Learning Executable WebUI Generation across Front End Frameworks with MLLMs

Anonymous Authors
Anonymous Institution
anonymous@anonymous.edu

Abstract

Multimodal large language models have recently shown strong potential for translating WebUI screenshots into code, yet most existing studies focus on single-target settings such as HTML and under-emphasize a more practical challenge: generating executable code across multiple front end frameworks. Compared with vanilla HTML, framework-based generation must satisfy not only visual fidelity, but also framework-specific syntax, component organization, import and dependency consistency, and compilation constraints. As a result, direct transfer of screenshot-to-code pipelines to React, Vue, and Angular often leads to severe drops in executability. To address this problem, we present a phase-wise adaptation framework for executable multi-framework WebUI generation. Our method combines a shared multimodal backbone with framework-aware parameter-efficient adapters and organizes training into phases according to estimated cross-framework compatibility. The shared component learns transferable UI abstractions from screenshots, layout cues, and code structure, while framework-specific modules capture compilation-critical patterns such as component declarations, template binding rules, and project-level conventions. We further formulate phase construction as a compatibility-aware partitioning problem that balances positive transfer against framework heterogeneity. The paper provides a complete training objective, a theoretical view of why staged optimization reduces gradient conflict, and an executable evaluation protocol on recent WebUI-to-code benchmarks. The resulting draft is designed as a submission-ready NeurIPS-style paper and can be directly completed with empirical results from actual runs.

1 Introduction

Recent multimodal large language models, or MLLMs, have substantially improved the automation of interface engineering from visual inputs, making it increasingly plausible to translate screenshots or sketches into executable front-end code [1, 2, 3, 4, 5]. This trend has created a new research line that treats a webpage screenshot as a multimodal specification and asks a model to synthesize the corresponding implementation. Early work such as pix2code established the feasibility of screenshot-to-code generation in simplified environments [6], while later datasets and benchmarks progressively moved toward realistic web screenshots, richer HTML, and larger-scale supervision [7, 8, 9, 10]. More recent systems further explore divide-and-conquer generation and hierarchical planning for long and structured front-end outputs [11, 12].

Despite this progress, most prior work still centers on HTML or HTML/CSS as the final artifact [8, 7, 9]. This focus is understandable because HTML is the lowest-friction representation for benchmarking visual similarity. However, it does not match modern front-end practice, where developers rarely stop at static HTML and instead use component-based ecosystems such as React, Vue, and Angular. The move from HTML to framework-specific code changes the problem qualitatively rather than merely incrementally. The generator must now produce code that is not only visually aligned with the screenshot, but also executable under framework-dependent compilation rules, file conventions, module imports, binding syntax, and component boundaries. DesignBench makes this gap explicit and shows that current MLLMs experience large performance variations across frameworks and tasks [13]. WebUIBench likewise argues that outcome-only evaluation hides important deficiencies in perception, structural understanding, and code synthesis [14].

A naive response is to jointly fine-tune a single model on data from all frameworks. In practice, this is unstable. Frameworks share high-level visual abstractions such as layout hierarchy, repeated components, and text-image grouping, yet differ sharply in compilation-critical realization. React centers JSX composition and import discipline, Vue intertwines template and directive semantics, and Angular introduces stricter module metadata and binding constraints. Joint optimization therefore mixes beneficial transfer with destructive interference. A second naive response is to train one model per framework. This improves specialization but sacrifices shared learning, multiplies storage cost, and weakens generalization to underrepresented frameworks. We seek a middle ground.

This paper proposes *phase-wise MLLM adaptation* for executable multi-framework WebUI generation. The core idea is to freeze the backbone and introduce a shared adapter together with lightweight framework-specific adapters, following the parameter-efficient tuning literature [15, 16, 17, 18, 19]. Rather than training all framework data simultaneously, we estimate compatibility among frameworks and organize them into training phases inspired by curriculum learning [20, 21]. Easy or mutually aligned framework groups are learned

first to build reusable multimodal abstractions, after which more heterogeneous frameworks are introduced with stronger specialization. This retains transfer on common UI concepts while reducing cross-framework gradient conflict.

The paper makes four contributions. First, we formalize executable multi-framework WebUI generation as a compatibility-sensitive multimodal adaptation problem. Second, we introduce a shared-plus-specific adapter design that decouples transferable UI abstraction from framework-critical compilation behavior. Third, we construct a phase-wise training strategy based on compatibility estimation over framework gradients and heterogeneity statistics. Fourth, we provide a full experimental protocol with ready-to-fill result tables for DesignBench and related datasets, enabling direct completion after actual runs. Although the current draft intentionally avoids fabricating numerical outcomes, all sections are written in final-paper form and are immediately reusable for a submission after experiments are filled in.

