SynapseIDE-X: A Deterministic, Stream-Aware AI Coding Workspace

System Architecture and Formalization of Provider-Agnostic Streaming (SSE/JSONL),
Token-Constrained Context Composition, and Editor-Safe Apply-Plan Execution[*]

Mustafa Raşit Şahin, Ph.D.

Draft Whitepaper — September 2025

View on GitHub: Synapse_IDE

Abstract

We present **SynapseIDE-X**, a browser-based AI coding workspace that integrates a deterministic chat state machine, a resilient streaming engine, formal context budgeting for prompts, virtualized message rendering, and an editor bridge for safe code insertion. The contribution is grounded exclusively in the implemented application modules, organized around three formalizations that capture the system's observable behavior:

(i) Deterministic Conversation FSM. We model the chat controller as a deterministic finite-state machine

$$\mathcal{M} = (S, \Sigma, \delta, s_0, F), \qquad S = \{\texttt{idle}, \texttt{sending}, \texttt{streaming}\}, \quad F = \{\texttt{idle}\}.$$

The alphabet is rendered as a display to avoid line overflow:

$$\Sigma = \{\texttt{SEND}, \texttt{OPEN}, \texttt{DELTA}, \texttt{USAGE}, \texttt{DONE}, \texttt{ERROR}, \texttt{ABORT}\}.$$

Transitions are typeset in an aligned block to respect A4 margins:

$$\begin{aligned}
\delta(\texttt{idle}, \texttt{SEND}) &= \texttt{sending}, \\
\delta(\texttt{sending}, \texttt{OPEN}) &= \texttt{streaming}, \\
\delta(\texttt{streaming}, \texttt{DELTA}) &= \texttt{streaming}, \\
\delta(\texttt{streaming}, \texttt{USAGE}) &= \texttt{streaming}, \\
\delta(\texttt{streaming}, \texttt{DONE}) &= \texttt{idle}, \\
\delta(\_, \texttt{ERROR}) &= \texttt{idle}, \quad \delta(\_, \texttt{ABORT}) = \texttt{idle}.
\end{aligned}$$

[*] Project repository: github.com/mustafaras/Synapse_IDE

The implementation enforces the single-active-stream invariant

$$\forall t:\ \#\{\text{activeRequest}(t)\} \le 1,$$

yielding predictable UI enable/disable behavior and bounded cleanup on failure.

(ii) Stream Semantics with Buffered Rendering. Provider responses are a time-ordered event sequence

$$E = \langle e_0 = \texttt{OPEN},\, e_1 = \texttt{DELTA}(\Delta_1),\, \dots,\, e_k = \texttt{DELTA}(\Delta_k),\, e_{k+1} = \texttt{USAGE}(u),\, e_{k+2} = \texttt{DONE}\rangle.$$

Let $B(t)$ denote the in-memory buffer at time t and $M(t)$ the user-visible partial message. Deltas accumulate as

$$M(t) = \text{concat}\big(\Delta_1, \dots, \Delta_{i(t)}\big), \qquad i(t) \le k.$$

The UI applies a frame-synchronous flush operator

$$\text{flush}(t_f) : B(t_f) \mapsto \text{DOM}, \qquad t_f \in \mathcal{T}_{\text{frame}},$$

which amortizes DOM mutations and prevents render thrash. When the arrival rate of `DELTA` events r transiently exceeds the commit capacity c, we have $|B| \uparrow$; periodic flush drives $|B| \to 0$ while preserving the ordering of Δ_j.

(iii) Deterministic Context Budgeting for Prompts. Given a token budget B and candidate context slices $C = \{c_i\}$ with priority $w_i \ge 0$ and length estimate $\ell_i \approx \lceil |c_i|/4 \rceil$, we select a subset $S \subseteq C$ that maximizes visible utility under the budget:

$$\max_{S \subseteq C} \sum_{c_i \in S} w_i \quad \text{s.t.} \quad \sum_{c_i \in S} \ell_i \le B.$$

The implemented policy is a deterministic greedy scheme: (1) system and user instructions are inserted first; (2) remaining slices are ordered by a fixed priority schedule (optionally by score w_i/ℓ_i); (3) the last slice is right-truncated to fit,

$$c_i' = c_i[1{:}\tau] \ \text{ with } \ \ell_i' \le B - \sum_{j<i} \ell_j,$$

ensuring reproducible context composition without relying on model-specific tokenizers.

Additionally, the virtualized message list renders a window R with overscan to keep per-frame layout work $O(|R|)$, and the editor bridge applies a guarded insertion operator Φ with a size threshold $L_{\max}$ to prevent large, unsafe writes:

$$\|\Delta\| \le L_{\max} \Rightarrow \Phi(\Delta) = \text{insert/replace}, \qquad \|\Delta\| > L_{\max} \Rightarrow \Phi(\Delta) = \text{confirm-and-chunk}.$$

Together, these design choices deliver low-latency streaming, deterministic UI state, and safe code application, consistent with the observable behavior of the deployed application modules.

1 Introduction & Scope

Problem. Modern code editors increasingly rely on AI assistance. In a browser-based workspace, the assistant must deliver *reliable, low-latency* interactions while the UI remains *deterministic* and code application *safe*. Concretely, the system must (i) avoid race conditions in chat control, (ii) render token streams smoothly without layout thrash, (iii) compose prompts under a fixed token budget, and (iv) apply generated code to the editor without violating size or selection constraints.

Scope. This work formalizes the observable behavior of an SPA-based IDE that integrates an AI assistant, virtualized chat, Monaco editor, and a file explorer. The focus is on client-side control logic—state machines, streaming, budgeting, rendering, and guarded editor writes. Server-side inference, training, and non-UI tooling are out of scope.

Design Objectives

We distill the implementation intent into five measurable objectives:

O1 Determinism. For any state s_t and event e_t with request id match, the next state s_{t+1} is unique:

$$(s_t, e_t, \mathrm{rid}) = (s'_t, e'_t, \mathrm{rid}) \;\Rightarrow\; s_{t+1} = s'_{t+1}.$$

O2 Low latency. Let $\mathbf{T} = (T_{\mathrm{open}}, T_{\mathrm{first}}, T_{\mathrm{done}})$ be stream milestones. We target a tail constraint

$$\Pr[T_{\mathrm{first}} \le \theta] \;\ge\; 0.95,$$

for a configurable threshold θ.

O3 Smooth streaming. DOM mutations amortized to frame times to minimize reflow:

$$\mathrm{flush}(t_f) : B(t_f) \to \mathrm{DOM}, \quad t_f \in \mathcal{T}_{\mathrm{frame}}.$$

O4 Token-constrained prompts. Deterministic context composition under budget B (cf. Sec. **??**).

O5 Editor safety. A guarded apply operator Φ with a size threshold $L_{\max}$ prevents unsafe writes.

System Assumptions

- Single-page React runtime with a single active assistant session.
- Provider responses arrive as ordered deltas (`OPEN` $\to$ `DELTA`* $\to$ `USAGE`? $\to$ `DONE`).
- Token budget B, max-apply length $L_{\max}$, and retry limits are finite and configurable.

Contributions (as implemented)

C1: Deterministic chat state machine. We specify $\mathcal{M} = (S, \Sigma, \delta, s_0, F)$ with $S = \{\texttt{idle}, \texttt{sending}, \texttt{streaming}\}$, $F = \{\texttt{idle}\}$, and a request-id guard that yields the single-active-stream invariant

$$\forall t : \ \#\{\text{activeRequest}(t)\} \leq 1.$$

This ensures predictable enable/disable of the composer, clean abort, and bounded recovery on error.

C2: Stream buffer with frame-synced flush. Provider chunks Δ_j are accumulated and flushed at frame times, which decouples network burstiness from UI commits. Let r be the arrival rate of `DELTA` events and c the sustainable commit rate. When $r > c$, buffer size $|B|$ increases; periodic flush drives $|B| \to 0$ while preserving order:

$$M(t) = \text{concat}(\Delta_1, \ldots, \Delta_{i(t)}), \quad i(t) \uparrow .$$

C3: Context compiler with token budget fit. Given slices $C = \{c_i\}$ with priorities $w_i \geq 0$ and estimates $\ell_i \approx \lceil |c_i|/4 \rceil$, select $S \subseteq C$:

$$\max_{S \subseteq C} \sum_{c_i \in S} w_i \quad \text{s.t.} \quad \sum_{c_i \in S} \ell_i \leq B.$$

The policy is deterministic: mandatory system/user instructions first; remaining slices ordered by a fixed schedule (optionally by w_i/ℓ_i); last slice right-truncated to fit B:

$$c'_i = c_i[1\!:\!\tau], \quad \ell'_i \leq B - \sum_{j<i} \ell_j.$$

C4: Virtualized message list. Let V be the viewport window and o the overscan; the render set is $R = V \oplus o$. Heights are measured via a resize observer; per-frame layout work is $O(|R|)$, independent of total message count.

C5: Editor apply-plan with size guards. We define a guarded apply operator Φ:

$$\|\Delta\| \leq L_{\max} \ \Rightarrow \ \Phi(\Delta) = \text{insert/replace}, \qquad \|\Delta\| > L_{\max} \ \Rightarrow \ \Phi(\Delta) = \text{confirm-and-chunk}.$$

Cursor and scroll state are preserved; failures trigger undo-safe recovery.

C6: Provider-agnostic adapters and bounded retries. HTTP/SSE adapters normalize events across providers and expose uniform `OPEN/DELTA/USAGE/DONE` semantics. Network errors use bounded retry with exponential backoff

$$t_k = t_0 \, r^{\,k-1}, \qquad k = 1, \ldots, N,$$

and any `ABORT` (user-initiated, superseded request, or unmount) resets $\mathcal{M}$ to `idle`.

Non-Goals

The design does not attempt optimal token packing, concurrent multi-stream rendering, or model-specific tokenizer accuracy. Instead, it prioritizes determinism, safety, and reproducibility with minimal assumptions.

Evaluation Targets

We instrument latency and throughput:

$$\mathbf{T} = (T_{\text{open}}, T_{\text{first}}, T_{\text{done}}), \qquad \hat{r} = \frac{\sum_j |\Delta_j|}{T_{\text{done}} - T_{\text{open}}}.$$

Success criteria include tail-latency bound $\Pr[T_{\text{first}} \le \theta] \ge 0.95$, stable FPS under streaming, and zero violations of $L_{\max}$ in editor apply.

2 System Overview

Runtime Architecture

The application is a single-page React runtime composed of cooperating subsystems:

- **App Shell** (routing, theming, error boundary),
- **AI Assistant** (chat, settings, provider adapters),
- **Editor (Monaco)** (tabs, apply/replace, undo safety),
- **File Explorer & Search** (tree state, async search worker),
- **Toast & Telemetry** (user feedback, instrumentation).

At a high level, the composition can be expressed as a product of modules observed in the codebase:

$$\text{App} \;\cong\; \text{Shell} \;\oplus\; \text{Assistant} \;\oplus\; \text{Editor} \;\oplus\; \text{Explorer} \;\oplus\; \text{Search} \;\oplus\; \text{UX}_{\text{toast,telemetry}}.$$

Data-Plane Pipeline

User interaction and model responses traverse a deterministic pipeline:

$$x_{\text{user}} \xrightarrow{\Gamma} C \xrightarrow{\mathcal{P}} E \xrightarrow{\beta} B \xrightarrow{\text{flush}} V \xrightarrow{\Phi} \text{Editor}.$$

x_{user}	: input event (text, selection, attachments)
$\Gamma : x_{\text{user}} \mapsto C$	: context compiler under budget B

$C = \{c_i\}$	: ordered context slices (system, user, file, selection, notes)
$\mathcal{P} : C \mapsto E$	: provider stream, events OPEN/DELTA/USAGE/DONE
$E = \langle e_j \rangle_{j=0}^{k+2}$	: time-ordered event sequence
$\beta : E \mapsto B(t)$	: UI buffer accumulation of text deltas
flush : $B(t_f) \mapsto V$	: frame-synchronous commit to virtual view
$\Phi : V \mapsto \text{Editor}$	: guarded apply (insert/replace) with $L_{\max}$

Control-Plane Coordination

A deterministic FSM $\mathcal{M}$ (cf. Abstract) governs transitions between `idle`, `sending`, and `streaming`. It gates input focus, composer enable/disable, abort semantics, and cleanup. The control plane and data plane are decoupled:

$$\text{Control}: \ \mathcal{M} \quad \perp \quad \text{Data}: \ (\Gamma, \mathcal{P}, \beta, \text{flush}, \Phi),$$

yet synchronized via request identifiers and lifecycle events to ensure single-stream invariants and ordered rendering.

