ChatGPT

Offline-First Market Monitor GUI Design

This report consolidates research for the GUI component of an offline-first stock-market monitor. The goal is to keep the GUI layer thin: it should orchestrate the existing pipeline (ingestion → features → gates/flags → scoring → reporting) and never re-implement core logic. The app must support reproducible runs with a *run_id* concept, local data caches and offline usage, transparent system status, power-user navigation, high-performance data tables, intuitive config editing, explainability for ranking, secure secret storage and accessible error handling. Each "bridge" below summarises options and recommendations.

Bridge A – UI framework choice & Windows packaging

A desktop GUI should remain independent of the engine and use a **ViewModel** boundary (e.g., MVVM) so the core engine stays testable. Several frameworks were evaluated for cross-platform desktop apps with solid Windows packaging support:

Framework	Pros & evidence	Cons/notes
Avalonia (cross-platform .NET)	Open-source UI framework similar to WPF with MVVM patterns. It draws its own UI (not OS controls) and supports Windows, Linux, macOS and mobile/Web. Documentation stresses separation of UI and logic and a solution structure with a core project and platform-specific projects [1].	Developing custom components may be necessary; smaller community than Electron/Flutter.
Electron (JavaScript/ Node)	Embeds Chromium and Node.js. Supports cross-platform packaging; community tools generate `.msi` installers for Windows and `.dmg`/`.rpm` for macOS/Linux [2]. Built-in auto-update and access to NPM ecosystem (e.g., React).	Apps bundle the whole browser, increasing size; heavy memory usage; security concerns if Node integration isn't sandboxed.
Tauri (Rust/JS)	Uses Rust for the backend and a web front-end (React/Svelte/etc.). The CLI builds platform-specific installers; on Windows, `.msi` via WiX Toolset or NSIS installers can be created [3]. `tauri build` bundles the frontend and resources into a single binary [4]. Lightweight and fast because it leverages the host's web engine rather than bundling Chromium.	Cross-compiling Windows `.msi` packages only works on Windows; Windows 7 support requires embedding WebView2 bootstrapper [5].

Framework	Pros & evidence	Cons/notes
Flutter	Google's framework; uses Dart and supports Windows via Flutter Desktop. Build output can be packaged into an installer using tools like **Inno Setup** or **WiX** [6]. Example packaging guides show including the compiled `.exe`, necessary DLLs (e.g., Visual C++ runtime), and data folder in the installer script [7].	Mature mobile ecosystem but desktop still evolving; heavy binary size; limited ability to run native .NET code.
.NET MAUI	Microsoft's cross-platform framework; uses C# and XAML; can package a Windows desktop app as an **MSIX** or unpackaged executable via Visual Studio. The Package.appxmanifest configures the app for Windows Store or side-loading [8].	Only runs on .NET; community smaller than Electron; cross-platform support still evolving.

Recommendation: For a thin, MVVM-friendly GUI that re-uses .NET skills from the engine, **Avalonia** or **.NET MAUI** provide strong separation of concerns and native MVVM patterns. **Avalonia** runs on Windows/macOS/Linux and has a clear solution structure separating core and platform layers [1]. Packaging as an **MSIX** (for MAUI) or **.msi** (for Avalonia/Electron/Tauri) provides a professional installation experience. For teams comfortable with web technologies and requiring advanced web components (e.g., React), **Tauri** offers a lightweight alternative with Rust for secure backend logic [4].

Packaging best practices

- **Deterministic builds**: ensure reproducibility; embed run manifest (config hash, counts, timestamps) into the installer metadata.
- **Installer choices**: use WiX to produce `.msi` bundles; include the compiled binary, resources, and any required dependencies (e.g., Visual C++ runtime for Flutter apps [7]). Provide an **auto-update mechanism** (Electron's `autoUpdater` or Tauri's update plugin).
- **Windows Store distribution**: for .NET MAUI and Tauri, packaging as MSIX allows publishing to Microsoft Store; ensure digital signature.

