Implementation Guide — Detailed Technical Specifications & Practical Workflow

1. System Architecture (practical implementation)

1.1 Operational modes (decide up front)

You must choose (or support both) two operational modes — each has different implementation requirements and limitations:

A. Client-only / Non-intrusive Mode (quick & safe)

- No DOM access of cross-origin sites.
- Visual effects are applied to the rendered iframe element or to an overlay: blur, grayscale, brightness/contrast, zoom, cursor overlay.
- Good for instant previews and fast UX feedback.
- Limitations: cannot modify inner DOM, run screen-reader simulation based on page text, or run axe-core inside iframe for cross-origin pages.

B. Instrumented / Full mode (recommended for full functionality)

- Backend proxy or headless browser (Puppeteer/Playwright) fetches the page server-side, injects scripts/CSS, runs axe-core, and serves an instrumented, same-origin version to the frontend.
- Enables full simulations: DOM modifications, keyboard/mouse interception inside the page, screen-reader walkthroughs, axe results.
- Trade-offs: extra infra, legal/robots/terms risks, higher resource use, latency.

Practical approach: implement Client-only mode first for low complexity; add Instrumented mode later for full features.

1.2 High-level modules & responsibilities

- **Frontend UI / Controller**
  - Render target (iframe or instrumented document).
  - Control panel (toggles, sliders, presets, severity).

    - Overlay layer for cursor, notifications, and safe filters.
    - Live feedback panel and charts.
- **Simulation Engine (frontend + backend split)**
    - Visual Module (filters, zoom, overlays).
    - Motor Module (input delay, jitter, restricted input).
    - Cognitive Module (delays, distractions, memory overload).
    - Screen-reader Module (TTS, focus walking) — requires instrumented mode for accurate reading.
- **Analyzer & Reporting**
    - axe-core runner and result normalizer.
    - Report formatter (JSON → PDF/CSV).
    - Screenshot/capture service.
- **Backend Worker Pool (if instrumented)**
    - Headless browser instances for rendering/injection/axe runs.
    - Job queue (Redis/worker) with rate limiting.
    - Storage for reports/screenshots (object storage e.g., S3).
- **Security & Governance**
    - Proxy sanitization, CSP, rate limits, user consent, logging & deletion policy.

---

2. Input Layer — Practical implementation guidelines

2.1 UI controls & UX

- Provide:
    - URL input with validation and optional "sample test sites" list.
    - Predefined impairment presets (e.g., "Colorblind — Moderate", "Motor — Severe") plus full granular controls.
    - Severity sliders with numeric readouts and unit labels.
    - Clear label: "Interactive mode available only for sites that permit embedding or when using Instrumented mode."

2.2 Embedding strategy & fallback

- Try to embed target in sandboxed iframe first.
    - Use sandbox attributes to maximize security.
    - If the iframe is blocked by `X-Frame-Options` or CSP, automatically switch to fallback modes:
        - **Snapshot (visual-only)**: capture a server-side screenshot and apply visual filters client-side (no interactivity).

    - **Instrumented**: proxy and serve an instrumented copy (if you support that and accept legal/ethical considerations).

2.3 Consent & Terms check

- Before proxying or storing any third-party page:
    - Check `robots.txt` and terms of service for scraping/rehosting.
    - Present explicit checkbox consent to the user for proxying a site and for storing screenshots/reports.

---

3. Simulation Layer — implementation details & safe parameter ranges

Below are practical strategies for applying each simulation, and exactly when/where they can be applied (client vs instrumented), plus recommended default ranges.

3.1 Visual Impairments

3.1.1 Blur / Low vision

- **Client mode:** apply CSS filter `blur(px)` to the iframe element (affects entire rendered frame).
- **Instrumented mode:** inject CSS into page for more nuanced control (e.g., blur only main content).
- **Severity mapping (recommend defaults):**
    - Mild: 2px blur
    - Moderate: 6px blur
    - Severe: 12px blur
- **Notes:** excessive blur can make testing impossible — warn the tester and provide a keyboard shortcut to toggle off quickly.