Virtualization Window and Rendering

Let V denote the viewport window and o the overscan. The render set is

$$R \ = \ V \oplus o, \qquad \text{per-frame layout work} \ \in O(|R|).$$

Element heights are measured via a resize observer; scroll-to-bottom and pinning are handled without reflow storms. Streaming text is appended through buffered flushes aligned to frame times, minimizing DOM mutation cost.

Configuration Parameters

- **Token budget** B for context composition (deterministic fit).
- **Apply threshold** $L_{\max}$ for editor safety.
- **Overscan** o for virtualized message lists.
- **Retry/backoff** (N, t_0, r) for network robustness with SSE/HTTP.

Temporal Milestones and Metrics

We instrument stream latency and throughput:

$$\mathbf{T} = (T_{\text{open}},\ T_{\text{first}},\ T_{\text{done}}), \qquad \hat{r} \ = \ \frac{\sum_j |\Delta_j|}{T_{\text{done}} - T_{\text{open}}}.$$

A target tail bound $\Pr[T_{\text{first}} \leq \theta] \geq 0.95$ defines acceptable responsiveness. Frame-synchronous flushes keep UI variance low while preserving event order.

Fault Handling and Abort Semantics

Errors are classified (network, provider, parse, user-abort). A bounded-retry policy with exponential backoff

$$t_k \;=\; t_0\, r^{\,k-1}, \qquad k = 1, \ldots, N,$$

is applied where appropriate; any `ABORT` resets $\mathcal{M} \to \texttt{idle}$ and clears transient buffers, guaranteeing consistent UI state after failure.

Interfaces to Adjacent Subsystems

- **Assistant $\leftrightarrow$ Editor:** apply-plan Φ (insert/replace), cursor preservation, undo safety.
- **Assistant $\leftrightarrow$ Explorer/Search:** active file, selection, and snippets as context slices c_i.
- **Assistant $\leftrightarrow$ Toast/Telemetry:** status notifications, timing traces, usage summaries.

This separation enables deterministic control with reproducible data-path behavior, as observed in the implemented modules.

3 Formal Models

3.1 Conversation FSM

We model the chat controller as a deterministic finite-state machine

$$\mathcal{M} = (S, \Sigma, \delta, s_0, F), \qquad S = \{\texttt{idle}, \texttt{sending}, \texttt{streaming}\}, \qquad F = \{\texttt{idle}\}.$$

The event alphabet is

$$\Sigma = \{\texttt{SEND},\ \texttt{OPEN},\ \texttt{DELTA},\ \texttt{USAGE},\ \texttt{DONE},\ \texttt{ERROR},\ \texttt{ABORT}\}.$$

Each runtime request carries a request identifier $\text{rid} \in \mathbb{N}$. A compatibility predicate $\text{match}(e, \text{rid})$ ensures that only events associated with the current request advance the machine.

Transition axioms (request-id guarded). All transitions below are taken *iff* $\text{match}(e, \text{rid})$ holds:

$$\begin{aligned}
\delta(\texttt{idle}, \texttt{SEND}) &= \texttt{sending},\\
\delta(\texttt{sending}, \texttt{OPEN}) &= \texttt{streaming},\\
\delta(\texttt{streaming}, \texttt{DELTA}) &= \texttt{streaming},\\
\delta(\texttt{streaming}, \texttt{USAGE}) &= \texttt{streaming},\\
\delta(\texttt{streaming}, \texttt{DONE}) &= \texttt{idle},\\
\delta(\_, \texttt{ERROR}) &= \texttt{idle},\\
\delta(\_, \texttt{ABORT}) &= \texttt{idle}.
\end{aligned}$$

Single-active-stream invariant. Let $\text{activeRequest}(t)$ denote the currently active request at time t (if any). Then

$$\forall t : \quad \#\{\text{activeRequest}(t)\} \leq 1.$$

This is enforced by (i) gating `SEND` when $S \neq \texttt{idle}$, and (ii) canceling any prior stream on a superseding `SEND` via a generated `ABORT` that resets $\mathcal{M} \to \texttt{idle}$ before the new request is issued.

Progress (liveness) under fair providers. Assuming a fair provider that eventually returns either `OPEN` or `ERROR`, and after `OPEN` eventually returns one of $\{\texttt{DONE}, \texttt{ERROR}, \texttt{ABORT}\}$, the following holds:

$$\boxed{\texttt{SEND} \;\Rightarrow\; \Diamond\,\texttt{idle}} \quad \text{(the machine eventually returns to } \texttt{idle}\text{)}.$$

In practice, timeouts also generate `ABORT` to uphold this guarantee.

Safety properties (race-free control).

S1 No cross-stream contamination. If $\text{rid}_1 \neq \text{rid}_2$ then events from rid_1 are ignored while rid_2 is active:

$$\neg\,\text{match}(e, \text{rid}_2) \;\Rightarrow\; \delta(s, e) = s.$$

S2 Idempotent abort. Applying `ABORT` in any state yields `idle`; repeating it is a no-op:

$$\delta(\texttt{idle}, \texttt{ABORT}) = \texttt{idle}.$$

S3 Monotone UI gating. Composer-enable, abort-visibility, and scroll-pinning are functions of S only. Thus identical (S, rid) pairs produce identical UI control states.

Timed constraints (configuration-level). Let $(T_{\text{open}}, T_{\text{idle}})$ be soft deadlines for opening a stream and returning to `idle`, respectively. The control loop triggers an internal `ABORT` if any deadline is exceeded:

$$\begin{aligned}
&\text{if } \texttt{sending} \wedge \Delta t > T_{\text{open}} \;\Rightarrow\; \texttt{ABORT},\\
&\text{if } \texttt{streaming} \wedge \Delta t > T_{\text{idle}} \;\Rightarrow\; \texttt{ABORT}.
\end{aligned}$$

State–UI contract (informal).

- `idle`: composer enabled; abort hidden; buffers cleared.
- `sending`: composer disabled (debounced); abort visible; spinner active.
- `streaming`: composer disabled; abort visible; buffered render active; on `DONE`/`ERROR`/`ABORT` $\to$ `idle`.

Proof sketch of determinism. Let (s, e, rid) and (s', e', rid) be two control-plane inputs with identical state and request id. Since δ is a total function on $S \times \Sigma$ and events are request-guarded, if $e = e'$ then

$$(s, e, \mathrm{rid}) = (s', e', \mathrm{rid}) \;\Rightarrow\; \delta(s, e) = \delta(s', e').$$

Hence the next state is unique, establishing determinism of $\mathcal{M}$.

3.2 Streaming Semantics (Event $\to$ UI)

Event sequence. A provider response is modeled as a time-ordered sequence

$$E = \langle e_0 = \texttt{OPEN},\ e_1 = \texttt{DELTA}(\Delta_1),\ \ldots,\ e_k = \texttt{DELTA}(\Delta_k),\ e_{k+1} = \texttt{USAGE}(u),\ e_{k+2} = \texttt{DONE}\rangle.$$

Each Δ_j is a text fragment (possibly empty). The optional $\texttt{USAGE}(u)$ carries accounting metadata $u = (t_{\text{prompt}}, t_{\text{completion}})$.

Buffers and visible state. Let $B(t)$ denote the in-memory buffer at wall time t and $M(t)$ the visible partial message. With $i(t) \in \{0, \ldots, k\}$ the index of the last applied delta,

$$M(t) = \mathrm{concat}(\Delta_1, \ldots, \Delta_{i(t)}), \qquad i(t) \le k. \tag{1}$$

Arriving deltas are appended to $B(t)$; a frame-synchronous operator commits buffered text to the virtualized view:

$$\mathrm{flush}(t_f) :\; B(t_f) \;\longmapsto\; V(t_f), \qquad t_f \in \mathcal{T}_{\text{frame}}. \tag{2}$$

Ordering guarantee. If deltas are received in order, then for all frame times t_f

$$\mathrm{prefix}(M(t_f)) = \mathrm{prefix}(\Delta_1 \| \cdots \| \Delta_{i(t_f)}), \tag{3}$$

where $\|$ denotes concatenation. Thus the UI never presents a re-ordered stream.

Arrival vs. commit (backpressure). Let $r(t)$ be the instantaneous arrival rate (chars/s) and $c(t)$ the sustainable commit rate of the UI. The buffer size obeys the fluid inequality

$$\frac{d}{dt}\|B(t)\| \;\leq\; r(t) - c(t), \tag{4}$$

with frame-synchronous drains at $t \in \mathcal{T}_{\text{frame}}$. If the long-run means satisfy $\bar{r} < \bar{c}$, then $\sup_t \|B(t)\| < \infty$ (bounded backlog). When transiently $r(t)>c(t)$, $\|B(t)\|$ increases; periodic flush drives $\|B\| \to 0$ as pressure subsides.

Amortized DOM work. Let J_f be the number of DOM mutations at frame f. With buffering, the expected J_f satisfies $\mathbb{E}[J_f] \ll \mathbb{E}[\#\text{deltas in } f]$, reducing reflow. In a virtualized list, per-frame layout cost is $O(|R_f|)$, where R_f is the render set (viewport $\oplus$ overscan).

Liveness under fairness. Assume a fair provider that eventually emits either `DONE` or `ERROR` after `OPEN`. Then the control plane (Sec. 3.1) ensures `SEND` $\Rightarrow \Diamond$ `idle`. Timeouts generate `ABORT` to restore quiescence.

Safety with request identifiers. With a current request id rid and a predicate $\text{match}(e, \text{rid})$, any non-matching event leaves the visible state unchanged:

$$\neg\,\text{match}(e, \text{rid}) \;\Rightarrow\; (B, M) \text{ unchanged}. \tag{5}$$

Hence cross-stream contamination is impossible.

Latency and throughput observables. We collect $\mathbf{T} = (T_{\text{open}}, T_{\text{first}}, T_{\text{done}})$ and define throughput

$$\hat{r} \;=\; \frac{\sum_{j=1}^{k}\|\Delta_j\|}{T_{\text{done}} - T_{\text{open}}}. \tag{6}$$

A target tail bound $\Pr[T_{\text{first}} \leq \theta] \geq 0.95$ expresses responsiveness of first contentful delta.

3.3 Context Budgeting (Deterministic Fit)

Problem statement. Given a token budget $B \in \mathbb{N}$ and a finite set of candidate context slices $C = \{c_i\}_{i=1}^{n}$, each with priority $w_i \geq 0$ and a length estimate $\ell_i \approx \lceil |c_i|/4 \rceil$, select a subset $S \subseteq C$ to maximize utility under budget:

$$\max_{S \subseteq C} \sum_{c_i \in S} w_i \quad \text{s.t.} \quad \sum_{c_i \in S} \ell_i \leq B. \tag{7}$$

Deterministic policy. The implementation follows a canonical, reproducible procedure:

1. Insert mandatory slices first (system and user instructions).

2. Order the remaining slices by a fixed priority schedule; optionally refine by the ratio score w_i/ℓ_i.
3. Greedily admit slices while capacity remains. If the next slice would overflow the budget, apply right-truncation:

$$c_i' = c_i[1{:}\tau], \qquad \ell_i' \le B - \sum_{j<i} \ell_j, \tag{8}$$

   and stop.

This yields a single, stable output for a given input configuration.

Feasibility. Right-truncation guarantees budget feasibility: $\sum_{c \in S'} \ell(c) \le B$, where S' is the final multiset after truncation. Hence no overflow can occur at dispatch time.

Complexity. Let m be the number of non-mandatory slices. Sorting dominates with $O(m \log m)$; the linear admission pass and the final truncation are $O(m)$.

