ChatGPT

Executive Summary

Building a Firefox-based browser fork in 3 weeks is extremely ambitious but **achievable** for an MVP with disciplined planning, the right tools, and AI assistance. The fastest approach is to leverage existing frameworks and tools rather than coding the build process from scratch. We recommend using *Surfer* (a build framework from the Zen team) or Mozilla's own `mach` system to fetch, patch, and compile the Firefox source [1] [2]. This framework-driven method avoids redoing complex build logic. Key tasks include rebranding (updating product name, icons, splash screen), theming (UI colors), and adding Zen-like features (Workspaces, Compact Mode). AI tools (e.g. GitHub Copilot, ChatGPT, Copilot Chat) will accelerate coding by generating boilerplate and debugging help.

Feasibility: A solo developer can deliver a *proof-of-concept* MVP in 3 weeks by tightly focusing on priority features: custom branding, theming, vertical tabs (Workspaces), and Compact Mode. Advanced features like Split View and Glance would be "nice-to-have" and can be deferred. A powerful development machine and AI code assistants will be critical to meet the schedule. Key risks are build system complexity, cross-platform bugs, and unforeseen integration issues. We mitigate these by using battle-tested tools (Surfer/`mach`), frequent automated builds/tests, and fallback plans (e.g. simpler UI modifications if advanced ones stall).

Approach: We recommend a *framework-based* approach. Surfer (a fork of Gluon) already automates downloading and patching Firefox [1]. It is MPL-licensed and designed exactly for building Firefox forks. Using it (or adapting Mozilla's official build steps) will be far faster than reinventing the wheel. Surfer can be configured via a project file (e.g. `surfer.json`) to apply patches and branding. Key challenges include understanding Firefox's modular source tree and build system, which Surfer/Gluon help manage. For branding, Firefox stores channel-specific names and icons in `browser/branding` (e.g. the "official" directory) [3]. We will clone the appropriate directory, replace images and strings, and set a new product name.

Key Challenges & Solutions: The large Firefox codebase and its C++/Rust build is daunting. To address this, we will use AI-assisted IDEs (like Copilot) for code navigation and generation, and start with a simple build config before adding complexity. Custom UI features require editing XUL/CSS or using the new WebExtensions APIs. AI agents can help write these modifications by generating snippets (for example, CSS to auto-hide the tab bar). Building for all platforms is another hurdle. We will use continuous integration (e.g. GitHub Actions) to automate Windows, macOS, and Linux builds, and follow Mozilla's multi-platform docs [4] [5]. Code signing and update servers will borrow Zen's approach (see below).

Detailed Implementation Plan

Firefox Fork Strategy

1. **Set up Build Environment:** Install Mozilla's build prerequisites: Python3, Rust, Node.js, and for each platform the required toolchain (Visual Studio for Windows, Xcode for macOS, GTK deps for Linux). Use a modern machine (16+ GB RAM, SSD) as builds are heavy.

2. **Acquire Source Code:** Use `git clone https://github.com/zen-browser/desktop.git --recurse-submodules` (Zen's repo) or `hg clone https://hg.mozilla.org/mozilla-central` if not using Surfer. We recommend using Zen's *surfer* tool (npm install @zen-browser/surfer [1]) to bootstrap, since it automates fetching the correct Firefox version.
3. **Configure Build:** Create a `mozconfig` file to define branding and build options [6] . E.g.:

   ```
   echo "ac_add_options --disable-crashreporter" >> mozconfig
   echo "ac_add_options --enable-application=browser" >> mozconfig
   ```

   Use `MOZILLA_OFFICIAL=1` if building for an "official" release, which tells the build to use the `browser/branding/official` directory [3] .
4. **Apply Branding & Name:** Replace Firefox images and names in the `browser/branding/official` directory (icons, splash). Also edit the resource bundles: e.g., modify `browser/locales/en-US/chrome/browser/browser.dtd` strings and `toolkit/` locales to change "Mozilla Firefox" to your browser name. (Zen's support forum shows exactly these files [7] .) Ensure the new app name appears in the About dialog and executable.
5. **Customize UI Theme:** Put custom colors/themes in `browser/themes/`. Firefox uses CSS for themes; for heavy UI changes, modify XUL/CSS in `browser/themes/` or use `userChrome.css`. For example, to auto-hide the horizontal tab bar (Compact Mode), add:

