Live Caster

A Real-Time Screen Narrator for Blind and Low-Vision Users

Yuvraj Gupta
San Francisco, CA

yuvrajgupta1808@gmail.com
+1 (415) 740-8804
linkedin.com/in/yuvraj2004
github.com/YuvrajGupta1808
yuvrajgupta.com

December 2025

Contents

What Live Caster Does

Live Caster is a deployed, authenticated web application that watches a shared screen and speaks what is on it. A user signs in, presses one button, and chooses a window; within about a second the system begins narrating, and it keeps narrating as the screen changes.

It is built for blind and low-vision users. Screen readers announce the element under the cursor; they do not tell you that a dialog just appeared, that a download finished, or that the error at the top of the page is the reason a form will not submit. Live Caster describes the screen as a whole and says which part of it matters now.

The session is a conversation rather than a feed. The user can speak at any moment: the narration stops mid-word, the question is answered, and narration resumes. Everything both sides say appears as live captions.

The project won third prize at the Google DeepMind and Cerebral Valley Gemini Hackathon, and was subsequently taken to a public production deployment.

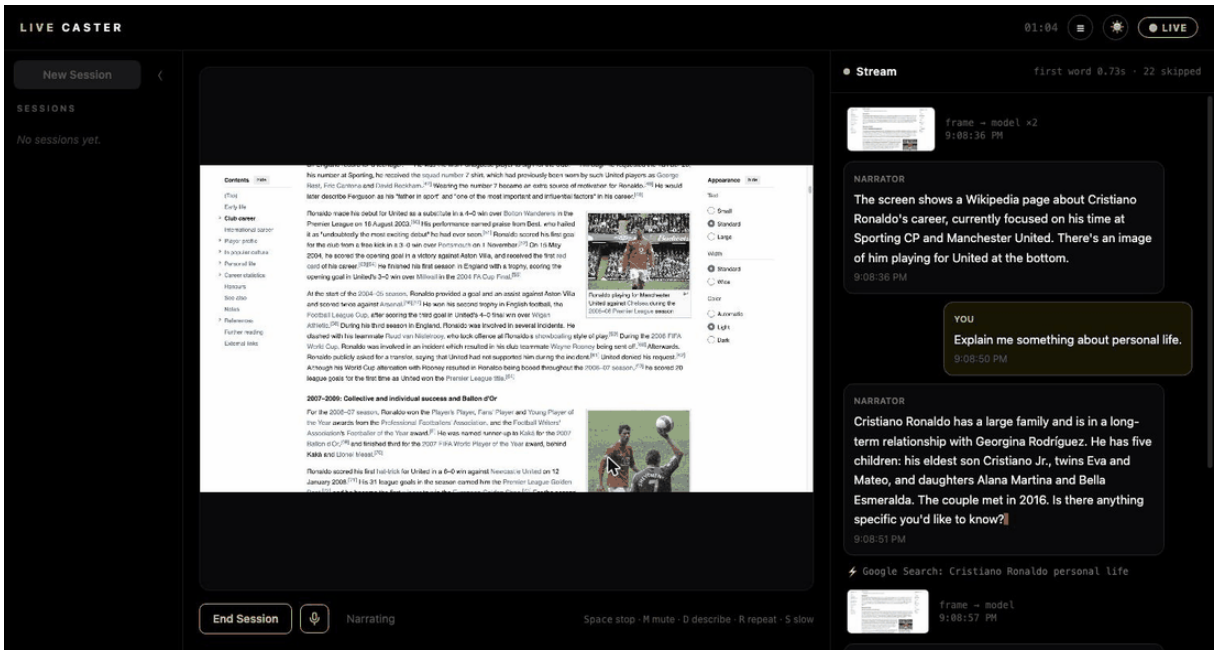

Figure 1: A live session. The narrator has described the page; the user has interrupted by voice to ask a follow-up question; the model has answered and run a Google Search, shown inline in the transcript. The header reports time to first word and the number of frames skipped by the change filter.

Why It Is Necessary

The conventional way to build this is a pipeline: capture a frame, caption it with a vision model, hand the text to a speech service, play the resulting audio. Each stage adds latency, and the total is high enough that the description arrives after the moment it describes.

