TU WIEN Informatics

Visuelle Komposition und Exploration finanzieller Datenflüsse

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Software und Information Engineering

eingereicht von

Filip Rozana
Matrikelnummer 12024732

an der Fakultät für Informatik
der Technischen Universität Wien
Betreuung: Dr. Manuela Waldner

Wien, 8. April 2026

Filip Rozana	Manuela Waldner

TU WIEN Informatics

Visual Composition and Exploration of Financial Dataflows

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software and Information Engineering

by

Filip Rozana
Registration Number 12024732

to the Faculty of Informatics
at the TU Wien
Advisor: Dr. Manuela Waldner

Vienna, April 8, 2026

Filip Rozana	Manuela Waldner

Declaration of Authorship

Filip Rozana

I hereby declare that I have written this thesis independently, that I have completely specified the utilized sources and resources and that I have definitely marked all parts of the work – including tables, maps and figures – which belong to other works or to the internet, literally or extracted, by referencing the source as borrowed.
I further declare that I have used generative AI tools only as an aid, and that my own intellectual and creative efforts predominate in this work. In the appendix "Overview of Generative AI Tools Used" I have listed all generative AI tools that were used in the creation of this work, and indicated where in the work they were used. If whole passages of text were used without substantial changes, I have indicated the input (prompts) I formulated and the IT application used with its product name and version number/date.

Vienna, April 8, 2026

Filip Rozana

Kurzfassung

Diese Arbeit untersucht, wie interaktive Visualisierungstechniken die Transparenz in knotenbasierten Editoren für finanzielle Zeitreihendaten verbessern können. Bestehende Werkzeuge in diesem Bereich ermöglichen den Aufbau von Workflows, bieten jedoch wenig Unterstützung für die Inspektion von Zwischenwerten oder das Nachvollziehen des Datenflusses durch die Verarbeitungskette. Ausgehend von einem Entwurfsdreieck aus Daten, Nutzern und Aufgaben leitet die Arbeit sieben Anforderungen aus der Literatur ab und setzt diese in einem webbasierten Editor um. Das System bietet knotenbasierte Sonden zur Inspektion von Zwischenzuständen, koordinierte Graph- und Diagrammansichten mit farbkodierter Verknüpfung, ausführungsbasierte Datenflusshervorhebung sowie eine integrierte Backtesting-Visualisierung. Eine Fallstudie anhand eines gleitenden-Durchschnitts-Crossover-Workflows demonstriert die Funktionsweise des Systems. Die Ergebnisse zeigen, dass bei statischen, vollständig sichtbaren Zeitreihendaten temporales Scrubbing keinen Mehrwert gegenüber direkter Navigation bietet, was im Gegensatz zu reaktiven Visualisierungssystemen steht, in denen Timeline-Replay notwendig ist. Persistente Farbkodierung ersetzt interaktives Brushing-and-Linking, und nutzergesteuerte Layoutflexibilität unterstützt sowohl Überlagerung als auch Nebeneinanderstellung je nach Inspektionsaufgabe.

Abstract

This thesis investigates how interactive visualization techniques can improve transparency in node-based editors for financial time-series data. Existing tools in this domain support workflow construction but provide limited means for inspecting intermediate values or tracing dataflow through a processing pipeline. Following a Data–Users–Tasks design triangle, the work derives seven design requirements from the literature and implements them in a web-based node-based editor. The system provides node-level probes for intermediate-state inspection, coordinated graph and chart views with color-coded linking, execution-based dataflow highlighting, and integrated backtesting visualization. A case study using a moving average crossover workflow demonstrates the system in operation. The results show that for static, fully visible time-series data, temporal scrubbing provides no benefit over direct crosshair navigation, contrasting with reactive visualization systems where timeline replay is necessary. Persistent color encoding replaces interactive brushing-and-linking, and user-controlled layout flexibility supports both superposition and juxtaposition depending on the inspection task.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Node-based approaches offer a visual way to construct and explore complex data-driven systems. By representing workflows as directed graphs of modular components, they allow users to build and reason about computational processes without relying on low-level programming [Kod20]. Each node encapsulates a self-contained operation, and edges define how data flows between them, making the structure of a pipeline explicit and modifiable through direct manipulation.

In the context of financial data analysis and model development, such systems can support users in interactively designing, analyzing, and refining algorithmic models. These models typically operate on regularly sampled, multivariate time-series data: datasets composed of uniform temporal intervals, each containing multiple quantitative attributes. Unlike irregular event streams, these series capture processes that evolve over consistent time spans, such as price movements, volume patterns, or derived indicators sampled at fixed frequencies.

The effectiveness of such environments depends not only on how easily users can assemble workflows, but also on how well they can understand the data and the behavior of their constructed models at each stage. Visualization and interaction techniques play a central role in supporting this understanding. Approaches such as Coordinated Multiple Views (CMV) [Rob07], visual debugging [HSH16], and always-on pipeline inspection [SCHP23] have demonstrated the value of making intermediate states visible and interactively explorable. However, these techniques have largely been developed outside the context of node-based editors for financial workflows.

1.2 Problem Statement

Current node-based editors in the financial domain provide convenient ways to connect components but offer limited support for understanding what happens inside them. Users can typically configure individual nodes and observe final outputs, yet the intermediate stages of a pipeline, where data is transformed, filtered, or combined, remain largely opaque. Research on dataflow systems in adjacent domains has repeatedly identified this pattern: Battle et al. found that without dedicated visualization, temporal query pipelines behave as "black boxes" in which intermediate states are entirely hidden from the user [BFD+16], and Shrestha et al. observed that data scientists working with wrangling pipelines resort to additional scripting just to inspect intermediate results, as existing tools provide no built-in means to do so [SCHP23]. Given that financial workflows share the same structural properties, namely multi-step pipelines where intermediate state is not directly visible, similar opacity is expected.

The consequences are compounded by the nature of financial time-series data. When an error is introduced at an early stage, for example a misconfigured indicator parameter or an incorrect normalization, it propagates silently through all downstream nodes, and the final output alone offers no indication of where the fault originated [BFD+16]. The temporal dimension adds further difficulty: a node may produce expected results across most of a time range but behave incorrectly around boundary conditions or unusual market periods. Hoffswell et al. addressed precisely this class of problem by introducing timeline replay for reactive visualizations, enabling users to step through time and observe how values change at each stage [HSH16]; this capability is absent from current financial node-based editors. When multiple signals interact, the challenge grows further, as a node that combines several inputs obscures each signal's individual contribution. Understanding why a particular output was produced at a given time step requires observing all upstream values simultaneously, which, as Baldonado et al. and Roberts have argued, demands coordinated visualization across multiple linked views [BWK00, Rob07].

Without such capabilities, debugging and interpreting workflows is reduced to manual trial and error. Kandel et al. showed that providing step-wise previews of data transformations can more than halve the time users spend on comparable tasks [KPHH11], suggesting that the absence of inspection support is not a minor inconvenience but a fundamental obstacle to effective model development.

Figure 1.1 illustrates this situation: the nodes and their connections are visible, but intermediate values and the relationship between graph structure and temporal data remain opaque. There is a need for interactive visualization techniques that allow users to inspect intermediate values, trace the propagation of data through connected nodes, and connect algorithm behavior to the underlying time-series signals. This thesis investigates how such techniques can improve transparency in node-based editor environments, with a focus on financial time-series data.

Figure 1.1: A node-based workflow for financial data analysis without visual analysis aids. An asset's price data is bound to the input of two indicator nodes, whose outputs feed into a comparison rule. The chart displays the raw price series. Without probes or other inspection features, intermediate values and the relationship between graph structure and temporal behavior remain opaque.

1.3 Research Questions

The main research question guiding this thesis is:

> Which interactive visualization techniques best support users in understanding and debugging node-based financial dataflow workflows?

This question is divided into three sub-questions:

1. **Inspection and Understanding:** Which visualization and interaction mechanisms most effectively support users in examining intermediate data and temporal behavior within node-based workflows?

2. **Linking and Tracing:** How can CMV and visual tracing techniques connect graph structure and time-based data to make signal flow and dependencies more transparent?

3. **Integration and Usability:** How can these techniques be combined in a coherent interface that remains clear, usable, and responsive for realistic financial data and workflow sizes?

1.4 Scope

Financial time-series data exists at multiple levels of temporal granularity. At the finest level, tick data records individual trades or order book changes as they occur, producing irregularly spaced event streams. At coarser levels, this raw activity is aggregated into fixed-interval summaries called *candles*, each capturing the open, high, low, and close prices as well as the traded volume over a defined period such as one minute, one hour, or one day.

This thesis focuses on regularly sampled financial time-series data, where all series share a common, fixed sampling interval. Open-High-Low-Close-Volume (OHLCV) candle data serves as the primary data representation. The approach accommodates any numeric series aligned to the same temporal grid, including derived quantities such as technical indicators or user-defined computations (formal definitions follow in Section 2.1).

Tick-level data, order book depth, and irregularly sampled event streams are outside the scope of this work. The contribution lies in the design and evaluation of visualization and interaction techniques for inspecting dataflow workflows within a node-based editor. While workflows may produce outputs suitable for trading decisions, the evaluation of such outputs is not a primary focus of this thesis. Portfolio-level concerns such as transaction costs, position sizing, and risk management are outside its scope entirely.

1.5 Methodology

This thesis follows a design-oriented approach in which visualization techniques identified in the literature are translated into concrete system features, implemented, and evaluated through use. The work proceeds in four phases:

Requirement analysis. Identify common user needs and challenges in debugging visual workflows, based on literature review and preliminary exploration.

Design. Develop and refine interaction concepts such as probes, timeline-based inspection, and CMV linking graphs to time-series plots.

Implementation. Build the node-based editor with the necessary visualization and interaction capabilities for inspection and debugging.

Evaluation. Evaluate the system through a case study on a representative workflow and through feedback from users with programming backgrounds who used the editor to construct and inspect financial workflows.

1.6 Expected Contribution

The expected outcome is a working system that demonstrates how temporal inspection, value tracing, and coordinated exploration techniques can be adapted to node-based financial time-series workflows, together with an evaluation of these techniques through a case study.

1.7 Thesis Structure

This thesis is organized as follows. Chapter 2 provides background definitions, surveys existing node-based editor environments, and reviews related work on intermediate-state inspection, coordinated views, time-series comparison, and interaction latency. Chapter 3 derives design requirements from the literature, presents the system architecture and the design of the visual analysis features, and describes key implementation decisions. Chapter 4 evaluates the system through a case study, assesses how well the requirements are met, and discusses design outcomes in relation to the research questions. Chapter 5 summarizes the findings, reflects on the design framework, and identifies directions for future work.

CHAPTER 2

State of the Art

This chapter provides the conceptual and empirical foundations for this thesis. Section 2.1 defines the core data structures and abstractions (time series, candle data, technical indicators, and dataflow graphs) that underlie the workflows addressed throughout this work. Section 2.2 surveys existing node-based editor environments across domains to establish which inspection capabilities are available in practice and where gaps remain. Section 2.3 then reviews prior research organized by the concrete design problems these workflows raise.

