



# TRQuant 平台模块可视化、数据库整合与工作流设计优化方案

## 引言

TRQuant量化投资平台已经构建了9步量化投资工作流的基础架构，包括数据、因子、策略、回测、优化等核心模块，并通过MCP-server进行模块调用。然而，为进一步提升系统的健壮性和工业级应用水平，需要对各模块功能边界、数据库整合方式、数据流依赖、前端可视化组件和整体工程结构进行优化。本文将结合当前业界最佳实践，对模块职责接口、数据库与缓存策略、模块数据流与工作流引擎、前端可视化框架选型与状态管理以及工程结构提出清晰可实施的优化方案。

## 模块功能边界与接口规范

各功能模块应职责单一、接口清晰，以便在工作流中松耦合协同。下面定义数据、因子、策略、回测、优化、工作流等模块的功能边界和输入/输出规范：

- **数据模块（Data）**：负责市场原始数据的采集、清洗与标准化存储。输入包括行情数据源配置、符号列表和时间范围等，输出为结构化的时间序列数据（如Pandas DataFrame）或数据库中相应表格。数据模块应提供统一的数据获取接口，例如 `get_price(symbol, start_date, end_date)` 等，以供因子和策略模块调用。数据模块将原始数据存入后端时序数据库，以便后续分析模块高效查询<sup>①</sup>。同时可实现基本缓存，如最近行情快照缓存在Redis以降低重复数据拉取延迟<sup>②</sup>。
- **因子模块（Factor）**：负责基于输入的数据计算alpha因子或指标信号。输入通常是多维行情数据（价格、成交量、财务指标等）或从数据模块获取的时序数据，输出为因子值时间序列（例如一个资产×时间的矩阵）。因子模块应定义清晰的接口如 `compute_factor_X(data)`，内部确保点位时序一致性（避免未来函数）。输出的因子可暂存于时序数据库的因子表中<sup>③</sup>，方便重复使用和横截面分析。例如业界常使用类似Quantopian Pipeline的方式，对历史数据进行打分排序<sup>④</sup>。<sup>⑤</sup> 因子模块边界应仅关注因子本身的计算，不直接参与交易逻辑。
- **策略模块（Strategy）**：根据数据和因子信号制定交易决策。输入是行情数据及因子信号（可由因子模块输出），输出为交易指令或持仓权重（如每日调仓信号）。策略模块应通过定义**策略类接口**实现，例如每个策略类包含 `generate_signals(data, factors)` 方法，根据规则生成买卖信号。为了标准化，可采用策略模式设计：所有策略继承统一基类，提供初始化参数、调仓频率、持仓计算等接口<sup>⑥</sup><sup>⑦</sup>。策略模块依赖因子模块的结果但不关心因子如何计算，使得策略和因子解耦。策略输出的信号将传递给回测模块执行模拟。
- **回测模块（Backtest）**：负责在历史数据上模拟策略的执行，评估业绩指标。输入包括**策略实例**（含参数配置）和历史行情数据（可由数据模块提供），输出为回测结果，包括资金曲线、收益率、风险指标和交易明细等。回测模块对策略调用其信号生成方法，逐步推进时间并撮合交易，需考虑交易成本、滑点和风险控制等逻辑。接口设计上，可提供 `run_backtest(strategy, data)` 方法返回包含关键绩效指标的结果

对象。例如QuantConnect的Lean引擎和QuantRocket的Moonshot均采用类似接口来执行策略回测并返回指标<sup>6</sup><sup>7</sup>。回测模块应将净值曲线和交易日志持久化到分析数据库中，以供后续查询和可视化<sup>1</sup>。此外，模块内部应实现事件驱动撮合或向量化回测引擎，以平衡仿真精度和性能。

- **优化模块（Optimizer）**：负责策略参数优化和组合优化。输入可以是策略策略参数空间定义或多策略组合，调用回测模块对不同参数组合进行批量回测，输出为**最佳参数集**或策略组合及其绩效比较。优化模块需要协调并行执行，可通过任务队列将回测任务分发到多个工作进程并行计算<sup>2</sup>。接口如 `optimize(strategy, param_grid)` 返回参数最优解和对应评估指标。优化模块也可用于**组合权重优化**（如利用均值-方差或风险平价模型），这时输入为资产历史收益数据，输出为权重配置。优化结果同样存储于数据库（如保存每次参数组合及绩效）以备分析和审计。模块应确保避免过拟合，可集成交叉验证、滚动窗口回测等功能提升稳健性。
- **工作流引擎（Workflow）**：负责串联上述模块，定义**任务依赖顺序和执行逻辑**，构成完整的量化研究流程。工作流引擎接收用户选择的流程配置（例如9步流程各步骤的参数），按照依赖关系依次触发各模块调用，并监控执行状态。典型流程包括：**数据准备 → 因子计算 → 策略信号生成 → 回测 → 结果分析/优化**，部分步骤可迭代循环。工作流引擎应提供DAG（有向无环图）或序列流程定义机制，保证例如只有因子计算完成后才执行策略回测等。实现上，可借鉴Airflow、Prefect等的调度思想，或使用VS Code扩展界面让用户以**可视化方式**调整工作流顺序。为了健壮性，工作流引擎需捕捉各模块异常并提供恢复或跳过机制，记录审计日志<sup>8</sup>。此外，可结合Redis消息队列，将任务分发给后台worker执行并通过发布/订阅推送进度，从而实现异步、可监控的流程执行<sup>2</sup>。这种解耦方式符合现代微服务架构，通过消息总线来编排任务而非紧耦合调用<sup>9</sup>。

上述模块通过清晰的输入输出契约和松耦合设计，既可各司其职，又能在工作流中顺畅衔接。例如，一次完整流程中，数据模块获取行情数据后存入时序库，因子模块读取数据计算出因子暴露矩阵，然后策略模块使用这些因子产生买卖信号，回测模块据此执行模拟交易并输出绩效指标，优化模块再根据回测结果调整策略参数，如此形成闭环。

**模块接口规范示例**：定义统一的数据结构和接口协议。例如可约定所有时间序列数据以Pandas DataFrame（索引为日期，列为资产）传递，因子输出为类似结构且带有因子名称元数据。策略信号可用DataFrame或策略类自带的持仓字典结构。回测结果采用对象属性或字典包含 `equity_curve`, `trades`, `statistics` 等字段。通过一致的数据接口格式，保证模块之间衔接方便且减少转换开销。文档中应清晰列出每个模块的输入字段、输出字段及格式，供开发者和AI辅助编排时查阅。