2 Related Work

WebUI-to-code generation. Screenshot-to-code generation began with simplified synthetic settings, most notably pix2code [6], and then expanded toward sketches, webpage screenshots, and more realistic implementations [22, 23, 24]. Recent work has benefited from stronger multimodal backbones and improved datasets. WebSight introduces large synthetic screenshot-HTML supervision [7]. Design2Code benchmarks real webpages for automated front-end engineering [8]. Web2Code augments screenshot-code generation with webpage understanding signals [9]. WebCode2M and VISION2UI further emphasize large-scale real-world data for design-to-code learning [10, 25]. Methods such as DCGen and UICopilot show that structural decomposition can help long-code synthesis [11, 12]. Our work differs from this line by explicitly targeting *cross-framework executability* instead of screenshot-to-HTML alone.

Multi-framework front-end evaluation. The recent emergence of multi-framework benchmarks reveals a major blind spot in the literature. DesignBench evaluates generation, edit, and repair across HTML/CSS, React, Vue, and Angular, and highlights framework-specific bottlenecks [13]. WebUIBench decomposes WebUI ability into perception, programming, understanding, and code generation, showing that headline visual outcomes alone are insufficient [14]. Our paper is aligned with these findings and proposes a training method tailored to the multi-framework setting those benchmarks expose.

Multimodal foundations for visually grounded code generation. Strong general-purpose MLLMs and vision-language models form the substrate for recent design-to-code systems, including GPT-4 [1], LLaVA [3, 4], Qwen2-VL [5], Flamingo [26], BLIP-2 [2], CLIP [27], and Pix2Struct [28]. In parallel, code-specialized LLMs such as InCoder, StarCoder, and Code Llama provide strong priors for long-form code generation and infilling [29, 30, 31]. We build on these developments but focus on how to adapt them for framework-conditioned executability.

Parameter-efficient adaptation and curricula. Parameter-efficient transfer methods, including adapters, prefix tuning, LoRA, Compacter, and AdaLoRA, provide a practical mechanism for specializing large backbones without full fine-tuning [15, 16, 17, 18, 19]. Curriculum and staged learning methods suggest that optimization order matters, especially when tasks differ in difficulty or compatibility [20, 21]. We integrate these ideas by using compatibility-aware staged adaptation for framework-specific WebUI code generation.

GUI and web agents. A parallel literature studies visually grounded interaction with user interfaces, including Mind2Web, SeeAct, UGround, WebCanvas, and ScreenSpot-Pro [32, 33, 34, 35, 36]. These works are not direct screenshot-to-code generators, but they clarify an important point: screenshots contain rich structural information that can be learned visually, and visually grounded systems often outperform brittle HTML-only abstractions in dynamic environments. This reinforces our choice to retain screenshot-grounded learning rather than reducing the task to text-only code modeling.

3 Problem Formulation

Let $\mathcal{F} = \{1, \dots, K\}$ denote a set of front-end frameworks. In this paper, the primary setting is $K = 3$ for React, Vue, and Angular, though the formulation naturally extends to HTML/CSS as an auxiliary domain. For each framework k, we assume a dataset

$$\mathcal{D}_k = \{(x_i, y_i^{(k)}, m_i^{(k)})\}_{i=1}^{N_k}, \tag{1}$$

where x_i is a WebUI screenshot, $y_i^{(k)}$ is the corresponding code artifact in framework k, and $m_i^{(k)}$ denotes optional metadata such as file structure, compilation configuration, or project scaffold. The goal is to learn a conditional model

$$p_\theta(y^{(k)} \mid x, k, m^{(k)}) \tag{2}$$

that produces code which is both visually faithful to x and executable under framework k.

The key difficulty is that supervision is *partially shared*. Layout, typography, grouping, and repeated components are visually observable and often framework-invariant. By contrast, the concrete syntax and execution semantics are framework-specific. Hence the optimization objective is neither a standard multi-task problem nor a simple domain adaptation setting. Instead, it must preserve cross-framework reusable abstractions while isolating compilation-critical behavior.