2.1 Background

This section defines the core concepts underlying this thesis.

2.1.1 Time Series

A *univariate time series* is an ordered sequence $\mathbf{x} = (x_1, x_2, \ldots, x_T)$ of T observations. A series is *regularly sampled* if consecutive observations are separated by a fixed interval Δt, such that observation x_t corresponds to the time point $t_0 + (t-1) \cdot \Delta t$ for a reference time t_0.

A *multivariate time series* consists of d concurrent univariate series on the same temporal grid. When values are real-valued, this can be represented as a matrix $\mathbf{X} \in \mathbb{R}^{T \times d}$.

In the workflows addressed by this thesis, regularly sampled time series are the primary data flowing between processing nodes.

2.1.2 OHLCV Candle Data

Financial markets produce price and volume information that is commonly aggregated into fixed-interval summaries for analysis. A *candle* (or *bar*) summarizes price activity

over a single time interval $[t, t + \Delta t)$ as a tuple (o, h, l, c, v) where:

- $o \in \mathbb{R}^+$: the *open* price, the first traded price in the interval,
- $h \in \mathbb{R}^+$: the *high* price, the maximum traded price in the interval,
- $l \in \mathbb{R}^+$: the *low* price, the minimum traded price in the interval,
- $c \in \mathbb{R}^+$: the *close* price, the last traded price in the interval,
- $v \in \mathbb{R}_0^+$: the *volume*, the total quantity traded during the interval.

By definition, $l \leq \min(o, c)$ and $\max(o, c) \leq h$ hold for every candle. A candle series is a regularly sampled sequence of such tuples, forming a multivariate time series with $d = 5$. OHLCV candle data serves as the canonical input format for the financial workflows considered in this thesis.

2.1.3 Technical Indicators

Technical indicators are quantitative measures derived from price or volume data, used to identify patterns or summarize market behavior. Formally, a *technical indicator* is a function $f\colon \mathbb{R}^{T\times d} \to \mathbb{R}^{T\times d'}$ that maps d input series of length T to d' output series of the same length, preserving the temporal grid. For example, a Simple Moving Average (SMA) with window size k computes

$$\mathrm{SMA}_k(t) = \frac{1}{k}\sum_{i=0}^{k-1} x_{t-i}$$

and maps a single input series to a single output series ($d = d' = 1$), while Bollinger Bands produce three output series from one input ($d = 1$, $d' = 3$). In practice, indicators that depend on past values require a warm-up period during which output values are not yet fully defined; the handling of these boundary values is implementation-defined. In a node-based workflow, each indicator is typically represented as a processing node with typed input and output ports.

2.1.4 Dataflow Graphs

A *dataflow graph* is a Directed Acyclic Graph (DAG) $G = (N, P, E)$ where N is a finite set of nodes, P is the set of all ports, and E is a set of directed edges between ports.

Each node $n \in N$ has a set of input ports $P_{\text{in}}(n) \subseteq P$ and output ports $P_{\text{out}}(n) \subseteq P$, together with a computation function f_n that maps the values at its input ports to values at its output ports. Each port carries a single univariate time series; multivariate data such as OHLCV candle series is decomposed across multiple ports, one per component. An edge $(p_o, p_i) \in E$ with $p_o \in P_{\text{out}}(n_a)$ and $p_i \in P_{\text{in}}(n_b)$ indicates that the series produced at p_o is consumed at p_i. Each input port accepts at most one incoming edge;

output ports may have multiple outgoing edges. The graph is acyclic in the sense that no sequence of edges induces a cycle through nodes.

In the context of this thesis, all series within a graph share a common temporal grid, a property that distinguishes these graphs from general-purpose dataflow systems and reflects the regularly sampled data model defined above. The visual composition and interactive inspection of such graphs is the central focus of this work.

2.2 Existing Node-Based Environments

The dataflow model defined in the previous section finds its user-facing form in node-based editors: visual environments where users place processing nodes on a canvas, connect their ports with directed edges, and manipulate the resulting graph through direct interaction. The spatial layout of the graph serves both as an authoring interface and as a structural overview of the pipeline. This section surveys existing environments that follow this paradigm, organized by application domain, to establish which inspection capabilities are available in practice and where gaps remain.

2.2.1 Creative and Engineering Applications

Node-based editors are well established in creative and engineering fields. Blender [Ble25] provides Shader Nodes for defining materials and Geometry Nodes for procedural modeling, both operating on domain-specific data types (color values, meshes, scalar fields) through composable graphs. Unreal Engine [Epi25] extends the paradigm beyond content creation with its Blueprint system, a general-purpose visual scripting environment for game logic. LabVIEW [Kod20], one of the earliest dataflow programming environments, applies the model to scientific measurement and industrial automation, where nodes represent instruments, signal processing steps, or control logic.

These environments share a common interaction model (nodes with typed ports, edges that define data dependencies, and a canvas for spatial arrangement) and each provides some degree of runtime inspection. LabVIEW, for instance, offers a “Highlight Execution” mode that animates data flow through the graph and displays intermediate values on edges [Kod20]. Blender shows live previews within shader node thumbnails. However, these inspection features are tightly coupled to their respective data types (geometry, audio signals, control signals) and do not generalize to continuous numeric time series.

2.2.2 Data Science and Analytics

In data science, platforms such as KNIME [BCD+07] and Orange [DCE+13] provide node-based workflow editors for constructing data processing and machine learning pipelines. These tools emphasize modularity and reproducibility: each node encapsulates a self-contained operation, and the graph defines the sequence of transformations applied to a dataset. Both offer table-level previews at each node, allowing users to examine

intermediate data states after execution. KNIME additionally provides column-level statistics and distribution summaries within its node outputs [BCD+07].

While these inspection capabilities resemble the always-on intermediate-state visibility advocated by Shrestha et al. [SCHP23], they are designed for tabular data (rows, columns, categorical and numeric attributes) rather than for regularly sampled time series where temporal ordering, alignment, and trend behavior are central to understanding the data.

2.2.3 Financial Domain

In financial analysis and algorithmic trading, node-based approaches are notably less common. The dominant paradigm relies on scripting environments and domain-specific languages, such as TradingView's Pine Script [Tra25] for indicator calculation or Python-based frameworks for backtesting, as well as spreadsheet-style interfaces.

One exception is StockSharp Designer [Sto25], an open-source platform that provides a visual editor for composing trading strategies by connecting blocks and lines on a schema without writing code. The editor includes a library of over 70 built-in indicators, supports backtesting of assembled strategies, and offers a formula debugger with step-by-step execution. However, based on inspection of the tool's documentation and interface, its visual design focuses on strategy assembly and execution control rather than on inspecting intermediate time-series values as they flow through the graph. The editor does not provide coordinated views linking the graph structure to temporal data, nor does it support inline visualization of intermediate results at individual nodes [Sto25].

The combination of a node-based editor with interactive visualization for inspecting time-series data at every stage of a pipeline, which is the focus of this thesis, is not addressed by existing tools in this domain.

2.3 Related Work

Designing visual analytics methods for time-oriented data requires jointly considering three factors: the characteristics of the data, the users of the system, and the analytical tasks they perform [MA14]. Miksch and Aigner formalize this as a *Data–Users–Tasks design triangle*, where the edges represent quality criteria that an effective solution must satisfy: *expressiveness* (visualizing exactly the information contained in the data), *effectiveness* (matching the cognitive capabilities and task context of the user), and *appropriateness* (providing sufficient benefit relative to the cost of the visualization) [MA14].

In the context of this thesis, the data vertex is defined by the regularly sampled financial time series and dataflow graphs introduced in Section 2.1. The users are practitioners constructing and refining algorithmic models, with varying levels of domain and technical expertise. Their tasks center on inspecting intermediate results, tracing signal propagation, and comparing temporal behavior across pipeline stages. These three factors give rise to the concrete design problems reviewed in the following subsections: how to make

intermediate data visible across a multi-step pipeline, how to coordinate a structural graph view with temporal data views, how to represent and compare time series effectively under space constraints, and how to keep the interface responsive enough that frequent inspection remains practical.

2.3.1 Intermediate-State Visibility in Data Pipelines

When a pipeline consists of multiple transformation steps, hiding intermediate results forces users to reason about cumulative effects from inputs and outputs alone. Several systems have addressed this by making each step individually inspectable.

Kandel et al. present Wrangler, a system for interactive data transformation that couples each operation with an inline preview of its effect on the data [KPHH11]. Quality indicators summarize column-level statistics (valid, missing, type-mismatched values) as compact bars above each column, functioning as always-on probes that surface problems without requiring manual inspection. Users can step through the transformation history, enable or disable individual operations, and observe how each change propagates through the data. A controlled study shows that this tight coupling between specification and immediate feedback reduces task completion time compared to manual editing in spreadsheets [KPHH11].

Battle et al. apply a similar principle to temporal query pipelines in data stream management systems [BFD+16]. Their system, StreamTrace, visualizes each operator in the query as a node in a workflow diagram, paired with a per-operator timeline that shows the events each operator produces. Intermediate results are first-class views rather than hidden internal state, and provenance-based highlighting, a form of visual tracing, allows users to trace how individual events propagate through the pipeline by hovering on any event to reveal its upstream sources and downstream effects [BFD+16]. The combination of structural overview (the workflow diagram) with temporal detail (per-operator timelines) supports fault localization: users can identify which operator first introduces incorrect behavior by scanning aligned timelines for unexpected patterns.

Shrestha et al. extend the idea of always-on intermediate inspection to contemporary data-science practice with Detangler, an IDE-integrated tool for debugging data wrangling pipelines [SCHP23]. A compact pipeline overlay color-codes each step by its effect on the data (no change, structural change, visible change, error), providing a high-level map of where transformations are consequential. Selecting any step immediately shows the intermediate data state in synchronized detail views: column-level summaries with distributions and a full data table. Their user study finds that participants use the tool in short bursts when confused about a transformation's effect, and that finer-grained step inspection correlates with higher task completion, particularly for less experienced users [SCHP23].

Hoffswell et al. address a related problem in reactive visualization specifications, where interactive behavior produces time-varying internal state that is difficult to debug [HSH16]. Their approach introduces a signal timeline, a matrix of signals over interaction events,

that makes the reactive update process explicit and navigable. Users can replay past states, inspect dependencies between signals at each update, and use dynamic data tables to examine how intermediate datasets change over time. Different debugging views prove useful for different classes of errors: the timeline and replay for interaction logic, data tables for transformation errors, and in-situ annotations for encoding problems [HSH16].

Across these systems, a consistent finding emerges: intermediate-state visibility is most useful when paired with mechanisms that help users localize where a change originates and trace how it propagates through subsequent steps [KPHH11, BFD+16, SCHP23, HSH16]. For node-based editor workflows operating on time series, this suggests that individual nodes should be inspectable and that the inspection should support both structural navigation (which node?) and temporal navigation (at which point in the series?).