Optimality note. Because ℓ_i is a tokenizer-agnostic estimate, the policy targets *deterministic visibility* rather than knapsack optimality. Empirically, the fixed schedule provides stable, predictable prompts across runs and providers.

Stability and reproducibility. Let Π denote the total order induced by the schedule and tie-breaking rules. Then for any two invocations with identical inputs $(B, C, \{w_i\}, \{\ell_i\})$, the composed context is identical:

$$(B, C, \{w_i\}, \{\ell_i\}) \equiv (B, C, \{w_i\}, \{\ell_i\}) \;\Rightarrow\; \text{Compose}_\Pi \text{ is constant.} \tag{9}$$

Editor-safety coupling. The final prompt length is bounded by B. Independently, the apply-plan Φ enforces an insertion threshold $L_{\max}$ in the editor. Thus budgeting and application are decoupled yet jointly ensure that neither the request nor the resulting code write exceeds configured limits.

4 Provider Abstraction & Networking

Unified Provider Interface

The networking layer exposes a provider-agnostic interface that normalizes transport- and vendor-specific details into a common event stream with the alphabet $\{\texttt{OPEN}, \texttt{DELTA}(\Delta), \texttt{USAGE}(u), \texttt{DONE}, \texttt{ERROR}, \texttt{ABORT}\}$. Each adapter implements:

- **Request construction:** headers, model id, temperature, max tokens.
- **Stream normalization:** chunk framing $\Rightarrow$ ordered Δ_j.
- **Usage emission:** a single $\texttt{USAGE}(u)$ with $u = (t_{\text{prompt}}, t_{\text{completion}})$ (and optionally tokens).

- **Termination:** exactly one terminal event in $\{\texttt{DONE}, \texttt{ERROR}, \texttt{ABORT}\}$.

Adapters are *pure* with respect to UI state; all side effects are confined to the event emitter that feeds the control plane (see Sec. 3.1).

SSE / Chunked Transfer Semantics

For server-sent events (SSE) or HTTP chunked transfer, the response body is decoded into a monotone sequence of text fragments:

$$E = \langle \texttt{OPEN},\ \texttt{DELTA}(\Delta_1), \dots, \texttt{DELTA}(\Delta_k),\ \texttt{USAGE}(u),\ \texttt{DONE} \rangle.$$

Δ_j may be empty (keep-alive) or partial code tokens. The parser must:

1. preserve order of Δ_j,
2. coalesce adjacent empty fragments,
3. tolerate partial frames and resume at the next valid boundary,
4. ignore unrecognized fields without failing the stream.

Heartbeat lines (e.g., `: keep-alive`) do not advance visible state.

Bounded Retry with Exponential Backoff

Network faults are retried with bounded attempts N and multiplicative delay:

$$t_k \;=\; \min\!\big(t_{\max},\ t_0\, r^{\,k-1}\big) \;+\; \xi_k, \qquad k = 1, \dots, N, \tag{10}$$

where $t_0{>}0$ is the base delay, $r{>}1$ the backoff factor, $t_{\max}$ an upper cap, and ξ_k is a small uniform jitter to avoid synchronization. A retryable fault class $\mathcal{E}_{\text{retry}}$ includes transient DNS, TCP reset, 429, and selected 5xx. Non-retryable classes ($\mathcal{E}_{\text{fatal}}$)—e.g., 401/403, schema errors—emit `ERROR` immediately.

Termination conditions. The retry loop stops on the first terminal event or after N failed attempts. On exhaustion, the adapter emits `ERROR`; the control plane then issues `ABORT` to reset $\mathcal{M} \to \texttt{idle}$ (see Sec. 3.1).

Timeouts and Deadlines

Two deadlines guard liveness:

$$\begin{aligned} &\text{open-timeout:} && \Delta t > T_{\text{open}} \Rightarrow \texttt{ABORT},\\ &\text{idle-timeout:} && \Delta t > T_{\text{idle}} \Rightarrow \texttt{ABORT}. \end{aligned}$$

The adapter cancels the underlying HTTP/SSE with an abort signal; the control plane consumes `ABORT` and returns to `idle` deterministically.

Error Classification and Mapping

Let E_{net} be transport errors, E_{proto} parser/schema errors, E_{quota} provider quota/limits, and E_{auth} authentication failures. The mapping is:

$$E_{\text{net}} \in \mathcal{E}_{\text{retry}} \Rightarrow \text{retry/backoff}; \quad E_{\text{auth}} \vee E_{\text{quota}} \vee E_{\text{proto}} \Rightarrow \texttt{ERROR}.$$

All terminals are unique per request (exactly-once termination).

Abort Semantics (Idempotent)

Abort can be user-initiated, supersession-induced, or timeout-derived. For any state $s \in \{\texttt{idle}, \texttt{sending}, \texttt{streaming}\}$:

$$\delta(s, \texttt{ABORT}) = \texttt{idle}, \qquad \delta(\texttt{idle}, \texttt{ABORT}) = \texttt{idle}.$$

Adapters must propagate cancellation to the transport promptly (e.g., `AbortController`), ensuring that no further Δ_j are delivered.

Ordering, Deduplication, and Integrity

The adapter preserves FIFO order. If provider frames carry sequence ids $q_1 < q_2 < \cdots$, then any $q_j \leq q_{\text{last}}$ is a duplicate and ignored. Optional integrity checksums $\sigma(\Delta_j)$ may be verified when provided; failure maps to E_{proto}.

Throughput and Flow Control

Let $r(t)$ be measured arrival rate and $c(t)$ the sustainable UI commit rate (see Sec. 3.2). The adapter may apply client-side throttling to cap the emit rate at $\tilde{r}(t) \leq r(t)$ to reduce UI pressure while preserving order. SSE buffering is disabled where possible to minimize head-of-line blocking.

Security Considerations

API keys are supplied by the user and stored client-side; requests are sent over TLS with HTTPS. The adapter must:

- never log raw secrets; redact keys in telemetry,
- enforce CORS and origin checks according to runtime constraints,
- cap response size and line length to prevent memory abuse.

Telemetry and Observability

For each request we record $\mathbf{T} = (T_{\text{open}}, T_{\text{first}}, T_{\text{done}})$, HTTP status, retry count, and a success label in $\{\mathsf{ok}, \mathsf{error}, \mathsf{aborted}\}$. Throughput is estimated as

$$\hat{r} = \frac{\sum_j \|\Delta_j\|}{T_{\text{done}} - T_{\text{open}}}.$$

Metrics enable the tail-latency SLO $\Pr[T_{\text{first}} \le \theta] \ge 0.95$.

Reference Pseudocode (Retry Loop)

```
for k in 1..N:
  send request()
  emit OPEN
  for chunk in stream:
    if isDelta(chunk): emit DELTA(chunk)
    if isUsage(chunk): emit USAGE(u)
    if isDone(chunk):  emit DONE; return
  if fatal(error): emit ERROR; return
  delay = min(t_max, t0 * r**(k-1)) + U(0, eps)
  sleep(delay)

emit ERROR
```

The control plane consumes the terminal event and restores `idle`, ensuring a single active stream invariant throughout (see Sec. 3.1).

5 Streaming Pipeline (Hook-Level Design)

The streaming hook orchestrates request lifecycle, buffering, and UI commits. It exposes a small API to upstream UI code while encapsulating transport details (see Sec. 4) and coordinating with the control-plane FSM (see Sec. 3.1).

Interface Summary

Let Hook provide the following surface:

- send(req): initiate a stream for request req;
- abort(): cancel the current stream if any;
- status() $\in$ {`idle`, `sending`, `streaming`};
- metrics(): returns $(T_{\text{open}}, T_{\text{first}}, T_{\text{done}}, \hat{r})$.

The hook emits UI-facing events $\{\texttt{start}, \texttt{delta}, \texttt{usage}, \texttt{done}, \texttt{error}\}$ to the chat view and to telemetry.

Internal State

At time t, the hook maintains:

$$\begin{aligned}
&\mathrm{rid} \in \mathbb{N} && : \text{request version (monotone)},\\
&B(t) && : \text{in-memory text buffer},\\
&\mathsf{sched} \in \{0,1\} && : \text{RAF flush scheduled flag},\\
&\mathsf{ac} && : \text{transport abort controller},\\
&\mathbf{T} = (T_{\text{open}}, T_{\text{first}}, T_{\text{done}}) && : \text{latency milestones},\\
&\mathsf{timers} = (\tau_{\text{open}}, \tau_{\text{idle}}) && : \text{open/idle deadline handles}.
\end{aligned}$$

Only one stream can be active; rid is incremented for each send.

Event Model and Lifecycle

The hook normalizes provider events (Sec. 4) into a canonical lifecycle:

$$\texttt{start} \;\rightarrow\; \texttt{delta}^* \;\rightarrow\; [\texttt{usage}]? \;\rightarrow\; \texttt{done} \;\mid\; \texttt{error}.$$

1. **start**: emitted after request dispatch and receipt of `OPEN`; records T_{open}.
2. **delta**(Δ): appends Δ to $B(t)$; schedules a RAF flush if $\mathsf{sched} = 0$.
3. **usage**(u): merges accounting metadata $u = (t_{\text{prompt}}, t_{\text{completion}})$.
4. **done**: finalizes buffer, clears timers, records T_{done}.
5. **error**(e): classifies error e, clears timers, resets to quiescent state.

RAF-Batched Flush

To decouple arrival from commit, the hook batches DOM-facing updates:

$$\text{flush}(t_f):\; B(t_f) \;\longmapsto\; V(t_f), \qquad t_f \in \mathcal{T}_{\text{frame}}.$$

Invariant (single-scheduled-flush). If $\mathsf{sched} = 1$, subsequent `delta` events do not schedule additional RAF callbacks. Upon callback execution, the hook drains B into the virtual view, resets $\mathsf{sched} \leftarrow 0$, and, on the first non-empty commit, records T_{first}.

Queueing and Supersession

A new send supersedes any active stream:

1. increment rid, set FSM to `sending`;

2. issue `ABORT` to the transport of the prior request (if any);
3. clear B, cancel outstanding RAF if $\mathsf{sched} = 1$;
4. start deadlines and dispatch the new request.

Guard (version match). Each event handler verifies the current rid; mismatched events are ignored, preventing cross-stream contamination.

Deadlines (Idle/Hard)

Two deadlines limit exposure to stalled transports:

$$\begin{aligned} &\text{open-timeout:} && \Delta t > T_{\text{open}} \Rightarrow \mathsf{abort}(), \\ &\text{idle-timeout:} && \Delta t > T_{\text{idle}} \Rightarrow \mathsf{abort}(). \end{aligned}$$

Both deadlines are cancelled on `done` or `error`. Timeouts surface as a classified `error` with reason $\mathsf{timeout}$.

Error Classification

Errors are mapped into a small taxonomy for deterministic UI responses:

$$e \in \{\, \mathsf{network},\ \mathsf{auth},\ \mathsf{quota},\ \mathsf{protocol},\ \mathsf{timeout},\ \mathsf{user\text{-}abort}\,\}.$$

Each class has a canonical message and severity, and determines whether retry is offered.

Cleanup Procedure

On `done`, `error`, or $\mathsf{abort}()$:

- cancel deadlines $\tau_{\text{open}}, \tau_{\text{idle}}$;
- clear B and ensure $\mathsf{sched} = 0$;
- call $\mathsf{ac.abort}()$ on the transport (idempotent);
- reset FSM to `idle`; emit terminal UI event.

Correctness Properties

Stability under batching. For any event sequence with fixed rid, the final visible message equals the concatenation of all received deltas, independent of RAF scheduling:

$$V_{\text{final}} = \Delta_1 \| \cdots \| \Delta_k.$$

At-most-one-active-stream. Enforced by supersession and version guards:

$$\forall t: \ \#\{\text{activeRequest}(t)\} \le 1.$$

First-delta latency. The hook records T_{first} on the first non-empty flush, enabling the SLO $\Pr[T_{\text{first}} \leq \theta] \geq 0.95$ (see Sec. 2).

Complexity

Let k be the number of deltas. Buffer appends and coalesced flushes yield $O(k)$ text work; virtualized rendering performs $O(|R_f|)$ layout per frame, where R_f is the render set (viewport $\oplus$ overscan).

