TECHNICAL WHITE PAPER · SDSTKSTUDIO

SDSTK Studio: A Native Swift Canvas for Visual Data Science and Compound Model Orchestration on Apple Silicon

A native Swift canvas for visual data science and compound model orchestration on Apple Silicon.

MLX-Native | Visual Programming | Mixture of Experts | MCP | iPad / macOS

ABSTRACT

SDSTK Studio is a universal SwiftUI application for iPadOS and macOS that brings visual-programming data science to Apple Silicon. Modeled on the interaction paradigm of Orange Data Mining, it lets a user construct a data-analysis pipeline by dragging widgets onto a canvas and wiring them together, watching results update live as the graph changes. What makes this possible architecturally is SDSTK: an MLX-native Swift reimplementation of the Python data-science stack — pandas, SciPy, scikit-learn, matplotlib, and beyond — running directly on Apple Silicon's unified memory and GPU with no Python interpreter, no server process, and no translation layer in between. Beyond its original scope as a data-science canvas, Studio has grown a second identity: the same node/port/link execution engine that schedules a group-by-and-scatter-plot pipeline turns out to be exactly the right substrate for composing independently trained Core ML models into a coordinated, mixture-of-experts-style system — and, through a companion MCP server, exposing that composed system as a callable tool for external LLM agents. This paper surveys the academic foundations of visual and dataflow programming, mixture-of-experts combination, and uncertainty-driven active learning; describes Studio's widget-contract architecture and live execution engine, including a dynamic-arity Coordinator, per-port selection propagation, and MLX-native neural-network training added in a subsequent hardening pass; catalogs its 48-widget capability surface across 12 categories, including the Experts category; and analyzes the economic and workflow case for a fully native, on-device visual environment that spans both data science and compound-model orchestration.

01 Introduction

The dominant interface to data science remains the script: a Python file or Jupyter notebook, executed top to bottom, with intermediate state held in interpreter memory that vanishes when the kernel restarts. This is a productive model for practitioners fluent in Python, but it is not the only viable one, and it is not obviously the best one for exploratory, iterative analysis where the *shape* of a pipeline — which transform feeds which model, which model feeds which evaluation — is often more important to reason about than any single line of code.

Visual programming environments for data science replace the script with a canvas: discrete processing widgets, wired together into a directed graph, executed automatically as the graph changes. Orange Data Mining [1] is the most established example of this approach, and has demonstrated over more than a decade of academic and industrial use that a well-designed widget canvas can cover a meaningful fraction of a data scientist's actual workflow — import, transform, visualize, model, evaluate — without requiring the user to write code at all.

SDSTK Studio adopts this interaction model natively on Apple platforms, built on a foundation Orange does not have available to it: SDSTK, an MLX-native [2] Swift data-science stack that runs pipeline computation

directly on Apple Silicon's unified memory architecture, with no Python interpreter and no attendant GIL-serialization overhead [3]. What began as a direct port of Orange's model onto that foundation has since grown a second, related identity. The same typed-port, dirty-marking execution graph that schedules a `CSVFile → ScatterPlot → LogisticRegression → TestAndScore` pipeline is, at the architecture level, indistinguishable from what a compound "mixture of experts" system needs: independently trained narrow models, wired into an orchestration graph, combined by a gating node. Studio's newest widget category, Experts, builds exactly that on top of the existing canvas — and a companion standalone server exposes the resulting composed graph to external AI agents over the Model Context Protocol [10]. This paper treats both identities as one continuous architectural story: a native, GPU-aware dataflow canvas that happens to be equally well-suited to classical data science and to compound-model orchestration, because both problems reduce to the same typed graph-scheduling primitive.

---

02 Background and Related Work

2.1 Visual Programming for Data Science

Orange [1] popularized the widget-canvas model for data mining: a user assembles a workflow from a palette of widgets — each performing one well-defined operation (load data, discretize a column, train a classifier, plot a scatter) — connected by typed input/output channels. The published toolbox emphasizes rapid prototyping and teaching, and its longevity (in continuous academic use since its JMLR publication) is evidence that the interaction model itself, independent of any particular underlying numerical library, has durable value for data-science workflows. SDSTK Studio's canvas, palette, and inspector layout are a direct application of this same interaction model to a new computational substrate.

2.2 Dataflow Execution Models

The formal underpinning of "wire widgets together and results update automatically" is dataflow programming: a program is expressed as a directed graph of nodes, and execution is driven by data availability at each node's inputs rather than by an explicit sequential control flow [4]. Dataflow languages and systems have a long history in both visual and textual form, and the central engineering problem they share is scheduling — determining a valid execution order for a graph, and re-deriving that order efficiently when the graph changes.

The classical solution to graph ordering is topological sort. Kahn's algorithm [5] computes a topological order by repeatedly removing nodes with no unresolved incoming dependencies, and is the standard approach for scheduling directed acyclic dataflow graphs in production systems. SDSTK Studio's `ExecutionEngine` applies Kahn's algorithm directly to the widget graph, combined with a dirty-marking scheme that limits re-execution, after an edit, to the transitive downstream closure of the changed node rather than the whole

graph — a memoization strategy that only re-runs a widget when neither its parameters nor its upstream outputs have changed.

2.3 MLX and the Apple Silicon Array Substrate

MLX [2] is Apple's open-source array framework for Apple Silicon, built around the Unified Memory Architecture (UMA) shared by the CPU and GPU on M-series chips — eliminating the host-device copy overhead that dominates GPU utilization time in discrete-GPU array frameworks. SDSTK, the package layer beneath Studio, treats MLX as filling the role NumPy plays in the Python stack, and builds pandas-, SciPy-, and scikit-learn-equivalent packages one layer above it, entirely in Swift.

