OCRBit: Rebuilding Screenshot OCR with Python, a Browser, and a Multimodal Model

A technical tour of OCRBit's lightweight Windows architecture, from global hotkeys and multi-monitor capture to browser-based selection, SSE streaming, formula-aware prompts, and DPAPI-protected credentials.

July 27, 2026 • OCR • Python • React • Windows • Multimodal AI

OCRBit is a screenshot OCR tool for Windows. Its workflow looks familiar: press a shortcut, select part of the screen, copy the recognized text, and optionally translate, summarize, or explain it. Its implementation is less conventional. It uses neither Electron nor a resident desktop GUI. Instead, it divides the application into three layers:

```
Win32 / Python
  ├─ Register the global hotkey
  ├─ Capture the monitor under the pointer
  ├─ Crop screenshots and manage temporary data
  └─ Host a local HTTP service
              |
              ▼
React / system browser
  ├─ Select a region inside a Kiosk window
  ├─ Render Markdown, tables, and KaTeX
  └─ Consume streamed SSE output
              |
              ▼
SiliconFlow API
  └─ Qwen3.5-397B-A17B multimodal OCR and text processing
```

This design is not only about reducing the installer size. It reuses capabilities already supplied by Windows and the browser: Python handles system calls and secrets, the browser handles interaction and typography, and a remote multimodal model interprets the image. This article examines OCRBit `v1.0.0` at its first public commit.

From shortcut to screenshot: a Win32 message loop

The entry point verifies the platform, then attempts this call:

```
ctypes.windll.user32.SetProcessDpiAwarenessContext(ctypes.c_void_p(-4))
```

`-4` is `DPI_AWARENESS_CONTEXT_PER_MONITOR_AWARE_V2`. This matters on a multi-monitor setup with different scaling factors: without per-monitor DPI awareness, Windows may virtualize coordinates and introduce a mismatch between pointer positions, window dimensions, and screenshot pixels.

Rather than using a third-party hotkey package, `hotkey.py` registers the fixed `Ctrl+Shift+A` shortcut through `RegisterHotKey`. A daemon thread blocks on `GetMessageW`; when it receives `WM_HOTKEY`, it starts another thread for the capture callback. The message loop therefore remains responsive while a screenshot is being processed.

Startup includes a small handshake. A `threading.Event` reports whether registration succeeded, and the main thread waits for up to two seconds. If another application already owns the shortcut, OCRBit closes the newly created HTTP server and exits instead of remaining in a state where it appears to run but cannot capture anything.

Multi-monitor capture and the in-memory store

The capture module obtains the pointer's virtual-desktop coordinates with `GetCursorPos`, then searches the physical monitors reported by `mss`:

```
monitor = next(
    m for m in monitors
    if m["left"] <= point.x < m["left"] + m["width"]
    and m["top"] <= point.y < m["top"] + m["height"]
)
```

OCRBit captures the **entire monitor currently under the pointer**, not a stitched image of the whole virtual desktop. A monitor may sit to the left of or above the primary display, so its `left` or `top` coordinate may be negative. Those coordinates are later reused to position the Kiosk window on the same display.

`CaptureStore` encodes the capture as PNG bytes and stores it with its dimensions, screen origin, and creation time in a process-local dictionary. The key comes from `secrets.token_urlsafe(18)`, and dictionary access is protected by a lock. No temporary screenshot file is created; the bytes disappear when the process exits.

The cleanup behavior deserves precise wording. Whenever a new capture is created, the store removes entries older than one hour. There is no periodic cleanup thread. If no later capture occurs, an old entry can remain in memory for more than an hour, until either another capture triggers cleanup or the application exits.

Why selection happens in a browser

After capture, the backend looks for Edge or Chrome and launches a separate `--kiosk --new-window` process. The window position and size come from the captured monitor's bounds. Its profile lives at `%LOCALAPPDATA%\OCRbit\browser-profile`, separate from the user's everyday browser profile. OCRBit falls back to the default browser only when no supported executable is found.

The Kiosk page opens:

```
http://127.0.0.1:8765/capture/<capture_id>
```

During selection, the React workspace and screenshot both fill `100vw × 100vh`. Dragging produces CSS-pixel coordinates. Before requesting recognition, the frontend converts them back into source-image pixels:

```
const scaleX = image.naturalWidth / image.clientWidth
const scaleY = image.naturalHeight / image.clientHeight
```

The selection code therefore does not assume that browser coordinates will always equal screenshot pixels. The backend clamps the received origin, width, and height to the image boundary and guarantees a crop of at least 1×1 pixels, preventing malformed parameters from reading beyond the screenshot.

When recognition is complete, the page can post to `/api/captures/<id>/close`. The backend waits 150 ms for the response to reach the browser, then invokes `taskkill /T /F` on the process tree it created for this capture. It does not close the user's ordinary browser windows.

A local backend without a Web framework

The backend uses Python's standard-library `ThreadingHTTPServer` instead of Flask or FastAPI. It serves the Vite-produced `web-dist` assets and implements the local API:

Route	Purpose
`GET /api/health`	Return service status and the active model
`GET /api/settings`	Return settings with the real key masked
`GET /api/captures/<id>/image`	Read a full-screen capture from memory
`POST /api/captures/<id>/close`	Close the capture's Kiosk process
`POST /api/settings`	Save API, key, and theme settings

Route	Purpose
`POST /api/ai`	Crop a selection and stream an AI operation

Static paths are resolved and accepted only if they remain inside `web-dist`; a missing route falls back to `index.html`, allowing React to interpret `/capture/<id>`. JSON request bodies are capped at 1 MB, and responses carry `X-Content-Type-Options: nosniff`.