## 底层数据库连接与缓存策略

TRQuant采用**分层多存储（Polyglot Persistence）**架构，根据数据类型选择最合适的数据库，以在性能和成本之间取得平衡<sup>10</sup>。各类数据库的定位及整合策略如下：

- **PostgreSQL – 事务主库**：使用PostgreSQL存放需要强一致性的**事务型数据**<sup>11</sup>。包括策略配置和版本、风控审批流程、实盘账户和交易流水、审计日志等关系数据<sup>8</sup>。Postgres提供ACID事务和丰富约束保证数据正确性<sup>12</sup>。通过JSONB列还可存储半结构化策略参数等信息<sup>12</sup>。在MCP服务器中，可采用ORM（如 SQLAlchemy）或原生异步驱动与Postgres交互，实现模块对元数据的读写。例如策略模块保存策略元信息、优化模块记录优化实验结果，都通过统一的数据库接口写入Postgres，保证事务完整性和审计追踪。

- **TimescaleDB / ClickHouse - 时序分析库**：采用专业的时序数据库来存储和查询**大规模时间序列数据**<sup>13</sup>。可根据规模和查询模式选择TimescaleDB或ClickHouse<sup>14</sup>：
  - *TimescaleDB* 是Postgres的扩展，适合与Postgres无缝集成，支持标准SQL查询和管理便利<sup>15</sup>。  
Timescale的超表（hypertable）机制和连续聚合使其能够高效存储查询多年历史行情和因子数据，并自动生成汇总数据（如日线从分钟线聚合）<sup>16</sup>。它适合中等规模数据并需要丰富SQL分析的场景。
  - *ClickHouse* 则是高性能列式OLAP数据库，适合处理高频tick级数据和复杂聚合分析，提供极致的查询性能<sup>14</sup>。  
*ClickHouse*在数据量极大、需要分布式扩展时有优势，同时其列存设计对压缩和向量化计算有卓越表现<sup>17</sup>。**存储内容**：行情市场数据（如日线、分钟线、Tick数据）、因子值矩阵（每支股票每天的多因子值）、回测输出（净值曲线、交易日志）以及实盘监控指标（收益率曲线、回撤序列）等均存于时序库<sup>1</sup>。这些数据以时间和标的为主键存储，表设计上可按日期或资产分区，提高查询性能。通过将大规模历史数据与分析引擎解耦，查询数年数据的性能大幅提升——这已成为量化平台的标配做法<sup>16</sup>。在MCP调用中，数据模块将清洗后的行情批量写入时序库（利用批量插入和压缩特性），因子模块计算结果也写入 `factor_data` 表供策略和回测使用<sup>1</sup>。查询时利用数据库的向量化执行和索引实现毫秒级聚合统计<sup>18</sup>。
- **MinIO 对象存储**：采用MinIO（S3兼容存储）保存**文档型和大文件**内容<sup>19</sup>。包括回测报告（HTML/PDF格式），策略分析图表截图，研究笔记等文件<sup>20</sup>。对象存储适合保存体积较大或非结构化的数据，并支持版本管理和权限控制<sup>21</sup>。方案是将此类文件的**元数据**（如文件路径/哈希、上传时间等）存入Postgres，而文件本身存放在MinIO以减轻数据库压力<sup>21</sup>。平台可提供统一的文件服务模块或API，供前端下载报告或同步研究文档。当用户在桌面端生成一份回测报告时，通过MCP接口将文件上传MinIO，并在数据库登记记录；监控面板前端可根据数据库记录生成链接获取报告，实现**文件同步与分享**。
- **Redis 缓存与队列**：使用Redis作为**高速缓存和消息队列**<sup>22</sup>。针对读频繁的数据，如最新市场快照、近期计算的因子值等，可缓存于Redis以减小数据库压力。例如行情快照以 `market:snapshot:{symbol}` 键存储，TTL设为60秒，实现一分钟内多次查询直接走缓存<sup>2</sup>。Redis也用于工作流的任务调度：各模块可将需要异步执行的任务（如一次回测）压入任务队列（如 `queue:backtest`），由后台工作进程弹出执行<sup>23</sup>。同时利用发布/订阅(pub-sub)机制，通知前端实时状态更新，实现秒级推送。Redis还可暂存风控检查状态等短期状态数据<sup>23</sup>。访问Redis可采用轻量客户端（如redis-py），并在MCP服务中实现统一的cache装饰器或读取接口，让各模块按需使用缓存。例如数据模块查询某天行情时，先查Redis缓存命中则返回，未命中再查数据库并回填缓存。通过**缓存+队列双重角色**，Redis提升了系统实时性能和解耦伸缩能力。
- **Chroma 知识库向量数据库**：引入Chroma向量数据库用于**知识库的语义检索**<sup>24</sup>。该模块主要服务于AI助理和研发知识管理，存储包括开发文档、代码段的Embedding向量<sup>25</sup>。比如，将平台开发手册、工程代码按章节/函数切分成约数万个chunk，并存入Chroma<sup>26</sup><sup>27</sup>。当用户或AI需要查询技术问题时，通过向量检索获取相关内容，实现**检索增强型问答（RAG）**<sup>28</sup>。Chroma支持向量+关键字的混合检索以及元数据过滤<sup>29</sup>，可保证在算法生成答案时引用最新准确的项目知识。这一组件虽不直接参与交易流程，但作为研发辅助是前沿实践，例如QuantConnect引入Claude等LLM辅助策略生成<sup>30</sup>。在整合上，可在MCP中部署一个“知识库服务”，供前端AI助手调用以检索资料。当LLM需要知道某函数含义时，经由MCP调用Chroma检索相关源码片段，再由LLM生成解释，从而提升开发效率。

**数据库访问层设计**：为了屏蔽底层多种数据库，实现模块与数据存储的解耦，建议构建统一的数据访问层（DAL）。例如提供 `DataRepository` 类，内部根据数据类型路由到对应的存储引擎：交易记录查询走Postgres，行情序列查询走TimescaleDB等。这样各模块调用DAL的方法即可，不必关心用的是哪种数据库驱动。同时采用**连接池**提高吞吐量，对频繁查询（如时序库）的连接保持长开或使用异步IO避免阻塞。对于分析型查询，

可在数据库端利用如Timescale的连续聚合和压缩功能预先优化：例如设置日线级别的continuous aggregate，以便在前端展示月度图表时直接查询汇总表而非逐条汇总分钟数据<sup>16</sup>。再如ClickHouse可设置物化视图提前计算常用因子横截面排名，读取时延迟更低。

