Explorations of Stress-Adaptive User Interfaces for Productivity Software

Adaptive User Interfaces driven by Galvanic Skin Response and other Biometric Data Signals

SIDNEY GRABOSKY

MS Human-Computer Interaction, Rochester Institute of Technology

This paper explores how biometric sensing of cognitive load signals from wearable IoT sensors can drive stress-adaptive user interfaces in everyday productivity software. I built an ecosystem of polished protype applications such as a Microsoft Word clone and a Slack clone that adapt to user cognitive load and stress levels using various strategies, including progressive interface simplification and context-aware adaptive content triaging; all facilitated by signal processing techniques and artificial intelligence systems. Together, these prototypes illustrate a promising design space for stress-adaptive and stress-reducing dynamic user interfaces using applied real-world workflows.

1 INTRODUCTION

This project involved the creation of a sensing and processing ecosystem of sensor hardware, custom-built software, and a series of applied prototypes that explore different approaches of biometric data based stress-adaptive productivity applications based on common real-world applications. Two applied prototypes were fully realized; adaptive clones of the Microsoft Word word-processing software and the Slack messaging client application, while a prototype of a web browser was not completed due to time constraints.

Biometric data from wearable sensing IoT devices is sent to a central “Biometric Relay” application which processes and interpolates the data, maps it to a synthetic integer value ranging from one and five representing user cognitive load and stress level, and broadcasts this metric as an inter-process notification in MacOS which an arbitrary number of client applications may subscribe to.

Each prototype application explores different methods of how stress-adaptivity could be incorporated into a common productivity application workflow. The concepts were determined after a series of casual short-form interviews regarding common productivity workflows, and what sorts of issues arise as stress levels increase while working. Also considered were issues which exacerbate stress levels further in a positive feedback loop. From these initial findings, the three prototype concepts were developed.

The Word prototype primarily explores adaptive visual simplification; the user interface is progressively simplified and distilled down to its most essential and frequently used components in order to minimize distractions as user stress levels rise. Some of this functionality was also inspired by another application, “iA Writer,” [2] which is a focus-oriented writing application that claims to be effective for individuals with ADHD; iA Writer, however lacks biometric stress-adaptivity. The Slack application primarily explores context-aware notification triage powered by artificial intelligence and a deep understanding of relationships, in order to mitigate the common problem of notification fatigue and context switching. The incomplete web browser application

concept sought to explore progressive implementation of AI content summarization as stress levels rise, the system would help you locate relevant pieces of information on a webpage; it would also intelligently chronologically cluster tabs and windows and fade stale windows into the background to reduce distractions from content that is not actively in use.

2 SYSTEM ARCHITECTURE

This project involved the creation of a sensing and processing ecosystem of sensor hardware, software, and a series of applied productivity application prototypes that adapt their behaviors in response to realtime biometric stress signals.

The Biometric Relay application, running on MacOS, serves as the central processing unit of the ecosystem. This application is responsible for receiving biometric signals from the sensing components, processing the information, mapping it to a computed stress metric ranging from 1-5, and broadcasting that value each second to an arbitrary number of subscriber client applications running on the OS via the MacOS DistributedNotificationCenter [3] inter-process communication dispatch system.

The custom StressLib framework is a dependency of all the MacOS applications and prototypes, this contains common functionality relating to stress levels and notification handling, essential to the adaptivity process.

The Slack prototype and the in-progress web browser prototype communicate with a local artificial intelligence large language model, Ollama, over an HTTP / REST API. This LLM enables powerful analysis of contextual cues, in Slack's case facilitating the calculation of a nuanced context-aware urgency heuristic.

The Apple Watch application senses heartrate BPM. The watch is unable to communicate directly with the Mac laptop application with security and sandbox limitations, so it communicates wirelessly to the laptop using a paired iPhone app as a proxy.

The galvanic skin response sensor communicates through a wired Grove connection to an Arduino Uno microcontroller. The Arduino is in turn connected through a wired USB connection to the Mac laptop, which is able to read GSR data over USB serial directly into the Swift programming environment using the third party library ORSSerialPort. [1]

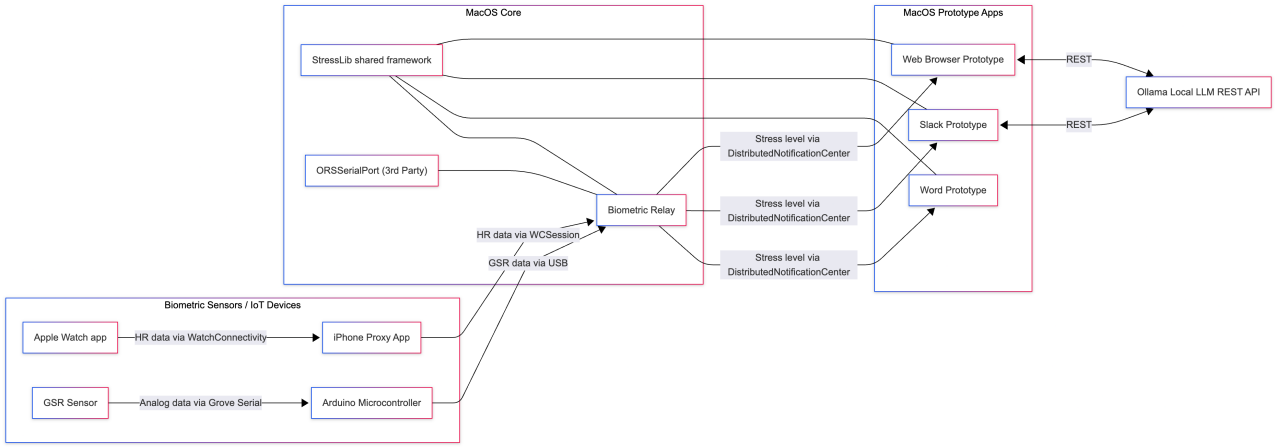

Figure 1 System Diagram

Table 1: System Components and Purposes

Name	Type	Group	Purpose
Apple Watch	Hardware	Biometric / IoT	Wearable device, senses heartrate
Apple Watch App	Software	Biometric / IoT	Software to measure heartrate; networking I/O
iPhone App	Software	Biometric / IoT	Software proxy to bridge data between Watch and Computer
Galvanic Skin Response Sensor	Hardware	Biometric / IoT	Wearable sensor, senses GSR
Arduino Uno Microcontroller	Hardware	Biometric / IoT	Microcontroller for GSR sensor
Arduino GSR Script	Software	Biometric / IoT	Arduino script to read from GSR sensor
Biometric Relay App	Software	MacOS Core	Central processing for reading, dispatching data and stress levels
ORSSerialPort	Software	MacOS Core	3rd party library to read Serial data over USB into Swift
StressLib Framework	Software	MacOS Core	1st party library of common functions shared across system Mac apps
Word Prototype	Software	Prototypes	Adaptive clone prototype of MS Word
Slack Prototype	Software	Prototypes	Adaptive clone prototype of Slack
Web Browser Prototype	Software	Prototypes	Adaptive prototype of a Web Browser. Incomplete.

