report - bubble_v2

Table of Contents

---

1. Project Overview

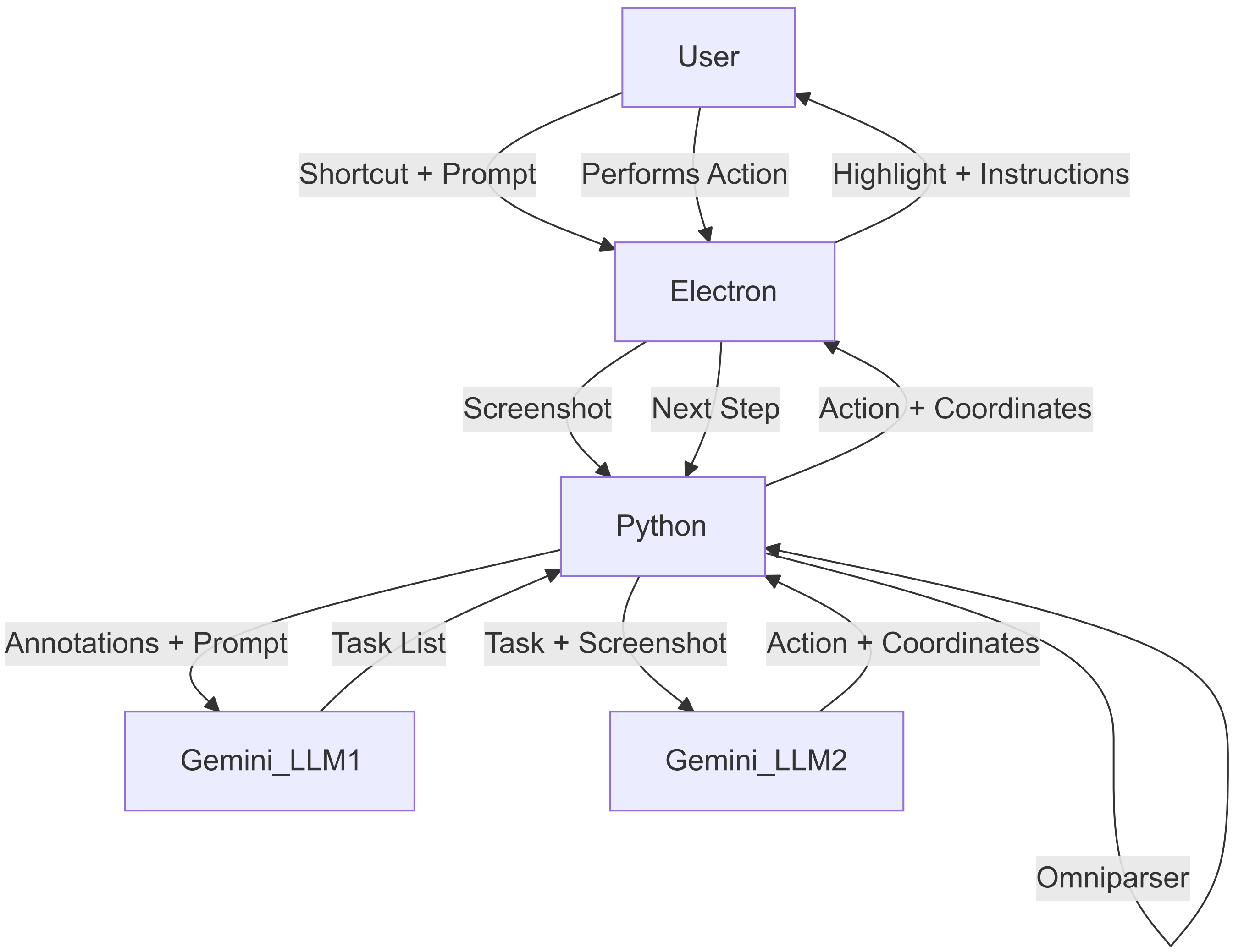
This application is a cross-platform (Mac/Windows) desktop assistant that overlays on top of any application. It uses AI to guide users through complex tasks (e.g., “Export to PDF in Word”) by visually highlighting UI elements and providing step-by-step instructions, leveraging on-screen intelligence.