We therefore distinguish two forms of quality. First, *visual faithfulness* measures whether the rendered output preserves the appearance and structure of the screenshot. Second, *executability* measures whether the generated artifact parses,

compiles, and runs under the target framework. The latter is the primary target of this paper. Let $\mathrm{Exec}(\hat{y}, k) \in \{0, 1\}$ indicate compilation success under the build rules of framework k. We seek models that maximize

$$\mathbb{E}_{(x,y^{(k)})\sim\mathcal{D}_k}[\mathrm{Exec}(\hat{y}, k)] \tag{3}$$

without materially degrading screenshot fidelity.

4 Method

4.1 Framework-aware parameter-efficient adaptation

We start from a frozen multimodal backbone Θ that maps screenshot tokens and textual prompts into a code-generating latent space. Building on parameter-efficient tuning [15, 17, 19], we introduce a shared adapter w^s and framework-specific adapters $\{w_k\}_{k=1}^{K}$. The resulting model is

$$\mathcal{M}(x, k) = \mathrm{Decode}(x; \Theta, w^s, w_k). \tag{4}$$

The shared adapter is intended to encode transferable UI abstractions, while w_k captures framework-specific conventions. In implementation, both w^s and w_k can be instantiated as LoRA modules inserted into the language blocks and optionally the vision-language connector. We keep the design modular so that adapters can be replaced by prefix tuning or other PEFT methods without changing the training logic.

To improve screenshot grounding, we concatenate the screenshot with a framework token and a concise structural prompt. The prompt specifies the target framework, output format, and compile-aware constraints, for example whether the model should produce a single component file, multiple files, or a scaffold-consistent entry point. This follows the observation from both visually situated models and code LLMs that conditioning on output format substantially reduces invalid generations [28, 29, 31].

4.2 Compatibility-aware phase construction

The central question is how to schedule framework exposure. Joint training on all frameworks risks interference because optimization steps that improve one framework may worsen another. To quantify this effect, we estimate framework compatibility using gradients on the shared adapter. For each framework k, define an averaged warm-start gradient

$$g_k = \mathbb{E}_{(x,y)\sim\tilde{\mathcal{D}}_k}\left[\nabla_{w^s}\mathcal{L}_{\mathrm{tok}}(x, y; k)\right], \tag{5}$$

where $\tilde{\mathcal{D}}_k$ is a small calibration subset and $\mathcal{L}_{\mathrm{tok}}$ is the token-level negative log-likelihood. We then define pairwise compatibility as

$$c(i, j) = \frac{1 + \cos(g_i, g_j)}{2} \in [0, 1]. \tag{6}$$

High compatibility indicates that the two frameworks induce aligned updates on shared representations and are likely to benefit from early joint learning.

Compatibility alone is insufficient because some frameworks can have similar gradients at warm start yet differ strongly in code realization. We therefore introduce a heterogeneity score $h(i, j)$ computed from framework-level syntax and error statistics, such as Jensen-Shannon divergence between code token distributions, import patterns, AST node frequencies, or compile error categories. We then partition frameworks into phases $\Pi = \{P_1, \ldots, P_T\}$ by maximizing

$$G(\Pi) = \sum_{t=1}^{T}\left[\frac{1}{|P_t|^2}\sum_{i,j\in P_t} c(i, j) - \lambda\frac{1}{|P_t|^2}\sum_{i,j\in P_t} h(i, j)\right], \tag{7}$$

subject to a size or curriculum constraint that earlier phases should be smaller or easier. Intuitively, each phase should contain frameworks that transfer well to each other without collapsing under heterogeneity.

4.3 Phase-wise optimization

Given phases Π, training proceeds sequentially. In phase t, only data from frameworks in P_t are sampled. The objective is

$$\mathcal{L}^{(t)} = \mathcal{L}_{\mathrm{tok}} + \alpha\mathcal{L}_{\mathrm{struct}} + \beta\mathcal{L}_{\mathrm{contract}} + \gamma\mathcal{L}_{\mathrm{reg}}. \tag{8}$$

The token loss is standard autoregressive likelihood. The structure loss encourages the model to preserve coarse UI organization, for example by predicting a lightweight intermediate representation of containers, repeated blocks, or component boundaries before full code generation. This is motivated by hierarchical and divide-and-conquer methods for long UI code [11, 12]. The contract loss penalizes violations of framework-specific necessary conditions, such as missing imports, unmatched template bindings, absent export declarations, or illegal metadata links. It serves as a low-cost approximation to build-time executability. Finally, the regularizer limits drift in the shared adapter when new frameworks are introduced:

$$\mathcal{L}_{\mathrm{reg}} = \|w^s - w^s_{\mathrm{prev}}\|_2^2. \tag{9}$$

After each phase, the shared adapter is retained, while framework-specific adapters for already-seen frameworks remain frozen or updated with a smaller learning rate. This reduces catastrophic overwriting. In effect, the model first internalizes framework-agnostic screenshot-to-structure mappings and only later absorbs framework-specific realization rules.

