ChatGPT

Real-Time Desktop AI Perception Agent (Windows) – Product Requirements Document

Overview: This document specifies a lightweight, real-time AI perception agent for Windows that minimizes heavy computer-vision (CV) loops (screenshots/video streaming). Instead, it leverages high-speed **system-level signals** (process events, window metadata, UI elements, input hooks, etc.) to understand user context in under 100 ms. The agent runs natively on Windows (using Win32 APIs, UI Automation, ETW, etc.) for maximal performance, and only escalates to large cloud LLMs (e.g. GPT-5.1) when advanced reasoning or summarization is needed. Key components, signals, technology choices, and integration plans are detailed below.

1. System Architecture

The perception agent is structured as a modular, event-driven pipeline that captures OS-level UI events and produces high-level context insights in real time. **Figure 1** illustrates the overall architecture and data flow:

Figure 1: System architecture of the real-time perception agent. Native Windows event sources feed into a high-performance "Event Collector" service (written in Rust/Go/C++ for efficiency). This service parses and fuses signals, then uses a fast on-device SLM (Small Language Model) or rule-based logic for immediate context inference. A decision engine determines if/when to escalate to a powerful LLM (GPT-5.1) for heavy analysis. The agent's insights are then provided to the broader DesktopAI system (for display, suggestions, or autonomous actions).

Pipeline Description: The agent operates in stages, decoupling fast local perception from slower cloud reasoning:

- **Native Event Capture:** Using low-level OS hooks and accessibility APIs, the agent subscribes to relevant Windows events (e.g. window focus changes, new window opened, UI control events, user input) in real-time. This avoids polling and heavy screen scans – the OS pushes only actual changes, improving efficiency [1]. The Win32/UIA infrastructure raises events selectively only when clients are listening [1], reducing overhead versus continuous polling. For example, rather than capturing the screen every 100 ms (which is costly at ~60–100 ms per 1080p frame [2]), the agent relies on event callbacks that fire only on meaningful changes (window open/close, focus shift, text input, etc.).

- **Event Collector (Edge Runtime):** A native service (written in a fast systems language) receives the OS events with minimal latency. It uses high-performance Windows APIs (Win32, UI Automation, Event Tracing) to gather additional context for each event. For instance, when a window-focus event fires, the collector might query the window title, application name, or accessible UI properties immediately. This module is optimized for concurrency and low jitter, ensuring events are processed within a few milliseconds of arrival. (Notably, UI Automation (UIA) is designed for out-of-process clients with good performance [3], avoiding the need for risky in-process hooks.) The Rust-based `ui-events` project exemplifies this approach: a Rust core listens to native accessibility events (UIA on Windows) and streams them with low latency [4] [5]. This real-time collector forms the backbone for sub-100 ms perception.

- **Local SLM (Fast Perception Model):** The agent employs an on-device "Small Language Model" or lightweight ML classifier to interpret the fused signals. This model is optimized for speed (potentially a few dozen million parameters or a rule-based system) and might run via an efficient runtime (ONNX, ggml, etc.) on the user's machine. It produces immediate **insights** from the raw events – e.g. classifying the user's activity ("User is browsing Slack chat" or "Editing a contract in Word") based on window titles, process names, and recent input patterns. The SLM essentially functions as a real-time pattern recognizer or state classifier that can be queried every time an event arrives, without introducing noticeable latency (<100 ms end-to-end). By keeping most perception on-device, we ensure responsiveness and privacy.

- **Decision Engine & State Manager:** A higher-level logic module (could be integrated with the SLM or separate) maintains the user's **context state** and decides when an event or situation warrants escalation to a cloud LLM. This engine fuses event insights over time to update a structured view of what the user is doing (for example, it might maintain flags like `activeApplication = "Slack"`, `currentDocument = "NDA_Contract.docx"`, `userIsTyping = true`). Lightweight rules or thresholds trigger state transitions (e.g. if a new app comes into focus or if the user has been idle for some time). The decision engine can handle routine context updates itself, but if a complex analysis is needed (e.g. summarizing a long document, reading multiple sources, planning a multi-step action), it prepares a query for the cloud LLM.

- **LLM Escalation (Cloud GPT-5.1):** For heavy-duty comprehension or multi-step reasoning, the agent will invoke a large language model. The escalation logic (detailed in section 5) ensures this happens *only when necessary* to minimize latency and cost. When invoked, the agent assembles a concise *message* describing the current context/state and the task for the LLM. For example, if the user opens a lengthy contract, the agent might send the document text (retrieved via UI Automation's text control pattern) or key excerpts along with a request like, "Summarize the key points of this contract." The LLM's response is then returned to the decision engine. By containing LLM calls to infrequent, high-value tasks, the system stays real-time for most interactions.

- **Integration & Output:** Finally, the insights (from either the SLM or LLM) are integrated into the DesktopAI environment. This could mean updating a shared context that the user's AI assistant references, triggering a notification or suggestion (e.g. "It looks like you are editing a contract – need help summarizing clauses?"), or logging the event to the agent's memory. The perception module itself does not directly manipulate UI (unlike an RPA tool); rather, it feeds understanding to higher-level agent components that decide *what* to do with that knowledge. This clean separation makes it easy to integrate the perception agent as a background service in any agentic platform (see Integration Plan).

Rationale: This architecture prioritizes **throughput and low latency** on the edge. By leveraging the OS's own event notifications and a local model, most user actions are processed **within tens of milliseconds**, rather than incurring the hundreds of ms or seconds that cloud vision/LLM calls might take. Similar designs have been validated in related systems – for instance, the AgentSight framework shows that monitoring system-level events combined with selective LLM analysis can incur under 3% CPU overhead [6] [7]. Likewise, prior Windows assistants like PyWinAssistant demonstrated that purely symbolic, event-driven UI understanding (without OCR) is feasible for general automation [8] [9]. In summary, the architecture ensures the agent is **reactive** (sub-100 ms reaction to user input), **efficient** (native code, minimal busy-looping), and **scalable** (only escalates to expensive computation when truly needed).

2. Signal Surface Map (Windows System-Level Signals)

This section catalogs the **Windows APIs and telemetry sources** the agent can tap into, including their event types, pros/cons, and best-use scenarios. By exhaustively mapping the "signal surface," we ensure the agent observes all relevant user interactions without resorting to constant screen captures. Key categories include: **UI accessibility events, windowing events, input hooks, process telemetry,** and other OS signals. The table below summarizes these sources, followed by detailed notes:

- **UI Accessibility (UI Automation & MSAA):** High-level UI events via Microsoft's Accessibility frameworks.
- *API:* Microsoft UI Automation (UIA) is the modern interface (with legacy MSAA as fallback).
- *Events:* Focus changes, control invocations (button clicks), text edits, menu opens/closes, UI structure changes, etc. For example, UIA raises events like `AutomationFocusChanged` when focus moves, `WindowOpened/Closed` for top-level windows, `ValueChanged` when a text box content changes, etc. [10] [11]. These cover most GUI interactions of interest.
- *Pros:* **Rich semantic info** – UIA exposes a tree of UI elements with properties (name, type, value) and control patterns. The agent can not only know that "a window opened" but also what the window *is* (e.g. a dialog with title "Save As") and even read its contents if accessible. **Event-driven** – the OS/UI framework sends events only on changes, enabling real-time updates without polling [1]. UIA is designed for out-of-process clients with minimal performance hit [3] (screen readers use it to promptly narrate changes). **No vision needed** – all data is textual/structural.
- *Cons:* **Requires app support:** Most standard Windows apps and frameworks support UIA or MSAA by default (for accessibility), but some highly custom or older apps might expose limited info. In worst cases, only their window title might be available. **Granularity trade-offs:** Subscribing to very fine-grained events (like every text change in a document) can lead to event floods. The agent should subscribe selectively (e.g. focus shifts, window opens) and possibly throttle or filter less critical events. **Permission:** Generally no special privilege needed, but UIA cannot penetrate *across security boundaries* – e.g. to inspect UI of an app running as Administrator, the agent process also needs to run elevated. This can be handled by design (the agent could run with user-level access unless full admin coverage is needed).

- *Use Cases:* **Primary source for UI context.** For example, when the user switches to a Slack window, a UIA focus event provides the new foreground window handle and title; the agent can then query UIA for the Slack window's name or any accessible text (like the channel name). When the user opens a Word document, UIA's events can inform the agent of the window open and potentially the document's name (often in the title bar). Essentially, UIA events are the *eyes* of the agent regarding GUI state changes, enabling statements like "User opened *Quarterly_Report.xlsx* in Excel" or "Focus moved to an email compose window."

- **Window System Events (WinEventHooks):** Lower-level windowing events from the Win32 subsystem.