Bridge B – Table virtualization strategy for 10k+ rows

The Universe/Scored view must handle tens of thousands of stock symbols without choking the UI. Two types of virtualization exist:

- **UI virtualization**: only renders visible row containers while all data resides in memory (supported by controls like `VirtualizingStackPanel` in WPF). Microsoft warns that virtualization can be disabled if you wrap items in `StackPanel` or add item containers yourself [9].
- **Data virtualization**: stores only visible data in memory; fetches pages on demand. Large data sets (10k–1M items) benefit from virtualization; the ComponentOne article notes that memory footprint is critical and virtualization allows viewing a billion rows with smooth scrolling [10].

Recommended approach

1. **Use an off-the-shelf virtualized table component**:

2. For web (Electron/Tauri) UI, choose a React table library with built-in virtualization (e.g., **Mantine React Table**). It renders only visible rows and columns; documentation suggests enabling virtualization for datasets over 100 rows [11] and provides column chooser, sorting, filtering, and sticky columns.
3. For .NET desktop (Avalonia/WPF/MAUI), use a **virtualized data grid** such as ComponentOne's FlexGrid or Telerik's DataGrid. They implement UI and data virtualization. When building custom lists, set `ItemsPanel` to `VirtualizingStackPanel` and enable `IsVirtualizing=True` [9].
4. **Asynchronous data loading**: fetch pages of data asynchronously from the local repository. Provide search, filtering and grouping operations client-side, but push expensive computations (scoring, gating) to the engine.
5. **Sticky columns & saved views**: freeze key columns (Symbol, Score, Flags, last_date) and persist column order/visibility in local settings. Provide quick filter presets and row detail expansion (e.g., right-click → details pane).
6. **User feedback**: display row counts, virtualization status and progress when loading more rows; show skeleton loaders to convey activity.

Bridge C – Charting library for fast time-series rendering

The symbol-detail page needs interactive charts (candlestick, technical indicators, features). Performance matters because the engine may produce high-resolution time series and computed features.

Library	Strengths	Evidence
SciChart JS / SciChart WPF	Commercial library for web and .NET; uses WebAssembly and GPU acceleration. Claims to render millions of points at 60 FPS [12] and that dashboards with 100 million points can run for days without freezing [13]. Supports real-time streaming, candlesticks, annotations and custom series.	Paid licensing; complexity for simple charts.
LightningChart JS	High-performance JS charting with proprietary GPU rendering. Handles billions of points; supports real-time plotting of 400 trends at 1 kHz while maintaining 60 FPS [14]. Includes a Trader Edition with candlestick charts and technical indicators [15].	Proprietary; pricing may be higher; JavaScript only.
Chart.js + plugins	Popular open-source JS library. For large datasets, documentation advises normalizing and sorting data, disabling parsing and using the **decimation plugin** to downsample points [16]. Suitable for moderate dataset sizes but may struggle with tens of thousands of points.	Good for general charts; limited performance for heavy time-series without decimation; limited interactive features compared with SciChart/LightningChart.

Library	Strengths	Evidence
OxyPlot (.NET)	Cross-platform plotting library for WPF/MAUI/WinForms. Performance guidelines note that adding points directly is faster than binding and that solid lines and unfilled markers render faster [17]. Good for moderate data sizes; open source.	Does not offer GPU acceleration; may lag with millions of points.

Recommendation: For **web-based UI** (Electron/Tauri), choose a GPU-accelerated library such as **SciChart JS** or **LightningChart JS** for large time series and candlestick charts. For **native .NET** apps, **SciChart WPF** or **OxyPlot** can be embedded; SciChart provides high-frequency rendering while OxyPlot suits simpler charts. Implement down-sampling (decimation) or progressive loading for extremely long series.

Bridge D – Run orchestration & progress reporting

The GUI must trigger engine runs (ingestion → features → scoring) and report progress without freezing. A proper pattern involves asynchronous tasks, ViewModels and throttled progress updates.

