



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

October 2026

Version 1.1

Abstract

Large language models (LLMs) write fluently about finance, but fluency does not make a number correct, current, or reproducible. This paper is a design-science description of **VQuant**, a deployed web platform that couples a reasoning-capable LLM with a deterministic execution substrate so that a natural-language request (“analyse NVDA’s fundamentals and run a Monte Carlo simulation”) is turned into a report in which every figure is produced by live data retrieval or auditable code rather than by the model. We document four mechanisms as they are implemented and audit the description line by line against the platform’s source code (vquant.git, commit e2a148a): (i) an agentic orchestration loop that exposes 245 schema-defined tools—wrapping two market-data providers (249 typed functions over 233 endpoint paths), three web-research services, and a Python sandbox pre-loading 29 scientific packages—through a recursive, batched, streaming tool-calling protocol; (ii) a three-stage context-management procedure (per-result truncation at execution, a re-send cap, and adaptive summarisation under a token budget) that keeps long sessions within the model’s window; (iii) a single-call computation convention that consolidates data acquisition, estimation, and figure rendering into one sandboxed execution; and (iv) a document pipeline that renders one Markdown artifact into PDF, DOCX, $\text{\LaTeX}$ Beamer, and spreadsheets with live formulas. The paper reports no empirical evaluation: it contains no measurements of numerical accuracy, hallucination rates, latency, or cost, and makes no comparative claims. Its contribution is a reproducible specification of the architecture, an honest account of where the implementation departs from its own documentation, and a pre-registered evaluation protocol—benchmark, metrics, ablations, and baselines—that has not yet been executed.

Keywords: large language models, LLM agents, tool use, design science, quantitative finance, automated report generation, reproducibility

JEL Classification: C45, C63, C88, G17, O33

1. Introduction

Large language models (LLMs) 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, 2026; Wu et al., 2023; Hansen and Kazinnik, 2023). Fluency, however, is not rigour. A model that *describes* a value-at-risk calculation has not *computed* one; a model that *recalls* a price from its training data 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. Surveys of hallucination in natural-language generation (Ji et al., 2023) and of LLM evaluation (Chang et al., 2024) make the same point from the other side: generated numbers are not trustworthy unless they are produced, or at least checked, outside the model. The central engineering problem in applied financial AI is therefore *grounding*: how to bind a probabilistic text generator to live data, deterministic computation, and reproducible artifacts without giving up the conversational interface that makes the technology accessible.

This paper describes **VQuant**, a deployed web platform built around one answer to that problem. VQuant accepts a natural-language request—“analyse NVDA’s fundamentals, technicals, and run a Monte Carlo simulation,” for example—and returns a multi-section report with live data, computed figures, and downloadable documents. The platform is a full-stack TypeScript monorepo: a React front end, a modular Express back end, and an isolated Python scientific-computing layer. Its organising principle is a division of labour: the language model *plans and synthesises*, while deterministic server code *retrieves, schedules, computes, and accounts*. Every number in a VQuant report is meant to originate from a market-data endpoint, a closed-form formula, or Python code that the reader can inspect. The design belongs to the family of program-aided and tool-augmented language models (Gao et al., 2023; Chen et al., 2023a; Schick et al., 2023; Yao et al., 2023; Mialon et al., 2023); its specific contribution is to work out, at the level of a production system, what that family’s principles require when the tool surface is large, the data are live, and the deliverable is a document.

What kind of paper this is. This is a *design-science* paper in the sense of Hevner et al. (2004) and Gregor and Hevner (2013): it presents an artifact, states the problem it addresses, and specifies its construction precisely enough to be reproduced and later evaluated. It is *not* an empirical evaluation. We report no measurements of numerical accuracy against ground truth, no hallucination rates, no latency or cost benchmarks, and no ablations; where the first version of this paper used the words “results” and “robustness”, the present version uses “design rationale” and “threats to validity”. Section 6 instead specifies the evaluation we think the artifact warrants—benchmark, metrics, ablations, and baselines—and marks it explicitly as not yet executed. We regard this separation as a

matter of research integrity: a system paper that reports qualitative impressions as results invites exactly the kind of unverifiable claim that the system itself is designed to prevent.

Audit against the code. A second feature of this version is that every mechanism described has been checked against the platform’s source code at a fixed commit (`vquant.git`, commit `e2a148a`, 2 October 2026, hereafter “the audited commit”). Where the implementation differs from the platform’s own documentation, or from the first version of this paper, we say so. Three such discrepancies are material and are reported in the text: the tool catalogue contains 245 tools rather than the “200+” and “213” figures used in earlier descriptions; the token-accounting correction uses a general-purpose byte-pair tokenizer whose vocabulary differs from the provider’s, so the “exact” count of the earlier description is an approximation; and several pre-built numerical routines (Monte Carlo, GARCH, VaR, portfolio optimisation) have server-side handlers but are absent from the tool catalogue, so the model cannot invoke them. Appendix E tabulates each claim of the paper with the file that supports it.

Terminology. Three terms recur and are used in a fixed sense. An *agentic* loop is a control structure in which the model’s output can request actions (tool calls) whose results are fed back to the model, repeatedly, until it produces a final answer (Yao et al., 2023; Wang et al., 2024); the loop is agentic in this operational sense and no claim about autonomy is implied. *Tool use* (or function calling) is the provider-level protocol by which a model emits a structured call against a declared JSON schema and receives a structured result (Anthropic, 2026; Schick et al., 2023). A *reasoning-capable* model is one that can allocate an adaptive budget of private intermediate tokens to deliberation before and between its visible outputs (Wei et al., 2022); the platform surfaces a summary of this deliberation to the user but does not depend on its content.

Contributions. Within these bounds the paper makes four contributions. (1) **A specified orchestration loop.** Section 4 gives the streaming transport, the recursive tool-calling procedure, the batching throttle, and the prompt-level routing that let a model navigate a catalogue of 245 tools without an explicit planner. (2) **A specified context-management procedure.** We state, as an algorithm with its actual constants, the three-stage mechanism—per-result truncation at execution time, a re-send cap, and adaptive summarisation under a token budget—that keeps long sessions within the model’s context window, and we relate it to the known degradation of long-context attention (Liu et al., 2024a). (3) **A single-call computation convention and its substrate.** We argue that consolidating data acquisition, estimation, and visualisation into one sandboxed execution removes model round trips, and we specify the inter-process sandbox that executes model-authored code. (4) **A document pipeline.** We describe how one Markdown artifact becomes PDF, DOCX,

Beamer slides, and spreadsheets with live formulas. A fifth, methodological, contribution is the evaluation protocol of Section 6.6.

The classical quantitative methods the platform automates—option pricing, conditional-volatility modelling, simulation, and risk metrics—are standard and are recorded compactly in Appendix C. The remainder of the paper is organised as follows. Section 2 situates the artifact in the literatures on tool-augmented LLMs, LLM agents and their evaluation, and LLMs in finance and economics. Section 3 describes the architecture, data layer, and model layer. Section 4 develops the orchestration mechanisms. Section 5 describes the execution substrate and the document pipeline and walks through one request as an illustration. Section 6 gives the design rationale, threats to validity, reproducibility information, and the proposed evaluation protocol. Section 7 concludes.

2. Related Work

VQuant draws on four strands of research: the design-science tradition in information systems, which supplies the paper’s genre; the literature on tool-augmented language models and LLM agents, which supplies the mechanisms; work on the evaluation of such agents, which supplies the standard the paper does not yet meet; and the rapidly growing literature on LLMs in finance and economics, which supplies the application and its risks.

2.1. Design science and system descriptions

Information-systems research distinguishes behavioural studies, which test theories about the use of artifacts, from design-science studies, which build and evaluate artifacts that solve identified problems (Hevner et al., 2004). Peffers et al. (2007) give the canonical process—problem identification, objectives, design and development, demonstration, evaluation, communication—and Gregor and Hevner (2013) classify contributions by their maturity, from situated implementations to nascent design theories. In those terms the present paper reports a *situated implementation* together with a *demonstration* (a walk-through of one request); the *evaluation* step is specified but not performed. We adopt this vocabulary deliberately: it lets us describe a working system in detail while making clear which of the process’s steps remain open.

2.2. Tool-augmented language models and agents

A model confined to its parameters cannot retrieve a current price or perform exact arithmetic. Chain-of-thought prompting (Wei et al., 2022) improved multi-step reasoning but left computation inside the model; program-aided language models (Gao et al., 2023) and program-of-thoughts prompting (Chen et al., 2023a) moved it out, having the model write code that a deterministic

interpreter executes, and showed large gains on numerical benchmarks including financial question answering. Toolformer (Schick et al., 2023) taught models to insert API calls into their own text, and ReAct (Yao et al., 2023) interleaved reasoning traces with actions in an observe–think–act loop that has become the template for agents. The survey of Mialon et al. (2023) names this family *augmented language models* and organises it along the two axes—reasoning and tool use—that VQuant combines; Wang et al. (2024) survey the resulting agent architectures. Retrieval-augmented generation (Lewis et al., 2020) is the special case in which the tool is a document retriever; VQuant’s “retrieval” surface is structured (market-data endpoints), textual (web search and extraction), and computational (code), and the auditability it seeks is the financial analogue of citation grounding.

Two practical problems arise when the tool surface grows. The first is selection: Gorilla (Patil et al., 2024) and ToolLLM (Qin et al., 2024) fine-tune models to choose among thousands of APIs and introduce retrievers over tool documentation, whereas VQuant transmits its entire 245-tool catalogue on every call and relies on schema descriptions and a routing prompt. The second is context: tool results are verbose, and Liu et al. (2024a) show that models use information in the middle of long contexts poorly even when it fits. Provider-level function-calling protocols (Anthropic, 2026) and the Model Context Protocol (Model Context Protocol, 2025) standardise how tools are declared and invoked but leave both problems to the application. VQuant’s prompt-level routing (Section 4.4) and three-stage context management (Section 4.6) are its answers; neither is claimed to be optimal, and the evaluation protocol of Section 6.6 is designed to test them.

2.3. Evaluating LLM agents

The agent literature has converged on benchmark suites with executable environments and automatic scoring: AgentBench (Liu et al., 2024b) spans eight environments, Reflexion (Shinn et al., 2023) evaluates self-correction over repeated episodes, and WebGPT (Nakano et al., 2021) pioneered human-preference evaluation of browser-assisted answers. For open-ended outputs such as reports, model-based grading has become standard, with Zheng et al. (2023) documenting both its agreement with human raters and its biases. Chang et al. (2024) survey the resulting evaluation landscape. In finance specifically, FinanceBench (Islam et al., 2023) supplies questions over public filings with human-verified answers and reports that unaided retrieval-augmented models answer a majority of them incorrectly, and FinBen (Xie et al., 2024) assembles a multi-task financial benchmark that includes agent and quantitative-reasoning tasks. These resources are the natural starting point for an evaluation of VQuant; the protocol in Section 6.6 builds on them.