- *API:* `SetWinEventHook` (Win32) allows registration for a range of events in the Windows GUI system (part of WinUser). This is actually the under-the-hood mechanism that UIA and MSAA use to receive events. Event constants include things like `EVENT_SYSTEM_FOREGROUND` (foreground window changed), `EVENT_OBJECT_CREATE` / `DESTROY` (objects like windows or controls created/deleted), `EVENT_SYSTEM_MINIMIZESTART/END` (window minimized or restored), etc. By setting hooks for specific event ranges, the agent can get callbacks for those occurrences.
- *Pros:* **Broad coverage:** WinEventHook can catch *all* events in the GUI system, including ones not surfaced by higher APIs. It can be restricted to certain processes or threads if needed, or set globally. **Efficiency:** It ties into Windows' internal event stream, which is highly optimized. Experts note that using WinEventHook has minimal overhead because it "hooks into the system that Windows already uses to monitor these events" (i.e. it piggybacks on the OS's own event notifications) and doesn't require constant checks. It's generally more efficient than polling for window changes every X ms [1]. **No injection required (out-of-process):** Unlike some other Win32 hooks, WinEventHook can call back in an external process context without injecting code into every app. This keeps things simpler and safer.
- *Cons:* **Less semantic detail:** The events give basic info (e.g. a window handle and event type), but to get details (like window title or control name) often you must call other APIs (e.g. `GetWindowText`, or use UIA on that handle). In practice, our agent will use WinEventHook in tandem with UIA: the hook tells us "something changed here", then UIA is queried for specifics. **Global impact:** Installing many wide-range hooks could, in theory, slightly affect system performance. However, targeting specific event ranges (like foreground changes and window open/close) greatly limits the load. For example, monitoring `EVENT_SYSTEM_FOREGROUND` (focus) and `EVENT_OBJECT_DESTROY/CREATE` for top-level windows results in just a handful of events on app switches – a negligible cost. Proper use (and unhooking when not needed) mitigates any issues.

- *Use Cases:* **Basic window tracking and focus changes.** The agent uses WinEventHook to know immediately when the active window changes (instead of polling `GetForegroundWindow` in a loop). This yields the *window handle of the now-active window*, which the agent then correlates to a process/app name (via `GetWindowThreadProcessId` and process lookup) and a UIA element for rich info. Similarly, by hooking window create/destroy events, the agent can detect when the user launches or closes applications or dialogs. For instance, if `EVENT_OBJECT_CREATE` comes in for a top-level window, the agent knows a new window opened – it can then query its title ("Install Updates?") and notify the agent logic. These hooks thus form the *skeleton* of the agent's knowledge of which app is in use at any time.

- **Input Events (Keyboard & Mouse Hooks):** Low-level input streams that indicate user actions.

- *API:* Win32 **Windows Hooks** (`SetWindowsHookEx`) provides WH_KEYBOARD_LL and WH_MOUSE_LL for low-level global hooks of keyboard and mouse, respectively. These allow capturing key presses and mouse clicks/movements system-wide before they reach applications. Alternatively, **Raw Input** (`RegisterRawInputDevices`) can be used to get input from HID devices with minimal overhead (often used in games to capture input outside of focus). The agent can also use higher-level queries like `GetLastInputInfo` to detect idle time (how long since last input event).
- *Pros:* **Complete user action capture:** These hooks can tell exactly *what* keys the user is pressing or where the mouse is clicking. This is useful for certain inferences (e.g. detecting the user started typing in an empty email body – could hint they are composing an email). For idle detection, `GetLastInputInfo` is extremely cheap (just returns a timestamp of last input) and can be polled

every few seconds to see if the user has stopped interacting. **Timely:** Low-level hooks are invoked immediately on input hardware events, so they are as real-time as it gets (order of milliseconds). They can help timestamp user actions to correlate with other events (e.g. knowing a key press coincided with a new window could mean a hotkey launched an app).

- *Cons:* **Intrusive and potentially high-volume:** Capturing every key press and mouse move is a lot of data. It can introduce overhead if not handled carefully (especially mouse movement which can fire many events). The agent should filter input events – e.g. it might only care about specific keys (like Enter, or certain shortcuts) or about general activity vs idle. Logging every character the user types is also a privacy concern and generally unnecessary for our use-cases. **Security restrictions:** Low-level hooks require the process to run at sufficient privilege. They can capture sensitive info (like passwords being typed), so on modern Windows there are some constraints (e.g. 64-bit processes can only be hooked by 64-bit, etc.). The agent needs to be signed/trusted to use these globally without being flagged by security software. However, since our agent is user-installed and privileged as an assistive tech, this is manageable.

- *Use Cases:* **Idle/Activity monitoring and special triggers.** The agent will primarily use input signals to know if the user is actively interacting or not. For instance, if no input events have occurred for, say, 5 minutes, the agent might infer the user is away and hold off on any suggestions (or conversely, use the idle time to summarize context in the background). Another use: detect certain global hotkeys or patterns – e.g. if the user presses a particular shortcut to summon the AI assistant, or if a rapid sequence of keystrokes might indicate the user is executing a repetitive task (potentially suggesting automation). Generally, detailed key logging is **avoided**; the agent focuses on higher-level intent, not the exact content the user types (content can be gleaned from UIA if needed in a privacy-respecting way). Mouse events might be used to detect *which* part of an app the user is interacting with (though UIA can often tell which control has focus already).

- **Process and System Telemetry:** Events about process lifecycle and system resource usage.

- *API:* **Event Tracing for Windows (ETW)** provides real-time events for process start/stop (via kernel provider), CPU/memory usage, disk I/O, etc. We can use ETW sessions to subscribe to process creation events (`Microsoft-Windows-Kernel-Process` provider) or module loads. Alternatively, **WMI** (Windows Management Instrumentation) offers event queries like `Win32_ProcessStartTrace` and `Win32_ProcessStopTrace` for process launch/exit.
- *Pros:* **Visibility into app launches** beyond just windows. For example, some background processes might start without a window; ETW or WMI can catch those if needed (though the agent primarily cares about user-facing apps with windows). **Lightweight for process events:** Process start/stop via ETW is very efficient (used in system monitors). This ensures if the user launches a new program via non-GUI means (say a command line script), the agent can still know a new process is running. **Resource context:** If needed, the agent could sample CPU or memory usage of the active app (via performance counters or ETW) to infer if the system is under heavy load (which might affect how/ when the agent presents suggestions).
- *Cons:* **Less critical for UI context:** Knowing every background process is often unnecessary for user workflow understanding. The agent should focus on *foreground* processes. ETW data can be very verbose if not filtered carefully – but we will use it sparingly (e.g. only process start events, not full-blown tracing of all system calls). **WMI latency:** WMI event delivery can sometimes lag (hundreds of ms to seconds) and WMI is relatively heavy-weight compared to ETW. If used, it should be as a backup. **Access:** ETW requires admin rights to listen to certain kernel events (though user-level might subscribe to some providers with reduced detail).

- *Use Cases:* **Process launch detection and context enrichment.** For instance, when the user opens a file by double-clicking in Explorer, there may be a race between the window creation and process start. Monitoring process start (with image name/path) can complement the window event – confirming the exact application exe, which can help identify the app more reliably. Also, if needed, the agent can log how long processes run or if a process crashes (though that veers into monitoring territory beyond core perception needs).

- **Other Signals:**

- **Shell/Explorer events:** The Windows shell can broadcast events for certain user actions (like file open, folder navigation). These are typically accessible via Shell APIs or .NET's `FileSystemWatcher` etc. Relevance to our agent is limited (since those usually correspond to opening apps or files which we catch via other means). One possible use: detecting when the user plugs in a projector or changes display (could influence UI scaling or agent UI placement).
- **Clipboard events:** If the user copies text or an image, Windows doesn't provide a direct event, but the agent could poll the clipboard sequence number occasionally. If we sense a lot of copy-paste activity, it might be context (e.g. user copying text from web to an email).
- **Network status:** If needed, network connectivity or VPN on/off might be obtained (to avoid cloud calls when offline, for example).
- **Time/user calendar:** Not an OS event per se, but the agent might query the system clock or calendar events to contextualize (e.g. user has a meeting starting – might not want interruption). This veers into contextual awareness beyond pure UI signals but can be integrated.

Pros/Cons Summary: Using **system-level signals** provides rich and immediate insight into user activity with minimal performance cost. Unlike analyzing screenshots (which are slow to capture and require expensive vision models), these signals are largely *event-driven and textual*. They leverage the fact that modern OSes like Windows already maintain a live model of the UI (windows, controls, focus, etc.) for accessibility and automation – we simply tap into that feed. The downside is that it requires careful engineering to correlate and interpret these disparate events (hence the need for a perception strategy and SLM). Some rare application scenarios might not emit enough signals (e.g. a game rendering a canvas has little accessible info), but those are outside typical productivity tasks. Overall, the chosen signals cover the vast majority of knowledge work scenarios in real-time and with **negligible CPU overhead** (the approach is akin to how screen readers operate, which is known to be efficient for the user). It's a trade of a bit more development complexity for a **lightweight runtime** – a favorable trade to meet the <100 ms latency goal.