3 IMPLEMENTATION DETAILS

3.1 MacOS, iOS, and WatchOS Applications: Overall

This project entailed the creation of seven applications in the Apple OS ecosystem: one WatchOS app, one iOS app, and five MacOS apps; the central Biometric Relay app, a shared framework between them called the StressLib, and three prototype productivity applications. Each of these projects were created in the Xcode environment using the Swift programming language and the SwiftUI reactive/declarative view framework, as well as numerous Apple libraries, frameworks, and components that are native to this software ecosystem.

Each of the MacOS applications uses the first party StressLib framework I created as a dependency, preventing code repetition between functionality shared between all the prototype apps. This framework was

published internally using the Swift Package Manager (SPM) system [4] and the Git version control system. It covers functionality including the definitions of the different cognitive load / stress levels, and data structures and API calls for publishing and subscribing to inter-process notifications using the MacOS DistributedNotificationCenter API [3]. It also contains common behaviors for RESTful / HTTP API requests and response-handling for a local artificial intelligent large language model, Ollama. The Slack and Web browser prototypes both make use of this functionality. In the Slack prototype, it powers context-aware notification triage behavior, while in the web browser prototype it powers web content summarization.

The architecture of this system with the Biometric Relay as a central processing hub and notification publisher allows for an arbitrary number of adaptive prototype applications to run on the desktop client operating system with minimal additional overhead. It could also theoretically allow for third party apps to build stress-adaptivity into their applications, though there is almost certainly a more robust and secure way to implement inter-process communication for this use case than the dated DistributedNotificationCenter API; however, this API was perfectly appropriate for a proof of concept.

3.2 Biometric Relay

The Biometric Relay app serves as the central processing component of the system. It was implemented as MacOS “Menu Bar” app; e.g. it lacks a Dock icon, and it’s typically intended to run in the background, daemon style, with no visible user interface. However, there is a user interface implemented that displays system state, customizable settings, and simulator controls to facilitate testing and demos.

The relay receives biometric data from the galvanic response sensor (wired over USB serial, using the third party ORSSerialPort library), and heartrate BPM data from an Apple Watch (wirelessly, over TCP/Wi-Fi 802.11 via an iPhone proxy app). This data is interpretated, mapped, and mixed to produce a synthetic “Stress Level” classification, which is then broadcast out to all subscribing applications on the OS using the DistributedNotificationCenter API.

The most significant behavior that this app does is the interpretation and mixing of the incoming data signals. GSR data comes in as a 0-1023 analog integer value, heart rate data comes in as timestamped heartrates which must be interpreted for a BPM calculation, and neither of these units match. Further, both signals can be messy or prone to jitter, especially the analog GSR data signal.

To deal with these problems, both signals are remapped to 0-1 Doubles and smoothed using an exponential moving average sampling (EMA), this helps to avoid momentary fluctuations and spikes in the signal. When combined, the signals are blended dynamically; the initial ratio is 60% GSR, 40% heartrate; however, very high values for either signal result in a rebalancing of the weights. If one of these signals spikes dramatically even without the other, it’s reasonable to presume that user stress is rising.

Users are able to set their age range, and whether or not their baseline heartrate BPM skews higher or lower than the average, this allows them to get more accurate stress classification than a one-size-fits-all approach. See the Challenges section for much more information on the implementation of demographic-based heartrate zones, baseline customizations, and the Karvonen formula.

Finally, a hysteresis guard is implemented on the raw calculated value before it is published out as a stress level to subscribing applications. In testing, I found that when user stress levels were very near a level boundary (i.e. near the threshold between stress level 4 and 5), the threshold boundary would be crossed back and forth numerous times in rapid succession, causing an extremely undesirable and distracting, rapidly-morphing user interface experience. The hysteresis guard prevents these rapid level changes by inducing some delay and sampling, only changing the stress level after it remains stable for some time, or if it jumps or declines dramatically.

The simulator, used for testing and demos, pseudo-randomly updates hypothetical state for the biometric data every second using a Timer loop. It has a configurable bias to trend up, down, or remain stable; or I may manually set a 1-5 stress level ignoring biometrics entirely to test specific features.

When visible and not in menu bar / daemon mode, the relay UI displays as 3 vertically stacked circle views, respectively displaying the computed 1-5 stress level, the current measured galvanic skin response input (mapped to a percent), and the current heartrate (measured in BPM). On the right size is a panel with settings , system state, simulator controls, connectivity controls to attach to the Watch sensor and Arduino, and dynamic visualization of the heart rate zones based on the user's selected age demographic and baseline heartrate override settings.

Figure 2 Screenshot of Biometric Relay, Settings panel, and simulator settings.

3.3 Word Prototype

The Word prototype makes use of many high-level MacOS and SwiftUI features that made implementing a functioning word processing application relatively easy. The application uses SwiftUI with the DocumentGroup scene type / architecture [5] and FileDocument protocol; a few lines of code automatically creates numerous flows that result in a native document-based application that handles the basics of opening, saving, and serializing documents. Other Apple frameworks and components were essential to building the “WYSIWYG” (what you see is what you get) editing environment. AppKit and NSTextView [6] were essential for rich text entry. NSAttributedString was used heavily, this type allows for the definitions of metadata associated with ranges of text, and integrates with other Text rendering view layer systems; this is indispensable for enabling rich text formatting metadata like bold, italic, underline, font styles, hyperlinks, and others.

The adaptivity showcased in this prototype is progressive user interface simplification. As user stress rises, the application progressively hides UI elements, starting with the most visually distracting and least-used features. At maximum stress levels, all interface elements are hidden, leaving behind only text. On the very

highest stress levels, even the text content fades away as it gets more distant from the input handle. All of this aims to allow the user to hyper-focus on their current context and not get distracted by unnecessary clutter.

The reality remains that many of those user interface elements are there for a reason. They may still be needed on occasion, even at high stress levels. To this end, they are not truly gone: the user may move their mouse cursor to where the elements usually live (e.g. in the Ribbon controls, or in the status bar, or elsewhere), and the hidden user interface elements will briefly fade back into existence before disappearing again after the user is done interacting with them.

A physical metaphor inspired this design and interaction: the idea of a “fogged mirror.” As stress levels rise, interface elements gradually blur away, in what is supposed to be a graceful, non-obtrusive animation. But as the cursor approaches where the elements typically reside, the fog is “wiped away” and the elements become clear again. As time elapses since this “wipe,” the elements fade back away, returning the user to their high-focus, low-distraction user interface state. This inspiration based in physical metaphor aims to be intuitive, and balance the need for a distraction-free productivity environment with the need to occasionally use the features afforded by those complex, hidden controls.

The graceful animations were handled by the SwiftUI animation system. I will not include the full, lengthy code necessary for the responsive animation system, but I will include a few illustrative examples.

