1. As-Built Technical Documentation

**Project:** Typing Test Assistant  
**Final Version Date:** July 20, 2025  
**Lead Engineer:** Gary Mix  
**Client:** LackOfSkillz

1. Project Overview

The Typing Test Assistant is a Python-based desktop utility for Windows designed to automate typing tests. The application allows a user to select a region of their screen using an overlay. It then performs Optical Character Recognition (OCR) on that region to extract the text. After an initial delay, it simulates human-like typing of the extracted text into the user's active window. The core feature is its ability to automatically monitor the selected screen region for new text (e.g., when a typing test scrolls) and continue typing the new content seamlessly. The application includes a GUI for configuring performance parameters (WPM, accuracy) and a separate interactive GUI for pausing and resuming the typing process.

2. Core Technologies & Dependencies

* **Language:** Python 3.9+
* **GUI Framework:** customtkinter for a modern Windows aesthetic.
* **Hotkeys:** pynput is used exclusively for its robust GlobalHotKeys listener.
* **Screen Capture:** Pillow (PIL Fork) is used for ImageGrab.
* **OCR Engine:** pytesseract library acting as a wrapper for Google's Tesseract OCR engine.
  + **External Dependency:** Tesseract OCR must be installed on the host system and accessible via the system's PATH.
* **Automation/Typing Engine:** pyautogui is used for all keyboard control (typing characters, pressing space/backspace). Its FAILSAFE feature has been explicitly disabled to allow for intended operation.
* **Window Management:** pywin32 is used for managing window focus, specifically for restoring focus to the target application after resuming from a pause.

3. Application Architecture

The application operates on a multi-threaded model to ensure a responsive user interface while background tasks are running.

* **Main Thread:** Dedicated exclusively to running the tkinter main event loop (root.mainloop()). All GUI creation, updates, and interactions are scheduled to run on this thread using root.after() to prevent instability and crashes.
* **Hotkey Thread:** A single, persistent background thread runs the pynput.GlobalHotKeys listener. When a hotkey is pressed, its callback function does not perform any logic directly; it schedules the corresponding handler function (e.g., on\_activation\_hotkey) to run on the main thread.
* **Worker Threads:** All long-running tasks, such as the OCR process and the main typing loop, are spawned as new, temporary threading.Thread instances. This prevents the application from freezing during OCR scans or typing.

4. Core Logic & Key Design Decisions

* **Workflow:** The application uses a simple state machine with two primary states: IDLE and TYPING. A dedicated hotkey (Ctrl+Alt+[) is used for the pause/resume functionality to decouple it from the main typing thread and prevent the race conditions that plagued earlier designs. The workflow is Hotkey -> Select Area -> 8s Delay -> Type -> Finish.
* **Typing Engine (start\_typing):**
  + **Engine Choice:** After encountering system-level blocking with ctypes and pynput.Controller, pyautogui was chosen as the definitive, most robust library for sending keystrokes.
  + **Fail-Safe:** The pyautogui.FAILSAFE = False command at the start of the script was a critical fix to prevent the application from crashing when the user's mouse moved to a screen corner during the 8-second delay.
  + **Speed Calibration:** The theoretical WPM calculation was found to be inaccurate due to system overhead. A CALIBRATION\_FACTOR was introduced, derived from real-world testing (Target 50 WPM / Actual 34 WPM), to ensure the output speed matches the user's setting.
  + **Accuracy Simulation:** The final implementation uses a "typo-and-correct" mechanism (write -> backspace -> write) as it provides the clearest visual feedback that the accuracy setting is functional.
* **OCR Processing (\_perform\_ocr):**
  + To improve accuracy, two pre-processing steps were implemented before sending an image to Tesseract:
    1. **Grayscale Conversion:** image.convert('L') removes color noise.
    2. **Page Segmentation Mode:** --psm 6 was added to the Tesseract config to instruct it to treat the selected area as a single, uniform block of text.
  + **Newline Handling:** A critical fix for typing tests was to replace all newline characters (\n) with spaces () immediately after OCR to prevent pyautogui from incorrectly pressing the 'Enter' key.
* **Interactive Control (Pause/Resume):**
  + The decision to use a dedicated hotkey (Ctrl+Alt+[) to summon the LiveControlGUI was made to resolve a critical race condition where the application would hang if it tried to create a new window and send keystrokes simultaneously.
  + **Focus Management:** The use of win32gui.GetForegroundWindow() before pausing and win32gui.SetForegroundWindow() after resuming is essential to provide a seamless user experience, allowing the user to interact with the control GUI and have the application automatically return focus to their typing test.