3. Runtime Technology Comparison (Rust vs Go vs C++ vs Zig)

To implement the core **Event Collector and signal processing service**, we evaluate several low-level runtime options: **Rust, C++, Go, and Zig** (and consider others briefly). The goal is an *edge-optimized* binary that is fast, reliable, and can interface with Windows APIs easily. Key criteria include execution speed, memory safety, tooling for Windows hooks/UIA, concurrency support, binary size, and GC overhead (we want none or very minimal). Below is a comparison and recommended choice:

- **Rust:** A modern systems language with strong performance and memory safety guarantees.
- *Pros:* **Memory safety + concurrency** – Rust's ownership model prevents common bugs (use-after-free, buffer overflows) without a garbage collector. This is crucial for a long-running agent that hooks into other processes; a crash or memory leak in the agent could destabilize user workflow, so Rust's

safety is valuable. It has excellent support for **multithreading**, allowing the agent to handle multiple event streams in parallel (e.g. one thread listening for UIA events, another for input hooks) without data races. **Performance** is on par with C/C++, and Rust can directly call Win32 APIs via FFI. The Rust ecosystem has the `windows` crate (official Microsoft crate) which provides bindings to Win32 and COM APIs, including UI Automation, making it feasible to interact with UIA in pure Rust. Community projects like `rustautogui` demonstrate Rust's prowess in GUI automation, achieving ~5× faster execution than equivalent Python scripts [12] (they optimize screenshot processing, but it underscores Rust's speed). **No runtime/GC:** Rust compiles to a static binary with minimal overhead, ideal for a lightweight agent.

- *Cons:* **Steeper learning curve and development effort.** Rust's strict compile-time checks mean a longer initial dev phase for those not already proficient. Debugging Windows-specific COM interactions in Rust can be tricky due to less mature examples compared to C++/C#. However, this is mitigated by growing resources (Microsoft's Rust/WinRT project, community crates). **Binary size** can be a bit larger than equivalent C++ (due to safety checks, though in release builds Rust is quite optimized). That said, a Rust binary for our agent would likely still be only a few MB. Overall, Rust's cons are mostly developer-facing; the end product's performance and safety are top-notch.

- **C++:** The tried-and-true language for Windows systems programming, with direct access to all APIs and no managed runtime.

- *Pros:* **Native integration** – Windows APIs are essentially made for C/C++. There are abundant examples and libraries for using UI Automation (COM interfaces like `CUIAutomation`), setting Win32 hooks, etc. Many existing projects (including the Windows Automation MCP server, or native RPA tools) use C++ for the heavy lifting. **High performance** – compiled to native code, no GC pauses, and you can optimize at a low level. **Control** – C++ allows fine-grained control of memory and threads (which is both a pro and a con). A skilled C++ developer can write an extremely efficient event loop and even handle COM callbacks with custom allocators, etc. The language has improved (C++17/20) to offer some safer abstractions (smart pointers) that we can leverage to reduce raw pointer mistakes.

- *Cons:* **Memory safety risk** – Unlike Rust, C++ cannot guarantee safety. A bug like a buffer overflow or dangling pointer in the agent could cause crashes or, worse, corrupt other processes if we injected hooks in-process. This risk can be mitigated by careful coding and using modern practices, but it's inherently present. **Concurrency pitfalls** – Writing multi-threaded C++ code that is free of races and deadlocks is non-trivial. It lacks Rust's compile-time checking for those. **Dev effort** – While easier to find Windows-specific code, debugging issues (like COM reference counting errors or subtle race conditions) can consume significant time. Also, implementing the higher-level logic (state management, etc.) might be more laborious in C++ compared to higher-level languages. Despite these, C++ is a strong contender due to its maturity on Windows. It would require rigorous testing to ensure stability given the agent's always-on nature.

- **Go:** A garbage-collected systems language known for ease of use and built-in concurrency (goroutines).

- *Pros:* **Rapid development** – Go's simplicity can speed up coding the agent logic. Built-in features like channels and goroutines make concurrent event handling straightforward. **Good FFI and some Windows support** – Go can call C APIs via cgo, and the community provides packages (e.g.

`golang.org/x/sys/windows`) for common Win32 calls. For example, there are Go libraries for low-level keyboard hooks and for UI Automation one could use COM via cgo. **Memory safety** – Go avoids explicit pointer arithmetic and has automatic memory management (reducing certain classes of bugs, though not the logic bugs). **Binary distribution** – Go builds static binaries easily, which is convenient for deploying the agent.

- *Cons:* **Garbage Collector (GC) and latency** – Go's GC, while fast, can introduce pause times. For a real-time perception loop targeting <100 ms, GC pauses might occasionally cause frame drops or delayed event handling, especially if the agent allocates a lot of short-lived objects. Careful coding (reusing buffers, etc.) can minimize this, but unpredictable latency spikes could occur under load. **Interfacing with COM/UIA** – Go has no native COM support; one would need to write C wrappers or use Windows COM libraries via cgo. This adds complexity and potential performance overhead crossing the language boundary. **Deployment size** – Go binaries can be larger (a simple Go Hello World exe is a few MB due to runtime). Our agent might be ~10–20 MB in Go, which is still fine, but heavier than Rust/C++ equivalents. In summary, Go could work for the agent, but we'd need to be mindful of GC and possibly limit Go's use to higher-level orchestration while using small helper DLLs in C for critical hooks to ensure consistent ultra-low latency.

- **Zig:** A newer systems programming language that is C-like but with safety features and no runtime overhead.

- *Pros:* **No runtime/GC** – Zig is designed to produce efficient binaries like C, with manual memory management but some modern niceties. **Simplicity and interoperability** – Zig can directly call C functions and use C headers, making it straightforward to call Win32 APIs. It also can compile C code as part of the build, which might simplify using any existing C libraries for UI Automation. **Safety features** – Zig has optional runtime checks and a safe programming model (no hidden control flow, etc.), potentially reducing bugs. **Small footprint** – Zig executables tend to be quite small and fast.
- *Cons:* **Maturity** – Zig is relatively young. Its ecosystem for Windows-specific tasks (like COM interop) is minimal compared to Rust or C++. We might have to implement more low-level plumbing ourselves. **Community and tooling** – Fewer developers know Zig, and debugging tooling is still improving. Adopting Zig would be a bold choice that could pay off in binary quality, but at the cost of higher development effort and risk (since less prior art is available).

In addition to these, we considered **C#/.NET** for a moment (since UIA has robust .NET support), but a managed VM with garbage collection in the inner loop is not ideal for our <100 ms latency target. .NET could be used for prototyping or parts of the stack (Microsoft's own PowerToys/Power Automate use .NET for UI automation tasks), but we'd prefer a leaner runtime for production. Python is unsuitable for the core due to slow execution (though Python could be used at higher layers if needed, given its rich AI ecosystem – e.g. PyWinAssistant uses Python with heavy reliance on UIA events, but for us Python would likely miss our performance requirements without native extensions).

Recommendation: Use **Rust** for the core event collection and signal processing service. Rust offers a unique blend of C++ performance with safeguards that are valuable in an always-on agent. Its growing support from Microsoft for Windows development [4] and existing examples of UI automation mean the learning curve is surmountable. We anticipate building the event hook and UIA interfacing in Rust (with perhaps some small amount of unsafe code for COM interop), and leveraging Rust's async or multi-threading for handling parallel event channels. The runtime will have deterministic performance (no GC pauses) and memory safety to reduce crashes. C++ would be the second choice if Rust expertise is lacking, as it certainly can accomplish the task but with higher risk of memory errors. Go could be considered for parts of the system that are less timing-critical (for instance, the decision engine or integration layer) if the team prefers its simplicity, but for the *perception loop itself* (which must be highly reliable and low-latency) Rust/C++/Zig are preferable. On balance, Rust hits the sweet spot for a modern, safe, and performant implementation, aligning with our objectives of an edge-optimized, deterministic agent.

4. Perception Strategy (Lightweight Event Fusion & State Inference)

The perception strategy defines *how the agent interprets raw events to produce meaningful context like "user is browsing Slack" or "editing a contract."* Rather than relying on computer vision to recognize applications or on lengthy prompts to an LLM, we use **lightweight event fusion and local inference**. The agent maintains an internal **state model** of the user's context, updated by rules and the edge SLM as events stream in. Key aspects of the strategy:

- **Defining User States and Transitions:** We enumerate the high-level "states" or activities relevant to the assistant. Examples: *Idle*, *Active – Communication (Slack/Teams)*, *Active – Document Editing*, *Active – Coding*, *Browsing Web*, *In a Meeting/Call*, etc. These are not rigidly exclusive (some could overlap), but they represent the primary focus of attention. The perception agent's goal is to keep track of which state the user is in and update it quickly when a shift occurs. **State transitions** occur on significant event patterns, e.g. **Focus-based transitions:** when the foreground window switches to an app of a different category (user goes from VSCode to Chrome – likely from "Coding" state to "Web Browsing" state). **Content-based transitions:** when the nature of content changes within the same app (user goes from one Word document to another; if one is identified as a "Contract" and another as "Presentation", that might be a sub-state change).