Key Features:

- Invoked by a global keyboard shortcut.
- Accepts natural language prompts.
- Understands the current screen using computer vision (Omniparser).
- Plans and guides the user through atomic UI actions using LLMs (Gemini).
- Highlights UI elements and tracks user progress.

---

2. High-Level Architecture



3. Pipeline: Step-by-Step Flow

1. **User invokes the overlay** (e.g., with Cmd+Shift+Space).
2. **User types a prompt** (e.g., “Export to PDF in Word”).
3. **App captures a screenshot** of the current screen.
4. **Screenshot is sent to Omniparser** (Python) to detect and annotate UI elements.
5. **Annotated screenshot and prompt are sent to LLM #1 (Gemini)** to generate a list of atomic tasks (e.g., “Go to File menu”, “Click Export”).
6. **For each task:**
   - The app captures a new screenshot.
   - The current task and screenshot are sent to LLM #2 (Gemini) to get a description and the coordinates of the UI element to highlight.
   - The frontend highlights the relevant UI element and displays the instruction.
   - The app waits for the user to perform the action, then proceeds to the next step.
7. **Repeat** until all tasks are completed.

4. Integrations

A. Electron (Frontend Overlay)

- Provides the user interface overlay.
- Handles global keyboard shortcuts.
- Captures screenshots using Electron's desktopCapturer API.
- Renders the prompt input, step-by-step instructions, and visual highlights.
- Communicates with the backend (either via HTTP or direct script calls).

B. Python (Backend)

- Handles all AI and computer vision logic.
- Integrates Omniparser for UI element detection.
- Integrates Gemini LLM APIs for task planning and action inference.
- Exposes endpoints (in server mode) or scripts (in direct mode) for Electron to call.

C. Omniparser

- A computer vision tool that analyzes screenshots to detect and label UI elements (buttons, menus, etc.).
- Returns a list of detected elements and their coordinates.

D. Gemini (LLM APIs)

- LLM #1: Receives the user prompt and UI annotations, returns a list of atomic tasks.
- LLM #2: Receives each task and a screenshot, returns a human-readable action description and the coordinates of the UI element to highlight.

---

5. Implementation Approaches

A. Backend Server (FastAPI) Approach

How it works:

- The Python backend runs as a FastAPI server.
- Electron communicates with the backend via HTTP requests to endpoints:
  - `/annotate` (POST): Receives a screenshot, returns UI annotations.
  - `/plan` (POST): Receives a prompt and annotations, returns a task list.
  - `/act` (POST): Receives a task and screenshot, returns action description and coordinates.
- CORS is enabled for cross-origin requests.

- This approach is familiar to web developers and easy to debug with tools like Postman.

Pros:

- Clear separation of frontend and backend.
- Easy to test endpoints independently.
- Scalable for future networked/multi-user scenarios.

Cons:

- Requires running a server process.
- Subject to Content Security Policy (CSP) issues in Electron.
- Port conflicts and firewall issues possible.

B. Direct Python Script Invocation (No Server) Approach

How it works:

- The Python backend is split into standalone scripts (`annotate.py` , `plan.py` , `act.py`).
- Electron calls these scripts directly using Node.js's `child_process.spawn` , passing data via stdin and reading JSON output from stdout.
- No HTTP, no server, no ports, no CSP issues.

Pros:

- No server process to manage.
- No CSP or port issues.
- Simpler for packaging and distribution.

Cons:

- Each call starts a new Python process (slightly slower for heavy workloads).
- Harder to scale to networked/multi-user scenarios.

