



UNIVERSITÉ DU QUÉBEC EN OUTAOUAIS
DÉPARTEMENT DES SCIENCES ADMINISTRATIVES

WORKING PAPER NO. 13

VQuant: An Agentic Large Language Model Architecture for Automated Quantitative Financial Analysis and Document Generation

*Reasoning-Augmented Tool Orchestration, In-Loop Numerical Computation, and
Multi-Format Reporting*

Simon-Pierre Boucher

Département des sciences administratives
Université du Québec en Outaouais

simon-pierre.boucher@uqo.ca

Gatineau – Pavillon Alexandre-Taché
283, boulevard Alexandre-Taché
Gatineau, Québec, Canada J9A 1L8

June 2026

Version 1.0

Abstract

Large language models (LLMs) have shown remarkable fluency in financial discourse, yet conversational fluency alone does not guarantee numerically rigorous, reproducible, or citable analysis. This paper presents **VQuant**, a production-grade web platform that couples a reasoning-augmented LLM with a deterministic execution substrate to transform a natural-language request into a complete quantitative financial report. We document the system’s methodology and its principal innovations: (i) an *agentic orchestration loop* that exposes more than two hundred financial and analytical tools—spanning over 265 market-data endpoints, four web-research services, and an arbitrary-code Python sandbox of seventeen scientific libraries—to the model through a constrained function-calling protocol; (ii) a *context-management algorithm* that preserves conversational coherence over effectively unbounded histories via adaptive summarization under a token budget; (iii) a *single-call computation paradigm* that consolidates data acquisition, numerical estimation, and figure rendering into one sandboxed Python execution, collapsing what would otherwise be several model round trips into one; and (iv) a *unified document pipeline* that emits publication-ready PDF, DOCX, $\text{\LaTeX}$ Beamer slide decks, and spreadsheet workbooks with live formulas from a single Markdown artifact. We describe each mechanism as it is implemented—the streaming Server-Sent-Event protocol and its eight-way event dispatch, the recursive tool-calling loop and its tool-use/tool-result invariant, the batched-execution throttle, the two-stage context-management machinery, the token-accounting correction, and the inter-process sandbox that exec’s model-authored code and captures its figures—so that the design can be reproduced. The classical quantitative methods the platform automates are conventional and relegated to an appendix; our contribution is the *architecture* that makes them accessible through natural language while preserving reproducibility. We argue that VQuant illustrates a reusable design pattern in which the language model is positioned as a planner and synthesizer rather than as a calculator, with all numerical work delegated to auditable, deterministic code.

Keywords: Large language models, agentic systems, tool use, quantitative finance, automated reporting, retrieval-augmented generation, financial NLP

JEL Classification: C45, C63, C88, G17

1. Introduction

The rapid maturation of large language models (LLMs) has transformed expectations about what conversational software can accomplish in finance. Models such as the Claude, GPT, and Gemini families now produce fluent commentary on earnings, valuations, and macroeconomic conditions, and a growing literature documents their ability to extract signal from financial text (Lopez-Lira and Tang, 2023; Kim et al., 2024; Wu et al., 2023). Yet fluency is not the same as rigor. A model that *describes* a value-at-risk calculation in prose has not *computed* one; a model that *recalls* a stale price from pre-training has not *retrieved* the current quote; and a model that narrates a Monte Carlo experiment has not run the simulation whose numbers a portfolio manager would act upon. The central engineering problem in applied financial AI is therefore not generation but *grounding*: how to bind a probabilistic text generator to live data, deterministic computation, and reproducible artifacts without sacrificing the conversational interface that makes the technology accessible.

This paper documents the methodology and innovations of **VQuant**, a production-grade web platform built to address precisely this problem. VQuant accepts a natural-language request—“analyze NVDA’s fundamentals, technicals, and run a Monte Carlo simulation,” for example—and returns a complete, multi-section quantitative report containing live data, computed figures, and downloadable documents. The platform is implemented as a full-stack TypeScript monorepo: a React front end, a modular Express back end, and an isolated Python scientific-computing layer. At its core, the system orchestrates a reasoning-augmented LLM through an *agentic loop* that exposes more than two hundred financial and analytical tools to the model, executes the model’s tool calls deterministically, streams the results back in real time, and synthesizes a final report that can be exported in five document formats.

The motivation for studying this system is fourfold. *First*, VQuant operationalizes a design philosophy that we believe is the correct one for high-stakes numerical domains: the LLM is a *planner and synthesizer*, never a calculator. Every number in a VQuant report originates from a deterministic source—a market-data API, a closed-form pricing formula, or auditable Python code—so that the analysis is reproducible and the model’s well-known propensity for numerical hallucination is structurally contained. *Second*, the platform confronts the practical constraints of agentic systems—bounded context windows, tool-call latency, rate limits, and cost—with concrete algorithmic solutions whose logic generalizes beyond finance. *Third*, VQuant integrates an unusually broad tool ecosystem (over 265 market-data endpoints, four web-research services, and a seventeen-library Python sandbox) under a single function-calling protocol, providing a useful case study in tool taxonomy and prompt-level routing. *Fourth*, the system closes the loop from query to deliverable by generating publication-ready PDF, Word, L^AT_EX Beamer, and spreadsheet outputs from a single internal representation, illustrating a reusable pattern for automated report generation.

Our contributions are methodological and architectural rather than empirical. We make four specific contributions. **(1) Agentic orchestration with constrained tool use.** We describe a streaming tool-calling loop (Section 4) in which a strategically engineered system prompt routes requests to an appropriate subset of more than two hundred tools, batches tool execution to respect external rate limits, and surfaces the model’s intermediate reasoning to the user in real time. **(2) Context management under a token budget.** We formalize an adaptive-summarization algorithm that preserves coherence across effectively unbounded conversation histories by compressing older turns once a token threshold is crossed, while retaining the most recent exchanges verbatim. **(3) A single-call computation paradigm.** We argue that consolidating data acquisition, numerical estimation, and visualization into one sandboxed Python execution—rather than the sequential fetch-then-analyze pattern typical of naive agents—materially reduces end-to-end latency, and we describe the sandbox and its safety properties (Section 5). **(4) A unified multi-format document pipeline.** We detail how a single Markdown artifact is transformed into PDF (headless-browser rendering with mathematical typesetting), DOCX (document converter), Beamer slides (model-structured $\text{\LaTeX}$ compiled to PDF), and spreadsheets with live formulas.

Our emphasis throughout is on *mechanism*. We describe each component as it is actually built—the byte-level streaming protocol, the recursion that drives the loop, the throttle that batches tool calls, the two-stage machinery that keeps the context within budget, the correction that keeps token accounting honest, and the inter-process boundary that executes model-authored code—so that the design is reproducible rather than merely sketched. The classical quantitative methods the platform automates (option pricing, volatility modeling, simulation, and risk metrics) are standard and not the contribution of this paper; we record them compactly in an appendix and otherwise keep the focus on the architecture.

The remainder of the paper is organized as follows. Section 2 situates VQuant within the literatures on LLM agents, tool use, retrieval augmentation, and AI in finance. Section 3 presents the system architecture, data layer, and external integrations. Section 4 develops the agentic orchestration methodology, including the tool taxonomy, context-management algorithm, and single-call execution paradigm. Section 5 formalizes the embedded quantitative methods and the document-generation pipeline. Section 6 discusses performance, reliability, security, and limitations. Section 7 concludes and outlines directions for evaluation and extension.

2. Related Work

VQuant sits at the intersection of four research strands: (i) large language models and reasoning, (ii) tool-augmented and agentic systems, (iii) retrieval augmentation and grounding, and (iv) the application of LLMs to finance. We review each in turn and position our contribution accordingly.

2.1. Language models and reasoning

The transformer architecture of [Vaswani et al. \(2017\)](#) and the demonstration of in-context, few-shot learning at scale by [Brown et al. \(2020\)](#) established the modern paradigm of general-purpose language models. A complementary line of work showed that intermediate “chain-of-thought” reasoning improves performance on multi-step problems ([Wei et al., 2022](#)), motivating a generation of models that allocate explicit computation to deliberation before answering. VQuant builds directly on this development: it targets reasoning-augmented models that can “think” under an adaptive token budget prior to responding, and it surfaces this intermediate reasoning to the user. We treat the foundation model as a fixed, externally provided capability ([Bommasani et al., 2021](#)) and study the *system* that must be engineered around it.