   ```
   @namespace url("http://www.mozilla.org/keymaster/gatekeeper/
   there.is.only.xul");
   #TabsToolbar { visibility: collapse !important; }
   ```

   to `userChrome.css`. AI can help generate these CSS rules via prompts (see AI prompts below).
6. **Implement Workspaces:** Zen's Workspaces logically group tabs. We have two options: (a) integrate code from Zen (if open) or (b) implement as an extension. In 3 weeks, the fastest is a WebExtension that manages window layouts. For example, use the `browser.windows.create()` and `browser.tabs` APIs to simulate workspace groups. Create a sidebar button or context menu to move tabs to named groups. This approach avoids deep C++ hacking.
7. **Compact Mode (Auto-hide Tab Bar):** Use CSS/JS to make the tab sidebar collapsible. For example, detect window resize or a button click to toggle `#TabsToolbar`. Zen did this with an "auto-hide" feature in their sidebar. We can prototype this with an event listener in a browser overlay script.
8. **Initial Build:** Run `mach build` (or `surfer build`) to compile. This may take hours. Fix any build errors (missing dependencies, path issues). Use `mach package` to create initial unbranded binaries for testing.
9. **Test Basic Functionality:** Run the browser, check that branding (name/icon) is updated and that extensions still work. Verify the homepage/prefs have custom defaults via about:config.
10. **Iterate on Features:** Gradually add Workspaces and Compact Mode, building and testing frequently. Use AI chat assistants to debug errors (e.g. "Why isn't my CSS rule applied to hide the tab bar?"). Ensure Firefox sync and extensions still function.
11. **Milestone Builds:** At end of each week, produce a "beta" build to use as milestones (branded, theming, minimal features working by Day 7; core features working by Day 14; polish and packaging by Day 21).

Architecture Overview

Firefox's source is modular [4] [3]. Key directories:
- `browser/`: Front-end code (XUL, JS, C++) for the desktop UI [4]. We will modify files here for branding and add overlays for new UI elements.
- `browser/themes/`: Contains OS-specific UI skins (images/CSS) [8]. Custom colors and icons go here.
- `browser/branding/`: Branding assets (logo, name, about-page strings) for Release builds. By default "official" is used [3]. We will replace these for our brand.
- `toolkit/`: Shared UI libraries (XUL widgets, preferences, overlays) [9]. Some key dialogs (e.g. about windows, preferences) live here.
- `config/` **&** `build/`: Build config scripts (autoconf, makefiles). We use them indirectly via `mozconfig`.
- `dom/`, `js/`, **etc.**: Core engine and JavaScript engine. These are left untouched; using a fork retains Gecko's rendering engine and security updates.

We will primarily customize under `browser/` and `toolkit/`. For example, Zen's vertical-tab sidebar and Glance features are likely implemented in modified XUL/CSS and JS under `browser/base/content` and related UI components. We will look at their source for guidance.

21-Day Development Schedule

- **Days 1–2 (Branding & Design):**
- **Milestone:** Finalize browser name, color scheme, logo concept.
- Activities: Brainstorm brand identity (Zen's aesthetic is *calm* and *clean* [10]). Choose a simple name and palette. Use AI image generators (DALL·E 3, Midjourney, or Stable Diffusion) with prompts (e.g. *"Generate a modern, minimalist blue-green abstract logo with a calm, zen theme"*) to create logo ideas [11]. Refine concept in vector graphics software. Finalize app colors (light mode/dark mode).
- **Day 3 (Dev Environment Setup):**
- **Milestone:** Build environment ready on Windows, macOS, Linux.
- Install development tools: VS Code or JetBrains (with GitHub Copilot/AI plugin), Python 3, Rust toolchain, Node.js. On Windows install MozillaBuild (Visual Studio, NASM, etc.). On macOS install Xcode and brew libraries. On Linux install build deps (GTK, etc.). Clone the Firefox source (version 139) using Surfer or Mercurial. Write a basic `mozconfig` to do a test build.
- **Day 4 (Initial Build & Branding):**
- **Milestone:** Successfully compile the original Firefox code; integrate initial branding assets.
- Run `mach build` to ensure the clean source builds on each OS. Use Surfer: `surfer init` and `surfer build`. Replace branding: copy new icon files into `browser/branding/official/` and edit `browser/branding/official/all-strings.dtd` to change product names. Update `about:config` defaults (e.g. homepage) in `defaults/pref`. Build again, verify the app name/icon reflect changes.
- **Day 5 (Theming and UI Scaffold):**
- **Milestone:** Apply custom theme colors; set up UI layout changes (vertical tab bar placeholder).
- Edit CSS in `browser/themes/` or `userChrome.css` to apply new color scheme. Hide the default horizontal tab bar (`#TabsToolbar`) to prepare for vertical tabs. Test that UI doesn't break. Implement a simple toggle (e.g. a button) to collapse/expand tabs. Use AI to generate missing CSS (prompts like *"Firefox userChrome.css rules for vertical tab bar"*). Verify theme (light/dark) on both.
- **Day 6 (Workspaces Prototype):**