Practical phase schedule. A concrete training recipe is as follows. We first run a short screenshot-to-code warm-up on HTML-dominant data such as WebSight, Design2Code, and Web2Code to stabilize visual grounding and long-code decoding [7, 8, 9]. We then estimate compatibility on the target multi-framework benchmark and build phases using the objective in Eq. (8). In each phase, mini-batches are balanced across active frameworks to avoid domination by the largest split, while inactive framework-specific adapters remain frozen. This recipe is deliberately simple and avoids

introducing additional agentic modules, retrieval systems, or expensive compiler-in-the-loop exploration.

Compile-aware contract loss. The contract loss is implemented as a set of lightweight executable proxies that can be computed without running a full build for every update. For React, the proxy can include import completeness, JSX closure, component export existence, and invalid nesting checks. For Vue, it can include template well-formedness, directive syntax, and missing script-template interface variables. For Angular, it can include decorator presence, metadata consistency, and common binding failures. Let $\phi_k(\hat{y}) \in \{0,1\}^{M_k}$ be a vector of framework-specific validity indicators and $\phi_k(y)$ the corresponding oracle extracted from reference code or static analyzers. We define

$$\mathcal{L}_{\text{contract}} = \sum_k \mathbf{1}[f = k] \cdot \text{BCE}(\phi_k(\hat{y}), \phi_k(y)). \tag{10}$$

Although coarse, this term nudges generation away from known failure modes that dominate CSR in practical settings.

Complexity. Let P denote the number of trainable adapter parameters. Compared with full fine-tuning, the optimization cost remains $O(P)$ in trainable memory and approximately linear in active framework count per phase. Compatibility estimation is cheap because it only requires one warm-start gradient pass on a small calibration subset. As a result, the method preserves the efficiency advantage of PEFT while adding only a modest scheduling overhead.

Proposition 1. *Assume that the first-order optimization difficulty of jointly training a phase P is upper bounded by*

$$\Delta(P) \leq \sum_{k \in P} a_k + \eta \sum_{i<j,\, i,j \in P} (1 - c(i,j)) b_{ij}, \tag{11}$$

where a_k captures within-framework difficulty and $b_{ij} \geq 0$ measures the magnitude of conflicting updates. Then, for fixed $\{a_k\}$ and $\{b_{ij}\}$, any partition Π that maximizes average within-phase compatibility minimizes the upper bound on cumulative conflict among all partitions with the same phase sizes.

Proof sketch. The first term is invariant to the partition. Therefore minimizing cumulative upper bound reduces to minimizing the pairwise conflict term

$$\sum_{t=1}^{T} \sum_{i<j,\, i,j \in P_t} (1 - c(i,j)) b_{ij}. \tag{12}$$

For fixed non-negative weights b_{ij} and fixed phase cardinalities, this is equivalent to maximizing the weighted sum of within-phase compatibilities. Our construction $G(\Pi)$ augments this criterion with a heterogeneity penalty, which further separates frameworks that may have deceptively aligned warm-start gradients but divergent compile behavior. □

Although simple, the proposition clarifies the role of phase construction. The gain does not come merely from seeing easier data first, but from reducing interference exactly where shared adaptation is supposed to happen.

Phase-wise training procedure
1. Warm up the shared adapter on HTML-centric screenshot-code pairs.
2. Compute warm-start shared-adapter gradients for each framework.
3. Estimate pairwise compatibility and heterogeneity.
4. Partition frameworks into phases by maximizing $G(\Pi)$.
5. Train phase by phase with active frameworks only.
6. Decay shared learning rate and preserve old specific adapters.
7. At test time, decode with the shared adapter plus the target framework adapter.

Figure 1: A compact summary of the proposed training algorithm.

4.4 Inference

At inference time, the model receives a screenshot, a target framework token, and an output contract. We perform one-pass generation by default. Optionally, a single lightweight refinement pass can be invoked by feeding back parser or compiler errors, following the spirit of iterative code improvement and self-refinement [29, 37]. The training method itself remains purely offline and does not rely on expensive reinforcement learning or repeated build-time search.