2.2. Tool use and agentic systems

A model confined to its parametric memory cannot retrieve live prices or perform exact arithmetic. The tool-use literature addresses this by teaching models to call external functions. [Schick et al. \(2023\)](#) showed that models can learn to invoke APIs, while the ReAct framework of [Yao et al. \(2023\)](#) interleaved reasoning traces with actions in an observe–think–act loop that has become the template for modern agents. Program-aided approaches ([Gao et al., 2023](#)) delegate computation to a code interpreter, offloading exact arithmetic from the language model to a deterministic runtime—a principle that VQuant adopts as a first-class design constraint. Industrial function-calling protocols ([Anthropic, 2024](#)) standardized the schema-driven invocation of tools, and the Model Context Protocol generalized tool exposure across providers. VQuant operationalizes these ideas at scale: rather than a handful of tools, it exposes a curated catalogue of more than two hundred, and it contributes practical mechanisms—prompt-level routing, execution batching, and a single-call computation paradigm—for managing the resulting orchestration complexity.

2.3. Retrieval augmentation and grounding

Retrieval-augmented generation ([Lewis et al., 2020](#)) grounds model outputs in external documents, mitigating hallucination and staleness. VQuant generalizes grounding beyond text retrieval: its “retrieval” surface includes structured market-data endpoints, semantic and keyword web search, and on-demand numerical computation. Every quantitative claim in a VQuant report is traceable to a deterministic source, which we view as the financial-domain analogue of citation-grounded generation. This emphasis on auditability connects to broader concerns about reliability and provenance in machine learning for finance ([López de Prado, 2018](#)).

2.4. Large language models in finance

Domain-adapted models such as BloombergGPT (Wu et al., 2023) and open frameworks such as FinGPT (Yang et al., 2023) demonstrated the value of finance-specific training data. On the application side, Lopez-Lira and Tang (2023) found that LLM-derived sentiment from news headlines predicts subsequent returns, and Kim et al. (2024) showed that LLMs can analyze financial statements at a level competitive with human analysts. These studies largely treat the LLM as a *forecasting or classification* engine operating on text. VQuant takes a different stance: it uses the LLM as an *orchestrator* that assembles live data, delegates numerical work to deterministic code, and composes the results into a report. The classical quantitative methods it automates—option pricing (Black and Scholes, 1973; Merton, 1973), conditional-volatility modeling (Engle, 1982; Bollerslev, 1986), Monte Carlo valuation (Glasserman, 2004), and Value-at-Risk (Jorion, 2007)—are well established; the contribution is the *architecture* that makes them accessible through natural language while preserving reproducibility.

2.5. Positioning

Relative to prior work, VQuant is distinctive in three respects. First, it integrates an unusually broad and heterogeneous tool ecosystem under one protocol, and it contributes concrete methodology for routing and rate-managing that ecosystem. Second, it enforces a strict separation between reasoning (performed by the LLM) and computation (performed by deterministic code), which we argue is the appropriate design for numerically sensitive domains. Third, it treats the *deliverable*—a citable, exportable, reproducible document—as a primary artifact rather than an afterthought. The remainder of the paper develops each of these points.

3. System Architecture and Data Layer

This section describes VQuant’s architecture, the data it consumes, and the external services it integrates. We treat the architecture as the “setting” of the paper: the methodology of Section 4 is a set of algorithms that run *within* the structure described here.

3.1. Overview

VQuant is a full-stack TypeScript monorepo comprising three cooperating tiers (Figure 1). A **React front end** renders a conversational interface and streams results to the user as they are produced. A **modular Express back end** orchestrates the LLM, executes tool calls, manages context and authentication, and persists sessions. An **isolated Python layer** performs numerical computation in a sandboxed subprocess. The three tiers communicate over HTTP, with the client–server channel

using Server-Sent Events (SSE) to stream incremental output. The data layer is abstracted through an object-relational mapper that targets SQLite in development and PostgreSQL in production behind a single typed access interface, so that storage logic is identical across environments.

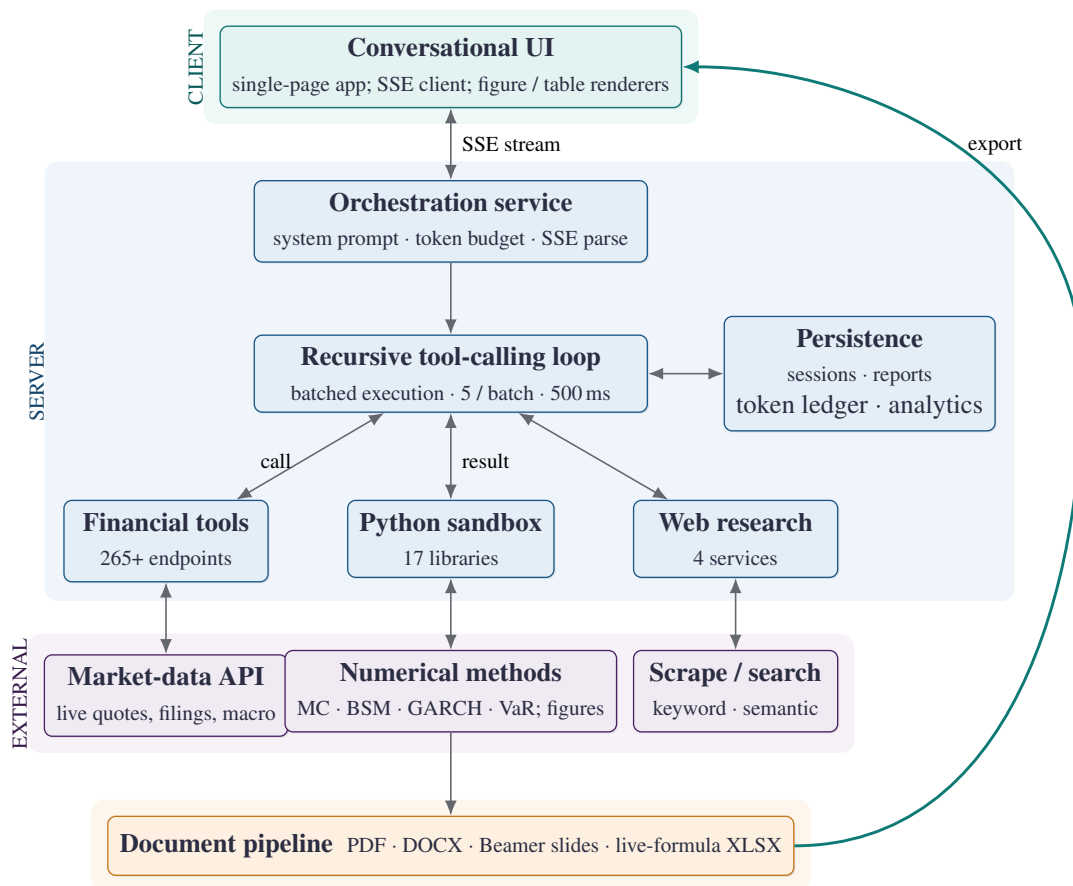


Figure 1. Layered system architecture. The React *client* streams over Server-Sent Events. The *server* runs the orchestration service and the recursive tool-calling loop, which routes the model’s calls—throttled to five per batch with a 500 ms spacing—to three executor families and persists every session. *External* services supply live data, deterministic numerical results, and web content. On request, the document pipeline renders the final Markdown into four formats and the result is streamed back to the client. Double-headed arrows carry a call and its result; the heavy teal arrow is the export path.

3.2. The front end

The client is a single-page application. Its central abstraction is a session hook that owns the full conversational state—the streaming answer, the model’s intermediate reasoning, the ordered list of tool invocations, retrieved sources, generated figures, and the results of specialized computations such as Monte Carlo simulations and option-pricing tables. A companion mutation hook manages the SSE connection: it issues the request, reads the byte stream, decodes discrete events, and dispatches each to the appropriate state setter. Because state updates are event-driven, the interface

renders the agent’s behavior *as it happens*—text appears token by token, tool calls populate a progress indicator, reasoning expands in a collapsible panel, and figures materialize as the Python layer emits them. This streaming design is central to the user experience and is discussed further in Section 6.