**缓存使用策略：**合理利用多级缓存来平衡性能与一致性。原则是**长周期重复使用的数据存盘，中短期频繁访问的数据走缓存**。例如，每日收盘后计算的因子值矩阵，可保存到时序库作为长期存档；而当日晚些时候用户反复调试策略使用该因子时，可将因子DataFrame缓存到Redis或内存，以避免重复从数据库取出反序列化。针对实时性要求高的数据（如行情快照），缓存TTL应极短并有失效机制。任务执行过程中生成的大型中间结果（如某策略的所有交易信号矩阵）可以暂存于本地文件或分布式缓存，以支持后续步骤快速读取，然后在流程结束时清理。采用诸如**LRU缓存**策略自动淘汰过期条目，保持内存使用受控。总之，通过**数据库+缓存**的组合，既确保权威数据落盘，又充分利用内存加速，满足工业级平台对性能和可靠性的要求<sup>31</sup>。

下表总结了各数据存储层的角色、主要数据内容及特性：

存储层	角色定位	示例数据	技术特点/优化点
<b>PostgreSQL</b>	事务型主数据库 (OLTP)	策略元数据、审批流程、交易流水 <sup>32</sup>	ACID事务一致性、外键约束保证数据完整 <sup>12</sup> ；JSONB支持灵活存储 <sup>12</sup>
<b>TimescaleDB</b>	时序数据库 (Postgres扩展)	日线/分钟线行情，因子矩阵，回测曲线 <sup>1</sup>	与Postgres集成便利，支持标准SQL；超表分区+连续聚合提升长历史查询性能 <sup>16</sup> ；适合中等规模数据
<b>ClickHouse</b>	分析型列式数据库 (OLAP)	高频tick数据，大规模历史行情，因子序列	分布式可扩展，列存压缩高效；对复杂聚合和超大数据集提供毫秒级查询 <sup>14</sup> ；需自行维护与Postgres同步
<b>MinIO (S3)</b>	对象存储（文档/报告）	回测报告HTML、策略研究笔记、图表图片 <sup>20</sup>	大文件集中管理，支持版本控制；仅在Postgres存元数据（路径/哈希） <sup>21</sup> ；易于横向扩展存储容量
<b>Redis</b>	缓存 / 消息队列	行情快照缓存、任务队列、风控状态 <sup>2</sup>	基于内存操作，亚毫秒响应；TTL机制保证数据新鲜度；Pub/Sub用于实时推送通知
<b>Chroma 向量库</b>	知识库语义检索	开发手册chunks、代码片段embedding <sup>25</sup>	向量近似搜索结合关键词BM25 <sup>29</sup> ；辅助LLM查询文档以增强回答 <sup>28</sup>

通过上述多种数据存储协调，系统各模块各取所需：交易过程数据可靠存储在Postgres，批量分析读取Timescale/ClickHouse提速，大文件移至对象存储减负，缓存加速热点查询，知识库支持AI助手。这种架构在业内被证明可同时满足**事务要求和分析性能**，例如Sebastien Laignel提到使用PostgreSQL+TimescaleDB作为量化研究的核心数据架构，实现灵活性与性能兼顾<sup>15</sup>。

## 模块数据流与依赖整合

在优化后的架构中，各模块通过清晰的数据流连接，形成**模块间的调用关系图**，如下所示：

```

flowchart LR
    subgraph 数据流程
        A[原始数据源<br/>（行情API/文件）]
        B[数据模块<br/>（采集清洗）]
        C[因子模块<br/>（计算指标）]
        D[策略模块<br/>（生成信号）]
        E[回测模块<br/>（撮合交易）]
        F[优化模块<br/>（参数调优）]
    end
    subgraph 数据库/缓存
        G[(Postgres<br/>事务库)]
        H[(Timescale/ClickHouse<br/>时序库)]
        I[(MinIO 对象存储)]
        J[(Redis 缓存/队列)]
    end
    A --- 市场数据 --> B
    B --- 写入行情表 --> H
    B --- 元数据记录 --> G
    B --- 预处理数据 --> C
    C --- 因子结果写DB --> H
    C --- 因子矩阵输出 --> D
    D --- 策略信号 --> E
    E --- 读行情&因子 --> H
    E --- 交易仿真 --> E
    E --- 结果写DB --> H
    E --- 报告文件 --> I
    F --- 多次调用回测 --> E
    F --- 最优参数输出 --> D
    subgraph 前端交互
        K[研究终端(PyQt)]
        L[VSCode扩展]
        M[监控面板(Streamlit)]
    end
    K .. 控制 .-> F
    K .. 查看报告 .-> I
    L .. 调用 .-> B
    L .. 调用 .-> C
    L .. 调用 .-> E
    M .. 订阅进度 .-> J
    M .. 查询结果 .-> H

```

图：*TRQuant*工作流中的数据依赖和模块调用关系示意图。各模块通过数据库实现数据共享，通过工作流引擎/MCP接口串联执行。

**流程说明**：数据模块从外部数据源 (A) 获取原始行情，清洗后存入时序数据库 (H) 并将相关元数据（如数据源配置、更新时间）记录在Postgres (G)。接着因子模块从时序库读取所需行情，对每个资产计算因子值，将结果（因子矩阵）写入因子数据表 (H)，并输出因子结果给策略模块。策略模块读取行情和因子（可直接通过内存传递或由数据库获取），按照策略规则生成交易信号（如持仓权重或订单列表），传递给回测模块执行。回测模块再从时序库获取行情（和必要的因子数据）逐时仿真策略交易，将成交、持仓变化记录下来，最终生成净值曲线、绩效指标等结果数据。回测结果一方面详细记录入分析库 (H) 中对应表（如 `equity_curve`, `trade_log` <sup>1</sup>），另一方面生成易读的报告文件保存到MinIO (I) 供日后查看。优化模块则调用回测模块多次：它遍历一系列策略参数组合，每次调用回测得到结果，通过比较指标选择最佳参数输出给策略模块（或直接产出优化报告）。在优化过程中，若并行执行，则借助Redis队列 (J) 分发任务至多个回测worker，加速完成批量实验。整个流程由工作流引擎根据预设的9步顺序调度：确保依赖数据准备就绪后再运行下游任务，并可以根据上一步结果动态调整后续步骤（如若所有候选策略绩效不佳，则可回到因子模块重新挑选因子）。