Below is an example of how the Ribbon view stress-level breakpoint and animations are implemented with SwiftUI. The entire Ribbon can blur in and out of visibility as referenced with the fogged mirror metaphor. When hidden, it has an invisible hitbox which is used to make it reappear on mouse hover. SwiftUI’s powerful view modifier method chaining system is used to set up complex animations. Details and state about the non-standard blur animation are encapsulated in a custom composable AnimationBlurModifier, while the SwiftUI animation system is capable of automatically animating other view environment state on change such as opacity. Some Ribbon elements have their own event handling, this information is passed into the Ribbon view as event handler functions.

```
ZStack(alignment: .top) {
            RibbonToolbar(
                // [...]
                focusLevel: focusLevel,
                // [...]
                onBulletListToggle: { toggleBulletList() },
                onNumberedListToggle: { toggleNumberedList() }
            )
            .modifier(AnimatableBlurModifier(blurRadius: focusLevel.showRibbonToolbar ? 0
: max(0, 20 * (1 - ribbonRevealAmount))))
            .opacity(focusLevel.showRibbonToolbar ? 1.0 : (0.15 + 0.85 *
ribbonRevealAmount))
            .allowsHitTesting(focusLevel.showRibbonToolbar || ribbonRevealAmount > 0.5)
            .onContinuousHover { phase in
                if !focusLevel.showRibbonToolbar {
                    handleRibbonHover(phase)
                }
            }

            // Invisible hover detection layer for triggering the reveal
            if !focusLevel.showRibbonToolbar {
                Color.clear
                    .contentShape(Rectangle())
                    .onContinuousHover { phase in
```

```
                handleRibbonHoverArea(phase)
            }
            .allowsHitTesting(true)
        }
    }
```

Figure 3 (above): Swift code, abridged, showing the implementation of stress-adaptive animation transitions and on-hover “mirror wipe” reveal behavior in the Ribbon toolbar view.

Some views such as the RichTextEditor have more complex animations that require custom animation event loops. An onChange() view modifier is applied to the main editor view, which fires when system stress level state changes. A Timer [7] is used to create an animation loop with an ease in/out interpolation to gradually transition text content to and from focus states.

```
RichTextEditor([...]) {

    }
    .onChange(of: focusLevel) { oldValue, newValue in
        // Animate paragraph fading in/out over 5 seconds
        let targetFade: CGFloat = newValue.fadeIntensity
        let startFade = paragraphFadeAmount
        let duration: TimeInterval = 5.0
        let steps = 60
        let interval = duration / Double(steps)

        var currentStep = 0
        Timer.scheduledTimer(withTimeInterval: interval, repeats: true) { timer in
            currentStep += 1
            let progress = Double(currentStep) / Double(steps)

            // Ease in-out interpolation
            let easedProgress = progress < 0.5
                ? 2 * progress * progress
                : 1 - pow(-2 * progress + 2, 2) / 2

            paragraphFadeAmount = startFade + (targetFade - startFade) * easedProgress

            if currentStep >= steps {
                paragraphFadeAmount = targetFade
                timer.invalidate()
            }
        }
    }
```

Figure 4 (above): Swift code (abridged) showing the implementation of a custom animation loop for the RichTextEditor animations.