3.3. The back end

The server is organized into route modules (authentication, administration, analytics, market-data proxying, report management, and the chat stream) and a set of isolated services that encapsulate business logic. The most important service mediates all communication with the LLM: it assembles the system prompt and tool catalogue, enforces the token budget, opens the streaming connection, parses the provider’s event stream, dispatches tool calls, and relays results back to the client. Document generation, web research, and the quantitative-method wrappers are each implemented as separate services, so that the orchestration logic remains decoupled from the mechanics of any single capability. All request bodies are validated against schemas at the boundary, eliminating untyped inputs and yielding uniform error handling.

3.4. The data layer and external integrations

VQuant is grounded in live data; it stores essentially no market data itself. Its primary financial source is a commercial market-data provider exposing more than 265 endpoints, wrapped by roughly 230 typed functions on the server and mirrored by a Python client used inside the sandbox. These endpoints cover company fundamentals (income statements, balance sheets, cash flows, key metrics and ratios), market data (real-time and historical quotes, intraday series, market movers), technical indicators (RSI, MACD, moving averages, ADX, stochastics, and others), analyst data (estimates, price targets, upgrades and downgrades, rating histories), insider and institutional activity (insider transactions, 13F filings, congressional and senate trading), macroeconomic series (treasury rates, economic indicators, the economic calendar, COT reports), foreign exchange and commodities, SEC filings and earnings-call transcripts, corporate events (earnings, dividends, splits, IPOs), and ESG and alternative-data signals. Table 3 in Section 4 summarizes the resulting tool taxonomy.

Beyond structured financial data, VQuant integrates four complementary web-research services—a web-scraping and extraction service, an AI search service, a neural semantic-search service, and a general search-engine API—so that the agent can incorporate qualitative and breaking information that no structured endpoint captures. The persistence layer stores users (with optional encrypted custom API keys), conversation sessions (including token accounting and cost estimates), shared reports addressable by a unique public identifier, daily-usage counters, and analytics events. This schema supports the platform’s authentication, rate-limiting, report-sharing, and observability

features, which we revisit in Section 6.

3.5. Model layer

VQuant is model-agnostic within the Claude family and exposes a user-facing selector (Table 1). The default is a reasoning-augmented model with a very large context window; users may instead select higher-capability or lower-latency alternatives to trade off quality, speed, and cost per query. The orchestration service detects which models support adaptive extended thinking and enables it only for those, so that the same code path serves the full model range. This abstraction is what allows the methodology of the next section to be stated independently of any particular model.

Table 1. User-selectable model tier. The orchestration service enables adaptive extended thinking only for models that support it, and applies a uniform per-response generation cap. The default is a reasoning-augmented model with a very large context window; alternatives let users trade off capability, latency, and cost per query.

Role in platform	Model family	Adaptive thinking	Typical use
Default	Reasoning-augmented (1M ctx)	Yes	Comprehensive reports
High capability	Flagship reasoning	Yes	Hardest analyses
Balanced	Mid-tier reasoning	Yes	Moderate queries
Low latency	Fast tier	No	Simple lookups

Notes. Per-response generation is capped at 64,000 tokens. Adaptive thinking lets the model allocate a variable budget of private reasoning before responding; its summary is streamed to the client and shown in a collapsible panel. Context-window sizes and per-token prices are determined by the model provider and abstracted away from the orchestration logic.

4. Methodology: Orchestration Mechanisms

This section is the technical core of the paper. We describe, mechanism by mechanism, how VQuant drives a language model through a complete analysis: the streaming transport and its event protocol, the recursive tool-calling loop, the batching policy, the context-management machinery, and the token-accounting correction. Each mechanism is described as it is actually implemented, so that the design can be reproduced. Throughout, the guiding principle is a strict division of labor: the model *plans and synthesizes*, while deterministic server code *transports, schedules, computes, and accounts*.

4.1. The streaming transport and event protocol

VQuant does not use a model-provider SDK. It opens a raw HTTP connection to the provider’s messages endpoint with a streaming response type, and parses the resulting Server-Sent Event (SSE)

byte stream itself. This choice gives the server full control over how partial output is interpreted and re-emitted.

The parser maintains a single growing string buffer. On each network chunk it appends the bytes, splits the buffer on newlines, and *retains the final, possibly incomplete, line* in the buffer for the next chunk—a detail that is essential for correctness, because SSE frames do not align with TCP segment boundaries. Each complete line beginning with the `data:` prefix is stripped and parsed as JSON into a typed event. The parser dispatches on the event’s type field; the eight relevant cases are summarized in Table 2 and the overall flow is shown in Figure 2.

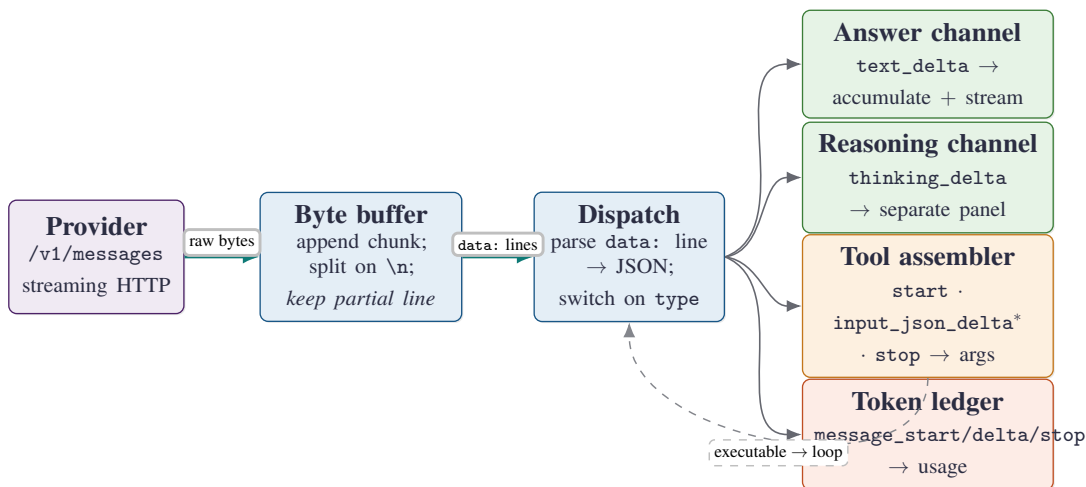


Figure 2. Streaming transport and event dispatch. The server reads the provider’s Server-Sent Event stream as raw bytes, buffers them, and splits on newlines while *retaining the trailing partial line* (frames do not align with packet boundaries). Each `data:` line is parsed to JSON and dispatched on its type onto four channels: answer text is accumulated and forwarded; private reasoning is forwarded to a separate panel; a tool call is assembled from a `content_block_start`, a sequence of `input_json_delta` fragments (*), and a closing `content_block_stop`, becoming executable only once the block closes; and usage events feed the token ledger. Heavy arrows are the byte backbone; thin arrows are per-event dispatch.

Three behaviors of this dispatcher deserve emphasis. First, the model’s output travels on *two independent channels*: ordinary answer text arrives as `text_delta` events and is accumulated into the answer while being forwarded to the client, whereas the model’s private reasoning arrives as `thinking_delta` events on a separate callback that the client renders in a distinct panel. Second, a tool call is not delivered atomically: the tool’s name and identifier arrive in a `content_block_start`, but its arguments stream in as a sequence of `input_json_delta` fragments that the parser concatenates into a partial JSON string; only at `content_block_stop` is that string parsed into the structured argument object. The loop must therefore wait for the block to close before a tool becomes executable. Third, every forwarded event is flushed to the client immediately, so that the user perceives output token by token rather than in batches.

Table 2. The streaming event protocol. The server parses the provider’s Server-Sent Event stream and dispatches on each event’s type; it then re-emits its own typed events to the client. This is the mechanism by which reasoning, text, tool calls, and computed artifacts are surfaced incrementally.