2.3.2 Coordinating Graph Structure and Temporal Data

A node-based editor for time-series workflows presents users with at least two fundamentally different representations: a graph showing the structure of the dataflow, and plots showing how values evolve over time. The problem of coordinating such heterogeneous views has been studied extensively under the framework of CMV.

Baldonado et al. propose eight design guidelines for multiple-view systems, organized around when to add views and how to coordinate them [BWK00]. Their rule of *parsimony* advises using as few views as possible while still supporting core tasks, to limit the cognitive overhead of maintaining awareness across panels. The rule of *self-evidence* calls for making relationships between views perceptually obvious, for example by aligning shared axes or using consistent color encodings, so that users do not need to infer correspondences manually. The complementary rule of *decomposition* argues that complex data should be partitioned across views rather than compressed into a single overloaded display. Taken together, these guidelines suggest that a workflow editor should present graph topology and temporal behavior in separate but explicitly linked panels, with coordination that is selective enough to support task-relevant cross-references without coupling views so tightly that local exploration becomes difficult [BWK00].

Roberts surveys the CMV design space more broadly, framing coordination as a set of interaction couplings applied across views [Rob07]. Common couplings include brushing and linking, synchronized navigation (where panning or zooming in one view propagates to others), and coordinated filtering. Roberts distinguishes between the *mechanism* of coordination (what events are shared between views) and the *intent* (why the coupling helps a task), noting that coordination can hinder exploration when it removes degrees of freedom that users need [Rob07]. For a dataflow editor linking graph nodes to time-series plots, this distinction is important: selecting a node should reveal corresponding temporal behavior, but the temporal view should remain independently navigable so users can compare different time ranges without affecting the graph selection.

Roberts also describes Module Visualization Environments, which follow a dataflow paradigm where processing modules are connected into visual programs and views are

attached at different points in the pipeline [Rob07]. This architecture closely resembles the node-based editor model addressed by this thesis, with the additional requirement that the editor itself must be the subject of visual inspection rather than merely a tool for constructing visualizations.

2.3.3 Visual Comparison of Time Series

Inspecting a dataflow workflow often requires comparing time series: an input against an output, the effects of different parameters, or multiple signals across pipeline stages. How these comparisons are presented has a measurable effect on what users perceive.

Gleicher et al. provide a taxonomy of comparative visualization strategies: *juxtaposition* (placing objects side by side), *superposition* (overlaying them in a shared space), and *explicit encoding* of relationships such as differences or alignments [GAW+11]. Each strategy involves trade-offs. Juxtaposition preserves the legibility of individual objects but relies on memory and attention shifts to connect views. Superposition enables direct pointwise comparison but introduces clutter and occlusion as the number of overlaid objects grows. Explicit encodings, such as a difference plot between two series, make relationships directly visible but can obscure the original signals [GAW+11]. For time-series inspection in a workflow editor, these strategies are not mutually exclusive: overlaying a small number of related signals on shared axes suits local comparison, while juxtaposed small multiples suit comparisons across many nodes, and difference views can highlight where a transformation has a large effect.

Javed et al. empirically compare shared-space and split-space layouts for multiple time series [JME10]. In a controlled study, shared-space line charts perform better for local comparison tasks where targets are close in time, because users can compare values directly within the same coordinate system. Split-space layouts (small multiples and horizon graphs) perform better for dispersed comparisons spanning the full series, because separating the series eliminates overlap and makes individual patterns easier to isolate [JME10]. Performance degrades with the number of concurrent series regardless of layout, but split-space techniques scale more gracefully, maintaining usable performance for up to 16 series where shared-space views become cluttered [JME10].

Heer et al. examine how chart height and layering affect the perception of time-series values, comparing standard line charts with mirrored and horizon graph variants [HKA09]. Mirroring negative values around zero halves the required chart height without measurable loss in estimation accuracy. Horizon graphs with two bands maintain accuracy at even smaller heights (down to approximately 12 pixels), at the cost of slower reading due to the cognitive effort of mentally unstacking bands. At four or more bands, both speed and accuracy degrade significantly [HKA09]. These findings provide concrete sizing guidelines for compact time-series views: approximately 24 pixels for standard or mirrored charts, and two-band horizon graphs when space is tighter.

Gogolou et al. investigate how visual encoding affects *perceived similarity* between time series [GTPB19]. Their study compares line charts, horizon graphs, and color-field

representations and finds that the choice of encoding shifts which candidates participants judge as most similar to a query. Horizon graphs promote tolerance to local time shifts (favoring dynamic time warping-based matches) but penalize amplitude differences, while line charts and color fields align better with amplitude-normalized similarity [GTPB19]. This result is relevant for workflow inspection because it implies that how intermediate results are visualized can influence a user's assessment of whether a transformation is behaving correctly.

2.3.4 Practical Constraints

Two practical concerns cut across the design problems discussed above: interaction latency and the ability to capture and revisit findings.

Liu and Heer study how latency affects exploratory visual analysis by injecting controlled delays into an interactive multi-view system [LH14]. Adding 500 milliseconds of latency significantly reduces interaction rates, particularly for brushing and linking and selection, and leads to fewer observations and hypotheses per unit time. Moreover, initial exposure to high latency depresses exploratory behavior even after the latency is removed, suggesting that early interactions with a system shape subsequent analysis habits [LH14]. For a workflow editor that relies on frequent probing and cross-view coordination, these findings reinforce the perceptual time-scale framework of Card et al. [CMN83]: brushing and linking should respond within approximately 100 milliseconds (the perceptual fusion threshold) to avoid shifting users toward more conservative exploration strategies, and heavier recomputations should remain below one second.

Heer et al. explore how visualization tools can support the externalization of analysis findings through annotations, bookmarks, and comment-linked view states [HVW07]. Their system, sense.us, treats each view state as a URL-addressable resource and attaches threaded discussions to specific configurations of the visualization. Geometric annotations (freehand marks, arrows, and text) are rendered as a separate overlay layer, enabling users to highlight specific features such as spikes, trend changes, or anomalies without modifying the underlying visualization [HVW07]. While designed for collaborative analysis, the underlying mechanisms (persistent, referenceable view states and lightweight annotations) are equally useful in single-user workflow editors where users must revisit prior inspection states or compare different configurations of a pipeline.

2.3.5 Summary

The findings yield concrete design implications for a node-based editor operating on financial time-series data.

Inspection. Each processing step should expose its intermediate state for direct examination, not only the pipeline's final output. Inspection is most effective when it supports both fault localization (identifying *where* in the graph a change originates) and propagation tracing (following *how* that change affects downstream nodes) [BFD+16, SCHP23].

Because the data is temporal, navigation along the time axis is equally important: a node may behave correctly for most of a series but fail at specific points [HSH16].

Coordination. Graph topology and temporal behavior should appear in separate but explicitly linked views [BWK00]. Coordination must be selective: selecting a node should reveal its temporal data, but temporal views should remain independently navigable so that users retain the freedom to explore different time ranges or compare signals without constraining the graph selection [Rob07].

Comparison. The layout of time-series views should match the comparison task at hand. Shared-space overlays support local comparison where targets are close in time; split-space layouts scale more gracefully to many series and dispersed comparisons [JME10]. Compact representations remain effective down to approximately 24 pixels for standard charts and two-band horizon graphs at smaller sizes [HKA09], but the choice of encoding also shapes which differences users perceive as salient [GTPB19].

Responsiveness. Interactive coordination, particularly brushing and linking, requires response times below the 100-millisecond perceptual fusion threshold [CMN83]; latency above this range measurably reduces exploratory behavior [LH14]. Persistent view states and lightweight annotations support the complementary need to capture and revisit findings across sessions [HVW07].

Among the systems surveyed, none addresses these four concerns together for time-series data within a node-based editor. The inspection tools reviewed in Section 2.3.1 operate on tabular or event data, not on continuous numeric time series within node-based editors. The CMV frameworks discussed in Section 2.3.2 address view coordination in general but do not consider pipeline inspection as a use case. The perception studies of Section 2.3.3 establish layout and sizing guidelines for standalone charts, outside of interactive workflow contexts. Integrating these concerns for financial time-series dataflow workflows is the focus of this thesis: enabling intermediate-state inspection within individual nodes (RQ1), coordinating graph structure with temporal behavior through linked views and visual tracing (RQ2), and combining these capabilities in a coherent, responsive interface (RQ3).

CHAPTER 3

Method

This chapter describes the design and implementation of the node-based editor. Section 3.1 derives design requirements from the literature reviewed in Chapter 2. Section 3.2 presents the overall system architecture. Sections 3.3 and 3.4 detail the core editor and the visual analysis features, respectively. Section 3.5 covers technology choices and key implementation decisions.

3.1 Requirements

The design implications identified in Section 2.3.5 (inspection, coordination, comparison, and responsiveness) establish what a node-based editor for financial time-series workflows should support. Following the Data–Users–Tasks design triangle of Miksch and Aigner [MA14], these implications reflect the interplay of the data characteristics (regularly sampled financial time series), the target users (practitioners building and refining algorithmic models), and the core analytical tasks (inspecting intermediate state, tracing signal propagation, comparing temporal behavior). This section translates those implications into concrete requirements that guide the design of the system. Each requirement is traced to the research question it primarily addresses.

R1: Per-node intermediate-state inspection (RQ1). Every processing node in the workflow should expose its output for direct examination, rather than requiring users to infer intermediate results from the pipeline's final output alone. Intermediate-state visibility is most effective when users can select specific points in the graph for inspection [BFD+16, SCHP23], which means the system must provide a mechanism for attaching inspection views to individual nodes and their output ports.

R2: Temporal navigation (RQ1). Because the data is temporal, inspection must support navigation along the time axis. A node may behave correctly across most of a

series but produce unexpected results at specific time points [HSH16]. The system should allow users to explore how intermediate values evolve over time, at minimum through synchronized time-axis navigation across all inspection views.

R3: Linked graph and chart views (RQ2). Graph topology and temporal behavior should be visible in coordinated views [BWK00]. User actions that connect a node to an inspection view, such as assigning a probe, should produce a corresponding representation in the temporal display, making the relationship between graph structure and time-series behavior explicit.

R4: Independent view navigation (RQ2, RQ3). While views must be linkable, they should also remain independently navigable [Rob07]. Exploring a time range in the chart should not constrain the graph view, and selecting nodes on the canvas should not force the chart to a particular zoom level. Users need the freedom to compare different time ranges or inspect different nodes without views overriding each other.

R5: Flexible time-series layout (RQ1). The layout of time-series views should support both shared-space comparison, where related signals are overlaid on common axes, and split-space comparison, where signals are separated into independent panels [JME10, GAW+11]. Users should be able to decide which signals share a coordinate system and which are displayed separately, since the appropriate layout depends on the comparison task at hand.