- **Milestone:** Basic workspace mechanism working (group tabs by topic).
- Option A: **Extension approach.** Create a minimal WebExtension: add a sidebar panel that lists "Workspaces." Use `browser.tabs` API to move tabs into new windows named by workspace. For example, a JSON structure in extension storage tracks groups. AI prompt could be *"Generate JavaScript for a browser extension that moves a tab to a new Firefox window called 'Workspaces'."*.
- Option B: **Core hack.** Edit `browser.xhtml` code to add a workspace switcher button. Given time constraints, Option A is faster.
- Test by creating a few workspaces (e.g. "Work", "Personal") and verifying tabs are isolated.
- **Day 7 (Compact Mode):**
- **Milestone:** Sidebar auto-hide functionality.
- Using CSS or JS, implement Compact Mode: when no tab is selected for a period or on hover, the sidebar hides. For example, detect `mouseout` from sidebar and collapse it. AI prompts like *"Write JavaScript to auto-hide an HTML element when the mouse leaves it"* adapted to XUL can help. Verify that moving the cursor to sidebar edge reveals it.
- **Day 8 (Weekend Buffer/Risk Mitigation):**
- **Milestone:** Catch-up day for any lagging tasks, or exploratory prototypes for Split View.
- Use this time to troubleshoot build issues, solidify branding, or skip ahead to packaging tests.
- **Day 9 (User Preferences & Defaults):**
- **Milestone:** Default preferences customized; unused features disabled.
- Edit `default prefs`: e.g. disable telemetry (`toolkit.telemetry.unified`), set new homepage (`browser.startup.homepage`), and any default extension (if any). Test that a fresh profile shows these defaults.
- **Day 10 (Testing & Refinements):**
- **Milestone:** Integrate continuous testing and fix UI bugs.
- Write basic automated tests (e.g. using `mach xpcshell-test` or `mach ui-test` if feasible) to verify core functions (browser launch, tabs, sync). QA the UI: ensure icons/buttons are visible, keyboard shortcuts (especially on macOS) work, no crashes. Collect user feedback (myself) and polish quickly.
- **Day 11 (Second Feature Pass):**
- **Milestone:** Polish Workspaces (pinned/essential tabs); implement WebExtension if needed for remaining bits.
- Enhance workspaces: allow "pinned" tabs visible in all workspaces and "essential" tabs that stay. We can simulate by copying URL to all windows or disabling unload. If too complex, skip.
- **Day 12 (Optional Split View):**
- **Milestone:** Simple split view prototype (nice-to-have).
- If time permits, use CSS grid or multi-panel to tile two tabs. For example, pressing a hotkey could place two tabs side-by-side using CSS transforms. This is low-priority; skip if over time.
- **Day 13 (Documentation & Final Logo):**
- **Milestone:** Finalize logo/icon set; write initial docs.
- Use AI prompts (e.g. "Refine logo concept into app icon") and vectorize the chosen design for high-res icons. Update splash screen. Document the build process and user features in a README.
- **Day 14 (Alpha Release & Feedback):**
- **Milestone:** Publish an alpha build on GitHub; gather early feedback.
- Package the browser (unbranded name) and push to a private repo release. Test updates manually by downloading new MARs. Ensure crash reporter is off (or points to custom server).
- **Day 15 (Bug Fixes & UI Tweaks):**
- **Milestone:** Address any major issues from alpha.