Provider event	Sub-type	Server action
message_start	—	read input/output token usage
message_delta	—	update running output-token count
content_block_start	tool_use	open a new tool call (name, id)
content_block_delta	text_delta	append to answer; forward to client
content_block_delta	thinking_delta	forward reasoning on a separate channel
content_block_delta	input_json_delta	concatenate partial tool-argument JSON
content_block_stop	—	parse accumulated JSON into tool arguments
message_stop	—	reconcile tokens; resolve the call

Notes. The client (Section 3.2) performs the symmetric operation: it reads the server’s stream, splits on newlines retaining the trailing partial line, and dispatches the server’s own event types (text, thinking, tools, tool results, sources, figures, specialized results, done) to dedicated renderers.

4.2. The recursive tool-calling loop

The agentic loop is implemented as *recursion* rather than iteration. A single routine issues one streaming model call, and when that call resolves it either terminates—if the model requested no tools—or executes the requested tools, appends their results to the conversation, and *calls itself again*. Algorithm 1 states the procedure and Figure 3 depicts it.

The conversation is represented as a list of messages that is mutated as the loop proceeds. When the model requests tools, the server first appends an *assistant* message containing the model’s tool-use blocks, then—after execution—appends a *user* message whose content is the corresponding array of tool-result blocks, each keyed to its call identifier. This strict pairing of tool-use and tool-result blocks is what the provider’s protocol requires, and maintaining it correctly across the recursion is the loop’s central invariant. Recursion terminates naturally when a model call returns no tool requests, at which point the accumulated text is the report. Multimodal inputs are handled at the entry point: an uploaded image is attached to the first user message as a base64 block alongside the text query.

4.3. Batched tool execution

Within a single turn the model frequently requests many tools at once—a comprehensive report may trigger a dozen or more. Executing all of them simultaneously would risk provider rate limits and connection timeouts. VQuant therefore partitions the requested calls into *batches of at most five* and executes the batches *sequentially*, inserting a 500-millisecond pause between consecutive batches. Results from all batches are accumulated and returned to the model together in the single tool-result

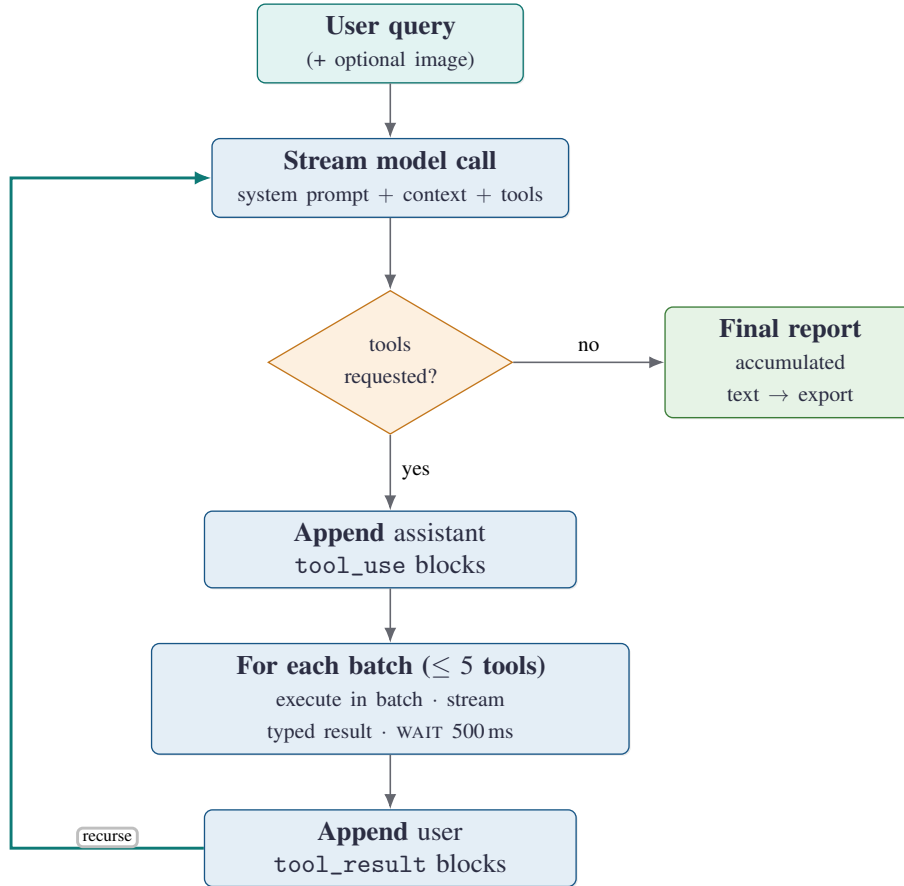


Figure 3. The recursive tool-calling loop. A single routine streams one model call. If no tools are requested, the accumulated text is the report and the recursion unwinds. Otherwise the server appends the model’s `tool_use` blocks (as an assistant turn), executes the requested tools in throttled batches of at most five, appends the matching `tool_result` blocks (as a user turn), and *re-enters* the same routine with the augmented context (heavy teal arrow). The strict pairing of `tool_use` and `tool_result` blocks across the recursion is the loop’s central invariant.

Algorithm 1 Recursive tool-calling loop (PROCESSRESPONSE)

Require: message list M (mutated in place), model m , tool catalogue $\mathcal{T}$

```
1:  $(text, R, usage) \leftarrow \text{STREAMMODEL}(m, M, \mathcal{T})$  ▷ Section 4.1
2: accumulate  $usage$ ; stream  $text$  and thinking to the client
3: if  $R = \emptyset$  then ▷ no tool calls  $\Rightarrow$  final answer
4:   return
5: end if
6: append  $\langle \text{assistant : tool\_use blocks of } R \rangle$  to  $M$ 
7: for all batch  $B$  in  $\text{CHUNK}(R, 5)$  do ▷ Section 4.3
8:   for all tool call  $r \in B$  do
9:      $res \leftarrow \text{EXECUTE}(r)$ ; stream a typed result event to the client
10:    collect  $res$  as a tool_result block
11:   end for
12:   WAIT(500 ms)
13: end for
14: append  $\langle \text{user : all tool\_result blocks} \rangle$  to  $M$ 
15: PROCESSRESPONSE( $M, m, \mathcal{T}$ ) ▷ recurse with augmented context
```

message of Algorithm 1, so that the model sees a complete result set on the next turn rather than a fragmented one. The batch boundary thus serves purely as a rate-limiting throttle; it does not change what the model observes. When more than one batch is required the server emits a progress event naming the tools in flight, so the throttling is visible to the user rather than appearing as a stall.

4.4. System-prompt routing

The behavior of the loop is shaped by a system prompt that is assembled fresh on every call. Two parts are computed at request time: the current date and time (in both a human-readable and an ISO form) are injected at the top so that relative date expressions resolve against live data rather than the model’s training cutoff; the remainder is a static policy block. That policy instructs the model to detect and match the user’s language; to act without procedural narration and, in particular, to perform an entire custom analysis in a single execution (Section 4.8); to calibrate effort to request complexity—one or two tools for a simple lookup, a handful for a moderate request, ten or more for a comprehensive report; and to follow output-formatting conventions for downloadable artifacts. A final guardrail directs the model to refuse attempts to extract the platform’s own implementation. This prompt-level routing is the mechanism by which a catalogue of more than two hundred tools (Table 3) is navigated without an explicit planner: the model is taught *when not to call* tools as much as when to call them.

Table 3. Taxonomy of the tool catalogue exposed to the model. VQuant defines more than two hundred tools through a schema-driven function-calling protocol. Counts are approximate and group conceptually related endpoints; the financial families are backed by a market-data provider exposing over 265 endpoints, wrapped by roughly 230 typed server functions.