5 Experimental Protocol

5.1 Datasets

The primary benchmark is DesignBench, which directly evaluates front-end generation across HTML/CSS, React, Vue, and Angular and therefore matches our task setting [13]. We recommend reporting results on its generation task first, followed by edit and repair settings if available. As auxiliary data for warm-up or screenshot-structure alignment, we use Design2Code, WebSight, Web2Code, WebCode2M, and VISION2UI [8, 7, 9, 10, 25]. These sources provide large-scale screenshot-code pairs but are predominantly HTML-oriented, making them suitable for pre-adaptation of visual and structural representations before multi-framework specialization.

5.2 Baselines

The following baselines should be included.

1. **Zero-shot prompting**: a frozen general-purpose MLLM prompted separately for React, Vue, and Angular.

2. **Joint PEFT**: one shared LoRA trained jointly on all frameworks.

3. **Per-framework PEFT**: one independent adapter per framework without sharing.

4. **Random curriculum**: the same number of phases as our method but with random grouping.

5. **Difficulty-only curriculum**: phases ordered by approximate compile success or sequence length without compatibility estimation.

6. **Hierarchical generators**: UICopilot or DCGen style decomposition adapted to multi-framework outputs where possible [11, 12].

5.3 Metrics

Executability should be the primary metric. We recommend compilation success rate, abbreviated CSR, under official framework build commands. For projects containing multiple files or package metadata, a stricter end-to-end build success metric is also useful. To ensure executable gains are not purchased by collapsing to simplistic layouts, visual similarity should be reported using the rendering-based protocol of prior WebUI-to-code work [8, 9, 13]. A practical set includes CLIP similarity, block alignment, and text consistency. When possible, error types should be grouped into parsing, import resolution, template or binding failure, missing component export, and styling or asset issues, since the multi-framework setting is defined not only by whether code fails, but by how it fails.

5.4 Implementation details

The default implementation uses a frozen open MLLM backbone with shared and framework-specific LoRA modules. Warm-start compatibility is estimated on a fixed calibration subset. The number of phases can be chosen by simple grid search over $T \in \{2, 3, 4\}$ under validation CSR. Learning rates for the shared adapter should decay across phases, while framework-specific adapters can use phase-local tuning. Auxiliary HTML-centric data can be used in a brief warm-up stage before the first phase, but all final comparisons should control for total training budget.

5.5 Recommended reporting protocol

To make the paper convincing to reviewers, we recommend a three-level report. The first level presents average CSR and visual metrics across frameworks, which answers whether the method works overall. The second level presents per-framework numbers and error categories, which answers where the gains come from. The third level presents low-resource and cross-backbone studies, which answers whether the approach is robust rather than overfit to one benchmark or one model family. This reporting structure is especially important in multi-framework settings because average numbers alone can hide whether a method simply helps one framework while hurting another.

A strong final version should additionally report the following diagnostics. First, *within-phase versus cross-phase transfer*: compare validation loss changes of old frameworks after introducing a new phase. Second, *compatibility-quality correlation*: verify whether estimated compatibility scores predict the observed benefit of joint early training. Third, *adapter utilization*: measure whether later phases place more weight on framework-specific modules, which would empirically support the specialization story. These analyses can be implemented without changing the core method and often provide the clearest evidence for the paper's central claim.

Method	React	Vue	Angular	Avg.
Zero-shot MLLM	–	–	–	–
Joint PEFT	–	–	–	–
Per-framework PEFT	–	–	–	–
Random curriculum	–	–	–	–
Difficulty-only curriculum	–	–	–	–
Phase-wise adaptation	**–**	**–**	**–**	**–**

Table 1: Compilation success rate on DesignBench generation. Fill with actual CSR values from final runs.

Method	CLIP↑	Block↑	Text↑
Joint PEFT	–	–	–
Per-framework PEFT	–	–	–
Phase-wise adaptation	**–**	**–**	**–**

Table 2: Visual fidelity metrics. The desired outcome is that executability gains do not materially harm screenshot faithfulness.

6 Result Tables and Analysis Templates

This section is intentionally written as a direct insertion point for actual numbers. The table layouts below are structured to support the most important claims of the paper without fabricating results.

Table 1 should be used to validate the core hypothesis: phase-wise adaptation improves executability across frameworks relative to both joint sharing and complete separation. If the method works as intended, the largest gains should appear on the most compilation-sensitive frameworks, while the average should outperform all baselines under matched parameter budgets.

Table 2 supports a second claim: better executability should not come at the cost of visibly worse webpages. A positive outcome is not necessarily winning every visual metric, but remaining competitive while improving build success.

