

4.3 Data & OOP Structural Models (ERDiagram, ClassDiagram, StateDiagr

4.4 Declarative Auto-Layout Engine (drawlib.graph: 5 Specialized Solver

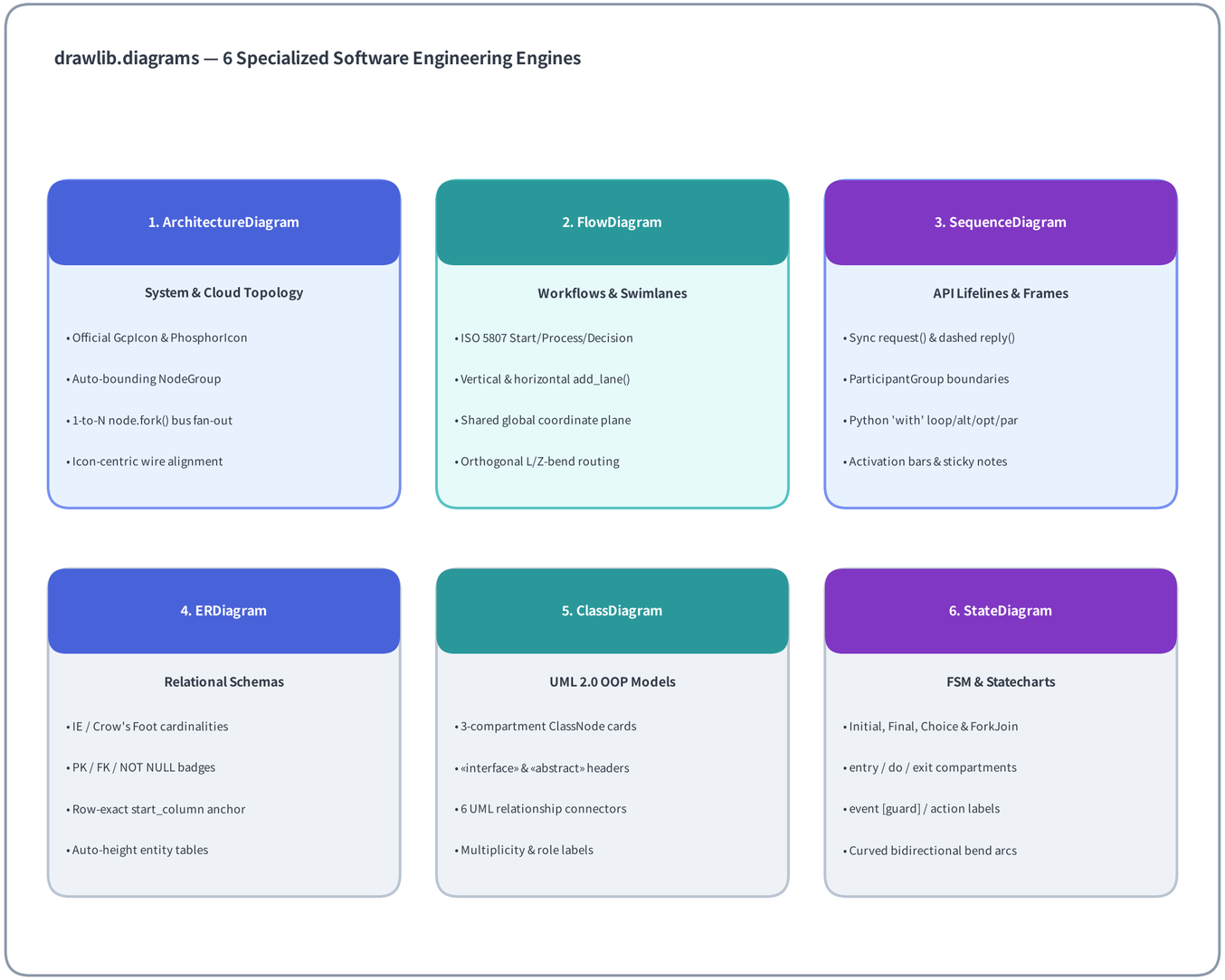
The 6 Domain Diagram Engines (`drawlib.diagrams`)

Purpose-Built Engines with Shared Architecture

Unlike black-box text-to-diagram tools (PlantUML, Mermaid) that scramble your layout whenever a label changes, `drawlib.diagrams` combines **deterministic coordinate placement** with **intelligent boundary & wire automation**:

4 Shared Architectural Guarantees

1. **Center & Icon Coordinate Anchoring**: Passing `xy=(x, y)` to `d.add(...)` anchors the exact geometric center of the card or icon—ensuring aligned nodes always connect with perfectly straight horizontal or vertical wires.
2. **Universal `Connectable` Protocol**: Nodes, groups, entities, classes, states, and `Junction` waypoints share dynamic boundary clipping, `start_side` / `end_side` port selection, and smart `"orthogonal"` / `"direct"` / curved `bend` routing.
3. **Layout-Preserving `show=True/False`**: Hiding any node (`node.show = False`) automatically hides its connected edges and dangling `fork()` stems while **keeping `NodeGroup` bounds, swimlanes, and lifeline spacing 100% locked**.
4. **Uniform `draw(xy, scale=1.0)`**: Scales the entire diagram proportionally around bottom-left `xy`.



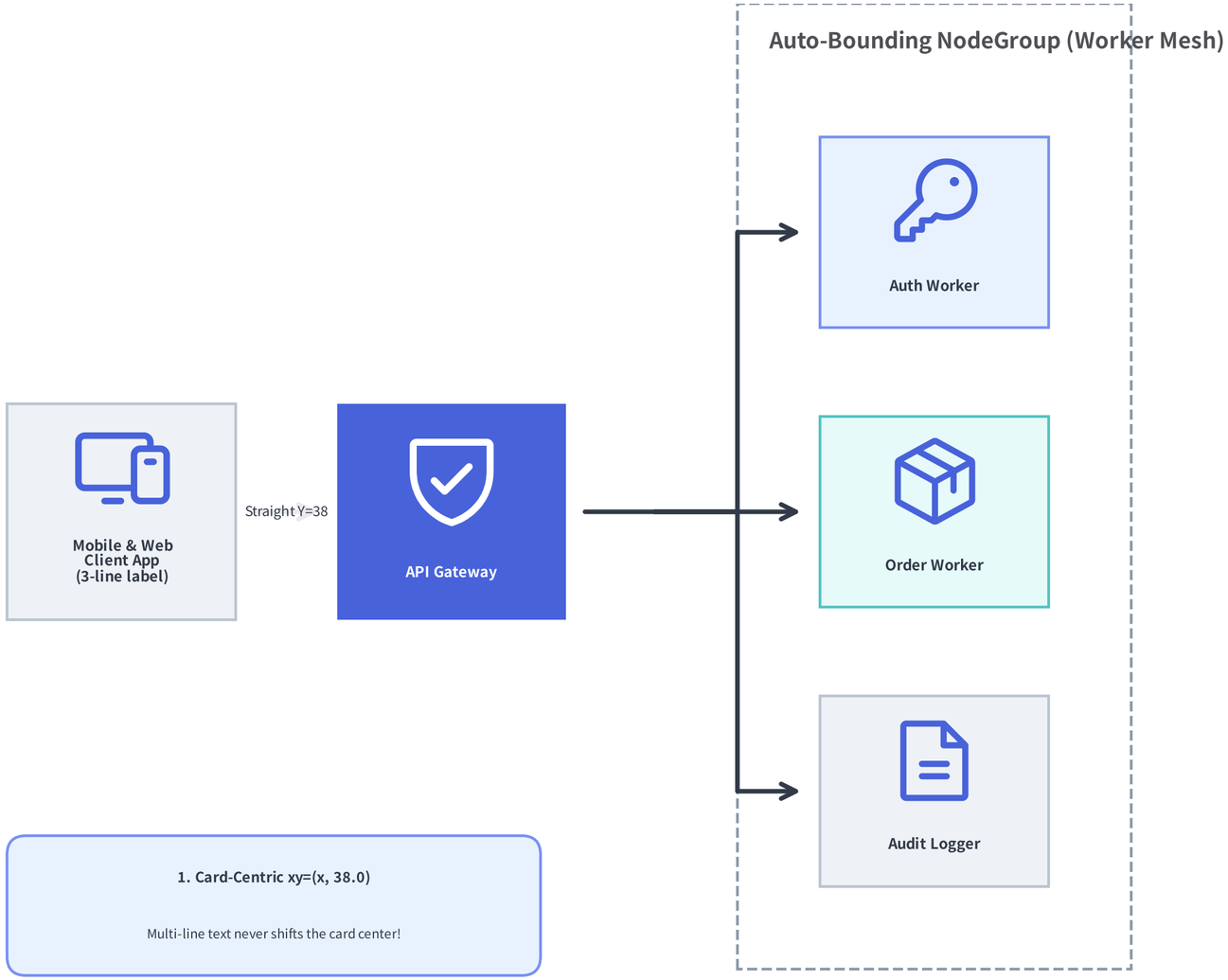
`ArchitectureDiagram` Fundamentals: Icons, Groups & Forks

Why `ArchitectureDiagram` Wires Never Kink

In naive diagramming tools, adding a 2-line label underneath an icon shifts the node's bounding center downward, causing horizontal arrows between aligned icons to develop ugly Z-bends.

3 Core Innovations

1. **Card-Centric Coordinates (`xy` = Card Center)**:
   - When you call `d.add(Node("Label", width=w, height=h, icon=...), xy=(x, y))`, `(x, y)` is strictly the **center of the node card** `(w, h)`.
   - Multi-line labels fit inside the card without shifting the card's center—so nodes sharing the same `y` always connect with a razor-straight horizontal line!
2. **Auto-Bounding `NodeGroup` Containers**:
   - Calling `group = d.add(NodeGroup(title="...", padding=6.0), xy=(gx, gy))` creates a relative coordinate group that automatically expands its boundary box to enclose all child nodes and nested subgroups.
3. **1-to-N Bus Fan-Out (`node.fork()`)**:
   - `lb.fork([svc1, svc2, svc3], at_x=55.0, padding=2.0)` creates a clean vertical distribution bus at `at_x` with orthogonal taps into every target node.

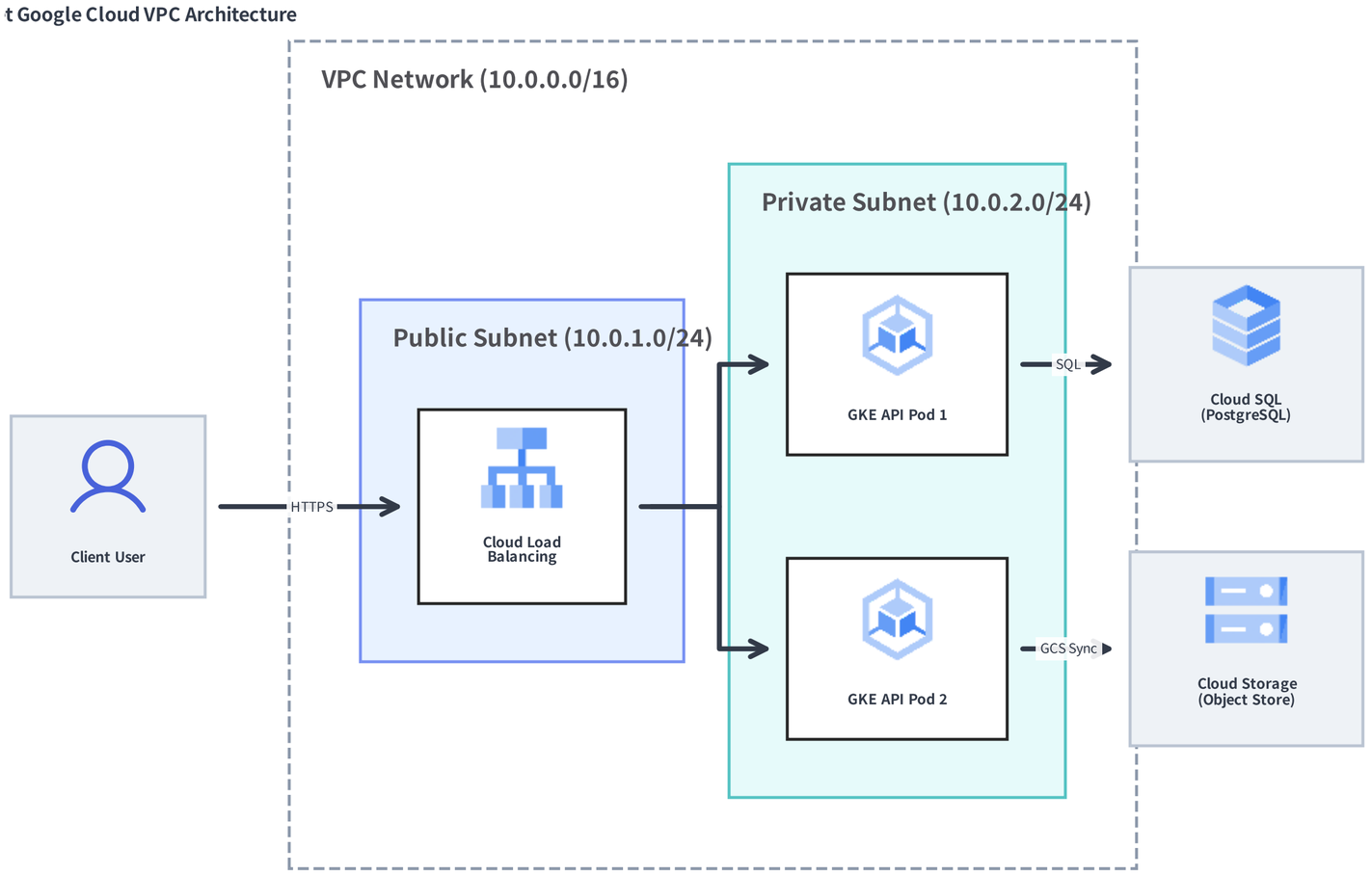


Multi-Tier Production Cloud Topology (`GcpIcon` & `PhosphorIcon`)

Official Cloud & Interface Icons Built In

`drawlib.diagrams.architecture` integrates directly with Drawlib's icon ecosystem:

- `GcpIcon` **(259 Official Icons)**:
  - Compute & Containers: `GcpIcon.GOOGLE_KUBERNETES_ENGINE`, `GcpIcon.CLOUD_RUN`, `GcpIcon.COMPUTE_ENGINE`
  - Networking & Security: `GcpIcon.CLOUD_LOAD_BALANCING`, `GcpIcon.CLOUD_ARMOR`, `GcpIcon.CLOUD_CDN`
  - Databases & Storage: `GcpIcon.CLOUD_SQL`, `GcpIcon.CLOUD_SPANNER`, `GcpIcon.CLOUD_STORAGE`, `GcpIcon.BIGQUERY`
- `PhosphorIcon` **(1,531 Vector Icons)**:
  - Clients, browsers, devices, locks, queues, and generic infrastructure symbols.