- Fix lingering UI glitches, performance lags (e.g. slow sidebar animations), and ensure extensions remain compatible. Use AI to debug (e.g. “Examine this error log from Firefox build”).
- **Day 16 (Platform Testing – Windows):**
- **Milestone:** Build and test Windows installer.
- Use Mozilla’s docs to set up a Windows build (Visual Studio). Run `mach build` on Windows, fix any OS-specific build failures. Generate an NSIS installer (`mach package`, `mach package --format=nsis`). Code-sign the EXE (trial with test cert). Verify install/uninstall.
- **Day 17 (Platform Testing – macOS):**
- **Milestone:** Build and test macOS app.
- On a Mac machine, compile with Xcode. Create a DMG or zip. Use `codesign` to sign the .app (Apple Developer cert required). Verify it runs on another mac. Adjust entitlements as needed.
- **Day 18 (Platform Testing – Linux):**
- **Milestone:** Build and test Linux packages.
- Compile on Ubuntu 22.04 (or Fedora). Create an AppImage and .tar.xz. Use `linux-mar` scripts to generate update .mar files for Linux, macOS, and Windows (Zen’s release logs show this step [5]). Upload a test build to a Zen-like update server (or simulate locally).
- **Day 19 (Update Server & Deployment Setup):**
- **Milestone:** Set up update server and release infrastructure.
- Deploy a simple update server (e.g. the Zen *updates-server* code or a minimal Flask app) at `updates.your-browser.app`. Configure `app.update.url.override` preference to point to it [12]. Prepare GitHub Actions workflows to build on each push/tag for all platforms, and publish to GitHub Releases (with checksums) [5].
- **Day 20 (Final QA & Website):**
- **Milestone:** Perform final cross-platform QA and finalize website.
- Test the entire update flow: install an older version and receive a notification to update (this may be manual). Finalize a simple landing page (could be a GitHub Pages site) with download links and a brief feature list.
- **Day 21 (Buffer & Release):**
- **Milestone:** Bug fix day and public release.
- Fix any blocker issues. Tag and publish the 1.0 release on GitHub, including `.exe` / `.dmg` / AppImage and source tarball (with SHA-256 sums). Announce the release (e.g. on social or forums) and prepare for user feedback.

AI Integration Strategy

We will leverage AI at every stage:

- **Code Generation & Assistance:** Use GitHub Copilot (in VSCode or JetBrains) to autocomplete functions and find API usage. For example, write a prompt like “Create a Firefox WebExtension that groups open tabs into named workspaces using the browser.tabs API.” Copilot/ChatGPT can draft the initial code, which the developer refines.
- **Documentation & Learning:** Query ChatGPT or GPT-4 for explanations of Firefox internals (“How to find where Firefox defines the About dialog title?”) or build errors. Use local LLM (e.g. GPT4All) for offline access.
- **UI Design & Branding:** Use DALL·E 3 or Stable Diffusion for logo concepts. Involve AI design assistants (Canva AI, Adobe Firefly) to quickly iterate icon ideas. For theming, ask AI for CSS color palette suggestions given “calm, green, blue” brand prompts.

- **Testing & Debugging:** Employ AI to analyze test failures. For example, prompt “The Firefox build failed with this error [log]. What might be wrong?” Copilot Chat can suggest missing dependencies or syntax fixes. Also, use AI to automatically generate unit tests via “Given this feature description, propose test cases.”
- **AI Toolchain Setup:** Recommended AI tools:
- **GitHub Copilot** for code completion (integrates with VSCode or JetBrains) [13] . Instructions: install the Copilot extension in VSCode, sign in with GitHub, and enable Copilot Chat for Q&A.
- **OpenAI API (ChatGPT/GPT-4)**: For ad-hoc queries. Alternatively, **Local LLMs** (e.g. LLaMA-based) for offline use, integrated via VSCode plugins (Tabnine or CodeWhisperer as backup).
- **AI-Powered Linters**: Tools like SonarLint or DeepSource may catch bugs.
- **Design AI**: Use Canva’s AI logo maker or GoDaddy’s Airo logo generator [11] . For vector icons, consider DALL·E/Stable Diffusion plus manual tracing.

Each day, the developer should have these agents at hand (in-chat, IDE) to speed tasks. For example, use Copilot to scaffold a `mozconfig` or to learn the syntax for hiding the tab bar, then verify and adjust manually.