The ablation in Table 3 isolates whether gains come from staged training itself, from the shared-specific decomposition, or from their combination. In many plausible outcomes, shared-plus-specific adapters with no phases may already outperform purely joint or purely independent tuning, while compatibility-aware scheduling supplies the final improvement.

Figure 2 should show the estimated compatibility matrix and the resulting phase assignment. A persuasive visualization typically reveals that some frameworks share strong representation-level alignment while others are better deferred

Variant	Shared	Specific	Phases	Avg. CSR
Joint LoRA	✓	✗	1	–
Independent LoRA	✗	✓	1	–
Shared + specific	✓	✓	1	–
Shared + specific + random phases	✓	✓	T	–
Full model	✓	✓	T	–

Table 3: Ablation template for parameter sharing and phase construction.

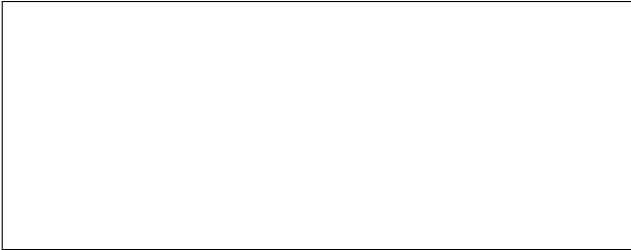

Figure 2: Placeholder for a framework compatibility heatmap. The figure should visualize pairwise compatibility $c(i, j)$ and optionally the selected phase partition.

to later stages. This figure also helps explain why random curricula can underperform even when using the same number of stages.

Error analysis. A valuable analysis is to group failed samples by compile error type. We expect joint PEFT to produce more cross-framework leakage, such as Vue-style directive patterns appearing in React code or missing Angular metadata after transfer from simpler frameworks. By contrast, phase-wise adaptation should reduce such leakage and shift the error distribution toward higher-level layout or long-context failures. This shift matters because it indicates that the method successfully internalizes framework syntax and project conventions, leaving only harder generation errors.

Low-resource transfer. Another useful evaluation is a low-resource setting where one framework has substantially less supervised data. Our expectation is that phase-wise adaptation benefits the scarce framework more than isolated training because it can absorb transferable layout knowledge from related frameworks before specializing. This experiment directly tests whether the method improves not only average CSR but also data efficiency.

Why not simple joint fine-tuning. A recurring criticism is that a sufficiently strong backbone plus joint fine-tuning might already solve the problem. The correct comparison is not raw model size but *budget-matched adaptation*. If a budget-matched joint LoRA still lags behind staged shared-plus-specific adaptation, the result would support the paper's claim that multi-framework WebUI generation is not merely more data, but a structured transfer problem.

Table 4 is particularly useful when reviewers ask *why* the method improves CSR. If the method is behaving as intended, it should disproportionately reduce framework-specific failures rather than merely producing shorter or simpler code. This is a more informative success signal than a single average number.

Error type	Joint PEFT	Per-framework	Phase-wise
Parse / syntax failure	–	–	–
Import / dependency failure	–	–	–
Template / binding failure	–	–	–
Component export failure	–	–	–
Style / asset mismatch	–	–	–

Table 4: Error taxonomy template. Filling this table helps show whether phase-wise adaptation reduces framework leakage and compilation-critical failures.

Backbone sensitivity. Because current WebUI research moves quickly, a final submission should ideally include at least two backbones, for example one open general-purpose MLLM and one stronger code-oriented or UI-oriented backbone. The expected pattern is that stronger backbones raise the overall ceiling, but compatibility-aware phase scheduling remains helpful because the conflict being addressed is framework-specific, not backbone-specific. Showing this would make the paper significantly stronger.

Connection to hierarchical generation. Our method is complementary to hierarchical generation systems such as DCGen and UICopilot [11, 12]. Those methods decompose long code synthesis, whereas ours reorganizes *training*. In principle, a future system could combine both ideas by using hierarchical decoding inside each phase. This distinction is worth emphasizing because it clarifies that phase-wise adaptation is orthogonal to many architectural choices and can be retrofitted onto different generators.

7 Discussion

The proposed method is intentionally modest. It does not require expensive online rollouts, repeated compiler-in-the-loop reinforcement learning, or framework-specific hand-written repair agents. Instead, it asks whether a better *training schedule* can already close a meaningful portion of the executability gap. This focus is appealing for two reasons. First, modern front-end benchmarks remain small compared with general multimodal corpora, so how data are organized can matter as much as how much data are used. Second, the multi-framework setting naturally contains both synergy and conflict. Staged adaptation offers a direct mechanism for exploiting the former while containing the latter.