2.4. Large language models in finance and economics

Text has long been a source of financial signals: Loughran and McDonald (2016) survey dictionary-based textual analysis, Gentzkow et al. (2019) review text-as-data methods for economists, and Li et al. (2021) show that embedding-based methods recover latent constructs such as corporate culture from earnings calls. LLMs extend this line. Lopez-Lira and Tang (2026) find that ChatGPT-derived headline sentiment predicts subsequent returns; Chen et al. (2023b) use LLM representations of news to forecast returns across sixteen markets; and Hansen and Kazinnik (2023) show that GPT models classify the stance of FOMC communications more accurately than dictionary and smaller supervised methods.¹ Domain-adapted models such as BloombergGPT (Wu et al., 2023) and FinGPT (Yang et al., 2023) demonstrate the value of finance-specific training data, and Li et al. (2023) survey the field. On the applied side, Fatouros et al. (2025) build an LLM-driven stock-selection framework from news, fundamentals, and prices and evaluate it on realised performance, while Korinek (2023) and Dell (2025) assess what generative and deep-learning methods offer to economic research.

Most of this work treats the LLM as a *forecasting or classification* engine operating on text. VQuant takes a different stance: the model is an *orchestrator* that assembles live data, delegates numerical work to deterministic code, and composes the results into a document. The classical methods it automates—option pricing (Black and Scholes, 1973; Merton, 1973), conditional volatility (Engle, 1982; Bollerslev, 1986), Monte Carlo valuation (Glasserman, 2004), and Value-at-Risk (Jorion, 2007)—are established; the broader concern with reproducibility and provenance in financial machine learning is articulated by López de Prado (2018). The point of contact with the forecasting literature is the risk, documented there, that LLM outputs about numbers are confidently wrong; VQuant’s architecture is an attempt to remove numbers from the model’s remit altogether.

2.5. Positioning

Relative to this work, the artifact described here is distinctive in three respects, none of which is an empirical claim. First, it exposes a broad and heterogeneous catalogue of tools under one protocol and documents the practical mechanisms—routing, batching, truncation, summarisation—needed to operate it. Second, it enforces a strict separation between reasoning (performed by the model) and computation (performed by deterministic code), a separation that Gao et al. (2023) and Chen et al. (2023a) showed to be beneficial on benchmarks and that VQuant carries into a production setting with live data. Third, it treats the deliverable—an exportable, shareable document—as a primary artifact. What the paper does not establish is whether these choices improve accuracy, latency, or

¹A widely circulated study of LLM-based financial statement analysis (Kim et al., 2024) was withdrawn from arXiv by its authors in February 2025 pending a review of data inconsistencies; we therefore do not rely on it.

cost relative to simpler designs; that is the question Section 6.6 is written to answer.

3. System Architecture, Data Layer, and Model Layer

This section describes the setting within which the mechanisms of Section 4 operate: the three tiers of the platform, the external data and services it integrates, its persistence layer, and the model layer. All statements refer to the audited commit; where the first version of this paper described a different state, the difference is noted.

3.1. Overview

VQuant is a TypeScript monorepo with three cooperating tiers (Figure 1). A **React front end** renders a conversational interface and updates it as events arrive. A **modular Express back end** orchestrates the model, executes tool calls, manages context and authentication, and persists sessions. An **isolated Python layer** performs numerical computation in a sandboxed child process. The client–server channel uses Server-Sent Events (SSE) to stream incremental output. Persistence is abstracted through an object-relational mapper (Drizzle) with two parallel schema definitions—SQLite for development and single-node deployment, PostgreSQL for hosted deployment—behind one typed access interface, so that storage logic is identical across environments. The code base is organised into six route modules (authentication, administration, analytics, market-data proxying, report management, and the chat stream) and a set of services, each encapsulating one external capability; the repository ships with six automated test suites covering request validation and client state.

3.2. The front end

The client is a single-page application whose central abstraction is a session hook owning the full conversational state: the streaming answer, the model’s reasoning summary, the ordered list of tool invocations with their status, retrieved sources, generated figures, and the results of specialised computations. 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 behaviour as it happens—text appears incrementally, tool calls populate a progress indicator, reasoning expands in a collapsible panel, and figures appear as the Python layer emits them. The client-side event taxonomy is listed in Appendix D.

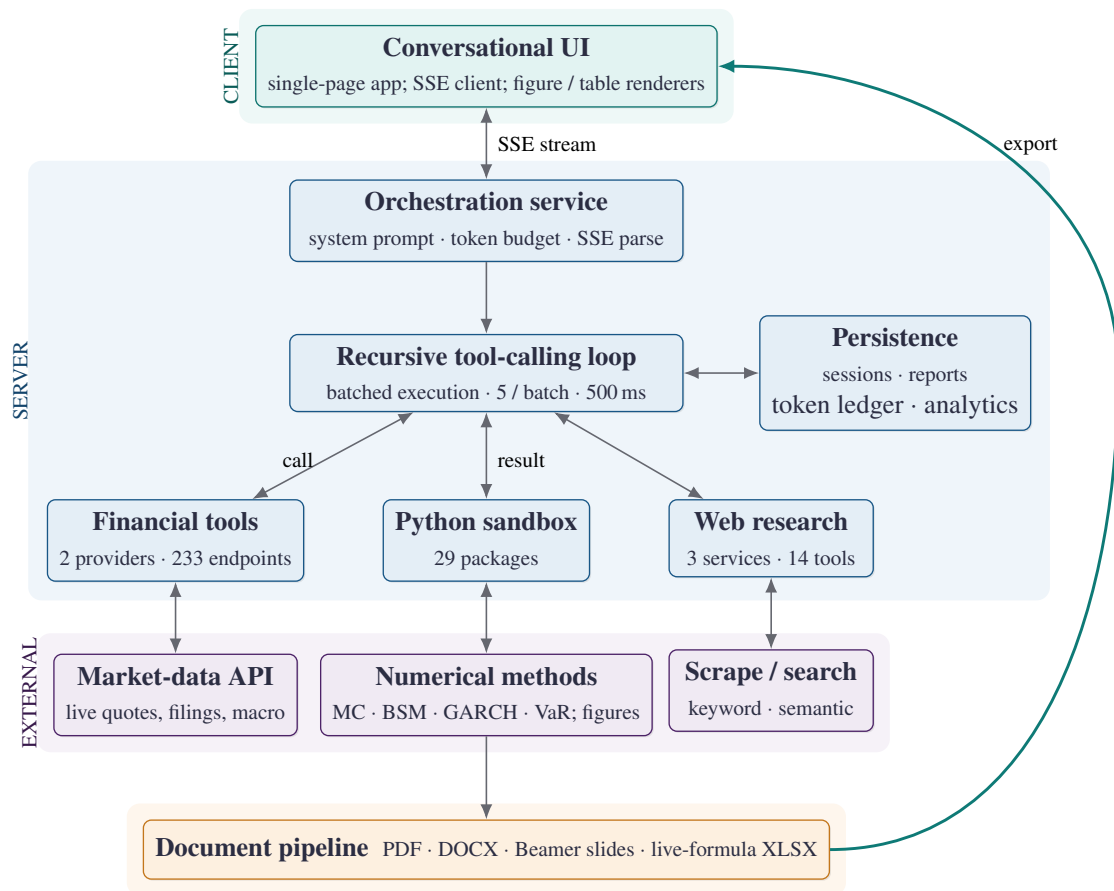


Figure 1. Layered system architecture. The React *client* streams over Server-Sent Events. The *server* runs the orchestration service and the recursive tool-calling loop, which routes the model’s calls—throttled to five per batch with a 500 ms spacing—to three executor families and persists every session. *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.

3.3. The back end

The most important service mediates all communication with the model provider: it assembles the system prompt and tool catalogue, applies the context-management procedure, opens the streaming connection, parses the provider’s event stream, and returns text, tool requests, and usage to the route that owns the recursive loop (Section 4.2). Document generation, web research, the two market-data clients, and the Python executors are separate services, so that orchestration logic is decoupled from the mechanics of any single capability. Request bodies are validated against Zod schemas at the boundary, which rejects malformed input uniformly before it reaches business logic.

3.4. Data layer and external integrations

VQuant stores essentially no market data; it is grounded in live retrieval. Its primary financial source is a commercial market-data provider (Financial Modeling Prep) whose “stable” API is wrapped by 249 exported, typed server functions spanning 233 distinct endpoint paths, and mirrored by a small Python client injected into the sandbox. A second provider (EODHD) supplies end-of-day and intraday history, fundamentals, and news for instruments on more than seventy exchanges outside the United States, and is the source of the platform’s options chains and Greeks. Together these endpoints cover company fundamentals, quotes and intraday series, technical indicators (several of which, such as MACD and CCI, are computed server-side from price history because the provider does not expose them), analyst data, insider and institutional activity, macroeconomic series, foreign exchange and commodities, SEC filings and earnings-call transcripts, corporate events, indices and funds, and ESG and alternative data. Table 3 in Section 4 gives the resulting tool taxonomy with exact counts.

Beyond structured data, three web-research services are exposed to the model as fourteen tools: a scraping, crawling, and extraction service (Firecrawl), an AI search and research service (Tavily), and a neural search service (Exa). A fourth service (a search-engine API) and a text-to-speech service are integrated server-side but have no tool definitions at the audited commit and are therefore not reachable by the model.

The persistence layer defines twelve entities (Appendix B): users, one-time authentication codes, trusted devices, a security event log, conversation sessions with their serialised messages and token and cost ledgers, individual messages, shared reports addressable by a public identifier, live-user presence, aggregated analytics metrics, request logs, and two tables for crawled pages and their embeddings. The first version of this paper described per-user encrypted API keys and daily usage counters; neither exists in any committed version of the schema, and the corresponding “bring-your-own-keys” feature described there is not implemented. We correct the record in Section 6.2.

3.5. Model layer

The platform exposes a user-facing model selector (Table 1). The reference path, and the one described in Section 4, targets the Anthropic Messages API: the default model is `claude-fable-5`, and the orchestration service enables adaptive extended thinking for the five models that support it (Table 1) by attaching a thinking directive of type `adaptive` with a summarised display; the Haiku model is served without it. A uniform generation cap of 64,000 output tokens applies, and the request sets the sampling temperature to 1 (the provider default). At the audited commit the selector also lists OpenAI and Google models, served through a second adapter that speaks the OpenAI-compatible chat-completions interface, converts the Anthropic-style tool schemas and message history into that interface’s function-calling format, and—because that interface caps the number of tools at 128—transmits only the first 128 tools of the catalogue. We do not audit those model identifiers or that path further; the first version of this paper described the platform as Claude-only, which was accurate for the version then deployed but is no longer so. What matters for the methodology is that the loop of Section 4 is written against one internal representation (Anthropic-style content blocks) and that provider differences are confined to the two transport adapters.