R6: Responsive interaction (RQ3). Interactive coordination, particularly operations that update multiple views in response to user input, must respond within approximately 100 milliseconds, the perceptual fusion threshold identified by Card et al. [CMN83]. Liu and Heer [LH14] provide empirical evidence that latency above this range significantly reduces exploratory activity. Heavier computations, such as re-executing a workflow after a parameter change, should remain below one second.

R7: Dataflow visibility (RQ2). The system should make the propagation of data through the workflow graph visually apparent. When a workflow executes, users should be able to see which nodes are active, which edges carry data, and how signals flow from inputs to outputs [BFD+16]. This supports fault localization by revealing the path along which a value was produced.

Table 3.1 summarizes the requirements and their traceability to research questions.

3.2 System Overview

The system is a web-based node-based editor for composing and inspecting financial time-series workflows. This section presents its high-level architecture, data model, and

Req.	Description	RQ
R1	Per-node intermediate-state inspection	1
R2	Temporal navigation	1
R3	Linked graph and chart views	2
R4	Independent view navigation	2, 3
R5	Flexible time-series layout	1
R6	Responsive interaction	3
R7	Dataflow visibility	2

Table 3.1: Design requirements derived from the literature, with traceability to research questions.

execution semantics. Subsequent sections detail the editing environment (Section 3.3), visual analysis features (Section 3.4), and implementation decisions (Section 3.5).

3.2.1 Architecture

The interface is organized into three coordinated regions, shown in Figure 3.1. A vertical panel on the left provides the *library panel group*, which offers access to saved workflows, a searchable component library, and an asset browser. The remaining space is occupied by the *workspace*, which is split vertically: the upper area contains the *graph canvas* where users place and connect processing nodes, and the lower area contains the *chart panel* that displays time-series data. A toolbar overlaid on the canvas provides access to inspection tools, including probes and dataflow controls.

The three regions serve complementary roles that reflect the requirements from Section 3.1. The graph canvas addresses workflow construction and structural inspection (R1, R7). The chart panel provides temporal visualization and navigation (R2, R5). Their spatial separation supports independent interaction: users can navigate time in the chart without affecting the graph selection and rearrange nodes without disturbing the chart viewport (R4), while shared data state keeps them coordinated (R3).

3.2.2 Data Model

The data model distinguishes between *definitions*, which describe what a component does, and *instances*, which represent a particular use of that component within a workflow.

Building blocks. A *building block* defines a reusable processing component. Each building block specifies a set of typed input and output *ports* through which data enters and leaves the component, a set of configurable *parameters* that control its behavior, and metadata such as a name, description, and category. Ports carry semantic information (such as whether a port represents a price series, a binary signal, or a momentum indicator) that guides users during workflow construction. Parameters define their data

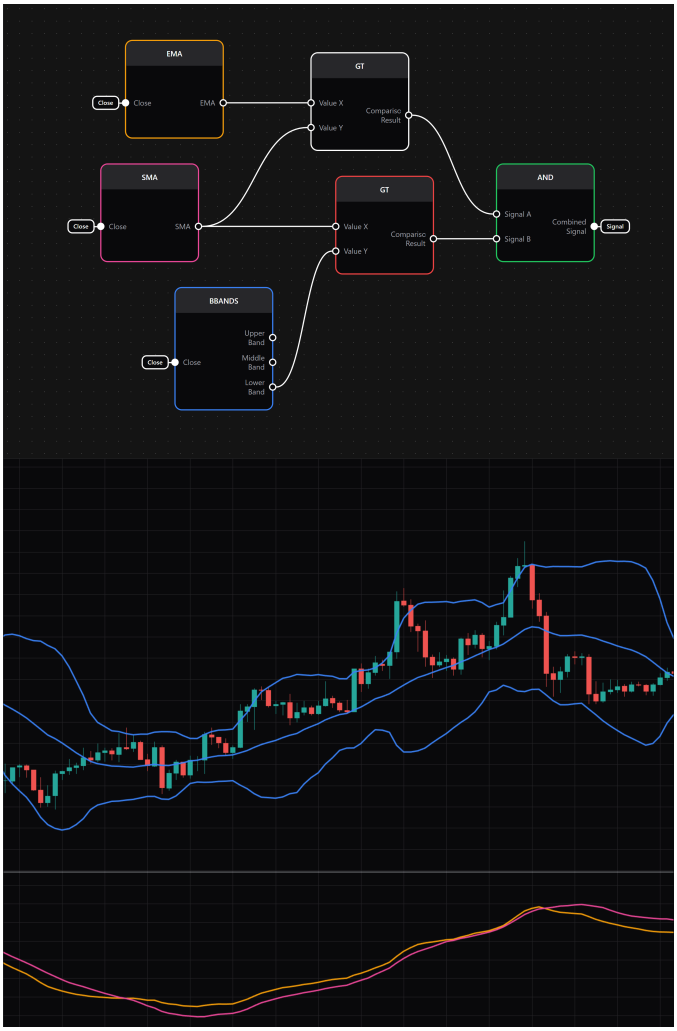

Figure 3.1: Layout of the node-based editor. The library panel group (left) provides access to components and assets. The graph canvas (top right) is the primary editing surface. The chart panel (bottom) displays time-series data across multiple panes.

type (integer, float, boolean, string, or enumeration), a default value, and optional validation constraints such as numeric bounds.

Component instances. A building block is placed on the canvas as a *component instance*. Each instance references its building block definition but is otherwise independent: it carries its own identity, a user-assignable name, a position on the canvas, and a set of parameter overrides that allow users to customize behavior without modifying the underlying definition.

Workflows as directed acyclic graphs. A workflow is composed of a set of component instances, a set of *connections* linking output ports to input ports across instances, and a set of *port bindings* that route external data (such as asset price series) into the graph. Section 2.1 formally defined a dataflow graph as a DAG $G = (N, P, E)$; the system realizes this model directly: component instances correspond to nodes N, their ports to P, and connections to edges E. The acyclicity constraint is enforced at execution time through topological sorting (see Section 3.2.3).

3.2.3 Execution Model

When a user triggers execution, the system processes the workflow in two phases: compilation and runtime evaluation.

Compilation. The compiler performs a topological sort of the component instances using Kahn's algorithm. This determines a valid execution order in which every component is evaluated only after all of its upstream dependencies have produced their outputs. If the graph contains a cycle, the sort will fail to process all nodes, and the system reports an error.

Runtime evaluation. The runtime iterates through the sorted components in order. For each component, it resolves parameter values by merging the building block's defaults with the instance's overrides, gathers the component's inputs (either from upstream connection outputs or from external port bindings), and invokes the component's processing function. Each function receives its inputs and resolved parameters and produces one or more output series. These outputs are stored in a result map keyed by component instance and port, making them available to downstream components and to the inspection tools described in Section 3.4.

Data representation. All data flowing through the graph takes the form of numeric arrays aligned to a common time grid. This aligns with the regularly sampled data model defined in Section 2.1: every series, whether an input price series, an intermediate indicator, or a final output signal, shares the same temporal indices and length. Components that require a warm-up period (for example, a moving average that needs n prior values) produce undefined values for the initial entries, which propagate as-is through downstream components.

3.3 Core Editor

3.3.1 Graph Canvas

The graph canvas is the primary interaction surface for building workflows. Users place component nodes on a two-dimensional canvas and connect them by dragging between port handles, forming the DAG structure described in Section 3.2.2. A dot-grid background

provides spatial reference, and an optional snap-to-grid mode (20-pixel increments) helps maintain alignment. Standard navigation controls (pan by dragging the background, scroll to zoom, and a fit-to-view button) allow users to work with graphs of varying sizes.

Connections are drawn as Bézier curves between output and input port handles. During connection creation, a live preview follows the cursor to provide immediate feedback. The system enforces two structural constraints: each input port accepts at most one incoming connection, and the resulting graph must remain acyclic. Cycle prevention is checked at connection time, and the underlying topological sort (Section 3.2.3) provides a second validation at execution time. Nodes and edges can be selected individually or via a selection rectangle, and deleted with the keyboard. Figure 3.2 shows an example workflow on the canvas.

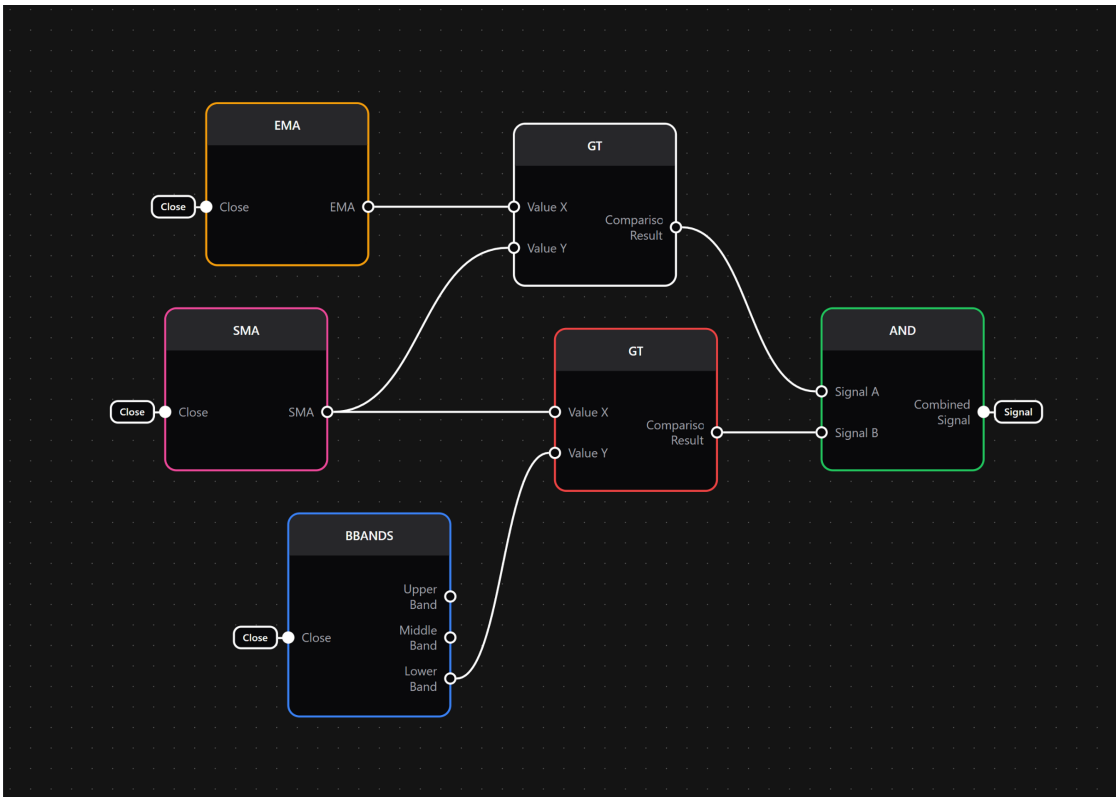

Figure 3.2: The graph canvas showing a workflow with indicator, rule, and operator nodes. Each node displays its name, input ports on the left edge, and output ports on the right edge. Connections between ports are rendered as Bézier curves. Colored node borders indicate assigned probes.