3.1.2 Contrast / Brightness

- **Implementation:** apply CSS filter adjustments on the iframe or inject via instrumented CSS.
- **Severity mapping (contrast multiplier relative to original):**
    - Mild: 0.9
    - Moderate: 0.7
    - Severe: 0.5
- **Practical tip:** display a “contrast preview swatch” to let developers compare original vs simulated.

3.1.3 Grayscale / Monochrome

- **Implementation:** CSS `grayscale(100%)` on iframe or injected on page.
- **Use case:** quickly test reliance on color to communicate meaning.

3.1.4 Color-blind simulations

- **Client approach (limited):**
  - Use an SVG color matrix filter layered over the iframe; works as a visual transform but doesn't let you test semantic color dependence (labels still inaccessible to machine analysis).
- **Instrumented approach (robust):**
  - Inject a script that computes color transformation on critical UI elements (buttons, charts) or run server-side rendering + color transform.
- **Types to support:** protanopia, deuteranopia, tritanopia (and combined severity).
- **Practical note:** color matrix transforms are approximations — document limitations to users.

3.1.5 Zoom / Magnification

- **Implementation:** CSS transform scale on the iframe or inject larger root font-size/zoom on instrumented page.
- **Severity mapping:** 1.25× (mild), 1.5× (moderate), 2.0× (severe).
- **Accessibility note:** ensure UI controls remain reachable — overlay controls should be separate from the zoomed iframe.

3.1.6 Screen-reader simulation (visualized)

- **Client only:** limited TTS options — you can read the URL and metadata, but not inner content of cross-origin iframe.
- **Instrumented:** walk DOM, gather ARIA/alt text/semantic structure, feed to Web Speech API with focus highlighting.
- **Reading speed mapping (wpm):** Mild 170 wpm, Moderate 120 wpm, Severe 80 wpm.
- **Best practice:** present transcripts and highlight nodes during reading so developers can follow.

3.2 Motor Impairments

3.2.1 Delayed Input (latency)

- **Instrumented mode:** wrap event handlers in page to introduce delay; map severity to delay durations:
  - Mild: 150 ms

    - Moderate: 400 ms
    - Severe: 800+ ms
- **Client mode alternative:** overlay intercepts clicks and forwards them to iframe after delay, but this may not work reliably for cross-origin pages — prefer instrumented mode.

3.2.2 Shaky Cursor

- **Visual-only client approach:** display a floating cursor overlay that visually jitters while pointer events go straight through — gives user a feel of tremor without modifying target.
- **Instrumented approach:** inject small random offsets to mousemove before dispatching events to page handlers.
- **Parameters:** jitter amplitude (±px):
    - Mild: ±2 px
    - Moderate: ±6 px
    - Severe: ±12 px
- **Safety:** keep jitter low enough not to make the page unusable; provide quick disable.

3.2.3 Restricted / Imperfect Key Input

- **Simulations:** require double-press for activation, random ignored keypresses, or sticky keys.
- **Mapping examples:**
    - "Require double enter" (Moderate)
    - "Randomly drop 10% of key events" (Moderate)
- **Implementation:** instrumented DOM event wrappers are required for reliable behavior.

3.3 Cognitive Impairments

3.3.1 Delayed Content Reveal

- **Method:** delay rendering of certain DOM regions to simulate processing time — e.g., 500 ms, 1500 ms, 3000 ms for mild/moderate/severe.
- **UX tip:** visually indicate that content is "loading" to separate real performance slowness from simulation.

3.3.2 Random Distractions

- **Methods:** subtle overlays, background noise, or transient popups.
- **Caution — safety warning:** do **not** create flashing/strobing effects without explicit opt-in. Avoid patterns likely to trigger photosensitive epilepsy. Default to non-flashing, low-contrast distractors; provide an explicit, prominent consent step before enabling anything that could cause discomfort.
- **Sound usage:** background noise (low volume) via Web Audio API; include mute control.

3.3.3 Memory Overload / Attention Difficulty

- **Simulations:** temporarily blur or dim previously seen elements, show ephemeral highlights, or require reconfirmation of previously seen form data.
- **Design goal:** nudge developers to reduce cognitive load (simpler forms, progress indicators, fewer interruptions).