Reference Pseudocode (Hook Core)

```
state: rid = 0, buffer = "", sched = 0
timers: openTimer, idleTimer
ctrl: abortController

function send(req):
  rid += 1
  abort()               // idempotent; cancels prior transport
  buffer = ""; sched = 0
  startOpenDeadline()
  ctrl = new AbortController()
  dispatch(req, ctrl)   // provider adapter
  // on first transport byte:
  onOpen(rid)

function onOpen(r):
  if r != rid: return
  emit("start"); setFSM("streaming"); startIdleDeadline()

function onDelta(r, chunk):
  if r != rid: return
  buffer += chunk
  if sched == 0:
    sched = 1
    requestAnimationFrame(flush)

function flush():
  // drain buffer to virtual view, coalesce DOM writes
  if firstNonEmptyCommit(): mark T_first
  render(buffer); buffer = ""
  sched = 0
```

```
function onUsage(r, u):
  if r != rid: return
  emit("usage", u)

function onDone(r):
  if r != rid: return
  flushIfPending(); clearDeadlines()
  emit("done"); setFSM("idle"); mark T_done

function onError(r, err):
  if r != rid: return
  classify(err); flushIfPending(); clearDeadlines()
  emit("error", err); setFSM("idle")

function abort():
  clearDeadlines()
  if ctrl: ctrl.abort()
  buffer = ""; sched = 0
  emit("error", { reason: "user-abort" })
  setFSM("idle")
```

This design realizes the event lifecycle `start` $\rightarrow$ `delta`* $\rightarrow$ [`usage`]? $\rightarrow$ `done` | `error`, enforces version-guarded race prevention, and amortizes UI work through RAF-batched flushes.

6 Conversation State Machine (FSM) and Context Compilation & Token Budgeting

6.1 Conversation State Machine (FSM)

We formalize the chat controller as a deterministic finite-state machine

$$\mathcal{M} = (S, \Sigma, \delta, s_0, F), \qquad S = \{\texttt{idle}, \texttt{sending}, \texttt{streaming}\}, \qquad F = \{\texttt{idle}\}.$$

The event alphabet is

$$\Sigma = \{\texttt{SEND},\ \texttt{OPEN},\ \texttt{DELTA},\ \texttt{USAGE},\ \texttt{DONE},\ \texttt{ABORT},\ \texttt{ERROR}\}.$$

Each runtime request carries a monotonically increasing request identifier rid $\in \mathbb{N}$ and a *lastActivity* timestamp $\tau \in \mathbb{R}_{\geq 0}$ that is updated on any visible progress.

Deterministic transitions (requestId-guarded). Events advance the machine only if they match the current rid:

$$\begin{aligned}
\delta(\texttt{idle}, \texttt{SEND}) &= \texttt{sending},\\
\delta(\texttt{sending}, \texttt{OPEN}) &= \texttt{streaming},\\
\delta(\texttt{streaming}, \texttt{DELTA}) &= \texttt{streaming},\\
\delta(\texttt{streaming}, \texttt{USAGE}) &= \texttt{streaming},\\
\delta(\texttt{streaming}, \texttt{DONE}) &= \texttt{idle},\\
\delta(\_, \texttt{ABORT}) &= \texttt{idle},\\
\delta(\_, \texttt{ERROR}) &= \texttt{idle}.
\end{aligned}$$

Invariants.

1. *At most one active stream:* $\forall t: \ \#\{\text{activeRequest}(t)\} \le 1$.
2. *Monotone activity clock:* if an event changes visible state then the new timestamp satisfies $\tau_{\text{new}} \ge \tau_{\text{old}}$.
3. *Idempotent abort:* $\delta(\texttt{idle}, \texttt{ABORT}) = \texttt{idle}$; repeated aborts are no-ops.

Deadlines and liveness. Two configurable deadlines drive quiescence:

$$\begin{aligned}
&\text{open-timeout:} \quad (\texttt{sending} \ \wedge \ t - \tau > T_{\text{open}}) \ \Rightarrow \ \texttt{ABORT},\\
&\text{idle-timeout:} \quad (\texttt{streaming} \ \wedge \ t - \tau > T_{\text{idle}}) \ \Rightarrow \ \texttt{ABORT}.
\end{aligned}$$

Assuming a fair provider (eventually emits `DONE` or `ERROR`), every `SEND` eventually leads back to `idle` (liveness).

Race prevention with rid. Let rid_{cur} be the active request id. Any event with $\text{rid} \ne \text{rid}_{\text{cur}}$ is ignored by the reducer and leaves the state and buffers unchanged. This forbids cross-stream contamination.

Reducer-style pseudocode (ASCII-safe).

```
state = { mode: "idle", rid: 0, lastActivity: 0 }

on(SEND):
  if state.mode != "idle": emit(ABORT)      // supersede prior
  state = { mode: "sending", rid: state.rid + 1, lastActivity: now() }

on(OPEN) if rid == state.rid:
  state = { ...state, mode: "streaming", lastActivity: now() }

on(DELTA) if rid == state.rid:
  appendToBuffer(delta); lastActivity = now()
```

```
on(USAGE) if rid == state.rid:
  recordUsage(u); lastActivity = now()

on(DONE or ERROR or ABORT) if rid == state.rid:
  flushIfPending(); state = { mode: "idle", rid: state.rid, lastActivity: now() }
```

UX contract (state → controls).

`idle`	composer enabled; abort hidden; buffers clear; pin-to-bottom optional
`sending`	composer disabled; abort visible; spinner active; no deltas yet
`streaming`	composer disabled; abort visible; buffered render; auto-scroll while pinned

UI logic depends only on `mode` (and pinning), yielding reproducible behavior.

Complexity. Transition evaluation is $O(1)$. Buffered rendering cost is governed by the virtualized list and frame-synchronous flush (see streaming pipeline section).

6.2 Context Compilation and Token Budgeting

We formalize the compilation of prompt context from deterministic sources and its admission under a fixed token budget.

Sources and slice assembly. Let the ordered set of sources be

$$\mathcal{S} = [\ \mathsf{system},\ \mathsf{user},\ \mathsf{activeFile},\ \mathsf{selection},\ \mathsf{pinnedNotes},\ \mathsf{projectBrief},\ \mathsf{attachments}\ ].$$

Each source produces zero or more *slices* c_i (strings) together with metadata: priority $w_i \geq 0$, per-slice cap $b_i \in \mathbb{N}$, and an estimate of token length ℓ_i. The compiler applies the following deterministic passes:

1. **Normalization:** trim whitespace; sanitize; reflow long lines.
2. **Capping:** if $|c_i| > b_i$ then $c_i \leftarrow c_i[1{:}b_i]$.
3. **Stable ordering:** concatenate in the fixed source order $\mathcal{S}$, preserving intra-source order (stable sort).

Length estimation (token heuristic). For any slice c, define a tokenizer-agnostic estimate

$$\ell(c) \ \approx\ \left\lceil \frac{|c|}{4} \right\rceil.$$

This constant-factor proxy enables model-agnostic budgeting with deterministic outcomes.

Budgeted selection (fit-to-budget). Given a global budget $B \in \mathbb{N}$, select a subset $S \subseteq C$ to maximize visible utility under capacity:

$$\max_{S \subseteq C} \sum_{c \in S} w(c) \quad \text{s.t.} \quad \sum_{c \in S} \ell(c) \leq B.$$

The implementation adopts a reproducible greedy procedure:

1. Admit all mandatory slices first (system, user), subject to caps.
2. Order remaining slices by a fixed schedule (e.g., selection before file), with optional tie-break by ratio $w(c)/\ell(c)$.
3. Greedily admit while capacity remains. If the next slice would overflow, apply right-truncation $c'[1{:}\tau]$ so that $\ell(c') \leq B - \sum_{d \in S} \ell(d)$, then stop.

Feasibility. Right-truncation guarantees $\sum_{d \in S'} \ell(d) \leq B$.

Prompt versioning and parameter interpolation. Prompts are stored as labeled templates in a registry and loaded by key. Let $\mathcal{T}$ be a template string with placeholders $\{\texttt{\$name}\}$. Given a parameter map $\theta : \texttt{key} \mapsto \texttt{value}$, the interpolation is the capture-avoiding substitution

$$\mathrm{Interp}(\mathcal{T}, \theta) \;=\; \mathcal{T}\;\big[\;\texttt{\$k} \;\mapsto\; \theta(\texttt{k})\;\big]_{\texttt{k} \in \mathrm{dom}(\theta)}.$$

Determinism: registry lookup and interpolation order are fixed; missing keys either use defaults or produce a validation error before dispatch.

Safety and redaction. Before admission, slices are sanitized (HTML/markdown), binary attachments are size-limited, and secrets are redacted. This ensures the composed prompt does not leak keys and does not exceed transport or provider limits.

Complexity and reproducibility. Let m be the number of non-mandatory slices. Stable ordering is $O(m \log m)$ if priority sorting is required (or $O(m)$ with a fixed bucket order); greedy admission and a single truncation are $O(m)$. Because ordering, caps, and tie-breakers are fixed, the compiler is a pure function: identical inputs (B, C, w, b) produce identical prompts.

Coupling to editor safety. Context budgeting bounds the request size; independently, the editor apply-plan enforces a maximum insertion length $L_{\max}$. The two are decoupled but jointly guarantee that neither the outbound prompt nor the inbound code write exceeds configured limits.

Pseudocode (ASCII-safe).

```
function compileContext(sources, B):
  slices = []
  for src in [system, user, activeFile, selection, pinnedNotes, projectBrief, attachments]:
    for c in normalize(loadSlices(src)):
      c = cap(c, capFor(src))            // per-slice cap
      slices.append({ text: c, w: weight(src) })

  // fixed stable order, optional ratio tie-break
  slices = stableSort(slices, by=priorityThenRatio)

  out = []
  used = 0
  for s in slices:
    need = estTokens(s.text)             // ceil(|c|/4)
    if used + need <= B:
      out.append(s.text); used += need
    else:
      room = B - used
      if room <= 0: break
      s.text = rightTruncateToTokens(s.text, room)
      out.append(s.text); used = B
      break

  return join(out)
```

Diagnostics. The compiler emits per-slice diagnostics: estimated tokens, admitted vs. dropped length, and final budget utilization $U = 100 \cdot \frac{\sum_{c \in S} \ell(c)}{B}\,\%$. These signals, together with FSM metrics (open/first/done), enable regression tracking for latency and determinism without relying on provider-specific tokenizers.

7 Message Rendering & Virtualization

Virtual List Model

Let the chat contain n messages with measured heights $h_i > 0$ and cumulative prefix sums $H_i = \sum_{j=1}^{i} h_j$ ($H_0 = 0$). The viewport is a closed interval $[y,\, y + W] \subseteq [0,\, H_n]$ where W is the viewport height. With overscan $o \geq 0$ (in device pixels), the *render window* is

$$R \;=\; \{\, i \in [1..n] \mid H_{i-1} < y + W + o \;\wedge\; H_i > y - o \,\}.$$

Only items in R are materialized, so per-frame layout work is $O(|R|)$, independent of n. Heights h_i are maintained via a ResizeObserver that updates H_i incrementally.

Debounced recompute. Let req be an RAF-scheduled recompute flag. Any height or scroll change sets req $\leftarrow 1$; if req $= 1$ at frame t_f, the list recomputes R once and resets req $\leftarrow 0$.

This frames recomputation and prevents thrash under streaming updates.

Pin-to-bottom behavior. Define a pinned predicate

$$\text{pinned} \;\equiv\; (H_n - (y + W) \le \delta),$$

with a small tolerance δ (e.g., a few pixels). When pinned is true and a new delta produces $H'_n > H_n$, the scroller advances so that the bottom remains visible. When pinned is false, the current $[y,\, y + W]$ is preserved by anchoring the first fully visible item and correcting for any Δh measured on that anchor.

Streaming-Friendly Code-Fence Handling

During streaming, partial code blocks must render incrementally without breaking layout or syntax highlighting. We model a small fence automaton over incoming text fragments.