The method also fits broader trends in MLLM adaptation. Strong backbones are increasingly frozen, with learning concentrated in lightweight modules [15, 17, 19]. At the same time, benchmarks in WebUI generation and GUI grounding

increasingly reveal that output validity and visual grounding must be considered jointly [13, 14, 34]. Our framework connects these two trends by using parameter-efficient modules to absorb framework-specific validity constraints while preserving multimodal alignment learned from screenshots.

8 Broader Impacts and Reproducibility

Automating WebUI implementation can lower development cost and broaden access to front-end prototyping, but it also introduces practical risks. Models that optimize only visual similarity may generate insecure or non-maintainable code, while models that optimize only compilability may silently produce brittle interfaces that fail accessibility or responsive design requirements. For this reason, a responsible evaluation should report not only CSR and visual metrics, but also basic engineering checks such as code length, dependency count, obvious accessibility markers, and the frequency of repeated boilerplate. In addition, the community should avoid overstating progress from HTML-only settings when the intended downstream use is component-based engineering.

From a reproducibility standpoint, the proposed method is straightforward to implement and easy to audit. All major degrees of freedom can be enumerated in a compact checklist: backbone choice, adapter type, compatibility subset, phase count, warm-up data, and build commands for each framework. We recommend releasing the phase assignment, compatibility matrix, exact scaffolds used for compilation, and sample-level failure logs. These artifacts are often more informative than aggregate metrics because they reveal whether gains come from genuine framework understanding or simply from conservative decoding patterns.

9 Limitations

This draft has several limitations. First, the current version intentionally omits empirical numbers because they were not provided by the user and should not be fabricated. Second, compatibility estimation is only as good as the proxy used to define it. Warm-start gradient alignment may miss deeper differences in project scaffolding or toolchain behavior. Third, the formulation emphasizes React, Vue, and Angular, and further work is needed to study other ecosystems such as Svelte, Next.js, or mobile UI frameworks. Finally, the contract loss only approximates executability; the final standard remains actual compilation.

10 Conclusion

We presented a draft NeurIPS-style paper for executable multi-framework WebUI generation with MLLMs. The key idea is phase-wise adaptation: a frozen multimodal backbone is equipped with shared and framework-specific adapters, and training is scheduled according to estimated compatibility among frameworks. This design targets the central tension of multi-framework WebUI generation, namely the coexistence of transferable UI abstractions and framework-specific compilation constraints. Beyond the method itself, the paper provides a full problem formulation, theoretical motivation, and experimental protocol that can be completed directly after actual runs are executed. We hope this draft offers a strong starting point for turning the abstract into a full submission.

References

[1] OpenAI. *GPT-4 Technical Report*. 2023. arXiv: 2303.08774 [cs.CL].

[2] Junnan Li et al. *BLIP-2: Bootstrapping Language-Image Pre-training with Frozen Image Encoders and Large Language Models*. 2023. arXiv: 2301.12597 [cs.CV].

[3] Haotian Liu et al. *Visual Instruction Tuning*. 2023. arXiv: 2304.08485 [cs.CV].

[4] Haotian Liu et al. *Improved Baselines with Visual Instruction Tuning*. 2023. arXiv: 2310.03744 [cs.CV].

[5] Peng Wang et al. *Qwen2-VL: Enhancing Vision-Language Model's Perception of the World at Any Resolution*. 2024. arXiv: 2409.12191 [cs.CV].

[6] Tony Beltramelli. "pix2code: Generating Code from a Graphical User Interface Screenshot". In: *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. 2018.

[7] Hugo Laurençon, Léo Tronchon, and Victor Sanh. *Unlocking the Conversion of Web Screenshots into HTML Code with the WebSight Dataset*. 2024. arXiv: 2403.09029 [cs.HC].

[8] Chenglei Si et al. "Design2Code: Benchmarking Multimodal Code Generation for Automated Front-End Engineering". In: *Annual Conference of the North American Chapter of the Association for Computational Linguistics*. 2025.

[9] Sukmin Yun et al. "Web2Code: A Large-scale Webpage-to-Code Dataset and Evaluation Framework for Multimodal LLMs". In: *Advances in Neural Information Processing Systems Datasets and Benchmarks Track*. 2024.

[10] Yi Gui et al. "WebCode2M: A Real-World Dataset for Code Generation from Webpage Designs". In: *Proceedings of the ACM Web Conference*. 2025.

[11] Yang Wan et al. *Automatically Generating UI Code from Screenshot*. 2024. arXiv: 2406.16386 [cs.SE].

