

# Bridging Design and Implementation: A Study of Multi-Agent LLM Architectures for Automated Front-End Generation

Caren Rizk  
c\_izk@live.concordia.ca  
Concordia University  
Montreal, Quebec, Canada

SayedHassan Khatoonabadi  
sayedhassan.khatoonabadi@concordia.ca  
Concordia University  
Montreal, Quebec, Canada

Emad Shihab  
emad.shihab@concordia.ca  
Concordia University  
Montreal, Quebec, Canada

## Abstract

Automating front-end development directly from design artifacts and textual requirements could accelerate iteration cycles and reduce implementation errors, yet most prior work addresses only a single modality (either design-to-code or text-to-code generation) without integrating complementary specifications. We propose a multi-agent framework that jointly reasons over user stories and Figma designs to synthesize complete React applications. The framework coordinates generation, validation, and repair through three architectural strategies: *Supervisor* (*tool-calling*) for centralized routing, *Hierarchical* for decomposed supervision, and *Custom* for deterministic workflow execution. Evaluated on four real-world projects (75 user stories) using six generator-judge model pairs (Claude, Gemini, GPT), the system achieves 54% full functional coverage and 58% full visual fidelity; including partial matches raises success rates to 77% and 85%, respectively. Architectural choice modestly affects quality (3–5 percentage-point variation) but substantially impacts cost: the Custom architecture reduces generator token usage by 21–65% compared to Hierarchical and Supervisor (*tool-calling*) configurations, while judge models consistently dominate overall cost (5.9× more tokens on average than generators). To further enhance pipeline stability and reduce manual intervention, we introduce a lightweight repair toolkit comprising automated refusal retries, JSX sanitization, and template scaffolding resolves the majority of generation-stage failures without regeneration. Overall, these results demonstrate that multimodal, agentic frameworks can reliably automate front-end synthesis, though achieving full production-grade quality still requires human refinement and improved handling of complex interaction behaviors.

## CCS Concepts

• **Software and its engineering** → **Automatic programming.**

## Keywords

Multimodal large language models, multi-agent systems, front-end code generation, figma-to-code, React applications

### ACM Reference Format:

Caren Rizk, SayedHassan Khatoonabadi, and Emad Shihab. 2026. Bridging Design and Implementation: A Study of Multi-Agent LLM Architectures for Automated Front-End Generation. In *23rd International Conference on Mining*

*Software Repositories (MSR '26)*, April 13–14, 2026, Rio de Janeiro, Brazil. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3793302.3793371>

## 1 Introduction

Modern user interface (UI) development follows a structured, multi-stakeholder process that transforms high-level business needs into functional applications [3]. In front-end workflows, stakeholders express features as user stories [11], UX designers translate these into visual mockups [14, 54], and engineers synthesize both artifacts into interactive code [52]. Integrating behavioral specifications (user stories) with visual specifications (mockups) remains resource-intensive and error-prone: misalignments between roles often lead to gaps between requirements, design, and implementation [8, 22]. Automating this transition could significantly improve consistency and accelerate iteration cycles [45].

Recent progress in Multimodal Large Language Models (MLLMs) has opened new opportunities for automating such front-end tasks, including UI code generation from text or visual inputs [26]. MLLMs excel at context-rich reasoning [22, 61], yet most existing work handles only one input modality; design mockups [5, 9, 16, 56] or textual descriptions [39, 58] and produces static layouts rather than dynamic, interactive applications. In contrast, real development integrates multiple artifacts: product managers define requirements, designers craft mockups, and engineers combine both into multi-page, stateful systems [13]. By isolating stages instead of modeling the full pipeline, prior systems cannot meet the functional and behavioral depth demanded by production code [26].

Moreover, there is little empirical evidence comparing LLM agent architectures on real software projects [15]. Most studies evaluate single-agent prompts or synthetic benchmarks [40, 44], leaving open questions about how coordination strategies affect quality and cost in realistic multimodal pipelines.

To bridge these gaps, **we propose a multi-agent framework that mimics real-world front-end workflows by combining user stories and Figma designs to generate dynamic React applications** [43, 52]. We adopt Figma due to its wide use in collaborative UI/UX prototyping [25, 50]. The two inputs are complementary: user stories which typically follows the format (As a [type of user], I want [some goal or feature] so that [reason or benefit].) define behavioral logic and state management, while Figma designs provide precise layout and style information. Integrating both allows our system to balance functional completeness and visual fidelity (capturing designer intent from Figma designs).

Each agent performs a specialized task (layout parsing, component generation, enrichment, or validation) mirroring practical development pipelines [1]. We investigate how different architectural strategies influence scalability and output quality. Specifically, **we compare Supervisor (tool-calling), Hierarchical, and Custom**



This work is licensed under a Creative Commons Attribution 4.0 International License. *MSR '26, Rio de Janeiro, Brazil*

© 2026 Copyright held by the owner/author(s).  
ACM ISBN 979-8-4007-2474-9/2026/04  
<https://doi.org/10.1145/3793302.3793371>

## architectural patterns [34] using a dataset of four real-world GitHub projects spanning 75 user stories paired with Figma frames.

Our study provides a reproducible benchmark and an architectural analysis of multimodal code generation. Unlike synthetic benchmarks, our dataset is mined directly from real GitHub repositories that contain both user stories and linked Figma designs, enabling empirical analysis of how agentic LLM systems reconstruct repository-sourced specifications. Using this dataset, we address the following three research questions:

**RQ1:** *How effectively can our multi-agent framework generate React applications that align with functional and visual user story requirements?* Our approach integrates multimodal grounding by combining Figma designs and textual requirements within a coordinated multi-agent framework that generates, verifies, and repairs React applications. Empirical results show that this framework achieves an average of 54.1% full functional coverage and 57.9% full visual fidelity, rising to 76.9% and 84.9% when partial matches are included. These findings indicate that the framework can reliably produce functionally correct and visually faithful applications.