- **Feature Extraction from Signals:** To decide these states, the agent relies on features derived from the raw signals:

- *Application identity:* The process name or window class often identifies broad category. E.g. `slack.exe` or window title containing "Slack" → Communication app; `winword.exe` (MS Word) → Document editing; `code.exe` (VS Code) → Coding/IDE; `chrome.exe` or `msedge.exe` → could be web browsing, but needs URL to be sure (though we might infer web if the window title has " - Google Chrome"). For known apps, we can maintain a mapping to state categories.
- *Window title and UI metadata:* These give more context. E.g. if Word is open with a document named "**Contract_Draft.docx**" in the title, the agent infers this is likely a legal contract document. If a browser's title contains "Slack | #general", it's actually the Slack web app (comm effectively). For browsers, the URL or title can distinguish (we might integrate with the browser via an extension for precise URL; if not, the title often contains the page title). Another example: an Excel window titled "Q4_Financials.xlsx" suggests the user is working on a spreadsheet – possibly analysis task.
- *User input behavior:* Key/mouse events can refine the understanding. If the user hasn't typed or clicked for a few minutes, the agent might mark the state as "Idle" (even if an app window is open, user is not actively engaged). Conversely, continuous typing in a Slack window vs. just reading messages can indicate an "actively communicating" state. If the agent sees rapid window switching and mouse clicks (perhaps copying info from one window to another), it might infer a "multi-tasking" state or a short-term goal (like transferring data).

- *Time of day and calendar:* (Optional) If it's during a scheduled meeting and a video-call app is in focus (Teams/Zoom), state could be "In a Meeting". Or late at night in a code editor might mean "Coding (after hours)".

- **Edge SLM for Classification:** We will train or configure a small local model to classify these states based on the features. For instance, we could feed a simple textual description of the context to a tiny model: "App: Word, Title: Contract_Draft.docx, Recent words: 'Agreement, Party, Clause'" and have it output "Editing a contract (Legal document)". Similarly, "App: Slack, Channel name: #design, User is typing: no" might output "Reading Slack channel (communication)". The SLM could be a fine-tuned small transformer or even a rule-based classifier with regex and keywords enhanced by ML for edge cases. The key is that it should be **fast (<20 ms)** and cover common patterns. PyWinAssistant's approach is noteworthy: it achieves "human-like abstraction across GUI semantics" entirely through the accessibility API, identifying interface elements and inferring tasks without any pixel analysis [13] [9]. We aim to similarly use semantics – e.g. recognizing a "Send" button in the UI means the user is composing a message, etc. Our SLM might incorporate some of these rules.

- **Fusion Logic (Rules + Model):** The agent uses a **hybrid of deterministic rules and model predictions** to update state. For high confidence patterns, simple rules are applied: *Example:* "If foreground process = slack.exe then state = Communication (Slack)" – easy. But to get "browsing Slack vs. writing a message", it might consider if the last input was in Slack's message textbox. A rule could be: "if Slack and user pressed Enter within Slack recently, assume they sent a message => active communication". More ambiguous cases, the SLM handles: e.g. distinguishing "writing code vs. writing prose in VS Code" could depend on file extension (.py vs .md) or contents – the SLM can look at recent text or file names to decide if the user is coding.

- The fusion logic also ensures *stability*: we don't want the state to flap on every minor event. For example, if a notification window pops up briefly stealing focus, the agent shouldn't immediately drop the "Coding" state; it might wait a fraction or require the new state to persist for say >2 seconds. The decision engine can implement a short delay or hysteresis to avoid rapid oscillations (sub-100 ms reaction is goal, but for certain noisy signals, a slight debounce can improve accuracy without hurting usefulness).

- Additionally, multiple concurrent states could be tracked: e.g. the agent can note "User is primarily editing a contract in Word, but also has an email draft open in background" – however the *active* state is the Word editing. The strategy will prioritize the foreground task but could surface secondary context if relevant (like reminding about that unsent email later).

- **Examples of State Inference:**

- *Slack Browsing vs. Editing Contract:* Suppose the user was editing a Word document named "NDA_Contract.docx" for the last 30 minutes – the agent state is "Editing a contract". Now the user Alt-tabs to Slack (Slack window gets focus, UIA event fires with Slack's title). The collector feeds "Slack – ACME Workspace" to the SLM. The SLM sees app=Slack, no immediate typing yet, so it outputs "Browsing Slack" (meaning user likely reading messages). The decision engine transitions state to "Communication – Slack (browsing)", possibly pausing any contract-related assistance. If the user then starts typing in Slack (we detect key events or UIA event on the message input control), we

refine to “Chatting on Slack” actively. This nuance could be used by the agent to, for instance, not bother the user with unrelated suggestions while they are actively chatting.
- *Editing a Contract:* Now say the user switches back to Word, or opens a new document. The agent sees either the known contract file name or, if new, it might quickly scan the first paragraph via UIA’s text pattern. If terms like “Agreement” or legal clauses are present, the SLM can label it as a likely contract or legal document. Thus state becomes “Document Editing – Legal Contract”. The agent could then proactively consider preparing a summary (but per our escalation logic, that heavy step would be deferred until asked or idle time, not immediately every time).

- *Web Research vs. General Browsing:* If Chrome is active, the agent could use the title to infer what kind of site is open. E.g. “Stack Overflow – [question title]” → probably coding research; “YouTube – How to…” → maybe the user is watching a tutorial (this might trigger a different kind of assistance or just be noted). For deep integration, a browser extension could send the URL/page text to the agent for analysis, but our spec avoids complex integrations – basic title inference is an option, with the SLM classifying known domains (“docs.google.com Document” vs “news site vs corporate wiki”).

- **No Vision Screenshot Loop:** It’s important to highlight *how* we manage to get these insights **without any pixel analysis**. We leverage the metadata: file names, window titles, control labels, and even accessible text content. For instance, PyWinAssistant achieved non-visual perception by relying exclusively on UI Automation, using control metadata and hierarchical relationships to understand the interface [9]. Our agent follows this principle. If needed, a single screenshot might be taken in rare cases as a fallback (PyWinAssistant mentions using an occasional screenshot to visually verify or for things like finding an icon [14]), but our primary pipeline avoids it. This dramatically reduces CPU/GPU usage and latency. A screenshot+OCR approach would be too slow (tens to hundreds of ms for capture and recognition) and unnecessary given the availability of structured data. We essentially treat the OS as the “sensor provider” rather than a camera.

- **Continuous Learning and Adaptation:** Initially, rules and the SLM are based on expected usage patterns. Over time, the agent can learn user-specific cues. For example, if the user often names files in a certain way (maybe “Contract - <Client>” vs. just “contract”), the agent can pick that up. Or if certain web domains are work-related vs personal. This learning could be done on-device by storing a small knowledge base (e.g. a list of keywords that indicate a contract, which the user can edit). The perception strategy, however, remains mostly general to avoid overfitting – it should work out-of-the-box for common scenarios.

In summary, the perception strategy is a **framework that turns raw OS events into semantic context** in a fast, lightweight manner. By fusing multiple hints (app, title, input, etc.), the agent forms a robust picture of user activity. This allows it to make statements like, *“User is editing a legal contract in Word (last active 5 min), has Slack open (no recent activity), and is likely focusing on the contract.”* These insights can trigger appropriate agent behaviors (like suggesting contract clause summaries during idle moments). Crucially, all of this is achieved **without dense visual processing** – we exploit the fact that Windows and applications already expose *what* the user is doing in their UI structures. The result is a real-time, low-resource perception capability.

5. SLM → LLM Escalation Logic (When and How to Go Big)

While the edge SLM and event logic handle most context recognition, there are moments when a **large language model (LLM)** is needed to provide deeper understanding or generate complex outputs. This section defines **when** to escalate to the LLM (GPT-5.1 or similar) and the design of those escalations: how to package the query ("messaging design") and how to use caching to avoid redundancy. The objective is to use the LLM **strategically** – leveraging its power for heavy tasks (summarization, planning, semantic reasoning) but **not** bogging down the system with frequent or unnecessary LLM calls.

- **Escalation Triggers (When to Invoke LLM):** The agent should escalate in the following situations:
- **Complex Content Summarization:** The user opens or dwells on a content-rich item that would benefit from summarization beyond the SLM's capability. Example: a 30-page PDF contract, a long email thread, or a dense web article. The small model might identify "this is a legal document" but cannot itself produce a summary or insights from it – that's when GPT-5.1 is brought in to summarize or extract key points. Trigger logic: e.g. if a document window stays open and focused for >N minutes (indicating user's reading it), or if the user explicitly scrolls through it quickly (implying scanning), escalate by asking LLM for a summary or "important highlights."
- **Multi-step Planning or Synthesis:** The user's context indicates a possible complex goal. For instance, the user has an email draft open referencing "attached contract" and the contract file open side-by-side – the agent might infer the user intends to send the contract with a summary. Generating a polished summary or drafting an email could be done by the LLM. Another example: user toggles between a spreadsheet and a PowerPoint – perhaps preparing a presentation of data. The agent could enlist GPT-5.1 to outline a presentation based on the data (if asked or if it detects user might benefit). These are creative leaps the SLM alone cannot do reliably.
- **Ambiguous Situations Requiring World Knowledge:** Sometimes the signals might not clearly map to a known state or the user might ask a spontaneous question. If the user invokes the assistant ("What does this error dialog mean?") or the agent is configured for proactive help and sees something like an error message or complex UI (e.g. a stack trace on screen), a large model can be used to interpret it (the agent could send the text of the error to GPT-5.1 for explanation). Another case: if an unfamiliar application is in focus and the agent doesn't have rules for it, GPT-5.1 might be queried: "What kind of app is XYZ.exe?" by providing context (maybe based on window title or a quick web search).
- **User Request or Explicit Prompt:** Of course, if the user asks the agent something that requires heavy lifting ("Summarize this document" or "Plan my day based on my calendar and emails"), the agent will escalate that request to the LLM.