**工作流引擎设计**：建议采用**DAG调度**模型定义流程，各步骤作为节点，数据依赖作为有向边，工作流引擎解析DAG按拓扑序执行。每个节点对应一次模块调用（可能包含子任务）。引擎可监控节点状态（未开始/运行/成功/失败），将状态更新通过MCP接口或消息推送给前端监控面板。为了增强灵活性，工作流引擎应支持**参数化和条件分支**：如用户可以指定跳过某些步骤或根据指标决定是否触发优化模块。这类似于Airflow的BranchOperator功能，但在此平台可简化为Python代码条件。由于TRQuant集成AI助手，工作流引擎还应允许AI根据上下文决定流程走向（例如AI建议增加一个风控检验步骤，则插入新的节点）。

**模块解耦与通信**：模块间不直接彼此调用函数，而是通过**MCP服务器统一接口**通信 <sup>33 34</sup>。MCP (Model Context Protocol) 服务器作为中介，提供各模块功能的API。例如，前端或工作流引擎调用“MCP-数据服务”的 `fetch_data` 方法来触发数据模块操作。模块输出数据主要通过数据库/缓存共享，而**状态和通知**通过消息机制传播：例如回测模块在完成后将结果概要发布到Redis频道，工作流引擎和前端订阅该频道以获知完成信号并获取结果。这样的架构确保模块物理上可以分布式部署，通过松耦合接口协调。这符合现代量化平台从单体应用向微服务迁移的趋势，易于扩展和维护 <sup>9 35</sup>。

值得注意的是，在业界先进方案中，如微软开源的Qlib，也采用数据处理与回测解耦的流水线，使研究人员将精力更多集中在因子和模型上 <sup>36</sup>。我们的优化方案与之一致：通过自动化数据->因子->策略->回测->分析的流水线，大幅减少人工干预和重复操作，提升研究迭代速度。同时整个数据流有完善的**审计追踪**（例如数据库审计日志记录了每次策略生成和回测的操作 <sup>8</sup>），满足金融场景对可追溯性的要求。

## 前端可视化层组件设计与框架选型

TRQuant平台包含**桌面端研究系统**（PyQt6）、**VS Code工作流扩展**（“Cursor”扩展）和**监控面板**（拟由Flask改为Streamlit）三大前端。为达到工业级的用户体验，我们在组件库选择、通信机制、状态管理和高性能渲染等方面对各前端进行优化。

### 桌面研究系统（PyQt6）

**技术选型**：采用**PyQt6**搭配 **PyQtGraph** 作为主要框架 <sup>37 38</sup>。PyQt6提供成熟的桌面GUI功能，而PyQtGraph是一款基于PyQt的高性能绘图库，特别适用于实时金融数据绘制。与Matplotlib相比，PyQtGraph在实时刷新情景下性能优势显著，可轻松达到屏幕刷新率的更新频率 <sup>39</sup>。因此我们弃用之前基于QPainter手工绘图的方案，改用PyQtGraph的现有组件来实现专业图表，包括K线蜡烛图、收益曲线、回撤曲线、因子热力图等 <sup>40</sup>。PyQtGraph使用Qt原生的QGraphicsScene架构，支持GPU加速绘制，能流畅处理成千上万数据点的动态更新 <sup>39</sup>。下表对比了几种方案，最终选择了PyQtGraph：

方案	优点	缺点	决定
PyQt6 + PyQtGraph	原生集成、高性能实时绘图 <sup>39</sup> 、与 NumPy高度兼容	学习使用新库需成本	推荐
PyQt6 + Matplotlib	Python社区成熟库，静态图效果好	实时刷新性能差，不适合长序 列 <sup>41</sup>	
PyQt6 + Plotly	交互性强，浏览器支持好	需嵌入WebView，增加复杂性 <sup>42</sup>	

**组件设计：**基于PyQtGraph规划和扩充图表组件库<sup>43</sup>。新增或重点组件包括： - K线图组件：支持专业K线（蜡烛图）绘制，叠加技术指标（均线、成交量柱状图等）<sup>40</sup>。可封装为CandlestickItem等类，支持缩放和平移交互。 - 收益曲线组件：绘制策略累计净值曲线，并实时更新最新收益及回撤<sup>44</sup>。使用PlotWidget绘制折线，结合 QTimer 定时刷新<sup>45</sup> 实现近实时更新（目标延迟<100ms<sup>46</sup>）。 - 回撤图组件：与收益曲线配套，显示最大回撤随时间变化，方便风险分析。 - 因子热力图组件：用于展示多因子在投资组合中的暴露和多空分布<sup>44</sup>。可用ImageItem显示矩阵形式的因子值，以颜色强度代表大小，支持交互式选择时间窗口放大细节。 - 持仓分布图组件：将当前持仓按行业或资产类别聚合为饼图，直观展示组合构成<sup>47</sup>。

这些组件文件分别位于gui/components/charts/ 目录下，采用面向对象方式封装，方便在主界面组合使用<sup>48</sup>。比如专业图表面板 ProChartPanel 聚合K线、收益曲线、回撤图等组件，提供统一的视图供用户查看策略表现<sup>49</sup>。此外，为提高代码组织和可维护性，桌面端采用**MVC（模型-视图-控制）**架构重构：在gui/framework/ 下定义 BaseModel、BaseView、BaseController 基类<sup>50</sup>。各界面窗口作为View，业务逻辑放入Controller，数据处理封装在Model，从而实现UI与逻辑分离，便于测试和迭代。这也是工业软件常见模式，有助于后续功能扩充。

**通信机制：**桌面应用直接运行在用户本地，通过MCP的Python客户端与后端交互<sup>51</sup>。也就是说，PyQt应用内置MCP调用能力，例如用户点击“运行回测”按钮时，界面层调用MCP客户端的相应方法（如mcp.run\_backtest(params)），由MCP服务器执行后台运算。由于桌面端与后端通常在同一机器，可采用本地IPC或HTTP调用MCP服务器API。整个调用是同步还是异步取决于任务：对于耗时操作（如回测），采用异步方式——即在后台线程发起MCP调用，同时UI主线程立即返回不阻塞，并利用信号槽机制在任务完成时收到通知更新界面。 PyQt6 提供 QThread 用于执行后台任务、pyqtSignal 用于线程间消息<sup>45</sup>。我们的实现为每个主要任务创建Worker线程，通过信号将结果传回主线程，在UI上呈现。这确保了UI在长时间运算时依然响应，不会冻结。