Patterns

1. **Async tasks with progress reporting**: Use `async/await` and **IProgress<T>** or a similar pattern. A CodeProject example shows a `TaskManagement` class exposing a `TaskProgressViewModel` that tracks task state and subtasks while raising property-changed events to update the UI [18]. Users can start, pause or cancel tasks and observe progress bars [19].
2. **Throttled updates**: Frequent progress updates can freeze the UI. Brian Lagunas demonstrates generating 1 M items and reporting progress every 1 500 items; throttling prevents the UI from being overwhelmed [20]. Throttle progress events (e.g., update every 0.1 seconds or N items).
3. **Report partial results**: Provide hooks for the engine to emit intermediate artifacts (e.g., partial score tables). Use `IProgress<ProgressUpdate>` or event streams so the ViewModel collects these updates and dispatches them to the UI.
4. **Run timeline & diffing**: Persist run manifests (config hash, timestamps, counts) in the engine's output. The GUI can display a timeline of past runs, allow selection of two runs and compute diffs (items moved, score changes, new flags). Each run should be reproducible by referencing the exact config and data snapshot.
5. **System status & offline awareness**: The orchestrator should expose state (last data date per ticker, lag days, cache health, offline/online). This aligns with Nielsen's heuristic of providing system visibility and error recovery. Display status in the dashboard and allow refreshing local data when online.

Bridge E – Config editor & schema validation UX

An effective settings editor must allow users to tweak gates, weights and theme rules without breaking validity. Research into schema-driven editors yields the following guidelines:

- **Schema-aware GUI**: MetaConfigurator, a tool for editing YAML/JSON, uses the Ajv JSON Schema validator to provide immediate feedback when a value violates constraints; errors are highlighted in the editor [21] . A panel displays data as a tree/table, letting users expand levels and zoom into sub-properties [22] .
- **Widget mapping**: The tool maps JSON schema keywords to UI components: strings → text fields; numbers/integers → numeric fields with increment/decrement; booleans → checkboxes; enums → dropdowns; required fields → red asterisks; deprecated fields → strikethrough; anyOf/oneOf → multi-select or dropdown; arrays → plus/minus buttons; objects → collapsible panels [23] . Schema information (format, min/max, description) appears as tooltips; validation errors show red messages with icons.
- **Incremental modeling**: The Altova blog notes that graphical schema editors improve development speed, reduce errors, enable incremental modeling of complex schemas and auto-generate documentation [24] .

Proposed design

- Use a **schema-driven form generator** (e.g., `react-jsonschema-form` for web or a .NET equivalent). Provide a YAML/JSON editor with syntax highlighting and a form view; keep them synchronized.
- **Real-time validation**: Use Ajv or `System.Text.Json` with JSON Schema Draft 2020–12 to validate on each change and highlight errors. Provide tooltips with allowed ranges and defaults.
- **Guardrails**: Mark required and deprecated fields (e.g., strikethrough). Warn users when altering thresholds to extreme values (e.g., lowering liquidity too much). Support import/export of config profiles and diffing against defaults.
- **Undo/redo and safety**: Provide undo/redo history and prompt the user before applying changes. Validate entire config on save; if invalid, disable run start.

Bridge F – Explainability patterns for ranking systems

Stock scoring and ranking algorithms can be opaque; providing explanations increases user trust. Research indicates several patterns:

- **Feature contribution explanations**: The ShaRP paper emphasises that ranking outcomes require specialized explainability methods. ShaRP uses Shapley values to compute feature contributions relative to a ranking's profit function, producing explanations such as which features increased or decreased a symbol's position [25] . Integrating such methods allows the engine to compute per-symbol contributions and pass an "explain pack" to the GUI.
- **Why this result** overlays*: Search UIs often include a "why this result"* preview; for example, Alexander Street's UI displays a pop-over explaining why a search result was returned [26] . Similarly, the GUI should allow users to click or hover on a symbol to see why it is ranked: highlight flags that passed/failed, show contributions to the final score and list themes or evidence strings.