3.3.2 Node and Port Model

Each node on the canvas is a *component instance* that references a building block definition. The node displays its name, its input ports along the left edge, and its output ports

along the right edge. Port handles are interactive targets for creating connections and for the binding and inspection mechanisms described in later sections.

Building block definitions specify the ports and parameters that characterize a component's interface. Ports carry semantic annotations (such as whether a port represents a price series, a momentum indicator, or a binary signal) that help users understand what kind of data a port expects or produces. Parameters expose configurable values (for example, the period of a moving average or the threshold of a comparison rule) with type information, default values, and optional validation bounds. When a building block is placed on the canvas, the resulting instance inherits the definition's defaults but can override individual parameter values without affecting other instances of the same building block.

3.3.3 Component Library

The component library is a searchable, categorized panel that provides the available building blocks. Components are organized into four categories:

Indicators. Technical analysis computations that transform input series into derived quantities. The library includes a SMA, an exponential moving average, the relative strength index, Bollinger Bands, and the commodity channel index.

Rules. Comparison operations that evaluate an input series against a threshold, producing a binary output. Available rules include greater-than, less-than, and equal-to comparisons.

Operators. Logical combinators (AND, OR, NOT, and XOR) that merge or invert binary signals.

Custom. User-defined components for operations not covered by the built-in categories, such as a price multiplier that scales an input series by a constant factor.

Users can search across component names, descriptions, and tags. Adding a component to the canvas creates a new instance at a default position, ready to be connected and configured. The library panel is shown in Figure 3.3.

3.3.4 Node Configuration

Each node provides access to a settings panel through a button in its header. The panel lists all configurable parameters with type-appropriate input controls: numeric fields for integer and floating-point values (with range hints when bounds are defined), checkboxes for booleans, dropdowns for enumerated options, and text fields for strings. Input validation provides immediate feedback, for example indicating when a value falls outside the permitted range, and changes are saved as parameter overrides on the instance.

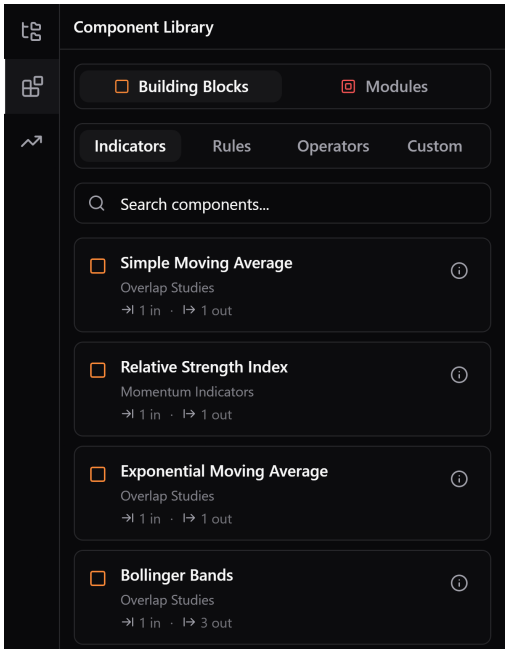

Figure 3.3: The component library panel with four categories (Indicators, Rules, Operators, Custom) and a search field. Each entry shows the component name, subcategory, and port counts.

An information panel, accessible through a separate button, displays the building block's metadata: its full name, description, category, input and output port definitions, and any associated tags.

3.3.5 Asset Selection

Financial time-series data enters the workflow through an asset selection mechanism. An asset browser panel organizes available data sources by category (indices, forex, commodities, and cryptocurrencies) and provides search functionality. Each workflow is associated with a single asset, which supplies the OHLCV price series that can be bound to component input ports. This single-asset constraint reflects the scope defined in Section 1.4: the system focuses on workflows operating on one regularly sampled time series at a time.

3.3.6 Session and Tab Management

The editor supports working with multiple workflows concurrently through a tab-based session model. Each tab corresponds to an open workflow, and users can create, open, save, and close workflows independently. Unsaved changes are tracked per tab, with a visual indicator and a confirmation dialog when closing a modified workflow. Workflow

state, including the graph structure, parameter overrides, asset selection, and canvas viewport position, is persisted and restored across sessions.

3.4 Visual Analysis Features

This section describes the visual analysis and inspection features that address the requirements derived in Section 3.1. Each subsection presents a feature, connects it to the literature that informed its design, and identifies the requirements it addresses.

3.4.1 Time-Series Plots

The chart panel provides the primary temporal view of data flowing through the workflow. It displays time-series data in a multi-pane layout where each pane occupies a horizontal band with its own vertical axis, while all panes share a common time axis along the bottom. This arrangement directly supports Requirement R5 (flexible time-series layout): users can overlay multiple series within a single pane for shared-space comparison, or separate them into distinct panes for split-space comparison.

The pane system defines three categories with a fixed display order: the *main* pane appears at the top, *measurement* panes in the middle, and the *backtest* pane at the bottom. The main pane displays the asset's candlestick chart, providing the raw price context against which all other data is interpreted. Measurement panes contain the output series of probes (described in Section 3.4.2). The backtest pane, when present, shows an equity curve that summarizes the cumulative performance of the workflow's output signal over the historical data (see Section 3.5.5).

The chart supports several series types to accommodate different kinds of data: candlestick charts for OHLCV price data, line charts for continuous numeric series such as indicator outputs, baseline charts that shade regions above and below a reference value, and histograms for discrete or categorical signals. The choice of series type is determined by the data source: asset prices use candlestick rendering, probe outputs default to line charts, and backtest equity curves use baseline rendering.

Each pane is labeled with a watermark in the upper-left corner that identifies its role ("Main", "Pane 1", "Backtest") and lists the series it contains, color-coded to match their chart lines. This labeling supports Baldonado et al.'s guideline of *self-evidence*, making relationships between views perceptually obvious without requiring the user to infer them [BWK00].

The layout choices follow the findings of Javed et al. [JME10] and Gleicher et al. [GAW+11]: shared-space overlay within a pane supports local comparison where signals are closely related (for example, a price series and a moving average derived from it), while split-space separation into distinct panes supports comparison across independently scaled quantities (for example, a price series and a momentum oscillator on different numeric ranges). Users control this trade-off by assigning probes to specific panes (R5). Figure 3.4 shows the chart panel with three active probes.

Figure 3.4: The chart panel with three probes across two panes. The main pane displays a candlestick chart with two price-scaled probe overlays (SMA in green, EMA in blue). A second pane shows a third probe (Comparison Result in orange), separated because its output is a binary signal on a different scale. Pane watermarks in the upper-left identify each series by name and color. All panes share a common time axis.

3.4.2 Node-Level Probes

Probes are the primary mechanism for inspecting intermediate state within the workflow (R1). A probe is a named measurement point that targets a specific output port of a component instance, causing the system to capture that port's output series after execution and display it as a chart layer in the time-series panel.

The interaction workflow proceeds in three steps:

1. The user creates a new probe through the toolbar overlay. The system assigns it a name and a color from a rotating palette of twelve visually distinct colors.

2. The user *arms* the probe by selecting it in the overlay, which activates an assignment mode on the canvas.

3. The user clicks a node on the canvas to assign the probe to that node's output. The system resolves the target and assigns the probe to an appropriate chart pane based on the output's semantic type: price-related outputs are placed in the main pane, while other output types receive a new measurement pane.

Once assigned, the probe is indicated on the canvas by coloring the target node's border in the probe's assigned color. This provides an at-a-glance indicator of which nodes are being inspected and by which probe, as the same color appears on the node border, the chart series line, and the pane watermark entry (see Figure 3.2 and Figure 3.4).

A limit of two probes per node keeps the canvas readable. A single probe suffices for most inspection tasks; the second slot allows duplicating a series into a separate pane for comparison without requiring a dedicated duplication mechanism. This follows the parsimony guideline of Baldonado et al. [BWK00]: the number of simultaneously active views is kept low to reduce cognitive overhead. Users can toggle probe visibility without removing them, reassign them to different panes, or delete them entirely.

This design prioritizes full-fidelity temporal inspection over compact inline previews. Rather than embedding miniature visualizations directly on canvas nodes, as in the always-on summaries of Shrestha et al. [SCHP23] or the inline previews of Kandel et al. [KPHH11], the system renders probe outputs as full chart layers that share the time axis and can be compared against the asset data and other probe outputs. The trade-off is that probe data requires a glance at the chart panel rather than being visible directly on the node, but the resulting visualization provides richer temporal context.

3.4.3 Temporal Navigation

All chart panes share a single time axis, which means that navigating time in any pane automatically updates the temporal context across all panes (R2). Users navigate time through standard chart interactions: horizontal scrolling to pan along the time axis, and scroll-to-zoom to adjust the visible time range. A crosshair synchronized across all panes shows the exact time point under the cursor, together with the corresponding value on each pane's vertical axis.

This shared time axis supports the temporal inspection envisioned by Hoffswell et al. [HSH16]: by moving the crosshair across the chart, users can observe how intermediate values at different nodes (via their probes) evolve over time, and identify specific time points where a node's output deviates from expectations. The synchronization ensures that the temporal context is consistent across all inspection views: when the crosshair is positioned at a particular bar, every pane displays the values of all visible series at that bar.

3.4.4 Coordinated Views

The system coordinates its graph and chart views through user-directed linking rather than automatic brushing and linking (R3, R4). Coordination occurs through three mechanisms:

Probe assignment. As described in Section 3.4.2, assigning a probe to a node creates an explicit link between a graph element and a chart layer. The shared color encoding,

matching the node border, the chart series line, and the pane watermark entry, makes this link visually traceable. Users decide which nodes to inspect and how to arrange the resulting series across panes, giving them direct control over what is compared and how.

Execution state. When the workflow executes, the system updates both views simultaneously: the chart panel receives the computed time-series data for all active probes, while the graph canvas highlights the execution state of the workflow (described in Section 3.4.5). This synchronization ensures that the chart always reflects the current state of the graph.

Port binding. External data enters the workflow through port bindings that connect an asset's price series to component input ports. The chart's main pane displays the same asset data that flows into the graph, providing a shared reference frame: what users see on the candlestick chart is the same data their workflow is processing.

Crucially, the graph canvas and chart panel remain independently navigable (R4). Panning and zooming the chart does not affect the canvas viewport, and selecting or rearranging nodes on the canvas does not change the chart's visible time range. This decoupling follows Roberts' recommendation that coordination should not remove degrees of freedom that users need for independent exploration [Rob07]. Users can zoom into a specific time range on the chart to examine a pattern while keeping the full graph visible on the canvas, or vice versa.