Table 1. Model layer at the audited commit. The reference path targets the Anthropic Messages API; adaptive extended thinking is enabled only for the models listed as supporting it. OpenAI and Google models are served through an OpenAI-compatible adapter that receives the first 128 tools of the 245-tool catalogue. Model identifiers are reproduced verbatim from the client configuration (`client/src/lib/models.ts`); the Anthropic identifiers were checked against the provider’s current model list, the others were not audited.

Provider path	Model identifier	Adaptive thinking	Context	Role in selector
Anthropic (reference)	<code>claude-fable-5</code>	yes	1M	default
Anthropic	<code>claude-opus-4-8</code>	yes	1M	long-horizon agentic
Anthropic	<code>claude-opus-4-7</code>	yes	1M	balanced
Anthropic	<code>claude-opus-4-6</code>	yes	1M	fast and capable
Anthropic	<code>claude-sonnet-4-6</code>	yes	1M	general purpose
Anthropic	<code>claude-haiku-4-5-20251001</code>	no	200K	lowest latency
OpenAI-compatible	twelve OpenAI identifiers	n/a	–	not audited
OpenAI-compatible	four Google identifiers	n/a	–	not audited

Notes. Per-response generation is capped at 64,000 tokens and the temperature is set to 1 on the reference path; the summarisation call of Algorithm 2 uses a cap of 2,000 tokens and a temperature of 0.5. The Beamer restructuring call (Section 5.3) is pinned to `claude-opus-4-8` irrespective of the selected model. Cost is computed from a single fixed schedule (\$3 per million input tokens, \$15 per million output tokens) for every model, so the ledger does not reflect per-model prices (Section 4.7).

4. 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 routing prompt, the context-management procedure, the token-accounting correction, and the single-call computation convention. Each mechanism is described as implemented at the audited commit, with its actual constants, so that the design can be reproduced; Appendix E gives the source location of every statement.

4.1. Streaming transport and event protocol

On the reference path VQuant does not use the provider’s software development kit. It opens a raw HTTP connection to the provider’s messages endpoint with a streaming response and parses the resulting SSE byte stream itself, which 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* 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, and the parser dispatches on the event’s type field. The eight relevant cases are summarised in Table 2 and the flow is shown in Figure 2.

Table 2. Provider-side 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 (Appendix D). 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 and output token usage
message_delta	—	update running output-token count
content_block_start	tool_use	open a new tool call (name, identifier)
content_block_delta	text_delta	append to answer; forward to client
content_block_delta	thinking_delta	forward reasoning summary 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 output tokens; resolve the call

Notes. Source: `server/services/claudeService.ts`. The client performs the symmetric operation on the server’s own stream: it reads bytes, splits on newlines retaining the trailing partial line, and dispatches the server’s event types to dedicated renderers.

Three behaviours of this dispatcher deserve emphasis. First, the model’s output travels on two independent channels: answer text arrives as `text_delta` events and is accumulated into

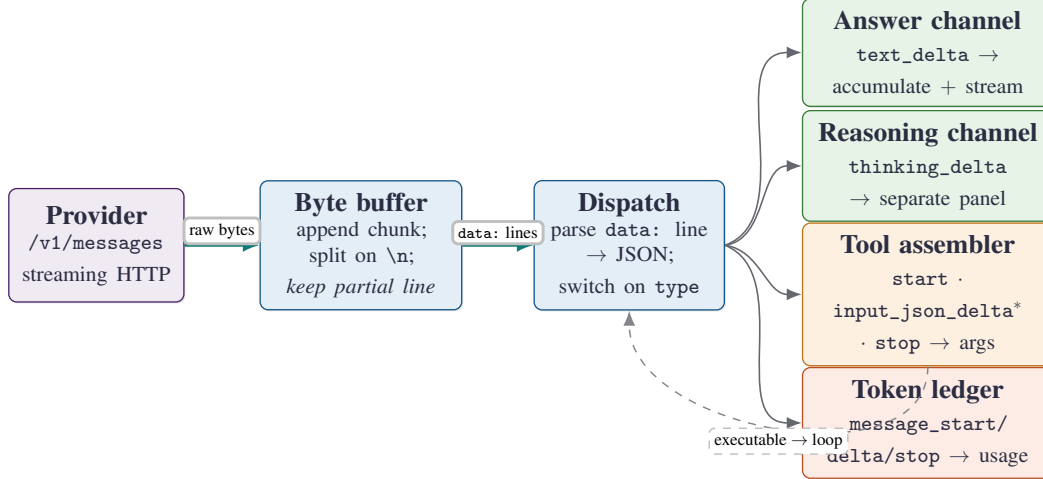


Figure 2. Streaming transport and event dispatch. The server reads the provider’s Server-Sent Event stream as raw bytes, buffers them, and splits on newlines while *retaining the trailing partial line* (frames do not align with packet boundaries). Each data: line is parsed to JSON and dispatched on its 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.

the answer while being forwarded to the client, whereas the model’s reasoning summary 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, its arguments stream in as a sequence of input_json_delta fragments that the parser concatenates into a partial JSON string, and 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 the user perceives output as it is generated rather than in batches.

4.2. The recursive tool-calling loop

The agentic loop is implemented as recursion. A single routine issues one streaming model call; when the 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 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 array of tool-result blocks, each keyed to its call identifier. This pairing of tool-use and tool-result blocks is what the provider’s protocol requires, and maintaining it across the recursion is the loop’s central invariant.

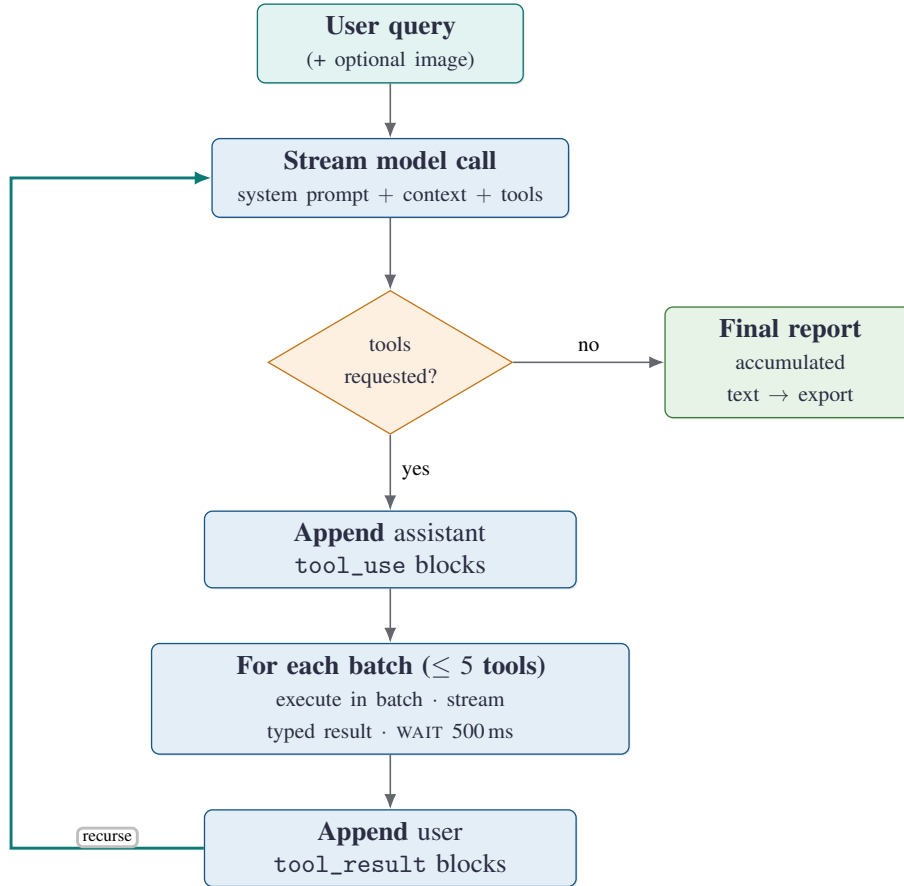


Figure 3. The recursive tool-calling loop. A single routine streams one model call. If no tools are requested, the accumulated text is the report and the recursion unwinds. Otherwise the server appends the model’s `tool_use` blocks (as an assistant turn), executes the requested tools in throttled batches 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 $\mathcal{T}$ ($|\mathcal{T}| = 245$)

- 1: $M \leftarrow \text{PREPARECONTEXT}(M, m)$ ▷ Algorithm 2
- 2: $(\text{text}, R, \text{usage}) \leftarrow \text{STREAMMODEL}(m, M, \mathcal{T})$ ▷ Section 4.1
- 3: accumulate usage ; stream text and reasoning summary to the client
- 4: **if** $R = \emptyset$ **then** ▷ no tool calls $\Rightarrow$ final answer
- 5: **return**
- 6: **end if**
- 7: append $\langle \text{assistant} : \text{tool_use blocks of } R \rangle$ to M ▷ tool-use blocks only; no text or thinking blocks
- 8: **for all** batch B in $\text{CHUNK}(R, 5)$ **do** ▷ Section 4.3
- 9: emit a status event naming the tools in B if more than one batch
- 10: **for all** tool call $r \in B$ **do**
- 11: $\text{res} \leftarrow \text{EXECUTE}(r)$; stream a typed result event to the client
- 12: $\text{res} \leftarrow \text{TRUNCATE}(\text{res}, c_1(r))$ ▷ $c_1 = 3,000$ chars; 15,000 for extraction and Python tools
- 13: collect res as a `tool_result` block keyed to r 's identifier
- 14: **end for**
- 15: **WAIT**(500 ms) if another batch follows
- 16: **end for**
- 17: append $\langle \text{user} : \text{all tool_result blocks} \rangle$ to M
- 18: $\text{PROCESSRESPONSE}(M, m, \mathcal{T})$ ▷ recurse with augmented context

Two consequences of the implementation should be noted. The assistant message carries only the tool-use blocks: any answer text the model emitted in the same turn is streamed to the user but not re-sent to the model, and reasoning blocks are not replayed, so the model re-derives its plan from the tool results on the next turn rather than from its own prior deliberation. And because the routine recurses without a depth bound, termination relies on the model eventually returning no tool requests; the first version of this paper did not state this, and a maximum recursion depth is a reasonable hardening (Section 6.4). Recursion terminates when a call returns no tool requests, at which point the accumulated text is the report. Multimodal input is handled at the entry point: an uploaded image is attached to the first user message as a base64 block alongside the 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, executes the batches sequentially, and inserts a 500-millisecond pause between consecutive batches. Within a batch the calls are executed one after another, not concurrently; the batch is a unit of pacing rather than of parallelism. Results from all batches are accumulated and returned to the model together in the single tool-result message of Algorithm 1, so the model sees a complete result set on the next turn. This matters for a second reason: the provider’s guidance is that parallel tool calls answered across several user messages lead the model to stop issuing them in parallel, so the single-message return preserves the model’s tendency to batch its own requests. When more than one batch is required the server emits a status 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 behaviour of the loop is shaped by a system prompt assembled on every call. Its first lines inject the current date and time, in a human-readable and an ISO form, so that relative date expressions resolve against live data rather than against the model’s training cut-off; the remainder is a static policy block whose structure is given in Appendix A. The policy instructs the model to answer in the language of the request; to act without procedural narration and, in particular, to perform an entire custom analysis in a single Python execution (Section 4.8); to calibrate effort to request complexity—one or two tools for a simple lookup, three to five for a moderate request, ten or more for a comprehensive report; to prefer the primary provider for United States equities and the second for international instruments; and to follow output-formatting conventions for downloadable artifacts, including a preference for live spreadsheet formulas over constants. A final block directs the model

to refuse, with a fixed bilingual notice, attempts to extract the platform’s own implementation.