Risk Assessment

- **Build Failures:** Building Firefox often fails due to missing libs or misconfigured mozconfig. *Mitigation:* Start with a known-good build (e.g. Zen’s surfer did this) and add changes incrementally. Keep a backup branch with the last working commit.
- **Platform Bugs:** Cross-compile issues are common. *Mitigation:* Test each OS early (by Day 16 onward). Use Docker/VMs or CI for Linux, real machines for macOS. Prioritize fixing platform-specific issues immediately.
- **Feature Overruns:** Some features (like Split View, Glance) may be too complex to implement fully. *Mitigation:* Make them stretch goals. If workspaces or Compact Mode consumes extra time, cut Split View entirely.
- **AI Hallucinations:** Relying on AI can produce incorrect code. *Mitigation:* Always review AI output critically. Use citations (e.g. Mozilla docs) to verify.
- **Time Management:** 21 days is tight. *Mitigation:* Strictly follow the schedule; use buffer days. De-scope aggressively if needed.

Technical Specifications

Minimum System Requirements

- **Development Machine:**
- **CPU:** Quad-core (preferably 6th gen or newer Intel/AMD) or Apple M1/M2.
- **RAM:** 16 GB (32 GB recommended for large builds).
- **Storage:** 50 GB free (SSD preferred).
- **OS:** Windows 10+, macOS 12+, Ubuntu 20.04+ (or similar).
- **Target Environments:**
- **Windows:** 64-bit (Intel/ARM). Code-signing requires a Windows Server environment with EV certificate.
- **macOS:** 64-bit Intel and Apple Silicon. Needs an Apple Developer account for signing.

- **Linux:** Recent distributions (we will provide an AppImage for broad compatibility).
- **Android:** (Stretch target) use Firefox for Android (Fenix) source with GeckoView. Requires Android SDK/NDK.
- **iOS:** (Stretch) Not feasible in 3 weeks due to Apple's WebKit requirement; would need a separate approach (e.g. using Xcode's WKWebView and building a minimal browser). This is beyond MVP scope.

Development Environment Setup

1. **Install Dependencies:**
2. **Python 3.9+**, **Rust (rustup)**, **Node.js 16+**, and **NPM**.
3. On **Windows**: install MozillaBuild (includes VS2019/VS2022 compilers, NASM, Python) [6].
4. On **macOS**: install Xcode (with CLI tools) and Homebrew (for libraries: `brew install gtk+3 libXtst`).
5. On **Linux**: install `build-essential`, `autoconf2.13`, `libgtk-3-dev`, `libxt-dev`, etc. Also Rust and Node.
6. **Clone Source:**

```
git clone https://github.com/zen-browser/desktop.git --recurse-submodules
cd desktop
# Alternatively: npm install @zen-browser/surfer then surfer init
```

7. **Set Up mozconfig:**
   Create `./mozconfig` as shown above [6]. For example, to disable unused features. This file drives the configure/build.
8. **Configure IDE:**
9. **VSCode**: Install *Mozilla Build Debugger* extension and *GitHub Copilot*.
10. **JetBrains**: Install *Rust*, *JavaScript*, and *Copilot* plugins.
11. Configure git remotes for easy commits.
12. **Test Build:**
    Run `./mach build` (or `surfer build`) on one platform to confirm setup. It should download dependencies and compile (initial run ~hour). If errors occur, check the Firefox docs for common issues (e.g. Autoconf version).

Core Feature Specifications

- **Branding:**
- App name in About, EXE name, bundle ID, etc. All icons (application icon, toolbar icon, splash) replaced.
- Default homepage and search engine set to brand's choice via prefs.
- **UI Theming:**
- Custom color scheme applied to toolbars and dialogs.
- Theme switch if needed (light/dark).
- **Workspaces:**
- Ability to create multiple named workspaces. Each workspace contains its own set of tabs/windows.
- "Switch workspace" UI (e.g. in sidebar or menu).

- Optionally, designate “essential” tabs visible in all workspaces.
- **Compact Mode:**
- Auto-hide vertical tab sidebar (and show when hovering or clicking a toggle).
- Hiding the horizontal tabs when sidebar is active.
- **Split View (optional):**
- Tiling of two tabs within one window. Shortcut or drag-and-snap to invoke. (Deprioritized for MVP)
- **Glance (optional):**
- Opening a link in a temporary modal overlay (like a quick peek) rather than full tab. (Nice-to-have)

All features should be enabled by default in the product, not require user extensions. They should not break normal browsing. E.g., Workspaces should use separate `windows`, Compact Mode uses CSS, etc.