3.4.5 Visual Tracing

When a workflow is executed, the system provides visual feedback on the graph canvas that reveals which parts of the workflow are active and how data flows through the graph (R7). This is implemented through execution-based dataflow highlighting:

- **Active nodes**, those that produced output during execution, are displayed with a highlighted border, while inactive nodes retain a muted border.
- **Active edges**, those whose source node produced output, are rendered with a highlighted stroke, making the data propagation path visible.
- **Active ports**, input ports that receive data from an active edge, are rendered with a highlighted border, indicating where data enters each node.

The highlighting state is derived from the execution result: the set of nodes that produced output determines active nodes, the graph's connection structure determines active edges, and port activity follows from edge activity. Because the visual state is derived from a single source, it cannot diverge from the actual data flow.

If a node is skipped during execution (for example, because a required input was not connected), it remains visually inactive, immediately signaling to the user that the expected data flow did not reach that node. This supports fault localization: a user investigating unexpected behavior can scan the graph for the first node in a path that is not highlighted, indicating where the data flow was interrupted.

The approach draws on the provenance highlighting of Battle et al. [BFD+16], adapted to the specific context of this editor: rather than highlighting individual events propagating through operators, the system highlights the structural paths along which time-series data flows during a complete execution pass.

3.4.6 Backtest Visualization

The editor includes a backtesting facility that evaluates the workflow's output signal against historical price data and visualizes the results. Its visualization integrates with the chart panel and provides temporal markers that support inspection.

When a backtest completes, two visual elements are added to the chart. First, *trade markers* (green upward arrows at entry points and red downward arrows at exit points) are overlaid on the main pane's candlestick chart, anchoring each trade to the time point at which it occurred. These markers function as system-generated annotations that highlight the temporal moments where the workflow's output signal produced concrete trading decisions. Second, an *equity curve* is displayed in a dedicated backtest pane as a baseline chart, with regions above zero shaded green and regions below zero shaded red, providing a cumulative view of the strategy's performance over time.

A performance summary panel reports aggregate metrics including cumulative return, number of trades, win rate, average return, maximum drawdown, expectancy, and profit factor. These metrics, combined with the temporal markers and equity curve, allow users to connect the workflow's structural behavior (visible on the graph and through probes) to its aggregate outcomes.

The backtesting model and its execution semantics are described in Section 3.5.5.

3.5 Implementation

This section describes the technology choices, architectural patterns, and key implementation decisions that support the features presented in the preceding sections.

3.5.1 Technology Stack

The editor is implemented as a web application using React and TypeScript. The graph canvas is built on ReactFlow, a library for rendering and interacting with node-based graphs. ReactFlow operates in controlled mode, meaning the application owns the graph state (nodes, edges, viewport) and passes it to the library for rendering, which enables undo/redo support, state persistence, and deterministic behavior.

Time-series visualization uses TradingView Lightweight Charts, a library designed for rendering financial chart data. It provides native support for candlestick, line, area, histogram, and baseline series types within a multi-pane layout that shares a single time axis across panes, forming the foundation of the chart panel described in Section 3.4.1.

Application state is managed with Zustand, a lightweight state management library. Stores are created as vanilla (non-React) instances, which allows the application layer to own state independently of the presentation layer. The user interface is composed from a component library built on Radix UI primitives and styled with Tailwind CSS.

3.5.2 Architecture

The codebase follows a layered architecture [Mar17] with strict dependency rules:

Domain layer. Pure business logic: entity definitions (building blocks, strategies, component instances), value objects (ports, parameters, connections), and repository interfaces. This layer has no external dependencies.

Application layer. Orchestrates domain operations, manages runtime state through Zustand stores, and produces data transfer objects for the presentation layer. The execution engine, probe management, chart projection, and backtest service reside here.

Presentation layer. React components that render the user interface and capture user interactions. Components access state through the application layer's stores and selectors, never the domain directly.

Infrastructure layer. Adapters for persistence (local storage) and external data loading. Implements the repository interfaces defined by the domain layer.

Dependencies point inward: presentation depends on application, application on domain, and infrastructure on domain. This separation ensures that business logic remains independent of rendering and storage concerns, supporting testability and the ability to replace infrastructure adapters without modifying the core logic.

3.5.3 Graph Execution

The execution engine is implemented as a standalone package with two components: a compiler and a runtime. The compiler performs a topological sort using Kahn's algorithm [Kah62], producing an execution plan that lists component instances in dependency order. If the graph contains a cycle, the compiler rejects it before execution begins. The runtime then iterates through this plan as shown in Algorithm 3.1: for each component, it resolves parameters, gathers inputs, invokes the processing function, sanitizes the outputs, and stores them for downstream components. Components whose required

Algorithm 3.1: Graph Execution

Input: Execution plan P (topologically sorted components), strategy graph G (connections, port bindings), external inputs E
Output: Component outputs O, set of skipped components S

$O \leftarrow \emptyset$;
$S \leftarrow \emptyset$;
foreach *component* c *in* P **do**
  Resolve parameters: merge definition defaults with instance overrides;
  foreach *input port* p *of* c **do**
    if p *has a port binding to external input* $e \in E$ **then**
      inputs$[p] \leftarrow E[e]$;
    else if p *has a connection from upstream component* u*, port* q **then**
      inputs$[p] \leftarrow O[u][q]$;
    else
      inputs$[p] \leftarrow$ undefined;
    end
  end
  if *any required input is undefined* **then**
    $S \leftarrow S \cup \{c\}$;
    continue;
  end
  result $\leftarrow$ execute$(c, \text{inputs}, \text{parameters})$;
  foreach *output series* v *in result* **do**
    Replace non-finite values in v with 0;
  end
  $O[c] \leftarrow$ result;
end
return O, S;

inputs are unavailable are skipped, and their outputs remain unavailable to downstream components.

Each processing function is a pure function that receives its inputs (as numeric arrays) and resolved parameters, and returns one or more named output series. Functions are registered in a static registry keyed by building block identifier. For example, a SMA function receives a numeric array and a period parameter and returns a single output series of the same length.

3.5.4 Chart Reconciliation

The chart panel follows a declarative rendering pattern: the application layer computes a complete description of the desired chart state (panes, layers, series data, and styling)

and passes it to a rendering adapter as a single snapshot. The adapter is responsible for reconciling this snapshot with the current state of the chart library.

Reconciliation distinguishes between *structural changes*, which require rebuilding the chart's pane and series structure (for example, when a pane is added or a series moves between panes), and *data-only changes*, which can be applied in place (for example, when execution produces new output values for an existing series). This distinction avoids unnecessary teardown and recreation of chart elements during routine updates, contributing to the responsiveness target of Requirement R6.

Panes are ephemeral: they are derived from the current set of probes and their pane assignments, rather than being stored as independent state. When a probe is assigned to a new pane, the projection recomputes the pane structure; when the last probe leaves a pane, the pane disappears. This derivation ensures that the chart structure is always consistent with the inspection state.

3.5.5 Backtesting

The backtesting facility evaluates a workflow's output signal against historical price data to produce simulated trading results. It uses a vectorized, signal-based model rather than an event-driven simulation.

Signal model. The workflow's output is interpreted as a position signal: non-zero values indicate that a position should be held, and zero values indicate no position. An *entry* occurs when the signal transitions from zero to non-zero, and an *exit* occurs when it transitions from non-zero to zero. To prevent look-ahead bias, signals are executed with a one-bar delay: a signal observed at bar n produces a trade at bar $n{+}1$. Only long positions are supported; the model does not consider short selling.

Trade detection and return calculation. Leading non-zero values at the start of the data (before any zero-to-non-zero transition) are ignored, requiring a clean entry signal before the first trade is counted. Trailing open positions at the end of the data are also not counted. Per-trade returns are computed as simple arithmetic returns: $(p_{\text{exit}} - p_{\text{entry}})/p_{\text{entry}}$. Cumulative returns are computed through geometric compounding: the product of $(1 + r_i)$ across all trades.

Equity curve and metrics. The equity curve starts at zero and steps at each trade close by the compounded return, providing a cumulative performance trace over time. The system reports the following aggregate metrics: cumulative return, number of trades, win rate, average return per trade, maximum drawdown, expectancy (probability-weighted average outcome), and profit factor (ratio of gross profits to gross losses).

Staleness detection. Backtest results are associated with a fingerprint computed from the workflow's execution-relevant fields: component instances (identities, definitions,

parameter overrides), connections, port bindings, and the selected asset. Cosmetic changes such as renaming a node or repositioning it on the canvas do not affect the fingerprint. When the fingerprint changes, existing results are marked as stale, prompting the user to re-execute the backtest.

CHAPTER 4

Results

This chapter evaluates the system described in Chapter 3 by demonstrating its capabilities on a representative workflow and assessing how well the design addresses the requirements and research questions. The discussion draws on the iterative development process and on feedback from a small number of users with programming backgrounds who used the editor to construct and inspect financial workflows. Section 4.1 presents an end-to-end case study. Section 4.2 evaluates the system against the design requirements. Section 4.3 discusses the findings in relation to each research question. Section 4.4 identifies limitations and open issues.

4.1 Case Study: Moving Average Crossover

This section demonstrates the system through a representative workflow: a moving average crossover strategy applied to historical price data. The scenario exercises the core editor features (graph construction, parameter configuration, asset binding) and the visual analysis features (probes, temporal navigation, coordinated views, execution highlighting, backtesting) in a realistic inspection task.

4.1.1 Workflow Construction

The workflow implements a simple moving average crossover, a common technical analysis pattern in which a shorter-period moving average crossing above a longer-period one is interpreted as a buy signal, and crossing below as a sell signal.

The user begins by selecting an asset from the asset browser, which supplies the OHLCV price series displayed as a candlestick chart in the main pane. From the component library, the user places two moving average nodes on the canvas: an exponential moving average (EMA) configured with a short period and a SMA configured with a longer period. Each node receives the asset's close price through a port binding. A *greater-than*

rule node compares the EMA output against the SMA output, producing a binary signal: non-zero when the shorter average is above the longer average, and zero otherwise. This signal serves as the workflow's output.

The resulting graph consists of four logical stages: data input (the bound asset), two parallel indicator computations (EMA and SMA), a comparison rule, and the output signal. The connections between these stages are visible on the canvas as directed edges, making the data dependencies explicit.

4.1.2 Intermediate-State Inspection

With the graph constructed, the user executes the workflow. Execution highlighting immediately reveals the active dataflow path: all four nodes display highlighted borders, the edges between them are rendered in a bright stroke, and the input ports receiving data show active indicators. If any node were disconnected or misconfigured, it would remain visually muted, signaling where the dataflow is interrupted.

To inspect intermediate values, the user creates two probes through the toolbar overlay and assigns them to the EMA and SMA nodes. The system routes both probes to the main chart pane, since their outputs are price-scaled, overlaying the two moving average lines directly on the candlestick chart. Each probe receives a distinct color, which appears consistently on the target node's border, the chart series line, and the pane watermark entry.

