ChatGPT

Tutorial: Building a macOS Fountain Editor with GPT-5, Teatro & Prompt-Based VRT

Introduction

In this tutorial, we will **build a macOS application by conversing with GPT-5** using a specialized prompt design language called **Teatro**, and then verify its GUI using **Prompt-Bound Visual Regression Testing (PB-VRT)**. The goal is to create a *Fountain* screenplay editor app – a non-trivial, professional tool – entirely through AI-assisted development, from initial layout to automated UI testing.

Teatro Prompt Language: Teatro is a prompt scripting system developed to talk to coding AI (like GPT-5) about macOS GUI creation [1]. It acts as a *bilingual bridge* between **SwiftUI** (Apple's UI framework) and *stage/cinematic metaphors*. Instead of writing code ourselves, we'll describe the app in Teatro's terms (Stage, Light, Tempo, etc.), and let GPT-5 (Codex) generate the SwiftUI code. This allows even beginners to specify complex UIs in plain language, while providing advanced expressive control for power users.

Prompt-Bound VRT: Once our app's UI is generated, we use PB-VRT to ensure the visual output matches our intent. **Prompt-Bound Visual Regression Testing** is a framework for verifying GUI rendering fidelity by treating a textual **prompt as the source of truth** for the UI's appearance [2]. In practice, we will capture baseline screenshots of our app's UI (bound to the original prompt description) and have automated tests compare future renders against those baselines. Any deviation (or "visual drift") beyond a tolerance triggers test failures, catching UI regressions early. This approach unifies *what we describe* in prompts with *what we see* on screen, closing the loop from design to testing.

Why this approach? Prompt-driven development can dramatically speed up UI prototyping – GPT-5 handles boilerplate and SwiftUI intricacies while you focus on high-level design. The Teatro language adds structure to these prompts so they are clear and *compiler-like* in precision [3]. Meanwhile, PB-VRT ensures our AI-generated UI isn't a brittle "toy" example, but a maintainable app with quality gates. In fact, our example *Fountain Editor* app will use a full Fountain format parser and rigorous visual tests (just as done in the real FountainKit project [4] [5]).

By the end of this tutorial, you'll see how a **beginner** can get started with AI-generated SwiftUI interfaces, and how **advanced** techniques (like cinematic prompt cues and visual regression tests) can be layered on for professional-grade results. Let's start directing our AI "performer" to build an app!

Understanding Teatro and Prompt-Bound VRT

Before jumping into prompting, let's briefly break down the key concepts:

- **Teatro Prompt Language:** Think of yourself as a *director* and GPT-5 as an *actor/coder*. Teatro provides the vocabulary and structure to give directions. It mixes **technical terms** (SwiftUI components, data models) with **theatrical cues** (stage, lighting, tempo) to describe not just what the interface *is*, but how it *feels*. For example, "Stage" refers to the interface layout as a scene, “Light” refers to the visual tone (clarity, emphasis), and “Tempo” to the pacing of interactions [6]. We’ll use this language to compose a prompt that clearly conveys:

- **Intent:** What we want the app to do (e.g. *“A macOS app for editing Fountain screenplay files.”*).
- **Structure:** The UI layout and components (e.g. *“Sidebar with scenes outline, main editor area, toolbar, etc.”*).
- **Behavior:** How the app behaves (e.g. *“File opens/saves, outline syncs with editor, state persists.”*).
- **Context:** Constraints or environment (e.g. *“Target macOS 14, use SwiftUI DocumentGroup for .fountain files, follow Apple Human Interface guidelines.”*).

Why structured prompts? The Teatro Field Guide suggests organizing prompts by **Intent → Structure → Behavior → Context** for clarity [3]. This ensures GPT-5 understands the request from high-level purpose down to technical details, much like writing a scene in a play: set the scene, describe the actors and actions, and note any background conditions.