Fence automaton. Let the marker be the ASCII triple backtick `‘‘‘`. The automaton has states {OUTSIDE, IN_FENCE(`lang`)}. On receiving a fragment Δ:

1. If OUTSIDE and Δ opens a fence (`‘‘‘lang`), transition to IN_FENCE(`lang`), stash `lang`, and start a fence buffer.
2. While IN_FENCE(`lang`), append text lines to the fence buffer. If a closing `‘‘‘` appears, flush the fenced block as a code widget with syntax highlighting for `lang`, then transition to OUTSIDE.
3. If the stream ends while in IN_FENCE, render the partial buffer as preformatted text (monospace) without language-scoped highlighting; the widget upgrades when the closing fence arrives.

Ordering guarantee. Because fragments Δ_j are processed in order (Sec. 3.2), fence open/close parity is preserved; no reordering is applied across fences.

Inline highlights and composition. Outside fences, the renderer applies lightweight markdown rules (inline code, emphasis) and a safe subset of links. The message composition pipeline proceeds:

$$\mathsf{delta} \;\rightarrow\; \mathsf{tokenize} \;\rightarrow\; \mathsf{fence\_fsm} \;\rightarrow\; \mathsf{compose\_nodes} \;\rightarrow\; \mathsf{virtual\_list\_commit}.$$

All DOM commits are batched to frame times by the hook (Sec. 5).

Stability & Correctness Properties

1. **Viewport invariance under height drift.** Let $i^\star$ be the top-anchored item before recompute and Δh the measured change in its offset. The scroll position is adjusted by $-\Delta h$,

preserving the first fully visible line.

2. **Monotone bottom under pin.** If pinned is true, after any delta commit the bottom remains within δ of H'_n.
3. **Fence safety.** Partially streamed fences are rendered as inert preformatted text; language highlighting is enabled only after a matched close, preventing invalid DOM nesting or expensive re-highlighting per character.

Complexity

For k deltas in a frame, tokenization and fence FSM are $O(\sum |\Delta_j|)$. Virtualization keeps layout work at $O(|R|)$. Resize observations are amortized; coalesced height writes avoid layout thrash.

Reference Pseudocode (ASCII-safe)

```
state:
  items = [message nodes]
  heights = []
  prefix = []
  pinned = true
  requestRecompute = 0

onResize(i, newHeight):
  delta = newHeight - heights[i]
  heights[i] = newHeight
  for j >= i: prefix[j] += delta
  requestRecompute = 1

onScroll(y):
  pinned = (prefix[n] - (y + viewportH) <= tol)
  requestRecompute = 1

onFrame():
  if !requestRecompute: return
  R = computeRenderWindow(y, viewportH, overscan, prefix)
  materialize(items[R])
  requestRecompute = 0
```

8 AI Assistant UI & Interactions

Component Responsibilities

- **AiAssistantSimple.tsx**: message composer, attachment bar, message list, abort control, keyboard scopes, and glue code for the streaming hook (Sec. 5) and virtualization (Sec. 7).
- **VirtualMessageList.tsx**: windowing, height measurement, pin-to-bottom, and incremental commits.
- **Settings Modal (components/ai/settings/*)**: provider keys, model selection, streaming toggle, safety level, thread import/export, provider self-tests.
- **Toasts (ui/toast/*)**: inline diagnostics, success/error notifications, rate-limited display.

Composer & Message Updates

Messages are updated via three deterministic operations:

1. **append(msg)**: push a new user or assistant message (immutable id).
2. **replace(id, patch)**: in-place patch for streaming assistant nodes (e.g., text delta, metadata).
3. **deferred(id, fn)**: queue a microtask that reads the latest node and applies a functional update node $\mapsto fn(\text{node})$.

Invariant (id-stable streaming). The streaming assistant node keeps the same id across `OPEN/DELTA/DONE`; only its content and flags change, ensuring referential stability for the virtual list.

Keyboard Scopes and Shortcuts

Keyboard handling is namespaced by a scope token (e.g., `chat`, `editor`). Within `chat`:

- `Enter`: send (unless `Shift` held),
- `Esc`: abort current stream,
- `Ctrl/Cmd+K`: focus quick actions,
- `Ctrl/Cmd+Up/Down`: navigate message history.

Scopes are mutually exclusive; scope switching updates ARIA live regions and focus rings to maintain accessibility.

Attachment Bar

The bar accepts files and references (active file, selection, project brief). For each attachment we record type, size, and a redaction flag. The context compiler converts attachments to slices with per-slice caps (Sec. 6).

Persistence Bootstrap and Multi-Tab Coherence

Assistant state (settings and threads) is persisted in a namespaced store. On mount:

1. load settings and last thread snapshot;
2. validate provider keys and models (self-test);
3. subscribe to a storage event channel for multi-tab coherence; remote writes invalidate local caches and trigger a soft refresh.

Consistency. A monotone version counter is stored with each thread; newer versions always replace older local copies.

Advanced Settings Modal

Fields include:

- **Provider** (enum), **API key** (secret), **Model** (enum),
- **Streaming** toggle, **Temperature**, **Max tokens**,
- **Safety level** (controls truncation or guardrails),
- **Thread import/export** (JSON), **Provider test** (sends a harmless probe to verify connectivity and streaming).

Validation errors surface inline; secrets are never logged, and previews redact middle characters (e.g., `sk-****-tail`).

Toasts and Inline Diagnostics

Diagnostics are funneled through a toast bus with severity levels $\{\mathsf{info}, \mathsf{success}, \mathsf{warning}, \mathsf{error}\}$.

1. **Deduplication:** a toast key k prevents duplicates within a time window.
2. **Rate limiting:** at most m visible toasts; older ones auto-dismiss.
3. **Context linking:** error toasts link to the failing message id, opening a details panel (request/response excerpts, timings).

Inline diagnostics appear as subtle banners inside messages (e.g., truncated context, redactions applied).

Accessibility

- Live regions announce stream start, first delta, and completion.
- Focus order: settings $\rightarrow$ composer $\rightarrow$ list; escape always returns to the composer in chat scope.
- Keyboard-only operation is complete; all controls have ARIA labels.

Telemetry Signals

Per request we emit $(T_{\text{open}}, T_{\text{first}}, T_{\text{done}})$, status, retry count, truncation ratio, and editor-apply length. UI traces record frame counts between deltas and commits to monitor buffering efficacy.

Reference Pseudocode (ASCII-safe)

```
function onSend():
  append({ id: uid(), role: "user", text: draft })
  hook.send(buildRequest())
  streamingId = append({ id: uid(), role: "assistant", text: "", streaming: true })

hook.on("delta", (txt) => {
  replace(streamingId, (m) => ({ ...m, text: m.text + txt }))
})

hook.on("done", () => {
  replace(streamingId, (m) => ({ ...m, streaming: false }))
  toast.success("Response complete", { key: "resp-complete", dedupe: 4s })
})

hook.on("error", (err) => {
  replace(streamingId, (m) => ({ ...m, streaming: false, error: classify(err) }))
  toast.error(humanize(err), { key: "resp-error", dedupe: 4s })
})
```

Failure Modes and Guards

- **Runaway growth:** large assistant nodes are folded with a “Show more” expander; code blocks collapse after a configurable line count.
- **Lost focus:** abort shows a persistent banner with a single “Retry” action; focus returns to the composer.
- **Secret leakage:** redaction pass runs on both outgoing context and rendered assistant text before commit.

These UI-level guards complement the streaming and budgeting guarantees described in Secs. 5, 3.2, and 6.

9 Editor Integration (Monaco)

Theme, Telemetry, Preview/Sanitize, Toolbar

Let Θ map design tokens to a Monaco theme:

$$\Theta : \ \mathsf{Tokens} \to \mathsf{MonacoTheme}, \qquad \Theta(\tau) = \big(\mathsf{rules}(\tau), \mathsf{colors}(\tau), \mathsf{font}(\tau)\big).$$

Θ is pure and idempotent; view state (cursor, scroll) is preserved across switches.

Editor telemetry uses the alphabet

$$\Sigma_{\text{edit}} = \{\mathsf{opened}, \mathsf{changed}, \mathsf{saved}, \mathsf{undo}, \mathsf{redo}, \mathsf{applied}, \mathsf{lint}\},$$

emitting tuples (e, t, id, ℓ) with wall time t and edit length ℓ. Preview sanitization is a total function

$$S : \ \mathsf{HTML}_{\text{in}} \to \mathsf{HTML}_{\text{out}} = \mathsf{strip} \circ \mathsf{normalize} \circ \mathsf{capLen}$$

with $|S(x)| \le L_{\text{preview\_max}}$.

Toolbar actions $A = \{\mathsf{copy}, \mathsf{cut}, \mathsf{paste}, \mathsf{format}, \mathsf{apply}\}$ have semantics

$$[\![a]\!] : \ \mathsf{EditorState} \to \mathsf{EditorState}, \qquad a \in A,$$

where $[\![\mathsf{apply}]\!]$ delegates to the bridge operator Φ (below).

Editor Bridge & Event Bus

Channels $\mathcal{C} = \{\mathsf{open}, \mathsf{insert}, \mathsf{replace}\}$ carry per-document FIFO payloads: $\mathsf{open}(p)$ with path p; $\mathsf{insert}(q, \Delta)$ at caret q; $\mathsf{replace}([q_1, q_2), \Delta)$ for a half-open range.

Size guard and atomicity. Let $L_{\max}$ be the apply threshold. Define

$$\Phi(\Delta) = \begin{cases} \mathsf{commit}(\Delta), & \|\Delta\| \le L_{\max}, \\ \mathsf{confirm\_and\_chunk}(\Delta), & \text{otherwise.} \end{cases}$$

commit is atomic on the model; $\mathsf{confirm\_and\_chunk}$ partitions $\Delta = \|_i \Delta^{(i)}$ with $\|\Delta^{(i)}\| \le L_{\max}$ and applies after consent.

Commutativity. Two replaces $\mathsf{replace}(R_1, \Delta_1)$ and $\mathsf{replace}(R_2, \Delta_2)$ commute iff $R_1 \cap R_2 = \varnothing$; overlapping edits are serialized by the bus.

Store-Driven Tab Lifecycle

A tab $t = (\mathsf{id}, \mathsf{doc}, \mathsf{cursor}, \mathsf{history}, \mathsf{idx}, \mathsf{savedIdx}, \mathsf{pinned}, \mathsf{dirty})$. Dirty iff $\mathsf{idx} \neq \mathsf{savedIdx}$. Undo/redo update idx in $O(1)$. Batch saves commit all selected tabs atomically (two-phase signal in the store).

Invariants. (i) Save idempotence; (ii) cursor and scroll preserved across theme/apply; (iii) failed apply leaves $\mathsf{history}$ and doc consistent.

Reference pseudocode (ASCII-safe).

```
onApply(rangeOrCaret, delta):
  if size(delta) > L_max:
    if !userConfirms(): return
    for chunk in chunkBySize(delta, L_max):
      commit(rangeOrCaret, chunk)  // atomic
  else:
    commit(rangeOrCaret, delta)
  pushHistory(); savedIdx = savedIdx   // dirty if idx != savedIdx
```

Complexity

Model writes are $O(\|\Delta\|)$. Undo/redo are $O(1)$. Tab activation is $O(1)$. Chunked apply is $\sum_i O(\|\Delta^{(i)}\|)$ with $\|\Delta^{(i)}\| \leq L_{\max}$.

10 File Explorer & Project Operations

Tree Model and Selection

The explorer maintains a rooted ordered tree $\mathcal{R} = (V, E, r, \prec)$ with root r, edges E, and sibling order $\prec$. A node v has path $\pi(v) = r \to \cdots \to v$. Selection $S \subseteq V$ with $|S| \leq 1$ by default.

Actions and Preconditions

Context actions $\{\mathsf{newFile}, \mathsf{newFolder}, \mathsf{rename}, \mathsf{delete}, \mathsf{move}\}$ obey:

$$\mathsf{rename}(v) : v \neq r; \quad \mathsf{delete}(v) : v \neq r; \quad \mathsf{new*}(p) : p \text{ is folder.}$$

Name predicate $\mathrm{ok}(s)$ forbids slashes/control chars and enforces a length cap.