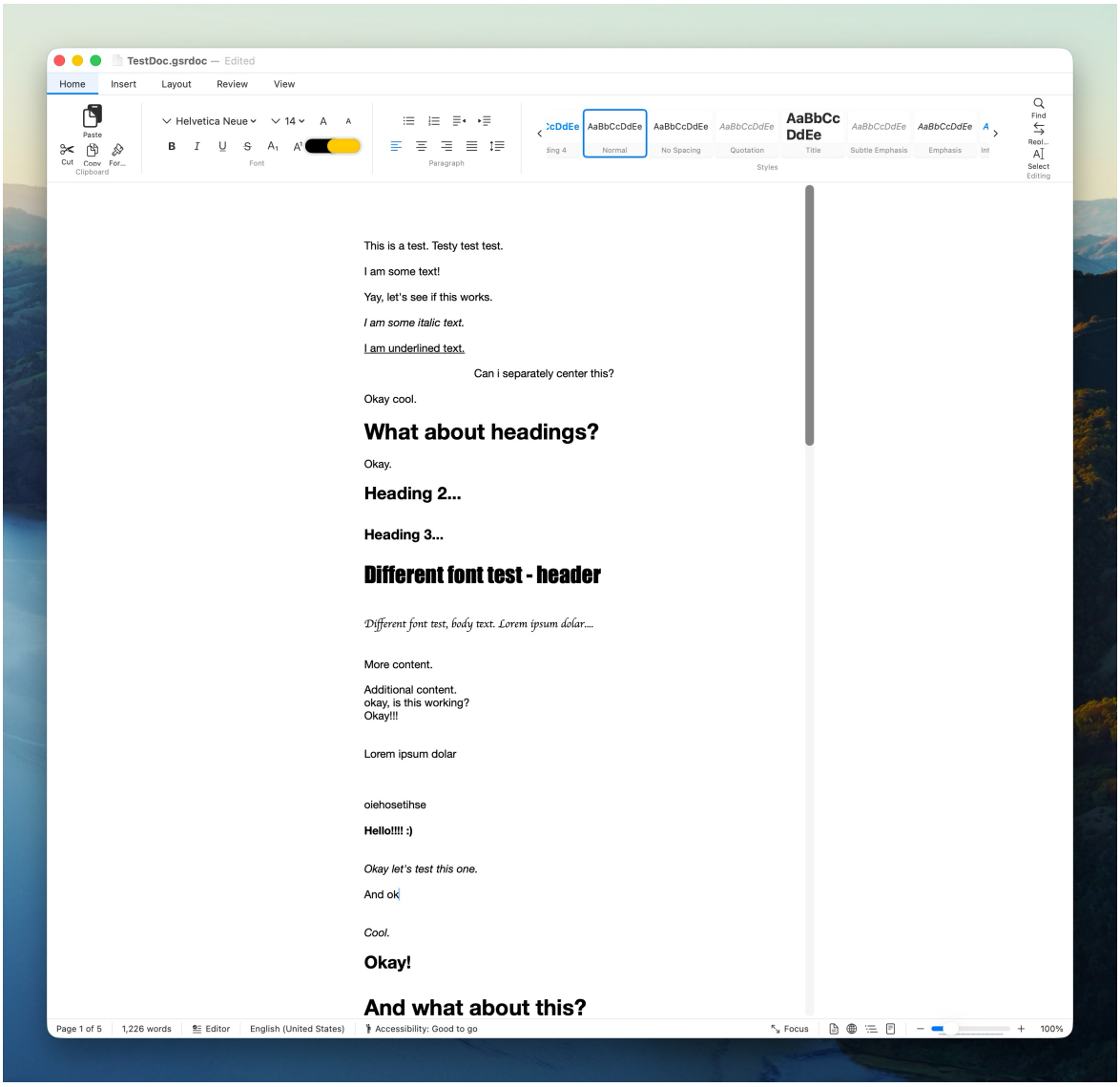

Figure 5 Word Prototype, Stress Level 1

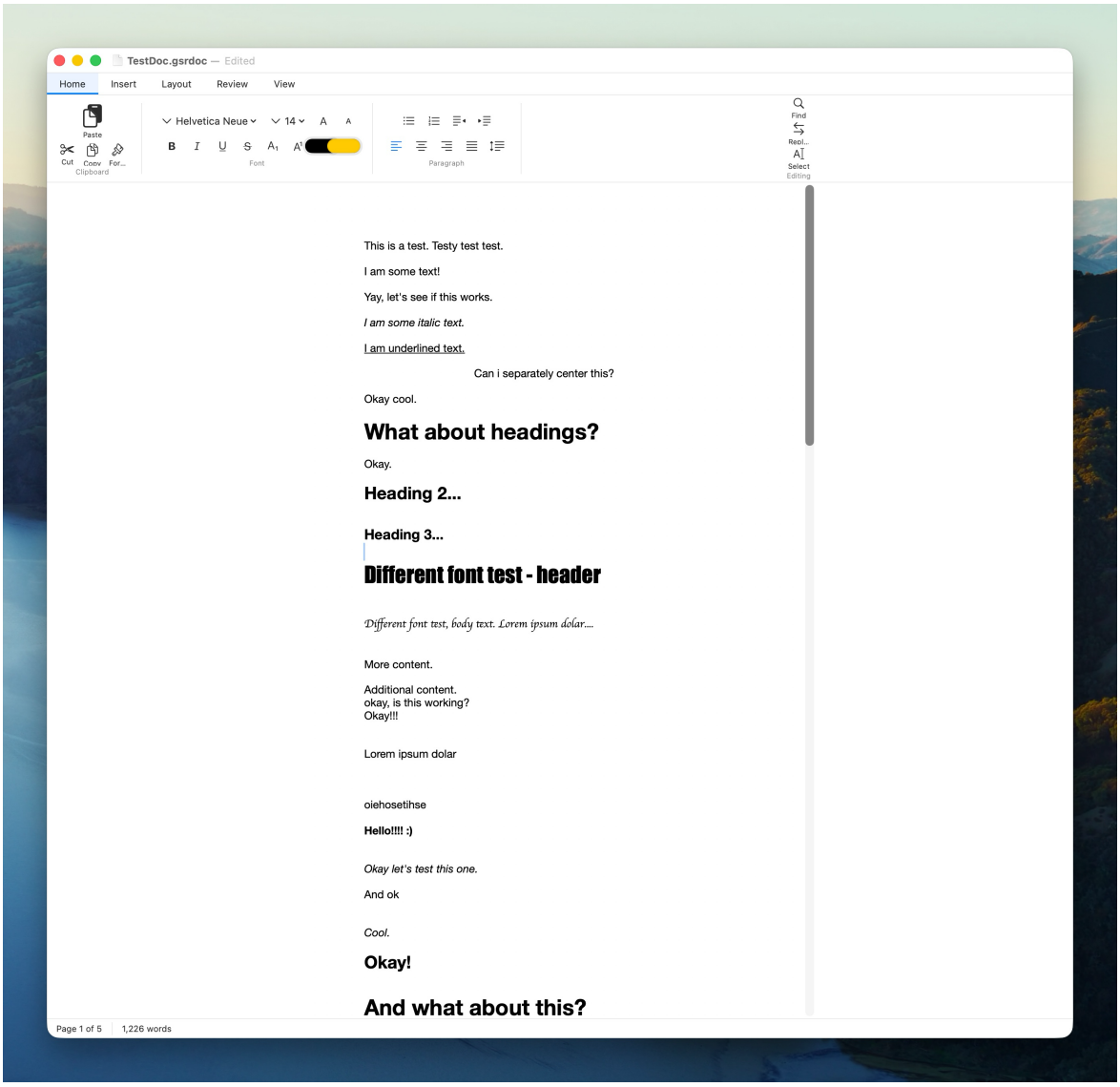

Figure 6 Word Prototype, Stress Level 3. Some Ribbon elements and Status Bar elements hidden.

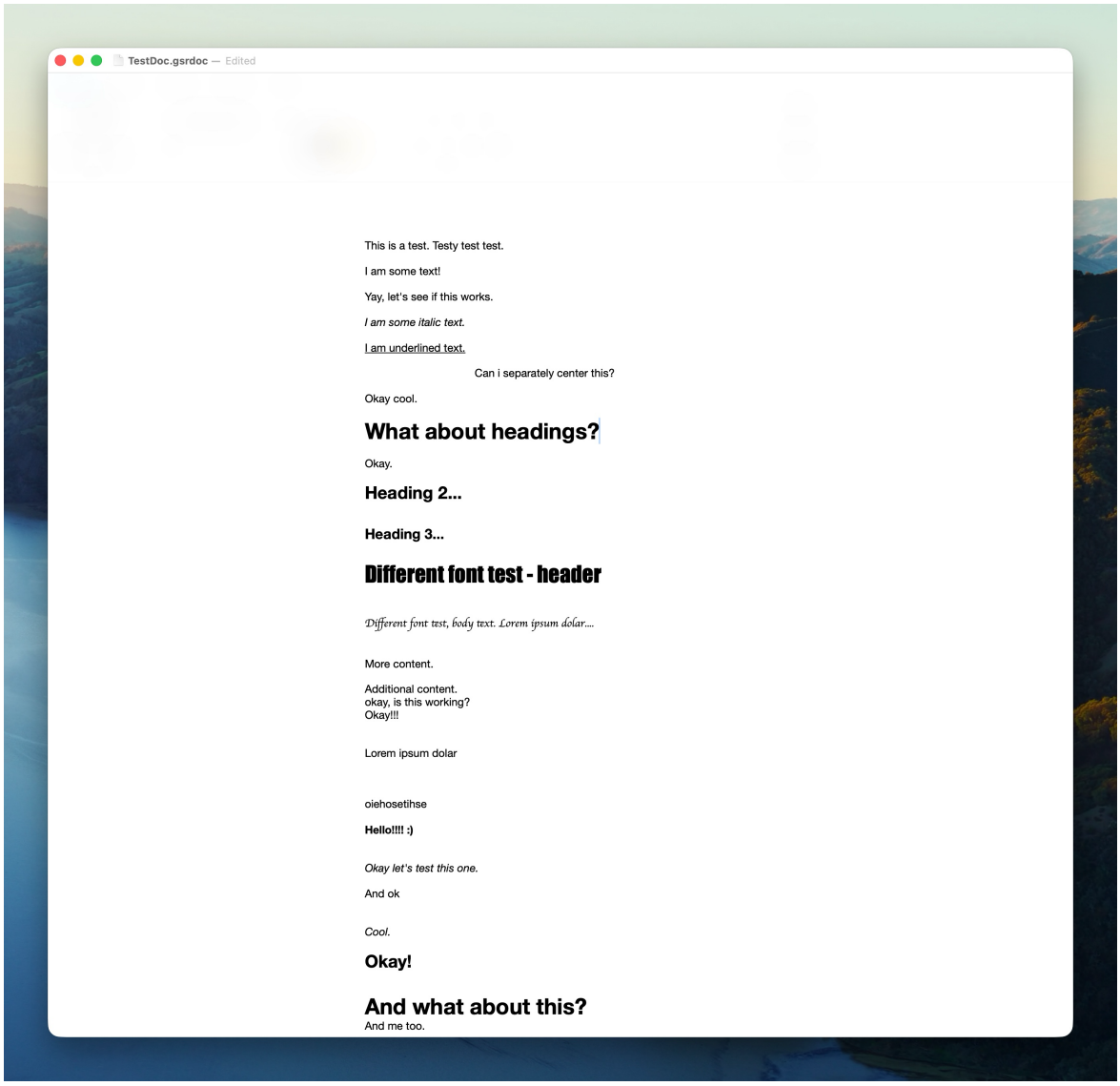

Figure 7 Word Prototype, Stress Level 4. All UI elements hidden.