This matters for a visual canvas specifically because interactive tools are latency-sensitive in a way batch scripts are not: a user dragging a slider or reconnecting a wire expects the canvas to recompute in the time it takes to notice the change, not the time it takes to re-run a script. Python's Global Interpreter Lock [3] serializes bytecode execution within a single interpreter process, which compounds with the interpreter's own dispatch overhead to make sub-frame-rate interactivity difficult to guarantee for anything beyond trivial pipeline sizes. A compiled, GPU-native execution substrate removes that specific class of latency risk at the architecture level rather than requiring case-by-case optimization.

2.4 Mixture-of-Experts Combination

The idea of combining several independently trained "expert" networks through a separate gating or combining mechanism predates the transformer-era Mixture-of-Experts layer by decades. The original formulation [6] trains a set of local expert networks alongside a gating network that learns to weight each expert's output — a supervised learning procedure explicitly designed for systems composed of many separate networks, each responsible for a subset of the input space, with no fixed limit on how many experts participate. This is the architectural lineage Studio's `Experts.Coordinator` widget follows directly: a combiner over independently trained, independently owned expert outputs, rather than the routing-inside-one-network mechanism used by modern sparse transformer MoE layers. The distinction matters because Studio's experts are not co-trained — each is a standalone Core ML model, potentially trained by a different tool (ModelBuilder) at a different time, and the Coordinator's job is purely post-hoc combination. Studio's own Coordinator initially shipped with a fixed two-expert arity before being generalized to a variable count (§4.2), which brings the implementation into closer alignment with the original formulation's arbitrary-N gating network rather than further from it.

2.5 Reject Options and Uncertainty-Driven Active Learning

A classifier that must commit to a label for every input will, at some error rate, be wrong on inputs it should instead have deferred on. The formal treatment of when a classifier should abstain rather than guess — trading a higher reject rate for a lower error rate on the inputs it does commit to — is a foundational result in pattern recognition [7]. Studio's `Experts.MemStacker` widget operationalizes a version of this reject option: rather than discarding low-confidence predictions, it routes them into a review queue for a human to label, and accumulates enough human-labeled examples per category to trigger an automatic retraining pass. The

specific uncertainty signal used — the margin between an expert's top prediction and its runner-up, rather than raw top-1 confidence alone — follows the uncertainty-sampling tradition in active learning [8], where margin has been established as a more reliable indicator of genuine ambiguity than confidence in isolation, since a model can be confidently wrong in a way raw confidence alone does not surface.

03 Architecture

3.1 The SDSTK Foundation

Every widget in Studio is a thin SwiftUI layer over an independently versioned, independently testable SDSTK package; nothing in the application re-implements data-science logic, it only exposes it.

SDSTK module	Replaces	What it gives Studio
Frame	pandas	Columnar `DataFrame`, group-by, joins, windows, nulls, CSV/Arrow/Parquet I/O — swappable `.cpu` / `.mlx` numeric backend
Sci (`LinAlg` + `Stats`)	SciPy	Linear algebra (BLAS/LAPACK via Accelerate), descriptive statistics, hypothesis tests, distributions
Learn	scikit-learn	Linear/logistic regression, decision trees, ensembles, k-means, PCA, cross-validation
Neural	PyTorch (training loop)	MLX-native MLP/CNN/RNN/LSTM training with a compiled train loop — wired into Studio as `Model.MLPRegressor` (§6)
Plot	matplotlib	Headless SVG rendering — Studio's export path, not its live canvas view
Signal	`scipy.signal`	FFT/STFT, windowing, filter design, peak detection
Text	spaCy / NLTK	Tokenization, TF-IDF, cosine similarity
Optimize	`scipy.optimize`	Levenberg–Marquardt curve fitting, gradient descent, Nelder–Mead
TimeSeries	`statsmodels.tsa`	Rolling statistics, ACF/PACF, ARIMA, exponential smoothing
Formulas	—	Closed-form physics/chemistry/biology equations, with no single Python equivalent reference
Graph	NetworkX	Dijkstra, A*, minimum spanning tree, connected components

Studio depends on these as local Swift Package Manager dependencies rather than a frozen vendored copy — a deliberate choice that keeps the application always building against SDSTK's live source. The Experts category (§4) sits outside this table by design: it does not wrap an SDSTK module at all, but instead wraps

externally trained Core ML models via Apple's own Vision and Foundation Models frameworks, discussed separately in §5.

3.2 The Widget Contract

Every widget conforms to a single `StudioWidget` protocol: a stable type identifier, a category, declared typed input and output ports (`.table`, `.classifierLearner`, `.regressorLearner`, `.scores`, `.chart`, and — added for the Experts category — `.image`, `.prediction`, `.text`), a `run(inputs:)` function, and a SwiftUI inspector view for its parameters. Ports are typed deliberately — a regressor cannot be wired into a slot expecting a classifier, and the error is caught by the compiler rather than surfaced as a runtime failure mid-pipeline. Adding three new port kinds for Experts required no change to this contract; the type system that keeps a `DataFrame` from being wired into a `Figure` slot generalizes without modification to keeping an image from being wired into a text slot.

3.3 Live Dataflow Execution

Studio's `ExecutionEngine` tracks the workflow as a directed graph, computes execution order via Kahn's algorithm [5], and re-runs only the subgraph a given edit actually invalidates. Editing a widget's parameters does not require an explicit "Run" action; the graph updates itself, matching the always-live feel of Orange's own canvas [1] while running on a substrate where the recomputation itself is materially cheaper.