---

6. File-by-File Summary

electron-app/forge.config.js

- Main configuration for Electron Forge and Webpack.
- Defines how the app is built, packaged, and which files are entry points.
- Now points the preload script to the raw file, not bundled by Webpack.

electron-app/src/main.js

- Electron’s main process script.
- Sets up the main window, global shortcuts, and IPC handlers.
- Now loads the raw `preload.js` file for the BrowserWindow.

electron-app/preload.js

- The preload script, running in a secure Node.js context.
- Exposes APIs to the renderer via `contextBridge`.
- Handles screenshot capture and direct invocation of Python scripts using `child_process`.
- Only exposes safe, controlled APIs to the renderer.

electron-app/src/renderer.js

- The renderer process script (the UI logic).
- Handles user input, task queue, and UI updates.
- Calls the APIs exposed by the preload script to interact with Python scripts.

electron-app/python-backend/app.py

- The original FastAPI backend (used in server mode).
- Defines `/annotate`, `/plan`, `/act` endpoints.
- Integrates Omniparser and (mock) Gemini logic.

electron-app/python-backend/annotate.py

- Standalone script for UI annotation.
- Reads image bytes from stdin, outputs JSON with UI elements and coordinates.

electron-app/python-backend/plan.py

- Standalone script for task planning.
- Reads JSON with prompt and annotations from stdin, outputs a list of atomic tasks.

electron-app/python-backend/act.py

- Standalone script for action inference.
- Reads JSON with the current task from stdin, outputs action description and coordinates.

electron-app/python-backend/requirements.txt

- Lists Python dependencies (e.g., FastAPI, Omniparser, Pillow).

electron-app/src/index.html

- The main HTML file for the overlay UI.
- Contains the prompt input and containers for step-by-step instructions.

electron-app/src/index.css

- Styles for the overlay UI, including highlight boxes and prompt input.

electron-app/webpack.*.config.js

- Webpack configuration files for main, renderer, and (if used) preload scripts.
- Now only bundle main and renderer, not preload.

---

7. Security, Packaging, and Permissions

- **Security:** The preload script uses `contextBridge` to expose only safe APIs to the renderer, minimizing risk.
- **Packaging:** The app can be packaged for Mac and Windows using Electron Forge, including all Python dependencies.
- **Permissions:** On Mac, the app may require screen recording and accessibility permissions to capture screenshots and track user actions.

---

8. Extensibility and Future Improvements

- **Real LLM Integration:** Swap out mock logic for real Gemini API calls.
- **User Action Tracking:** Implement logic to detect when the user performs the highlighted action.
- **Error Handling:** Add robust error handling and user feedback.
- **Logging:** Add logging for debugging and analytics.
- **Extensibility:** The modular design allows for easy addition of new AI models, UI features, or integrations.

---

9. Glossary

- **Electron:** A framework for building cross-platform desktop apps with web technologies.
- **Preload Script:** A special script in Electron that runs in a secure context, bridging Node.js and the browser.

- **Renderer Process:** The part of Electron that displays the UI (like a web page).
- **Main Process:** The part of Electron that controls the app lifecycle and windows.
- **Omniparser:** A tool for detecting and labeling UI elements in screenshots.
- **LLM (Large Language Model):** An AI model (like Gemini) that can understand and generate human language.
- **CSP (Content Security Policy):** A browser security feature that can block certain types of resource loading.
- **IPC (Inter-Process Communication):** Mechanism for communication between Electron's main, preload, and renderer processes.

---

10. Summary Table: Approaches

Approach	How Electron Talks to Python	Pros	Cons
Backend Server	HTTP (FastAPI)	Familiar, scalable	CSP, ports, server needed
Direct Scripts	child_process (stdin/stdout)	No CSP, no server, simple	New process per call, local

---

11. Conclusion

This project demonstrates a robust, modern approach to building a cross-platform, AI-powered desktop assistant.
It leverages the best of Electron for UI, Python for AI, and modern LLMs for intelligent task planning and guidance.
The architecture is modular, secure, and extensible—ready for both rapid prototyping and future production hardening.