Figure 8 Word Prototype, Stress Level 5. All UI elements hidden. Text fades away as it becomes more distant from the input handle.

3.4 Slack Prototype

The Slack prototype showcases a stress-adaptive context-aware notification triage kind of adaptivity. In casual interviews as I was determining prototype varieties, communication notifications were a frequently recurring theme; whether they were Slack, E-Mail, or some other platform. The problem was the same; large quantities of communications are distracting and cause a loss of productivity through context-switching, and they can exacerbate stress more in a positive feedback loop. Existing solutions such as Do Not Disturb mode are too binary, silencing everything. More complex approaches to this involving whitelists of people who are allowed through the Do Not Disturb filters suffer from allowing low-urgency messages from important people through, while still suppressing high-urgency messages from people not on the list.

The solution in this prototype is using an artificial intelligence system, a local large language model (LLM), to use a more nuanced approach to notification filtering. Using a combination of the LLM, a data structure that encapsulates relationships and social dynamics (hierarchy, friendships, group membership, etc.), and a custom "TriageEngine," the system assigns a "Triage Heuristic" to each incoming Slack message. This heuristic uses four pieces of information:

1. The relationship of the Sender to the User. (Supervisor, friend, group member, unaffiliated, etc)
2. The actionability of the message
3. The urgency of the message
4. In channels / groups, whether the message directly involves the User

This nuanced and context-sensitive approach allows for a far more useful triage of incoming messages than the simplistic binary mode or list-based approaches common today. Further, each messages preserves an explainable summary in natural human language of how the triage assessment was decided, promoting algorithmic transparency for the user.

As for the adaptivity behavior of the system; the triage scores assigned to each message are incorporated into an adaptive filtering system. As user cognitive load and stress levels rise, messages that are assigned lower urgency assessments no longer trigger distracting notifications. The urgency of each message is indicated in channel and DM threads with a color-coded signifier symbol, facilitating the rapid skimming of important information in a busy channel. At higher urgency levels, messages under a threshold of importance have are dimmed to a lower contrast ratio, making this skimming even easier (they revert to normal contrast on mouse hover for legibility if the user wishes to read them). At the highest level of stress, contiguous sequences of low and medium priority messages get fully collapsed in threads, replaced with a row that says "N Lower Priority Messages Hidden", (see Figure 12 Slack Prototype in High-Stress State). Additionally, the system includes a few interface simplification adaptations similar to the Word prototype. At higher stress levels, the sidebar navigation is simplified. Further, the lists of channels and DMs with pending unread notifications gets collapsed, progressively only showing higher-urgency threads. The lower-urgency threads may still be accessed through explicit user action, if necessary.

The AI behavior is powered using a local Ollama LLM installation running on port 11434, which is communicated with over HTTP with Ollama's included REST API. The inputs and responses are performed using structured JSON formatting, which is necessary for predictable and structured programmatic access to the AI system. Below is some of the Swift code involved in creating a REST OllamaClient, sending requests using JSON, and receiving the responses asynchronously using the Swift concurrency async feature.

```swift
struct OllamaClassification: Codable {
    let urgency: Int
    let actionable: Bool
    let involvesUser: Bool
    let explanation: String?
}

final class OllamaClient {
    static let shared = OllamaClient()
    private init() {}

    private let session: URLSession = {
        let config = URLSessionConfiguration.default
        config.timeoutIntervalForRequest = 2.0
        config.timeoutIntervalForResource = 3.0
        return URLSession(configuration: config)
    }()

    func classify(model: String? = nil, prompt: String) async throws ->
OllamaClassification? {
        let url = URL(string: "http://127.0.0.1:11434/api/generate")!
        var req = URLRequest(url: url)
        req.httpMethod = "POST"
        req.addValue("application/json", forHTTPHeaderField: "Content-Type")
        let body: [String: Any] = [
            "model": model ?? "llama3.1:latest",
            "prompt": prompt,
            "stream": false,
            "format": "json"
        ]
        req.httpBody = try JSONSerialization.data(withJSONObject: body, options: [])

        let (data, _) = try await session.data(for: req)
        guard let root = try JSONSerialization.jsonObject(with: data) as? [String: Any],
              let response = root["response"] as? String,
              let responseData = response.data(using: .utf8) else {
            return nil
        }
        let decoded = try JSONDecoder().decode(OllamaClassification.self, from:
responseData)
        return decoded
    }
}
```

Figure 9 (above): Swift code showing JSON / REST communication with the AI system