**状态管理与持久化：**桌面应用的大部分状态保存在内存中，包括当前加载的数据集、策略参数、最近的回测结果等。对需要跨会话保存的设置（如用户偏好的主题、窗口布局），可使用Qt提供的QSettings 或将配置序列化至本地文件。当与其他端协同时，重要状态也可上载服务器。例如策略编辑完成后，可以通过MCP上传策略代码或配置，使VS Code扩展和监控面板也能访问。总体而言，桌面端重心在交互效率，本地状态即可满足需求，不强制要求长久同步。但当需要时，可集成一个“云同步”选项，将关键状态持久化到服务器Postgres中，以便多设备协同。

**增量渲染：**利用PyQtGraph支持，桌面端实现增量更新而非整图重绘。例如K线图在新数据到来时，仅append新的蜡烛条数据点，而不是重绘全部历史，以减少闪烁和计算量。PyQtGraph的setData() 方法被调用时，如果传入的是增长的数据序列，可实现局部更新。此外，通过合理使用 QTimer，将图表刷新频率限制在如50ms一帧，避免过度刷新。对于表格等UI组件，也尽量采用Qt的Model/View框架，在数据变化时发射数据更新信号，以触发视图仅重绘变更部分。上述措施确保即使实时数据频繁更新，UI依然流畅。

## VS Code 工作流扩展（“Cursor”扩展）

**技术选型**：VS Code扩展采用 **TypeScript + React + ECharts** 构建<sup>52 53</sup>。其中React用于在VS Code Webview中构建前端界面，ECharts用于绘制图表。TypeScript提供类型安全保障与VS Code API的良好支持。选择React主要因为VS Code官方扩展样例多用React，生态成熟，利于复杂交互界面实现<sup>54</sup>。ECharts作为开源可定制的图表库，可满足扩展中嵌入K线图、进度图等需求，也有VS Code扩展使用先例，结合React可以方便地管理其状态。比较而言，Vue或Svelte也能做，但React在团队已有90%完成度且类型系统完备，因此保持方案不变<sup>55</sup>。

**界面与组件**：VS Code扩展主要用于**AI辅助的量化工作流**，界面包含：9步工作流各步骤的面板、AI提示/生成区域、策略代码编辑嵌入，以及图表展示区域（策略绩效、因子分析图等）。采用React将界面拆分成多个组件，如 `WorkflowStepsView` 显示工作流步骤及状态、`StrategyEditor` 嵌入代码编辑、`ResultChart` 显示回测结果图表等。React的Context用于跨组件共享全局状态，如当前工作流执行进度<sup>56</sup>。ECharts图表通过React集成，定义为子组件，使用 `echarts-for-react` 或类似binding，使其能随React状态刷新。典型图表需求包括工作流甘特图（展示各步骤起止状态）、策略收益曲线和因子IC曲线等，均可用ECharts灵活绘制。

**通信机制**：VS Code扩展运行在VS Code沙箱环境，与后端MCP服务器通信需通过扩展后端（Extension Host）的能力。通常，VS Code扩展可以直接使用Node模块发送HTTP请求，或开启子进程与后端交互。在本方案中，扩展将利用MCP提供的**标准输入/输出 (stdin/stdout) 协议**来调用后台功能<sup>57</sup>。例如，扩展后端启动一个MCP客户端进程，作为桥梁接受扩展指令并发送给MCP Server，然后将结果通过stdout拿回。这种方式适用于LLM需要调用外部函数的场景，因为MCP本就是为AI助手设计的通讯协议。另一种可行方式是扩展直接通过HTTP请求调用MCP的REST接口，但考虑安全和兼容性，使用官方支持的VS Code扩展通信更佳。扩展前端React部分与后端通过VS Code提供的 `vscode.postMessage` 机制通信：React页面可以向扩展后端发送消息请求数据或操作，然后后端处理后回传结果。我们在后端实现一个消息路由，比如接收 `command: "run_step", stepId: 3` 的消息，就调用对应MCP接口执行第3步模块，并将状态通过 `vscode.window.setStatusBarMessage` 或 `postMessage` 通知前端。前后端消息序列化采用JSON，确保可扩展性。

**状态持久化**：为增强扩展的健壮性，重要的工作流状态会**持久化**保存，避免VS Code重启或扩展重载导致状态丢失。VS Code扩展API提供了 `ExtensionContext.globalState` 和 `workspaceState` 用于存储数据<sup>58</sup>。本方案使用globalState保存工作流进度和参数，因为这些状态需要跨工作区甚至不同机器同步<sup>59</sup>。实现上，扩展维护一个 `WorkflowStateManager` 类，提供 `saveState` 和 `loadState` 方法，将当前步骤编号、已完成的数据摘要等写入globalState<sup>60</sup>。VS Code会将globalState持久化在用户配置数据库中，扩展下次激活时可读取恢复<sup>58</sup>。例如，当用户关闭VS Code再打开，之前执行到第5步的工作流信息会自动恢复显示。这种persist机制在Stack Overflow上也被推荐用于扩展保存关键状态<sup>59</sup>。另外，对于较大数据（如完整回测结果集），不适合放在globalState，可选择将其暂存于扩展的全局存储目录（`ExtensionContext.storageUri`）中文件，然后在需要时再加载。结合这两种方式，保证扩展在异常退出后能自动恢复90%以上的场景<sup>61</sup>。

**增量渲染与性能优化**：由于VS Code Webview运行在Chromium环境，性能有限，我们采取多种优化：  
- **图表增量更新**：对于ECharts图表，不每次都 `setOption` 全量数据，而是使用其增量更新特性<sup>62</sup>。例如，实时进度图只更新最近一个点的数据。通过调用 `chart.setOption({series: [{data: newData}], {notMerge: false, lazyUpdate: true}})`，ECharts会智能合并新数据而非重绘全部<sup>62</sup>。这样在更新频率较高时（如每秒更新进度），性能提升显著。  
- **条件渲染**：利用React的`shouldComponentUpdate`或`memo`，避免不必要的组件重新渲染。例如当某步骤日志更新时，仅日志组件刷新，其他UI部分不变。  
- **减少DOM元素**：界面设计简洁，能用图表展示的信息就不重复用表格，尽量减少DOM节点数量，减轻Webview负担。  
- **异步处理**：大量计算尽量放在扩展后端进行，React前端只负责渲染结果。比如因子分析结果排序由后台MCP算好再传给前端，从而前端JS不需执行复杂计算。  
- **错误恢复**：针对Webview可能发生的崩溃或白屏，扩展后端实现监控，一旦检测到异常