- **Citing sources & reasoning**: A Stack AI article argues that AI systems should cite underlying documents, separate factual information from suggestions and explain why a result is relevant by listing signals or filters used to rank [27]. In the market monitor, show which data points (e.g., last price, liquidity, theme matches) were used and display supporting evidence (news text snippets, analogies) along with their timestamps.

UI design for explainability

- **Explain panel**: A dedicated section in the symbol detail page summarises the top features contributing to the score (positive and negative). The engine returns a list of features with Shapley values; the GUI visualises them as horizontal bars or waterfall charts. Provide tooltips explaining each feature.
- **Gates & flags**: Show which gates (e.g., minimum liquidity, price range) passed or failed; list flags raised (e.g., extended period of lag). Clicking a flag reveals details.
- **Theme labels & analog evidence**: Show detected themes (e.g., "AI", "cybersecurity") and their evidence strings (sentences from documents). Indicate the recency of the underlying news.
- **Rank changes**: In run comparisons, highlight how a symbol's rank changed and which features drove the change.

Bridge G – Secure local secret storage

Offline-first design means storing API keys locally; these keys must be encrypted and never logged. Research into OS-integrated credential storage provides guidance:

- **Electron safeStorage**: Electron's `safeStorage` module encrypts strings using OS cryptography. On macOS it uses Keychain; on Windows it uses **DPAPI** which generates keys tied to the user account; on Linux it stores keys in secret stores (KWallet, GNOME libsecret) and falls back to unprotected text if none is available [28]. Use this API when building an Electron/Tauri frontend to encrypt secrets before storing them on disk.
- **OS credential stores**: Tools such as R's `keyring` package interface with the OS credential store (macOS Keychain, Windows Credential Store, Secret Service on Linux). Credentials are stored in a "keyring" protected by the user's login; retrieval functions decrypt secrets on demand [29]. .NET apps can call DPAPI directly to encrypt data; DPAPI uses per-user keys, and copying encrypted data across machines is not possible [30].
- **DPAPI best practices**: A StackExchange answer recommends encrypting secrets via DPAPI and storing them in a file under LocalAppData. Credential Manager just wraps DPAPI; thus reading/writing encrypted bytes yourself is sufficient [30]. When the user changes password, DPAPI transparently re-encrypts the keys [31].
- **Linux secret service**: On Linux desktops, secrets are managed by `gnome-keyring`/`libsecret` via the Secret Service API. For sandboxed applications (e.g., Flatpak), apps may need to request an encryption key from a portal and implement local encrypted storage when the secret service is unavailable.

Implementation recommendations

- **Abstraction layer**: Create a secret storage service with methods `setSecret(key, value)` and `getSecret(key)` implemented separately for each platform.

- **Windows**: Use `ProtectedData.Protect` and `Unprotect` (DPAPI) to encrypt and decrypt strings; store the cipher text in the user's LocalAppData folder. Provide a fallback to Windows Credential Manager if preferred but note that DPAPI suffices.
- **macOS**: Use Keychain via `security` CLI or frameworks; in Electron use `safeStorage`, in .NET use `SecurityFoundation` or `MacKeychain` wrappers.
- **Linux**: Use `libsecret`/`gnome-keyring` via bindings. For Tauri/Electron, check whether `safeStorage.getSelectedStorageBackend()` returns `basic_text`; if so, warn the user that secrets may not be secure [28]. Provide an option to store secrets encrypted locally using AES with a key derived from a user password.
- **No secrets in logs**: Scrub logs and telemetry; never display secrets in UI. Use separate configuration files for sensitive values and treat them as environment variables when possible.