3.4 Document Model

A data-science workflow is a single `.sdstkflow` file — JSON describing every widget's type, position, and parameters, plus the links between them — opened through SwiftUI's `DocumentGroup`, so it behaves like any other document on either platform: Open Recent, iCloud Drive, drag-and-drop. A second document type, `.mbexpert` (§4.4), extends this model to graphs that embed trained model weights alongside the graph description.

3.5 A Concurrency Decision Worth Naming

SDSTK's `DataFrame` and `Figure` types wrap MLX/Metal state that is not `Sendable` under Swift's structured-concurrency model [9], so Studio's graph and execution engine are deliberately pinned to the main actor. SwiftUI's `ReferenceFileDocument`, however, declares its requirements as nonisolated and `Sendable`-constrained — a genuine conflict between "this state must stay on one thread" and "this protocol assumes it might not." The resolution: the document class conforms as `@unchecked Sendable`, and every protocol-required method wraps its body in `MainActor.assumeIsolated`, converting an invariant the type system cannot verify statically into a runtime-checked guarantee rather than a silent data race. The same pattern recurs, with a variant, in the Experts widgets: `MLModel.prediction` on macOS is only available as an `async` overload, and a loaded model held in `@MainActor`-isolated widget state trips Swift's sending-risk check even under single-owner use the compiler cannot itself prove safe — resolved with `nonisolated(unsafe)` on that one binding, confirmed against the actual `CoreML` module interface rather than assumed from documentation.

3.6 Selection Propagation

A single-output-per-widget contract cannot express Orange's "Selected Data" pattern, where a user lassoes points in a chart and only that subset flows to whatever is wired downstream. Studio's engine gained a minimal, additive extension for this: a `PortValue.outputs([String: PortValue])` case that `gatherInputs` unwraps per-link by source port name, so downstream widgets never see the wrapper and single-output widgets required no migration at all. `Visualize.ScatterPlot` is the first widget built against it, adding a second `selected` `.table` output alongside its existing chart output, with drag-marquee selection available directly in its own inspector. The selection is persisted as a data-coordinate rectangle rather than raw row indices — Orange's own choice — so that if the upstream data is re-sampled or re-filtered, the same region re-selects whatever rows now fall inside it instead of silently pointing at rows that have since shifted or disappeared. An empty selection produces an empty table rather than nothing at all, the closest honest match to Orange's "sends nothing" behavior available in an engine where every declared output must produce a value. Selection propagation currently exists on `ScatterPlot` alone; extending it to Studio's other Visualize widgets is now a mechanical follow-up rather than a design problem, since the engine-level plumbing is in place.

---

04 The Experts Category: Compound Model Orchestration

4.1 Motivation

ModelBuilder, a sibling application in this family, trains one Core ML model at a time — an image classifier, a tabular regressor — and stops at a working `.mlpackage` file. A recurring pattern in applied machine learning is that several such narrow, single-purpose models are more useful combined than any one of them is alone: a "mixture of experts" in the classical sense [6], where independently competent specialists are arbitrated by a combining mechanism rather than replaced by one larger generalist model. Studio already has the exact primitive this requires — a typed node/port/link graph with a live execution engine — so rather than building a separate orchestration tool, the same canvas that composes `CSVFile → Scatter → Regression` composes `CoreMLExpert → CoreMLExpert → Coordinator`.

4.2 The Widgets

`Experts.CoreMLExpert` loads an externally trained Core ML model — typically exported from ModelBuilder's model library — through the same security-scoped-bookmark file-picker pattern used by every other `Data.*File` widget. It takes an `.image` input and produces a `.prediction` output, running inference through Vision's `VNCoreMLRequest`, which handles image resizing and pixel-buffer preparation automatically and normalizes both classifier-shaped (`VNClassificationObservation`) and regressor-shaped (`VNCoreMLFeatureValueObservation`) model outputs into one common `ExpertPrediction` type. The

compiled `VNCoreMLModel` is cached after first load, since model compilation is a genuinely slow blocking call and only a freshly picked model should pay that cost.

`Experts.CoreMLTabularExpert` is the table-in counterpart, for ModelBuilder's other working trainer (Create ML tabular regression). It takes one selectable row from an incoming table as a single instance's feature values, auto-matching feature columns to the loaded model's declared input names — training preserves original CSV column names through to the model's input schema, so no manual mapping UI is needed — and raises a specific, named error if the table is missing a column the model expects, rather than guessing.

`Experts.Coordinator` is the gating/combining node, taking a variable number of `.prediction` inputs — two to eight experts, each with its own adjustable weight — and producing one combined `.prediction`, via either highest-confidence-wins or a weighted average restricted to experts carrying a numeric value (a classifier-shaped expert with no numeric value is skipped in that mode, not coerced into a meaningless number). This is architecturally the gating network from the original mixture-of-experts formulation [6], applied post-hoc to independently trained, independently owned models rather than co-trained ones.