The user can now observe the relationship between the raw price data and the two derived indicators in a single coordinated view. Periods where the short average crosses above the long average are visually apparent as line intersections on the chart. Moving the crosshair across the time axis displays the exact values of all visible series at each bar, allowing the user to identify the precise bars at which crossover events occur.

4.1.3 Signal Verification and Debugging

To verify the rule node's output, the user creates a third probe and assigns it to the greater-than node. Because this output is a binary signal rather than a price-scaled value, the system places it in a separate measurement pane below the main chart. The user can now observe the binary signal alongside the indicator lines: bars where the EMA is above the SMA correspond to non-zero values in the signal pane, and the shared time axis ensures that this correspondence is visually aligned.

If the signal does not match the user's expectation, for example, if it remains at zero throughout the series, the probes on the upstream indicator nodes provide immediate diagnostic context. The user can check whether the two averages ever cross, whether the configured periods are appropriate for the selected asset's volatility, or whether a port binding is missing. This backward tracing through the graph, guided by probes placed at successive stages, is the primary debugging workflow. Nodes that perform logical operations or threshold comparisons are particularly important inspection targets, as

they can filter out large portions of a signal and thereby have a strong influence on the final output.

4.1.4 Backtesting and Iteration

With the signal verified, the user triggers a backtest. The system evaluates the binary output against the historical price data, applying a one-bar execution delay to prevent look-ahead bias. Two visual elements appear: trade markers (green arrows at entry points, red arrows at exit points) overlaid on the candlestick chart, and an equity curve in a dedicated backtest pane showing cumulative performance over time. A performance summary reports aggregate metrics including cumulative return, number of trades, win rate, and maximum drawdown.

The trade markers anchor each decision to a specific point on the time axis, allowing the user to cross-reference entries and exits with the indicator behavior visible through the probes. For example, the user can position the crosshair at a trade entry and verify that the EMA did indeed cross above the SMA at that bar, confirming that the workflow behaves as intended.

The combination of probed intermediate values and backtest results creates a feedback loop: the user can adjust parameters (for example, changing the short period from 10 to 20 bars), re-execute the workflow, and immediately observe how the modified indicators, the resulting signal, and the trading outcomes change together. This cycle of adjustment and observation supports iterative refinement without requiring the user to reason about the workflow's behavior from final outputs alone. Figure 4.1 shows the editor in this state.

4.2 Requirements Evaluation

Table 4.1 evaluates the system against the seven design requirements derived in Section 3.1.

The walkthrough in Section 4.1 demonstrates these capabilities working together: probes provide intermediate-state inspection (R1) within a shared temporal context (R2), linked to the graph through color encoding (R3) while preserving independent navigation (R4), with user-controlled layout flexibility (R5), responsive interaction (R6), and execution-based dataflow feedback (R7).

4.3 Discussion

This section discusses the design outcomes in relation to each research question, drawing on observations from iterative development and informal use.

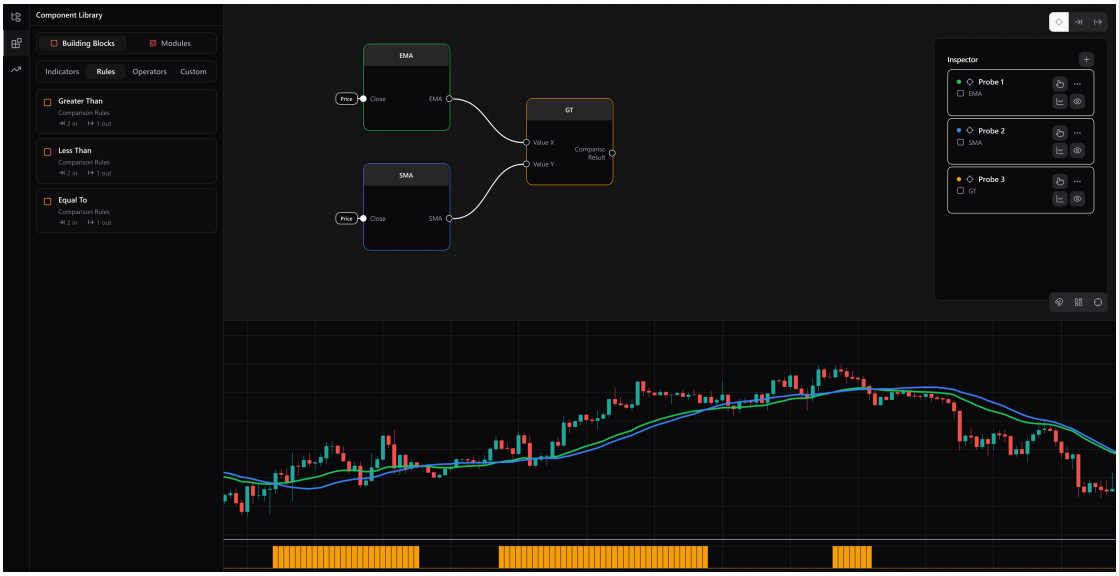

Figure 4.1: The editor showing a moving average crossover workflow with three probes active. The graph canvas shows the EMA and SMA indicator nodes and the greater-than rule node, each with a colored border indicating its probe assignment. The probe inspector (right) lists the three active probes. The chart panel displays the candlestick chart with both moving average overlays in the main pane and the binary comparison signal in a separate pane.

4.3.1 RQ1: Inspection and Understanding

The first research question asks which visualization and interaction mechanisms most effectively support users in examining intermediate data and temporal behavior within node-based workflows.

Full chart layers over inline previews. The system renders probe outputs as full chart layers rather than compact inline visualizations embedded in canvas nodes. This decision reflects the nature of the data: in financial time-series workflows, what matters is the precise values a node produces over time and how they relate to the values of other nodes. Users work with fluctuating signals, boolean comparisons, and threshold crossings where the temporal alignment of values across multiple nodes is central to understanding whether a workflow behaves correctly. Inline sparklines would show the shape of an output but not support the pointwise comparison that these tasks require. The shared time axis and crosshair allow users to read exact values at any bar across all probed nodes simultaneously, which was important for verifying logical and comparison operations.

Temporal scrubbing evaluated and rejected. During development, a temporal scrubber and playback controls were implemented and tested. The scrubber allowed

Requirement	Evaluation
R1. Per-node intermediate-state inspection	Probes can be assigned to any node's output port, capturing the resulting series as a chart layer. Up to two probes per node allow comparison within the same chart.
R2. Temporal navigation	All chart panes share a single time axis with synchronized pan, zoom, and crosshair. The crosshair displays values across all visible series.
R3. Linked graph and chart views	Probe assignment creates a color-coded link between a canvas node and a chart layer. The color appears on the node border, the series line, and the pane watermark.
R4. Independent view navigation	The graph canvas and chart panel maintain separate viewports. Panning or zooming one does not affect the other.
R5. Flexible time-series layout	Users control whether series share a pane or occupy separate panes by reassigning probes. Semantic routing provides defaults based on output type.
R6. Responsive interaction	Canvas interaction, chart navigation, and probe operations respond without perceptible delay. Only backtesting on large datasets introduces a brief computation.
R7. Dataflow visibility	Execution-based highlighting distinguishes active from inactive nodes, edges, and ports. Inactive nodes signal where the dataflow is interrupted.

Table 4.1: Evaluation of the system against the design requirements from Section 3.1. All requirements are met by the implemented system.

stepping through time points sequentially, following the timeline replay approach of Hoffswell et al. [HSH16], but was ultimately not included in the final system. However, this mechanism did not provide additional insight beyond what the static chart already offers. Because all temporal data is visible at once and probes expose intermediate state at every node, no dimension of the data remains hidden. The crosshair provides equivalent point-in-time inspection with less interaction overhead: the user moves the cursor to any bar of interest rather than advancing through time sequentially. The scrubber was replaced by the crosshair-based approach described in Section 3.4.3.

This outcome reflects a key difference between the present system and the reactive visualizations addressed by Hoffswell et al. [HSH16]: in reactive systems, internal state changes with each interaction event, making temporal replay necessary to observe past

states. In the present system, all temporal data is computed once and displayed in its entirety, so sequential playback adds no information beyond what direct navigation already provides.

Debugging through backward tracing. When a workflow produces unexpected output, the primary debugging approach is to place probes at successive upstream stages, working backward from the output toward the inputs. Nodes that perform logical operations or threshold comparisons are the most important inspection targets, as they can filter out large portions of a signal and have a disproportionate effect on the final result. Execution highlighting complements this process by revealing which nodes are active, allowing the user to quickly identify where the dataflow is interrupted before placing any probes.

4.3.2 RQ2: Linking and Tracing

The second research question asks how coordinated multi-view and visual tracing techniques can connect graph structure and time-based data to make signal flow and dependencies more transparent.

Color-coded linking. The color encoding that connects probe node borders to chart series lines and pane watermarks is the primary linking mechanism. During development, the initial design used small colored dots on nodes. Extending the color to the full node border made the correspondence between graph elements and chart layers more immediately visible, even with several active probes. This approach differs from traditional brushing and linking, where hovering in one view highlights elements in another. Because the color is persistent and does not depend on cursor position, users can inspect the chart without needing to simultaneously interact with the canvas.

User-controlled layout. The system supports both superposition (overlaying series within a shared pane) and juxtaposition (separating series into distinct panes), following the comparative visualization strategies described by Gleicher et al. [GAW+11] and the empirical findings of Javed et al. [JME10]. Users can reassign probes between panes at any time, choosing the layout that best suits the comparison at hand. Stacking price-scaled outputs in the main pane supports the local comparison tasks that Javed et al. [JME10] found to benefit from shared-space layouts, while separating differently scaled signals into their own panes avoids the clutter that shared-space views introduce when series differ in range. This flexibility is important because the appropriate layout depends on the specific inspection task, and no fixed arrangement can anticipate all comparison needs.

Execution-based dataflow highlighting. Highlighting active nodes, edges, and ports based on the execution result provides an immediate structural overview of where data flows through the graph. Because the highlighting is derived from actual execution output

rather than from graph topology alone, it reliably reflects which nodes produced data and which did not, supporting fault localization.

4.3.3 RQ3: Integration and Usability

The third research question asks how the described techniques can be combined in a coherent interface that remains clear, usable, and responsive.

Responsiveness. The system meets the performance targets established in Requirement R6. Canvas interaction, chart navigation, probe assignment, and crosshair movement all respond without perceptible delay. Workflow execution completes synchronously and produces updated chart data immediately, with no loading states required for typical workflows. Only backtesting on larger datasets introduces a brief computation period. These response characteristics are consistent with the findings of Liu and Heer [LH14], who showed that adding 500 milliseconds of latency significantly reduces exploratory activity and hypothesis generation.