---

4. Feedback & Analytics — practical rules

4.1 Integrating axe-core for actionable results

- For instrumented pages, run axe-core within the page context to detect WCAG violations. For client-only mode, run axe against a proxied page or provide a non-interactive checklist.
- Normalize axe results into categories: Critical / Serious / Moderate / Minor.
- Map each axe rule to plain-language remediation guidance (not just the rule key).

4.2 Mapping simulation findings to developer recommendations

- Combine axe results with simulation observations:
    - Example: color-blind simulation + contrast issues → recommend added non-color labels, patterns, or icons.
- Provide triage severity for recommendations (High/Medium/Low) with examples and links to WCAG guidelines.

4.3 Analytics & visualization

- Show at-a-glance charts: Issues by category, % Pass vs Fail, Simulation impact heatmap.
- Provide timeline of applied simulations and screenshots for comparison (Before / After).
- Store analytics per session with unique session IDs and timestamps.

---

5. Reporting Layer — content & formats

5.1 Report contents (what each exported report should include)

- Report header: Target URL, timestamp, tester name (optional), mode (client/instrumented), consent note.
- Executive summary: high-level score, major issues, top 3 recommended fixes.
- Simulation log: list of simulations applied with severity and parameter values.
- WCAG / axe summary: counts by severity + list of top violations and explanation.
- Visual evidence: screenshots for each simulation state (if privacy/terms allow).
- Detailed remediation guidance: per issue actionable steps and code sample references (links only).
- Appendix: raw JSON of axe results + session metadata.

5.2 Formats & practical notes

- **JSON:** canonical, includes full data and raw axe output. Use as primary export for programmatic consumption.
- **CSV:** flattened summary table for spreadsheets. Columns: session_id, url, simulation_type, severity, issue_key, issue_severity, recommended_fix.
- **PDF:** human-readable report built from a template with logos, charts, screenshots. Use server-side rendering for consistent output (Puppeteer or server PDF library).

5.3 Storage & retention

- If storing reports/screenshots, define retention policy (e.g., delete after 30 days) and allow users to opt out of storage.
- Store minimal personal data and secure access to reports (authenticated endpoints, secure object storage).

---

6. Backend & Instrumentation — practical architecture

6.1 When to use a headless browser

- Required if:
  - Target site blocks iframe embedding.
  - You need to inject JS/CSS into the page (keyboard/mouse simulation, screen reader).
  - You need reliable axe runs inside page context.
  - You need screenshots of final rendered page.

6.2 Recommended backend design

- **Stateless API** receiving simulation job requests, returning job IDs.
- **Worker pool** that handles headless browser instances and performs:
    - Fetch page, obey robots/ToS check.
    - Optionally sanitize / remove third-party tracking scripts.
    - Inject simulation scripts and run axe-core.
    - Capture screenshots + normalized axe results.
- **Queue & rate limiting:** use Redis + Bull or equivalent to queue jobs to prevent headless instance overload.
- **Storage:** store generated reports/screenshots in object storage; return secure signed URLs to client.

6.3 Security & isolation

- Run headless browsers in containers or sandboxed processes, avoid persistent state across jobs.
- Apply network egress restrictions to prevent the worker from accessing internal networks.
- Scan returned HTML for malicious content before re-serving (if you serve instrumented HTML).

---

7. Dependencies, Dev Environment & Tooling (practical tips)

7.1 Frontend essentials

- Recommended: modern React + component library for rapid UI. But you can implement plain JS if you prefer.
- Include robust state management for session state (e.g., Redux, Context).
- Accessibility of your own UI: apply WCAG to control panel (keyboard nav, screen reader labels).

7.2 Backend essentials

- Node.js + Express + Puppeteer or Playwright recommended for parity across browsers.
- For Python shops, Playwright Python + Flask works too, plus ReportLab for PDFs.

7.3 Libraries

- axe-core (always) — run it in instrumented mode in the page context, not only in the parent.

- Chart.js or D3.js for visuals — Chart.js is simpler to integrate.
- jsPDF or server-side Puppeteer for consistent PDF rendering.

7.4 Local dev

- Provide a dev mode that allows using local test pages (same origin) to test full instrumented features without a backend.
- Use Docker to create reproducible Puppeteer environments (headless Chrome in container).