Coordinator shipped initially with a fixed two-expert arity: `StudioWidget.inputPorts` is declared `static`, one port list per widget *type*, which has no way to express "this many ports, chosen per node instance." The fix generalizes the contract rather than special-casing Coordinator: new `dynamicInputPorts` / `dynamicOutputPorts` protocol members default to forwarding the existing static declarations, so none of Studio's other roughly 45 fixed-shape widgets needed any change, while the seven call sites that actually consult a widget's ports — `WorkflowGraph.gatherInputs`, `NodeView`'s port-dot layout and node height, and `CanvasView`'s link-completion and hit-testing — now read the live instance instead of the type. Coordinator overrides both to generate `expert1…expertN` ports from its own `expertCount` parameter (renamed from the original `expertA` / `expertB`, safe to rename since nothing had shipped against the old names), with per-slot weights that survive shrinking the count and restore themselves if the count grows back. A link left dangling by a shrink is benign by construction — link rendering already guards on port lookups succeeding, `gatherInputs` only ever iterates the current port set, and the link reattaches on its own if the node regrows to include that slot again. Verified against the real combine logic with a genuine three-expert vote (the third expert wins at equal weight, as the highest-confidence rule requires).

`Experts.ResearchExpert` is a different kind of expert — a general-knowledge node backed by Apple's on-device Foundation Model rather than a narrow, closed-domain Core ML model. It takes `.text` in and produces `.text` out, checking the system language model's availability before running rather than assuming Apple Intelligence is enabled on every device. Live web search or tool-calling for this node is deliberately deferred: the underlying session API supports a `tools:` parameter, but what a search result should look like as a typed port value is its own scoped design problem.

`Experts.MemStacker` closes the loop between low-confidence predictions and better-trained experts. Rather than raw top-1 confidence, it flags a prediction as uncertain using margin — the gap between an expert's top prediction and its runner-up [8] — since a model can be confidently wrong in a way raw confidence does not surface, and margin has been established as the more reliable uncertainty signal in the active-learning literature. Flagged predictions are filed for human review one at a time in the widget's own inspector; a confirmed label promotes the example into a per-category staging folder, and once a label's folder crosses a

promotion threshold (default 20 examples) *and* at least one other category also has staged examples, a background retraining pass fires automatically, producing a genuine new Core ML classifier via the same `ImageFeaturePrint` + `FullyConnectedNetworkClassifier` pipeline ModelBuilder itself uses. The retrained model is not auto-wired back into the canvas as a new node — mutating live graph structure from an asynchronous background completion was judged a materially different, riskier class of side effect than anything else the engine does, so promoting a freshly trained expert into the workflow remains a deliberate, manual step.

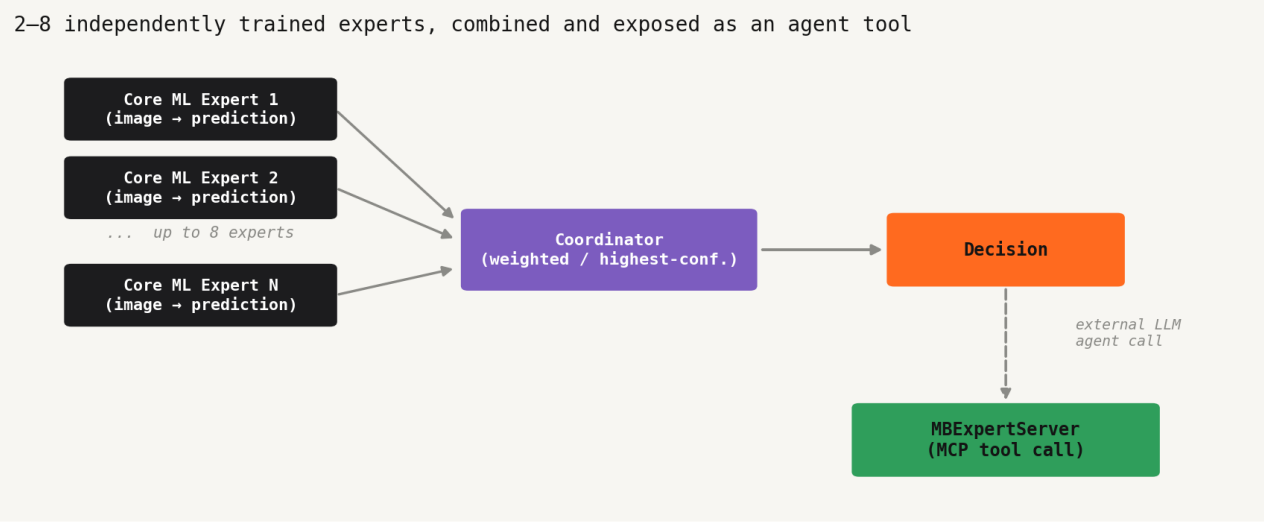

Figure 1. Two to eight independently trained Core ML experts, combined by the Coordinator, and exposed as a callable tool via MBExpertServer.

4.3 New Port Kinds

Three port kinds were added to support Experts: `.image` (backed by `CGImage`, carried onto the canvas by a new `Data.ImageFile` source widget), `.prediction` (a small structured type carrying label, confidence, runner-up confidence, numeric value, and source-widget name — scoped to exactly what ModelBuilder's two working trainers produce, not a speculative generic schema), and `.text` (backed by `String`, carried onto the canvas by a new `Data.TextPrompt` source widget). Each generalizes cleanly to future use beyond the widget that introduced it — `.text`, for instance, is immediately reusable by any future text-in/text-out node, not hard-coded to `ResearchExpert` alone.