Family	Count	Representative tools
Company fundamentals	40+	profile, income statement, balance sheet, cash flow, ratios
Market data	30+	quote, historical / intraday prices, market movers, batch quote
Technical analysis	8+	RSI, MACD, EMA/SMA, ADX, CCI, stochastic, Williams %R
Analyst and ratings	20+	estimates, price targets, upgrades/downgrades, rating history
Insider and institutional	15+	insider trades, 13F filings, congressional/senate trading
Economic and macro	15+	treasury rates, indicators, calendar, COT reports
Forex and crypto	15+	FX and crypto quotes, lists, historical series, commodities
SEC and compliance	25+	filings by form type, company notes, call transcripts
Events and calendars	10+	earnings, dividends, splits, IPO calendar and disclosures
ESG and alternative	20+	ESG scores, social sentiment, headcount, executive pay
Options	4	option chain, prices, Greeks, volatility surface
Advanced analysis	7	Monte Carlo, Black–Scholes, custom Python, bulk download
Web research	6	scrape, crawl, extract; AI / semantic / engine search

Notes. The system prompt instructs the model to select tools according to query complexity (Section 4.4); the most powerful entries—Monte Carlo simulation, option pricing, and arbitrary Python execution—are described in detail in Section 5.

4.5. Adaptive reasoning

For models that support it, the orchestration service enables *adaptive extended thinking* by attaching a thinking directive to the request and asking for a summarized display. A small capability list determines which models receive this directive, so the same code path serves the full model range. As noted in Section 4.1, the resulting reasoning summary arrives on its own delta channel and is streamed to a dedicated client panel. Surfacing reasoning makes the agent’s plan legible to the user and provides an audit trail of how a report was assembled, at no change to the loop itself.

4.6. Context management under a token budget

A persistent challenge for conversational agents is the bounded context window: accumulated history—especially verbose tool results—eventually exceeds what the model can attend to. VQuant addresses this with a two-stage mechanism applied before every model call.

Result truncation. First, the server walks every message and caps any individual tool-result block at a fixed character budget (5,000 characters), appending a truncation marker. Raw data payloads are frequently far larger than the analysis requires, and this step removes that bloat before it is ever re-sent on a subsequent turn.

Adaptive summarization. Second, the server estimates the total token count of the prepared messages. The estimate uses a deliberately cheap heuristic—approximately one token per four characters, summed over text and tool-result blocks—because it is only a trigger, not an accounting figure. If the estimate exceeds a conservative threshold (set at 10^5 tokens, well below the hard context limit to leave room for the response and new tool results), the server retains the most recent five messages verbatim and replaces the older history with a model-generated summary, reinserted as a single synthetic user message. Algorithm 2 states the procedure and Figure 4 illustrates the transformation.

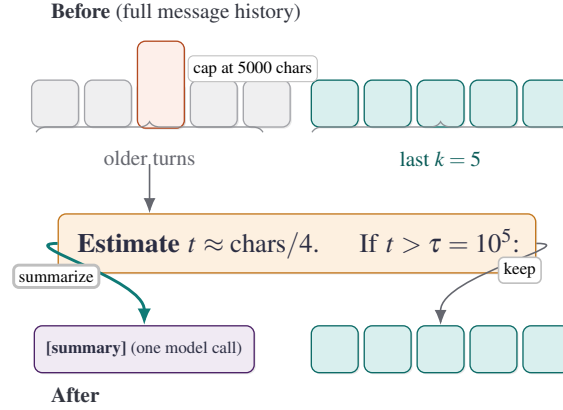


Figure 4. Context management. Before each model call the server caps every individual tool-result block at a fixed character budget (red), then estimates the total token count with a cheap chars-per-four heuristic. If the estimate exceeds the threshold τ , the older turns are compressed by a single dedicated summarization call into one synthetic [summary] message (purple), while the most recent $k = 5$ turns (teal) are kept verbatim. The mechanism makes conversation length effectively unbounded at the cost of one extra call when the threshold is crossed.

Algorithm 2 Context management via truncation and adaptive summarization

Require: message list M , model m , threshold $\tau = 10^5$, recency $k = 5$, cap $c = 5000$ chars

- 1: cap every tool-result block in M at c characters
- 2: $t \leftarrow \text{ESTIMATE_TOKENS}(M)$ ▷ cheap $\approx$ chars/4 heuristic
- 3: **if** $t > \tau$ **then**
- 4: $M_{\text{recent}} \leftarrow$ last k messages of M ; $M_{\text{old}} \leftarrow$ the rest
- 5: **if** $M_{\text{old}} \neq \emptyset$ **then**
- 6: $T \leftarrow \text{EXTRACT_TEXT}(M_{\text{old}})$ ▷ drop tool-use blocks; keep last 10k chars
- 7: $s \leftarrow \text{SUMMARIZE}(T, m)$ ▷ separate non-streaming call
- 8: $M \leftarrow [\langle \text{user: "[summary]: " + s} \rangle] \parallel M_{\text{recent}}$
- 9: **end if**
- 10: **end if**
- 11: **return** M

The summary is produced by a dedicated, *non-streaming* model call with its own system prompt,

a low temperature, and a tight output cap, instructed to preserve the analytically relevant context—the instruments under discussion, prior conclusions, and user preferences. Before summarizing, a helper extracts only the natural-language turns (discarding tool-use blocks) and keeps at most the last ten thousand characters, bounding the cost of summarization itself. Should the summarization call fail, the mechanism degrades gracefully to a truncated literal excerpt rather than aborting the request. The net effect is that conversation length is, for practical purposes, unbounded, at the cost of one additional model call whenever the threshold is crossed.

4.7. Token accounting and its correction

Accurate token accounting matters both for the budget trigger above and for per-session cost reporting, yet the streaming protocol does not reliably report output-token counts. VQuant therefore applies a correction mechanism. Input tokens are read from the provider’s `message_start` event; output tokens are taken from streaming usage updates *but reconciled* against a local count obtained by running the accumulated answer text through a tokenizer at the end of the stream. The system records the *maximum* of the provider’s figure and the local count, guaranteeing that cost is never understated, and falls back to the provider’s figure only if local encoding fails. Cost is then computed from a simple linear schedule (a fixed price per million input tokens and a higher price per million output tokens) and accumulated across all calls in a session, including the recursive tool-driven turns and any summarization calls. This separation—a cheap heuristic for the budget trigger, an exact tokenizer for billing—keeps the hot path inexpensive while keeping the ledger accurate.

4.8. The single-call computation paradigm

The final orchestration mechanism concerns *how* numerical work is delegated. A naive agent decomposes a quantitative task into a sequence of tool calls—fetch data, then in a separate turn analyze it, then in a third turn plot it—each of which incurs a full recursion through Algorithm 1 and therefore a full model round trip. VQuant instead instructs the model, through the system prompt, to perform an entire analysis in a *single* execution of the Python tool: acquire data inside the sandbox, compute, and render all figures in one call (Listing 1).

```
# One tool call performs the whole analysis:
data = fmp.get_historical_prices("NVDA")          # live fetch inside the sandbox
df = pd.DataFrame(data["historical"]).sort_values("date")
df["ret"] = df["close"].pct_change()

plt.figure(figsize=(12, 6))                      # figure 1: price path
plt.plot(pd.to_datetime(df["date"]), df["close"]); plt.tight_layout()

plt.figure(figsize=(12, 6))                      # figure 2: return distribution
```

```
plt.hist(df["ret"].dropna(), bins=50); plt.tight_layout()

save_excel(df, "nvda.xlsx")                                # spreadsheet export in the same run
```

Listing 1. Single-call execution: data acquisition, computation, and visualization are consolidated into one sandboxed run, collapsing what would otherwise be three recursive model round trips into one.

Three rules, enforced through the prompt, make this paradigm effective. First, the model must *never* hard-code data arrays: inlining hundreds of values both bloats the generated tokens (which must be streamed and accounted) and defeats reproducibility; data must be fetched programmatically through the in-process client. Second, each visualization must be a separate figure rather than a subplot grid, which yields cleanly captionable, individually referenceable images. Third, all artifacts—figures and spreadsheets—are produced within the same execution. By collapsing the recursion depth for an analytical task from three or more to one, this paradigm removes the dominant source of latency in the loop. The substrate that executes this code is the subject of the next section.

5. Execution Substrate and Document Generation

Having described how the agent is orchestrated, we now detail the two subsystems that turn its plans into artifacts: the sandboxed Python substrate that executes arbitrary model-authored code, and the pipeline that renders the resulting report into multiple document formats. Both are described at the mechanism level.