The server binds only to `127.0.0.1:8765`, so other machines on the LAN cannot connect directly. Loopback binding is not the same as full Web authentication, however. The API has no session or CSRF token; its practical boundary is the loopback interface, random capture IDs, and the assumption of a trusted local environment.

The OCR request: crop, encode, then upload

Only after the user starts recognition does the backend crop the selected region from the full-screen PNG. It Base64-encodes that crop as a Data URL and constructs an OpenAI Chat Completions-compatible multimodal message:

```
{
  "model": "Qwen/Qwen3.5-397B-A17B",
  "stream": true,
  "enable_thinking": false,
  "temperature": 0.1,
  "messages": [{
    "role": "user",
    "content": [
      {"type": "text", "text": "<OCR prompt>"},
      {"type": "image_url", "image_url": {
        "url": "...",
        "detail": "high"
      }}
    ]
  }]
}
```

Two choices stand out:

1. `detail: high` gives small text, subscripts, and complex formulae a higher-resolution visual input.
2. `enable_thinking: false` and a low temperature reduce latency and improvisation, keeping the result closer to a transcription.

Consequently, “screenshots stay in local memory” does not mean that no image leaves the computer. **The whole screen is not uploaded, but the selected crop is sent to the configured SiliconFlow API.** A

user should still consider whether screens containing passwords, private information, or unreleased code are suitable for remote processing.

More than generic character recognition

OCRBit's prompt is tuned for mathematical, chemical, and technical material. It asks the model to preserve headings, paragraphs, lists, tables, and code blocks, while emitting mathematics as LaTeX. Chemical formulae use upright letters and subscripts:

```
$\mathrm{MnC_2O_4}$
```

Reaction-arrow semantics receive even more specific treatment. The prompt distinguishes a one-way reaction annotated with conditions from an equilibrium arrow and forbids the model from “correcting” the source according to its chemical knowledge. This is no longer ordinary character recognition: the visual language model must understand the layout while being constrained not to invent absent information.

The OCR result can become the input to a second text-only request for translation, summarization, explanation, or code analysis. This phase accepts up to 200,000 characters and retains the rules for Markdown, code, tables, and formulae:

```
image ──multimodal OCR──> structured Markdown
                              │
                              └──text processing──> translate / summarize /
                                                    explain / analyze code
```

Streaming the model output with SSE

`ai.py` consumes the upstream response through `httpx.Client.stream()` and extracts each `choices[0].delta.content`. The local service repackages those chunks as Server-Sent Events:

```
data: {"text":"first chunk"}

data: {"text":"second chunk"}

event: done
data: {}
```

The browser reads the `fetch()` response with a `ReadableStream` and incrementally decodes blank-line-delimited events through `TextDecoder`. React appends every text chunk to the current result, so the user does not wait for the full response. An `AbortController` lets the user stop an active request.

The finished text is rendered by `react-markdown`; `remark-gfm` handles GFM features such as tables, while `remark-math` and `rehype-katex` render formulae. Raw HTML is not enabled, so model output such as `<script>` is not executed as page code. External links open in a new tab with `rel="noreferrer"`.

The API key boundary

Settings may come from a `.env` file beside the executable or from the Web interface. When the interface saves a key, the configuration module encrypts it with Windows DPAPI's `CryptProtectData` before writing:

```
%LOCALAPPDATA%\OCRbit\settings.json
```

Because the code does not request `CRYPTPROTECT_LOCAL_MACHINE`, the ciphertext is bound to the current Windows user by default. The browser receives only a `••••••••••••` mask when it reads settings. The real key is decrypted inside Python and added to the upstream `Authorization` header by the backend.

This prevents static frontend code and ordinary browser developer tools from directly reading the credential, but it cannot defeat malware already operating with the current user's privileges. DPAPI addresses plaintext credentials at rest; it does not turn a compromised machine into a trusted execution environment.

Building the single-file executable

During development, Vite provides frontend hot reload and proxies `/api` to the Python service. For a release, TypeScript checking and the Vite build first produce `web-dist`. PyInstaller then uses:

```
--onefile --console --add-data "web-dist;web-dist"
```

to package the backend, its dependencies, and the frontend assets into `OCRbit.exe`. The console remains visible for startup state, request logs, and errors, and closing it provides a direct way to stop the entire service.

“Single file” does not mean “independent of the system.” OCRBit still requires Windows 10/11, network access, a SiliconFlow API key, and an available system browser. What it avoids is bundling another copy of Electron and Chromium with the application.

The value—and limits—of this architecture

OCRBit is small, but it demonstrates a useful desktop-tool pattern: use a few Win32 calls for the system-level entry point, Python for local data and remote APIs, and mature Web technology for complex interaction. Compared with a complete desktop framework, its module boundaries are straightforward, and frontend and backend development remain independently testable.

Its current limits are equally explicit:

- Windows is the only supported platform, and both shortcut and model are fixed.
- Recognition depends on a remote service and does not work offline.
- The loopback API has no additional authentication.
- Capture cleanup is lazy and runs when the next capture is created.
- Kiosk behavior, forced process termination, and the dedicated browser profile require continued testing across browser versions.

Those constraints do not diminish the design. They make its tradeoffs legible: OCRBit is not a Web page wrapped in a desktop shell, but a system in which the operating system, browser, and multimodal model each perform the part they are best suited to handle.

References

- OCRBit source code
- v1.0.0 changelog
- Python backend entry point
- React frontend entry point