Worse, a pipeline cannot be interrupted. Audio generated ahead of time must either play to completion or be discarded; there is no mechanism for a user to speak over it and be understood, because the component that produced the speech has already finished its work and is not listening.

Live Caster removes the middle of that pipeline. The model that sees the screen is the model that speaks, and its voice arrives as audio while it is still forming the sentence. Because a single bidirectional session carries both the screen and the microphone, interruption is a property of the session rather than a feature bolted onto it.

That single decision — one session, both directions, no separate speech step — is what makes the system feel like a person watching alongside you, and it is the idea the rest of the platform is built to support.

Infrastructure

Live Caster runs as two Cloud Run services in `us-central1` with authentication in front of every route. Infrastructure decisions came first because the application spends money per second of use: an ordinary web app that gets this wrong leaks a filesystem, but this one leaks a budget.

Deployment

The backend is a FastAPI application deployed from source; the frontend is a static Vite build served by nginx behind a custom domain. Both scale to zero, so an idle deployment costs nothing.

```
gcloud run deploy live-caster-backend --source .
gcloud builds submit --config frontend/cloudbuild.yaml
```

The frontend's API and WebSocket URLs are baked in at build time rather than read at runtime, so changing them requires a rebuild rather than a redeploy. This is recorded in the deployment log because it is easy to forget.

The custom domain is attached with a Cloud Run domain mapping and a single CNAME record. A global external load balancer is the textbook approach and would have cost roughly eighteen dollars a month before serving a request; the domain mapping does the same job for nothing.

Authentication

Every route is gated. Sign-in uses Firebase email-link authentication: the user receives a one-time link and no password is stored or transmitted.

The verified Firebase ID token is attached as a bearer header on REST calls and inside the WebSocket start message. The backend verifies it on every request and on the socket handshake before a Gemini session is opened, so an unauthenticated socket cannot reach the model.

Sign-up is deliberately open rather than restricted to an allowlist, which keeps the application demonstrable. The spending controls below are what make that choice safe.

Cost Control

Because any signed-in user can start a session that spends money, the monthly budget is enforced by a mechanism rather than an alert.

- **Budget notifications.** A fifty-dollar monthly budget publishes to a Pub/Sub topic at every threshold crossing, not only at the end.
- **Automatic hard stop.** A Cloud Function compares reported spend against the budget and, at parity, unlinks the billing account from the project outright.
- **Least privilege.** That function runs as a dedicated service account holding `roles/billing.projectManager` and nothing else.

- **Verified partially.** The under-budget path was tested live with a synthetic notification and confirmed correct in logs. The disable path was not fired, because verifying it would have caused the outage it exists to prevent.

The consequence is deliberately severe: every billed service stops at once and must be restored by hand. A narrower option that revoked only model access was considered and rejected, because against an open sign-up form the strongest guarantee was worth more than graceful degradation.

Continuous Integration

GitHub Actions runs the backend test suite and a production frontend build on every push and pull request. The tests cover the WebSocket contract, the authentication guards, and the session bridge under a mocked model client, so the suite runs without credentials or model spend.

Backend

The backend is a FastAPI application organised into a live bridge, an authentication layer, session storage, and prompt definitions. Roughly 1,000 lines of Python.

Session Bridge

One WebSocket from the browser maps to one Gemini Live session for the entire broadcast. Because the session is long-lived, the model retains everything it has already said, so narration continues a story rather than emitting disconnected captions.

The bridge runs two concurrent tasks against that session.

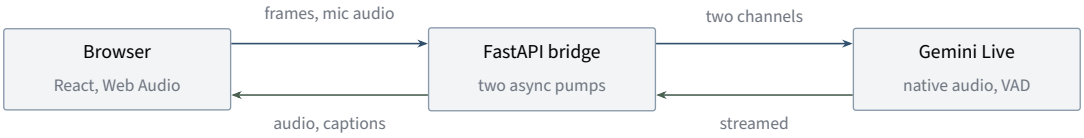

Figure 2: One WebSocket from the browser to the bridge, one Gemini Live session beyond it, for the whole broadcast. Both legs carry traffic in both directions simultaneously.