Importantly, **routine context updates will NOT trigger LLM calls.** Simply switching apps or typing normally shouldn't involve the LLM. This ensures day-to-day latency stays sub-100 ms, and cloud usage (which might be ~1–2 seconds per call or more) happens only for value-added tasks.

- **Messaging Design (How to Package Context for LLM):** When escalation is triggered, the agent prepares a concise and informative **prompt/package** for the LLM. This includes:
- *Relevant Context:* The agent compiles the necessary information the LLM needs. For a document summary, this would be the text or a chunk of it (possibly after some compression by the SLM or by selecting only certain sections). For a planning task, it might include a list of open tasks or data gleaned from multiple windows. We ensure to only include information the user has access to or that is relevant (to avoid prompt bloat).

- *Instruction:* A clear instruction to the LLM about what is needed. We might have pre-defined prompt templates. For example: *"You are an AI assistant embedded in a desktop. The user is reading a document titled 'NDA_Contract.docx'. Summarize the key points of this contract in bullet form."* The instructions will be framed so that the LLM's output is directly useful and not too verbose (since we may display it in a UI tooltip or sidebar). We will also instruct the LLM to be mindful of any privacy or tone considerations (if needed).
- *Format and Role:* The prompt might adopt a specific role or style if the agent requires. E.g. *"Role: Legal assistant. Task: Summarize… Keep it concise."* However, these details will be refined with testing GPT-5.1's behavior. The messaging should also establish what the LLM can see (the provided context) and cannot (it shouldn't hallucinate access to anything beyond).
- *Example:* For the Slack and contract scenario: if the user asks "Can you draft a summary of this contract for Slack?", the agent might gather the contract text, then prompt LLM: *"The user needs a short summary of the following contract to share on Slack. Summarize the contract's key points in 5 sentences:"* followed by the contract text (or key clauses extracted).

This packaging is done quickly on the agent side – potentially the most time-consuming part is collecting the text from the UI. We might use caching (below) to mitigate repeated data gathering. Also, the agent will sanitize or truncate context to fit LLM input limits (ensuring important parts are included first).

- **Caching Strategy:** To avoid redundant LLM calls and data transmissions, the agent will employ caching on multiple levels:
- *Context Cache:* If the user is working with a particular document or data repeatedly, we cache the processed form. For example, if we already summarized *NDA_Contract.docx* once, store that summary (and perhaps an embedding). Next time if the user revisits that document or requests a similar summary, we can reuse or at least provide the cached summary instantly (and maybe refresh it in background if content changed). We might maintain a small database keyed by document name + last-modified timestamp to know if our summary is up-to-date.
- *LLM Query Cache:* For certain prompt + input combinations, store the LLM's response. For instance, if the agent has a habit of asking "what app is X?" for unknown processes, we cache those answers. So the first time it might ask GPT-5.1 "What is XYZ.exe?" and get an answer ("XYZ.exe is a VPN client app"), then store that. Next time the same process appears, no need to ask again.
- *Rate-limiting and Debouncing:* If a trigger condition persists, we avoid hammering the LLM. For example, if the user is scrolling through a long document, we don't want to resend the entire document on every scroll. Instead, perhaps we wait until the scrolling stops (user is likely reading) then send one summary request. Similarly, if the user toggles rapidly between two tasks that would each warrant an LLM call, we might delay to see if they settle on one.
- *Local Summaries and Incremental Updates:* We can also cache intermediate representations. Suppose we already got a summary of each section of a document (maybe using the SLM to identify sections and then LLM to summarize each). If the user later asks for the full summary, the agent can compile from the section summaries rather than sending everything anew. This is more of a sophisticated optimization which can be explored if usage patterns demand it.

- *Memory of LLM Output:* The agent can incorporate the LLM outputs into its context. E.g., after summarizing a contract, the agent "knows" the summary. If the user later asks a question about that contract, the agent can provide the summary to the LLM along with the specific question, rather than the full text again – this reduces token usage and speeds up response.

- **Escalation and User Experience:** When an escalation happens, the agent's integration (DesktopAI UI) should handle it gracefully. Usually, LLM calls are asynchronous (taking 1-3 seconds). During that time, the agent might show a subtle indicator ("Thinking..." or no indicator if it's in background mode). It's important that *escalations do not block the real-time pipeline*. Our design ensures that the local event handling keeps running (the user might keep working, generating events) while the LLM operation is in progress. The decision engine can handle this by having separate threads or async tasks for LLM queries. We also design the agent to be resilient: if an LLM call fails or is slow (network hiccup), it should time out and perhaps retry or defer that help. The user's workflow should not be stalled waiting for an LLM.

- **Comparison to Other Agents:** Notably, some contemporary systems like OpenAI's "Operator" rely heavily on the LLM for every step (it uses GPT-4 Vision to literally see the screen every time it needs to understand UI [15]). Our approach consciously *defers* to LLM only when simpler methods run out. This leads to far fewer LLM calls. For example, where Operator might send a screenshot on each new window, we send nothing to the cloud until a high-level query or summary is needed. This difference is huge for latency and privacy. Anthropic's Claude with "computer use" also currently exhibits multi-second latencies and sometimes even minutes for UI tasks [16], partly because it's orchestrating complex tool use with the model in loop. Our design keeps the model *out of loop* for as much as possible, achieving responsiveness. When we do bring GPT-5.1 in, it's for its unique talents (language understanding & generation at a high semantic level), not for basic perceptual tasks.

- **Security & Privacy in Escalation:** Since data is being sent to a cloud LLM potentially, we incorporate user controls and safe handling. The agent will not escalate confidential content without user consent or a clear need. Possibly, in enterprise settings, the LLM could be an on-prem model to keep data local. Our messaging design will include redaction of any sensitive personal identifiers if not needed for the task, and caching ensures we don't repeatedly transmit the same sensitive data.

In summary, **escalation logic** ensures the LLM is used *like a specialist consultant*: rarely, but effectively. The agent handles the day-to-day observations, and when something big comes up or the user explicitly asks, it formulates a concise question for the big model. By carefully designing the prompts and caching results, we minimize cost and latency. The user thus experiences the best of both worlds – a snappy local response for routine context (thanks to the SLM and rules) and the wisdom of an advanced AI when dealing with complex tasks – all integrated seamlessly.

6. Existing Work Survey (Open Source, Research & Startups)

Building a real-time desktop AI agent that eschews heavy vision is a cutting-edge endeavor, but several projects and research efforts align with or inform our goals. We surveyed open-source projects, academic research, and startup products that address similar challenges: desktop automation, context sensing, and lightweight observation. Below is a summary of relevant work and how we can learn from or integrate with them:

- **PyWinAssistant (open-source project):** *PyWinAssistant* is a notable open-source implementation of a Windows "Computer-Using Agent" that explicitly avoids computer vision, using Windows Accessibility APIs instead [13] [9]. It can operate GUIs via natural language commands, utilizing UI Automation under the hood to perceive and manipulate UI elements. PyWinAssistant's key achievement is demonstrating that **pure symbolic/spatial reasoning on the GUI** is possible – it was the first agent framework to fully bypass OCR/pixel methods for GUI automation [17]. It tracks UI hierarchy, control types, and uses chain-of-thought reasoning to plan GUI actions. While its focus is more on executing tasks (it actually moves the mouse, clicks buttons in response to user instructions), the perception component (understanding what is on screen) is directly relevant. For example, PyWinAssistant can enumerate all open windows and their controls via UIA, and maintain a representation of the GUI state. We can draw on its strategies for mapping UI element properties to semantics (e.g. recognizing a "Save" button by its name/property) and its event handling loop. However, PyWinAssistant is implemented in Python, which wouldn't meet our latency requirements – but it provides a high-level blueprint. We might also leverage its data like common GUI element patterns or even integrate some of its code for non-performance-critical pieces. Suitability: It's a strong validation of our approach, though for integration we'd likely reimplement its ideas in Rust/C++ for speed. License is MIT, so we can reuse concepts freely [18].

- **Windows MCP Automation Server (Mario Andréschak's project via Anthropic's MCP):** This is a project implementing Anthropic's **Model-Context Protocol (MCP)** for Windows GUI automation [19] [20]. It's essentially a server that an AI agent can send JSON-RPC commands to, and the server executes those on the Windows desktop (click, type, etc.), bridging the gap between an LLM's intent and GUI actions [21]. Technologically, it uses Node.js with an AutoIt DLL interface [22] – AutoIt provides functions like finding windows, clicking controls, reading text, etc., by leveraging the Windows API and some OCR when needed [23] [24]. This project is more about **control** than perception, but it inherently has to *get* information from the UI to do its job (for example, `controlGetText` to read a control's text). It thus overlaps with our perception needs. We likely don't want to use Node/AutoIt in our stack (due to performance considerations), but the **MCP protocol** idea is interesting for integration (see section 8). We could design our agent to expose a similar interface for obtaining context, making it compatible with frameworks like BlackBox AI or others that speak MCP [25]. In terms of existing work survey, this confirms that:

- The idea of an AI "seeing and controlling" the desktop is actively being pursued by industry (Anthropic, OpenAI).
- Their initial solutions have leaned on *screenshots and vision* (Anthropic's computer-use involves looking at screens [26], OpenAI's Operator uses GPT-4's vision on screenshots [15]). These achieve functionality but at the cost of speed. For instance, early users have reported noticeable latency (Claude's approach sometimes taking many seconds per action as per anecdotal evidence).
- The MCP Windows server, by using AutoIt, likely speeds things up by using UI info when available, only resorting to screenshots if necessary [24]. This hybrid approach is in line with our philosophy: rely on programmatic UI info primarily, and only escalate to vision as last resort.
- We can learn from the categories of functions it exposes (as listed in the article's table [27] [28]) to ensure our perception agent gathers data relevant to those functions (like window lists, control texts).