- **Prompt-Bound Visual Regression Testing:** Once GPT-5 produces the UI code and we have a working app, we need to guarantee that the UI remains true to our described intent over time. PB-VRT treats the original **prompt description as a test oracle**. For a given prompt (say, “Show the screenplay editor with the sample script loaded”), we generate a **baseline**: this includes a reference screenshot (render) of the UI in that state, and possibly other data like control input sequences or embeddings [7]. Later, when the app changes (due to code edits or evolving AI output), we run tests that:
- **Re-render the UI** for the same prompt or scenario.
- **Compare** the new rendering to the baseline using multiple probes:
    - **Image similarity (SSIM):** checks pixel-level differences [8].
    - **Embedding distance:** checks high-level visual features via ML embeddings [8].
    - *(Optional)* **Prompt-visual consistency:** checks if the UI still semantically matches the prompt (using a text/image model).
- **Alert on drift:** If differences exceed thresholds (e.g. SSIM below 98% or embedding distance too high [9]), the test fails, signaling a regression.

Under the hood, PB-VRT uses components like a **RenderProbe** to capture deterministic screenshots and possibly **MIDIProbe** to simulate user interactions for consistent state setup [10]. But you don’t need to deeply understand MIDI for this tutorial – simply know that we can automate getting the app into a known state and snapshotting it. We’ll use a testing approach (XCTest or similar) to load our prompt’s baseline and assert that the UI hasn’t drifted. For instance, a test might load a prompt descriptor and compare current vs. expected image using a helper like `DriftAnalyzer.compare`, then assert the drift score is below a small threshold [11].

With these concepts in mind, let's move on to actually building our app step by step.

Step 1: Outline the App Idea and Setup

First, define what we're building and ensure the development environment is ready:

- **App Concept:** *Fountain Editor* – a macOS application for writing and editing screenplay files in Fountain format. It should provide an outline of the script (organized by acts and scenes) and a text editor area for the screenplay content. The UI will resemble professional screenwriting software (like Slugline or Final Draft): a sidebar for navigation and a main editor that supports screenplay formatting. We’ll also include standard app features like a menu (File Open/Save, etc.), a toolbar, and maybe a status bar showing page count and other info.

- **Project Setup:** Create a new SwiftUI Lifecycle app in Xcode (macOS target). For simplicity, we can use SwiftUI's **DocumentGroup** API to manage .fountain files as documents (this gives us basic file open/save for free) [12]. Name the document type "FountainDocument" with the uniform type identifier for Fountain (if one exists, or use plain text). Ensure you have Xcode 15+ (for macOS 14 SDK) since we target macOS 14 (Sonoma) features [13].

- **Dependencies:** We plan to leverage an existing Fountain parsing utility so that the outline (acts/scenes) can be derived from the text. FountainKit's **TeatroCore.FountainParser** is an example of a full Fountain syntax parser we could use [4]. For this tutorial, you can either include a simple parser (even the minimal one from FountainKit's sources) or instruct GPT-5 to help create one if needed. Having a parser means our app can automatically break the script into a hierarchy of acts/scenes ("Outline view"), rather than treating it as plain text.

- **PB-VRT Tools:** To use Prompt-Bound VRT, ensure you have the necessary testing tools. If using FountainKit as a reference, note that it includes a spec for PB-VRT and scripts to capture baselines. In our case, we'll set up a UI test target in Xcode to integrate PB-VRT. This might involve adding any helper code (e.g. a `RenderProbe` class or simply using Xcode's snapshot APIs). We'll come back to this after the UI is working. For now, just be aware that after building the app, we'll generate baseline images and write tests.

With the concept clear and project created, let's move to the core of this tutorial: using GPT-5 with Teatro prompts to actually build the interface.

Step 2: Crafting the GPT-5 Prompt (Teatro Style)