Bringing it all together

With these research findings, the team should implement a thin, orchestrating GUI that leverages existing engine modules via a clear ViewModel boundary. Use a cross-platform framework (Avalonia/Tauri) with virtualization-enabled tables, GPU-accelerated charts and robust progress reporting. Provide a schema-driven settings editor with validation and guardrails. Include explainability features for rankings and secure secret storage integrated with the OS. By adopting these guidelines, the offline-first market monitor can deliver a powerful yet maintainable UI that keeps the engine testable and transparent.

---

[1] Setting Up A Cross Platform Solution | Avalonia Docs
https://docs.avaloniaui.net/docs/guides/building-cross-platform-applications/solution-setup

[2] Build cross-platform desktop apps with JavaScript, HTML, and CSS | Electron
https://www.electronjs.org/

[3] [4] [5] Windows Installer | Tauri v1
https://v1.tauri.app/v1/guides/building/windows/

[6] Building Windows apps with Flutter
https://docs.flutter.dev/platform-integration/windows/building

[7] Packaging and Distributing Flutter Desktop Apps : The Missing Guide for Open Source & Indie Developers — Creating Windows .exe installer [Part 2 of 3] | by Flutter Gems | Medium
https://medium.com/@fluttergems/packaging-and-distributing-flutter-desktop-apps-the-missing-guide-part-2-windows-0b468d5e9e70

[8] Publish a .NET MAUI app for Windows - .NET MAUI | Microsoft Learn
https://learn.microsoft.com/en-us/dotnet/maui/windows/deployment/overview

[9] Optimize control performance - WPF | Microsoft Learn
https://learn.microsoft.com/en-us/dotnet/desktop/wpf/advanced/optimizing-performance-controls

[10] How to Load a Billion Rows in a WPF Datagrid | ComponentOne
https://developer.mescius.com/blogs/how-to-load-a-billion-rows-in-a-wpf-datagrid

[11] Virtualization Feature Guide - Mantine React Table Docs
https://www.mantine-react-table.com/docs/guides/virtualization

[12] [13] Advanced JavaScript Chart and Graph Library | SciChart JS
https://www.scichart.com/javascript-chart-features/

[14] [15] High-Performance JavaScript Charts | WebGL JS Charts Library
https://lightningchart.com/js-charts/

[16] Performance | Chart.js
https://www.chartjs.org/docs/latest/general/performance.html

[17] Performance guidelines — OxyPlot 2015.1 documentation
https://oxyplot.readthedocs.io/en/latest/guidelines/performance.html

[18] [19] ASYNCHRONOUS TASK WITH MVVM AND PROGRESS BAR - CodeProject
https://www.codeproject.com/articles/Task-MVVM-ProgressBar

[20] Does Reporting Progress with Task.Run Freeze Your WPF UI? - Brian Lagunas
https://brianlagunas.com/does-reporting-progress-with-task-run-freeze-your-wpf-ui/

[21] [22] [23] DIGIN_2025_paper_2.pdf
https://is.ijs.si/wp-content/uploads/2025/09/DIGIN_2025_paper_2.pdf

[24] 5 Reasons to Choose a Graphical JSON Schema Editor
https://www.altova.com/blog/2023/09/5-reasons-to-choose-a-graphical-json-schema-editor

[25] ShaRP: Explaining Rankings and Preferences with Shapley Values
https://www.vldb.org/pvldb/vol18/p4131-stoyanovich.pdf

[26] Help | Alexander Street, part of Clarivate
https://search.alexanderstreet.com/help

[27] How AI Systems Remember Information in 2026
https://www.stack-ai.com/blog/how-ai-systems-remember-information-in-2026

[28] safeStorage | Electron
https://www.electronjs.org/docs/latest/api/safe-storage

[29] Securing Credentials – Solutions
https://solutions.posit.co/connections/db/best-practices/managing-credentials/

[30] [31] Where to store secrets in .NET applications? - Information Security Stack Exchange
https://security.stackexchange.com/questions/215690/where-to-store-secrets-in-net-applications