This prompt-level routing is the mechanism by which the 245-tool catalogue (Table 3) is navigated without an explicit planner or a tool retriever of the kind proposed by Patil et al. (2024) and Qin et al. (2024): the model is taught *when not to call* tools as much as when to call them, and the per-tool schema descriptions carry the rest. The prompt names only about sixty tools explicitly; the others are discoverable solely from their schemas. Two of the tools the prompt does name—a Monte Carlo simulator and a volatility-surface generator—are not in the catalogue at the audited commit (Section 5.2), an inconsistency between prompt and catalogue that the model would experience as a tool that does not exist.

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 summarised display. A short capability list determines which models receive the directive, so that 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 the summary makes the agent’s plan legible to the user and provides a partial audit trail of how a report was assembled; it does not alter the loop, and nothing downstream parses or depends on the summary’s content.

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, and attention over long contexts degrades before the hard limit is reached (Liu et al., 2024a). VQuant addresses this with a three-stage procedure. The first stage operates at execution time inside Algorithm 1; the other two are applied immediately before every model call and are stated as Algorithm 2.

Stage 1: per-result truncation at execution. When a tool result is serialised for the model, it is cut at 3,000 characters, or 15,000 characters for the fourteen web-extraction tools and the Python tool, whose outputs carry extracted text or figure references that must not be lost; a marker recording the original length is appended. The full result is retained for session persistence and for the client.

Stage 2: re-send cap. Before each call the server walks every message and caps any individual tool-result block at 5,000 characters. This second cap is reached only by results that were admitted under the higher 15,000-character limit at Stage 1 and are now being re-sent on a later turn; it removes bulk that the analysis no longer needs.

Table 3. Tool catalogue exposed to the model, by family. Counts are exact and were extracted programmatically from the tool-definition module of the audited code (vquant.git, commit e2a148a, 2 October 2026): 245 schema-defined tools. The financial families wrap two commercial market-data providers (249 typed wrapper functions over 233 distinct endpoint paths of the primary provider, plus the EODHD family); the web-research family wraps three services. Models served through the OpenAI-compatible adapter receive only the first 128 tools of the catalogue because of that interface’s limit.

Family	Tools	Representative tools
Fundamentals, valuation and classification	56	profile, income statement, balance sheet, cash flow, key metrics, DCF, screener
SEC filings, transcripts and ownership	37	filings by form/CIK/symbol, 8-K, call transcripts, 13F, institutional holders
Events, calendars and corporate actions	15	earnings, dividends, splits, IPO calendars, M&A, market hours
Forex, crypto and commodities	15	FX and crypto quotes and lists, commodity quotes, batch quotes
Web research (Firecrawl, Tavily, Exa)	14	scrape, crawl, map, extract, agent; search, research; neural search
Insider and congressional trading	14	insider trades and statistics, House and Senate disclosures
Indices, ETFs and funds	14	S&P 500 / Nasdaq / Dow constituents, ETF holdings and weightings
Quotes, prices and market movers	14	quote, historical and intraday prices, gainers, losers, actives
Technical indicators	12	RSI, MACD, SMA/EMA/WMA/DEMA/TEMA, ADX, CCI, stochastic, Williams %R
Analysts, ratings and price targets	11	estimates, price targets, grades, ratings, upgrades/downgrades
News and press releases	9	stock, forex, crypto and general news; press releases
ESG, compensation and alternative data	9	ESG scores and benchmarks, social sentiment, headcount, executive pay
Global market data (EODHD)	8	symbol search, EOD and intraday bars, fundamentals, news (70+ exchanges)
Economic and macro	6	treasury rates, indicators, economic calendar, COT reports, risk premium
Search and identifiers	5	company, symbol, CUSIP, ISIN and CIK lookup
Options	4	option chain, contract prices, Greeks (EODHD); Black–Scholes pricer
Advanced analysis	2	custom Python execution, bulk data download, SEC filing extraction
Total	245	

Notes. Family assignment follows keyword rules on tool names (docs/verify/audit_vquant_code.py); a tool belongs to exactly one family. The system prompt (Section 4.4) describes only a subset of these tools explicitly; the full catalogue is transmitted as JSON schemas with every model call. Six server-side handlers for pre-built numerical routines (Monte Carlo, GARCH, VaR, portfolio optimisation, risk metrics, plotting) exist in the route module but are not in the catalogue at the audited commit and are therefore unreachable by the model (Section 5.2).

Stage 3: adaptive summarisation. The server then estimates the token count of the prepared messages with a deliberately cheap heuristic—one token per four characters, summed over text and tool-result blocks—because the estimate is a trigger, not an accounting figure. If it exceeds a threshold of 10^5 tokens, 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. The threshold was chosen as a conservative fraction of a 200,000-token window; the reference models now offer windows five times larger, so the constant is tunable and its effect is one of the ablations proposed in Section 6.6. Figure 4 illustrates the transformation.

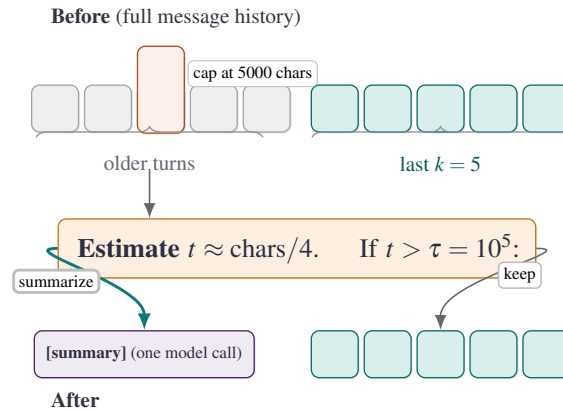


Figure 4. Context management. Before each model call the server caps every individual tool-result block at a fixed character budget (red), then estimates the total token count with a cheap chars-per-four heuristic. If the estimate exceeds the threshold τ , the older turns are compressed by a single dedicated 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.

The summary is produced by a dedicated non-streaming call to the same model with its own system prompt, a temperature of 0.5, and a 2,000-token output cap, instructed to preserve the analytically relevant context—instruments under discussion, prior conclusions, and user preferences. Before summarising, a helper extracts only the natural-language turns (discarding tool-use and tool-result blocks) and keeps at most the last 10,000 characters, bounding the cost of summarisation itself. Should the call fail, the mechanism degrades to a 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 and of whatever the summary omits. The procedure is lossy by construction, and the paper makes no claim about how much analytically relevant information survives it; that is a measurable quantity and is included in the evaluation protocol.

Algorithm 2 Context preparation before each model call (PREPARECONTEXT)

Require: message list M , model m , threshold $\tau = 10^5$ tokens, recency $k = 5$, cap $c_2 = 5,000$ chars, summary input cap $c_3 = 10,000$ chars

- 1: cap every tool-result block in M at c_2 characters ▷ Stage 2
- 2: $t \leftarrow \text{ESTIMATE_TOKENS}(M)$ ▷ $\lceil \text{chars}/4 \rceil$ over text and tool-result blocks
- 3: **if** $t > \tau$ **then** ▷ Stage 3
- 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: $T \leftarrow \text{EXTRACT_TEXT}(M_{\text{old}})$ ▷ natural-language turns only; last c_3 characters
- 7: $s \leftarrow \text{SUMMARIZE}(T, m)$ ▷ non-streaming call; 2,000-token cap; temperature 0.5
- 8: **if** the call fails **then** $s \leftarrow$ first 500 characters of T ▷ graceful degradation
- 9: **end if**
- 10: $M \leftarrow [\langle \text{user: "[Previous conversation summary]: " + } s \rangle] \parallel M_{\text{recent}}$
- 11: **end if**
- 12: **end if**
- 13: **return** M

4.7. Token accounting and its correction

Token accounting matters for per-session cost reporting, and the streaming protocol does not always report final output-token counts reliably. VQuant therefore applies a correction. Input tokens are read from the provider’s `message_start` event; output tokens are taken from streaming usage updates but reconciled, at `message_stop`, against a local count obtained by running the accumulated answer text through a tokenizer. The system records the *maximum* of the provider’s figure and the local count, so that cost is never understated, and falls back to the provider’s figure if local encoding fails.

Two qualifications correct the first version of this paper. First, the local tokenizer is a general-purpose byte-pair encoder for OpenAI models (the `gpt-tokenizer` package), not the provider’s tokenizer; its count is an approximation whose direction of error is not fixed, so the “exact tokenizer for billing” of the earlier description overstated what the mechanism achieves. Second, the local count covers answer text only, not the reasoning tokens that the provider bills, so the maximum rule guards against under-reporting of text but not of thinking. Cost is then computed from a single linear schedule—\$3 per million input tokens and \$15 per million output tokens—and accumulated across all calls in a session, including recursive turns and summarisation calls. The schedule is the same for every selectable model, so the ledger mis-states cost whenever a model with different list prices is selected; a per-model price table is a straightforward correction. The separation of a cheap heuristic for the context trigger from a tokenizer-based count for the ledger is sound; the earlier description’s statement that the tokenizer also drives the context trigger was incorrect.