Now we'll write a prompt to instruct GPT-5 to generate our SwiftUI UI code. A well-structured prompt is crucial. We will follow the **Intent → Structure → Behavior → Context** format recommended by the Teatro field guide [14]. Here's an example prompt for our Fountain Editor app:

```
**Intent:** Generate a macOS SwiftUI application called "Fountain Editor" for
editing Fountain screenplay files.

**Structure:** Describe a two-pane layout using `NavigationSplitView`
(sidebar + content). The sidebar is an outline of the screenplay (Acts and
Scenes), and the content area is a text editor for the screenplay text.
Include a toolbar with common actions (e.g. open file, save, toggle view
modes) and a status bar showing information like page count, word count, etc.
Use a `DocumentGroup` or similar to support opening and saving `.fountain`
files. The window should have a standard title bar with the app name.

**Behavior:** The sidebar outline should reflect the structure of the loaded
Fountain document. Use a model object (ObservableObject) that parses the
`.fountain` text into an outline (Acts → Scenes) whenever the text changes.
Selecting an outline item scrolls/jumps to that scene in the editor. The
editor view should allow typing and basic formatting (Courier font, proper
spacing as per screenplay format). When the document is edited, mark the
document as dirty and enable Save. Use `AppStorage` or preferences to
remember user settings (e.g. dark mode or font size). Include menu commands
for File > New, Open…, Save, and a Preferences window for any app settings.
```

```
**Context:** Target **macOS 14** using SwiftUI. Follow Apple Human Interface
Guidelines for macOS (e.g. use standard sidebar list style). Ensure the app
is sandboxed (for file access, use security-scoped bookmarks if needed). You
can use `TeatroCore.FountainParser` to parse the Fountain file into outline
data (assume it's available as a Swift package). The code should be
structured clearly with separate SwiftUI views for the sidebar and editor,
and use MVVM principles (e.g. a ViewModel for the document). No third-party
UI libraries, only SwiftUI and Combine.
```

Let's break down what we did here:

- We explicitly stated the **app's purpose and name** (Intent), so GPT-5 knows this is a SwiftUI app for editing screenplays.
- Under **Structure**, we listed the major UI components and how they fit together (split view, sidebar outline, editor, toolbar, status bar). We even named a specific SwiftUI container (`NavigationSplitView`) to guide the model toward using the right layout component [15]. This is like describing the stage set and props in a play.
- For **Behavior**, we described the functional requirements in plain terms: syncing outline selection with text, updating a model on text changes, save functionality, etc. We mention relevant SwiftUI mechanisms (ObservableObject, AppStorage) so GPT-5 knows how to implement state and persistence.
- Under **Context**, we gave environmental details and constraints: OS version, guidelines, sandboxing, and even the availability of our FountainParser. By saying "assume it's available", GPT-5 might simply call it rather than trying to write a parser from scratch, which keeps the code focused. We also nudge on architecture (MVVM pattern), which helps in structuring the output.

This prompt is quite detailed, which is good – GPT models perform better when the instructions are specific and structured. Notice we used **bold headers** for each section (Intent, Structure, etc.) to make it very clear; this isn't strictly necessary, but it can help format the query. The language is **clear and declarative** (almost like speaking to a junior developer), which is exactly how Teatro encourages us to speak to Codex [16].

Sending the prompt: In your GPT-5 interface (ChatGPT or API), send the above prompt. It might be useful to prepend something like *"You are an expert SwiftUI developer. Please follow the requirements below."* to reinforce the role. However, given the detail we have, it may not be needed.

Step 3: From Prompt to SwiftUI Code – Iteration with GPT-5

GPT-5 should respond with a proposal – likely a mix of explanations and code. For example, it might output a SwiftUI `App` struct, a `Document` struct for the Fountain file, and SwiftUI `View` structs for the sidebar and editor. Review the output carefully, keeping an eye on whether it meets our described intent:

- **Check the Structure:** Did it create a `NavigationSplitView` (or similar) with two columns? Is the sidebar a list of acts/scenes? If the output is missing the outline logic, we may need to prompt further. For instance, if GPT-5 didn't implement parsing of the fountain text into outline items, we might follow up with: *"Great, can we add functionality to parse the Fountain text and populate the sidebar outline (Acts and Scenes)? Assume `FountainParser` can break the text into a list of acts each containing scenes."*

- **Review the Code:** Ensure the code compiles (you can copy it into Xcode to test). GPT-5 might occasionally use slightly incorrect API calls or omit some minor details. Common issues to look for:

- Document handling: If using `DocumentGroup`, check that the model conforms to `FileDocument` properly (reading/writing the text).
- ObservableObject usage: Did it mark the model class with `@ObservableObject` and properties with `@Published`? This is needed for the UI to update.
- Outline view selection: Is it using something like `List(selection: $selection)` to bind the selection? And does it scroll the editor to the selected scene? (This might be complex; if not present, we can implement a basic version or skip detailed scrolling).
- Toolbar and menus: GPT might include a Toolbar in SwiftUI, but menu commands may require the old AppKit menu integration. If the output lacks menus, we can manually add a basic `Commands` section to the App struct or ask GPT to add menu commands.