(例如前端未响应)，则自动重启Webview并从globalState恢复状态<sup>63</sup>。通过try-catch捕获前端渲染错误，在catch中发送通知要求刷新界面，实现一定程度的自愈<sup>63</sup>。

通过上述优化，目标是将扩展界面的图表渲染性能提升30%以上，达到复杂图表操作依然流畅的体验<sup>61</sup>。ECharts本身在5.0+版本中针对大数据绘制和渐进渲染做了优化，我们将充分利用。例如ECharts的progressive属性可以用于因子热力图按需渲染。总之，VS Code扩展需在功能和性能间找到平衡：一方面提供丰富交互（AI聊天、图表展示等），另一方面保证不拖慢整体IDE。

## 实时监控面板（Streamlit Dashboard）

**技术选型**：原文件管理/监控系统由Flask+静态页实现，交互性和实时性不足<sup>64</sup>。我们计划升级为**Streamlit**框架开发<sup>65</sup>。选择Streamlit的原因：1) **开发效率高**，使用纯Python即可快速搭建复杂前端，无需深厚前端技术<sup>66</sup>；2) **内置实时性支持**，基于WebSocket自动刷新UI<sup>67</sup>；3) 丰富的组件库，社区有许多现成的金融图表组件（如streamlit-echarts、streamlit-aggrid）可用<sup>68</sup>；4) 易于部署，可在服务器或云端一键启动Web应用。对比来看，Plotly Dash也可构建交互仪表盘，但需要手动管理回调，不如Streamlit即写即所得；Flask+前端框架灵活但开发量大；Gradio虽简单但组件不够丰富<sup>69</sup>。因此最终选定Streamlit作为监控面板技术栈<sup>69</sup>。

**主要功能**：监控面板将作为独立Web应用，供用户在浏览器中查看系统状态和结果。按照规划，将实现以下页面<sup>70</sup>：  
- 系统状态页：展示各MCP服务运行状态（如注册的6个服务是否在线）以及硬件资源监控（CPU/内存占用）等。<sup>71</sup> - 侧边栏提供整体状态概览指标，如“MCP服务器：6/6 运行中”<sup>71</sup>。  
- 工作流进度页：可视化当前9步工作流的执行进展，每步显示开始/完成时间、耗时，类似甘特图或步骤条。<sup>72</sup> 利用st.progress组件动态展示整体进度，比如“5/9 步骤完成”<sup>71</sup>。  
- 回测结果页：提供选择某次回测结果进行详细展示，包括净值曲线、风险指标表格、交易分布图等<sup>73</sup>。可集成Plotly绘制可交互收益曲线，支持悬停查看具体数值；ECharts用于K线图；AgGrid用于交易明细表格，高亮盈亏等。  
- 策略库页：列出所有策略清单及文件，允许在线浏览策略代码（读取Git仓库或文件系统内容）<sup>73</sup>。也可提供简单的版本管理操作入口。  
- 文档中心页：嵌入研究文档（如Jupyter笔记、Markdown说明书），便于用户查阅模型说明或使用Manual知识库内容<sup>74</sup>。

**通信与状态**：Streamlit应用以Python脚本形式运行，可以直接导入MCP Python客户端与后台交互<sup>75</sup>。例如，在系统状态页的代码中，调用mcp\_client.get\_server\_status()获取各服务状态数据，并用st.metric组件实时显示<sup>71</sup>。由于Streamlit本质上会在用户每次交互时重跑脚本，我们利用Session State保持跨交互状态<sup>51</sup>。例如用户选中了某策略查看结果，这个选择可存入st.session\_state['selected\_strategy']，切换页面时仍可访问<sup>51</sup>。同时，为实现实时推送，计划使用Streamlit的WebSocket支持（底层基于asyncio）。  
**具体做法**：在后台运行一个asyncio任务订阅Redis的某些频道（如回测进度发布），一旦收到新消息就利用st.experimental\_rerun()或st.websocket更新界面组件<sup>76</sup>。例如，当回测模块发布进度50%时，后台任务捕获后更新session\_state中的进度变量，触发页面重新绘制进度条到50%。Streamlit官方并未完全开放后台推送，但可通过组合st.empty()占位符和循环检查的办法接近实时刷新。借助这种机制，我们期望监控面板对核心事件的更新延迟<1秒<sup>77</sup>。

**可视化组件**：借助Streamlit的插件，我们选择以下组件库：  
- **Plotly**: 用于绘制交互性强的图表，如权益曲线、月度收益柱状图等。Plotly图表在Streamlit中渲染后可支持拖拽缩放、曲线隐藏等交互，提升用户体验<sup>78</sup>。  
- **Altair**: 用于声明式地快速绘制统计图，如收益分布直方图、回归分析图等<sup>78</sup>。Altair基于Vega-lite，适合静态可视化且与Pandas结合方便。  
- **streamlit-echarts**: 直接在Streamlit中使用ECharts<sup>79</sup>。这方便我们在监控面板复用VS Code扩展里定义的某些图表配置，使不同前端展示一致。例如将ECharts配置通过MCP下发给Streamlit端渲染相同的因子暴露热力图。  
- **streamlit-aggrid**: 强化版表格组件，支持冻结列、排序过滤，非常适合展示回测交易明细、

大量K线数据表格<sup>79</sup>。 - **streamlit-elements** (备选): 允许直接在Streamlit里嵌入完整的前端小应用 (比如 Monaco代码编辑器)。未来可用于策略库页显示代码高亮编辑器。

通过组合以上组件, 监控面板能实现媲美专业Web端监控系统的效果, 但开发却大为简化。例如一个实时净值曲线图, 只需几十行Python调用Plotly Express即可完成, 而传统前端可能要写大量JS代码。再比如将回测指标以多列数值和图标展示, Streamlit的列布局和 `st.metric` 组件使其轻松实现<sup>71</sup>。同时, 借助MCP客户端, 监控面板可以方便地调用平台功能: 比如一键启动某策略回测 (调用MCP的回测方法并立即显示进度)。

**增量更新**: Streamlit的运行模型是每次交互重绘整个页面, 这对实时刷新来说不够高效。我们通过拆分页面和优化逻辑减轻重绘开销: 利用 `st.tabs` 或 `pages/` 多页, 把不同板块隔离, 减少单页元素数量<sup>70</sup>; 关键图表使用 `st.empty()` 占位符, 在循环中反复更新其中内容而不是刷新整个页面。对于长表格, 引入分页加载, 只显示当前页数据, 避免一次渲染上万行。借助streamlit-aggrid的按需渲染特性, 大表格滚动时动态加载可行, 使性能保持流畅。将这些增量渲染技巧与Streamlit自动刷新机制结合, 最终用户将感受到类似实时仪表盘的顺畅体验: 数据在后台变化, 前端相应组件即时更新, 而其他界面元素不闪烁、不重复加载。

