System Design Document

Self-Reflection App

Functional Requirements

Intent Phase

- User enters his intent
- User enters answers to the clarifying questions
- User can modify the intent generated by AI

Capture Phase

- User confirms starting the recording

End Session Phase

- User can end the session definitely
- User can pause the session
- User can resume session

Feeling Log Phase

- User enters their feelings

Report Phase

- User views the report

Non Functional Requirements

- Low latency to view report
- Privacy
- Capture performance: background capture must not noticeably slow the user's computer or drain battery.
- Data reliability — capture data must not be lost due to crashes or unexpected quits.

Architecture

Client (Electron)

Renderer Process (UI)

Displays UI, captures data, sends data to main process using IPC.

Main Process

Receives IPC messages from renderer. Routes them to the right service. Sends responses back to renderer via IPC.

Session Service
All business logic lives here.

AI Service
Builds prompts from session data. Calls the external AI provider API. Parses the response back into structured data.

Capture Service
One job: poll the active window every 3 seconds. Writes captures to SQLite.

SQLite
Local database. All data stored here.

External: AI Provider API

The AI service calls the external AI provider API for intent checking and report generation.

Data Model

session

- session_id
- original_intent (what the user first typed)
- final_intent (after AI refinement or user edit)
- status (active, paused, ended)
- started_at
- ended_at
- ended_by (auto, user)

capture

- capture_id
- session_id
- window_title
- app_name

- captured_at

feeling

- feeling_id
- session_id
- text
- created_at

session_events

- event_id
- session_id
- event_type (paused, resumed)
- created_at

reports

- report_id
- session_id
- summary (the AI verdict / session overview)
- patterns (the behavioral patterns with evidence)
- suggestions (the action items)
- status (generating, ready, failed)
- created_at

IPC Channels

Intent Phase

session:create → creates a session, checks intent with AI service
session:clarify → sends user's answers to AI service, returns final intent

Capture Phase

session:start → sets status to active, starts capture service

Feeling Logs Phase

feeling:create → stores a new feeling for the session

End Session Phase

session:pause → sets status to paused, stops capture
session:resume → sets status to active, restarts capture
session:end → sets status to ended, stops capture, triggers report generation

Report Phase

report:get → returns the report if ready, or "generating" status

Flow Explanations

1st FR: User enters intent → creates a session

The user types an intent in the renderer. It sends it via IPC to the session service. The session service calls the AI service to check if the intent is vague or specific. If specific, the session service creates a session in SQLite and returns it to the renderer. If vague, the AI service returns clarifying questions, the session service passes them to the renderer.

2nd FR: User answers clarifying questions

The user answers in the renderer. Sent via IPC to the session service. The session service does basic validation, then sends the answers to the AI service. This time the AI service returns the final refined intent. The session service stores it in the session in SQLite and returns the final intent to the renderer.

3rd FR: User edits the AI-generated intent

Same flow as 1st FR — the edited intent goes through the session service → AI service → SQLite.

4th FR: User starts recording

Renderer sends "start" via IPC to the session service. The session service updates the session status to "active" in SQLite and tells the capture service to start polling (every 3 seconds, check active window, write capture to SQLite).

5th FR: User ends/pauses session

Renderer sends "pause" or "end" via IPC to the session service. The session service stops the capture service, updates session status in SQLite, creates a session_event record. If "end" — the session service immediately tells the AI service to generate the report (reads all data from SQLite, sends to AI provider, stores the report in SQLite when done).