Below is a partial Swift excerpt of how the TriageEngine calls the Ollama client, and the internal API for how the data model can make a determination whether or not to surface a given message dynamically based on the current user stress levels using our shouldSurface() method.

```swift
struct TriageAssessment {
    var urgency: Int // 1–5
    var actionable: Bool
    var involvesUser: Bool // channels only; for DMs always true
    var explanation: String
}
```

```
final class TriageEngine {
    static let shared = TriageEngine()
    private init() {}

    // MARK: - Public API
    func assessDM(text: String, sender: Person, appState: AppState) async ->
TriageAssessment {
        let context = buildContext(for: sender)
        if appState.useAI, let ai = try? await askOllama(text: text, context: context,
isChannel: false) {
            return ai
        }
        return heuristicDM(text: text, sender: sender)
    }

    func assessChannel(text: String, sender: Person, channel: Channel, appState: AppState)
async -> TriageAssessment {
        let context = buildContext(for: sender) + " | channel_kind=\(channel.kind)"
        if appState.useAI, let ai = try? await askOllama(text: text, context: context,
isChannel: true) {
            return ai
        }
        return heuristicChannel(text: text, sender: sender, channel: channel)
    }

    func shouldSurface(_ triage: TriageAssessment, isDM: Bool, appState: AppState) -> Bool
{
        let lvl = appState.cognitiveLoadLevel
        // Level 1: Slack default — surface all
        if lvl <= 1 { return true }
        if isDM {
            switch lvl {
            case 2: return triage.actionable || triage.urgency >= 3
            case 3: return triage.actionable && triage.urgency >= 3
            case 4: return triage.actionable && triage.urgency >= 4
            default: return (triage.actionable && triage.urgency >= 5)
            }
        } else {
            // Channels: require involvesUser more as load increases
            switch lvl {
            case 2: return (triage.urgency >= 3) || (triage.actionable &&
triage.involvesUser)
            case 3: return triage.involvesUser && (triage.actionable || triage.urgency >=
4)
            case 4: return triage.involvesUser && triage.actionable && triage.urgency >= 4
            default: return triage.involvesUser && triage.actionable && triage.urgency >=
5
            }
        }
    }

    // MARK: - Private Helpers
    private func buildContext(for p: Person) -> String {
        let authorityTags: Set<String> = ["labSupervisor", "courseProfessor",
"taOfYourCourse"]
        let hasAuthority = p.role == "admin" || p.role == "faculty" ||
!authorityTags.isDisjoint(with: Set(p.relationshipTags))
        return "role=\(p.role); tags=\(p.relationshipTags.joined(separator: ","));
authority=\(hasAuthority)"
```

```
    }

    private func askOllama(text: String, context: String, isChannel: Bool) async throws ->
TriageAssessment? {
        let schema = "{" +
        "\"urgency\": number (1–5), \"actionable\": boolean, \"involvesUser\": boolean,
\"explanation\": string}" // guidance only
        let prompt = """
        You are a classifier. Analyze a message for triage.
        Message:
        """
        + text.replacingOccurrences(of: "\"", with: "'") +
        """
        Context:
        """ + context + """
        Task: Return strict JSON with keys: urgency (1–5), actionable (true/false),
involvesUser (true/false), explanation (why).
        If DM, involvesUser must be true.
        Examples:
        – Banter or greetings: urgency 1, actionable false, involvesUser false (channel)
or true (DM).
        – Direct request from professor: urgency 5, actionable true, involvesUser true.
        – Project teammate status update due soon: urgency 3–4, actionable maybe true,
involvesUser true if asking the user.
        The explanation should be concise, e.g. "Urgent because this is an actionable
message from a faculty supervisor." Output only JSON.
        """
        return try await OllamaClient.shared.classify(prompt: prompt)
            .map { TriageAssessment(urgency: clamp($0.urgency, 1, 5), actionable:
$0.actionable, involvesUser: $0.involvesUser, explanation: $0.explanation ??
"Automatically classified.") }
    }
```

Figure 10 (above): Swift code showing the structured JSON AI prompt and response schema for performing a triage heuristic on an incoming message's content, using the total system context available. The AI helps make assessments about urgency, actionability, and user involvement. The shouldSurface() method is a valuable internal API used to adaptively determine whether a message should be surfaced to the user, depending on their stress levels.

In addition to the triage and adaptivity, this prototype required the creation of a complex simulation of incoming DMs and Channel messages with which to bombard the user, this subsystem was named the SimulationEngine. To enable it, 100 sample system users were seeded, and they were randomly assigned into membership of various channels and populated with different relationships with the User. The prototype presume a university Slack setting like RIT, so the relationships include professor, supervisor, administrator, student, classmate, friend, and groupMember. Several channels were also seeded. A variety of dynamic message content template were created, with content placeholders that are randomly populated at runtime with example text to provide variety. Some of these messages are grouped into themes to make work-oriented channels, group project channels, and social channels have thematically distinct content. Some channels and DM threads are hard-coded to have more activity than others, recreating likely real-world settings.

An event loop and a central data structure were created, periodically generating new messages and assigning them to different channels and DM threads. New message creation is designed with a variable delay jitter and random lulls and burst periods, to emulate a real workload. As each message is created, it is send through the TriageEngine.