4.4 The `.mbexpert` Document Format

A `.sdstkflow` graph referencing a Core ML Expert is only as portable as the security-scoped bookmark to wherever that model file happened to live — which breaks the moment the graph moves to another machine, or the source model file moves or is deleted. `.mbexpert` solves this by packaging the graph and its models together: a directory bundle (the same mechanism `.rtfd` and `.pages` use) containing `graph.json` — the same JSON the `.sdstkflow` format already uses — plus an `Experts/` folder holding a copy of every `CoreMLExpert` node's model file, keyed by node ID. On open, embedded models are copied out to

Application Support (mirroring ModelBuilder's own model-store pattern, since Core ML needs a real on-disk URL to compile from, not an in-memory file wrapper), and each expert widget rebinds to its own local copy. No new model format is introduced — every embedded model stays a plain `.mlmodel` or `.mlpackage`; only the bundle around them is new.

---

05 MBExpertServer: Exposing a Composed Graph as an Agent Tool

A trained, composed expert graph is only useful inside Studio's own canvas unless something can call it from outside. `MBExpertServer` is a standalone Swift package — deliberately independent of Studio's own application code, so that making the canvas reusable by a second consumer never requires restructuring code that already ships — that loads an `.mbexpert` bundle from the command line and serves it over the Model Context Protocol [10], the open standard for connecting AI systems to external tools and data sources.

The server initially shipped scoped to a single-expert bundle, with multi-node graphs marked as future work. That gap has since closed: `ExpertBundle` now parses both the nodes and the links in a `.mbexpert` bundle's `graph.json`, loads every expert the bundle contains, and decodes the Coordinator's persisted strategy and per-slot weights — mapping each link into port `expertN` to `weights[N-1]`, mirroring `CoordinatorWidget`'s own weight-lookup logic exactly, so the server's combination behavior cannot silently drift from the app's. A `tools/call` request runs every expert in the bundle; a failure in any single expert becomes an `error` entry in that expert's evidence rather than failing the whole call, and the final combined decision is produced by the same highest-confidence or weighted-average logic Coordinator itself implements. The server speaks newline-delimited JSON-RPC 2.0 over stdio: `initialize`, then `tools/list` (one tool per bundle, named after its primary expert), then `tools/call`, which accepts an image path and returns the combined prediction as MCP tool-result content. The JSON-RPC layer is hand-rolled rather than built on a registry-fetched SDK — both because this development environment cannot reliably reach a remote package registry, and because JSON-RPC's `id` field can legitimately be a number, string, or null, a dynamic shape that a loosely typed `[String: Any]` representation handles more directly than `Codable`'s static typing would.

Verification here closed a caveat that had stood since the server's first working version: a compound bundle — two experts plus a weighted Coordinator, exported by the app's own real export code — served correctly over genuine JSON-RPC, with weights of 1.0 and 0.5 arriving intact through the link-to-slot mapping; and, separately, a model *actually produced* by `MemStacker`'s automatic retraining loop (§4.2), rather than a hand-authored placeholder, was bundled and served, and correctly classified a real staging-queue sample as `"red"` through a live `tools/call` — the first time a genuinely trained-in-Studio model had been exercised end-to-end through the server rather than a synthetic or dummy one.

This closes a loop implicit in the rest of the application family: ModelBuilder trains a narrow expert, Studio composes and packages it (now including MemStacker's own auto-trained experts), and MBExpertServer makes the composed result callable by an external LLM agent — Claude Desktop, Cline, or any other MCP-speaking client — without that agent needing any awareness of Core ML, Vision, or Swift at all. What remains

deferred: the server still assumes at most one Coordinator per bundle and ignores any non-expert node it encounters, the tabular-expert case is not yet wired into the server's own bundle parser, and no live end-to-end test against a real external MCP client has been run yet — this phase establishes that the protocol, multi-expert combination, and bundle-loading plumbing all work correctly against real inputs, not that an external client is happy with it.

06 Capability Survey: 48 Widgets, 12 Categories

Studio ships 48 widgets across 12 categories. The first four categories are direct parity with Orange's own core surface (data import/transform, matplotlib-equivalent visualization, scikit-learn-equivalent modeling and evaluation); six categories — Signal, Text, Time Series, Optimize, Formulas, and Graph — have no native equivalent in Orange at all; and the twelfth, Experts, has no equivalent in Orange or, to this paper's knowledge, in any comparable visual-programming data-science tool, since it is not a data-science category in the traditional sense but a compound-model-orchestration one built on the same canvas.

Category	Widgets	Highlights
Data	19	CSV/Parquet import, a bundled Iris dataset, table preview, column selection, sampling, imputation, normalization, feature construction, table joins, outlier detection, schema summary, CSV export, a backend speed benchmark, a synthetic-signal generator, a bundled text-sample corpus, a bundled US-city route table, an image file source, and a text-prompt source
Visualize	6	Scatter (now with drag-marquee selection propagation, §3.6), histogram, bar, box plot, correlation heatmap, univariate feature ranking — native Swift Charts, not the headless SVG renderer
Model	7	Logistic and linear regression, decision trees, random forests, gradient boosting, and an MLX-native MLP regressor (§7) trained on-device via Adam [15]
Evaluate	3	Cross-validated Test & Score (classifier and regressor), confusion matrix from a real held-out split
Unsupervised	2	k-means, PCA
Signal	1	FFT magnitude spectrum
Text	1	TF-IDF cosine-similarity heatmap
Time Series	1	Autocorrelation
Optimize	1	Levenberg–Marquardt curve fitting
Formulas	1	Closed-form equation sweep (currently projectile motion)
Graph	1	Dijkstra shortest path over a weighted edge list

Category	Widgets	Highlights
Experts	5	Core ML image and tabular experts, a mixture-of-experts coordinator, an on-device language-model research expert, and an active-learning expert-growth loop

That distribution is the point of the comparison, not an incidental fact about it: Studio is not cloning Orange, it is using Orange's proven interaction model to demonstrate what a visual canvas looks like when the underlying stack extends past pandas and scikit-learn into signal processing, symbolic-adjacent scientific formulae, graph algorithms, and — furthest from Orange's own scope — compound orchestration of independently trained models.