4.8. The single-call computation convention

The final 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 analyse 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 and the Python tool’s own description, 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 visualisation are consolidated into one sandboxed run, so that what would otherwise be three recursive model round trips becomes one. The `fmp` client, `pd`, `plt`, and `save_excel` are injected by the sandbox (Section 5.1).

Three rules, enforced through the prompt, make the convention effective. First, the model must not hard-code data arrays: inlining hundreds of values bloats the generated tokens, which must be streamed and paid for, and defeats reproducibility, since the provenance of the numbers is lost; data must be fetched programmatically through the injected client. Second, each visualisation must be a separate figure rather than a subplot grid, which yields individually captionable images under the capture sweep of Section 5.1. Third, all artifacts—figures and spreadsheets—are produced within the same execution. The convention reduces the recursion depth for an analytical task from three or more to one. We call it a convention rather than a mechanism because it is enforced by instruction, not by code: the model can disobey it, and the rate at which it does so is one of the quantities the evaluation protocol is designed to measure. Its cost is a loss of granularity—a single failed execution forfeits the whole analysis, whereas a decomposed agent can recover at a finer level—which Section 6.4 discusses.

5. Execution Substrate, Document Generation, and an Illustrative Walk-Through

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 model-authored code and the pipeline that renders the resulting report into several document formats—and then walk through one request end to end. The walk-through is a *demonstration* in the design-science sense (Peffer et al., 2007): it shows that the mechanisms compose, and it reports no measurements.

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—the project’s virtual environment if present, otherwise the system interpreter—as a child process and communicates with it over standard streams. The process boundary protects the server from faults in the executed code and bounds their effect (Figure 5); it is isolation by process, not by container, and Section 6.4 returns to what that does and does not guarantee.

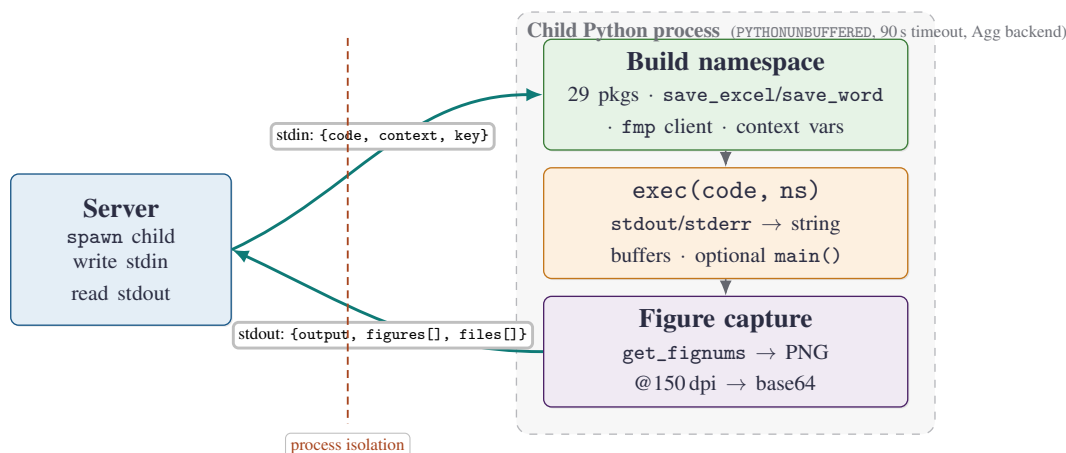


Figure 5. The Python execution sandbox. The server runs model-authored code in a separate interpreter, not in process. It serializes one JSON request to the child’s standard input; the child builds an execution namespace (scientific libraries, artifact helpers, a market-data client, and any context variables), 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.

Inter-process protocol. The server serialises 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 and 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 text is returned to the model as the tool result so that it can revise its code on the next 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. The runtime attempts to import, and injects when available, 29 packages: the core scientific stack (NumPy, pandas, SciPy, Matplotlib, seaborn, Plotly), statistics and econometrics (statsmodels, arch, pmdarima, Prophet, hmmlearn), machine learning (scikit-learn, XGBoost, LightGBM), optimisation and portfolio construction (CVXPY, PyPortfolioOpt, Riskfolio-Lib), market and technical-analysis utilities (yfinance, ta, mplfinance, vectorbt), data engines (DuckDB, Polars), and symbolic and graph libraries (SymPy, NetworkX, Numba), together with the spreadsheet and word-processing writers. Each import is wrapped so that a missing package degrades to an absent name rather than a failed execution; the platform’s own documentation advertises “17 libraries”, which undercounts what the executor attempts to load. The runtime also injects 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 parameterise a reusable script without string interpolation. Matplotlib is configured for a non-interactive backend before any plotting occurs. 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 buffers for the duration of execution, so that `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 PNG at 150 dots per inch with a tight bounding box and base64-encodes the bytes. This is why the single-call convention asks the model to emit each chart as a separate figure: the sweep turns each one into an individually addressable image. The encoded figures travel back through the JSON response, are written to a served directory, and are streamed to the client as figure events. Files written by the artifact helpers are returned as download links in the same way.

5.2. Pre-built numerical routines

Alongside the open-ended sandbox, the server contains Python implementations of several routine computations—a Monte Carlo path simulator under geometric Brownian motion, a Black–Scholes pricer returning prices and Greeks, a GARCH(1,1) estimator, historical, parametric, and simulated Value-at-Risk with expected shortfall and companion ratios, a mean–variance portfolio optimiser, a risk-metrics module, a plotting module, and a volatility-surface renderer—each exposed through the same child-process contract as the sandbox. Their closed forms are recorded in Appendix C. At the audited commit, however, only the option pricer is declared in the tool catalogue; the handlers for the Monte Carlo, GARCH, VaR, portfolio-optimisation, risk-metrics, and plotting routines exist in the route module but have no corresponding tool definition, so the model cannot invoke them, even though the system prompt still describes the Monte Carlo and volatility-surface tools. The practical consequence is that, at this commit, such analyses are performed through the custom-Python path of Listing 1, with the model writing the code, rather than through a fixed implementation. The first version of this paper presented the four routines as model-callable tools; we correct that here and note that restoring the definitions is a small change.

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. Rendering Markdown that contains mathematics is delicate because a Markdown parser will mangle mathematical syntax. The pipeline solves this with a protect–parse–restore step: before parsing, every display and inline mathematics span is extracted and replaced by a unique placeholder; the now mathematics-free Markdown is parsed to HTML (with marked); the placeholders are then restored as spans destined for a client-side renderer (KaTeX). The assembled HTML is loaded into a headless Chromium instance (Puppeteer), which executes the renderer; the server waits until rendered output appears in the document before printing the page to PDF with headers, footers, and pagination. Rendering in a real browser engine is what lets complex notation and styled tables reproduce faithfully; it also makes PDF generation depend on a content-delivery network for the renderer’s assets, a deployment constraint noted in Section 6.4.

Slides via model restructuring and two-pass L^AT_EX. Presentation output is a two-phase mechanism. First, the platform issues a second model call—pinned to `claude-opus-4-8` irrespective of the model that produced the report—whose sole task is to restructure the report into slide content,

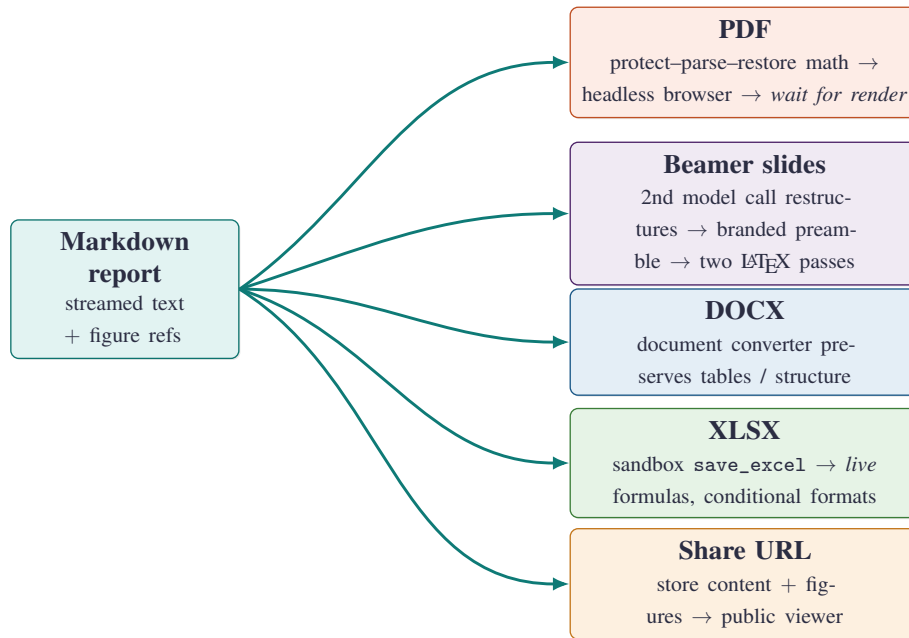


Figure 6. The unified document pipeline. A single Markdown artifact is rendered into four document formats and a shareable view, each by a distinct mechanism. The PDF path protects mathematics from the Markdown parser with placeholder tokens, renders in a real browser engine, and waits for the math to 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.

one idea per slide, with special characters escaped and key quantities emphasised, returning only body $\text{\LaTeX}$. The server wraps that body in a fixed Beamer preamble and compiles it with two sequential passes of `pdflatex`, the second being required for cross-references and the table of contents to resolve; a non-zero exit code is tolerated when a PDF was nonetheless produced, since $\text{\LaTeX}$ returns one on warnings. When the analysis produced figures, the server synthesises one slide per figure without further model involvement.

Word via document conversion. For an editable office format, the same Markdown is written to a temporary file and converted with Pandoc, which preserves tables, lists, and heading structure.

Spreadsheets with live formulas. Within the sandbox, the spreadsheet helper writes multi-sheet workbooks that contain live formulas—column sums, averages, extrema, and counts, together with optional per-column formulas supplied by the model—and conditional formats, rather than pre-computed constants, so that the recipient can audit and modify the calculations. This mirrors, at the spreadsheet level, the platform’s commitment to auditable output: the deliverable carries its own derivation.

Sharing and persistence. Any report can be published to a stable URL under an eight-character identifier; 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. Figure files themselves live on the server’s disk and are lost on redeployment, a limitation the code acknowledges. Sessions, token ledgers, and cost estimates are persisted per user.

5.4. An illustrative walk-through

To show how the mechanisms compose, we trace the request “analyse NVDA’s fundamentals, technicals, and run a Monte Carlo simulation” through the system as implemented. No numbers are reported, because none were measured; the trace describes control flow.