Task	Direction	Responsibility
Client pump	Browser to model	Forwards frames and microphone audio, routing each to the correct channel
Model pump	Model to browser	Relays audio, transcription of both speakers, tool activity, and turn markers

The two are joined with `FIRST_COMPLETED`, so whichever side terminates first tears the other down cleanly. Shared state between them is what makes interruption behave correctly, and it is deliberately small: whether the model is currently speaking, the most recent held-back frame, and the latest session resumption handle.

Binding the Frame to the Turn

An early version sent the image over the realtime channel and a text instruction separately. The model would answer the instruction without attending to the image, producing fluent narration unanchored to the actual screen.

For an accessibility tool this is the worst available failure: the user cannot check the description against the screen, so a confident invention is indistinguishable from a correct reading.

The fix was structural rather than a prompt adjustment. The frame and the instruction are sent as two parts of a single content turn, so the image is bound to the request it answers.

```
await session.send_client_content(
    turns=types.Content(
        role="user",
        parts=[
            types.Part(inline_data=types.Blob(data=jpeg_bytes,
                                              mime_type="image/jpeg")),
            types.Part(text=FRAME_NUDGE),
        ],
    ),
    turn_complete=True,
)
```

Two Upstream Channels

Not every frame should produce speech. While the user is talking, the screen must stay in the model's context without triggering a narration turn that competes with the answer they asked for.

These are different intents and the API expresses them differently. Frames that should produce speech go through `send_client_content` with the turn marked complete. Frames that are context only go through `send_realtime_input`, alongside the microphone audio, where voice-activity detection drives barge-in.

When the user interrupts, any frame waiting to be narrated is discarded as stale and the conversation takes priority.

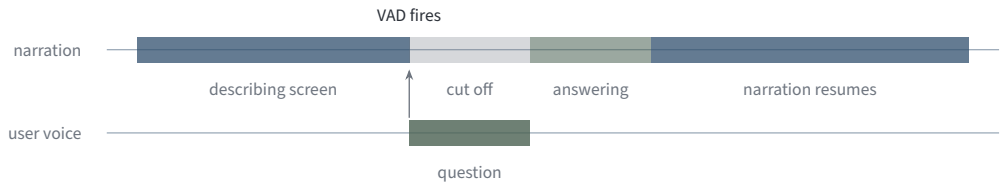

Figure 3: Barge-in. Voice-activity detection on the realtime channel cuts the narration mid-sentence, the question is answered, and narration resumes. Any frame that arrived during the exchange is discarded as stale.

Pacing

At one frame every two seconds, a naive implementation interrupts its own narration continuously and never finishes a sentence. Queueing frames instead is worse: narration falls behind and begins describing a screen that has since changed.

A shared flag holds incoming frames while a sentence is in flight, retaining only the newest. When the turn completes, that single freshest frame is sent and the stale ones are dropped. Only the user's voice interrupts a sentence in progress.

Session Resumption

A long-lived streaming session will drop. Reconnecting naively loses the model's memory, and it begins re-describing a screen it already narrated, which is immediately obvious to the listener.

The bridge requests resumption handles as the session runs and re-attaches with the most recent handle on failure, bounded at five attempts. A browser disconnect is distinguished from a model failure, so a user closing the tab does not trigger reconnection.

Frontend

The interface is a React and Vite application with Tailwind styling, using the Web Audio API to play raw 24kHz PCM as it streams rather than waiting for a complete clip. Roughly 950 lines.

Reducing Work Before It Is Sent

Two filters run in the browser, because the cheapest inference call is the one never made.

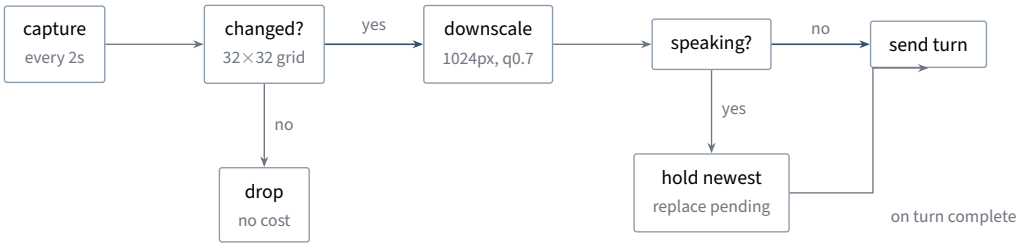

Figure 4: The path a captured frame takes. Two gates stand before the model: an unchanged frame is dropped without cost, and a frame arriving mid-sentence is held rather than queued, replacing any frame already waiting.