Inline rename. Postconditions: $\mathsf{name}(v) \leftarrow s$ with $\mathrm{ok}(s)$; $\pi(v)$ unchanged; $\prec$ stable among siblings.

Move (DnD). A move $\mathsf{mv}(u \to p)$ is valid iff p is a folder, $u \neq p$, and $u \notin \mathrm{desc}(p)$. Updates $(E, \prec)$ without cycles. Time $O(1)$ for edge rewiring plus $O(\deg(p))$ for stable reindex.

Store API Boundaries

The store exports

$$\mathcal{A}_{\mathrm{fs}} = \{\mathsf{select}, \mathsf{expand}, \mathsf{collapse}, \mathsf{newFile}, \mathsf{newFolder}, \mathsf{rename}, \mathsf{delete}, \mathsf{move}\}.$$

Editor/assistant receive read-only projections (active path, open doc handles), ensuring separation of concerns.

Complexity

Creation/rename/delete are $O(1)$ (amortized), bounded by path map updates. Traversal is $O(|V|)$. Rendering uses incremental diffing; list virtualization (if present) limits DOM work to visible nodes.

11 Search & Indexing

Worker-Based Line Scanning

Given documents $D = \{(f_i, \mathsf{text}_i)\}_{i=1}^{n}$, a Web Worker scans lines for a query q and returns snippets.

Match and snippets. For file f with lines $\ell_1, \ldots, \ell_m$,

$$M_f = \{\, j \in [1..m] \mid \mathrm{match}(\ell_j, q) = 1 \,\}.$$

For a window size $w \in \mathbb{N}$, define $W_j = [\max(1, j-w), \min(m, j+w)]$ and emit $\mathsf{snip}_j = (j,\ \ell_j,\ \{\ell_k\}_{k \in W_j})$.

Async Protocol and Cancellation

Requests carry id ρ. The worker maintains at most one active ρ. Messages: $\mathsf{req}(\rho, q, w)$, $\mathsf{part}(\rho, f, \mathsf{snip})$, $\mathsf{done}(\rho)$, $\mathsf{cancel}(\rho)$, $\mathsf{error}(\rho)$. A new req cancels the prior ρ.

Reference pseudocode (ASCII-safe).

```
onMessage(msg):
  if msg.type == "req":
    if activeRho: cancel(activeRho)
    activeRho = msg.rho
    for (f, text) in documents:
      lines = splitLines(text)
      for (j, line) in enumerate(lines):
        if matches(line, msg.q):
          snip = extractWindow(lines, j, msg.w)
          post({ type: "part", rho: activeRho, file: f, snip })
    post({ type: "done", rho: activeRho })

  if msg.type == "cancel" and msg.rho == activeRho:
    activeRho = null
```

Index Lifecycle, Ordering, Complexity

Maintain $S_{\text{idx}} \in \{\mathsf{cold}, \mathsf{building}, \mathsf{ready}, \mathsf{stale}\}$. On project change, mark stale; next query triggers $\mathsf{building} \to \mathsf{ready}$. Deterministic ordering uses $\mathsf{rank}(f, j) = (\mathsf{path}(f), j)$. Worst-case scan is $O(\sum_i |\mathsf{text}_i|)$; snippet assembly is $O(|W_j|)$.

Safety and Limits

Bound UI and memory by caps: max results $R_{\max}$, max line length $L_{\text{line\_max}}$ (truncate with ellipses), and total payload per ρ. Errors (protocol, worker crash) surface via $\mathsf{error}(\rho)$ with a stable message shape.

12 State Management (Zustand)

Model

Let the global app state be a product

$$\mathcal{S} = \mathcal{S}_{\text{app}} \times \mathcal{S}_{\text{ai}} \times \mathcal{S}_{\text{editor}} \times \mathcal{S}_{\text{explorer}}.$$

Each slice is a finite record with pure, total reducers $f : \mathcal{S} \to \mathcal{S}$. Selectors are projections $p : \mathcal{S} \to V$ such that referential stability holds:

$$p(s) = p(s') \text{ whenever } s \restriction \text{dom}(p) = s' \restriction \text{dom}(p).$$

This guarantees shallow-equality memoization for React components.

Global App Layout & Errors (stores/appStore.ts)

Let

$$\mathcal{S}_{\text{app}} = (\mathsf{layout},\ \mathsf{panels},\ \mathsf{errors}).$$

$\mathsf{layout} \in \{\mathsf{default}, \mathsf{zen}\}$. panels contains split sizes (normalized): for any panel set $\mathbf{w} = (w_1, \dots, w_m)$,

$$w_i \geq 0, \qquad \sum_{i=1}^{m} w_i = 1.$$

Errors are maintained as a deduplicated multiset $\mathsf{errors} = \{(k, c, t)\}$ with key k (hash of {type, origin}), count c, timestamp t. The *add* reducer implements idempotent aggregation:

$$\text{add}(k) : \begin{cases} (k, c, t) \mapsto (k, c{+}1, t') & \text{if } k \in \mathsf{errors}, \\ \varnothing \mapsto (k, 1, t') & \text{otherwise}, \end{cases}$$

and clear(k) removes the key. A finite *TTL* yields eventual collection: entries with $t' < t - \text{TTL}$ are dropped by a maintenance tick. Selectors: $\mathsf{useLayout}$, $\mathsf{usePanelSize}(i)$, $\mathsf{useErrors}()$.

AI Settings Schema & Persistence v2 (stores/aiSettings.schema.ts)

Let the settings vector be

$$\theta = (\mathsf{provider},\ \mathsf{model},\ \mathsf{languageId},\ T,\ M,\ S,\ (\tau_{\text{open}}, \tau_{\text{idle}}),\ \mathsf{keys}),$$

with $T \in [0, 2]$ (temperature), $M \in \mathbb{N}$ (max tokens), $S \in \{0, 1\}$ (streaming toggle). The schema validator $V(\theta)$ is total and returns either ok or a finite set of field errors $\{(f, \text{reason})\}$, where each constraint is explicit, e.g.

$$V(T) = \begin{cases} \mathsf{ok}, & 0 \leq T \leq 2, \\ (\mathsf{temperature}, \text{“out of range”}), & \text{otherwise}. \end{cases}$$

Persistence is versioned. Let $v \in \mathbb{N}$ be the on-disk version tag and $\mu_{a \to b}$ the migration function. For v2 we require:

$$\mu_{1\to2} \text{ is idempotent}, \qquad \mu_{1\to2} \circ \mu_{1\to2} = \mu_{1\to2}.$$

The load pipeline is a left-to-right, deterministic override chain (resolving *env* $\prec$ *query* $\prec$ *disk* $\prec$ *defaults*):

$$\theta^\star = \text{merge}(\theta_{\text{defaults}},\ \theta_{\text{disk}}^{(v\to2)},\ \theta_{\text{env}},\ \theta_{\text{qs}}),$$

where merge is associative and key-wise (last writer wins). Secrets in keys are stored with middle-character redaction in logs; no raw secret is emitted in telemetry.

Editor & File Explorer Stores (selectors, actions, persistence)

Editor slice. Let

$$\mathcal{S}_{\text{editor}} = (D,\ \mathcal{T},\ \mathsf{active},\ \mathsf{undo},\ \mathsf{redo}),$$

where D maps $\mathsf{DocId} \to \mathsf{Text}$, $\mathcal{T}$ is the tab list, and $\mathsf{active} \in \mathcal{T} \cup \{\varnothing\}$. Actions $\{\mathsf{open}, \mathsf{insert}, \mathsf{replace}, \mathsf{save}\}$ are pure transforms. Undo/redo operate on an index ι into a bounded history. Selectors: $\mathsf{useDoc}(id)$ projects by id; referential stability holds whenever Text is unchanged.

Explorer slice. Let the file tree be $\mathcal{R} = (V, E, r, \prec)$ (Sec. 10). State includes expanded set $X \subseteq V$ and selection $S \subseteq V$ with $|S| \leq 1$. Actions: $\{\mathsf{select}, \mathsf{expand}, \mathsf{collapse}, \mathsf{rename}, \mathsf{newFile}, \mathsf{move}\}$. Persistence stores only serializable projections $(V, r, \prec, X, S)$; editor models live in $\mathcal{S}_{\text{editor}}$. Cross-slice invariants: if a file is renamed in $\mathcal{S}_{\text{explorer}}$, all editor tabs referencing its path are updated in a single atomic reducer step.

Complexity and re-render budget. All reducers are $O(1)$ to $O(\log n)$ aside from text edits, which are $O(\|\Delta\|)$. Selectors are structured to avoid object identity churn; hence React re-renders scale with the number of genuinely affected leaves.

13 Design System & Theming

Token System (ui/theme/synapseTheme.ts, constants/design.ts)

Tokens are total maps from semantic roles to numeric/color values:

$$\mathcal{T}_{\text{spacing}} = \{s_k\}_{k=0}^{K}, \qquad s_k = s_0\,\rho^k \quad (\rho > 1),$$

$$\mathcal{T}_{\text{radii}} = \{r_0 < r_1 < \cdots < r_J\}, \qquad \mathcal{T}_{\text{type}} = \{(\text{font}, \text{size}, \text{weight}, \text{lh})\}.$$

Color roles:

$$\mathcal{C} = \{\mathsf{bg},\ \mathsf{surface},\ \mathsf{border},\ \mathsf{text\text{-}primary},\ \mathsf{text\text{-}muted},\ \mathsf{accent\text{-}gold},\ \mathsf{neutral\text{-}graphite}\}.$$

Graphite + soft gold are defined as role anchors; derived roles are affine combinations in an appropriate color space (e.g., OKLCH) to maintain perceptual uniformity.

Glassmorphism preset. A preset is a quadruple

$$G = (\alpha_{\text{bg}},\ \beta_{\text{blur}},\ \alpha_{\text{border}},\ \sigma_{\text{shadow}}),$$

with constraints $0 \leq \alpha_{\text{bg}}, \alpha_{\text{border}} \leq 1$, $\beta_{\text{blur}} \geq 0$, $\sigma_{\text{shadow}} \geq 0$. Surfaces apply G to synthesize CSS variables for backdrop-filter, border, and shadow elevation.

Theme Runtime (contexts/ThemeContext.tsx, styles/theme.ts)

Let the theme mode be $m \in \{\mathsf{light}, \mathsf{dark}, \mathsf{neutral}\}$. The runtime computes a variable map

$$\Lambda(m) :\ \mathsf{TokenKey} \to \mathsf{CSSVarValue},$$

applied to the IDE scope root. Theme transitions are continuous interpolations $\Lambda(m_0) \to \Lambda(m_1)$ along a parameter $\lambda \in [0, 1]$, component-wise:

$$\Lambda_\lambda = (1 - \lambda)\, \Lambda(m_0) + \lambda\, \Lambda(m_1),$$

for numeric channels (with appropriate color-space interpolation for colors).

Contrast constraints. For foreground/background pair (F, B) with relative luminance $L(\cdot)$, the WCAG contrast ratio is

$$\mathrm{CR}(F, B) = \frac{L_{\max} + 0.05}{L_{\min} + 0.05}.$$

The token compiler enforces CR ≥ 4.5 for body text and CR ≥ 3.0 for large text, rejecting configurations that violate the bound.

IDE scope CSS. Variables are scoped to the IDE root (e.g., `.ide-root`) to avoid global leakage. Surface/border variables are layered: `surface-0` (base), `surface-1` (elevated), with monotone opacity and shadow parameters $\sigma^{(0)}_{\text{shadow}} \leq \sigma^{(1)}_{\text{shadow}}$.

Stability & idempotence. Applying $\Lambda(m)$ twice is a no-op (idempotent). Switching modes preserves layout metrics; only paint properties change.

14 Telemetry & Observability

Traces and Spans (utils/obs/instrument.ts, utils/obs/types)

A *trace* is a tree $T = (V, E)$ rooted at v_0, whose nodes are *spans*. Each span v carries

$$v = (\text{id},\ \text{parent},\ t_{\text{start}},\ t_{\text{end}},\ \text{attrs}),$$

with duration $d(v) = t_{\text{end}} - t_{\text{start}} \geq 0$. The active-trace id is a global (*per tab*) token τ^*; spans form a proper tree:

$$\forall v \neq v_0 : \ \text{parent}(v) \in V, \qquad \text{no cycles.}$$

Additivity holds along any path $P \subseteq T$:

$$d(P) = \sum_{v \in P} d(v),$$

while wall-clock of siblings can overlap.