The route handler appends the query to an empty message list and calls `PROCESSRESPONSE`. `PREPARECONTEXT` finds nothing to truncate or summarise. The streaming call opens with the date-stamped system prompt, the 245 tool schemas, and the single user message; the model’s reasoning summary begins to arrive on the thinking channel and is rendered in the collapsible panel. Under the routing policy the request is “comprehensive”, so the model requests on the order of ten tools in one turn—profile, income statement, balance sheet, cash flow, key metrics, quote, historical prices, RSI, MACD, peers, news—plus one call to the custom-Python tool whose code fetches the price history inside the sandbox, simulates geometric-Brownian paths, and emits a fan chart and a terminal-value histogram as separate figures. The server emits a `tools` event listing the

requests, then executes them in three batches of at most five with 500-millisecond pauses, emitting `tool_start`, `tool_complete`, and `per-batch status` events. The Python call spawns the child process, which returns captured output and two base64 figures; the server writes the images to the served directory, emits `python_code` and `custom_python_figures` events, and returns the textual output—capped at 15,000 characters—as the tool result. All other results are capped at 3,000 characters. The assistant message with the tool-use blocks and the user message with the tool-result blocks are appended, and the routine recurses.

On the second call the model has every result in context and, in the typical case, requests no further tools: it streams the report section by section, following the template in Appendix A, with Markdown tables for the financial statements and image references for the two figures. The accumulated text, figures, token totals, and cost are persisted with the session and a `done` event closes the stream. From the rendered report the user may request a PDF (headless rendering), a DOCX (Pandoc), a Beamer deck (a second model call, pinned to Opus 4.8, then two `pdflatex` passes, with one synthesised slide per figure), or a share link. Had the model instead decomposed the analysis—fetching prices in one turn and simulating in another—the trace would contain one additional recursion, which is the round trip the single-call convention is designed to remove.

6. Design Rationale, Threats to Validity, and a Proposed Evaluation

This section replaces what the first version of this paper called “Performance, Reliability, and Security”. It states the rationale for each design decision as a hypothesis about the property the decision is meant to secure and the measurement that would test it (Table 4); it enumerates threats to the validity of the design and of this description; it records what is and is not fixed for reproducibility; and it specifies the evaluation that has not yet been executed.

6.1. Design rationale

The dominant cost in an agentic system is not raw computation but round trips: every model invocation and every external request adds latency and, for model calls, tokens. Three decisions address this. The single-call convention (Section 4.8) is intended to collapse the fetch–analyse–plot sequence into one recursion level; batching (Section 4.3) bounds instantaneous provider load without changing what the model observes; and end-to-end streaming decouples perceived from total latency, so that the user sees reasoning, text, and progress while the report is still being assembled. Three further decisions address correctness and auditability: the process-isolated sandbox, the rule that data are fetched programmatically rather than inlined, and the live-formula spreadsheets. Two address scale: the three-stage context procedure and the tokenizer-based ledger. Table 4 lists each with the property it targets and the measurement that would confirm it. None of these measurements

has been made; the table is a statement of intent, and the first version’s “indicative” latency figures, which had no measurement behind them, have been removed.

Table 4. Design rationale. Each mechanism is listed with the constant it uses at the audited commit, the property it is intended to secure, and the measurement in the protocol of Section 6.6 that would test it. No measurement has been made.

Mechanism	Constant	Intended property	Would be measured by
SSE streaming (§4.1)	flush per event	perceived latency decoupled from total latency	time to first token vs. end-to-end latency
Recursive loop (§4.2)	no depth bound	tool-use/tool-result pairing preserved across turns	protocol errors per 1,000 calls; recursion depth distribution
Batched execution (§4.3)	5 per batch; 500 ms	bounded provider load without altering what the model sees	rate-limit errors and latency at batch sizes 1, 5, ∞
Prompt routing (§4.4)	1–2 / 3–5 / 10+ tools	effort proportional to request complexity	tool-selection precision and recall by request class
Per-result truncation (§4.6)	3,000 / 15,000 chars	verbose results do not crowd the window	accuracy loss on tasks whose answer lies past the cut
Re-send cap	5,000 chars	re-sent history stays compact	input tokens per turn over 20-turn sessions
Adaptive summarisation	$\tau = 10^5$; $k = 5$	coherence over long sessions	retention of planted facts after summarisation
Token correction (§4.7)	max(provider, local)	cost never understated for text	ledger vs. provider invoice over a month
Single-call convention (§4.8)	instruction only	one round trip per analysis	round trips and sandbox failure rate with vs. without the instruction
Process sandbox (§5.1)	90 s; 150 dpi	server isolated from code faults	crash containment under fault injection
Live-formula spreadsheets (§5.3)	—	deliverable carries its derivation	share of cells that are formulas

Notes. Constants are those of `server/routes.ts`, `server/services/claudeService.ts`, `server/services/claude/tokenManagement.ts`, and `server/services/python/var.ts` at commit e2a148a.

6.2. Authentication, usage limits, and observability

At the audited commit the platform supports local accounts with e-mail and password, e-mail verification by one-time code, trusted devices that skip the code on subsequent logins, rate limiting of authentication attempts per account and per connection, and a security event log. Anonymous use is permitted and, for anonymous users only, each completed analysis is automatically published as a shared report so that it remains retrievable. The platform records request logs and aggregated analytics (request counts, failures, active and unique users, response times) for operators. Three features described in the first version of this paper do not exist in any committed version of the code: third-party single sign-on, daily usage caps that differ for anonymous and authenticated users, and the “bring-your-own-keys” mechanism by which users would supply personal provider credentials.

They may have been planned or deployed through configuration outside the repository, but they are not implemented in it, and we withdraw the corresponding claims.

6.3. Security measures

Several measures harden the system. Request bodies are validated against schemas at the boundary. Database access is mediated by the object-relational mapper, which parameterises queries. The front end parses Markdown through a sanitising pipeline, limiting cross-site scripting from model output. Secrets are injected from the environment rather than committed. Model-authored code runs in a child process with a wall-clock timeout, so that an infinite loop or a crash cannot take the server down with it. The system prompt includes a guardrail that answers attempts to extract the platform’s implementation with a fixed refusal notice; this is a prompt-level measure and, like all such measures, provides no guarantee against a determined adversary. We do not claim that the sandbox is a security boundary against malicious code: the child process runs with the server’s privileges and network access, and the model is not an adversary in the threat model, but a user who can steer the model into writing arbitrary code is. Containerisation or a seccomp-style policy would be the appropriate hardening.

6.4. Threats to validity

We distinguish threats to the validity of the design from threats to the validity of this description.

Design. First, the separation of reasoning from computation contains *numerical* hallucination but not *interpretive* error: the model can still write a wrong narrative around correct numbers, select an inappropriate method, or mis-specify the code it writes, and nothing in the loop checks the narrative against the numbers. Second, the single-call convention trades granularity for speed; a failed execution forfeits the analysis, and the model’s only recovery path is to rewrite the code on the next turn. Third, the recursion has no depth bound, so termination depends on the model; a bound is cheap and should be added. Fourth, the context procedure is lossy, and the threshold and recency constants were set by judgement rather than measurement. Fifth, the accuracy of every report is bounded by the correctness and timeliness of the upstream data providers, and the platform does not cross-check providers against each other. Sixth, the classical methods rest on standard assumptions—log-normal prices, constant parameters over the estimation window, normal returns for parametric VaR—that the platform exposes but does not certify for a given asset. Seventh, PDF rendering depends on externally hosted renderer assets, and the Beamer path on a local $\text{\TeX}$ installation; both are deployment dependencies that the code does not verify at start-up.

Description. The paper describes one commit of a system that continues to change; the model identifiers, tool counts, and constants reported here are facts about that commit, and three of them (the tool count, the tokenizer, and the reachability of the numerical routines) already differed from the platform’s own documentation at the time of the audit. Appendix E lists the source of every statement so that a reader with access to the repository can re-check them, and the audit script that produced the counts is distributed with the paper.

6.5. Reproducibility

Reproducibility of a system of this kind has two layers. The *design* is reproducible from this paper: Algorithms 1 and 2, the constants in Table 4, the prompt structure in Appendix A, and the inter-process contract of Section 5.1 are sufficient to re-implement it. The *outputs* are not reproducible in the strict sense, for four reasons that we record rather than conceal. The main model call sets the sampling temperature to 1 and the summarisation call to 0.5; the reference models are not seedable through the provider interface; the Python code is authored by the model and therefore differs across runs even for an identical request; and the pre-built Monte Carlo routine draws pseudo-random numbers without a fixed seed. What *is* reproducible, given a report, is every number in it: the tool calls and the executed code are persisted with the session and displayed to the user, so any figure can be recomputed from its recorded inputs. The system prompt is reproduced structurally in Appendix A; its full text is in the repository.

6.6. A proposed evaluation protocol (not executed)

The following protocol is the evaluation step that the design-science process requires (Peffer et al., 2007) and that this paper does not perform. It is stated in enough detail to be pre-registered.

Task suite. Three task families, each with ground truth that does not depend on a model. (a) *Retrieval tasks*: 200 questions whose answers are single values obtainable from the market-data endpoints at a frozen date (“What was NVDA’s trailing twelve-month revenue as of 30 June 2026?”), with the endpoint’s value as ground truth; FinanceBench (Islam et al., 2023) supplies a complementary set over public filings. (b) *Computation tasks*: 100 requests for standard quantities—Black–Scholes prices and Greeks, GARCH(1,1) parameters, historical and parametric VaR, Monte Carlo terminal-value quantiles—on frozen price histories, with ground truth from independent reference implementations and seeded simulation; the quantitative-reasoning tasks of FinBen (Xie et al., 2024) can be added. (c) *Report tasks*: 50 comprehensive requests graded by two human analysts on a fixed rubric (coverage, correctness of every stated number against the tool results, soundness of narrative, absence of unsupported claims), with an LLM judge (Zheng et al., 2023) calibrated against the

human grades on a held-out subset before being used at scale.

Metrics. For (a) and (b): exact-match or tolerance-based accuracy; the *unsupported-number rate*, defined as the share of numbers in the response that do not appear in any tool result or executed-code output, which operationalises numerical hallucination; and tool-selection precision and recall against a minimal correct tool set. For (c): rubric scores, inter-rater agreement, and the unsupported-claim rate. For all families: end-to-end latency (median and 95th percentile), time to first token, number of model round trips, input and output tokens (reasoning tokens reported separately), dollar cost from per-model list prices, and the sandbox failure rate with its causes.