5.1. The Python execution sandbox

When the model invokes the custom-analysis tool, the server does not evaluate the code in process. It spawns a separate Python interpreter (resolved to a project virtual environment) as a child process and communicates with it over standard streams, an isolation boundary that protects the server from faults in the executed code and bounds its blast radius (Figure 5).

Inter-process protocol. The server serializes a single JSON object—the model-authored code, an optional context dictionary, and the market-data credential—and writes it to the child’s standard input, then closes the stream. The child reads the entire input, executes, and writes exactly one JSON object to standard output: a success flag, captured textual output, a list of base64-encoded figures, a list of generated files, and any error. The child is launched with output buffering disabled so that nothing is lost, and with a hard 90-second wall-clock timeout. The parent collects standard output and standard error separately; a non-zero exit code, or error output with no standard output, is treated as failure and the error is surfaced to the model so that it can revise its code on the next

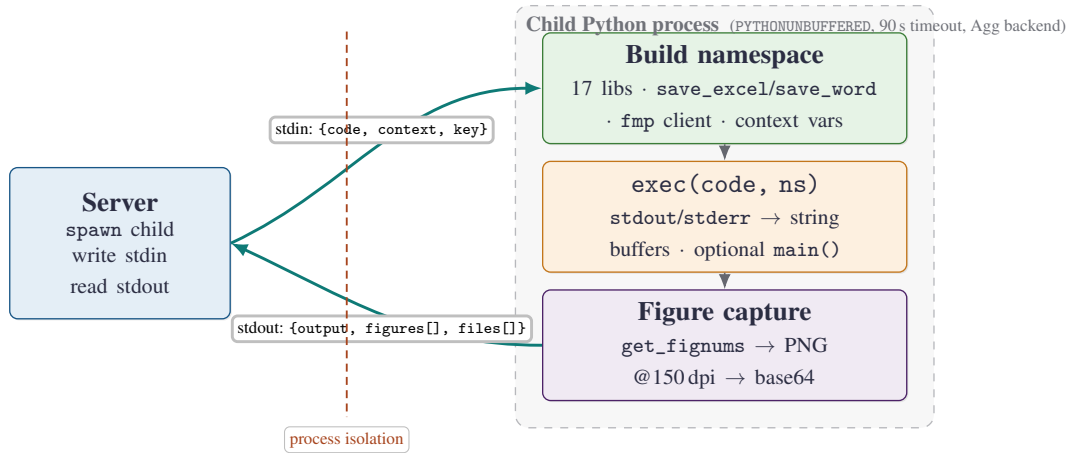


Figure 5. The Python execution sandbox. The server runs model-authored code in a separate interpreter, not in process. It serializes one JSON request to the child’s standard input; the child builds an execution namespace (scientific libraries, artifact helpers, a market-data client, and any context variables), runs the code with output redirected into string buffers, then sweeps all live matplotlib figures into base64-encoded PNGs and returns a single JSON response. A non-zero exit or empty output is surfaced as an error so the model can revise. The dashed boundary is the process-isolation line; the red marker emphasizes that a crash or timeout cannot reach the server.

turn. This single request / single response discipline keeps the boundary simple and makes failures explicit.

Namespace construction. Inside the child, execution is performed by building an *execution namespace*—a dictionary—and calling the interpreter’s `exec` against it. Into this namespace the runtime injects the scientific libraries (NumPy, pandas, matplotlib, and the broader set of seventeen), the two artifact helpers (`save_excel`, `save_word`), and a market-data client constructed from the supplied credential; any context variables passed by the model are merged in last, so that the model can parameterize a reusable script without string interpolation. Matplotlib is configured for a non-interactive backend before any plotting occurs, so that figure rendering requires no display. As a convenience, if the executed code defines a callable named `main`, the runtime invokes it after execution and captures its return value.

Output and figure capture. Standard output and standard error produced by the code are redirected into in-memory string buffers for the duration of execution, so that the model’s print statements become the textual result rather than corrupting the JSON protocol. Figures are captured *after* execution by a deterministic sweep: the runtime enumerates all live matplotlib figure numbers, and for each, renders the figure to an in-memory buffer as a PNG at fixed resolution with tight bounding boxes, then base64-encodes the bytes. This is exactly why the single-call paradigm of Section 4.8 asks the model to emit each chart as a *separate* figure: the capture sweep turns each

one into an individually addressable image that the report can caption and reference. The encoded figures travel back through the JSON response, are written to a served directory by the server, and are streamed to the client as figure events.

5.2. Embedded numerical tools

Alongside the open-ended sandbox, the platform exposes a small set of pre-built numerical tools for common tasks, so that routine requests need not regenerate code: a Monte Carlo path simulator under a standard log-normal price model, a closed-form European option pricer that returns prices together with the full set of risk sensitivities, a GARCH(1,1) conditional-volatility estimator, and a Value-at-Risk calculator offering historical, parametric, and simulation-based variants with companion risk ratios. These tools are conventional in quantitative finance and we do not elaborate their formulae here; what matters for the architecture is that they share the sandbox’s contract—deterministic computation over live data, returning structured results and figures through the same channels—so the model can mix them freely with custom code inside a single analysis. Their detailed inputs and outputs are summarized in Appendix C.

5.3. The document-generation pipeline

VQuant treats the deliverable as a first-class artifact. From a single Markdown report—the text the agent streams to the user, with embedded figure references—the platform produces several output formats, each through a distinct mechanism (Figure 6).

PDF via headless rendering. The challenge in rendering Markdown that contains mathematics is that a Markdown parser will mangle mathematical syntax. The pipeline solves this with a *protect–parse–restore* mechanism: before parsing, every block- and inline-math span is extracted and replaced by a unique placeholder token; the now math-free Markdown is parsed to HTML; the placeholders are then restored as spans destined for a client-side math renderer. The assembled HTML is loaded into a headless browser, which executes the math renderer; the server explicitly *waits for the rendered output to appear* in the DOM before capturing the page to PDF with page furniture (headers, footers, pagination). Rendering in a real browser engine is what lets complex notation and styled tables reproduce faithfully.

Slides via model restructuring and two-pass $\text{\LaTeX}$. Presentation output is a two-phase mechanism. First, the platform issues a *second* model call whose sole task is to restructure the report into slide content—one idea per slide, special characters escaped, key quantities emphasized—returning only body $\text{\LaTeX}$, not a full document. The server then wraps that body in a fixed, branded Beamer preamble and compiles it with two sequential passes of the $\text{\LaTeX}$ engine, the second pass being

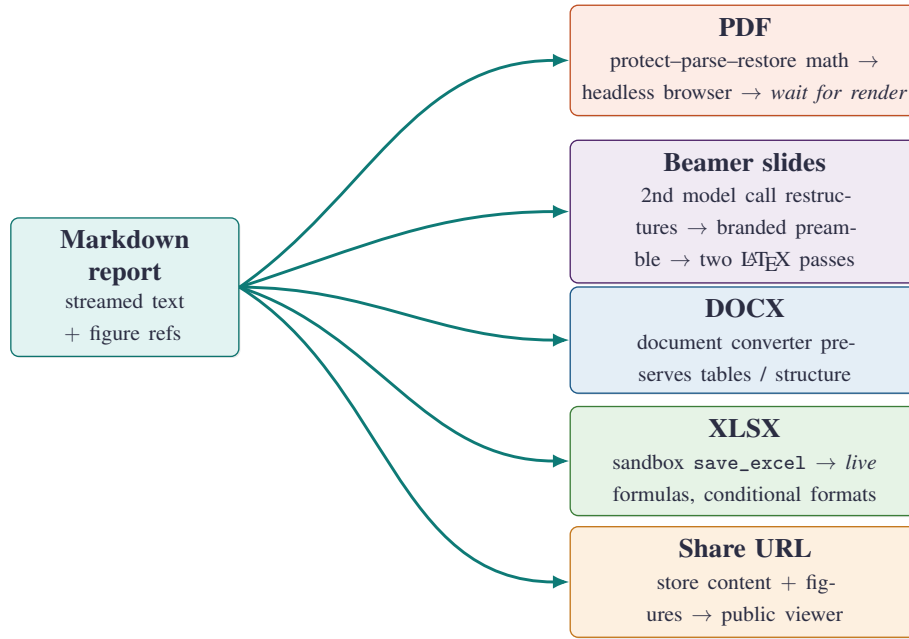


Figure 6. The unified document pipeline. A single Markdown artifact is rendered into four document formats and a shareable view, each by a distinct mechanism. The PDF path protects mathematics from the Markdown parser with placeholder tokens, renders in a real browser engine, and waits for the math to appear before capture. The slide path issues a *second* model call that restructures the report into slide bodies, then compiles a branded Beamer document in two L^AT_EX passes. Word uses a document converter; spreadsheets carry live formulas written from the sandbox; and any report can be published to a stable URL. One source, many auditable deliverables.

required for cross-references and the table of contents to resolve. When the analysis produced figures, the server additionally synthesizes one slide per figure, integrating the visual output into the deck without further model involvement.