Nested Hierarchical `NodeGroup` s

- Nest `Public Subnet` and `Private Subnet` groups inside an outer `VPC Network` group (`vpc.add(NodeGroup(...), xy=...)`).
- Child coordinates inside a `NodeGroup` are local to that group's origin, making subnet refactoring effortless.

Workflows & Swimlanes (`FlowDiagram`)

ISO 5807 Flowcharts with Cross-Lane Coordinates

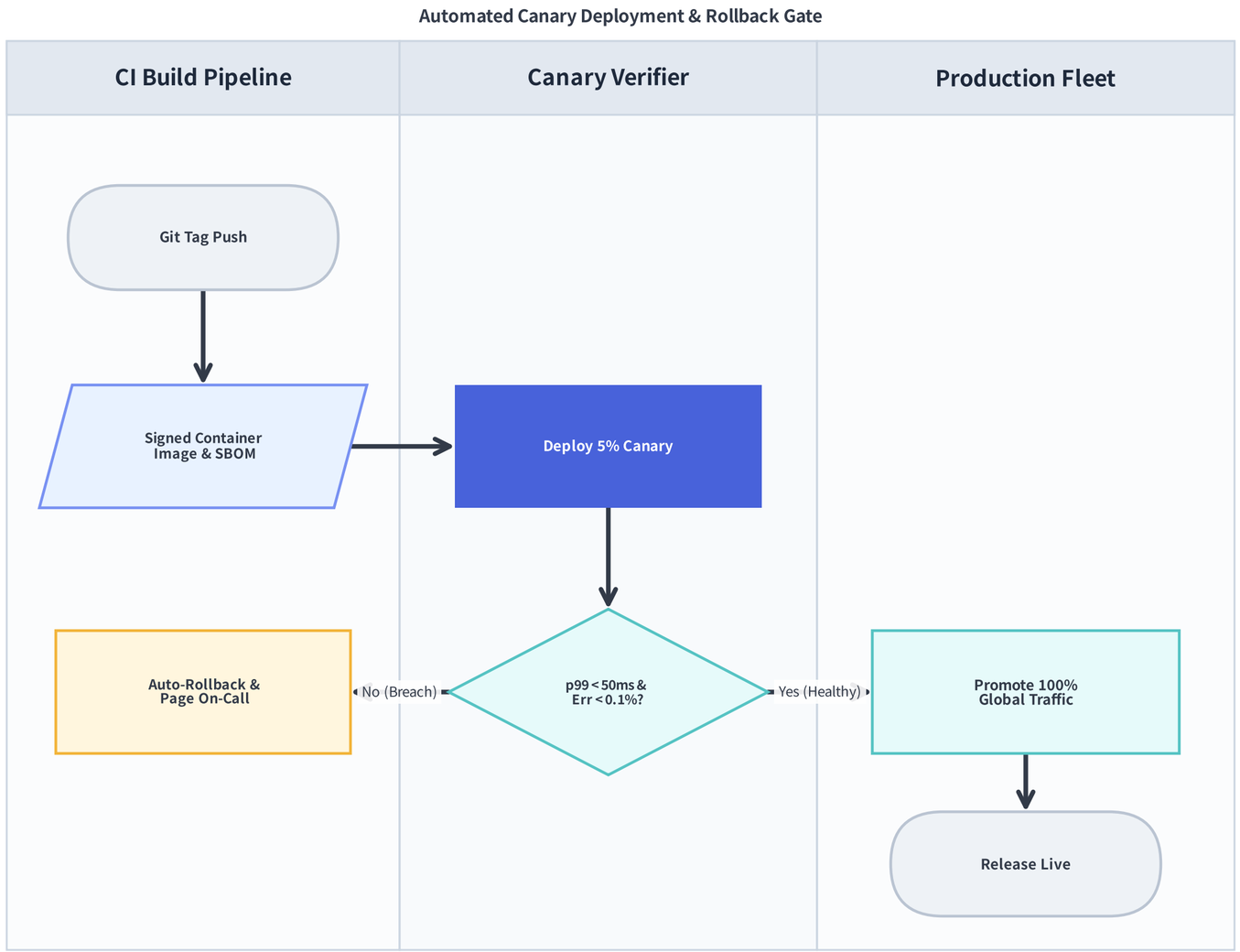
`FlowDiagram` models operational workflows, CI/CD release gates, and approval processes across organizational or system swimlanes:

Standard Flowchart Node Classes

- `Start` / `End`: Rounded stadium/pill terminals (`shape_r=5.0`).
- `Process`: Rectangular execution step (`24 x 12`).
- `Decision`: Diamond conditional gate (`22 x 14`) with explicit `start_side="right" | "bottom" | "left"` branch routing.
- `Data`: Slanted parallelogram for I/O artifacts or payloads (`24 x 12`).

Unified Swimlane Coordinate System

- Call `flow.add_lane(name, width=...)` for vertical columns or set `lane_orientation="horizontal"` for horizontal rows.
- **Global Coordinate Plane**: All nodes share the exact same global `(x, y)` canvas space—so placing steps across different lanes at the same `y` coordinate guarantees horizontal alignment!



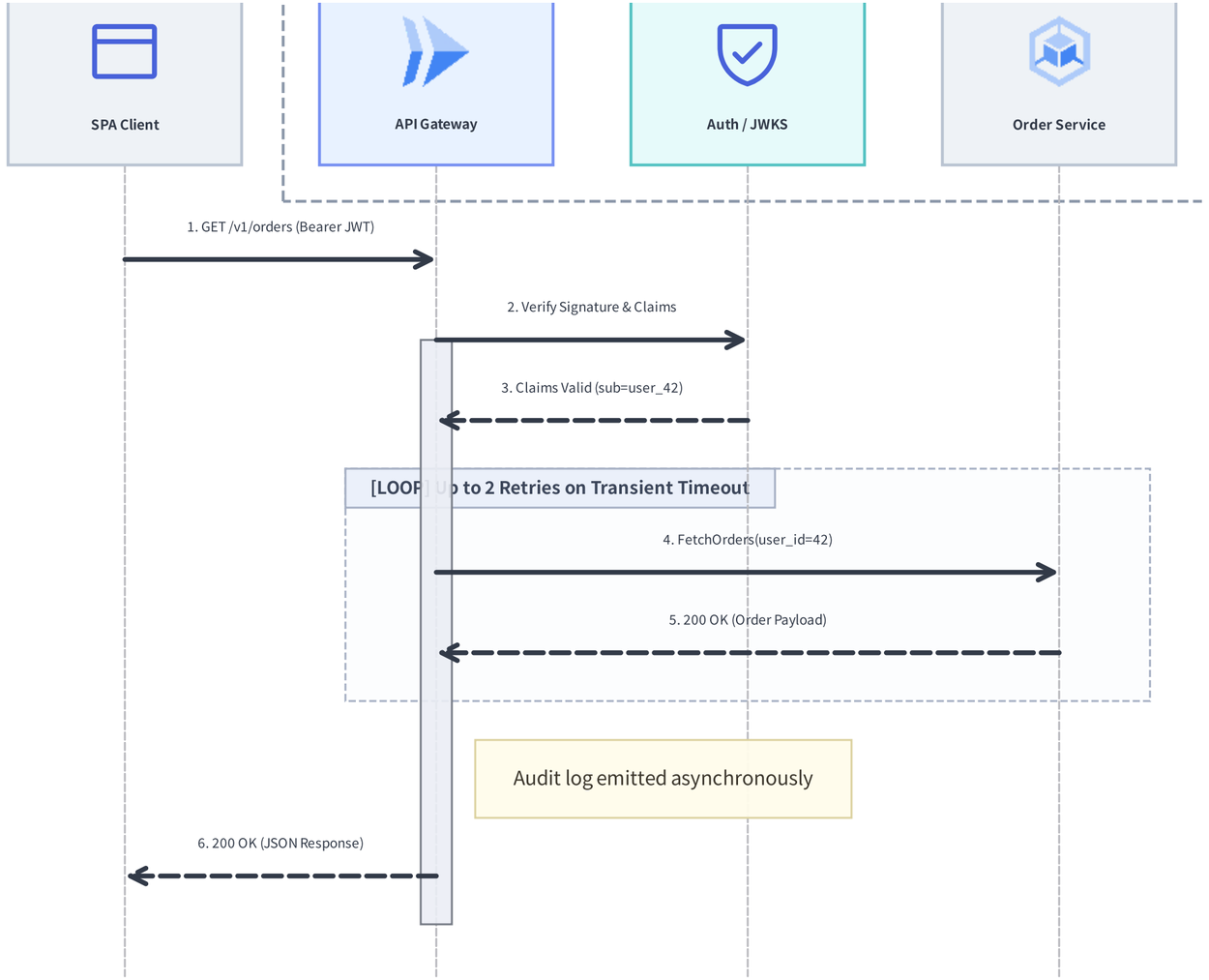
API Interactions & Lifelines (`SequenceDiagram`)

Chronological Protocols & Python Context Managers

`SequenceDiagram` turns protocol specifications into clean lifeline diagrams without manual Y-coordinate bookkeeping:

Expressive Python Idioms

- **Participants & Groups**:
  - `d.add(Participant("Name", width=18, height=13, icon=GcpIcon.CLOUD_RUN))`
  - `vpc = d.add(ParticipantGroup(title="Zero-Trust Mesh", style=Styles.MutedDashed))`
- **Message Verbs (`autonumber=True`)**:
  - `a.request(b, "label")`: Solid synchronous call arrow (`-►`).
  - `b.reply(a, "label")`: Dashed return arrow (`---►`).
  - `is_async=True`: Open stick arrowhead (`->`) for Pub/Sub or async events.
- **Execution & Annotations**:
  - `p.activate()` / `p.deactivate()` for lifeline execution bars.
  - `p.note("...", pos="right")` or `d.note("...", over=[p1, p2])` for sticky notes.
- **Python `with` Condition Frames**:



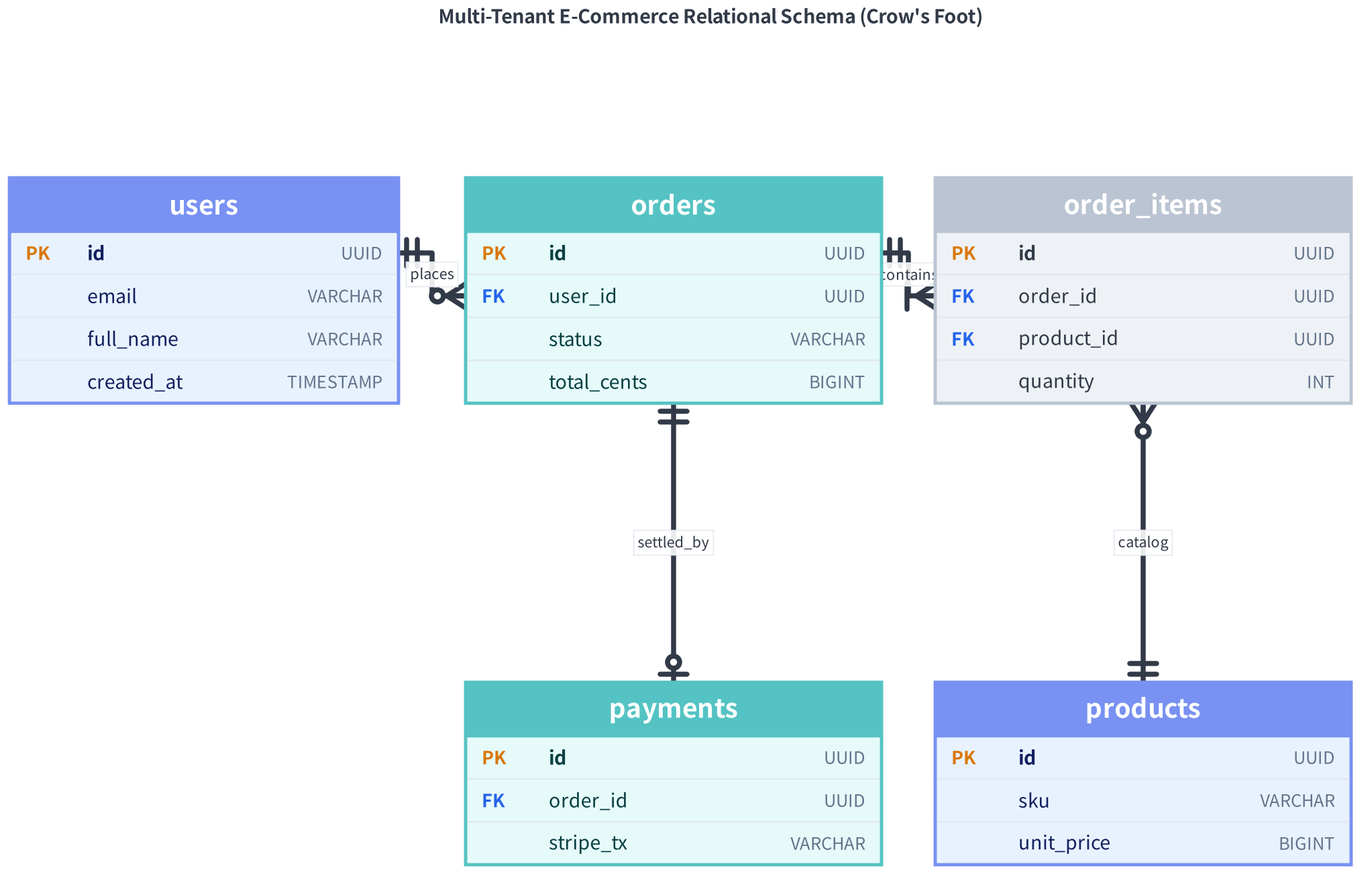
Relational Schemas & Crow's Foot (`ERDiagram`)

Information Engineering (IE) Database Modeling

`ERDiagram` visualizes physical and logical relational database schemas with strict Crow's Foot cardinality markers and row-aligned foreign key connectors:

Entity & Column Capabilities

- **Auto-Height Tables**: `Entity(name="users", width=28.0)` automatically scales its vertical height based on registered columns.
- **Constraint Badges**: `entity.add_column(name, type="...", pk=True, fk=False, nullable=False)` renders `[PK]`, `[FK]`, and mandatory field indicators cleanly aligned in the right column.
- **7 Crow's Foot Cardinalities**:
  - `"1:*"` (One-to-Zero-or-More `|| ── o<`)
  - `"1:1..*"` (One-to-One-or-More `|| ── |<`)
  - `"1:1"`, `"1:0..1"`, `"0..1:1"`, `"0..1:*"`, `"*:*"`
- **Row-Exact Foreign Key Anchoring**:
  - Passing `start_column="id", end_column="user_id"` attaches the relationship wire directly to the vertical Y-center of those specific table rows!