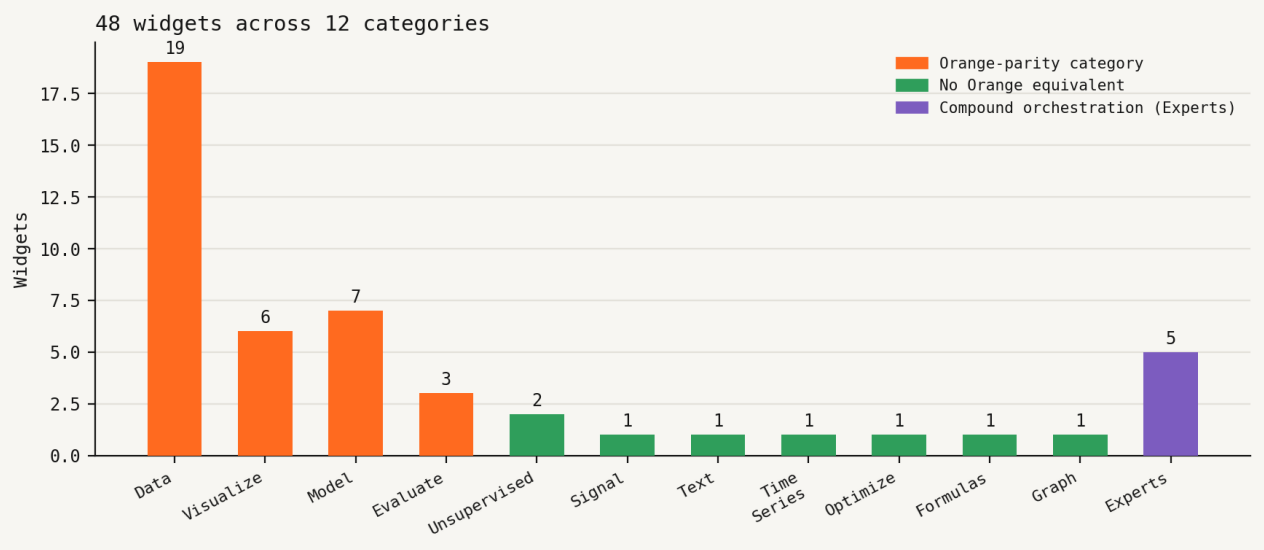

Figure 2. Widget count by category. Data, Visualize, Model, and Evaluate are direct Orange parity; six categories have no Orange equivalent; Experts has no equivalent in any comparable tool.

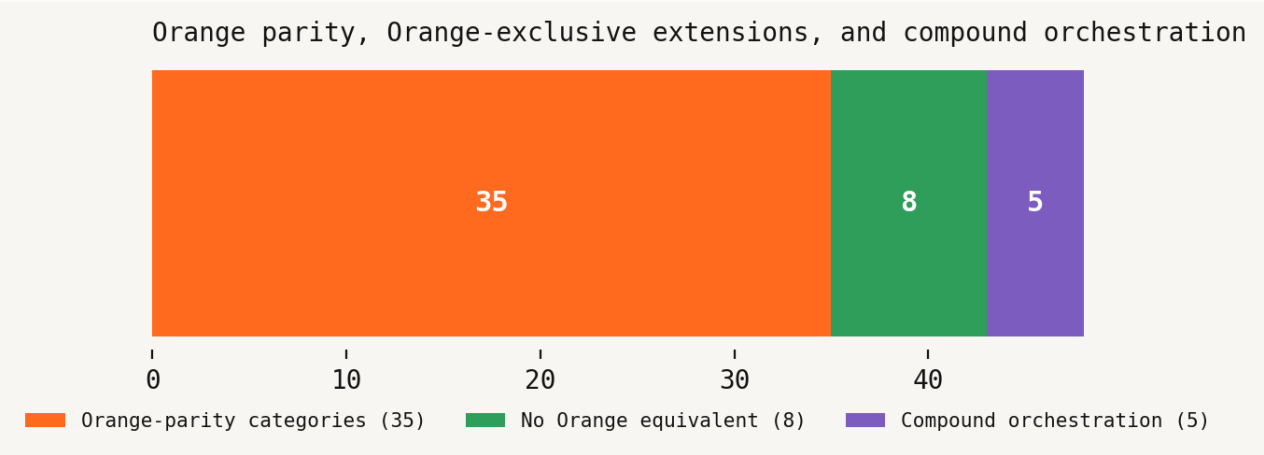

Figure 3. Of Studio's 48 widgets, 35 are direct Orange parity, 8 extend past Orange's own scope, and 5 build the new Experts orchestration layer.

07 Performance Analysis: Making the Backend Tradeoff Visible

Frame's numeric columns run on a swappable backend — `.mlx` (GPU, via `MLXArray`) or `.cpu` (portable, pure Swift). Rather than asserting a single performance number, Studio's **Backend Benchmark** widget makes the CPU/GPU tradeoff empirically visible on the canvas itself: it times the identical `Column` construct-and-sum operation on both backends over a user-configurable row count (up to millions of rows) and charts the result directly. The consistent, honest finding — corroborated by SDSTK's own internal Frame-vs-pandas benchmark suite — is that the GPU backend wins decisively on genuinely compute-bound work, but is not automatically faster on memory-bound operations like a plain column sum at moderate sizes, where per-dispatch and host/device-conversion overhead can exceed the trivial compute itself. This is presented to the user as a real, reproducible result rather than a marketing claim, and it directly informs why Studio pins the rest of the application to the `.cpu` backend by default and treats `.mlx` as an opt-in path gated on the Metal shader library's confirmed presence — a guard written against an observed failure mode (an immediate, clearly logged process exit when the shader library is absent) rather than an assumption about it.