---

8. Deployment, Scaling & Ops

8.1 Hosting choices

- **Frontend-only deployment:** Netlify or Vercel for static SPA.
- **Backend & workers:** Railway, Render, or DigitalOcean App Platform. For heavier loads, use Kubernetes with autoscaled worker pods.
- **Storage:** S3 (or S3-compatible) for screenshots/reports.

8.2 Scaling the headless workers

- Limit concurrent headless instances per worker node to preserve memory/CPU.
- Use autoscaling rules keyed to queue length.
- Cache static assets and instrumented copies where possible.

8.3 Observability & monitoring

- Track: job queue length, average job duration, failure rate, CPU/memory for worker pods.
- Logging: store only necessary logs; redact URLs if required for privacy.

---

9. Testing & Validation (practical checklist)

9.1 Functional tests

- Verify client mode supports all visual filters with target iframe pages.
- Verify instrumented mode can:
    - Inject CSS/JS into proxied pages.

    - Run axe-core and return normalized results.
    - Simulate motor and cognitive conditions inside the page.

9.2 Integration tests

- End-to-end workflow: submit URL → run job → view report → download PDF.

9.3 User testing

- Recruit a mix of users (developers, designers, and ideally people with disabilities) to validate realism and usefulness.
- Collect structured feedback: realism of simulations, clarity of remediation guidance, UI accessibility.

9.4 Performance testing

- Load test worker capacity (simulate many jobs).
- Test time-to-first-result for typical pages (expect 2–10s for simple pages; 10–30s for complex instrumented runs).

---

10. Privacy, Ethics & Safety (must-do items)

- **Explicit opt-in** for simulations that could distress (distractions, sound, flashing). Provide a popup with detailed warnings before enabling those modes.
- **No flashing** by default. If you support flash/distracting elements, require a two-step opt-in and an easily visible kill switch.
- **Respect ToS & robots** — do not proxy or store content from sites that disallow scraping unless you have explicit permission.
- **Data retention policy:** make it transparent and provide deletion mechanisms.
- **Accessibility of the tool itself:** ensure the control panel is keyboard accessible, labels exist, and color contrast is good.

---

11. Failures, Fallbacks & UX expectations

- If embedding fails:
    - Show clear message with three actions: (1) try Snapshot mode, (2) use Instrumented mode (if available), (3) provide manual upload option (developer can upload static HTML).

- If instrumented job times out: report partial results (e.g., axe results for loaded parts) and offer retry.
- Document which features are available in which mode (client vs instrumented) in the UI.

---

12. Example session schema (fields to store / export)

Use as a checklist for what to include in JSON/PDF/CSV exports:

- session_id (unique)
- created_at (timestamp)
- target_url
- embedding_mode (iframe / snapshot / instrumented)
- simulations_applied: list of {type, severity, parameters}
- axe_summary: {violations_count_by_severity}
- top_issues: list of {rule_id, description, nodes_affected, remediation}
- screenshots: URLs (before/after if available)
- performance: {load_time_ms, analysis_time_ms}
- recommendations: list of textual, prioritized actions
- consent_flags: {proxy_accepted: true/false, store_reports: true/false}

---

13. Roadmap & incremental implementation plan (practical phasing)

1. **Mvp (Client-only):**
    - iframe embedding, visual filters (blur/contrast/grayscale/zoom), overlay cursor, control panel, PDF basic export (metadata + simulation log), basic axe runs for same-origin pages.
2. **Phase 2 (Instrumented):**
    - Backend with Puppeteer, injected DOM simulations (motor/cognitive), axe-core runs for cross-origin, screenshots, JSON export.
3. **Phase 3 (Polish & hardening):**
    - User accounts (optional), retention & access controls, advanced analytics, presets, accessibility of simulator UI, peer testing & refinement.

Final practical checklist (what you should do next)

- Decide which operational mode(s) to support for your semester timeline.
- Implement the control panel and client visual filters first (fast wins).
- Build a safe backend prototype (single worker) for instrumented tests if you plan full simulation.
- Create thorough consent & privacy flows before storing target pages or screenshots.
- Write templated remediation guidance mapped from axe-core rule keys — this is high value for developers.
- Test with sample sites and real users; iterate on realism and safety.