If GPT-5's first attempt isn't perfect, don't worry – *this is where the iterative, conversational aspect shines*. As the "director," use **Teatro meta-commands** to refine the performance [17]. For example: - If the layout is off or missing elements, you might say: *"Focus on stage – ensure the sidebar and editor are correctly laid out and visible, and add a status bar at the bottom."* The phrase "focus on stage" is a cue to concentrate on the UI structure. - If the code is too high-level and skipped important details, you can *shift to a technical register*: e.g., *"Shift to technical – include the actual implementation of the FountainParser to get acts and scenes from text."* This tells GPT to get more low-level in the response. - If output is verbose or not focused, you might say *"compress timeline – provide just the updated code for the Document and Outline view"* to get a concise answer.

These are inspired by the conversational protocol in Teatro (where you iterate as Director and Codex as Performer) [18] [19]. In practice, you don't need to use the exact phrasing, but the concept is: **iterate with targeted instructions**. GPT-5 will remember the context from the previous answer, so you can ask it to modify or extend the code rather than rewriting everything.

Continue this process until you have a working implementation. For our Fountain Editor, an ideal result would be: - The app runs, shows a window with a sidebar (currently probably empty or with a placeholder until a file is loaded) and an editor area. - You can create or open a `.fountain` file (DocumentGroup handles the file dialogs if set up). - When a file is loaded, the text appears in the editor, and the sidebar populates with a list of acts/scenes parsed from the text. *(You might need to add a sample Fountain file in your project for testing.)* - Basic editing works (you can type in the editor and save). - The toolbar and status bar show expected info (for example, page count might be a stretch goal—GPUs likely won't calculate screenplay pages accurately, but word count or character count is doable).

Tip: If GPT-5 is unsure about Fountain parsing specifics, you could provide it a brief example of Fountain formatted text and the expected outline. For instance: *"Example Fountain text:* `TITLE: My Script\n**** ACT 1 ****\nINT. HOUSE - DAY\nSome action...\nCHARACTER\nDialogue...\n**** ACT 2 **** ...` *— this should result in outline entries Act 1 and Act 2, each containing respective scenes."* This guidance can help it implement or use the parser correctly.

At this stage, we assume we have a functioning **Fountain Editor app** codebase. Save this code, run the app, and manually verify the UI looks as intended. Once satisfied, commit this baseline version – it's time to set up prompt-bound visual tests to lock in the UI's fidelity.

Step 4: Creating a Baseline (Prompt and Snapshot) for Visual Testing

With the app running correctly, define the scenarios (states) you want to guard with visual regression tests. Typically, you'd cover the critical screens or states of your UI. In our case, a primary state might be **the main editor window with a sample screenplay loaded**.

We'll create a **prompt descriptor** for this state, which is basically a text (and an identifier) describing what the UI should show. For example, a prompt YAML (as per PB-VRT spec) could be:

```
id: "main-editor-default"
text: "Display the Fountain Editor main window with a screenplay loaded. The
sidebar shows the outline of acts and scenes, and the editor is filled with
the screenplay text. One scene is selected in the sidebar."
tags: ["FountainEditor", "UI", "baseline"]
modality: "visual"
```

This descriptive text is our semantic truth for the UI. Now we need to capture the actual **baseline image** that corresponds to this description: - Open a sample screenplay in the app (one with multiple acts/scenes so the outline is populated). - Arrange the window (if needed) so that it's in a deterministic state (e.g., default size). - Take a screenshot of the application window. Name it something like `main-editor-default.png`.

In a more automated setup, you might have a script or test code generate this screenshot (for example, FountainKit's PB-VRT tools can launch the app and capture a PNG for a given prompt). But doing it manually the first time is fine. Place this image in your test bundle (e.g., in an `ReferenceImages` folder).

Additionally, PB-VRT can record other artifacts like **MIDI input sequences** (to simulate user actions) or **embedding vectors** for the image [20]. For a simple static UI, these might not be needed, but keep in mind that if your UI requires a series of actions to reach a state, you'd want to capture those interactions too. In our case, just opening a document is the main interaction, which we can handle with a simple open-file command in the test.