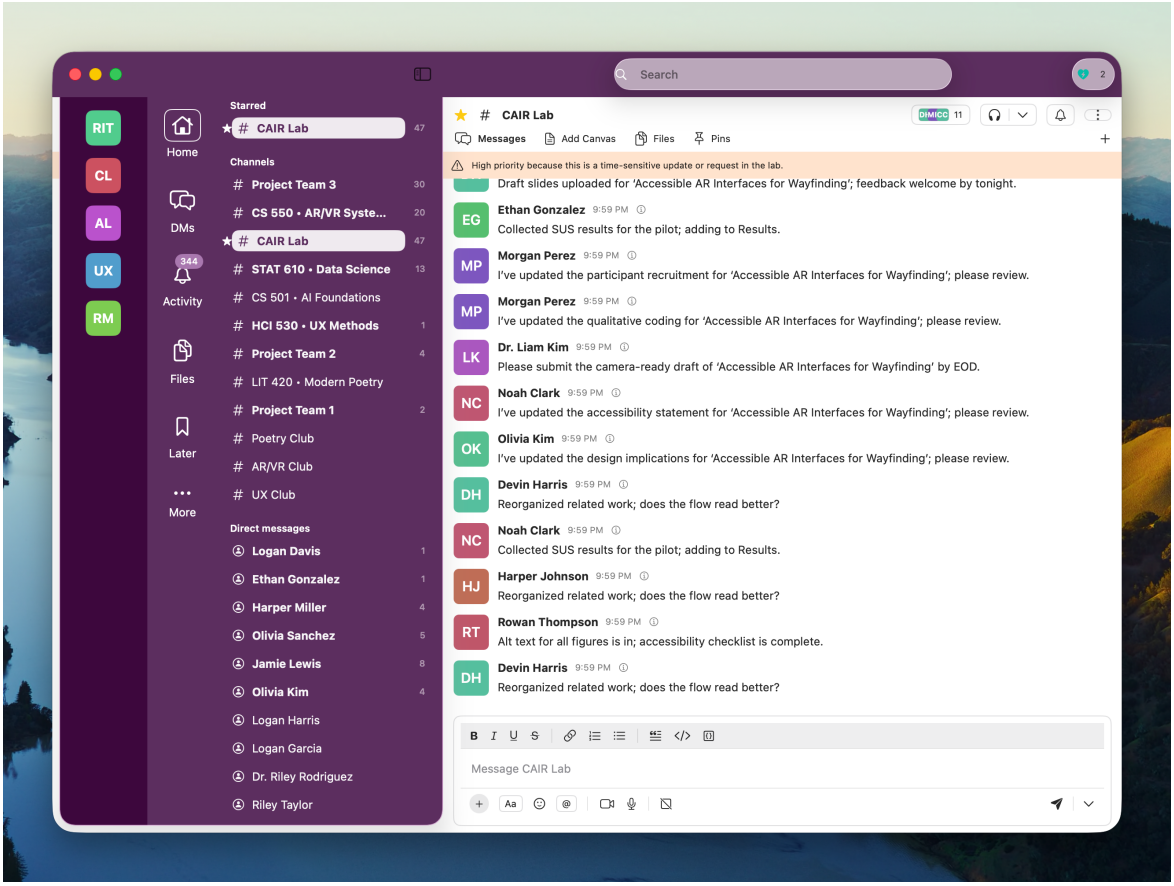

Figure 11 Slack Prototype in Low-Stress State

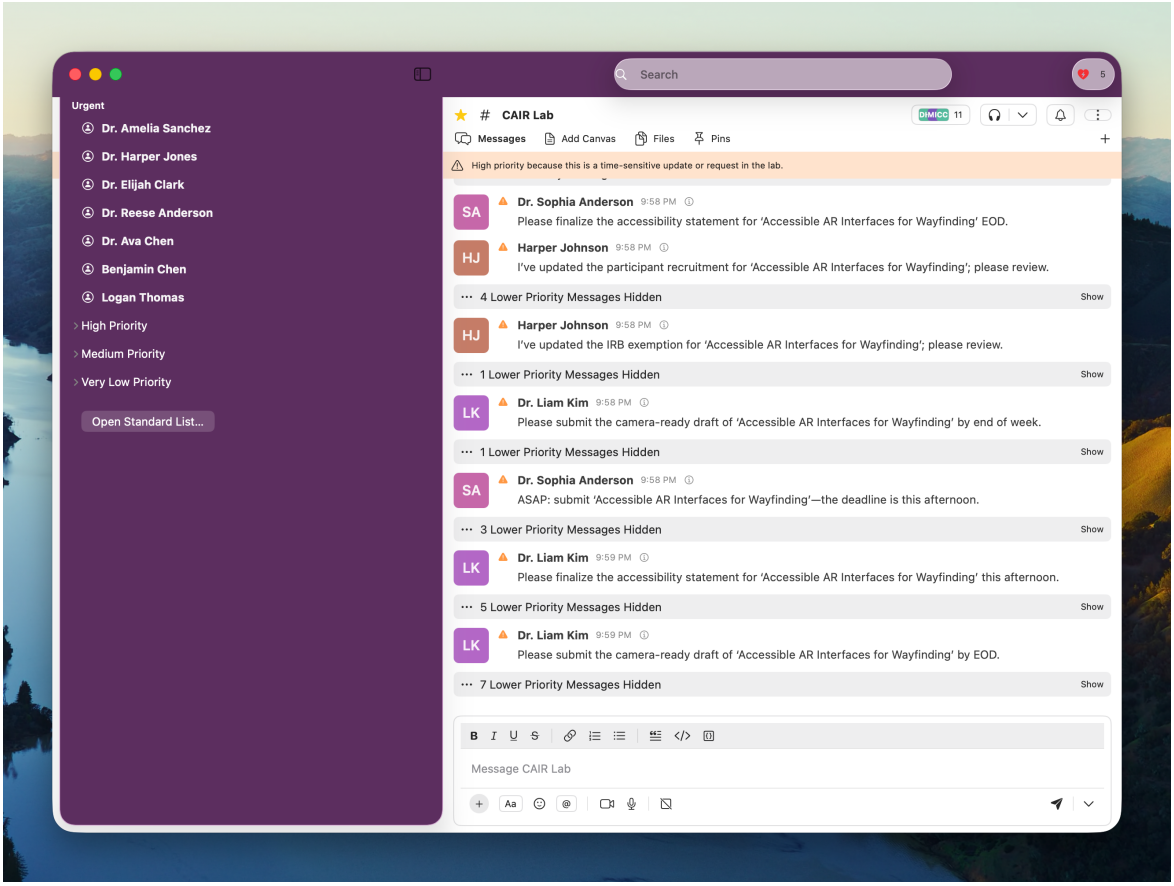

Figure 12 Slack Prototype in High-Stress State. Lower-priority messages below the dynamic threshold get collapsed into accordions. Message channels and threads in the sidebar are likewise categorized based on the urgency of their unread messages, and only urgent ones are displayed at the highest stress level.

4 CHALLENGES

4.1 Artificial Intelligence incorporation

The incorporation of artificial intelligence complicates the Slack prototype significantly. This would have proven to be a significant challenge for me 6 months ago, but since then I have started incorporating AI features into multiple projects: the "Affective Caption RPG" project at the RIT CAIR Lab, the AI/AR Vocational Coach and Assistant project at the RIT AIR Lab, and several of my personal projects. So even though this has a learning curve, it ended up being straightforward in this project due to recent experience. The most difficult aspect was simply experimenting with different prompts, structured JSON input/output formats, latency and performance balance, and experimenting with how effective my chosen model (Ollama) was at determining context, urgency, and actionability from conversation contents. With additional work, this system could become quite robust.

4.2 Security limitations

The modern security features and “sandboxing” of modern MacOS and iOS versions provided frequent obstacles for inter-device networking and I/O, as well as inter-process communication between the different apps.

For inter-process communication, the DistributedNotificationCenter API is a classic, simple method for sending small payloads at relatively low-frequency intervals between processes on MacOS. However, more recent MacOS versions prohibit almost all data payload types using this API for security reasons. [3] Ultimately, I found a simple workaround appropriate for my use case: instead of sending notifications with one name and then including the stress level in a payload, I could broadcast and subscribe to 5 different types of notifications. With only 5 classes, the performance impact is negligible, and this solution is far simpler than adopting one of the heavier, more secure modern IPC notification APIs.

Inter-device networking also proved challenging. The Apple Watch for example cannot directly communicate with the MacOS application. For this reason, I developed a simple paired iPhone proxy application which serves as a wireless bridge between the watch and the laptop (See Figure 16 iPhone Proxy App).

4.3 Networking

Wireless networking proved to be an issue. That was particular the case with the Apple Watch, which cannot communicate directly with MacOS, requiring an iPhone app intermediary. The Watch app communicates with the iPhones over Bluetooth using the WatchConnectivity framework. The iPhone communicates with MacOS over wireless LAN / Wi-Fi 802.11 / TCP. This system has proven one of the most brittle, as to avoid hard-coding IP addresses, it uses the Bonjour discovery protocol. As was seen in the class demo, Bonjour does not work over the RIT Wi-Fi. In the future, I would like to explore alternate networking solutions, such as using an ad-hoc WiFi connection directly between iOS and MacOS, creating an online API intermediary which would avoid the need for LAN IP address discovery, or using Bluetooth rather than Wi-Fi.

4.4 Human variability and demographics

The initial implementation of the prototypes used a “one-size-fits-all” approach mapping biometric inputs to stress levels. This approach is naïve, even for a exploratory prototype. Human baseline heartrate bpm and stress levels vary markedly, to the extent that a system calibrated to work with one person may not work at all for another person. While these prototypes did not require a truly high-fidelity approach, I wanted it to at least be able to provide a usable, approximate classification for different users.

To overcome this problem, I first implemented a set of age-based heart rate zones sourced from literature that approximately map to differently levels of cognitive load and stress. This was done using the Swift programming languages’ powerful Enum feature and various helper functions. I added controls to the central Biometric Relay app settings for users to customize their age, and the system uses appropriate heartrate zones based on the input.