Testing Strategy

- **Unit/Integration Tests:** Use Mozilla’s existing test suite as a base. Run `mach mochitest` and `mach crashtest` where relevant. Add simple tests if possible (e.g. ensure branding strings appear).
- **Manual QA:** On each platform, perform smoke tests: open multiple tabs, switch workspaces, hide/show tabs, browse popular sites, sync Mozilla account, install a Chrome/Firefox extension (to ensure compatibility). Use developer tools to catch console errors.
- **Cross-Platform:** Maintain a build matrix on GitHub Actions to compile on Windows/macOS/Linux for each commit, failing fast on build errors.
- **User Feedback:** If possible, have a few users or colleagues try early builds and report major issues.
- **Performance Tests:** Run a browser benchmark (e.g. Speedometer) and compare to baseline Firefox [14] to ensure no regressions beyond expected UI overhead.

Performance Benchmarks

Zen’s own benchmarks show it scores slightly lower than stock Firefox (Zen 1.7.3b scored 31.6 vs. Firefox 34.8 on Speedometer) [14]. We should expect similar results: our fork’s custom features may cause a small slowdown. The goal is to stay within ~10% of Firefox’s performance. Use profiling (about:performance) if needed and optimize hot paths (e.g. avoid excessive reflows in CSS, minimize JS overhead in UI). Track memory usage; extra UI (like always-ready sidebar) may use a bit more RAM. Overall, without changing the engine, performance should remain close to Firefox’s [14].

Resource Library

- **Mozilla Docs:**
- *Firefox Source Docs – Directory Structure* [4] (understand modules like `browser/`, `toolkit/` for UI).
- *Firefox Branding* [3] (details on `browser/branding` folders and channels).
- *Build System Overview* [2] (how `configure` / `mach build` works).
- *Configuring Build Options* [6] (how to set up `mozconfig`).
- *Setting Up an Update Server* [15] (guide for serving Firefox MAR updates).
- MDN Web Docs (developer.mozilla.org) for **WebExtensions** APIs and **XUL/CSS** customization.
- **Zen Browser Resources:**

- *Zen Browser GitHub (desktop)* – study file structure, commits, and issue history for clues on where features live.
- *Zen Browser Docs* – docs.zen-browser.app/guides/building (instructions to build Zen, e.g. language pack scripts and `mach` usage).
- *Zen Update Server Repo* (zen-browser/updates-server) – shows how updates are hosted.
- *Zen Wikipedia* – for feature summaries: e.g. vertical tab bar, workspaces, Glance [16] [17].
- **AI & Dev Tools:**
- *VSCode Copilot Setup* [13] – official guide to install Copilot.
- *GitHub Copilot Documentation* (docs.github.com).
- *AI Logo Design Guide* [11] – advice on using AI for logo generation and prompt-writing.
- **Code Examples & Boilerplates:**
- **Branding Snippet:** Example prefs in `defaults/pref/mydefaults.js`:

```
pref("toolkit.telemetry.unified", false);
pref("browser.startup.homepage", "https://your-brand.example.com");
```

- **UI CSS Snippet:** In `userChrome.css` to hide horizontal tabs:

```
@namespace url("http://www.mozilla.org/keymaster/gatekeeper/
there.is.only.xul");
#TabsToolbar { visibility: collapse !important; }
```

- **Workspaces Extension Sketch (JS):**

```
// Popup script for Workspaces (using browser.windows API)
browser.windows.create({titlePreface: "Workspace: Project A"}).then(win =>
{
  browser.tabs.create({windowId: win.id, url: "https://example.com"});
});
```

- **Update URL Override:** In `defaults/pref`:

```
pref("app.update.url.override", "https://updates.your-browser.app/update/
3/%PRODUCT%/%VERSION%/%BUILD_TARGET%/%BUILD_ID%/%BUILD_ARCH%/update.xml");
```

- **AI Prompt Templates:** *(Use as starting points for ChatGPT/Copilot)*
- *Feature Implementation:* "Write a Firefox extension that lets the user group tabs into named workspaces using a sidebar UI."
- *Build Configuration:* "Generate a mozconfig file that builds Firefox with the browser UI only and disables crash reporter."
- *UI Customization:* "Create CSS rules for Firefox's user interface to implement a vertical tab sidebar and hide the normal tab strip."
- *Debugging Help:* "The Firefox build failed with an 'undefined symbol in libxul' error. What does it mean and how to fix it?"