Now we have: - A prompt descriptor (`main-editor-default`) with a textual description of the expected UI. - A baseline render image (`main-editor-default.png`) showing the UI in that state.

Next, let's write a test that uses these.

Step 5: Writing Prompt-Bound Visual Regression Tests

We will add a new UI test case (XCTestCase subclass) to our project, say `VisualRegressionTests`. In this test, we'll compare the current UI against our baselines using the PB-VRT approach:

1. **Launch the app and load the test screenplay** – In UI tests, you can use `XCUIApplication()` to launch. We might need to ensure it opens our sample document. One approach is to include the sample screenplay in the test bundle and have the app load it via a launch argument or a special UI test mode. To keep it simple, you could also automate the menu: e.g., use

`app.menuItems["Open…"]` to trigger an open dialog, though scripting file dialogs is tricky. Alternatively, modify the app for test builds to auto-load a “Sample.fountain” file if present.

2. **Capture current UI** – Once the app is in the desired state (sample file open), capture a screenshot. In XCUITest, `XCUIScreen.main.screenshot()` or `XCUIApplication().windows.first.screenshot()` can get the current UI image. Save it or keep it in memory for comparison.

3. **Compare with baseline** – We can compare images directly with an algorithm. If using PB-VRT libraries, they might provide an API. For illustration, let's assume we have a function `compareImages(base: URL, current: XCUIScreenshot) -> Double` that returns a “difference score” (lower is better). Or better, use the PB-VRT metric definitions:

4. Compute SSIM (structural similarity). You might find or write a small function for this, or use Core Image/CoreGraphics to diff images.
5. Additionally, for a robust test, compute an embedding difference: Apple’s Vision framework can produce a feature vector for each image and you calculate cosine distance [21] [22]. However, that’s advanced – feel free to skip if not readily available.

6. If any difference is beyond threshold, record a failure.

7. **Assertion** – Finally, assert that the differences are within allowed range. For example, using a made-up `drift` value, we expect it to be very low:

```
let baselineImage = ... // load baseline PNG from bundle
let currentImage = XCUIScreen.main.screenshot().image
let drift = compareImages(base: baselineImage, current: currentImage)
XCTAssertLessThan(drift, 0.01, "UI has visually regressed from
baseline")
```

   In a real PB-VRT setup, `drift` might be a composite of metrics as defined in the spec (weighted sum of 1-SSIM, embedding diff, etc.) [23]. For example, FountainKit’s spec suggests failing if SSIM < 0.98 or embedding cosine similarity drops too much [9]. The exact numbers depend on how strict you want to be.

If implementing from scratch seems daunting, note that the concept is what matters: you are effectively doing an **automated screenshot comparison**. Even a basic pixel-by-pixel diff with a tolerance can catch unintended changes. The key improvement with PB-VRT is using semantic comparisons (like embeddings and prompt text) to be more robust to minor acceptable changes while catching meaningful ones.

For instance, here’s a pseudo-code snippet inspired by the PB-VRT example stub [11]:

```
func testMainEditorVisualFidelity() {
    // Launch app with test document...
    let app = XCUIApplication()
    app.launchArguments = ["UI-TEST-AUTO-OPEN-SAMPLE"]  // (If we coded app
to open a sample on this flag)
    app.launch()
    // Assume app opened the sample screenplay; take screenshot after a brief
delay
    sleep(1) // wait for UI to settle
    let currentImage = XCUIScreen.main.screenshot().image

    // Load baseline image from test resources
    let baselineURL = Bundle(for: type(of: self)).url(forResource: "main-
editor-default", withExtension: "png")!
    let baselineImage = NSImage(contentsOf:
baselineURL)!.cgImage(forProposedRect: nil, context: nil, hints: nil)!

    // Compare images (using SSIM or similar)
    let ssimValue = computeSSIM(between: baselineImage, and:
currentImage.cgImage!)
    XCTAssertGreaterThan(ssimValue, 0.98, "Structural similarity fell below
threshold – possible UI regression.")

    // (Optionally, compute embedding distance)
    // let Δembed = computeCosineDistance(featureVector(of: baselineImage),
featureVector(of: currentImage))
    // XCTAssertLessThan(Δembed, 0.05, "Visual embedding drift exceeds
tolerance.")
}
```

The above test would fail if the UI diverged significantly – for example, if a future change accidentally removed the sidebar or changed the layout. In our scenario, since we just generated the UI, the test should pass now, establishing a **baseline confidence**. Going forward, if we or GPT-5 modify the UI, these tests give a safety net.