UML 2.0 Object-Oriented Models (`ClassDiagram`)

Three-Compartment Classes & 6 UML Connectors

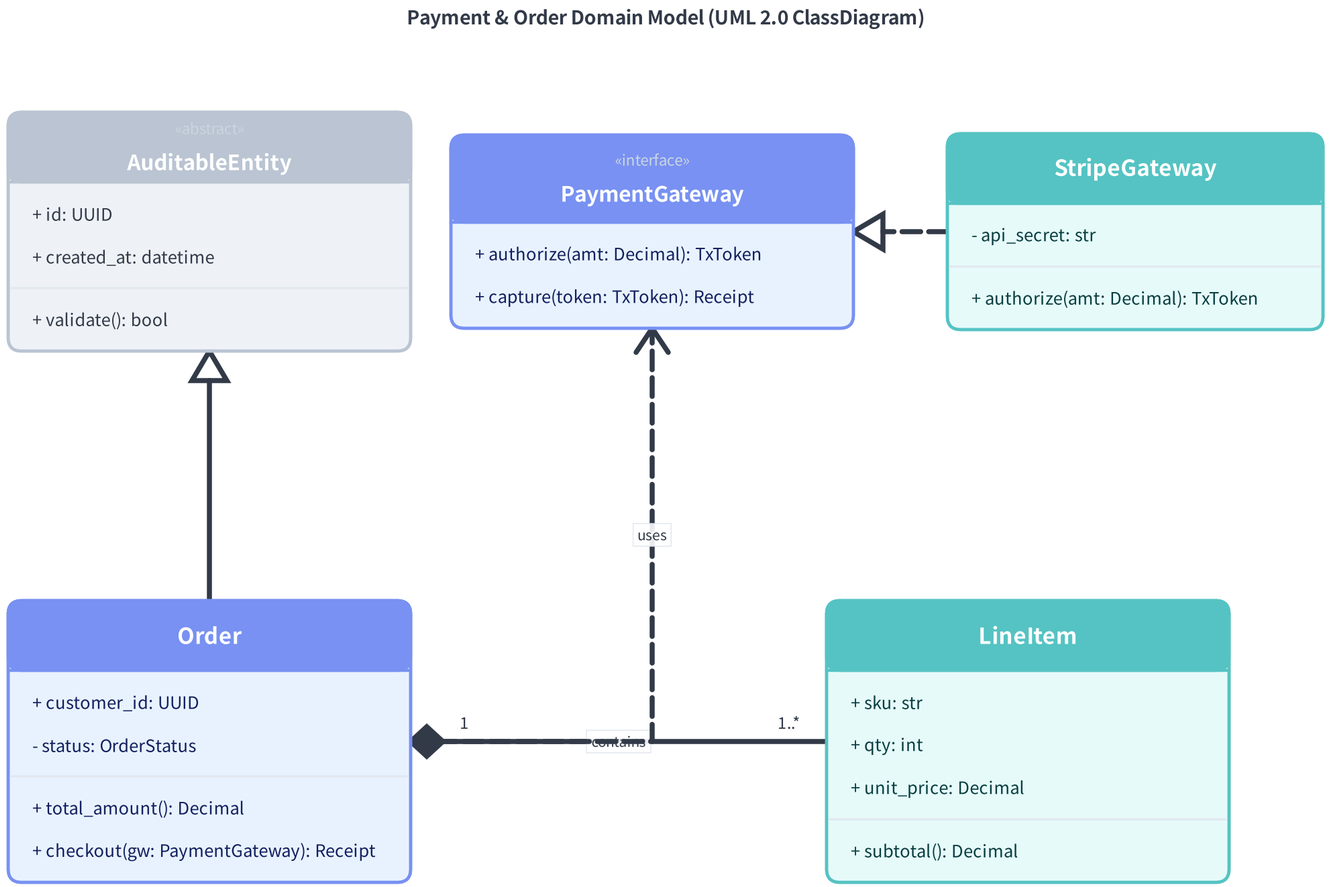
`ClassDiagram` models object-oriented domain architectures, design patterns, and SDK type hierarchies in standard UML 2.0 notation:

`ClassNode` Structure

- **Header Compartment**: Supports `stereotype="interface"` (`«interface»`) and `is_abstract=True` (`«abstract»`).
- **Attributes (`add_attribute`)**: Automatic visibility prefixes (`+` public, `-` private) and static underlines.
- **Methods (`add_method`)**: Parameter signatures, return types, and abstract method styling.

All 6 Standard UML Relationship Types

`relationship_type`	Line & Marker	Semantics
`"inheritance"`	Solid + Hollow Triangle	Subclass generalizes superclass



Finite State Machines & Lifecycles (`StateDiagram`)

UML 2.0 Statecharts & Concurrent Forks

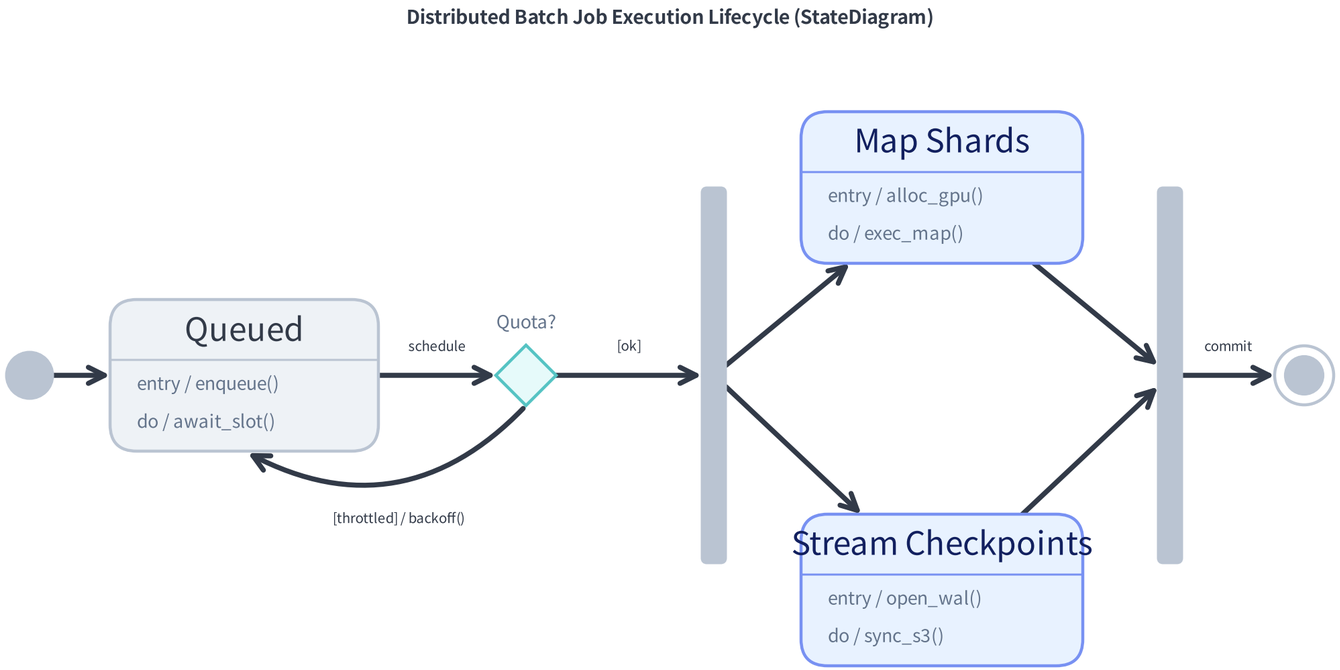
`StateDiagram` models finite state automata, protocol lifecycles, and distributed job orchestrators:

Rich State & Pseudo-State Palette

- `State(name, shape="box", entry=..., do=..., exit=...)`:
  - Supports 5 shapes (`"box"`, `"ellipse"`, `"circle"`, `"double_circle"`, `"text_only"`) and internal action compartments (`entry / ...`, `do / ...`, `exit / ...`).
- **UML Pseudo-States**:
  - `InitialState()`: Solid black entry circle.
  - `ChoiceState(name="...")`: Dynamic conditional diamond branch.
  - `ForkJoinState(orientation="vertical" | "horizontal", length=...)`: Solid synchronization bar for concurrent splits and joins.
  - `FinalState()`: Bullseye termination target.

Formal Transition Formatting (`sd.connect`)

- `sd.connect(src, dst, event="...", guard="...", action="...", bend=0.25)` automatically formats



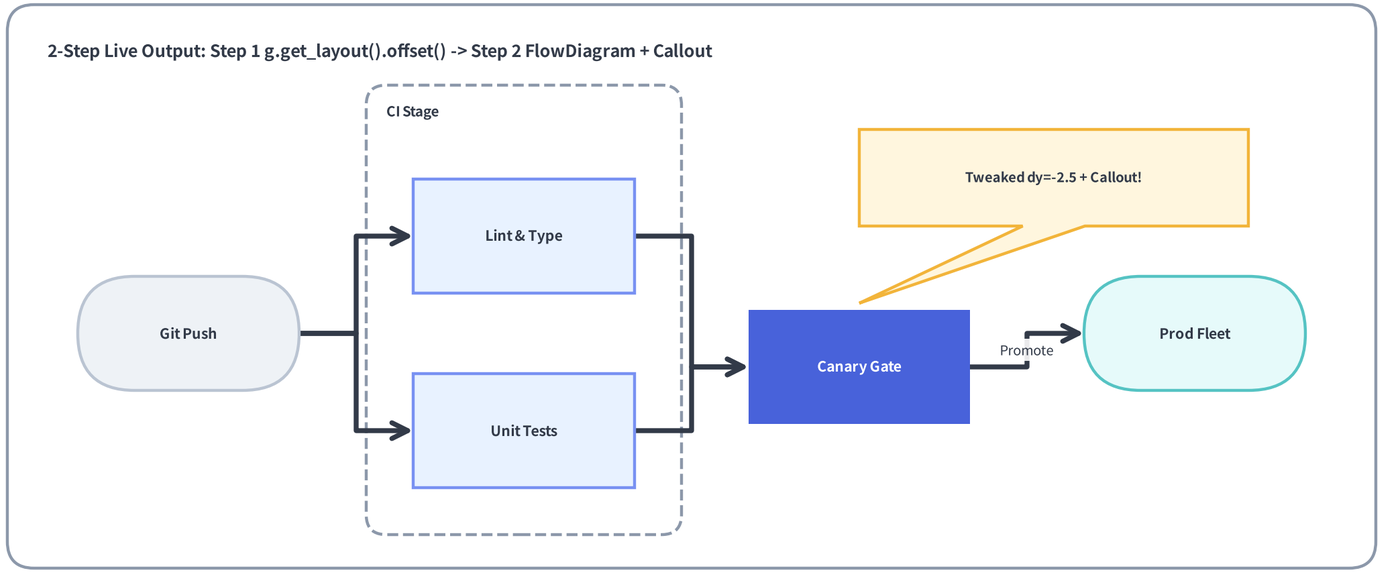
2-Step Auto-Layout Engine (`drawlib.graph` → `get_layout()`)

Pure Coordinate Solver + Rich Rendering

Manual `(x, y)` coordinates give total precision, while black-box tools like Graphviz lock you out of custom icons and styles. `drawlib.graph` bridges both worlds as a **pure coordinate calculator (`get_layout()` only)**:

The 2-Step Workflow of `drawlib.graph`

1. **Step 1 — Solve & Inspect (`layout = g.get_layout()`)**:
   - Declare topology via `.node(id, width=..., height=...)`, `.edge(src, dst)`, and `.cluster(id, nodes=[...])`.
   - Call `layout = g.get_layout(width=..., height=..., margin=...)` to compute a `GraphLayout`.
   - Inspect all solved coordinates via `print(layout)` or fine-tune positions with `layout.offset("canary", dy=-2.5)`.
2. **Step 2 — Render with `diagrams` or `shapes`**:
   - Pass `layout["id"].xy`, `.local_xy`, `.bottom_left`, `.local_bottom_left`, or `edge.points` directly into `ArchitectureDiagram`, `FlowDiagram`, or `drawlib.shapes` / `drawlib.lines`.
   - Attach custom callouts (`bubblespeech`) or charts to exact solved coordinates!



The 2-Step Workflow — Pure Layout Calculation (Step 1) + Rich Diagram Rendering (Step 2)

```python
# Step 1: Solve & Inspect Topology
g = LayerGraph(direction="LR", default_node_width=19)
g.tier("ci", ["lint", "test"], layer=1)
g.cluster("ci_cluster", ["lint", "test"])
g.edge("git", "lint"); g.edge("lint", "canary")
layout = g.get_layout(width=100, height=35)
layout.offset("canary", dy=-2.5)
print(layout)  # Inspect xy, local_xy, bottom_left
```

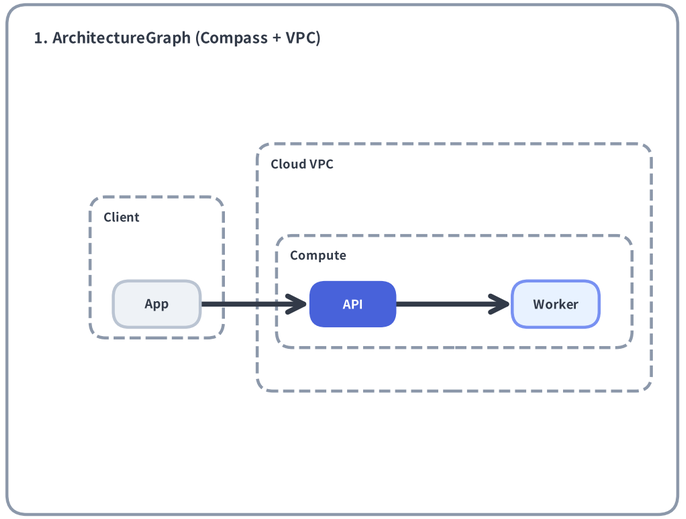
```python
# Step 2: Render with Diagrams or Shapes
flow = FlowDiagram(node_style=Styles.Neutral, ...)
git = flow.add(Start("Git Push"), xy=layout["git"].xy)
can = flow.add(Process("Canary"), xy=layout["canary"].xy)
flow.connect(git, can); flow.draw()
# Or attach callouts to layout["canary"]:
c = layout["canary"]
bubblespeech((c.x, c.y + 10.5), ...)
```