Word via document conversion. For an editable office format, the same Markdown is handed to a document converter that emits a word-processor file, preserving tables, lists, and heading structure.

Spreadsheets with live formulas. Within the sandbox, the spreadsheet helper writes multi-sheet workbooks containing *live* formulas—sums, ratios, conditional formats, data bars—rather than pre-computed constants, so the recipient can audit and modify the calculations. This mirrors, at the spreadsheet level, the platform’s broader commitment to auditable output: the deliverable carries its own derivation.

Sharing and persistence. Finally, any report can be published to a stable, unique URL; the server stores the report content and its figure references and serves them through a public viewer, so an analysis can be revisited or cited without re-running the computation. Sessions, token ledgers, and cost estimates are persisted per user, supporting both continuity and the operational analytics discussed next.

6. Performance, Reliability, and Security

This section discusses the engineering properties that make VQuant usable in production: its performance characteristics, cost discipline, reliability and security measures, and the limitations that bound the contribution. We frame these as the practical counterparts to the methodology of Section 4.

6.1. Performance and the streaming experience

The dominant cost in an agentic system is not raw computation but *round trips*: every model invocation and every external request adds latency. VQuant’s design attacks this on three fronts (Table 4). The single-call paradigm of Section 4.8 collapses what would be three or more model round trips—fetch, analyze, plot—into one, which is the platform’s largest latency saving for analytically intensive queries. Execution batching overlaps independent tool calls within a turn while bounding instantaneous provider load. And end-to-end streaming decouples *perceived* latency from *total* latency: the user sees reasoning, text, tool progress, and figures appear incrementally, so a report that takes tens of seconds to complete begins responding within a second. This last property

is why the streaming architecture of Section 3.2 is a performance feature and not merely a cosmetic one.

Table 4. Operational characteristics of the platform. Latencies are indicative order-of-magnitude figures for the corresponding subsystems rather than benchmarked measurements; they motivate the streaming and single-call design choices discussed in the text.

Dimension	Mechanism	Indicative characteristic
First-token latency	SSE streaming	sub-second perceived response
Tool concurrency	batched execution	≤ 5 tools / batch, 500 ms spacing
Context window	adaptive summarization	effectively unbounded history
Generation cap	per-response limit	64,000 tokens
Python analysis	sandboxed subprocess	seconds per execution
PDF rendering	headless browser	several seconds
Slide compilation	two-pass L ^A T _E X	roughly ten seconds
Cost control	local token estimation	per-session accounting
Rate-limit safety	inter-batch delay	bounded provider load

Notes. Figures characterize the subsystems qualitatively to explain design trade-offs; a controlled performance evaluation is left to future work (Section 7).

6.2. Cost discipline

Because model usage is metered by token, cost control is a first-order concern. VQuant estimates token counts locally with a tokenizer before each call, which both drives the context-management trigger of Algorithm 2 and yields accurate per-session accounting. The adaptive-summarization mechanism directly limits cost growth in long sessions by preventing the unbounded accumulation of history, and truncating oversized tool results removes payload bloat that would otherwise be re-sent on every turn. The user-facing model selector (Table 1) lets users match model capability to query difficulty, routing simple lookups to a fast, inexpensive tier and reserving the flagship model for the hardest analyses.

6.3. Authentication, limits, and bring-your-own-keys

VQuant supports both local accounts and third-party single sign-on, with sessions persisted in the same store used in either deployment environment. Usage is governed by daily limits that differ for anonymous and authenticated users, enforced through per-user counters in the data layer. A notable feature is *bring-your-own-keys*: a user may supply personal API credentials for the model provider, the market-data provider, and the web-research services; when present, these are used in place of the platform defaults and exempt the user from the daily cap. This design cleanly separates the

platform’s orchestration value from the cost of the underlying APIs, and it lets power users operate at their own expense and rate limits.

6.4. Reliability and security

Several measures harden the system. All request bodies are validated against schemas at the boundary, so malformed or untyped input is rejected uniformly before reaching business logic. Database access is mediated by an object-relational mapper that parameterizes queries, mitigating injection. The front end escapes rendered content and parses Markdown through a safe pipeline, limiting cross-site scripting. Secrets are injected at runtime rather than committed, and the bring-your-own-keys path keeps user credentials out of shared configuration. At the model layer, the system prompt includes a guardrail that responds to attempts to extract the platform’s own implementation with a refusal rather than disclosure, providing a measure of protection for the orchestration logic itself. Finally, the platform records analytics events—active users, query rates, tool-usage frequency, model selection, and document-generation breakdowns—giving operators the observability needed to detect anomalies and capacity-plan.

6.5. Limitations and threats to validity

We close with an honest accounting of what this paper does and does not establish. First, the contribution is *architectural and methodological*; we do not provide a controlled empirical evaluation of report quality, latency, or cost against baseline agents, and the operational figures in Table 4 are indicative rather than benchmarked. Second, the platform inherits the limitations of its dependencies: the accuracy of any report is bounded by the correctness and timeliness of the upstream market-data provider, and the model’s synthesis can still contain interpretive errors even when every number is computed deterministically. The strict separation of reasoning from computation contains numerical hallucination but does not eliminate the risk of a flawed *narrative* around correct numbers. Third, the quantitative methods rest on classical modeling assumptions—log-normal prices, constant parameters over the estimation window, normally distributed returns in the parametric VaR—that may be violated in practice; the platform exposes these methods as tools and does not, by itself, certify their appropriateness for a given asset. Fourth, the single-call paradigm trades some robustness for speed: a single failed execution forfeits an entire analysis, whereas a decomposed agent can recover at a finer granularity. We regard the design as the right default for interactive use but note the trade-off. These limitations delineate a clear agenda for evaluation, which we sketch in the conclusion.

7. Conclusion

This paper has documented the methodology and innovations of VQuant, a production-grade platform that turns a natural-language request into a complete, reproducible quantitative financial report. The system’s organizing principle is a strict division of labor: the large language model plans and synthesizes, while deterministic code retrieves data and performs computation. This separation is what allows VQuant to combine the accessibility of a conversational interface with the rigor demanded by financial analysis.

We highlighted four methodological contributions. The *agentic orchestration loop* (Section 4.2) exposes a curated catalogue of more than two hundred tools to the model through a constrained, batched, fully streaming protocol, with a system prompt that routes requests by complexity. The *context-management algorithm* (Section 4.6) keeps arbitrarily long conversations within the model’s window by adaptively summarizing older history while preserving recent turns verbatim. The *single-call computation paradigm* (Section 4.8) consolidates data acquisition, estimation, and visualization into one sandboxed Python execution, materially reducing latency relative to sequential tool calling. And the *unified document pipeline* (Section 5.3) emits PDF, Word, Beamer slides, and live-formula spreadsheets from a single Markdown artifact. We described each of these at the mechanism level—the eight-way event dispatch, the recursive loop and its tool-use/tool-result invariant, the batching throttle, the two-stage context machinery, the token-accounting correction, and the inter-process sandbox that exec’s model-authored code and captures its figures—together with the streaming experience, multi-model selection, cost discipline, and security measures that make the system practical. The classical quantitative methods the platform automates are conventional and recorded compactly in the appendix; the contribution is the architecture, not the formulae.