Integrating into CI: If you use a CI system (GitHub Actions, etc.), you can have it run these tests on each commit. The PB-VRT spec even suggests running `swift test --filter VisualRegressionTests` as part of the pipeline [24], and blocking merges if any drift is detected. This is exactly how we ensure our AI-designed UI remains consistent with the original design intent [5].

Conclusion

We have successfully orchestrated the creation of a macOS app using AI and ensured its quality with novel testing techniques:

- We used **GPT-5 with the Teatro prompt language** to go from a blank project to a functional **Fountain Editor** app, covering both beginner-friendly guidance (high-level intent and SwiftUI component hints) and advanced details (parsing logic, state management).
- We treated the development process as a *conversation*, iteratively refining the output just like a director working with an assistant – leveraging the structured approach (Intent, Structure, Behavior, Context) and even meta-commands to steer the AI's focus.
- We set up **Prompt-Bound Visual Regression Tests** to lock in our UI's design. The prompt that defined our UI isn't just documentation – it became a test oracle. By coupling semantic descriptions with pixel-level and feature-level comparisons, we achieved a testing rigor normally reserved for hand-crafted UIs, now applied to AI-generated code.

This workflow demonstrates that AI-generated interfaces need not be throwaway prototypes or “toys.” With the right prompting discipline and testing guardrails, they can be part of a serious development lifecycle. Our Fountain Editor app, backed by a full Fountain parser and continuous visual testing, is a testament to that.

Next Steps: You can extend this approach to other platforms (e.g., iOS) or more complex apps. The Teatro language and methodology scales from beginner prompts to advanced multi-turn design sessions – for further learning, explore the *Teatro Codex macOS Prompt Field Guide* which covers topics like Stage dynamics, Cinematography cues, and Montage (for multi-scene flows) in depth [25]. On the testing side, consider integrating more of the PB-VRT pipeline (like automated input sequences with MIDIProbe for interactive UI flows).

By combining creative prompt-based design with strict regression testing, we unlock the best of both worlds: rapid **AI-assisted development** and **robust UI quality**. Happy prompting, and enjoy your new app!

---

[1] [25] README.md
https://github.com/Fountain-Coach/teatro-codex-macos-prompt-field-guide/blob/4b59ca5c26f2ef73c1f30cf2cfd6e55709133d49/README.md

[2] [7] [8] [9] [10] [11] [20] [21] [22] [23] [24] prompt-bound-visual-regression-spec.md
https://github.com/Fountain-Coach/FountainKit/blob/1589ecb89d1a9384d1a06776487899957b972e36/Public/pb-vrt-spec-and-openapi/prompt-bound-visual-regression-spec.md

[3] [13] [14] [15] [16] part-01.md
https://github.com/Fountain-Coach/teatro-codex-macos-prompt-field-guide/blob/4b59ca5c26f2ef73c1f30cf2cfd6e55709133d49/parts/part-01.md

[4] [5] main.swift
https://github.com/Fountain-Coach/FountainKit/blob/1589ecb89d1a9384d1a06776487899957b972e36/Packages/FountainApps/Sources/fountain-editor-seed/main.swift

[6] GLOSSARY.md
https://github.com/Fountain-Coach/teatro-codex-macos-prompt-field-guide/blob/4b59ca5c26f2ef73c1f30cf2cfd6e55709133d49/GLOSSARY.md

[12] CODE-POINTERS.md
https://github.com/Fountain-Coach/teatro-codex-macos-prompt-field-guide/blob/4b59ca5c26f2ef73c1f30cf2cfd6e55709133d49/CODE-POINTERS.md

[17] [18] [19] part-06.md
https://github.com/Fountain-Coach/teatro-codex-macos-prompt-field-guide/blob/4b59ca5c26f2ef73c1f30cf2cfd6e55709133d49/parts/part-06.md