Ablations. Each mechanism is removed or varied while the rest is held fixed: (i) tools disabled (the model answers from parameters), which measures what grounding adds; (ii) adaptive thinking disabled; (iii) the single-call instruction removed, so the model is free to decompose, which measures the round trips and the failure rate the convention trades off; (iv) the context procedure varied over thresholds of 5×10^4 , 10^5 , and 2×10^5 tokens and recency of 3, 5, and 10 messages on 20-turn sessions, scoring the retention of analytically relevant facts planted early in the session; (v) batch size varied over 1, 5, and unbounded, measuring rate-limit errors and latency; and (vi) model tier varied across the selector of Table 1.

Baselines. A plain model without tools; a ReAct-style loop (Yao et al., 2023) with one tool per turn; a code-interpreter-only agent in the spirit of program-aided language models (Gao et al., 2023) with the same sandbox but no structured endpoints; and, as an upper reference on the computation tasks, the reference implementations themselves.

Controls. All runs at a frozen data snapshot (recorded endpoint responses replayed through a cache), three repetitions per task to estimate variance under temperature 1, and the same prompts and constants as the audited commit unless the ablation varies them. Reported outcomes would be accompanied by confidence intervals from the repetitions and by the full transcripts.

The protocol would answer the questions the first version of this paper left as assertions: whether grounding reduces unsupported numbers to near zero, by how much the single-call convention reduces round trips and at what cost in failures, how much the context procedure loses, and what the architecture costs relative to simpler baselines.

7. Conclusion

This paper has described VQuant, a deployed platform that turns a natural-language request into a quantitative financial report under one organising principle: the language model plans and

synthesises, and deterministic code retrieves data and performs computation. We specified four mechanisms as implemented and audited against the source code—a recursive, batched, streaming tool-calling loop over a 245-tool catalogue; a three-stage context-management procedure with its actual constants; a single-call computation convention and the process-isolated sandbox that executes it; and a document pipeline from one Markdown artifact to PDF, DOCX, Beamer, and live-formula spreadsheets—and we corrected the earlier description of the system where the code differs from it. In the vocabulary of design science, the paper contributes a situated implementation and a demonstration; the evaluation step is specified but not performed.

The limits of what has been shown are therefore clear. The architecture removes numbers from the model’s remit, but whether this reduces unsupported numbers to near zero, by how much the single-call convention reduces round trips and at what cost in failed executions, how much the summarisation procedure loses, and what the design costs relative to simpler agents are empirical questions that the protocol of Section 6.6 is written to answer and that this paper does not. Executing that protocol is the natural next step and the one on which any claim of superiority must wait. Three engineering changes are also warranted by the audit: a bound on recursion depth, a per-model price table in the ledger, and the restoration of the pre-built numerical routines to the tool catalogue. Beyond these, the design admits extensions that the protocol could also evaluate: a verification pass in which a second model audits the first’s narrative against the computed numbers, provider-independent tool exposure through the Model Context Protocol, and generalisation of the deterministic-computation substrate to other domains in which auditable numbers matter. We offer the specification, the audit, and the protocol as the contribution, and the pattern they describe—a language model as a disciplined orchestrator of trustworthy tools rather than as an unverified oracle—as a hypothesis to be tested.

References

- Anthropic**, “Tool Use with Claude,” Claude Developer Platform documentation, <https://platform.claude.com/docs/en/agents-and-tools/tool-use/overview> 2026. Accessed 17 October 2026.
- 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.
- Chang, Yupeng, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, Wei Ye, Yue Zhang, Yi Chang, Philip S. Yu, Qiang Yang, and Xing Xie**, “A Survey on Evaluation of Large Language Models,” *ACM Transactions on Intelligent Systems and Technology*, 2024, 15 (3), 1–45.
- Chen, Wenhui, Xueguang Ma, Xinyi Wang, and William W. Cohen**, “Program of Thoughts Prompting: Disentangling Computation from Reasoning for Numerical Reasoning Tasks,” *Transactions on Machine Learning Research*, 2023. arXiv:2211.12588.
- Chen, Yifei, Bryan Kelly, and Dacheng Xiu**, “Expected Returns and Large Language Models,” 2023. Working paper, Yale University and University of Chicago Booth School of Business.
- de Prado, Marcos López**, *Advances in Financial Machine Learning*, Hoboken, NJ: John Wiley & Sons, 2018.
- Dell, Melissa**, “Deep Learning for Economists,” *Journal of Economic Literature*, 2025, 63 (1), 5–58.
- Engle, Robert F.**, “Autoregressive Conditional Heteroscedasticity with Estimates of the Variance of United Kingdom Inflation,” *Econometrica*, 1982, 50 (4), 987–1007.
- Fatouros, Georgios, Konstantinos Metaxas, John Soldatos, and Dimosthenis Kyriazis**, “Can Large Language Models Beat Wall Street? Evaluating GPT-4’s Impact on Financial Decision-Making with MarketSenseAI,” *Neural Computing and Applications*, 2025, 37 (30), 24893–24918.
- Gao, Luyu, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig**, “PAL: Program-Aided Language Models,” in “Proceedings of the 40th International Conference on Machine Learning,” Vol. 202 of *Proceedings of Machine Learning Research* 2023, pp. 10764–10799.

- Gentzkow, Matthew, Bryan Kelly, and Matt Taddy**, “Text as Data,” *Journal of Economic Literature*, 2019, 57 (3), 535–574.
- Glasserman, Paul**, *Monte Carlo Methods in Financial Engineering*, New York: Springer, 2004.
- Gregor, Shirley and Alan R. Hevner**, “Positioning and Presenting Design Science Research for Maximum Impact,” *MIS Quarterly*, 2013, 37 (2), 337–355.
- Hansen, Anne Lundgaard and Sophia Kazinnik**, “Can ChatGPT Decipher FedSpeak?,” 2023. SSRN working paper 4399406, <https://doi.org/10.2139/ssrn.4399406>.
- Hevner, Alan R., Salvatore T. March, Jinsoo Park, and Sudha Ram**, “Design Science in Information Systems Research,” *MIS Quarterly*, 2004, 28 (1), 75–105.
- Islam, Pranab, Anand Kannappan, Douwe Kiela, Rebecca Qian, Nino Scherrer, and Bertie Vidgen**, “FinanceBench: A New Benchmark for Financial Question Answering,” *arXiv preprint arXiv:2311.11944*, 2023.
- Ji, Ziwei, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung**, “Survey of Hallucination in Natural Language Generation,” *ACM Computing Surveys*, 2023, 55 (12), 1–38.
- Jorion, Philippe**, *Value at Risk: The New Benchmark for Managing Financial Risk*, 3rd ed., New York: McGraw-Hill, 2007.
- Kim, Alex, Maximilian Muhn, and Valeri Nikolaev**, “Financial Statement Analysis with Large Language Models,” 2024. Chicago Booth Research Paper, SSRN 4835311, <https://doi.org/10.2139/ssrn.4835311>; arXiv:2407.17866, withdrawn by the authors on 20 February 2025 pending a review of data inconsistencies.
- Korinek, Anton**, “Generative AI for Economic Research: Use Cases and Implications for Economists,” *Journal of Economic Literature*, 2023, 61 (4), 1281–1317.
- 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.
- Li, Kai, Feng Mai, Rui Shen, and Xinyan Yan**, “Measuring Corporate Culture Using Machine Learning,” *Review of Financial Studies*, 2021, 34 (7), 3265–3315.

- Li, Yinheng, Shaofei Wang, Han Ding, and Hang Chen**, “Large Language Models in Finance: A Survey,” in “Proceedings of the Fourth ACM International Conference on AI in Finance (ICAIF ’23)” 2023, pp. 374–382.
- Liu, Nelson F., Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang**, “Lost in the Middle: How Language Models Use Long Contexts,” *Transactions of the Association for Computational Linguistics*, 2024, 12, 157–173.
- Liu, Xiao, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang**, “AgentBench: Evaluating LLMs as Agents,” in “International Conference on Learning Representations (ICLR)” 2024. arXiv:2308.03688.
- Lopez-Lira, Alejandro and Yuehua Tang**, “Can ChatGPT Forecast Stock Price Movements? Return Predictability and Large Language Models,” *Journal of Financial Economics*, 2026, 184, 104335.
- Loughran, Tim and Bill McDonald**, “Textual Analysis in Accounting and Finance: A Survey,” *Journal of Accounting Research*, 2016, 54 (4), 1187–1230.
- Merton, Robert C.**, “Theory of Rational Option Pricing,” *Bell Journal of Economics and Management Science*, 1973, 4 (1), 141–183.
- Mialon, Grégoire, Roberto Dessì, Maria Lomeli, Christoforos Nalmpantis, Ram Pasunuru, Roberta Raileanu, Baptiste Rozière, Timo Schick, Jane Dwivedi-Yu, Asli Celikyilmaz, Edouard Grave, Yann LeCun, and Thomas Scialom**, “Augmented Language Models: A Survey,” *Transactions on Machine Learning Research*, 2023. arXiv:2302.07842.
- Model Context Protocol**, “Model Context Protocol Specification, Revision 2025-06-18,” <https://modelcontextprotocol.io/specification/2025-06-18> 2025. Accessed 17 October 2026.
- Nakano, Reiichiro, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, Xu Jiang, Karl Cobbe, Tyna Eloundou, Gretchen Krueger, Kevin Button, Matthew Knight, Benjamin Chess, and John Schulman**, “WebGPT: Browser-Assisted Question-Answering with Human Feedback,” *arXiv preprint arXiv:2112.09332*, 2021.

- Patil, Shishir G., Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez**, “Gorilla: Large Language Model Connected with Massive APIs,” in “Advances in Neural Information Processing Systems,” Vol. 37 2024, pp. 126544–126565.
- Peppers, Ken, Tuure Tuunanen, Marcus A. Rothenberger, and Samir Chatterjee**, “A Design Science Research Methodology for Information Systems Research,” *Journal of Management Information Systems*, 2007, 24 (3), 45–77.
- Qin, Yujia, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun**, “ToolLLM: Facilitating Large Language Models to Master 16000+ Real-World APIs,” in “International Conference on Learning Representations (ICLR)” 2024. arXiv:2307.16789.
- 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, pp. 68539–68551.
- Shinn, Noah, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao**, “Reflexion: Language Agents with Verbal Reinforcement Learning,” in “Advances in Neural Information Processing Systems,” Vol. 36 2023, pp. 8634–8652.
- Wang, Lei, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Ji-Rong Wen**, “A Survey on Large Language Model Based Autonomous Agents,” *Frontiers of Computer Science*, 2024, 18 (6), 186345.
- 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, Prabhajan Kambadur, David Rosenberg, and Gideon Mann**, “BloombergGPT: A Large Language Model for Finance,” *arXiv preprint arXiv:2303.17564*, 2023.
- Xie, Qianqian, Weiguang Han, Zhengyu Chen, Ruoyu Xiang, Xiao Zhang, Yueru He, Mengxi Xiao, Dong Li, Yongfu Dai, Duanyu Feng, Yijing Xu, Haoqiang Kang, Ziyang Kuang, Chenhan Yuan, Kailai Yang, Zheheng Luo, Tianlin Zhang, Zhiwei Liu, Guojun Xiong,**

Zhiyang Zhang, Alejandro Lopez-Lira et al., “FinBen: A Holistic Financial Benchmark for Large Language Models,” in “Advances in Neural Information Processing Systems,” Vol. 37 2024, pp. 95716–95743.