- Suitability: We might not directly integrate their Node.js server, but we ensure interoperability (maybe our agent can act as an MCP server or client). The project is open-source (under Mario's GitHub), so we can examine its implementation details for pointers (e.g., how it identifies UI controls robustly, how it uses AutoIt's window matching – which uses window titles and control IDs similar to UIA).

- **Mediar's `ui-events` (Rust library):** This is a young project aiming to provide a cross-platform Rust library to stream UI events (macOS, Windows, Linux) via a WebSocket to AI consumers [29] [4]. Essentially, it's doing part of what our agent needs: capturing focus changes, window events, etc. On Windows it plans to use UI Automation events [30]. The fact it is cross-platform and Rust means it aligns very well. If it's sufficiently advanced, we could use `ui-events` as a component or at least as inspiration. It already defines an **architecture** of a Rust core event listener feeding events to a client over a socket [31]. That architecture is similar to our design (except we might keep the processing in-process rather than via WebSocket, but we could expose a WS if needed for integration). The library's focus is on **UI focus events, window create/destroy, value changes** and explicitly not on raw input [32] [33]. This matches our emphasis on accessibility events.

- Suitability: We should track this project's progress. If ready, we could save development time by integrating it for the event capturing layer. At minimum, its cross-platform abstraction might be overkill for us (we only target Windows), but it confirms that a *performant Rust core for UI events* is viable and being worked on. It's MIT licensed, so usable. Even if not used directly, any gotchas they discovered (like handling UIA threading or specific event ordering) could inform our implementation.

- **AgentSight (Research, 2025):** *AgentSight* [34] [35] is an academic framework for observability of AI agents, which uses eBPF (in Linux) to monitor system calls and correlates them with LLM prompts, aiming to detect anomalies. While its domain (LLM agent safety) differs, it embodies the philosophy of using **system-level signals for semantic insights**. AgentSight shows that by capturing events at a low level (system calls, network traffic) and doing real-time correlation, one can achieve rich understanding with minimal overhead [6] [7]. This resonates with our approach of using OS telemetry as a reliable, low-overhead source, rather than expensive high-level observation. The difference is AgentSight then uses an LLM to explain those traces (they essentially do LLM-on-LLM monitoring). For our purposes, the takeaway is: **efficient real-time instrumentation is feasible** (AgentSight adds <3% overhead on a system by using eBPF for events). On Windows, we have ETW and hooks analogously. So research backs that the performance budget is realistic.

- Suitability: Not directly integrable (AgentSight is Linux and more ops-focused), but conceptually validating. It also suggests potential future enhancements: e.g., using an LLM to analyze *streams of events* for patterns. Currently our agent uses SLM and simple logic for patterns, but perhaps an on-device medium model could observe a sequence (like "user did X then Y repeatedly") and suggest an automation. This is speculative, but AgentSight's idea of semantic analysis on event traces could inspire a future feature where our agent notices inefficiencies or errors and suggests fixes (somewhat orthogonal to core context tracking though).

- **Commercial "Desktop AI" products:** There are emerging products (like BlackBox AI's Desktop Agent, Replite's upcoming agent, MS Copilot on Windows) that claim to integrate AI with desktop use. BlackBox AI, for instance, markets natural language control of the desktop, and it uses an *MCP client* architecture (meaning it likely interfaces with an MCP server or similar) [25]. Microsoft's Windows Copilot is internally using the capabilities of Windows (probably leveraging accessibility, UWP APIs, etc.) but details are not public. The key point for us is that **the trend is clear**: Instead of purely vision-based RPA, tools are moving toward *integrated OS-level approaches*. Even OpenAI's Operator, while vision-heavy, is a stepping stone – they integrated it into ChatGPT for agent mode, meaning the idea of an AI controlling your PC is mainstream now [36] [37]. Our agent's unique angle is focusing on *perception* more than control, and doing so with minimal overhead.

- Potential collaboration: If our agent provides a robust context API, it could be used alongside these systems. For example, a ChatGPT-based agent might query our agent for "what's the user doing" to better respond (since large models themselves don't have stateful memory of user context unless fed). Startups might also be interested in the signal-fusion tech we develop to incorporate into their products for faster performance.

- **RPA and Automation Tools:** Traditional RPA software (UiPath, Automation Anywhere, Microsoft Power Automate Desktop) use a combination of UI Automation, image matching, and scripting to automate GUIs. They aren't "AI perception" tools per se, but they have modules to **identify UI elements** (via selectors or OCR). Microsoft's Power Automate Desktop uses UIA under the covers for many actions and has recently improved performance by moving to UIA events instead of polling in some cases. These tools demonstrate what can be done with UIA: e.g., reading text from almost any standard control, clicking by automation IDs, etc. The performance of such actions is usually fine (sub-second). The drawback is they often require the user to configure flows manually. However, they sometimes include **recorders** that watch user actions and then generate automation scripts. Our perception agent shares the "watch user actions" part – except we aim to interpret them, not just record blindly. Perhaps in future, our agent could even generate RPA flows if it notices repetitive tasks (that would involve LLM planning and so on).

- In terms of existing work, these RPA tools confirm that using the Windows UI Automation and input simulation stack can fully replicate user actions (so by extension, our perception of those actions via the same stack is feasible). They also highlight the importance of robust selector logic – to identify the same button reliably. While not directly needed for perception, if we eventually mark certain UI elements in context (like "the 'Send' button is currently disabled"), we may borrow selector strategies from RPA frameworks.

- **OpenAI's Operator (ChatGPT Agent mode):** Operator is OpenAI's agent that uses a **Computer-Using Agent (CUA)** model to control a browser via vision and actions [38]. It effectively sees the browser through screenshots and can click/type. Why it matters for us: Operator demonstrates the *upper-bound* approach (heavy model, heavy vision) – our agent is almost the opposite in strategy. Operator's initial release is browser-focused and uses GPT-4 with vision for each step, which is powerful but not lightweight [15]. We expect our system to vastly outperform it in responsiveness for recognized patterns (e.g. our agent can detect a Slack message instantly via event, whereas a vision-based one would have to "read" the screen, consuming time and compute).

- Nevertheless, the existence of Operator indicates user demand for such capabilities. It also means by the time we deliver, users may have seen the slow version and will appreciate a faster one. Integration-wise, nothing direct, but we might ensure our agent can feed context into ChatGPT's agent mode if needed (for instance, instead of ChatGPT taking a screenshot to know what's going on, it could query our agent for a structured description of the screen – a possible extension that bridges approaches).

Summary of Survey: We have strong validation that **system-level signals + AI** is the future of desktop agents: - PyWinAssistant and similar open projects prove we can operate without pixels [17]. - Research like AgentSight and industry protocols like MCP indicate a move toward event-driven, semantic monitoring and standardized tooling [6] [21]. - Our job is to combine these insights into a cohesive, highly-optimized product. We will stand on the shoulders of these works by using their best ideas (UIA for perception, small models for quick reasoning, caching and event streams for efficiency) and avoiding their pitfalls (e.g. Python's slowness in PyWinAssistant, or heavy screenshot reliance in Operator).

Where possible, we'll collaborate or align with open standards (like MCP) to make our system integrable. But in terms of core tech, our approach appears to be novel in its **extreme focus on low-latency edge processing**. This could set us apart as the only solution that truly achieves the sub-100 ms insight loop on Windows.

7. Performance Targets

To ensure the agent meets the "lightweight, real-time" requirement, we establish concrete performance targets across latency, throughput, resource usage, and overhead. These targets will guide implementation and act as acceptance criteria in testing:

- **Latency:** *User action to model insight in under 100 ms (end-to-end).* This is the headline objective. Breaking it down:
- **Event Detection Latency:** The time from a user generating an event (e.g. pressing Alt+Tab to switch windows, or focusing a window) to our agent *receiving* the event. Using OS hooks, this is typically on the order of 1–5 ms (the OS dispatches events almost immediately to listeners). Our target is to not add more than a few milliseconds on top of OS notification. So, *target:* **<10 ms** from OS event to our internal event handler.
- **Processing + Inference Latency:** Once an event is received, the time to update state or produce any output (not involving cloud). This includes any UIA queries (which might take a few ms if cross-process) and running the SLM classification. We aim for **<50 ms** for most single-event processing. UIA calls are usually fast for simple property grabs, but e.g. retrieving large text could take longer – in those cases, we might do partial loading or offload to a background thread. The SLM model (if neural) will be a small one; we target its inference time at **<20 ms** on a modern CPU for a single input (this is feasible for models in the 100M parameter range or smaller, especially if quantized or using accelerators).
- **Overall Insight Latency:** Combining the above plus any slight queuing, we want the majority of context updates (say 95th percentile) under 100 ms. Many will be far faster (~30 ms). Only rarely (like on a first load of a heavy document where we choose to quickly scan content) might it approach or slightly exceed 100 ms, but even then ideally under 200 ms.