Ad-hoc inspection over pre-configured views. The design initially considered a grouping mechanism that would allow users to define named sets of probes and toggle them as a unit. In practice, this added complexity without matching how the system was actually used: inspection tends to be exploratory and ad-hoc, with probes added and removed as understanding evolves. Toggling individual probe visibility proved sufficient, and the overhead of managing named groups did not justify its cost. This aligns with the parsimony guideline of Baldonado et al. [BWK00], which recommends minimizing view management overhead to keep cognitive load focused on the data.

Discoverability. Users unfamiliar with the system needed an introduction before they could work productively. The editor's interaction model, particularly probe creation and assignment, was not immediately self-explanatory. Users who understood data analysis concepts and had experience with visual tools adapted after a brief explanation, suggesting that the system's learning curve reflects the specificity of its interaction patterns rather than a fundamental usability problem. Improving discoverability through onboarding guidance or contextual hints remains an opportunity for future work.

4.4 Limitations

This section discusses the boundaries of the current system and its evaluation.

The evaluation is based on a case study, iterative development experience, and informal use rather than a controlled experiment with structured tasks and quantitative measures. While the case study demonstrates that the system functions as designed and the discussion identifies concrete design outcomes, a formal evaluation with a larger participant group would be needed to generalize claims about usability and effectiveness.

The workflows used during development and informal evaluation were of moderate complexity. The system's behavior with significantly larger graphs, involving many nodes, connections, and concurrent probes, has not been systematically tested. Performance and visual clarity may degrade at larger scales. Related to this, the system was designed to support hierarchical composition through reusable modules, which would allow users to encapsulate sub-workflows and use them as single nodes in larger graphs. This capability was not implemented within the scope of this thesis. Without it, larger workflows lack the structural abstraction needed to manage complexity, as every processing step must be represented as an individual node on the canvas.

Each workflow operates on a single asset's price series. Multi-asset workflows, where signals from different instruments are combined, are not supported, limiting the range of analysis tasks the system can address.

Finally, the system does not support user-created annotations such as bookmarks, text notes, or graphical marks on the chart. Heer et al. [HVW07] demonstrated the value of such mechanisms for capturing and revisiting analysis findings. Only system-generated markers (backtest trade entry and exit points) are available.

CHAPTER 5

Conclusion

This thesis investigated how interactive visualization techniques can support users in understanding and debugging node-based financial dataflow workflows. Following the Data–Users–Tasks design triangle of Miksch and Aigner [MA14], the work derived seven design requirements from prior research, designed and implemented a node-based editor with probes, coordinated graph and chart views, execution-based dataflow highlighting, and temporal navigation, and evaluated the system through a case study and iterative use.

The central finding is about the relationship between the data model and the choice of inspection techniques. The literature on visual debugging, particularly Hoffswell et al. [HSH16], suggests that temporal scrubbing is valuable for understanding time-varying behavior. In this system, however, scrubbing provided no additional insight. The reason is that in reactive visualization systems, internal state changes with each interaction event, so replay is necessary to observe past states. In the present system, the full time series is computed once and displayed in its entirety, and probes expose intermediate state at every node. No part of the data remains hidden, and stepping through time sequentially adds nothing over direct navigation with the crosshair. This suggests that inspection techniques should be chosen based on the computational model of the system, rather than adopted from systems with different execution semantics.

Looking at this through the design triangle, the *data* vertex constrained the design more than expected. The *users* and *tasks* vertices suggested that scrubbing and interactive brushing and linking would help, but the static, fully observable nature of the data made both redundant. Persistent color-coded linking and crosshair-based navigation turned out to be sufficient, reducing interaction overhead without losing coordination. The design triangle was useful here not as a checklist, but as a way to surface the conflict between what the task analysis suggested and what the data characteristics actually required [MA14].

For RQ1, probes with full chart layers were chosen over inline previews, because financial time-series inspection requires precise value comparison across temporally aligned series rather than visual summaries of output shape. For RQ2, persistent color encoding across the canvas and chart panel provided coordination without the interaction cost of hover-based brushing and linking, and user-controlled pane assignment supported both superposition and juxtaposition depending on the task [GAW+11, JME10]. For RQ3, ad-hoc probe management fit the exploratory nature of inspection better than pre-configured view groups, consistent with the parsimony guideline of Baldonado et al. [BWK00], and the system met the sub-100-millisecond perceptual fusion threshold [CMN83] that Liu and Heer [LH14] showed to be critical for sustained exploration.

5.1 Future Work

The most immediate gap is evaluation. A controlled study comparing probed and non-probed workflows on debugging tasks would test whether the inspection features actually reduce error rates and task completion time.

The system currently operates on static, pre-computed data. Financial analysis increasingly involves streaming data and parameters that change in real time. Under such conditions, the full time extent is no longer available at once, and the argument against temporal scrubbing no longer holds. Investigating at what point temporal replay becomes necessary again would generalize the present finding beyond this specific system.

On the design side, hierarchical composition through reusable modules would let users manage larger workflows through structural abstraction. User-created annotations, as described by Heer et al. [HVW07], would help users capture and revisit analysis findings across sessions. Both extensions address limitations identified in Section 4.4 and would broaden the range of workflows the system can support.

Overview of Generative AI Tools Used

AI-based tools were used during the development and writing of this thesis. Claude (Anthropic; models Claude 3.5 Sonnet and Claude 4 Opus) was used for code development of the editor, for literature research and extraction of thesis-relevant content from papers, and as a writing assistant during thesis composition.

List of Figures

List of Tables

List of Algorithms

Glossary

brushing and linking An interaction technique in which selecting or highlighting elements in one view automatically highlights corresponding elements in other coordinated views. 12, 14, 15, 27, 40, 43, 44

dataflow A computational model in which data passes through a directed graph of processing nodes, where each node performs a transformation on its inputs and forwards the results to connected outputs. 2–4, 7–10, 12, 13, 15, 21, 43

node-based editor A visual environment in which users construct workflows by placing and connecting discrete processing components (nodes) on a canvas, forming a directed graph. 1, 2, 4, 5, 7, 9, 10, 12–15, 17, 18, 20, 43, 47

probe A named measurement point that targets a specific output port of a node, causing its output series to be captured and displayed as a chart layer in the time-series panel for temporal inspection. 3, 4, 11, 18, 19, 22, 25–30, 32, 36–44, 47

visual tracing Highlighting or animating propagation paths through connected nodes to reveal how signals flow through a workflow. 4, 11, 15

Acronyms

CMV Coordinated Multiple Views. 1, 4, 12, 15

DAG Directed Acyclic Graph. 8, 21

OHLCV Open-High-Low-Close-Volume. 4, 8, 24, 25, 35

SMA Simple Moving Average. 8, 23, 31, 35–38, 47

Bibliography

[BCD+07] Michael R. Berthold, Nicolas Cebron, Fabian Dill, Thomas R. Gabriel, Tobias Kötter, Thorsten Meinl, Peter Ohl, Christoph Sieb, Kilian Thiel, and Bernd Wiswedel. KNIME: The Konstanz Information Miner. In *Data Analysis, Machine Learning and Applications (GfKl 2007)*, Studies in Classification, Data Analysis, and Knowledge Organization, pages 319–326. Springer, 2007.

[BFD+16] Leilani Battle, Danyel Fisher, Robert DeLine, Mike Barnett, Badrish Chandramouli, and Jonathan Goldstein. Making sense of temporal queries with interactive visualization. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems*, 2016.

[Ble25] Blender Online Community. Blender – a 3D modelling and rendering package. Blender Foundation, Amsterdam, 2025.

[BWK00] Michelle Q. Wang Baldonado, Allison Woodruff, and Allan Kuchinsky. Guidelines for using multiple views in information visualization. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, pages 110–119, 2000.

[CMN83] Stuart K. Card, Thomas P. Moran, and Allen Newell. *The Psychology of Human-Computer Interaction*. Lawrence Erlbaum Associates, 1983.

[DCE+13] Janez Demšar, Tomaž Curk, Aleš Erjavec, Črt Gorup, Tomaž Hočevar, Mitar Milutinovič, Martin Možina, Matija Polajnar, Marko Toplak, Anže Starič, Miha Štajdohar, Lan Umek, Lan Žagar, Jure Žbontar, Marinka Žitnik, and Blaž Zupan. Orange: Data mining toolbox in Python. *Journal of Machine Learning Research*, 14(35):2349–2353, 2013.

[Epi25] Epic Games. Unreal engine, 2025.

[GAW+11] Michael Gleicher, Danielle Albers, Rick Walker, Ilir Jusufi, Charles D. Hansen, and Jonathan C. Roberts. Visual comparison for information visualization. *Information Visualization*, 10(4):289–309, 2011.

[GTPB19] Anna Gogolou, Theophanis Tsandilas, Themis Palpanas, and Anastasia Bezerianos. Comparing similarity perception in time series visualizations. *IEEE Transactions on Visualization and Computer Graphics*, 25(1):388–400, 2019.

[HKA09] Jeffrey Heer, Nicholas Kong, and Maneesh Agrawala. Sizing the horizon: The effects of chart size and layering on the graphical perception of time series visualizations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2009.

[HSH16] Jane Hoffswell, Arvind Satyanarayan, and Jeffrey Heer. Visual debugging techniques for reactive data visualization. *Computer Graphics Forum*, 35(3):271–280, 2016.

[HVW07] Jeffrey Heer, Fernanda B. Viégas, and Martin Wattenberg. Voyagers and voyeurs: Supporting asynchronous collaborative information visualization. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2007.

[JME10] Waqas Javed, Bryan McDonnell, and Niklas Elmqvist. Graphical perception of multiple time series. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):927–934, 2010.

[Kah62] Arthur B. Kahn. Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562, 1962.

[Kod20] Jeffrey Kodosky. LabVIEW. *Proceedings of the ACM on Programming Languages*, 4(HOPL):1–54, 2020.

[KPHH11] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. Wrangler: Interactive visual specification of data transformation scripts. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems*, 2011.

[LH14] Zhicheng Liu and Jeffrey Heer. The effects of interactive latency on exploratory visual analysis. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2122–2131, 2014.

[MA14] Silvia Miksch and Wolfgang Aigner. A matter of time: Applying a data–users–tasks design triangle to visual analytics of time-oriented data. *Computers & Graphics*, 38:286–290, 2014.

[Mar17] Robert C. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall, 2017.

[Rob07] Jonathan C. Roberts. State of the art: Coordinated & multiple views in exploratory visualization. In *Proceedings of the Fifth International Conference on Coordinated and Multiple Views in Exploratory Visualization*, pages 61–71, 2007.

[SCHP23] Nischal Shrestha, Bhavya Chopra, Austin Z. Henley, and Chris Parnin. Detangler: Helping data scientists explore, understand, and debug data wrangling pipelines. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 102–110, 2023.

[Sto25] StockSharp LLC. StockSharp – algorithmic trading and quantitative trading open source platform, 2025.

[Tra25] TradingView. TradingView – Pine Script language reference manual, 2025.