- *Packaging:* "What are the steps and tools to create a signed Windows installer and macOS DMG for a custom Firefox browser?"
- **Debugging & Troubleshooting:**
- **Build Issues:** Check `build/README.md` and MOZILLA_DEV or MDN for missing dependencies. Ensure `python` in path and `autoconf2.13` present.
- **Firefox Crash:** Run with `--no-remote --profile tempdir` to isolate issues. Use `about:crashes`.
- **Update Failure:** Confirm update XML and MAR files are correctly generated (use `./mach package`). Check `app.update.channel` preference.
- **UI Bugs:** Inspect with Browser Toolbox (developer mode) to see if overlays/CSS are applied.
- **Platform-Specific:** On Linux, missing GTK themes can cause blank icons; install system icons. On Windows, ensure PATH variables point to Visual Studio tools.

Alternative Scenarios

- **Plan A: Full Firefox Fork (like Zen)** – *Comprehensive customization.* Use Surfer/Gluon to fork the entire Firefox codebase [1]. This allows deep UI changes (vertical layout, new features in core UI). It is the most flexible but also the most work. Use this if you need total control and can handle the complexity.
- *Pros:* Fully customized UI, integrated features (Zen's approach).
- *Cons:* Slow to implement, complex merging with new Firefox updates.
- **Plan B: Lightweight Fork** – *Minimal modifications to Firefox.* Instead of heavy UI hacks, only change branding and maybe a few about:config defaults. Possibly use an add-on for extra features. This reduces coding: e.g. leave Firefox's UI intact except for theme colors or optional sidebar toggles.
- *Pros:* Much easier and faster; most of Firefox's behavior untouched (fewer bugs).
- *Cons:* Cannot fully recreate Zen's unique interface; limited feature set.
- **Plan C: Theme/Extension Approach** – *No custom build.* Use standard Firefox (ESR) and distribute only a theme (userChrome.css) plus extensions (WebExtensions) to mimic features. For example, use a vertical-tabs add-on, a workspace manager extension, and a userChrome theme for colors.
- *Pros:* Quickest to start; no need to build Firefox. Updates handled by Mozilla automatically.
- *Cons:* Extremely limited branding (can't change app name or update channel), and features are clunkier. Not a true fork or standalone browser.

Plan C is fastest but very limited – good only as a **proof-of-concept hack**. Plan A is recommended if the goal is a **full-featured branded browser** like Zen; Plan B is a middle ground if development time is too short, implementing only the highest priorities with minimal code changes.

Conclusion: This plan details every step needed to fork Firefox into a Zen-like browser. By following the timeline, using AI assistance, and focusing on critical features, you can build a working prototype within 3 weeks [1] [14]. The provided citations and resources ensure you know exactly which files and tools to use.

After 21 days, you'll have a cross-platform, branded browser MVP ready for public release, along with a roadmap for further improvements.

---

1 GitHub - zen-browser/surfer: Simplifying building firefox forks!
https://github.com/zen-browser/surfer

2 Build System Overview — Firefox Source Docs documentation
https://firefox-source-docs.mozilla.org/build/buildsystem/build-overview.html

3 browser/branding - mozsearch
https://searchfox.org/mozilla-release/source/browser/branding/

4 8 9 Firefox Source Code Directory Structure — Firefox Source Docs documentation
https://firefox-source-docs.mozilla.org/contributing/directory_structure.html

5 Releases · zen-browser/desktop · GitHub
https://github.com/zen-browser/desktop/releases

6 Configuring Build Options — Firefox Source Docs documentation
https://firefox-source-docs.mozilla.org/setup/configuring_build_options.html

7 Modify Firefox source code to compile a browser with different name and icon | Firefox Support Forum | Mozilla Support
https://support.mozilla.org/en-US/questions/1243141

10 16 17 Zen Browser - Wikipedia
https://en.wikipedia.org/wiki/Zen_Browser

11 AI Logo design guide - How to create a logo with AI
https://www.godaddy.com/resources/skills/ai-logo-design

12 App.update.url.override - MozillaZine Knowledge Base
http://kb.mozillazine.org/App.update.url.override

13 Set up GitHub Copilot in VS Code
https://code.visualstudio.com/docs/copilot/setup

14 Zen Browser review and benchmark vs Chrome, Brave, Firefox and Safari
https://www.jitbit.com/alexblog/zen-review-benchmark/

15 Setting Up An Update Server — Firefox Source Docs documentation
https://firefox-source-docs.mozilla.org/toolkit/mozapps/update/docs/SettingUpAnUpdateServer.html