- **LLM Response Latency:** When we do call out to GPT-5.1, obviously that depends on network and model speed. This can be ~1–3 seconds typically. We will not consider this in the 100 ms budget since it's an asynchronous, infrequent step. However, we do set a target that any user-visible LLM operation should ideally complete in **<5 s** (worst-case), with typical tasks ~2 s. And the agent will be designed to hide this latency as much as possible by doing them asynchronously (so from user perspective, it's like a background fetch that then pops up). Caching results as discussed will also reduce repeated latency.

- **Throughput & Scalability:** *Ability to handle event bursts without dropping events or slowing down significantly.* In normal usage, events are relatively sparse (a few per second), but consider edge cases: user holding down a key (which generates many key events), or a flurry of UI events (opening a window with many child controls can fire multiple events).

- **Event Throughput:** The agent should comfortably handle **100 events per second** sustained. This is a high bar (much higher than typical UI event rates), but it ensures headroom. In testing, we might simulate a user mashing keys or a script opening many windows to ensure we don't overflow queues. Our architecture using multi-threading or async should allow concurrent handling (e.g., keyboard events processed on one thread, UIA events on another).
- **Pipelining:** If events come in while the SLM is processing the previous one, we won't stall input capture – events will queue briefly. The design target is that even under load, the queue latency stays low. For example, under a burst of 50 events in a second, none should experience more than 50 ms queue delay. We may prioritize certain events (focus changes might flush other minor events) to maintain relevant context.

- **Scaling with Cores:** The agent should utilize multiple CPU cores when available. For instance, we might parse UI events on one core and run the SLM on another. Our target is to keep each core's load modest so that the agent never maxes out a CPU core for extended periods.

- **CPU Utilization:** *Keep CPU usage low to avoid impacting user's system.*

- **Idle Overhead:** When the user is idle or not doing much (or just typing in a single app steadily), the agent should consume **~0% CPU** (effectively sleeping until an event arrives). The event hooks are passive and wake the process only on events, so idle overhead should be near zero. We target <1% of one core usage on average when events are infrequent.
- **Active Overhead:** During normal active use (user clicking, typing, switching apps moderately), we target **<5% of one CPU core** on average for the agent. This means the user shouldn't notice any slowdowns. Peak spikes might go higher (e.g. a complex UIA tree parse might spike to ~10-20% for 50 ms), but sustained usage should be low. We explicitly avoid heavy loops or polling; everything is event-driven to keep CPU usage minimal.

- **Content Analysis:** If we do heavier work (like reading a large text from UI to send to LLM), we might briefly use more CPU (e.g. parsing text). But those will be sporadic and likely bounded (reading 100KB of text might use some CPU for maybe 100 ms – still not huge). If we find such tasks heavy, we may spawn them in background with a lower process priority to not steal time from user's app.

- **Memory Footprint:** *Fit in memory comfortably, even on lean systems.*

- **Base Memory:** The core runtime (Rust/Go/C++) should ideally start with a small memory footprint – target **<100 MB** resident memory for the agent when idle. This includes any loaded ML model. If our SLM is small (say a few tens of MB for model weights), plus some overhead for UIA libraries, etc., 100 MB is reasonable. If we use a slightly larger local model, it might go up; we set an upper target at **<250 MB** even in heavy use. We want the agent to be viable on 8 GB RAM machines without issue.
- **Caching and Data:** If caching summaries or storing context, we should do so efficiently. Text summaries are small (kilobytes). If we keep an embedding store, that could grow, but likely it's tiny relative to LLM size. We'll also allow some cache size limit (e.g. keep last N documents' summaries).

- **Avoiding Leaks:** Thanks to using safe languages (or careful C++), we target zero memory leaks. Long uptime (days/weeks) should not see memory creep. We will test with long runs.

- **Disk and Startup:** The agent binary size target is **<50 MB** (likely achievable, Rust or Go might be ~10-20 MB, plus model file if separate). Cold start time target is **<2 seconds** (the bulk might be loading the SLM model from disk if not using on-demand loading). Once started, it will run continuously, so startup isn't critical beyond the first launch or system boot.

- **Determinism & Jitter:** We aim for *predictable performance*. This is why we favor no GC and event-driven design. The user's actions themselves might not be strictly periodic, but if they perform an action repeatedly, the agent's response time should not vary widely. Ideally, the standard deviation of our response latency is low (e.g. if we say 50 ms average, maybe ±10 ms variance typically).

- We will measure 95th and 99th percentile latencies. The goal is 99th percentile still under, say, 150 ms for local processing. Any outliers beyond that should be extremely rare (maybe first-time setup of a particular UIA interface or a sudden OS hiccup).
- We also consider *multitasking scenarios*: e.g., user triggers an LLM summary while quickly switching to another window. The agent should continue capturing events while the LLM thread works. This concurrent handling ensures no jitter introduced by waiting for an LLM call to finish, etc. Proper asynchronous design will ensure determinism for high-priority tasks (like immediate context updates).

- **Testing Performance:** We will include performance tests such as:

- Simulated rapid user interactions (scripted sequences) to see if any backlog occurs.
- Memory and CPU profiling under heavy use (open 10 apps, switch rapidly, type in each).
- Long-duration soak tests to ensure no gradual slowdown or leak.
- Specific tests around known expensive operations: e.g., measure how long retrieving a large text via UIA takes and optimize if needed (maybe fetch only parts of text).
- These targets are documented so that if any metric fails (say, average latency is 120 ms), we treat it as a bug to fix via optimizations.

In summary, the performance targets ensure the agent truly feels **lightweight**: it should not perceptibly drag the system or lag behind the user. Achieving sub-100 ms means interactions feel instantaneous (100 ms is roughly the threshold where humans start to feel delay). By hitting these numbers, we differentiate from heavier solutions (that might take seconds or eat CPU with vision). This PRD's focus on events and native code is precisely to meet these targets, and we will use them to evaluate our success.

8. Integration Plan (DesktopAI Environment Integration)

Finally, we outline how this perception agent will be **integrated into a larger DesktopAI or agentic environment**. The assumption is that we have (or are building) an overarching system (called "DesktopAI" here) that provides user-facing interfaces (like a chatbot window or command palette) and orchestrates AI assistants and tools. Our perception module will serve as a **context provider and possibly an automation helper** within that ecosystem. Key integration considerations:

- **Modular Service Design:** We will package the perception agent as a **modular service** that can run independently but communicates with the main DesktopAI app. Concretely, this could be:
- A background process (Windows .exe) that starts on user login (we can register it to autostart, possibly with a tray icon for status).

- The DesktopAI app (which might be an Electron app, or a .NET app, etc.) can connect to this process via an IPC mechanism. Options include:
    - **WebSocket/HTTP interface:** For example, adopting the *MCP (Model Context Protocol)* as mentioned. We could implement an MCP server, allowing any AI client supporting MCP to query context or request actions. BlackBox AI's agent uses MCP, so aligning with that could make our solution plug-and-play for such clients [25] [39]. MCP defines how an AI can discover tools and call them via JSON-RPC [40]; we would define a set of context "tools" (like `getActiveWindowInfo`, `summarizeDocument`, etc.) accessible through the protocol.
    - **Native messaging or gRPC:** If DesktopAI is a native app, we could use shared memory or named pipes. But a WebSocket/HTTP API has the advantage of being language-agnostic.
    - **Plugin integration:** If DesktopAI is based on an existing AI platform (say a special build of ChatGPT or a local UI), we might integrate by writing a plugin for that platform which communicates with our agent. For example, if DesktopAI had a Python backend, our agent could provide a Python API wrapper that calls into the service.

- The key is *loose coupling*: the perception service should be able to run and provide data without being tightly bound to UI, so it can be updated or restarted independently.

- **Context Data Sharing:** The perception agent will continuously update a **context state** that the main AI assistant can use when generating responses. Integration strategy for this:

- We maintain a structured representation of context (could be a JSON object) including things like:

```
{
  "active_app": "Microsoft Word",
  "active_title": "NDA_Contract.docx - Word",
  "active_state": "Editing a contract document",
  "other_open_apps": ["Slack", "Chrome"],
  "recent_activity": "User copied text from Word at 3:45pm",
  "suggestions": ["Summarize contract", "Remind to send email"]
}
```

  The assistant can query this on-demand or subscribe to updates.
- If DesktopAI has a prompt pipeline, it might insert context into the LLM's system prompt. For instance: *"The user is currently editing a contract in Word. Keep responses brief to not disturb too much."* or the assistant might proactively use it to tailor suggestions.
- We likely implement a **publish-subscribe** model: DesktopAI registers interest in context updates, and our agent pushes updates (debounced, maybe when state changes significantly) to it. For example, when the user switches to Slack, the agent sends an update "state changed: now in Slack (communications)".
- This allows the assistant UI to maybe display the context too (some UIs have a "what's the AI seeing?" panel).