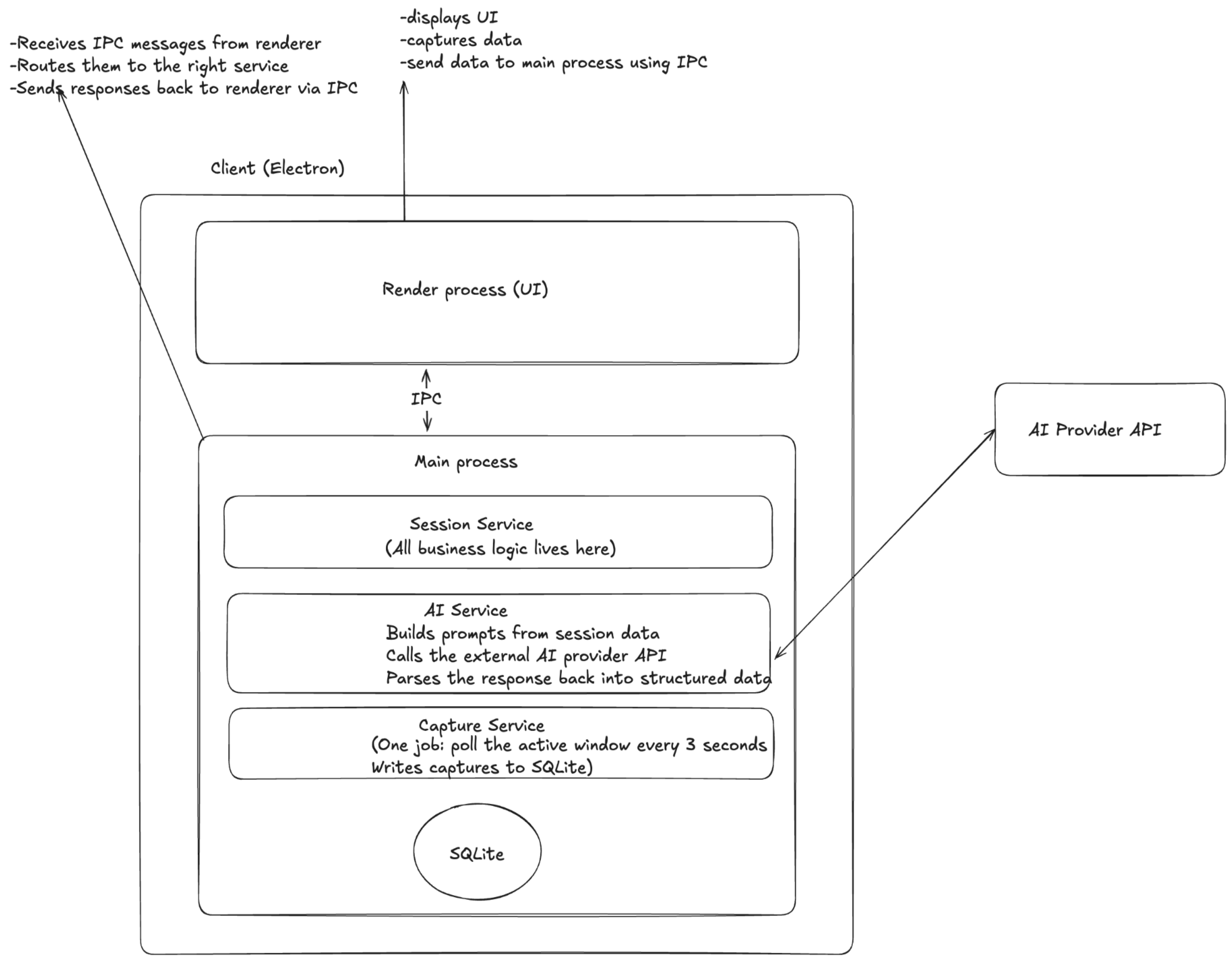
6th FR: User enters feelings

The floating window sends the text via IPC to the session service. The session service writes to the feelings table in SQLite with the timestamp.

7th FR: User views report

Renderer requests the report via IPC from the session service. The session service reads it from SQLite. If ready, returns it. If still generating, returns a "generating" status and the renderer shows a loading state.

General diagram



-Receives IPC messages from renderer
-Routes them to the right service
-Sends responses back to renderer via IPC
-displays UI
-captures data
-send data to main process using IPC
Client (Electron)
Render process (UI)
IPC
Main process
Session Service
(All business logic lives here)
AI Service
Builds prompts from session data
Calls the external AI provider API
Parses the response back into structured data
Capture Service
(One job: poll the active window every 3 seconds
Writes captures to SQLite)
SQLite
AI Provider API

Deep Dives

Privacy

Limiting capturing: only capture window titles

Are window titles safe from private data?