Lifecycle. APIs $\mathsf{spanStart}(k, \text{attrs})$ and $\mathsf{spanEnd}(k)$ are total. Ending an unknown span is ignored (idempotent end); duplicate starts are resolved by generating child spans with a numeric suffix for uniqueness.

Usage/cost estimation. Let $(t_{\text{prompt}}, t_{\text{completion}})$ be usage counters (if present), and let $\widehat{u}_{\text{prompt}}, \widehat{u}_{\text{completion}}$ be model-agnostic estimates (e.g., $\lceil |x|/4 \rceil$). A generic unit cost model is

$$\widehat{\text{cost}} = c_p\, \widehat{u}_{\text{prompt}} + c_c\, \widehat{u}_{\text{completion}},$$

reported as *dev-only indicative* telemetry (no billing claim). When exact usage u arrives, it supersedes $\widehat{u}$ monotonically.

Dev-Only AI Telemetry (services/telemetry.ts)

Action taxonomy. Events are categorized by a finite taxonomy $\mathcal{A}$ (e.g., send, abort, apply, search). Each event emits

$$(\text{traceId},\ \text{spanId},\ \text{action} \in \mathcal{A},\ t,\ \text{attrs}),$$

with attrs including latency, retry count, truncation ratio, and selection sizes.

Endpoints & configuration. The sink endpoint is resolved deterministically:

$$\mathsf{env} \prec \mathsf{querystring} \prec \mathsf{defaults},$$

favoring explicit runtime overrides. Transport uses buffered batching with a cap on batch size $B_{\max}$ and an interval $\Delta t_{\max}$. The sender satisfies

$$\text{flush on } \big(|B| \geq B_{\max}\big) \ \lor\ \big(\Delta t \geq \Delta t_{\max}\big).$$

Rate limiting applies a token bucket with capacity C and fill rate λ, ensuring throughput $\leq \lambda$ in steady state.

Privacy & safety. No raw secrets or PII are logged. Keys are redacted at source; payloads are size-capped and schema-validated. In case of backpressure or offline state, telemetry degrades to local no-op without affecting the app.

Observability invariants.

1. **At-most-once span close:** spanEnd for a completed span is ignored.
2. **Monotone time:** $t_{\text{end}} \geq t_{\text{start}}$ per span; clock skew is corrected by clamping negative deltas to zero for reporting.
3. **Trace integrity:** every non-root span has a valid parent in the local process; orphan drops are counted.

Metrics

The following counters and summaries are emitted:

$$\mathbf{T} = (T_{\text{open}}, T_{\text{first}}, T_{\text{done}}), \qquad \widehat{r} = \frac{\sum_j \|\Delta_j\|}{T_{\text{done}} - T_{\text{open}}}, \qquad \widehat{\text{cost}}.$$

Quantiles are computed via a streaming sketch (e.g., q-digest) to bound memory. A tail SLO is evaluated as

$$\Pr[T_{\text{first}} \le \theta] \;\ge\; 0.95,$$

with alerts (dev-only) when violated persistently.

15 Reliability & Resilience

Network Events & Policies (utils/resilience/*)

Let the transport emit network-side events

$$\mathcal{N} = \{\mathsf{net{:}online},\ \mathsf{net{:}offline},\ \mathsf{timeout{:}open},\ \mathsf{timeout{:}idle}\}.$$

Policies are parameterized by deadlines $(T_{\text{open}}, T_{\text{idle}})$ and a bounded backoff $(N, t_0, r, t_{\max})$ (cf. Sec. 4). The k-th retry delay is

$$t_k = \min(t_{\max},\, t_0\, r^{\,k-1}) + \xi_k, \qquad \xi_k \sim \mathcal{U}[0, \varepsilon].$$

If attempt k succeeds with probability p_k (conditionally independent), the success probability within N attempts is

$$P_{\le N} = 1 - \prod_{k=1}^{N} (1 - p_k).$$

The expected added waiting time (conditioned on eventual success at attempt J) is bounded by

$$\mathbb{E}[W \mid J] \;\le\; \sum_{k=1}^{J-1} \big(\min(t_{\max}, t_0\, r^{\,k-1}) + \tfrac{\varepsilon}{2}\big).$$

Offline tap. On net:offline the adapter enters a *quiescent* mode: new `SEND` requests are rejected with a classified network error; existing streams are aborted (Sec. 3.1). Upon net:online normal dispatch resumes.

Error Taxonomy & Classification

Let the raw error space be $E = E_{\text{http}} \cup E_{\text{transport}} \cup E_{\text{schema}} \cup E_{\text{auth}} \cup E_{\text{quota}}$. A total classifier

$$\kappa : E \to \{\mathsf{network}, \mathsf{auth}, \mathsf{quota}, \mathsf{protocol}, \mathsf{timeout}, \mathsf{user\text{-}abort}\}$$

drives deterministic UI outcomes. Define the retryable set

$$\mathcal{E}_{\text{retry}} = \{\, 429 \,\} \cup \{\, 5xx \setminus \{501, 505\} \,\} \cup E_{\text{transport}} \setminus \{\mathsf{TLS\text{-}fatal}\}.$$

Mapping (excerpt):

$$\kappa(401) = \kappa(403) = \mathsf{auth}, \quad \kappa(429) = \mathsf{network}, \quad \kappa(408) = \mathsf{timeout},$$
$$\kappa(\mathsf{AbortController}) = \mathsf{user\text{-}abort}, \quad \kappa(\mathsf{JSON\text{-}parse}) = \mathsf{protocol}.$$

Policy. If $e \in \mathcal{E}_{\text{retry}}$ and attempts $< N$, schedule retry; else emit terminal `ERROR`. All terminals are exactly-once per request.

Abort Semantics

Abort sources:

$$\mathcal{A} = \{\mathsf{user},\ \mathsf{supersede(new\text{-}request)},\ \mathsf{timeout{:}open},\ \mathsf{timeout{:}idle},\ \mathsf{provider\text{-}abort},\ \mathsf{unmount}\}.$$

Let state $s \in \{\texttt{idle}, \texttt{sending}, \texttt{streaming}\}$ (Sec. 3.1). The reducer satisfies idempotent abort:

$$\forall s \;\; \delta(s, \texttt{ABORT}) = \texttt{idle}, \qquad \delta(\texttt{idle}, \texttt{ABORT}) = \texttt{idle}.$$

Source	**Precondition**	**Effects**
user	$s \neq \texttt{idle}$	cancel transport; clear timers/buffer; UI toast
supersede	new `SEND`	abort old rid; start new rid
timeout:open	$s = \texttt{sending}$, $\Delta t > T_{\text{open}}$	classify timeout; abort
timeout:idle	$s = \texttt{streaming}$, $\Delta t > T_{\text{idle}}$	classify timeout; abort
provider-abort	adapter signal	propagate `ABORT`
unmount	teardown	abort if $s \neq \texttt{idle}$; drop telemetry

Race-freedom. All aborts are version-guarded by rid; non-matching events are ignored (Sec. 5).

16 Security & Safety Considerations (as implemented)

API Key Handling

Keys live only in local settings; there are no server-side secrets. A redactor

$$R : \ \mathsf{Secret} \to \mathsf{Redacted}, \qquad R(s) = \texttt{head}(s) \parallel \texttt{****} \parallel \texttt{tail}(s)$$

is applied at the UI boundary and in logs. **Invariants:** (i) idempotence $R(R(s)) = R(s)$; (ii) non-invertibility in UI; (iii) no plaintext keys in telemetry.

Editor Size Guards

The apply operator Φ enforces $L_{\max}$ (Sec. 9):

$$\|\Delta\| \le L_{\max} \Rightarrow \mathsf{commit}(\Delta), \qquad \|\Delta\| > L_{\max} \Rightarrow \mathsf{confirm\_and\_chunk}(\Delta).$$

Safety. (No-Write-On-Decline) If the user rejects confirmation, the model state is unchanged; history/undo remain consistent.

HTML Sanitization for Preview

A sanitizer

$$H = \mathsf{strip\_disallowed} \circ \mathsf{normalize} \circ \mathsf{capLen} : \ \mathsf{HTML}_{\text{in}} \to \mathsf{HTML}_{\text{out}}$$

removes scriptable nodes/attributes and caps size. Properties:

$$H(H(x)) = H(x) \quad \text{(idempotent);} \qquad |H(x)| \le L_{\text{preview\_max}};$$

$$\text{if } x \text{ contains } \langle \text{script} \rangle \Rightarrow \langle \text{script} \rangle \notin H(x).$$

Provider Tests with User-Supplied Keys

Connectivity tests are executed only when the user supplies keys and opts in. Endpoints are fixed; no secrets are embedded in the build. Errors are classified via κ (Sec. 15).

17 Performance Characteristics

Asymptotic Costs

- **Virtual list.** Per-frame layout $O(|R|)$ for render set R (Sec. 7); height maintenance amortized via ResizeObserver.
- **Streaming updates.** Buffer appends $O(\sum_j \|\Delta_j\|)$; RAF-batched commits reduce DOM mutations from per-chunk to per-frame (≤ 1 commit/frame in steady state).
- **Token estimate.** For k slices, $\ell_i \approx \lceil |c_i|/4 \rceil$ costs $O(k)$; sort $O(k \log k)$ if priority reordering applies (Sec. 3.3).
- **FSM.** Transition evaluation is $O(1)$; deadline checks are timer-driven (Sec. 3.1).
- **Search.** Off-main-thread line scan is $O(\sum_i |\mathsf{text}_i|)$; snippet assembly $O(|W_j|)$.

Throughput–Latency Envelope

Let $r(t)$ be arrival rate (chars/s) and $c(t)$ commit capacity. The buffer obeys

$$\frac{d}{dt}\|B(t)\| \le r(t) - c(t) \quad \text{with drains at } t \in \mathcal{T}_{\text{frame}}.$$

If $\bar{r} < \bar{c}$ then $\sup_t \|B(t)\| < \infty$ (bounded backlog). First-delta latency satisfies

$$T_{\text{first}} \approx T_{\text{open}} + \delta_{\text{net}} + \delta_{\text{RAF}}, \qquad \delta_{\text{RAF}} \in [0, \tfrac{1}{f_{\text{display}}}],$$

so $\delta_{\text{RAF}} \le 16.7$ ms at 60 Hz. Bounded retry increases completion time by at most $\sum_{k=1}^{N-1} t_k$ in the worst case (before a success/terminal).

Memory & GC Considerations

Message text is appended in-place to a single buffer per streaming node; buffers are nulled on `done`/`error`/`abort`. Virtualization restricts live DOM nodes to $|R|$; code-fence widgets are upgraded in-place, avoiding additional allocations.

Instrumentation Targets

We report

$$\mathbf{T} = (T_{\text{open}}, T_{\text{first}}, T_{\text{done}}), \qquad \widehat{r} = \frac{\sum_j \|\Delta_j\|}{T_{\text{done}} - T_{\text{open}}}.$$

SLO: $\Pr[T_{\text{first}} \le \theta] \ge 0.95$. Regressions are flagged when the empirical $q_{0.95}$ of T_{first} exceeds θ over a sliding window.

18 Limitations

Heuristic, Model-Agnostic Token Estimates

Context budgeting uses a tokenizer-agnostic length proxy $\ell(c) \approx \lceil |c|/4 \rceil$. This proxy is *not* a hard upper bound on provider tokens and can err in either direction depending on language, whitespace, and provider-specific byte-pair rules. Formally, let $t_\star(c)$ be the true token count under the deployed model. Then there exist residuals $\varepsilon(c)$ s.t.

$$t_\star(c) = \ell(c) + \varepsilon(c), \qquad \varepsilon(c) \in \mathbb{Z}.$$

While the deterministic fit guarantees $\sum_{c \in S} \ell(c) \le B$, it does not guarantee $\sum_{c \in S} t_\star(c) \le B$. Practical mitigations (usage echo, provider caps) reduce but do not eliminate this approximation gap.