- We must be cautious to not overload the assistant with too frequent updates. Integration testing will find the right balance (maybe update context at most a couple times per second, and aggregate minor changes).

- **User Consent & Controls:** Integration includes user-facing controls. DesktopAI should have settings for the perception agent: e.g.

- An on/off toggle: user can disable the agent (it should then stop capturing events). This might be needed for privacy or performance concerns (though we aim to alleviate those).
- Granular controls: maybe turn off certain signals (e.g. "don't monitor my keystrokes" – the agent could then not hook keyboard, at cost of not detecting idle or not capturing typed content for summary).
- Privacy mode: perhaps pause all monitoring when user is in a sensitive app (like a password manager). We can maintain a blacklist of processes where the agent automatically pauses event handling. The integration config can expose that list to users to edit.

- These controls would be surfaced in DesktopAI's settings UI, with our agent providing the ability to implement them (the agent could expose an API like `setMonitoring(enabled)` or a config file that DesktopAI writes to).

- **Security & Permissions:** On Windows, to access UI Automation and global hooks, typically no special install is needed if running as the logged-in user. But:

- If DesktopAI runs in user mode (non-admin), it can't inspect admin apps as noted. Our agent could run as user by default. For full control, user might opt to run it as admin (we'd provide guidance or an option).
- Code signing: We will sign the agent binary to avoid it looking suspicious (global hooks from unsigned code can trigger antivirus).
- Integration testing on corporate environments (where certain accessibility might be locked down) will be needed. Possibly providing a fallback or limited functionality mode if some APIs are unavailable.

- We should also ensure the communication channel between DesktopAI and the agent is secure (especially if using WebSocket/HTTP, it should be on localhost only and with an auth token or similar to prevent other processes from spoofing requests).

- **Agent Actions via Integration:** While our focus is perception, integration with DesktopAI raises the question of controlling the desktop (like an RPA) if needed. Our agent could be extended to not just observe but also act (through UIA or synthetic input) upon requests from the assistant. This is more of a phase-2 capability, but we should architect with this in mind. For example, if the assistant says "Closing Slack for you to minimize distractions," it could call an action on our agent to close the Slack window. Since our agent already knows window handles and uses UIA, implementing an action like `activateWindow` or `sendKeys` is straightforward. The integration plan would then include defining a set of allowed actions (with safety checks). MCP, as mentioned, enumerates such actions (mouse clicks, keystrokes, window management [27]). We could align with those so that any AI using MCP can use our agent as the backend tool executor.

- Initially, if we're cautious, we might run in a read-only mode (perception only) and slowly roll out action capabilities as confidence grows. DesktopAI's UI could ask user confirmation for any action the agent is about to do, which BlackBox AI's agent supports (different autonomy levels [41]).

- **Integration with LLM Orchestration:** DesktopAI likely manages the LLM interactions (maybe it calls OpenAI/Anthropic APIs or runs local models). Our agent's escalation to LLM (section 5) can either go through DesktopAI (i.e., when our agent decides to escalate, it sends a request to DesktopAI's LLM module which then returns the result to us), or the agent might even call the LLM API directly. We need to decide integration here:

- If DesktopAI already has an open channel to GPT-5.1 (with API keys, usage tracking, etc.), it makes sense to route our LLM queries through it. We'd essentially send a message like "Need summary of X" and DesktopAI would respond with the summary after contacting the LLM. This centralizes AI API usage.
- Alternatively, our agent could have its own LLM client (especially if using a local model or a different service). This is simpler to implement internally but could complicate things if, say, rate limits or billing need to be coordinated.
- As a product decision, likely we integrate via the main app. So in the escalation logic, instead of calling GPT-5.1 directly, the agent raises an "assist request" event that DesktopAI handles by invoking the LLM and then passes back the answer to us or directly to the user interface.

- Caching can also be coordinated – e.g., DesktopAI might have caching for LLM calls that we can utilize.

- **Testing in DesktopAI workflows:** We will test integration in realistic scenarios:

- User asks the DesktopAI assistant, "What is this document about?" while a Word doc is open. The flow: the assistant (LLM) doesn't know the document -> it queries our agent (via something like a tool usage), our agent returns either a summary if we have cached or the raw text or triggers an LLM call -> final answer is produced. We want this to be smooth.
- Agent proactively offering help: e.g., user stops typing, agent (perception) notices and perhaps triggers DesktopAI UI to show "Need help summarizing what you've written?". This needs a defined channel for agent-initiated suggestions to the UI. Possibly the perception agent can send "suggestion" events that DesktopAI UI will display as a notification or prompt.

- Multi-modal integration: If DesktopAI has voice or other channels, ensure our agent works headlessly as well (it doesn't require its own GUI). Possibly our agent logs context that can be included in voice responses.

- **Documentation & DevX:** For integration, we will document the API our agent provides (whether it's WebSocket messages, function calls, etc.), so that other developers (if DesktopAI is extensible) can utilize it. If we align with something like MCP or OpenTelemetry for GenAI context, that will make it more standard.

- **Future Integration (Extensibility):** Our perception agent could become a platform for further extensions:

- Other modules could plug in new signal sources (for instance, if one wanted to monitor specific app APIs or a database of user documents to cross-reference).
- We'll ensure the architecture allows adding such modules (maybe via a plugin interface or at least clean code separation).

- DesktopAI might eventually merge the perception agent functionality directly (but keeping it modular initially is best for development and possible open-sourcing).

Integration Plan Summary: We will deploy the perception agent as a **background Windows service/ process** that **feeds real-time user context** to the DesktopAI agent. Communication likely via a local WebSocket or RPC, using structured data (JSON). The DesktopAI front-end/back-end will incorporate this context into the AI's decision-making and user interactions. We'll provide controls for the user to manage this feature. By designing to a standard protocol (or at least a documented API), we ensure our agent can integrate not just with our DesktopAI but potentially with others (fostering an ecosystem, e.g. other AI agents could consume our context feed instead of reinventing that wheel). The integration will be tested thoroughly to ensure it enhances the user experience (e.g. more personalized, context-aware AI responses) without causing confusion or security issues.

In conclusion, this PRD lays out a full vision for a **Windows AI Perception Agent** that is modular, efficient, and deeply integrated. It leverages Windows' native capabilities to observe user context in real-time, uses edge computation to derive meaning quickly, and smartly escalates to powerful AI only when needed. By integrating with the DesktopAI environment, it provides the "eyes and ears" for the user's AI assistant, enabling a truly intelligent and responsive desktop experience. All requirements – from system architecture to performance and integration – align to meet the core objectives of <100 ms latency, low resource use, and rich context awareness, setting the stage for the next generation of agentic computing on the desktop.

Sources: [1] [13] [17] [4] [15] [23] [24] [2]

---

[1] [10] [11] UI Automation Events Overview - .NET Framework | Microsoft Learn
https://learn.microsoft.com/en-us/dotnet/framework/ui-automation/ui-automation-events-overview

[2] c# - Reduce time to take a screenshot to 40ms - Stack Overflow
https://stackoverflow.com/questions/55578139/reduce-time-to-take-a-screenshot-to-40ms

[3] Microsoft Active Accessibility and UI Automation Compared - Win32 apps | Microsoft Learn
https://learn.microsoft.com/en-us/windows/win32/winauto/microsoft-active-accessibility-and-ui-automation-compared

[4] [5] [29] [30] [31] [32] [33] GitHub - mediar-ai/ui-events: Library to stream operating system events to AI
https://github.com/mediar-ai/ui-events

[6] [7] [34] [35] AgentSight: System-Level Observability for AI Agents Using eBPF
https://arxiv.org/html/2508.02736v1

[8] [9] [13] [14] [17] [18] GitHub - a-real-ai/pywinassistant: The first open-source Artificial Narrow Intelligence generalist agentic framework Computer-Using-Agent that fully operates graphical-user-interfaces (GUIs) by using only natural language. Uses Visualization-of-Thought and Chain-of-Thought reasoning to elicit spatial reasoning and perception, emulates, plans and simulates synthetic HID interactions.
https://github.com/a-real-ai/pywinassistant

[12] GitHub - DavorMar/rustautogui: Highly optimized GUI automation rust library for controlling the mouse and keyboard, with template matching support.
https://github.com/DavorMar/rustautogui

[15] [36] [37] [38] Introducing Operator | OpenAI
https://openai.com/index/introducing-operator/

[16] Too much latency with computer-use-preview model - Microsoft Q&A
https://learn.microsoft.com/en-us/answers/questions/2261889/too-much-latency-with-computer-use-preview-model

[19] [20] [21] [22] [23] [24] [27] [28] [40] Your AI Agent's Hands: A Deep Dive into the Windows Desktop Automation MCP Server
https://skywork.ai/skypage/en/ai-agent-windows-automation/1978660549309014016

[25] [39] [41] Introduction - BLACKBOX AI
https://docs.blackbox.ai/features/desktop-agent/introduction

[26] Introducing computer use, a new Claude 3.5 Sonnet, and Claude 3.5 Haiku \ Anthropic
https://www.anthropic.com/news/3-5-models-and-computer-use