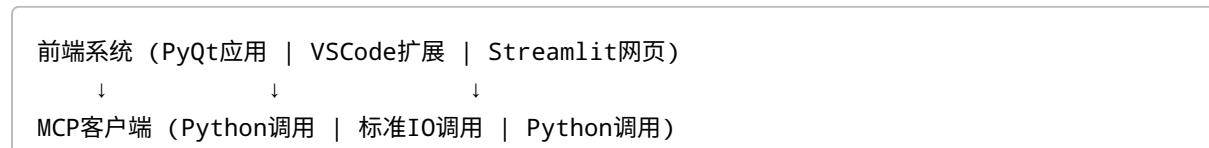
**安全与部署**: 监控面板作为运维组件, 默认部署在内部网络。结合Streamlit的身份认证插件, 可保护敏感数据。部署上, 可直接在服务器以Docker容器运行Streamlit应用, 或托管到云服务。由于是纯Python实现, 可以和后端服务部署在一起, 充分利用本地MCP客户端减少延迟。Streamlit升级简便, 也利于根据用户反馈迅速调整监控展示。

## 前端系统技术概览与比较

综合上述方案, 我们将TRQuant三大前端的最终技术栈和特性汇总如下<sup>51</sup>:

系统	技术框架	图表库	后端通信	状态管理
桌面研究 (PyQt)	Python + PyQt6	PyQtGraph (高性能原生)	MCP本地接口 (Python客户端) <small>80</small>	进程内Memory, QThread + Signal机制 <small>51</small>
VS Code扩展	TypeScript + React	ECharts (Web)	MCP子进程通讯 (标准IO管道) <small>80</small>	VS Code globalState全局存储 <small>81</small>
监控面板 (Web)	Python + Streamlit	Plotly / ECharts等	MCP远程接口 (Python客户端) <small>80</small>	Session State + Redis订阅状态

从上表可见, 我们在前端通信上实现了**统一的MCP接口**: 无论是本地GUI、VS Code还是Web端, 都通过各自语言的MCP客户端与后台进行交互。这使得业务逻辑集中在服务器, 实现**前后端解耦**和一致的功能调用方式。状态管理则根据终端特点各异: 本地GUI依赖进程内状态 (利用Qt信号机制同步UI)、VS Code扩展借助IDE提供的globalState保存关键数据, 而Web监控依托服务器和浏览器会话状态结合。图表库的选择考虑了运行环境: 桌面端采用原生库发挥性能, Web相关端采用JavaScript图表库保证丰富交互和一致风格。在通信架构上, 多前端通过中间的MCP服务器汇聚, 与后端核心服务连接, 如下所示<sup>82</sup><sup>83</sup>:



→ 统一路由至 → MCP服务器（6个微服务）<sup>84</sup>  
包含：核心quant服务(trquant-core)、工作流服务(trquant-workflow)、项目管理服务  
(trquant-project)、交易执行服务(trquant-trading)、开发辅助服务(trquant-dev)、文件服务  
(filesystem) 等<sup>85</sup>

这意味着，不同前端的用户操作最终都会转化为对后台这些服务的请求。例如用户在PyQt点击“运行优化”，经过MCP客户端路由到 trquant-workflow 服务执行优化任务；而VS Code扩展的AI助手要求获取某策略回测结果，同样通过MCP请求 trquant-core 服务查询 equity\_curve 数据。**多前端共享同一套后端服务**既减少重复开发，又确保各端数据一致。工业界量化平台（如QuantConnect本地平台）也采用VS Code扩展+本地服务的模式，让研究代码、数据和结果无缝同步<sup>86 87</sup>。我们的方案把桌面应用与此融合，使得用户无论使用哪种界面，都在同一个后端环境上协作，极大提高了易用性。

## 工程结构与代码组织优化

要实现上述优化方案，需要有良好的工程结构来支撑，做到**高内聚、低耦合**并方便团队协作。结合当前代码基础和业界实践，我们建议按以下思路组织项目：

1. **按领域分层的模块化工程**：将代码按功能领域拆分为数个核心子项目或包。例如：
  2. trquant-core：核心量化逻辑库，包含数据处理、因子计算、策略框架、回测引擎、优化算法等纯逻辑代码。可进一步细分子模块，如 core/data, core/factor, core/strategy, core/backtest, core/opt，每个子模块对应上文定义的功能模块实现。这样所有业务逻辑集中在一个可独立测试的库中，前后端都可调用。<sup>88 89</sup>
  3. trquant-workflow：工作流调度代码，定义9步流程的DAG和任务执行实现。可依赖core模块调用各步具体逻辑，并与数据库/缓存交互以跟踪状态。
  4. trquant-project：策略项目/文件管理模块，例如创建新策略、保存版本，接口封装Git操作或文件读写。
  5. trquant-trading：实盘交易和风控模块，包含模拟下单接口、实时风控检查等（若平台未来用于实盘，可以预留）。
  6. trquant-dev：开发辅助模块，如AI助手接口（连接Chroma向量库）等，提供给IDE扩展或其他开发工具使用。
  7. trquant-gui：PyQt桌面应用前端代码。
  8. trquant-vscode-ext：VS Code扩展前端代码(TypeScript)。
9. trquant-dashboard：Streamlit监控面板代码。每个模块都有清晰的职责边界。这样的分层方式类似于Colman M. Quinn提出的QuantSystem架构，其中核心C++引擎、API服务、客户端各自独立<sup>88</sup>。我们的Python实现可以类比，确保核心算法与界面、接口分离。
10. **微服务与接口设计**：上述 trquant-xxx 模块中，需要对外提供服务的部分，按照MCP接口规范实现。例如用FastAPI或自定义RPC包装core模块的关键函数。在实现上，可创建 mcp\_servers/ 目录，其中6个子服务器分别引用对应模块逻辑。例如 mcp\_servers/trquant\_core\_server.py 加载trquant-core库，注册函数如 fetch\_data, compute\_factor, run\_backtest 等为MCP可调用接口； trquant\_workflow\_server.py 则加载workflow模块，注册例如 start\_workflow, get\_progress 等接口。每个MCP服务器进程启动时，都会监听来自前端的调用指令。通过这种**轻量服务**包装，我们保持了逻辑与接口的松耦合，方便将来替换通信协议或部署扩展。在实际工程中，可能会将多个相关接口合并到一个进程以减少服务数量（如core和trading接口合并），正如前文通信架构所示6个服务