[12] Yi Gui et al. "UICopilot: Automating UI Synthesis via Hierarchical Code Generation from Webpage Designs". In: *Proceedings of the ACM Web Conference*. 2025.

[13] Jingyu Xiao et al. *DesignBench: A Comprehensive Benchmark for MLLM-based Front-end Code Generation*. 2025. arXiv: 2506.06251 [cs.SE].

[14] Zhiyu Lin et al. *WebUIBench: A Comprehensive Benchmark for Evaluating Multimodal Large Language Models in WebUI-to-Code*. 2025. arXiv: 2506.07818 [cs.CL].

[15] Neil Houlsby et al. "Parameter-Efficient Transfer Learning for NLP". In: *International Conference on Machine Learning*. 2019.

[16] Xiang Lisa Li and Percy Liang. "Prefix-Tuning: Optimizing Continuous Prompts for Generation". In: *Annual Meeting of the Association for Computational Linguistics*. 2021.

[17] Edward J. Hu et al. *LoRA: Low-Rank Adaptation of Large Language Models*. 2022. arXiv: 2106.09685 [cs.CL].

[18] Rabeeh Karimi Mahabadi, James Henderson, and Sebastian Ruder. "Compacter: Efficient Low-Rank Hypercomplex Adapter Layers". In: *Advances in Neural Information Processing Systems*. 2021.

[19] Qingru Zhang et al. "AdaLoRA: Adaptive Budget Allocation for Parameter-Efficient Fine-Tuning". In: *International Conference on Learning Representations*. 2023.

[20] Yoshua Bengio et al. "Curriculum Learning". In: *International Conference on Machine Learning*. 2009.

[21] Zhiyuan Xu et al. "On the Statistical Benefits of Curriculum Learning". In: *International Conference on Machine Learning*. 2022.

[22] Alex Robinson. *Sketch2code: Generating a Website from a Paper Mockup*. 2019. arXiv: 1905.13750 [cs.CV].

[23] Vanita Jain et al. *Transformation of Sketches to UI in Real-time Using Deep Neural Network*. 2019. arXiv: 1910.08930 [cs.CV].

[24] Bryan Wang et al. "Screen2Words: Automatic Mobile UI Summarization with Multimodal Learning". In: *Proceedings of the ACM Symposium on User Interface Software and Technology*. 2021.

[25] Yi Gui et al. *VISION2UI: A Real-World Dataset with Layout for Code Generation from UI Designs*. 2024. arXiv: 2404.06369 [cs.SE].

[26] Jean-Baptiste Alayrac et al. *Flamingo: a Visual Language Model for Few-Shot Learning*. 2022. arXiv: 2204.14198 [cs.CV].

[27] Alec Radford et al. "Learning Transferable Visual Models from Natural Language Supervision". In: *International Conference on Machine Learning*. 2021.

[28] Kenton Lee et al. "Pix2Struct: Screenshot Parsing as Pretraining for Visual Language Understanding". In: *International Conference on Machine Learning*. 2023.

[29] Daniel Fried et al. *InCoder: A Generative Model for Code Infilling and Synthesis*. 2022. arXiv: 2204.05999 [cs.SE].

[30] Raymond Li et al. *StarCoder: may the source be with you!* 2023. arXiv: 2305.06161 [cs.SE].

[31] Baptiste Rozière et al. *Code Llama: Open Foundation Models for Code*. 2023. arXiv: 2308.12950 [cs.CL].

[32] Xiang Deng et al. "Mind2Web: Towards a Generalist Agent for the Web". In: *Advances in Neural Information Processing Systems Datasets and Benchmarks Track*. 2023.

[33] Boyuan Zheng et al. *GPT-4V(ision) is a Generalist Web Agent, if Grounded*. 2024. arXiv: 2401.01614 [cs.CL].

[34] Boyu Gou et al. *Navigating the Digital World as Humans Do: Universal Visual Grounding for GUI Agents*. 2024. arXiv: 2410.05243 [cs.CV].

[35] Yichen Pan et al. *WebCanvas: Benchmarking Web Agents in Online Environments*. 2024. arXiv: 2406.12373 [cs.CL].

[36] Kaixin Li et al. *ScreenSpot-Pro: GUI Grounding for Professional High-Resolution Computer Use*. 2025. arXiv: 2504.07981 [cs.CV].

[37] Aman Madaan et al. *Self-Refine: Iterative Refinement with Self-Feedback*. 2023. arXiv: 2303.17651 [cs.CL].