Window titles do not contain information such as passwords. However they provide context, which means knowing the window titles helps view what the person works on. This is not critical but to provide the best possible privacy, in the call made to the AI service we do not send any identity of the user (so data remains anonymous).

Local storage

All data is stored locally in SQLite on the user's machine.

No backend server

The only external data transmission is to the AI provider for intent checking and report generation. We eliminated a backend server because it would add a second external location where sensitive data exists without adding any functionality. (Here we will need to be transparent about what we send and how the AI external service handles the user data.)

User control

Although not practical, the user can pause the capture if they worry about sharing a window title.

Capture must not slow the user's computer

Now the question we need to ask before thinking about any other solution: is this problem even real in our application? So let's think through this:

What does the capture service actually do?

It does two things repeatedly:

- Get the active window title
- Store it

Both are tiny operations. Getting a window title is a single system call — under 1ms. Storing a row in SQLite is ~100 bytes — under 1ms.

Getting the active window is a single system call (~1ms). Writing to SQLite is one row (~130 bytes). Even if this runs every second, that's 1ms of CPU work per 1,000ms —

0.1% CPU. For comparison, a single idle Chrome tab uses 1-3% CPU. Our capture is 10-30x less than doing nothing in a browser. Disk and memory impact are similarly negligible — one 130-byte write per second is invisible to an SSD that handles hundreds of megabytes per second.

So based on this, performance issues are not a concern in our app. This means that there's no need to search for different solutions to avoid the performance problems of capture because the risk doesn't even exist and for that we will simply pick the simplest solution:

Solution: polling every X time

Now what we will do is define the granularity (X time):

To define X, ask: what's the shortest app switch that the AI needs to know about?

If someone opens an app for 1-2 seconds, that's a reflex — misclick, muscle memory, immediately closed. Not a behavioral pattern.

If someone is on an app for 4-5+ seconds, they're engaging with it — reading, watching, scrolling. That's a conscious action the AI should analyze.

So the interval needs to be short enough to catch intentional switches but doesn't need to catch reflexive ones. 3 seconds. This also filters noise. For example:

If someone opens Twitter and closes it in 1 second, do you want the AI to say "you got distracted by Twitter"? That person wasn't distracted. They opened it by accident and immediately closed it. Reporting that as a distraction would be wrong and annoying.

Data reliability

We do not have a risk of data loss because we are writing to disk. The only data we lose is non-intentional actions.

Storage

Let's see if storage is a concern or not:

Extreme case: 12 hours of capture per day for 3 years.

12 hours = 43,200 seconds ÷ 3 = 14,400 captures per day

Each capture ≈ 130 bytes

Per day: 14,400 × 130 = ~1.9MB

Per year: ~690MB

Per 3 years: ~2GB

Everything else (feelings, reports, sessions, events) adds maybe 50MB total.

Total: ~2GB after 3 years of 12 hours/day usage. So not a concern.

Low latency to view report

To think about low latency we need to think of the whole chain and see what will make the latency higher and then optimize for that.

The chain when the user clicks "View Report":

Read intent from SQLite → ~1ms

Read captures from SQLite → ~1-5ms (even with thousands of rows)

Read feelings from SQLite → ~1ms

Read session events from SQLite → ~1ms

Collapse captures into time spans → ~10ms

Build AI prompt → ~1ms

Send to AI provider → ~200ms network

AI provider processes and generates report → 10-20 seconds ← needs to be optimized

Store report in SQLite → ~1ms

Return report to renderer → ~1ms

Optimizations:

Trigger processing before the user wants the report: when the session ends, we already have all the data needed. So we trigger report generation at session end, not when the user clicks "View Report".

It's important to know that the AI service we have is not slow — it only reads data and prepares input and sends it. The AI provider is the one that is slow and we have no control on making it faster. However what we can do is control the inputs (less inputs, less processing) and control the output (when it needs to generate a lot, it is slower).

However it's also important to note that even with optimizing the inputs and outputs the AI provider can still be slow. That's why it's important to help more with the optimization of triggering processing before.

To optimize the input:

Instead of sending raw capture rows to the AI, collapse consecutive identical captures into spans.

To optimize the output:

It's about writing a great prompt so that the AI will give me what is needed and then stop — no filler text.