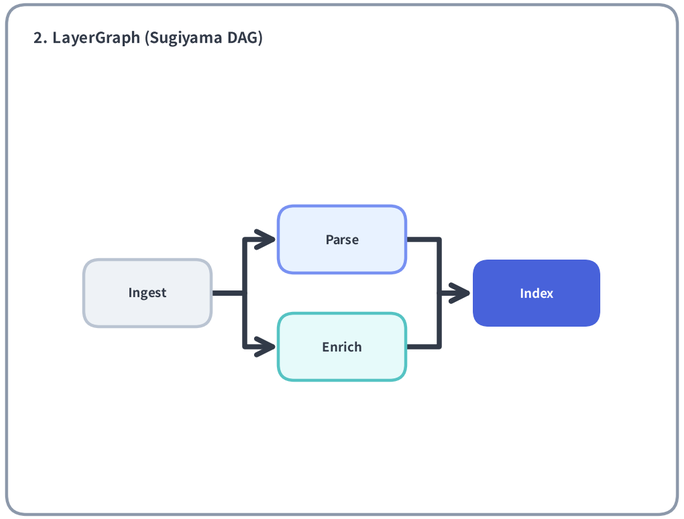
The 5 Specialized Graph Layout Solvers

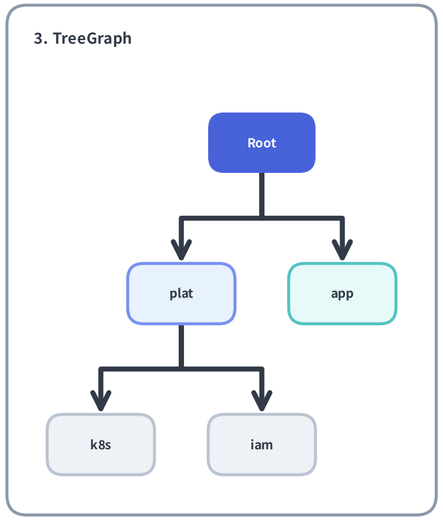
Choosing the Right Solver for Your Topology

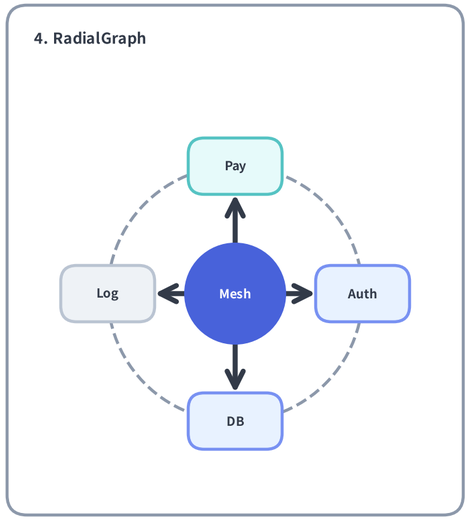
Instead of forcing one generic algorithm onto every graph, `drawlib.graph` provides **5 domain-tuned layout solvers** sharing the `BaseGraph.get_layout()` API:

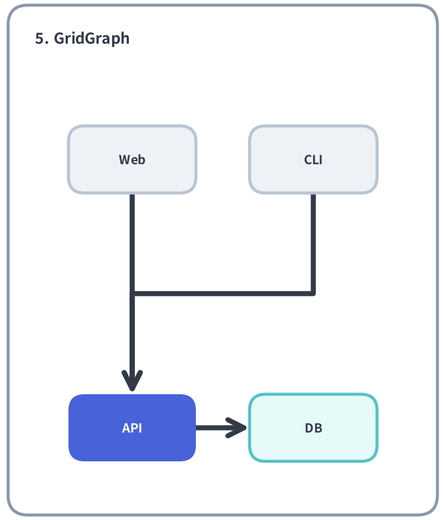
1. `ArchitectureGraph` **(Compass + Nested Packing)**:
   - 2-level macro/micro packer: nests child clusters inside parent containers (`parent="vpc"`) and positions top-level zones via compass coordinates (`pos="left" | "center" | "right" | "top" | "bottom"`).
2. `LayerGraph` **(Sugiyama Hierarchical DAG)**:
   - Cycle removal, longest-path rank assignment, `.tier()` rank pinning, and barycenter edge-crossing minimization.
3. `TreeGraph` **(Reingold-Tilford Compact Tree)**:
   - Proportional subtree contour spacing via `.child(parent, child)` so sibling branches never overlap.
4. `RadialGraph` **(Concentric Hub & Spoke)**:
   - Places a central `hub` at `ring=0` , allocates angular sectors across concentric rings, and exposes radii via `layout.ring_radii` .
5. `GridGraph` **(2D Matrix + Smart Channel Routing)**:
   - Places cells at `(row, col)` with `.cluster_row()` / `.cluster_column()` and routes non-adjacent edges through inter-cell channels.











CHAPTER

05

Part 5 of 7

Multi-Frame Animations

Programmatic Keyframes, Progressive Reveal & Interactive Slide Controls

5.1 Declarative Frame Context Manager (Animation & anim.frame)

5.2 Layout-Preserving Element Reveal (.show = False -> True)

5.3 Quantitative Chart Growth (.draw_ratio & .draw_direction)

5.4 Interactive Slide Playback (anim:click, anim:once, anim:pause)

Animation Architecture (`drawlib.anim.Animation`)

Declarative Frame Context Manager

Drawlib compiles multi-frame **APNG (`.png`)** and **Animated WebP (`.webp`)** directly from pure Python loops—with zero external video encoders required.

```
from drawlib.anim import Animation
from drawlib.canvas import clear, save, setup
from drawlib.shapes import circle
from drawlib.styles import Styles

clear()
setup(width=100, height=40)
anim = Animation(fps=8.0, loop=0)

for idx, x in enumerate([20, 40, 60, 80]):
```



Key Architectural Guarantees

- **Per-Frame Redraw (`clear=True` default)**: Each `with anim.frame():` clears the canvas and captures a clean snapshot on exit.
- **Per-Frame Duration Overrides**: Hold the final frame (`duration=2.0`) so viewers can absorb the completed state before looping.
- **Poster Frame Compatibility**: Frame 0 serves as the static fallback for GitHub Markdown and vector PDF exports.

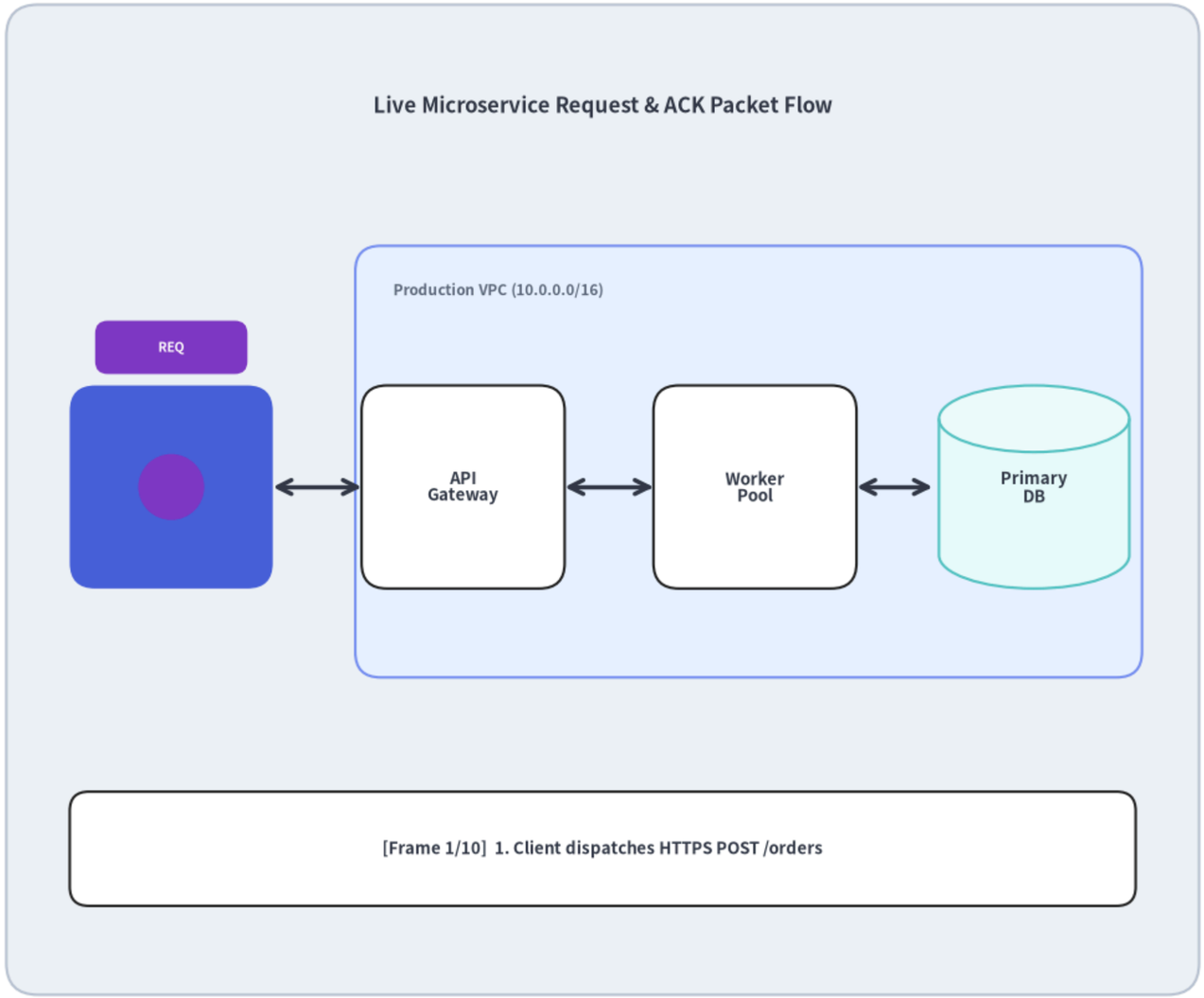
Animating Primitives & Packet Traffic

Coordinate & Color Interpolation

By computing coordinates `(x, y)` or interpolating colors via `get_intermediate_colors()` across frames, primitive shapes illustrate live request flows, heartbeats, and state transitions.

```
from drawlib.anim import Animation
from drawlib.canvas import clear, save, setup
from drawlib.shapes import circle, rectangle
from drawlib.styles import Styles

clear()
setup(width=98, height=84)
anim = Animation(fps=8.0, loop=0)
```



Best Practices for Motion Loops

- **Hoist Invariant Coordinates**: Calculate node anchors and packet waypoints before entering the frame loop.
- **Highlight Active Receiver**: Use `Styles.PrimaryFlat` on the node currently processing the request while keeping idle nodes in `Styles.Neutral`.
- **Smooth Color Fades**: Use `get_intermediate_colors(c1, c2, num=3, include_ends=True)` for gradual heatmaps or status transitions.

Progressive Component Reveal (`.show = False` → `True`)

Zero Layout Jump Across Keyframes

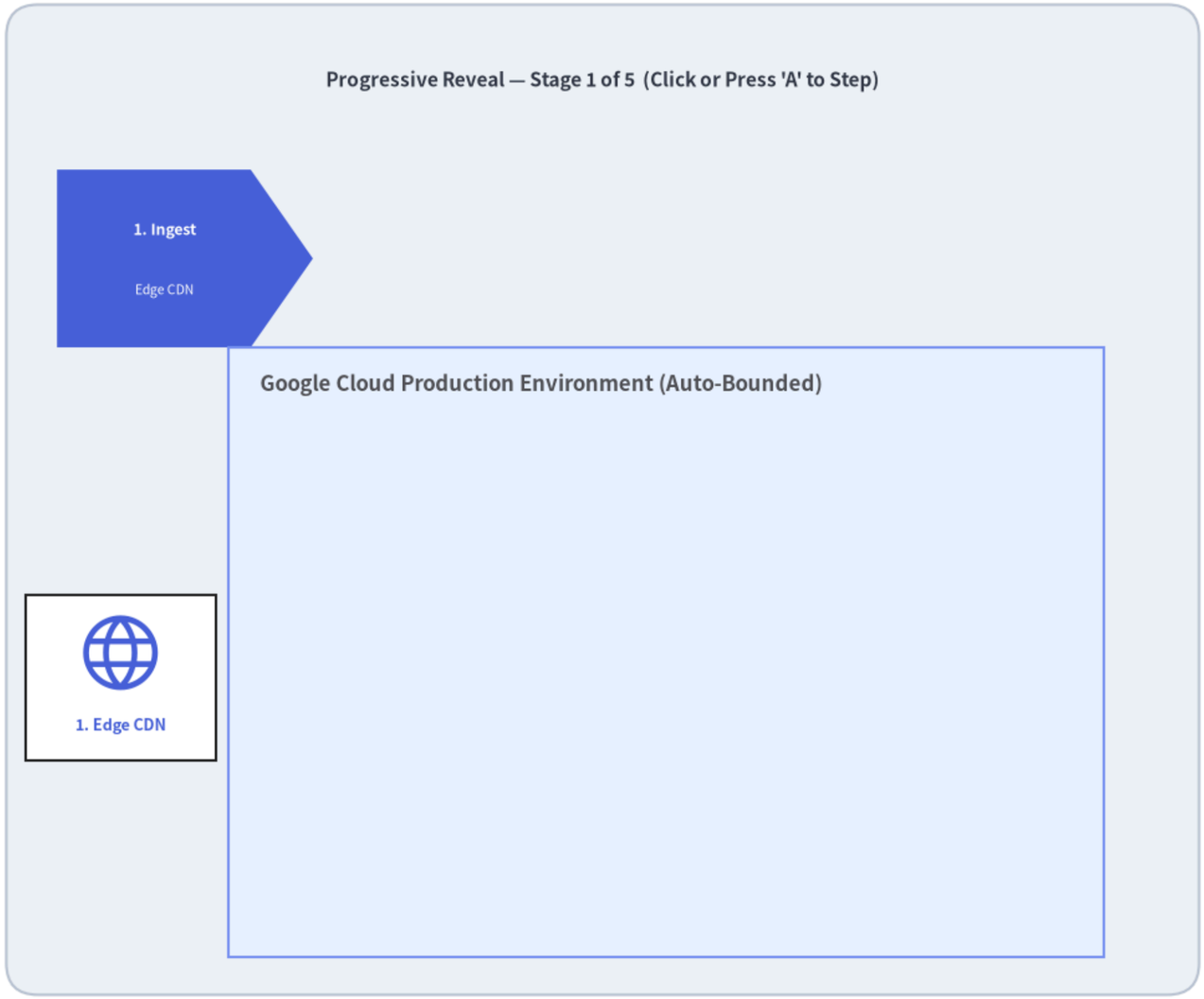
Every element in `drawlib.smartarts` , `drawlib.diagrams` , and `drawlib.graph` supports a mutable `.show: bool` property.

```
# Build full topology once outside the loop
d = ArchitectureDiagram(...)
vpc = d.add(NodeGroup("Cloud VPC"), xy=(6, 8))
n1 = vpc.add(Node("1. Edge LB", width=16, height=14,
n2 = vpc.add(Node("2. App Tier", width=16, height=14,
n3 = vpc.add(Node("3. Storage", width=16, height=14,
d.connect(n1, n2, label="gRPC")
d.connect(n2, n3, label="SQL")

# Reveal progressively across frames
for step in range(3):
    n2.show = (step >= 1)
```

Why `.show` Beats Omitting `.add()`

- **Fixed Container Bounds**: `NodeGroup` , `ChevronProcess` , `GridLayout` , and `LayerGraph` compute full bounding boxes across all registered items—even when `show=False` .
- **Automatic Edge Hiding**: Hiding a diagram node (`n3.show = False`) automatically hides its incoming/outgoing edges and dangling junctions.