<sup>83</sup>。开发时，可先使用FastAPI/UVicorn快速跑通HTTP接口，后期根据AI MCP协议调整为特定格式。保持接口的一致性和文档化非常重要，应使用开放API文档（Swagger）或README清楚列出每个接口的方法名、参数和返回，供前端和AI调用。

11. **公共库与配置**：提取公共的工具函数（如日志、数据库连接管理、缓存封装）放入一个 `trquant-common` 或 `trquant-utils` 模块。尤其是数据库连接部分，可统一在此模块进行初始化（例如读取配置文件，建立Postgres和ClickHouse连接池），各业务模块通过调用common模块的方法来获取连接<sup>90</sup>。同时，一份全局配置（如YAML/JSON）管理不同环境（开发、生产）的数据库URI、缓存地址等，由common模块解析提供给各处使用。这防止各模块重复配置，降低出错概率。
12. **代码规范与测试**：采用统一的编码规范（例如PEP8风格，Type Hinting，全局异常处理），并为各模块编写充分的单元测试和集成测试<sup>91</sup>。比如：
13. **数据模块测试**：给定模拟行情CSV，验证导入数据库正确且查询结果符合预期。
14. **因子模块测试**：构造已知数据计算因子，验证输出是否正确并无未来数据泄露。
15. **回测模块测试**：针对简单策略（如均线交叉）在小数据集回测，验证收益曲线和交易记录符合手工计算结果<sup>92</sup>。
16. **接口测试**：使用测试客户端调用MCP接口，确保每个函数返回结构正确，错误处理如期望。可以模拟前端调用序列测试整个工作流。为此，可引入pytest框架组织测试<sup>91</sup>。持续集成CI可配置自动跑测试、lint检查等，提高工程质量<sup>93</sup>。
17. **文档与知识库维护**：充分利用Chroma知识库，将工程文档、API说明、模块设计决策等都纳入向量库，方便新开发者或AI助手检索<sup>26</sup><sup>94</sup>。在代码仓库中维护 `docs/` 目录，包括架构总览、模块API文档、开发指南等Markdown文档，并同步更新知识库内容<sup>95</sup>。通过RAG技术，让AI辅助编码时能够引用最新文档<sup>28</sup>，减少沟通成本。定期更新架构图、模块依赖图（如上文Mermaid图）以反映最新实现，确保文档与实现同步演进。
18. **部署与运维**：工程结构上支持不同部署拓扑。开发阶段可本地单机运行所有模块进程；生产可将各服务分别容器化部署，使用Docker Compose或Kubernetes编排。日志集中管理，每个服务的日志流入集中日志系统便于排查。利用Redis队列和服务心跳，构建简易**监控告警**：MCP服务注册自身心跳到Redis，监控面板定期检查，如发现某服务心跳超时则在UI报警提示维护人员。通过Streamlit监控页面<sup>96</sup>，运维人员能直观看到各模块状态，符合工业级稳定性要求。

综上所述，经过模块职责划分、数据库架构优化、数据流解耦，以及前端系统的全面升级，TRQuant平台将具备**工业级的健壮与易用性**。此方案利用了当今业界在量化平台方面的诸多最佳实践：例如**多存储分层设计**确保既有交易级数据可靠，又有分析级数据高效<sup>15</sup>；**工作流自动化与AI辅助**降低人工成本<sup>36</sup>；**前后端解耦多界面协同**提升用户体验和开发效率<sup>86</sup>。按照规划分阶段实施（见前端优化计划的四周时间表<sup>97</sup>），逐步完成各项改进，并以验收标准严格测试功能和性能<sup>98</sup><sup>99</sup>。最终，优化后的TRQuant将拥有清晰合理的模块边界、稳定高效的数据底座、流畅智能的用户界面，能够更好地支持量化研究与实盘策略开发的全流程需求。

- 3 6 7 QuantRocket - Data-Driven Trading with Python  
<https://www.quantrocket.com/>
- 4 5 91 93 GitHub - 0xemmkty/QuantMuse: A comprehensive quantitative trading system with AI-powered analysis, real-time data processing, and advanced risk management  
<https://github.com/0xemmkty/QuantMuse>
- 9 35 Quant 2.0 Architecture: Rewiring the Trading Stack for the AI Era | AltStreet  
<https://altstreet.investments/blog/quant-2-architecture-modern-trading-stack-ai-mlops>
- 15 16 30 Building a Full Quant Research Stack | by Sebastien M. Laignel | Nov, 2025 | InsiderFinance Wire  
<https://wire.insiderfinance.io/building-a-full-quant-research-stack-7cea9518eec1?gi=681b655d4ee4>
- 17 18 ClickHouse vs. Postgres: 5 key differences and how to choose  
<https://www.instaclustr.com/education/clickhouse/clickhouse-vs-postgres-5-key-differences-and-how-to-choose/>
- 33 34 37 38 40 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 60 61 62 63 64 65 66 67 68  
69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 96 97 98 99
- FRONTEND\_OPTIMIZATION\_PLAN.md  
[file:///file\\_00000001a6c722fbca61814eec057cf](file:///file_00000001a6c722fbca61814eec057cf)
- 36 R&D-Agent-Quant: A Multi-Agent Framework for Data-Centric Factors and Model Joint Optimization  
<https://arxiv.org/html/2505.15155v2>
- 39 Which is one better pyqtgraph or matplotlib for realtime applications?  
[https://groups.google.com/g/pyqtgraph/c/A\\_R-LuNDXyQ/m/2XaiwgOoCAA](https://groups.google.com/g/pyqtgraph/c/A_R-LuNDXyQ/m/2XaiwgOoCAA)
- 41 Fast real-time plot (100Hz) - Using Streamlit  
<https://discuss.streamlit.io/t/fast-real-time-plot-100hz/8155>
- 58 59 visual studio code - How to persist information for a vscode extension? - Stack Overflow  
<https://stackoverflow.com/questions/51821924/how-to-persist-information-for-a-vscode-extension>
- 86 87 QuantConnect - Visual Studio Marketplace  
<https://marketplace.visualstudio.com/items?itemName=quantconnect.quantconnect>
- 88 89 92 QuantSystem: Modular Trading Architecture Summary  
<https://www.linkedin.com/pulse/quantsystem-modular-trading-architecture-summary-colman-marcus-quinn-ifeu>