Single-Request Streaming

For simplicity and predictability, the control plane enforces $\#\{\text{activeRequest}(t)\} \leq 1$ at all times. This excludes: (i) concurrent multi-model fan-out, (ii) speculative decoding with early-exit, and (iii) interleaved tools/stream co-scheduling. The benefit is stronger UI determinism; the cost is the absence of concurrency-level speedups in favorable network conditions.

Client-Side Secrets Only

API keys live exclusively in local settings. There is no server-side secret management, proxying, or key rotation service in scope. Implications: (i) users must provision and rotate keys themselves; (ii) enterprise policies (e.g., IP-allowlisting, vault auditing) are not addressed by the client; (iii) telemetry is dev-only and redacted, with no authoritative billing.

Determinism over Optimality

The budgeting policy is greedy and deterministic, not globally optimal (knapsack) under the true tokenizer. The streaming flush policy favors frame-synchronous smoothness over minimal latency for each individual character. The editor apply guard ($L_{\max}$) is a coarse safety bound and may block large but safe patches that would fit with finer-grained diffs.

UI & Render Constraints

The virtual list assumes measurable, positive heights and stable font metrics. Exotic content (e.g., very large inline images injected mid-stream) can cause height churn and temporary scroll jitter despite debouncing.

Observability Scope

Tracing, usage and cost estimates are intended for development and tuning. They are approximate, model-agnostic and may diverge from provider-reported billing. No SLA enforcement or autoscaling feedback loop is implemented.

19 Future Work (Application-Aligned)

Tokenizer-Aware Budgeting

Replace $\ell(c)$ with $t_\star(c)$ from a model-specific tokenizer:

$$\ell(c) \;\rightsquigarrow\; t_\star(c) = \text{Tok}_\star(c).$$

Introduce a calibrated upper envelope $\bar{t}(c)$ with a confidence level $\Pr[t_\star(c) \le \bar{t}(c)] \ge 1 - \alpha$ to keep hard guarantees while reducing over-truncation. Learn $\bar{t}$ via piecewise-linear regression on $(|c|, t_\star(c))$ pairs with per-language features.

Guarded Multi-Step Tool/Action Plans

Formalize tool plans as sequences with pre/post conditions:

$$\Pi = \langle (a_i,\ \mathrm{pre}_i,\ \mathrm{post}_i) \rangle_{i=1}^{m}.$$

Define a verifier V that checks pre_i against the current IDE state and simulates post_i under size and selection guards before execution. Abort on first violation; surface a minimal counterexample (failed predicate, offending range, and proposed Δ).

Search Worker Ranking & Snippet Weighting

Augment boolean line matches with ranking:

$$\mathrm{score}(f, j) = \alpha \cdot \mathrm{BM25}(q, \ell_j) + \beta \cdot \mathrm{sim}(q, \ell_{W_j}) + \gamma \cdot \mathrm{path\_prior}(f),$$

where ℓ_{W_j} is the snippet window and $\alpha, \beta, \gamma \ge 0$. Emit results in non-increasing score, stable within equal-score buckets.

Multi-Stream Arbiter with Priorities

Allow $K > 1$ concurrent streams with a preemptive arbiter:

$$\text{WFQ over } \{1, \ldots, K\}, \quad \text{quantum } q_k \propto \mathrm{prio}_k.$$

Guarantee fairness and tail bounds per priority class; preempt when a new high-priority request arrives. UI isolates streams by thread and enforces per-thread pinning to avoid cross-scroll jumps.

Editor-Safe Semantic Patching

Replace raw Δ insertion with syntax-aware diffs:

$$\Phi : (\mathsf{AST},\ \Delta) \rightarrow \mathsf{AST}',$$

with node-level guards (scope, imports, diagnostics) and a fallback to textual chunking when parsing fails. Expose a dry-run mode that previews AST changes and their textual rendering before commit.

Streaming Enhancements

Coalesce micro-deltas under a maximum inter-commit budget ($\leq$ one commit per frame or per X ms, whichever is later). Introduce optional grammar-constrained decoding (when supported) to reduce invalid intermediate tokens and improve fence stability.

Security Hardening

Add optional remote keystore integration (e.g., OAuth device flow) so keys never touch persistent local storage. Introduce per-origin CORS pinning and a subresource integrity (SRI) policy for provider script samples in tests.

20 Reproducibility Checklist

Execution Steps

1. ☐ Install Node runtime and lock dependencies (`npm ci` with a committed lockfile).
2. ☐ Build SPA in production mode (`NODE_ENV=production`).
3. ☐ Serve over HTTPS (required by some providers for SSE).
4. ☐ Verify browser GPU compositing is enabled (smooth scrolling).

Environment Variables & Keys

1. ☐ Set provider API key(s) in local settings (UI), not in code.
2. ☐ Optionally pass `?provider=.. & model=..` to override defaults.
3. ☐ Confirm redaction in logs (keys appear as `head****tail`).

Feature Flags & Locations

1. ☐ Streaming toggle (Assistant Settings → Streaming).
2. ☐ Safety level / truncation behavior (Assistant Settings).
3. ☐ Provider self-test (Assistant Settings → Test).

Tracing / Telemetry

1. ☐ Enable dev telemetry endpoint (env or querystring).
2. ☐ Confirm per-request milestones ($T_{\text{open}}, T_{\text{first}}, T_{\text{done}}$) are recorded.
3. ☐ Validate usage echo when available; compare against $\widehat{u}$.

Thread Export / Import

1. ☐ Export the current thread (JSON) from the Assistant.
2. ☐ Import on a clean session; verify identical render and ordering.
3. ☐ Check that pin-to-bottom and code fences behave identically.

Configuration Constants (Match Application Settings)

1. ☐ Token budget B (context compiler). Record numeric value.
2. ☐ Editor apply threshold $L_{\max}$. Record numeric value.
3. ☐ Deadlines $(T_{\text{open}}, T_{\text{idle}})$ for resilience.
4. ☐ Overscan o for virtual list.
5. ☐ Retry policy $(N, t_0, r, t_{\max})$.

Determinism Checks

1. ☐ Same inputs $\Rightarrow$ identical composed prompt (byte-equal).
2. ☐ Same event order $\Rightarrow$ identical visible message.
3. ☐ Superseding `SEND` aborts prior stream (no cross-contamination).

A Notation Summary

B	Token budget for context composition.
$C = \{c_i\}$	Context slices (strings) considered for inclusion.
ℓ_i	Length estimate for c_i (token proxy), $\approx \lceil \lvert c_i \rvert / 4 \rceil$.
w_i	Priority weight for c_i, $w_i \in \mathbb{R}_{\geq 0}$.
$E = \langle e_j \rangle$	Stream event sequence in time order.
Δ_j	Text fragment payload of a `DELTA` event.
$M(t)$	Visible message at time t (concatenated prefix of Δ_j).
$B(t)$	In-memory buffer awaiting flush at time t.
$\text{flush}(t_f)$	Frame-synchronous buffer commit at frame time t_f.
$\mathcal{M} = (S, \Sigma, \delta, s_0, F)$	Conversation FSM: states, alphabet, transition, initial and terminal sets.
S	FSM states $\{\texttt{idle}, \texttt{sending}, \texttt{streaming}\}$.
Σ	FSM alphabet $\{\texttt{SEND}, \texttt{OPEN}, \texttt{DELTA}, \texttt{USAGE}, \texttt{DONE}, \texttt{ERROR}, \texttt{ABORT}\}$.
Φ	Editor apply operator (insert/replace with size guards).
$L_{\max}$	Editor safety limit (max insert/replace length before chunk/confirm).
$\mathbf{T}$	Latency milestones $(T_{\text{open}}, T_{\text{first}}, T_{\text{done}})$.
$\hat{r}$	Throughput estimate $\frac{\sum_j \lVert \Delta_j \rVert}{T_{\text{done}} - T_{\text{open}}}$.
o	Overscan margin (pixels) for the virtual list render window.
$(N, t_0, r, t_{\max})$	Retry/backoff parameters (attempts, base delay, factor, cap).

B Recommended Sources

Streaming, Networking, and Browser Primitives

- MDN Web Docs. "Server-Sent Events (EventSource)." https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events
- WHATWG. "Fetch Standard." https://fetch.spec.whatwg.org/
- WHATWG. "Streams Standard." https://streams.spec.whatwg.org/
- MDN Web Docs. "AbortController." https://developer.mozilla.org/en-US/docs/Web/API/AbortController
- MDN Web Docs. "requestAnimationFrame." https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame

Retry, Backoff, and Resilience

- Marc Brooker. "Exponential Backoff And Jitter." AWS Builders Library. https://aws.amazon.com/builders-library/timeouts-retries-and-backoff-with-jitter/
- Google SRE. "Handling Overload." In: Site Reliability Engineering (O'Reilly). https://sre.google/sre-book/handling-overload/

Tokenization and Prompt Budgeting

- T. Kudo and J. Richardson. "SentencePiece: A Simple and Language Independent Subword Tokenizer and Detokenizer." arXiv:1808.06226. https://arxiv.org/abs/1808.06226
- R. Sennrich, B. Haddow, A. Birch. "Neural Machine Translation of Rare Words with Subword Units." arXiv:1508.07909. https://arxiv.org/abs/1508.07909
- M. Schuster, K. Nakajima. "Japanese and Korean Voice Search." 2012 IEEE ICASSP. (WordPiece). https://ieeexplore.ieee.org/document/6289079

Editor and IDE Foundations

- Monaco Editor. Official documentation. https://microsoft.github.io/monaco-editor/
- Visual Studio Code API. Text model and edit operations. https://code.visualstudio.com/api/references/vscode-api

State Management and React Integration

- Zustand (pmndrs) documentation. https://docs.pmnd.rs/zustand/getting-started/introduction
- React Docs. "Optimizing Performance" and "UI Patterns." https://react.dev/
- TanStack Virtual (virtualization utilities). https://tanstack.com/virtual
- react-window (windowed lists). https://github.com/bvaughn/react-window

Virtualization, Rendering, and UI Throughput

- Chrome Developers. “Rendering Performance.” https://developer.chrome.com/docs/devtools/performance/
- MDN Web Docs. “ResizeObserver.” https://developer.mozilla.org/en-US/docs/Web/API/ResizeObserver

Search, Ranking, and Snippets

- C. D. Manning, P. Raghavan, H. Schütze. *Introduction to Information Retrieval.* Cambridge University Press, 2008. https://nlp.stanford.edu/IR-book/
- S. E. Robertson, H. Zaragoza. “The Probabilistic Relevance Framework: BM25 and Beyond.” *Foundations and Trends in IR*, 2009. https://doi.org/10.1561/1500000019

Telemetry, Tracing, and Observability

- OpenTelemetry. “Specification.” https://opentelemetry.io/docs/specs/
- Google SRE. “Monitoring Distributed Systems.” In: *SRE Workbook*. https://sre.google/workbook/

Security, Safety, and Privacy

- OWASP. “Cheat Sheet Series” (CSP, XSS, secrets handling). https://cheatsheetseries.owasp.org/
- W3C. “Content Security Policy Level 3.” https://www.w3.org/TR/CSP3/
- W3C. “Subresource Integrity.” https://www.w3.org/TR/SRI/

Accessibility and Text Semantics

- W3C WAI-ARIA 1.2. https://www.w3.org/TR/wai-aria-1.2/
- W3C. “Web Content Accessibility Guidelines (WCAG) 2.1.” https://www.w3.org/TR/WCAG21/
- CommonMark Spec (Markdown). https://spec.commonmark.org/

Color, Contrast, and Theming

- Björn Ottosson. “OKLab and OKLCH Color Spaces.” https://bottosson.github.io/posts/oklab/
- W3C. “Contrast (Minimum) Success Criterion.” WCAG 2.1. https://www.w3.org/TR/WCAG21/#contrast-minimum

Standards and Platform References

- WHATWG. “HTML Living Standard” (EventSource, workers). https://html.spec.whatwg.org/

- MDN Web Docs. “Web Workers API.” https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API