Quantitative Chart Growth (`.draw_ratio` & `.draw_direction`)

Spatial Interpolation with Locked Axes

All 7 chart types in `drawlib.charts` return mutable `Series`, `Slice`, or `Task` objects supporting `.draw_ratio` (`0.0` → `1.0`) and `.draw_direction`.

```
bar_s = bar_chart.add_series(
    "Throughput", [42, 68, 95, 128],
    style=Styles.PrimaryFlat,
    draw_direction="bottom_to_top",
)
line_s = line_chart.add_series(
    "P99 Latency", [38, 29, 19, 12],
    style=Styles.AccentFlat,
    draw_direction="left_to_right",
)
```



Dual Direction Modes across Chart Families

- `"bottom_to_top"`: Bars grow vertically from baseline; `RadarChart` polygons expand radially outward; `PieChart` wedges grow from inner hole to outer radius.
- `"left_to_right"`: `LineChart` & `AreaChart` sweep continuously along arc length; `GanttChart` tasks extend across timeline; `PieChart` sweeps angularly.

Interactive Slide Animation Controls (`click`, `once`, `pause`)

Presenter-Driven Keyframe Stepping

In slide decks, you control `<canvas>` animation playback directly from the ```` ```drawlib ```` code fence header—turning multi-frame APNGs into interactive slide builds:

Fence Option	Shorthand / Alias	Behavior in `slide.js`
`anim-trigger:auto`	`anim:auto`	Starts playing automatically when entering the slide.
`anim-trigger:click`	`anim:click`	Holds on **Frame 0 (`READY`)** until clicked or `A` is pressed.
`anim-loop:infinite`	`loop:infinite`	Loops continuously back to Frame 0 after the last frame.
`anim-loop:once`	`loop:once`	Stops on the final frame (`ENDED` badge; click to reset).
`anim-pause:1,2`	`pause:1,2`	Pauses at 0-based frame indices (`PAUSED` badge) until the next trigger.



Synchronized Dual-Window Control

Zero-Dependency Base64 Canvas Playback (`file://` Ready)

Why Standard `<img>` APNGs Fail in Slide Decks

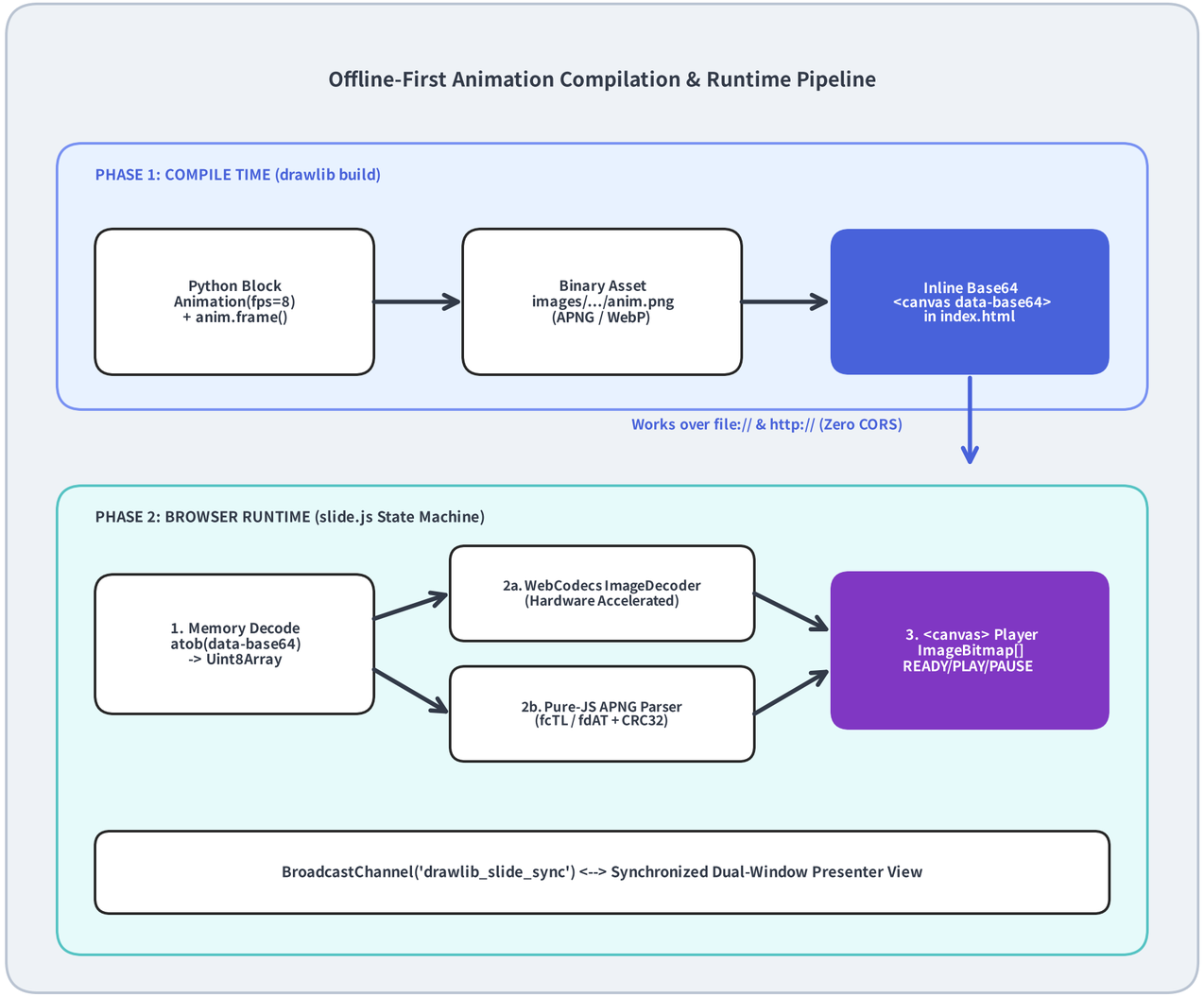
Browsers play `<img src="anim.png">` passively with **zero JavaScript API** to pause, step, reset on slide entry, or synchronize with Presenter View—and `fetch("anim.png")` fails with CORS errors when opened via `file://`.

Drawlib's Offline `<canvas>` Architecture

1. **Build-Time Base64 Embedding (`_assets.py`)**:
   - During `drawlib build`, the compiled APNG/WebP binary is Base64-encoded directly into `<canvas class="drawlib-anim-canvas" data-base64="...">` alongside the file in `images/`.

2. **Zero-CORS Binary Decoding (`slide.js`)**:
   - `window.atob(base64Data)` reconstructs the raw `Uint8Array` buffer in memory—working 100% offline via double-click (`file://`) without an HTTP server.

3. **Dual-Engine Frame Extractor**:
   - **Primary Path**: Hardware-accelerated WebCodecs `ImageDecoder` API extracts `ImageBitmap` frames + microsecond durations.
   - **Fallback Path**: Built-in pure-JS APNG chunk parser (`acTL` , `fcTL` , `IDAT` , `fdAT` + CRC32) composites frames onto an offscreen canvas.



CHAPTER

06

Part 6 of 7

Documentation & Slide Engine

One Authoring Paradigm for Sites, Technical Docs, Vector PDFs & 16:9 Slides

6.1 Four Project Archetypes via drawlib init project (site, doc, slide, imag

6.2 Multi-Page Documentation Sites & Single-Page Standalone Docs

6.3 16:9 Widescreen Stage (1920x1080) & Spatial Freedom

6.4 Reusable Helpers (utils.py, styles.py) & Presenter View

Four Project Archetypes (`drawlib init project`)

Zero-Config Project Scaffolding

Never assemble documentation directories or build scripts manually. `drawlib init project` scaffolds production-ready project structures with synchronized themes (`styles.py` + `style.css`) and modular `build_*.sh` scripts:

```
# 1. Multi-page documentation website with sidebar
uv run drawlib init project site -s default

# 2. Linear technical specification / RFC / whitepape
uv run drawlib init project doc -s google

# 3. 16:9 widescreen presentation slide deck
uv run drawlib init project slide -s default

# 4. Standalone batch Python illustration repository
```

Strict Source-of-Truth Separation

- **Edit Only** `<target>_src/` : All human- and agent-authored Markdown, `styles.py` , and `utils.py` live in `<target>_src/` .
- **Deterministic Outputs**: `./<target>_src/build.sh` compiles `<target>_html/` , `<target>.pdf` , `<target>_markdown/` , and `<target>_images/` .



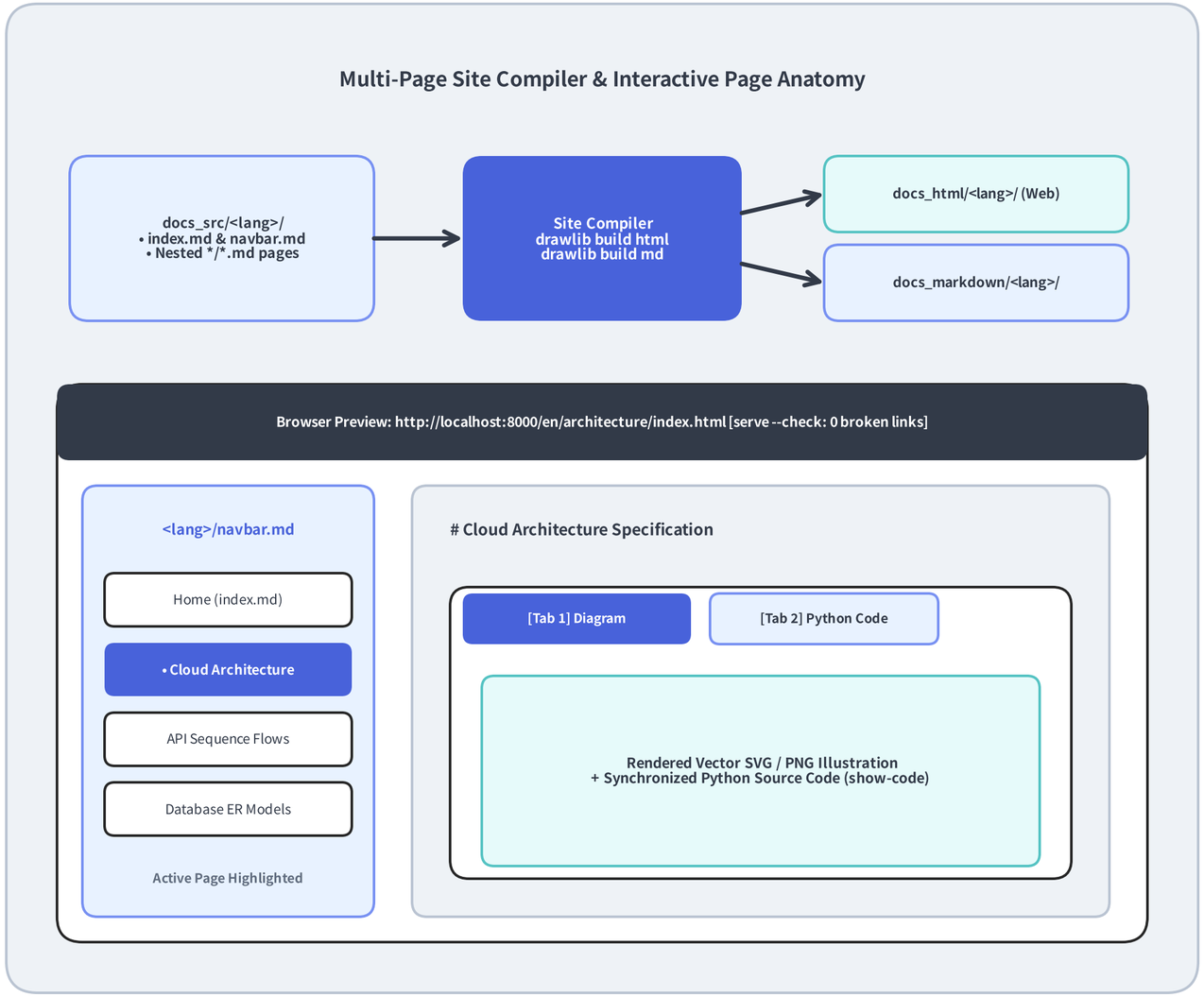
Multi-Page Documentation Sites (`site`)

Technical Portals & GitHub Mirrors

A `site` project (`docs_src/<lang>/` + `drawlib.toml`) compiles nested Markdown folders into both a responsive HTML documentation portal (`docs_html/<lang>/`) and a GitHub-browsable Markdown mirror (`docs_markdown/<lang>/`).

Core `site` Capabilities

- **Declarative Sidebar (`<lang>/navbar.md`)**:
  - `# Brand Title` sets the top-left header; `## Section` groups links into collapsible categories; every internal `.md` link is validated at build time.
- **Interactive Code Tabs (`show-code` / `fold-code`)**:
  - Adding `show-code` to a ```` ```drawlib ```` fence renders an interactive UI card allowing readers to toggle between the **Rendered Diagram** and **Copyable Python Source**.
- **Pre-Flight Link & Asset Scanner (`drawlib serve --check`)**:
  - `scan_broken_links()` verifies all internal `<a href>` links, section anchors, and `<img src>` assets before launching the live preview server.