`Model.MLPRegressor` is the newest widget to exercise this same GPU path for genuine compute rather than a benchmark demonstration: a table-in regressor that trains an MLX-compiled multilayer perceptron — configurable hidden layers, epoch count, and learning rate — via `Neural.train`, using the Adam optimizer [15] against mean-squared error, and outputs the per-epoch loss curve as a chart rather than a single held-out score, so the user can see whether training actually converged. It follows the same `metallibAvailable`-guard-before-touch discipline as Backend Benchmark, since MLX exits the process rather than throwing a catchable error when its shader library is absent, and it is deliberately a self-contained train-and-report widget rather than shoehorned into the `RegressorSpec`/Test & Score path the other Model widgets share — `Neural`'s MLX training loop has no equivalent to `Learn`'s `crossValScore`, and forcing one would be abstraction for its own sake rather than a genuine simplification. It was verified with real GPU training rather

than a typecheck-only pass: trained against a synthetic `y = 2x + 1` target, loss fell from 120.78 to 0.20 over 30 epochs, one real chart point per epoch. That same verification pass independently reconfirmed the original Backend Benchmark finding above: the first version of the standalone check harness crashed on the exact MLX hard-exit this section describes, because the harness — unlike the shipping app — never runs the startup code that pins the default backend to `.cpu`; mirroring that one line of app startup logic into the harness fixed it, which is itself a small, independent confirmation that the underlying failure mode this guard exists for is real and not merely a documented concern.

---

08 Universal by Design

Studio is a single SwiftUI codebase targeting iPadOS and macOS through `supportedDestinations` — not two applications sharing a code library, but one application. The same `.sdstkflow` or `.mbexpert` file opens identically on either platform, and the same three-column layout (palette, canvas, inspector) adapts to a Mac window or an iPad's larger multitasking surface without platform-specific branches in the widget layer itself. This mirrors the cross-platform design discipline used elsewhere in the same application family (ModelBuilder, Model Cartography) and reflects a broader thesis: that a single well-typed SwiftUI codebase, rather than platform-specific rewrites, is now a viable strategy for serious technical tools on Apple hardware — including tools whose scope grows, as Studio's did, well past its original brief.

---

09 Economic and Strategic Analysis

9.1 The Cost of Environment Management

A conventional Python data-science setup carries recurring operational cost that is easy to under-count because it is amortized across many small interruptions rather than billed as a line item: virtual environment creation, dependency resolution conflicts, Jupyter kernel restarts after a bad import, and — at the team level — the drift between "works on my machine" and a colleague's environment. None of this is a defect of any single tool; it is the structural cost of a scripting-language ecosystem built from independently versioned C-extension packages [3]. A Swift Package Manager–based application with no Python dependency at all removes this cost category rather than optimizing it: `swift build` and the App Store distribution model produce a single binary artifact, and there is no environment to drift.

9.2 On-Device Analysis for Sensitive Data

Because Studio performs every computation locally — no server process, no notebook kernel reachable over a network port, no cloud execution backend — it is suitable for exploratory analysis of data that a practitioner

is not willing or permitted to place on shared or cloud infrastructure: early-stage clinical research data, proprietary financial models, or any dataset under an institutional data-use agreement that restricts egress. This mirrors the on-device privacy argument made for ModelBuilder and Model Cartography elsewhere in this document family, applied here to the exploratory-analysis stage of the ML lifecycle rather than the trained-model stage.

9.3 Composing Specialists Instead of Retraining Generalists

The Experts category makes a distinct economic argument from the rest of the application: rather than retraining or fine-tuning one larger model every time a new capability is needed, an organization can train narrow, cheap, single-purpose Core ML models independently — on whatever schedule and whatever team each one needs — and compose them after the fact with `Coordinator`. This is a materially lower-cost path to expanding capability than the alternative of continually retraining a single larger model, and it composes naturally with `MemStacker`'s own incremental-improvement loop: a specific expert gets better in place, from real review decisions, without touching any other expert in the graph. Coordinator's move from a fixed two-expert ceiling to a variable two-to-eight-expert count (§4.2) widens this argument rather than merely fixing a limitation: a panel of several narrow specialists, added incrementally as new categories become worth covering, is now a first-class shape the canvas supports directly.

9.4 Agent-Tool Exposure as a Distribution Channel

MBExpertServer changes who can be a consumer of a Studio-built expert graph. Without it, a composed graph is only usable by a person operating Studio's own canvas. With it, the same graph is a callable tool for any MCP-speaking LLM agent [10] — meaning the practical audience for a narrow, privately trained Core ML model extends from "someone who opens this specific app" to "any agent workflow a user has already configured with an MCP client." For a specialist model — a defect classifier for one product line, a triage classifier for one document type — that is a meaningfully larger and more useful distribution surface than the model achieves sitting alone in a `.mlpackage` file.