Some users have heartrate baselines that trend above or below the average for their demographic age group; personally, I have a heart rate baseline that trends below average, so a system calibrated for me would not work effectively for the general population. Skewing low, it would detect elevated stress levels too quickly for other users. To handle this problem, I added another UI element where the user can optionally configure their baseline heartrate. Then, I implemented a Karvonen formula that morphs each of the stress zone ranges

in proportion to the user-defined baseline. For transparency, the full set of calculated heartrate zone ranges that have been calculated for the user are displayed.

```
public enum StressHRCategory: String, CaseIterable {
    case baseline
    case low
    case medium
    case high
    case veryHigh
}

public struct StressHRZones {
    public let ageRange: ClosedRange<Int>
    public let baseline: ClosedRange<Int>
    public let low: ClosedRange<Int>
    public let medium: ClosedRange<Int>
    public let high: ClosedRange<Int>
    public let veryHigh: ClosedRange<Int>

    public func range(for category: StressHRCategory) -> ClosedRange<Int> {
        switch category {
        case .baseline: return baseline
        case .low: return low
        case .medium: return medium
        case .high: return high
        case .veryHigh: return veryHigh
        }
    }
}

// Base zones by age cohorts
public let hrStressZones: [StressHRZones] = [
    // 18–25
    StressHRZones(
        ageRange: 18...25,
        baseline: 0...90,
        low: 90...100,
        medium: 100...110,
        high: 110...130,
        veryHigh: 130...300
    ),
    // 26–35
    StressHRZones(
        ageRange: 26...35,
        baseline: 0...90,
        low: 90...100,
        medium: 100...110,
        high: 110...125,
        veryHigh: 125...300
    ),
    [Truncated]
]
```

Figure 13 (above): Truncated Swift code example showing heartrate stress zones grouped by age range

```
/// Classifies a BPM into a category for given zones.
```

```swift
public func category(for bpm: Int, zones: StressHRZones) -> StressHRCategory {
    for cat in StressHRCategory.allCases {
        if zones.range(for: cat).contains(bpm) { return cat }
    }
    // If outside declared ranges, place below or above
    if bpm < zones.baseline.lowerBound { return .baseline }
    return .veryHigh
}
```

Figure 14 (above): Swift code showing helper function discretizing an integer heartrate BPM value into the defined stress categories

```swift
/// Computes target HR using Karvonen formula for a given intensity [0,1].
public func karvonenTargetHR(intensity: Double, maxHR: Int, restingHR: Int) -> Int {
    let reserve = max(0, maxHR - restingHR)
    return restingHR + Int((intensity * Double(reserve)).rounded())
}

/// Returns base zone group for a given age, clamped to nearest group if out of range.
public func baseZones(for age: Int) -> StressHRZones {
    if let exact = hrStressZones.first(where: { $0.ageRange.contains(age) }) { return
exact }
    if let first = hrStressZones.first, age < first.ageRange.lowerBound { return first }
    if let last = hrStressZones.last, age > last.ageRange.upperBound { return last }
    return hrStressZones[0]
}

/// Karvonen-adjusted stress zones for a specific user.
public func adjustedZones(for age: Int, restingHR: Int) -> StressHRZones {
    let base = baseZones(for: age)
    let maxHR = 220 - age

    // Intensity fractions (approximate cognitive load bands)
    let baselineIntensities = (0.00, 0.25)
    let lowIntensities      = (0.25, 0.40)
    let mediumIntensities   = (0.40, 0.55)
    let highIntensities     = (0.55, 0.70)
    let veryHighIntensities = (0.70, 1.00)

    func range(_ pair: (Double, Double)) -> ClosedRange<Int> {
        let lower = karvonenTargetHR(intensity: pair.0, maxHR: maxHR, restingHR:
restingHR)
        let upper = karvonenTargetHR(intensity: pair.1, maxHR: maxHR, restingHR:
restingHR)
        return min(lower, upper)...max(lower, upper)
    }

    return StressHRZones(
        ageRange: base.ageRange,
        baseline: range(baselineIntensities),
        low:      range(lowIntensities),
        medium:   range(mediumIntensities),
        high:     range(highIntensities),
        veryHigh: range(veryHighIntensities)
    )
}
```

Figure 15 (above): Swift code showing the implementation of a Karvonen formula, which is used to transform the heartrate zones for an individual user if their baseline heartrate trends above or below the average for their demographic age.

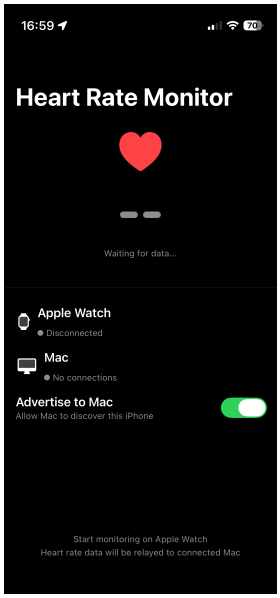

Figure 16 iPhone Proxy App. Simple, rarely if ever requires user interaction. Displays wireless connectivity state between the watch, phone, and MacOS relay system.

5 LIMITATIONS AND FUTURE WORK

This set of prototypes use relatively simple signal processing and blending to calculate a metric for user stress and cognitive load. The system would benefit from a more complex and well-tested technique, such as a empirically validated machine learning backed solution.

The prototype applications did not undergo any sort of usability testing or evaluation process. A compelling subsequent study could include an empirical evaluation comparing user stress levels, error rates, productivity, and affect while performing complex workloads using the actual applications and their stress-adaptive prototype alternatives. It would be fascinating to see if these prototypes' methods could significantly improve any of these metrics in a direct matchup. A longitudinal study with users doing their actual real-world workflows over time rather than prescribed tasks would be particularly fascinating. A longitudinal study would also have the benefit of allowing users to become accustomed to behaviors like morphing, adaptive user interfaces or dynamically collapsed or disclosed content. These behaviors may prove to be distracting or alarming initially, but become normal over time.

6 CONCLUSION

This study explored a novel method of powering stress-adaptive productivity application user interfaces using biometric inputs. A variety of methods were explored, including adaptive interface simplification and focus modes, and context-aware dynamic notification triage facilitated through artificial intelligence. Although proof-of-concept prototypes, the prototype systems were implemented using production-grade programming techniques and frameworks, and already offer promising and usable stress-adaptive behaviors. Future research in this domain exploring adaptivity methods, improving the biometric signal processing and classification, and empirically validating the benefits of these sorts of protypes offers promising opportunities in futuristic productivity workflows that can improve quality of life, stress levels, and lower error rates in daily workflows.

7 REFERENCES

[1] Andrew Madsen. 2025. armadsen/ORSSerialPort. Retrieved December 9, 2025 from https://github.com/armadsen/ORSSerialPort

[2] iA Writer: The Benchmark of Markdown Writing Apps. Retrieved October 21, 2025 from https://ia.net/writer

[3] DistributedNotificationCenter. *Apple Developer Documentation*. Retrieved December 9, 2025 from https://developer.apple.com/documentation/foundation/distributednotificationcenter

[4] Swift Package Manager | Documentation. Retrieved December 9, 2025 from https://docs.swift.org/swiftpm/documentation/packagemanagerdocs/

[5] DocumentGroup. *Apple Developer Documentation.* Retrieved December 9, 2025 from https://developer.apple.com/documentation/swiftui/documentgroup

[6] NSTextView. *Apple Developer Documentation*. Retrieved October 21, 2025 from https://developer.apple.com/documentation/appkit/nstextview

[7] Timer. *Apple Developer Documentation*. Retrieved December 9, 2025 from https://developer.apple.com/documentation/foundation/timer