Single-Page Technical Docs & Vector PDFs (`doc` & `pdf`)

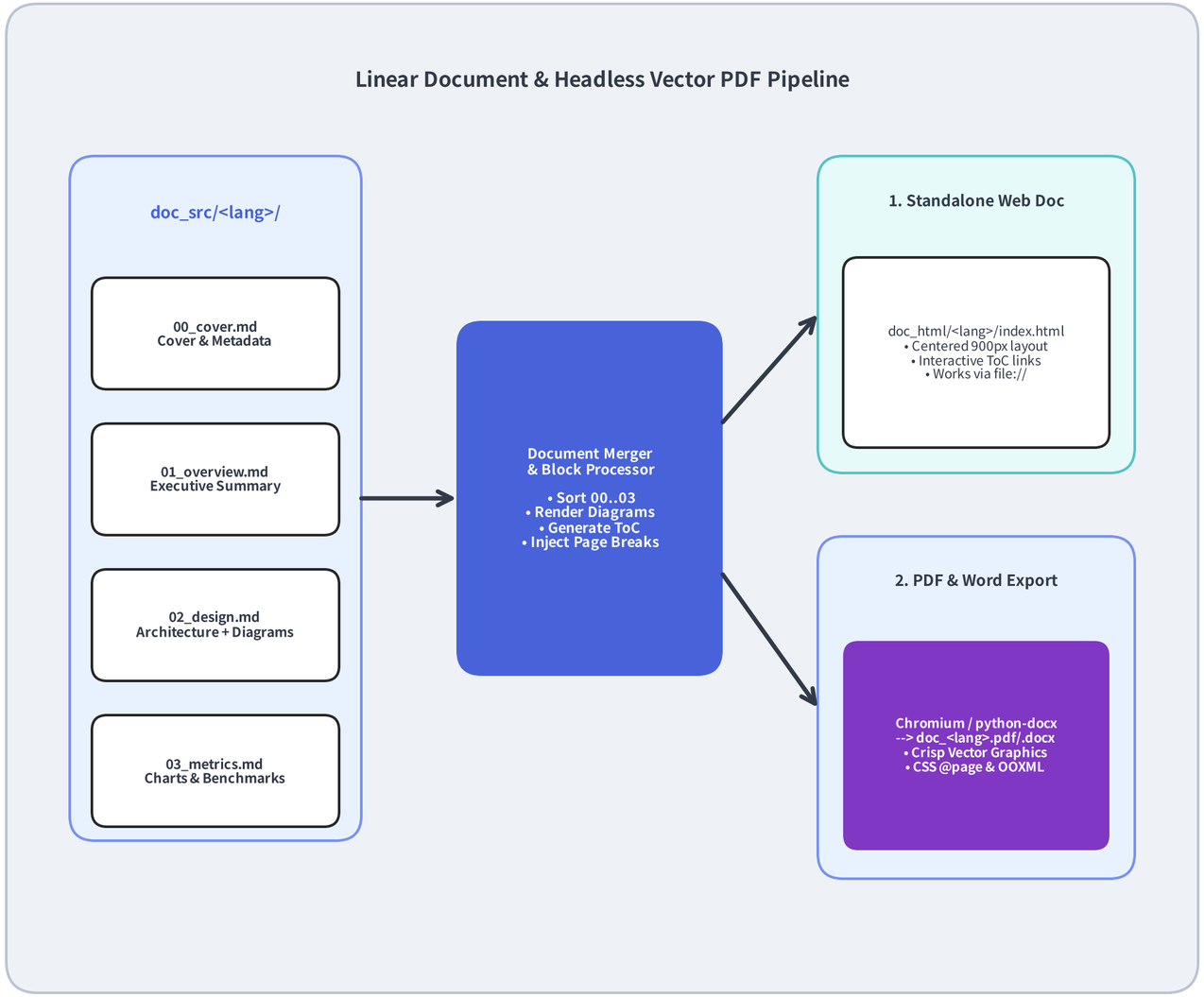
Chapter-Merged Whitepapers & RFCs

The `doc` archetype (`doc_src/`) is purpose-built for linear technical documents—such as architecture RFCs, whitepapers, and quickstart manuals:

```
doc_src/
├── drawlib.toml          # [project] + [i18n] manifes
└── en/                   # Locale chapter tree (-l en
    ├── 00_cover.md       # Title, subtitle, author me
    ├── 01_overview.md    # Chapter 1
    └── 02_design.md      # Chapter 2 (with ```drawlib
```

Multi-Format Publication Outputs from One Source

1. **Unified Standalone Web Document (`doc_html/<lang>/index.html`)**:
   - Merges all numbered `.md` chapters in sorted order into a per-locale `index.html` with an auto-generated **Table of Contents**, a centered **900px reading container**, and full `file://` portability.
2. **Print-Ready Vector PDF & Word (`doc_<lang>.pdf` / `.docx`)**:
   - `drawlib build pdf --toc` renders crisp vector A4 PDFs via headless Chromium, while `drawlib build docx` exports native editable MS Word documents.



16:9 Widescreen Stage Architecture (`1920x1080`)

Dual Coordinate Systems Working in Harmony

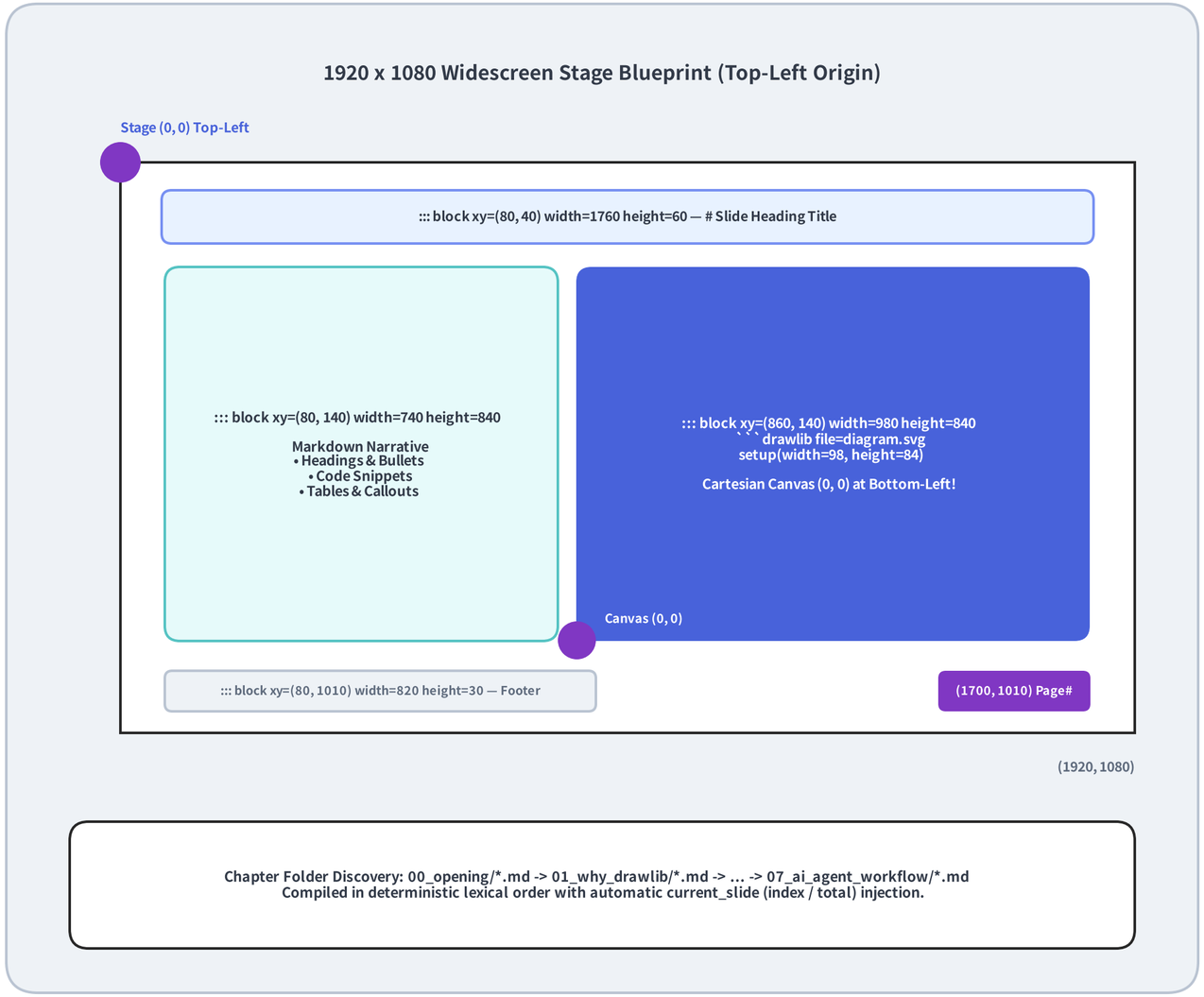
Drawlib slides (`drawlib init project slide`) combine pixel-accurate stage containers with Cartesian vector canvases:

1. **Slide Stage (`::: block xy=(x, y) width=W height=H`) — Top-Left `(0, 0)`**:
   - Fixed **1920 × 1080** pixel widescreen stage (automatically scaled to fit any browser window or projector).
   - Supports hierarchical chapter folders (`00_opening/`, `01_why_drawlib/`, ...) sorted alphabetically.
2. **Drawlib Canvas (`setup(width=W, height=H)`) — Bottom-Left `(0, 0)`**:
   - Inside any ```` ```drawlib ```` block, coordinates use standard Cartesian geometry where `(0, 0)` is the bottom-left corner.

Golden Aspect-Ratio Rule

Keep `setup(width=W, height=H)` proportional to the enclosing `::: block width` and `height` (typically `width / 10, height / 10`) so inline SVGs fill their containers with zero letterboxing:

- `::: block xy=(860, 140) width=980 height=840` → `setup(width=98, height=84)`
- `::: block xy=(0, 0) width=1920 height=1080` → `setup(width=192, height=108)`



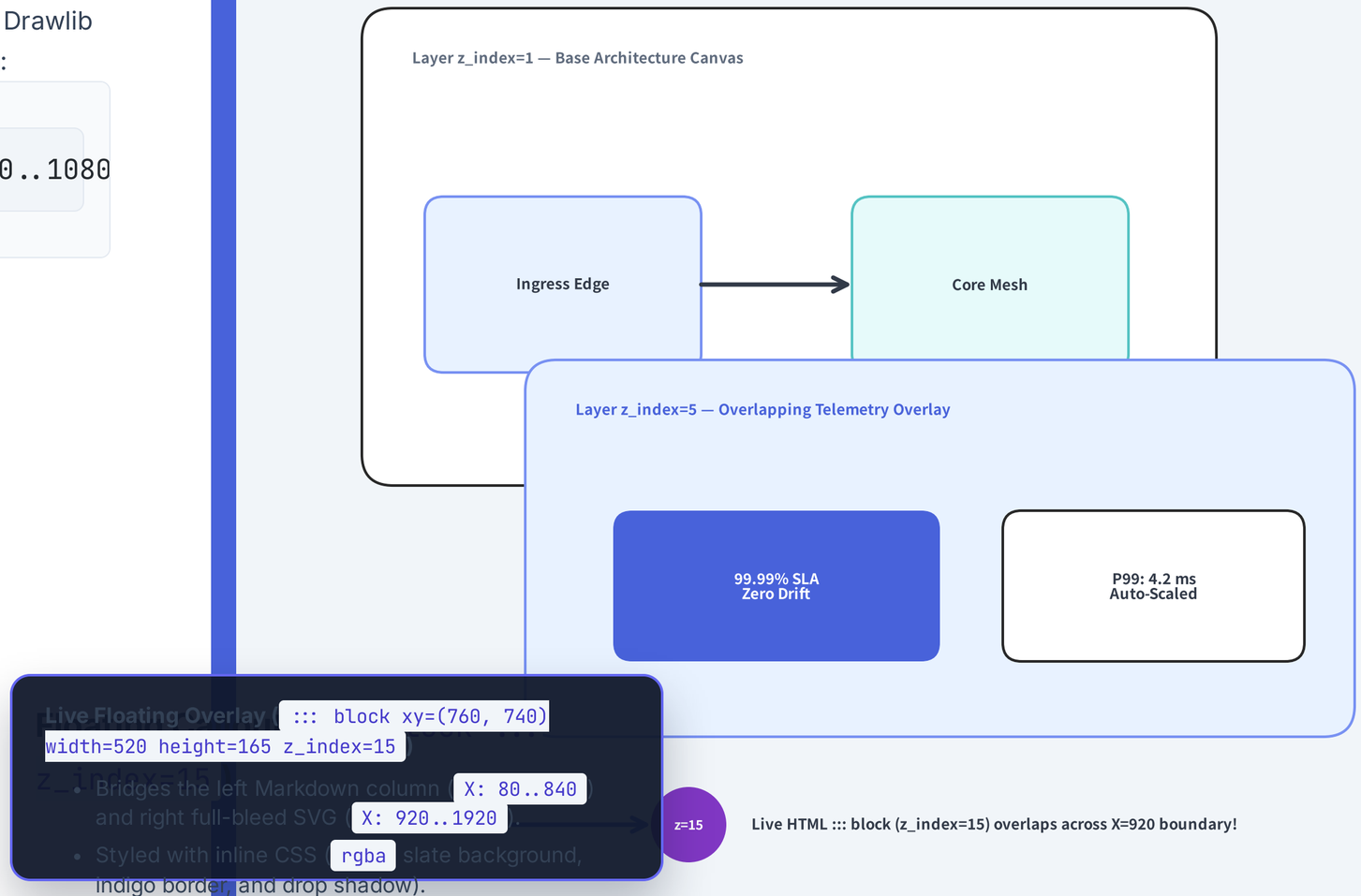
block xy=(920, 0) width=1000 height=1080 z_index=2 ```drawlib file=spatial_freedom.svg setup(width=100, height=108) `
blocks. - **`::: block` & `style="..."`**: Apply custom CSS backgrounds, shadows, or glassmorphism cards directly over ve

Spatial Freedom: Full-Bleed & Z-Index Layering

Breaking Free from Rigid Slide Templates

Traditional Markdown-to-slide tools lock you into centered bullet lists. Drawlib gives you **absolute pixel coordinates** plus `z_index=<layer>` stacking:

```
←!── Layer 1: Right-half full-bleed diagram (Y: 0..1080
```

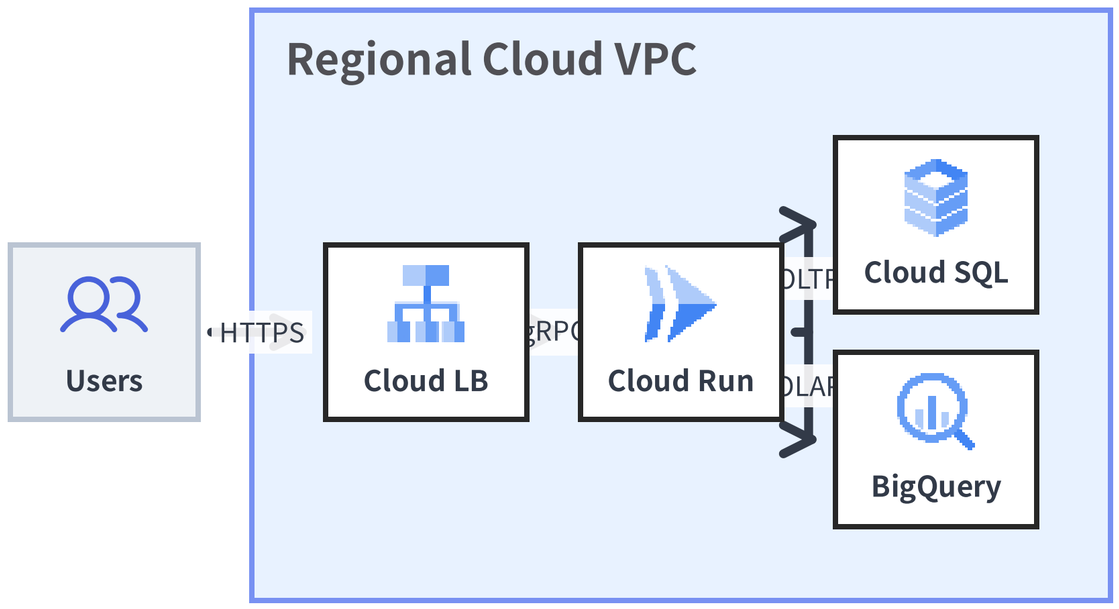


Full-Canvas Pure Stage — ::: block xy=(0, 0) width=1920 height=1080 + setup(width=192, heig