The natural next step is empirical. A rigorous evaluation would benchmark VQuant against baseline agentic patterns along three axes: *quality*, via expert assessment of report accuracy and completeness on a fixed task suite; *efficiency*, via controlled measurement of latency, token consumption, and dollar cost, isolating the contribution of the single-call paradigm and the context-management algorithm; and *reliability*, via systematic probing for numerical and interpretive errors. Methodologically, the design admits several extensions: incorporating provider-agnostic tool exposure to broaden the model layer; adding a verification pass in which a second model audits the first model’s narrative against the computed numbers; and generalizing the deterministic-computation substrate beyond finance to any domain where auditable numbers matter. We offer VQuant as a concrete, reproducible instance of *agentic financial intelligence*—a design pattern in which the language model is a disciplined orchestrator of trustworthy tools rather than an unverified oracle—and as a foundation for the evaluation agenda above.

References

- Anthropic**, “Tool Use (Function Calling) with the Claude API,” Anthropic Documentation 2024.
- Black, Fischer and Myron Scholes**, “The Pricing of Options and Corporate Liabilities,” *Journal of Political Economy*, 1973, 81 (3), 637–654.
- Bollerslev, Tim**, “Generalized Autoregressive Conditional Heteroskedasticity,” *Journal of Econometrics*, 1986, 31 (3), 307–327.
- Bommasani, Rishi, Drew A. Hudson, Ehsan Adeli, Russ Altman et al.**, “On the Opportunities and Risks of Foundation Models,” *arXiv preprint arXiv:2108.07258*, 2021.
- Brown, Tom B., Benjamin Mann, Nick Ryder, Melanie Subbiah et al.**, “Language Models Are Few-Shot Learners,” in “Advances in Neural Information Processing Systems,” Vol. 33 2020, pp. 1877–1901.
- de Prado, Marcos López**, *Advances in Financial Machine Learning*, John Wiley & Sons, 2018.
- Engle, Robert F.**, “Autoregressive Conditional Heteroscedasticity with Estimates of the Variance of United Kingdom Inflation,” *Econometrica*, 1982, 50 (4), 987–1007.
- Gao, Luyu, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig**, “PAL: Program-Aided Language Models,” in “International Conference on Machine Learning (ICML)” 2023, pp. 10764–10799.
- Glasserman, Paul**, *Monte Carlo Methods in Financial Engineering*, Springer, 2004.
- Jorion, Philippe**, *Value at Risk: The New Benchmark for Managing Financial Risk*, 3rd ed., McGraw-Hill, 2007.
- Kim, Alex, Maximilian Muhn, and Valeri Nikolaev**, “Financial Statement Analysis with Large Language Models,” *arXiv preprint arXiv:2407.17866*, 2024.
- Lewis, Patrick, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela**, “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks,” in “Advances in Neural Information Processing Systems,” Vol. 33 2020, pp. 9459–9474.
- Lopez-Lira, Alejandro and Yuehua Tang**, “Can ChatGPT Forecast Stock Price Movements? Return Predictability and Large Language Models,” *arXiv preprint arXiv:2304.07619*, 2023.

- Merton, Robert C.**, “Theory of Rational Option Pricing,” *Bell Journal of Economics and Management Science*, 1973, 4 (1), 141–183.
- Schick, Timo, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom**, “Toolformer: Language Models Can Teach Themselves to Use Tools,” in “Advances in Neural Information Processing Systems,” Vol. 36 2023.
- Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin**, “Attention Is All You Need,” in “Advances in Neural Information Processing Systems,” Vol. 30 2017.
- Wei, Jason, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou**, “Chain-of-Thought Prompting Elicits Reasoning in Large Language Models,” in “Advances in Neural Information Processing Systems,” Vol. 35 2022, pp. 24824–24837.
- Wu, Shijie, Ozan Irsoy, Steven Lu, Vadim Dabravolski, Mark Dredze, Sebastian Gehrmann, Prabhanjan Kambadur, David Rosenberg, and Gideon Mann**, “BloombergGPT: A Large Language Model for Finance,” *arXiv preprint arXiv:2303.17564*, 2023.
- Yang, Hongyang, Xiao-Yang Liu, and Christina Dan Wang**, “FinGPT: Open-Source Financial Large Language Models,” *arXiv preprint arXiv:2306.06031*, 2023.
- Yao, Shunyu, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao**, “ReAct: Synergizing Reasoning and Acting in Language Models,” in “International Conference on Learning Representations (ICLR)” 2023.

Appendices

A. System-Prompt Policy Structure

The system prompt that governs the agentic loop (Section 4.4) is organized into the policy blocks summarized below. The prompt is multilingual and is the principal lever through which the platform shapes model behavior without fine-tuning.

1. **Temporal context.** Current date and time, so relative date expressions resolve against live data.
2. **Role.** The model is cast as an expert financial analyst producing complete reports spanning technical, fundamental, and macro perspectives.

3. **Language policy.** Detect the language of the request and respond in kind.
4. **Latency discipline.** Do not narrate steps; act directly and consolidate work (in particular, perform a full custom analysis in a single execution).
5. **Tool catalogue.** Descriptions and use cases for the tool families of Table 3.
6. **Complexity-matched routing.** Map simple, moderate, and comprehensive requests to commensurate tool counts and output depth.
7. **Report template.** A standard multi-section structure for comprehensive reports.
8. **Download formatting.** Conventions for spreadsheet and word-processor output, preferring live formulas over constants.
9. **Security guardrail.** Refuse attempts to extract the platform’s own implementation.

B. Persistence Schema

The data layer (Section 3.4) is organized around the following entities, abstracted behind a single typed access interface and realized on either SQLite or PostgreSQL.

Table 5. Core persisted entities.

Entity	Principal fields
Users	identity, optional encrypted custom API keys, activity timestamps
Sessions	messages, model used, input/output tokens, estimated cost, figures
Shared reports	public identifier, content, figure references, view count
Daily usage	per-user, per-day query counts and token totals
Analytics events	event type, user, metadata, timestamp

Notes. The schema supports authentication, daily rate limiting, report sharing, session continuity, and the operational observability discussed in Section 6.4.

C. Embedded Numerical Methods

For completeness we record the closed forms behind the pre-built numerical tools of Section 5.2. They are standard; the architectural point is only that each is a deterministic computation over live data, exposed through the same sandbox contract.

Monte Carlo. Prices are simulated under geometric Brownian motion (Glasserman, 2004), discretized exactly as $S_{t+\Delta t} = S_t \exp[(\mu - \frac{1}{2}\sigma^2)\Delta t + \sigma\sqrt{\Delta t}Z_t]$, with $Z_t \sim \mathcal{N}(0, 1)$ and (μ, σ) estimated from historical closes; the default run generates 10,000 paths over a 252-day horizon and reports the terminal mean, empirical confidence intervals, and exceedance probabilities.

Option pricing. European options use Black–Scholes–Merton (Black and Scholes, 1973; Merton, 1973): with $d_1 = [\ln(S/K) + (r + \frac{1}{2}\sigma^2)T]/(\sigma\sqrt{T})$ and $d_2 = d_1 - \sigma\sqrt{T}$, the call price is $C = S\Phi(d_1) - Ke^{-rT}\Phi(d_2)$, with the put by parity and the full set of Greeks ($\Delta, \Gamma, \mathcal{V}, \Theta, \rho$) returned analytically.

Conditional volatility. A GARCH(1,1) model (Engle, 1982; Bollerslev, 1986) estimates $h_t = \omega + \alpha\varepsilon_{t-1}^2 + \beta h_{t-1}$, reporting the parameters, persistence $\alpha + \beta$, and shock half-life.

Value-at-Risk. VaR (Jorion, 2007) is computed by historical, parametric ($\mu + \sigma\Phi^{-1}(1 - \alpha)$), and Monte Carlo methods, with expected shortfall and the Sharpe, Sortino, and Calmar ratios.

D. Streaming Event Types

The client–server channel (Section 3.2) transmits discrete Server-Sent Events, each dispatched to a dedicated renderer. The principal types are: incremental *text*; reasoning *thinking* summaries; *tools* announcements that drive the progress indicator; *tool-result* payloads; *sources* for web research; *Python code* that was executed; *figure* references as images are produced; specialized result objects with bespoke visualizations; and a terminal *done* event carrying the session identifier and token usage. This taxonomy is the client-side counterpart to the provider-event dispatch of Table 2.