- **Change detection.** Each frame is drawn to a 32 by 32 grayscale grid and compared against the previous one. A mean per-pixel delta under six counts as unchanged and the frame is dropped before it reaches the network.
- **Downscaling.** Surviving frames are scaled to 1024 pixels wide at JPEG quality 0.7. Gemini tiles images at 768 pixels square, so a 4K frame costs fifteen tiles and this one costs two.

Frame source	Tiles	Approximate tokens
Raw 4K capture	15	3,870
Downscaled to 1024px	2	516

Sessions also end themselves after five minutes without activity, and returning to the cover page tears the session down immediately, so a forgotten tab cannot narrate an empty room overnight.

Observability

The interface displays time to first spoken word for every line it produces, alongside a running count of frames skipped by the change filter. Both the latency and the efficiency are therefore visible as measurements rather than claims.

Operable Without Sight

An accessibility tool that requires a mouse is a contradiction, so every control has a single-key binding.

Key	Action
Space	Start or stop the session
M	Mute the microphone
D	Describe the screen now, bypassing the change filter
R	Repeat the last spoken line

A slow-speech toggle and a light and dark theme switch sit alongside these. A demo mode at `/?demo=1` replaces the screen-share picker with a synthetic animated canvas and runs the full pipeline against it, which makes the system reviewable and recordable without granting screen access to anything.

Model Layer

Provider Configuration

The model is reached through Vertex AI using the Cloud Run service account's application-default credentials, rather than an API key.

Setting	Value
Model	`gemini-live-2.5-flash-native-audio`
Region	`us-central1`
Authentication	Service account, `roles/aiplatform.user`
Response modality	Native audio, 24kHz PCM

Narration Prompt

Two instructions in the system prompt carry most of the safety weight.

The first forbids invention: describe only what is visible in the current frame, and say so explicitly when something cannot be made out. The second forbids repetition: react to what changed rather than restating the screen, which is only possible because session memory spans the whole broadcast.

Narration lines are held to roughly fifteen words, twenty-five at the outside, so the voice stays ahead of a changing screen.

Tool Use

The model is given a Google Search tool for real-world questions the user asks, and is explicitly barred from using it to infer screen contents, since the screen already arrives as images. Searches that do run surface inline in the transcript, so it is always visible when an answer came from the web rather than from the window.

Quiet Mode

An opt-in second system prompt inverts the default behaviour: stay silent through ordinary activity and speak only for things that need attention, such as errors, alerts, security prompts, and completed downloads. The model still answers immediately when spoken to.

This converts the tool from a narrator into a monitor without changing any pipeline code.

Measured Latency

Measured on Vertex AI in `us-central1` across three consecutive frames of a live session.

Metric	Result
Fastest time to first word	0.58 s
Slowest time to first word	1.09 s
Audio format	Native 24kHz PCM, streamed during generation
Transport	One WebSocket session end to end

Time to first word is the metric that matters for this product, because it measures the gap between something happening on screen and the user hearing about it. It is displayed live in the interface for every line.

Summary

Live Caster is a production-deployed, authenticated application that narrates a live screen and holds a spoken conversation about it, built as an accessibility tool for blind and low-vision users.

- **Infrastructure.** Two Cloud Run services scaling to zero, Firebase-gated routes, a custom domain without a load balancer, and a monthly budget enforced by automatic billing shutdown rather than an alert.
- **Backend.** One long-lived Gemini Live session per broadcast, two concurrent pumps with shared interruption state, frames bound to their turns, and invisible reconnection through session resumption.
- **Frontend.** Client-side change detection and downscaling before any inference, streaming PCM playback, full keyboard operation, and latency displayed as a measurement.
- **Model layer.** Native audio over Vertex AI with no stored credential, prompt constraints that treat invention as the primary harm, fenced tool use, and a measured 0.58 to 1.09 second time to first word.

Three of the four hardest problems in this project — the model ignoring the image, narration interrupting itself, and disorienting reconnection — were found by using the system rather than by reading documentation, and each was fixed at the protocol layer rather than patched in the prompt.

The engineering position throughout is that a real-time system is judged by its behaviour under interruption and failure, not by its behaviour when everything arrives on time.