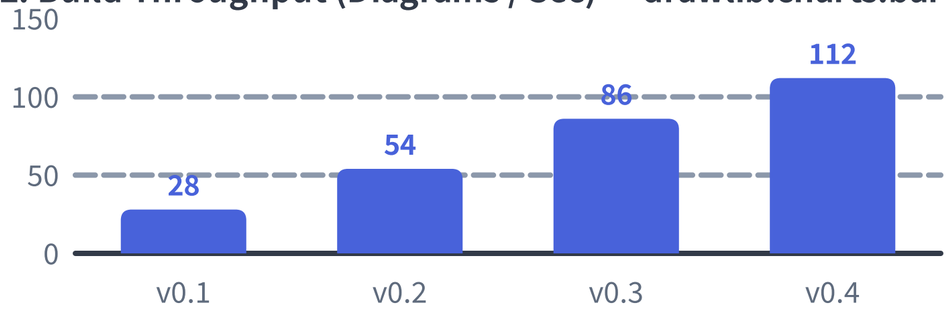
Pure SVG (55 / 64)

100% Programmatic Slide Design: Header Banner, Cloud ArchitectureDiagram, BarChart & KPI Cards on One Unified Vector Canvas

1. Cloud Topology (drawlib.diagrams.architecture)



2. Build Throughput (Diagrams / Sec) — drawlib.charts.bar



3. Programmatic KPI Badges & Cross-Zone Connectors

192 x 108
Cartesian Grid

100% Vector
Zero Pixel Blur

< 1 ms
Warm SQLite Hit

Project-Wide Consistency with `styles.py` & `utils.py`

DRY Presentation Engineering at Scale

Maintaining visual consistency across a 60+ slide deck in GUI tools is tedious and error-prone. Drawlib solves this with two shared Python modules in `slide_src/`:

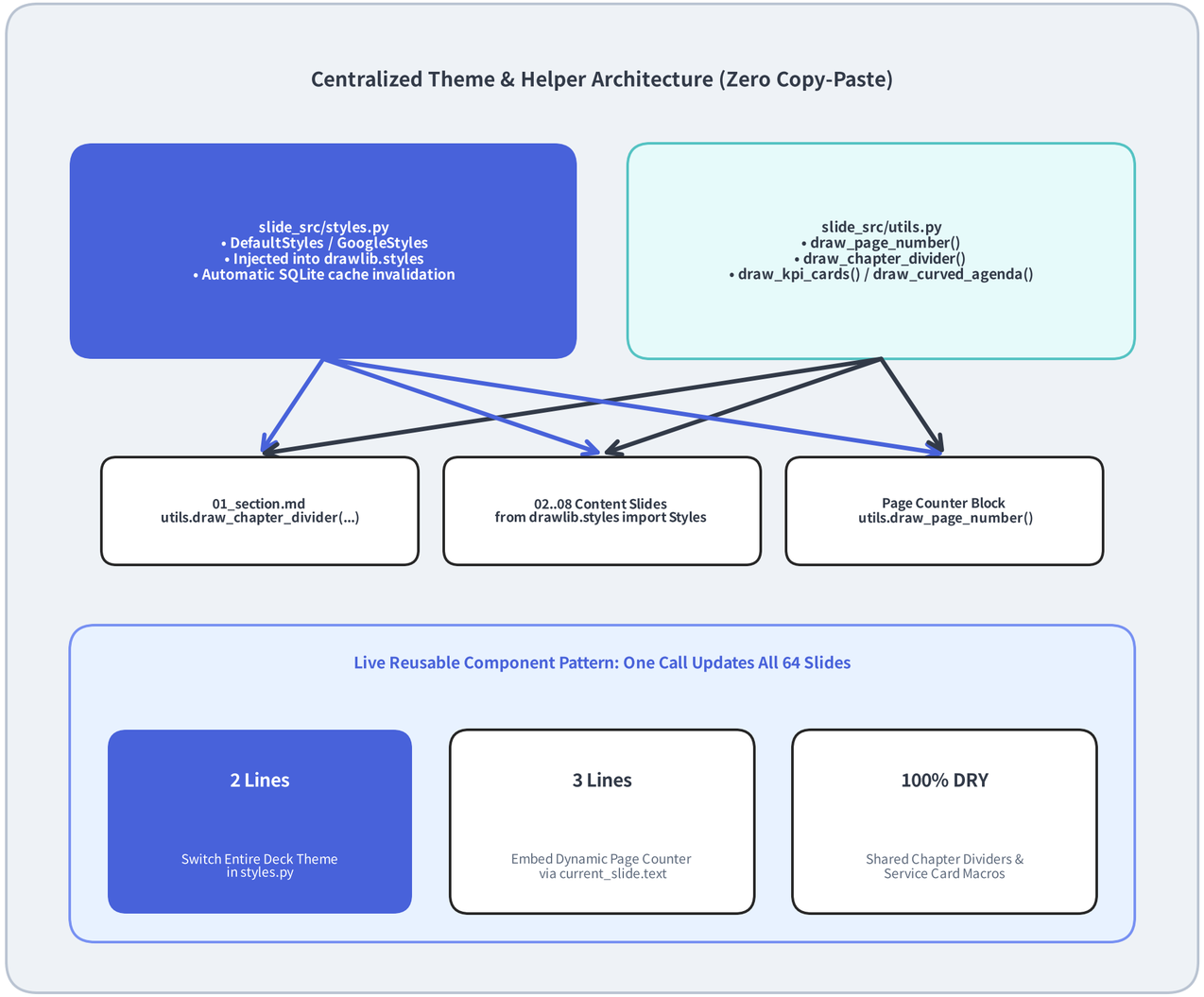
1. `styles.py` — Global Design Tokens

```
from drawlib.preset_colors import DefaultColors
from drawlib.preset_styles import DefaultStyles
```

Switching a 64-slide presentation from `DefaultStyles` to `GoogleStyles` or `MonochromeStyles` is a **two-line change** in `styles.py`.

2. `drawlib.slide` & `utils.py` — Reusable Slide Macros

- `drawlib.slide.draw_page_number()`: Renders `current_slide.text` (`"42 / 64"`) on a transparent canvas (`alpha=0.0`).
- `drawlib.slide.draw_section_header(...)`: Full-bleed 16:9 chapter cover (`split`, `centered`, `banner`, `minimal`) with progress dots and topic cards.
- `utils.py` **Domain Macros**: Project-specific helpers (`service_card()`, `connect()`, `draw_kpi_cards()`) callable in 3 lines of Python.

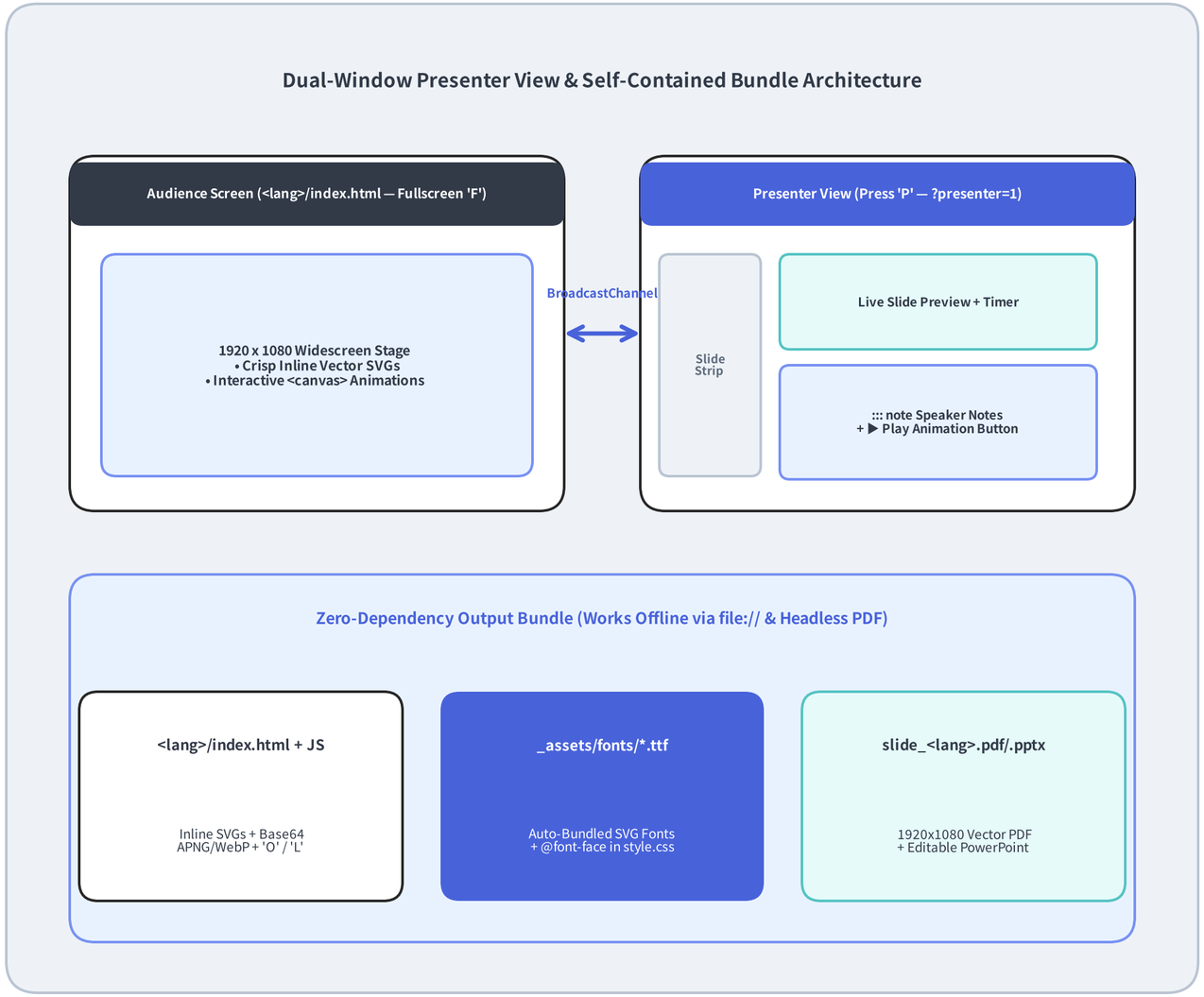


Presenter View, Overview Grid & Standalone Portability

Built for Live Stage Delivery & Offline Sharing

Compiled Drawlib slide decks (`slide_html/`) include a zero-dependency presentation runtime (`slide.js`) and self-contained assets:

Shortcut	Feature	Description
`P` / `S`	**Presenter View**	Opens dual-window `?presenter=1` synced via `BroadcastChannel` + `postMessage` .
`O` / `Esc`	**Overview Grid**	Interactive 16:9 thumbnail grid of all slides for instant navigation.
`A`	**Animation Step**	Plays, pauses, resumes, or resets `<canvas>` animations on the current slide.
`L`	**Language Switch**	Cycles multi-locale decks (`<lang>/index.html#N`) preserving slide number.
`F`	**Fullscreen**	Toggles native widescreen browser fullscreen mode.
`→` / `Space`	**Next Slide**	Advances slides (`←` / `Home` / `End` for navigation).



100% Standalone `file://` , PDF & PowerPoint Portability

CHAPTER

07

Part 7 of 7

AI Agent Workflow & Ecosystem

Built-in Agent Rules, Autonomous Visual Self-Healing & High-Speed Caching

7.1 Embedded Agent Knowledge Base (drawlib rules show)

7.2 The 6-Step Autonomous Visual Self-Healing Loop

7.3 Sub-Millisecond SQLite Incremental Build Cache

7.4 Live Dogfooding, CLI Reference & Getting Started

Built-in AI Agent Knowledge Base (`drawlib rules`)

Zero Hallucination via On-Demand Rules

Why do AI coding agents struggle with niche libraries? Because their training data is outdated or incomplete. Drawlib solves this by **shipping its complete AI agent manual inside the Python package**:

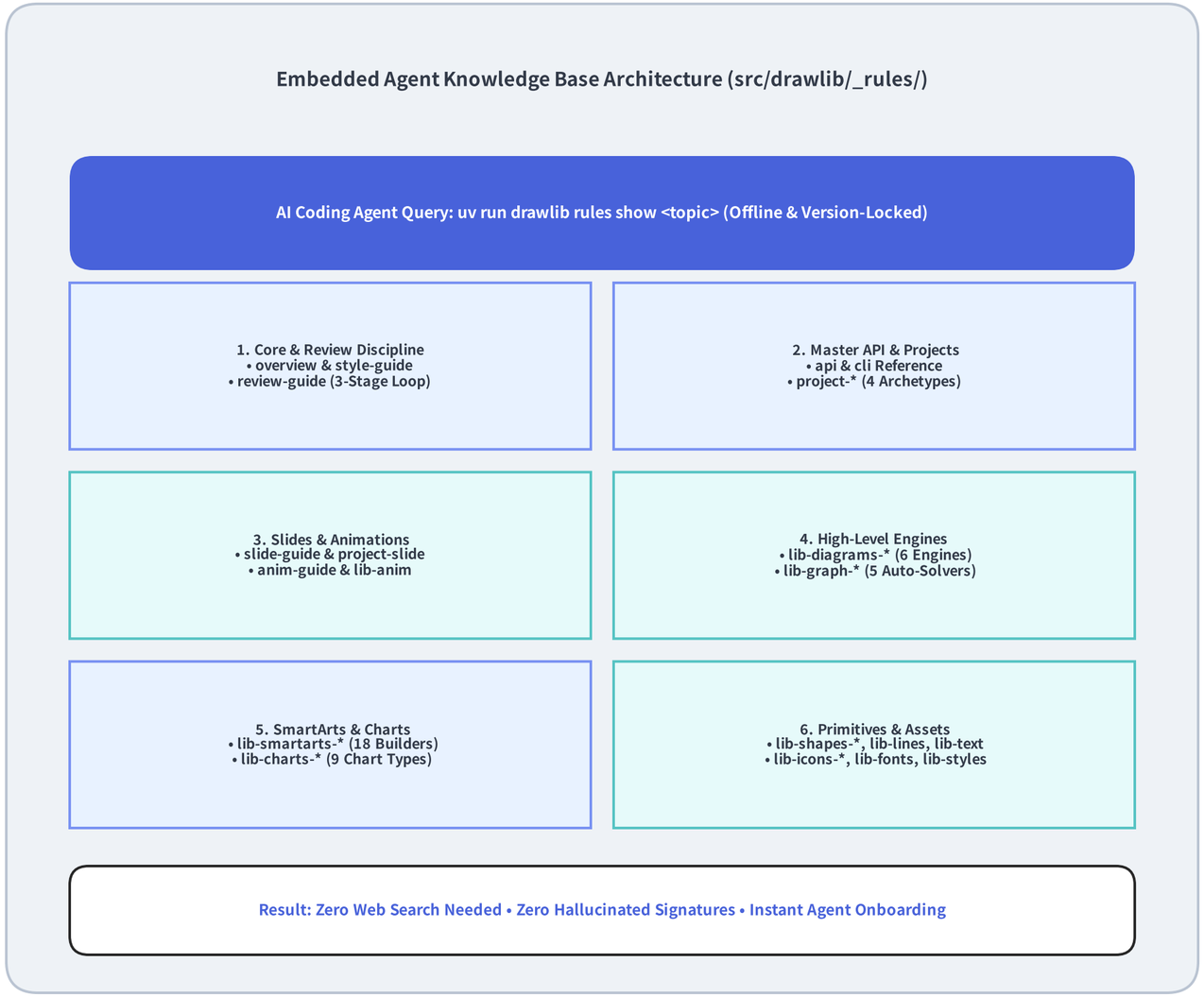
```
# 1. List all 92 embedded rule manuals
uv run drawlib rules list

# 2. Core orientation & design discipline
uv run drawlib rules show overview
uv run drawlib rules show style-guide
uv run drawlib rules show review-guide
uv run drawlib rules show api

# 3. Project archetypes & module deep-dives
uv run drawlib rules show project-overview
uv run drawlib rules show project-slide
uv run drawlib rules show anim-guide
```

Progressive Context Loading

- Agents start with a lightweight `.agents/rules/drawlib.md` bootstrap and query specific `lib-<module>` manuals on demand—keeping LLM context windows lean and 100% version-accurate.