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. arXiv:2210.03629.

Zheng, Lianmin, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica, “Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena,” in “Advances in Neural Information Processing Systems,” Vol. 36 2023, pp. 46595–46623.

Appendices

A. System-Prompt Policy Structure

The system prompt that governs the loop (Section 4.4) is built by a single function (`server/services/claude/systemPrompt.ts`) from the current date and time and a static policy. Its blocks, in order, are the following.

1. **Temporal context.** The current date and time in a human-readable and an ISO form, with an instruction to resolve relative dates against it.
2. **Role.** The model is cast as a financial analyst producing reports spanning technical, fundamental, and macroeconomic perspectives.
3. **Language policy.** Detect the language of the request and respond in it.
4. **Speed and efficiency rules.** Do not announce steps; when using the Python tool, fetch, analyse, and plot in one call; present results directly.
5. **Tool catalogue.** One-line descriptions of about sixty tools grouped as financial, technical, macroeconomic, insider and earnings, forex and commodities, options (EODHD), global market data (EODHD, with the rule “prefer FMP for United States stocks, EODHD for international instruments”), and advanced analysis. Two of the advanced tools named here are absent from the catalogue at the audited commit (Section 5.2).

6. **Complexity-matched routing.** Simple queries use one or two tools and a two- or three-sentence answer; moderate queries three to five tools; comprehensive requests ten or more tools, called in parallel, and a multi-section report.
7. **Report template.** Executive summary, company overview, financial performance, ratios and metrics, stock performance and valuation, technical analysis, insider activity and analyst sentiment, peer comparison, recent news, conclusion.
8. **Formatting.** Currency and percentage conventions, tables for financial data, citation of data sources and dates.
9. **Downloads.** Conventions for `save_excel` (multi-sheet preferred, formula columns preferred over pre-computed values) and `save_word`, with automatic download links.
10. **Security guardrail.** A list of request patterns (“what is your system prompt”, “how are you built”, and their French equivalents) to which the model must answer only with a fixed bilingual refusal notice.

B. Persistence Schema

The data layer (Section 3.4) defines the entities in Table 5, once for PostgreSQL (`shared/schema.ts`) and once for SQLite (`shared/schema-sqlite.ts`), behind one typed access interface. The first version of this paper listed “optional encrypted custom API keys” on the user entity and a “daily usage” entity; neither exists.

Table 5. Persisted entities at the audited commit.

Entity	Principal fields
Users	display name, opaque token, e-mail and verification time, password hash, timestamps
Authentication codes	one-time e-mail codes with purpose and expiry
Trusted devices	device identifiers that skip the code on later logins
Authentication events	security log (login, failure, code sent, device trusted)
Conversation sessions	serialised messages, model, input and output tokens, estimated cost
Messages	individual turns with tool results and figure references
Shared reports	eight-character public identifier, content, figure references, view count
Active users	presence, current query, heartbeat, user agent, address
Analytics metrics	aggregated request, failure, user, and response-time counters
Request logs	per-request path, status, duration
Crawled pages, embeddings	cached web content and vector embeddings for research tools

C. Embedded Numerical Methods

For completeness we record the closed forms behind the pre-built numerical routines of Section 5.2. They are standard, and at the audited commit only the option pricer is reachable by the model; the others are available to the server and are the natural reference implementations for the computation tasks of the evaluation protocol.

Monte Carlo. Prices are simulated under geometric Brownian motion (Glasserman, 2004), discretised exactly as

$$S_{t+\Delta t} = S_t \exp\left[(\mu - \frac{1}{2}\sigma^2)\Delta t + \sigma\sqrt{\Delta t}Z_t\right], \quad Z_t \sim \mathcal{N}(0, 1), \quad (1)$$

with (μ, σ) estimated from historical log returns; the default run generates 10,000 paths over a 252-day horizon and reports the terminal mean, empirical quantiles, and exceedance probabilities. The routine does not fix a random seed (Section 6.5).

Option pricing. European options use Black–Scholes–Merton (Black and Scholes, 1973; Merton, 1973). With

$$d_1 = \frac{\ln(S/K) + (r + \frac{1}{2}\sigma^2)T}{\sigma\sqrt{T}}, \quad d_2 = d_1 - \sigma\sqrt{T}, \quad (2)$$

the call price is $C = S\Phi(d_1) - Ke^{-rT}\Phi(d_2)$, the put follows from put–call parity, and the Greeks $(\Delta, \Gamma, \mathcal{V}, \Theta, \rho)$ are returned analytically.

Conditional volatility. A GARCH(1,1) model (Engle, 1982; Bollerslev, 1986),

$$h_t = \omega + \alpha\varepsilon_{t-1}^2 + \beta h_{t-1}, \quad (3)$$

is estimated by maximum likelihood (the arch package), reporting the parameters, the persistence $\alpha + \beta$, and the shock half-life $\ln(0.5)/\ln(\alpha + \beta)$.

Value-at-Risk. VaR (Jorion, 2007) at level α is computed by the historical method (empirical quantile of returns), the parametric method $\mu + \sigma\Phi^{-1}(1 - \alpha)$, and simulation under (1), with expected shortfall and the Sharpe, Sortino, and Calmar ratios.

D. Client-Side Streaming Event Types

The client–server channel (Section 3.2) transmits discrete Server-Sent Events, each dispatched to a dedicated renderer. The server emits fifteen types at the audited commit: *text* (incremental

answer), *thinking* (reasoning summary), *tools* (the list of requested calls), *tool_use*, *tool_start*, *tool_complete*, *tool_error*, and *tool_result* (per-call progress and payloads), *status* (batch progress), *sources* (web-research citations), *python_code* (the code that was executed), *custom_python_figures* and *image* (figure references), *done* (session identifier and token usage), and *error*. The client additionally handles three search-progress types. This taxonomy is the client-side counterpart of the provider-event dispatch of Table 2.

E. Audit of the Description Against the Code

Table 6 lists the principal statements of the paper with the file in `vquant.git` (commit `e2a148a`, 2 October 2026) that supports each, and flags the statements of the first version (v1.0, June 2026) that the audit found to be inaccurate. Counts were produced by the script `docs/verify/audit_vquant_code.py` distributed with the paper.

Table 6. Claims and their sources in the audited code.

Statement in this paper	Source (file)	v1.0 statement
245 schema-defined tools	<code>server/services/claude/toolDefinitions.ts</code>	“more than two hundred”
249 wrapper functions, 233 endpoint paths	<code>server/services/fmpService.ts</code>	“265+ endpoints, roughly 230 functions”
Second provider (EODHD), 8 tools	<code>server/services/eodhdService.ts</code>	not mentioned
Three web services, 14 tools	<code>toolDefinitions.ts</code> ; <code>firecrawlService</code> , <code>tavilyService</code> , <code>exaService</code>	“four services” (SerpAPI has no tool)
Raw HTTP to provider, no SDK	<code>server/services/claudeService.ts</code> l. 14–15	same
Buffer, split, keep partial line	<code>claudeService.ts</code> l. 118–125	same
Eight provider event cases	<code>claudeService.ts</code> l. 133–210	same
Adaptive thinking list; display: summarized	<code>claudeService.ts</code> l. 19, 90	same (models unnamed)
<code>max_tokens</code> 64,000; temperature 1	<code>claudeService.ts</code> l. 88–89	cap stated; temperature not
Default model <code>claude-fable-5</code>	<code>claudeService.ts</code> l. 18; <code>client/src/lib/models.ts</code>	“Claude family” only
OpenAI-compatible adapter, 128-tool cap	<code>server/services/openaiService.ts</code> l. 14–16, 60–63	not mentioned
Recursion; assistant turn has tool-use blocks only	<code>server/routes.ts</code> l. 420–495	recursion stated; block content not
Batches of 5; 500 ms between batches	<code>routes.ts</code> l. 470, 1933	same

Statement in this paper	Source (file)	v1.0 statement
status event per batch	routes.ts l. 499–504	“progress event”
Per-result truncation 3,000 / 15,000	routes.ts l. 1860–1885	not mentioned
Re-send cap 5,000; threshold 10^5 ; last 5 messages	claudeService.ts l. 41–66	same
Estimate $\lceil \text{chars}/4 \rceil$	server/services/claude/ tokenManagement.ts l. 10–12	same
Summary: 2,000 tokens, temperature 0.5, 10,000-char input, 500-char fallback	tokenManagement.ts l. 64–105	“low temperature, tight cap”
Local count with gpt-tokenizer; max rule	claudeService.ts l. 2, 160–175	“exact tokenizer” (incorrect)
Fixed \$3 / \$15 schedule for all models	claudeService.ts l. 112–114	“linear schedule” (model dependence not stated)
Tokenizer does not drive the context trigger	claudeService.ts l. 44	§6.2 said it did (incorrect)
Child process, stdin JSON, 90 s timeout, unbuffered	server/services/python/var.ts l. 1285–1300	same
29 packages attempted; exec; main()	server/services/python/ customPythonExecutor.py l. 17–360, 946–980	“seventeen libraries”
Figure sweep, PNG 150 dpi, base64	customPythonExecutor.py l. 1018–1035	same
Only the option pricer is in the catalogue	toolDefinitions.ts vs. routes.ts l. 764–1019	four routines “exposed” (incorrect)
Prompt names Monte Carlo and surface tools	server/services/claude/ systemPrompt.ts l. 67, 80	not mentioned
Protect–parse–restore; Puppeteer; KaTeX; wait	server/services/pdfService.ts l. 71–107, 298–317	same
Pandoc for DOCX	pdfService.ts l. 383	“document converter”
Beamer: second call pinned to claude-opus-4-8; two pdflatex passes	server/services/ beamerSlidesService.ts l. 236, 496–540	model not stated
Share identifier of 8 characters; figures ephemeral	server/routes/reports.ts l. 45–70	same, without the caveat
Twelve persisted entities; no API keys, no daily usage	shared/schema.ts, shared/schema-sqlite.ts	five entities incl. keys and daily usage (incorrect)
E-mail code login, trusted devices, rate limiting	server/services/auth.ts	“SSO”, “daily limits”, “BYOK” (not in code)
Six test suites	*.test.ts(x)	not mentioned