9.5 Teaching and Onboarding Value

Orange's own adoption history [1] demonstrates that a widget canvas has particular value as a teaching tool: it makes the shape of a pipeline — which step depends on which — legible to a learner before they are fluent enough in a scripting language to express that same pipeline as code. Studio's eleven bundled example workflows, each opening fully wired and ready to run, extend this same value to Apple-platform learners, with the added property that several of the covered domains (signal processing, graph algorithms, closed-form scientific formulae) have no bundled teaching equivalent in Orange's own example gallery.

---

10 Limitations and Roadmap

An earlier version of this paper listed four items here — selection propagation, the Neural widget, Coordinator's fixed arity, and MBExpertServer's single-expert scope — as open future work. An external critique flagged those four specifically as the gap between the paper's framing and what had actually shipped; all four were subsequently closed, each with runtime verification rather than a typecheck-only pass (§3.6, §4.2, §5, §7). What remains honestly open:

Selection propagation is ScatterPlot-only. The engine-level plumbing (§3.6) supports any widget adding a selected-subset output, but only `ScatterPlot` currently does; extending it to Studio's other Visualize widgets is now mechanical rather than a design problem. The drag-marquee gesture itself is UI code that cannot be runtime-verified outside Xcode — the dataflow it feeds has been verified against the real execution engine, but the gesture's own feel has not yet been confirmed by a human running the app.

The MLP widget is regressor-only. `Model.MLPRegressor` (§7) covers regression; a classifier variant needs a different loss function and label encoding on top of the same training skeleton, and has not yet been built.

Broader Model coverage. Naive Bayes, support vector machines, and k-nearest-neighbors do not yet exist in SDSTK's `Learn` module; adding Model widgets for them requires extending SDSTK itself first, not just the Studio UI layer.

Live tool-use for the Research Expert. Web search or other tool-calling for `Experts.ResearchExpert` is deferred pending a design for what a search result looks like as a typed port value.

MBExpertServer's remaining scope. The server (§5) still assumes at most one Coordinator per bundle and ignores any non-expert node it encounters; the tabular-expert case is not yet wired into its bundle parser; and it has not yet been exercised against a real external MCP client (Cline, Claude Desktop) in a live interactive session.

Report generation and canvas annotations, and **pinch-to-zoom** on the canvas (currently pan-only), remain open.

---

11 Conclusion

SDSTK Studio was built on the premise that the best way to demonstrate a data-science toolkit works is to build something with it that a person actually wants to open. Every widget on the canvas is a real call into real SDSTK code — there is no mocked execution path and no simplified demo-only version of the underlying math. What the Experts category and MBExpertServer add is a second, related premise: the same typed, dataflow-scheduled execution engine that makes a data-science canvas feel alive is not specific to data

science at all — it is a general substrate for composing independently trained models, and, through a standard agent protocol, for making that composition useful to software that never opens the canvas itself. Combined with the architectural case made throughout this document family for MLX-native Swift as a serious alternative computational substrate to the Python data-science stack, Studio's contribution now spans both ends of that story: a well-typed, GPU-native visual canvas for building models, and the same canvas's graph engine repurposed as an orchestration layer for composing and shipping them.

// References

[1] Demšar, J., Curk, T., Erjavec, A., et al., "Orange: Data Mining Toolbox in Python," Journal of Machine Learning Research, vol. 14, pp. 2349–2353, 2013. https://jmlr.org/papers/v14/demsar13a.html

[2] Hannun, A., et al., "MLX: Efficient and flexible deep learning on Apple silicon," 2023. https://github.com/ml-explore/mlx

[3] Beazley, D., "Understanding the Python GIL," PyCON 2010, Atlanta, GA, 2010.

[4] Johnston, W.M., Hanna, J.R.P., and Millar, R.J., "Advances in Dataflow Programming Languages," ACM Computing Surveys, vol. 36, no. 1, pp. 1–34, 2004.

[5] Kahn, A.B., "Topological sorting of large networks," Communications of the ACM, vol. 5, no. 11, pp. 558–562, 1962.

[6] Jacobs, R.A., Jordan, M.I., Nowlan, S.J., and Hinton, G.E., "Adaptive Mixtures of Local Experts," Neural Computation, vol. 3, no. 1, pp. 79–87, 1991.

[7] Chow, C.K., "On Optimum Recognition Error and Reject Tradeoff," IEEE Transactions on Information Theory, vol. IT-16, no. 1, pp. 41–46, 1970.

[8] Settles, B., "Active Learning Literature Survey," Computer Sciences Technical Report 1648, University of Wisconsin-Madison, 2009.

[9] Swift Core Team, "SE-0302: Sendable and @Sendable closures," Swift Evolution, 2021. https://github.com/swiftlang/swift-evolution/blob/main/proposals/0302-concurrent-value-and-concurrent-closures.md

[10] Anthropic, "Introducing the Model Context Protocol," Anthropic News, November 2024. https://www.anthropic.com/news/model-context-protocol

[11] Apple Inc., "Accelerate Framework — LAPACK," Apple Developer Documentation, 2024. https://developer.apple.com/documentation/accelerate/lapack

[12] Apple Inc., "DocumentGroup," Apple Developer Documentation, 2024. https://developer.apple.com/documentation/swiftui/documentgroup

[13] Apple Inc., "Vision — VNCoreMLRequest," Apple Developer Documentation, 2024. https://developer.apple.com/documentation/vision/vncoremlrequest

[14] Lattner, C., and Adve, V., "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," Proceedings of the International Symposium on Code Generation and Optimization (CGO'04), 2004.

[15] Kingma, D.P., and Ba, J., "Adam: A Method for Stochastic Optimization," 3rd International Conference on Learning Representations (ICLR 2015), 2015. https://arxiv.org/abs/1412.6980