Autonomous Visual Self-Healing Loop

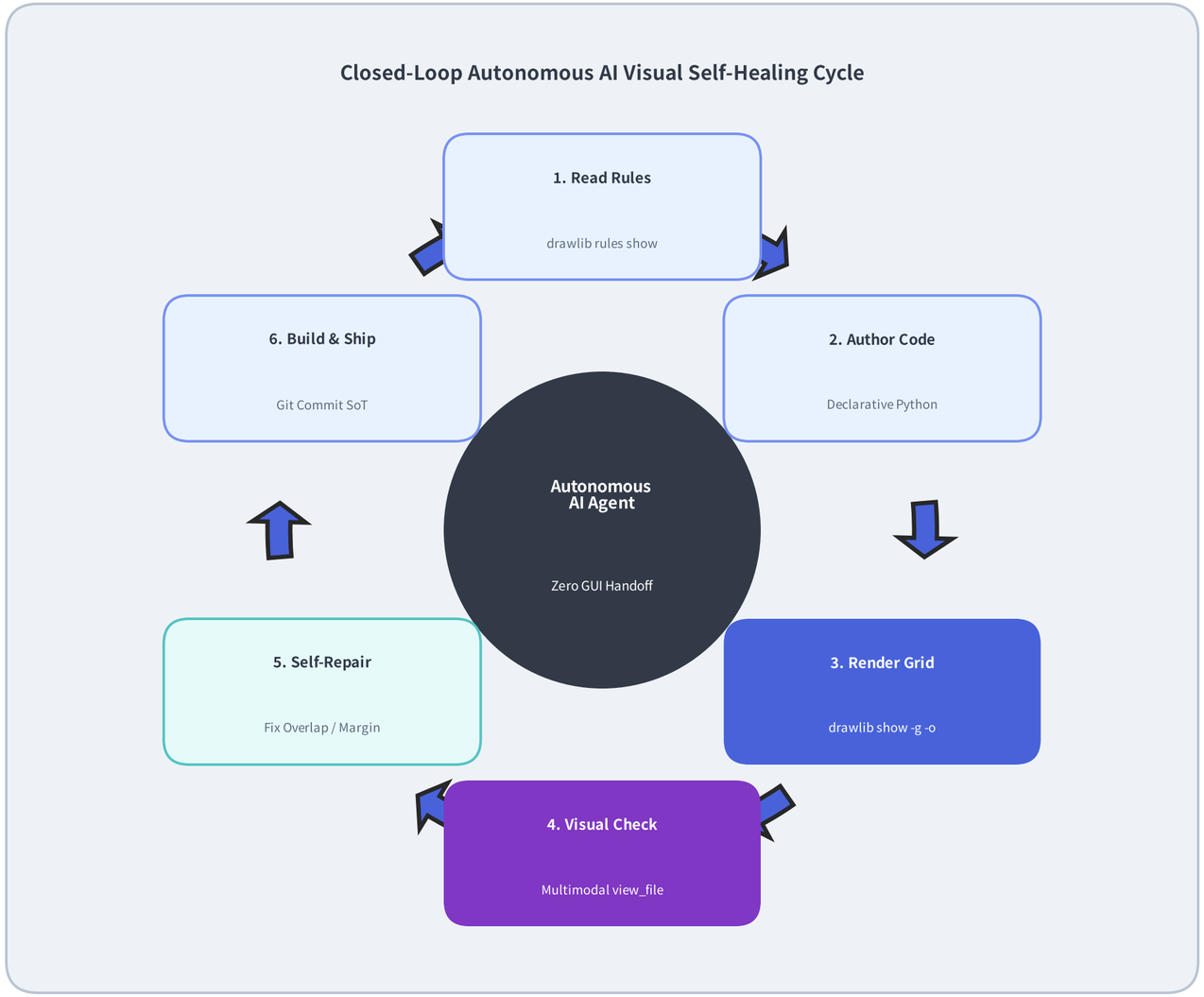
Closing the Loop with Multimodal Vision

Code that runs without syntax errors can still produce overlapping labels or cramped margins. Drawlib pairs **headless coordinate-grid rendering (`-g`)** with **multimodal agent vision (`view_file`)**:

```
# Render a single diagram block with a 10x10 coordina
uv run drawlib show docs_src/arch.md service_arch.png
    -g -o .drawlib/scratch/preview.png
```

The 6-Step Autonomous Agent Cycle

1. **Read Rules (`drawlib rules show`)**: Load coordinate & style rules.
2. **Author Declarative Python**: Write high-level diagrams, charts, or SmartArts.
3. **Render with Coordinate Grid (`-g`)**: Export headless preview image in `< 1s`.
4. **Multimodal Inspection (`view_file`)**: Agent visually inspects the rendered PNG + coordinate grid lines.
5. **Self-Repair Visual Defects**: Adjust exact `(x, y)` coordinates, padding, or contrast if any clipping is spotted.
6. **Commit Source of Truth**: Check clean Markdown + Python into Git.



Sub-Millisecond Incremental Builds (SQLite Image Cache)

Instant Rebuilds for Large Decks & Docs

A comprehensive technical site or 64-slide presentation can contain **100+ embedded diagrams and multi-frame animations**. Re-executing every Python block on every edit would slow down live authoring.

Deterministic SHA-256 Cache Key (`.drawlib/cache.sqlite3`)

Drawlib hashes four inputs before executing any ```` ```drawlib ```` block:

1. **Normalized Python Code Block**: Exact script source inside the fence.
2. **Global `styles.py` & `utils.py` Hashes**: Editing a shared helper or theme invalidates dependent diagrams automatically.
3. **Target Output Format**: `.svg` , `.png` (APNG), or `.webp` .
4. **Slide Context (`total_slides`)**: Ensures `current_slide.text` (`12 / 64`) stays accurate when slides are added or removed.

Cache Hit Performance

- **Cold Build (64 Slides)**: Renders all SVGs & APNGs from scratch (~14s).
- **Warm Cache Build (1 Slide Edited)**: Restores 63 unchanged slides from SQLite in **< 1 ms per diagram**—completing the full deck build in **~1.1 seconds**!



Live Dogfooding in the Drawlib Repository

Eating Our Own Cooking at 100% Scale

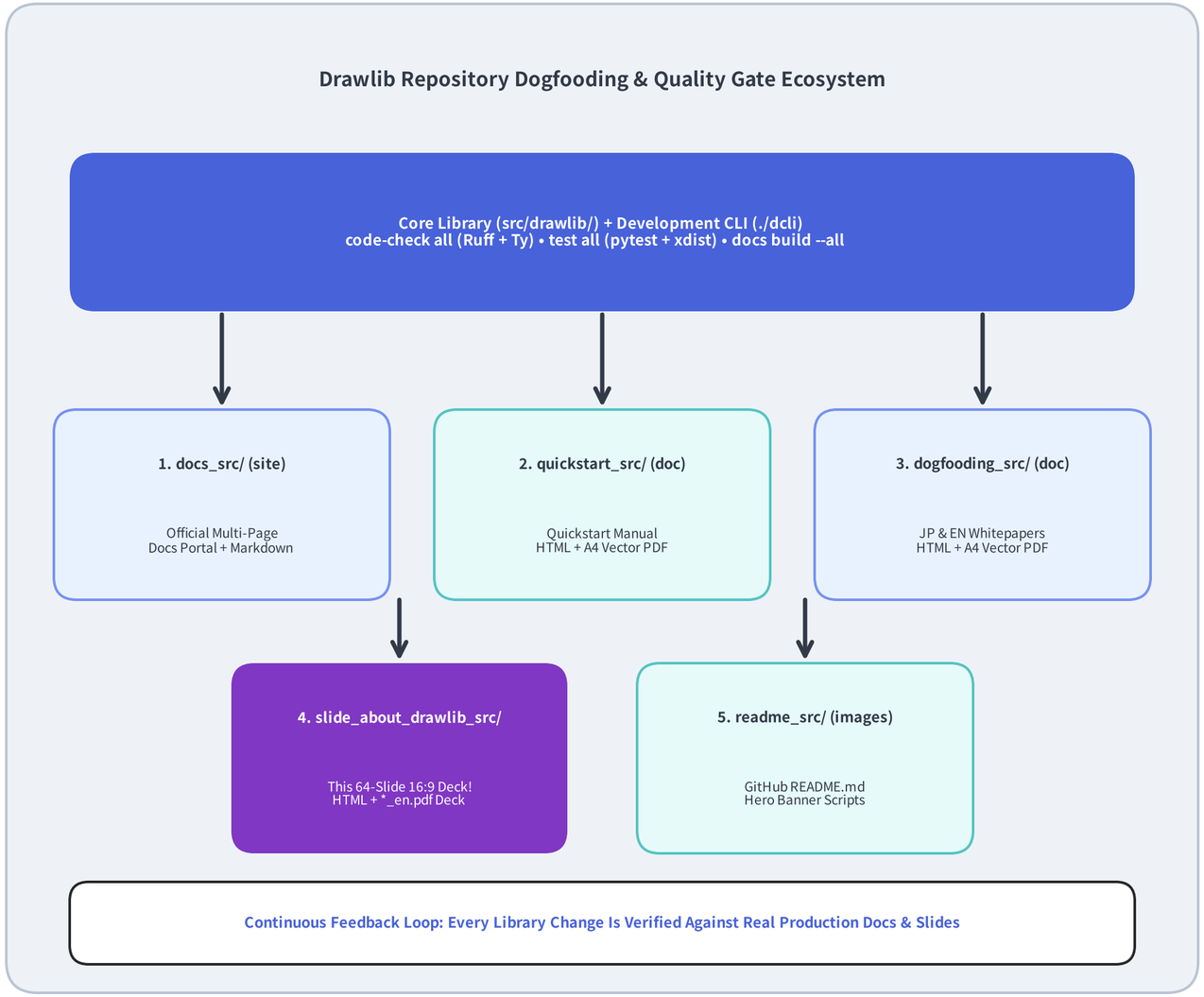
The `drawlib` repository is simultaneously the library codebase (`src/drawlib/`) and a **live production dogfooding environment** where every project archetype is continuously built and tested via `./dcli`:

```
# 1. Static analysis (Ruff lint/format, Ty type check
./dcli code-check all

# 2. Parallel unit & integration test suite
./dcli test all
```

Live Dogfooding Targets in `docs/`

1. `site` (`docs/docs_src/`): Bilingual (`en` / `ja`) documentation website (`docs_html/` & `docs_markdown/`).
2. `quickstart` (`docs/quickstart_src/`): Quickstart guide & vector PDF (`doc` archetype).
3. `dogfooding` (`docs/drawlib-dogfooding_src/`): Bilingual (`en` / `ja`) architectural whitepapers (`doc` + `_en.pdf` / `_ja.pdf`).
4. `slide` (`docs/slide_about_drawlib_src/`): This 64-slide 16:9 showcase presentation deck!
5. `readme` (`docs/readme_src/`): Standalone hero illustrations for `README.md` .



Complete CLI & Workflow Cheat Sheet

One Unified CLI (`drawlib`)

From initial project scaffolding to headless visual inspection and PDF / Office publishing, every workflow runs via `uv run drawlib`:

1. Scaffold & Author

```
uv run drawlib init project <site|doc|slide|image
uv run drawlib init agent <claude|cursor|antigrav
```

2. Inspect & Verify Headlessly

```
# Render single block with coordinate grid
uv run drawlib show page.md arch.png \
    -g -o .drawlib/scratch/preview.png
```

3. Compile & Export All Targets

```
./docs_src/build.sh
# Or export HTML, PDF, Word & PowerPoint:
uv run drawlib build html docs_src/ -o docs_html/
uv run drawlib build pdf doc_src/ -o doc.pdf
```

Drawlib Command-Line Interface Reference Matrix

CLI Command	Primary Purpose	Key Flags & Options
drawlib init project|agent	Scaffold project or AI agent rules	-s <theme>, -l <lang>, -m <mode>
drawlib build html	Compile Markdown to multi-page or doc HTML	-o <dir>, -s styles.py, --no-cache
drawlib build pdf	Export A4 or 16:9 vector PDF (auto Chromium)	-o <pdf>, --generate-index
drawlib build docx	Export editable MS Word (.docx) document	-o <docx>, --generate-index
drawlib build slide	Compile 1920x1080 widescreen HTML slide deck	-o <dir>, -f svg|png|webp
drawlib build pptx	Export editable 16:9 MS PowerPoint (.pptx)	-o <pptx>, -t <theme>, --dpi
drawlib build md / image	Build GitHub Markdown or batch Python scripts	-o <dir>, -g (grid), --no-cache
drawlib show / serve	Headless grid preview or HTTP + link scanner	show -g -o, serve --check
drawlib rules / cache	Query AI manuals or pre-fetch assets/browser	rules show, cache download --all

Start Drawing & Presenting as Code

Unify Your Code, Diagrams, Docs & Slides

Stop context-switching between Git, GUI drawing tools, wikis, and slide editors. Bring your entire visual communication stack into version-controlled Python and Markdown.

3-Step Quickstart

```
# 1. Install Drawlib (PDF, DOCX & PPTX built in)
uv add drawlib   # or: pip install drawlib

# 2. Scaffold a 16:9 slide deck, doc, or site
uv run drawlib init project slide my_deck -s default
```

Core Takeaways

- **Batteries Included**: PDF (`build pdf`), MS Word (`build docx`), and 16:9 PowerPoint (`build pptx`) work out of the box—headless Chromium auto-downloads on first build (`cache download --all` / `--browser` for CI; `playwright install-deps chromium` on minimal Linux containers).
- **AI-Native Source of Truth**: Diagrams, docs, and slides live in Git with `drawlib init agent` , `drawlib rules` , and headless grid review (`drawlib show -g`).
- **Explore & Contribute**: `https://github.com/yuichi110/drawlib`