**RQ2:** *How do different agent architectures affect token consumption and code quality?* We compare three architectural strategies (Supervisor (tool-calling), Hierarchical and Custom) using identical datasets, judges, and token-based cost metrics. Results show that architectural choice modestly affects quality (within 3–5 percentage points) but substantially impacts efficiency: Custom controllers reduced token usage by 21–65% compared to Hierarchical and Supervisor (tool-calling) configurations, without loss in output quality. Across all settings, Judge agents consumed on average 5.9× more tokens than generators, highlighting evaluation as the primary cost driver.

**RQ3:** *What types of failures occur in the generation process, and how can they be automatically detected or repaired?* We analyze generation-stage failures across all model–architecture combinations, identifying two dominant failure modes: LLM refusals and code-generation artifacts. Refusals occurred exclusively with GPT during complex generation prompts, but 29.2% were recovered automatically through a regex-based retry mechanism bringing the total passes to 91%. Code artifacts (such as code fences, pre-code text, and missing exports) were systematic, deterministic, and easily corrected via the automated JSX\_cleanup repair tool. These results indicate that most failures stem from low-level formatting inconsistencies rather than conceptual reasoning errors, and that lightweight recovery and cleanup methods effectively stabilize the generation pipeline.

**Contributions.** This paper makes the following contributions:

- A novel multi-agent framework integrating Figma designs and user stories to generate complete React applications.
- A comparative analysis of three coordination strategies Supervisor (tool-calling), Hierarchical, and Custom and their trade-offs in terms of quality and cost.
- A curated benchmark of real Figma files and user stories with labeled functional and visual coverage.
- To promote the reproducibility of our study and facilitate future research on this topic, we publicly share our scripts and dataset online [53].

**Paper Organization.** Section 2 reviews related work. Section 3 introduces our framework and experimental setup. Section 4 describes the different architectural strategies, Section 5 analyzes results, Section 6 discusses threats to validity, and Section 7 concludes.

## 2 Related Work

Research on automated front-end generation spans four complementary domains that directly inform our study: (1) text-to-code generation from requirements, (2) image-based and multimodal UI generation, (3) commercial design-to-code systems, and (4) multi-agent orchestration for software development. Together, these areas trace the evolution from single-prompt code synthesis toward integrated, collaborative systems capable of grounding code in both textual intent and visual design.

### 2.1 Text-to-Code from Requirements

Text-to-code generation has advanced rapidly with models such as Codex and GPT [47], which synthesize UI logic directly from high-level natural language specifications [57]. GUIDE [29] uses retrieval-augmented prompting to decompose textual goals into UI components, while Kretzer et al. [30] propose an assistant that cross-checks Figma designs against user stories to improve coverage. These works emphasize grounding UI generation in functional intent rather than visual layout. However, most systems focus solely on textual requirements and offer limited guarantees of visual fidelity to design artifacts. Our work explicitly integrates visual and textual constraints, enabling end-to-end evaluation and repair of both functional and visual aspects within the same framework.

### 2.2 Image-Based and Multimodal Code Generation

This area explores how visual UI representations can be transformed into executable code, often enhanced with multimodal reasoning. Nikiforova et al. [46] proposed a rule-based, model-driven approach converting draw.io mockups into AngularJS components through model-to-text rules this approach is limited to static sources.

Recent multimodal frameworks address these limitations. ScreenCoder [27] introduces a modular, three-stage process (grounding, planning, generation) to improve layout accuracy and interpretability. DesignCoder [10] adopts a hierarchy-aware, self-correcting pipeline to boost structural fidelity, while UICopilot [23] decomposes long HTML generation hierarchically. These advances show a shift from static, rule-based generation toward modular and self-correcting frameworks. Our approach extends this trajectory by combining textual user stories with Figma designs in a unified multi-agent pipeline that generates dynamic, interactive applications instead of static layouts.

### 2.3 Commercial and Industrial Systems

Commercial tools like Builder.io [7] and Locofy.ai [42] translate Figma designs into production-ready components using proprietary parsing and generation models. Builder.io’s Visual Copilot refines generated code with AST transformation and LLM prompting, while Locofy exports interactive flows to multiple frameworks. Startups such as Bolt.new [6] provide chat-based scaffolding from

design artifacts. Unlike these closed-source systems, our framework is open and research-oriented. It enables transparent comparison of multi-agent orchestration strategies and quantitative evaluation of functional and visual generation quality using reproducible datasets.

## 2.4 Multi-Agent LLM Orchestration for Code Generation

This research stream examines how multiple specialized agents coordinate to perform complex software engineering tasks through structured communication and role specialization. MetaGPT [24] simulates a software company with role-based agents, AutoGen [59] supports conversational collaboration via tool use, and ChatDev [51] organizes designer-coder-tester pipelines for incremental development. Despite these advances, most evaluations rely on synthetic datasets, focus only on textual requirements, and rarely assess visual fidelity or automated repair.

Our work extends this line by empirically comparing three architectural strategies Supervisor (tool-calling), Hierarchical, and Custom on real, multi-page React projects grounded in both user stories and Figma designs. We evaluate functional coverage, visual fidelity, and automated repair while analyzing token and image usage across architectures, bridging the gap between conceptual multi-agent frameworks and practical, multimodal front-end generation.

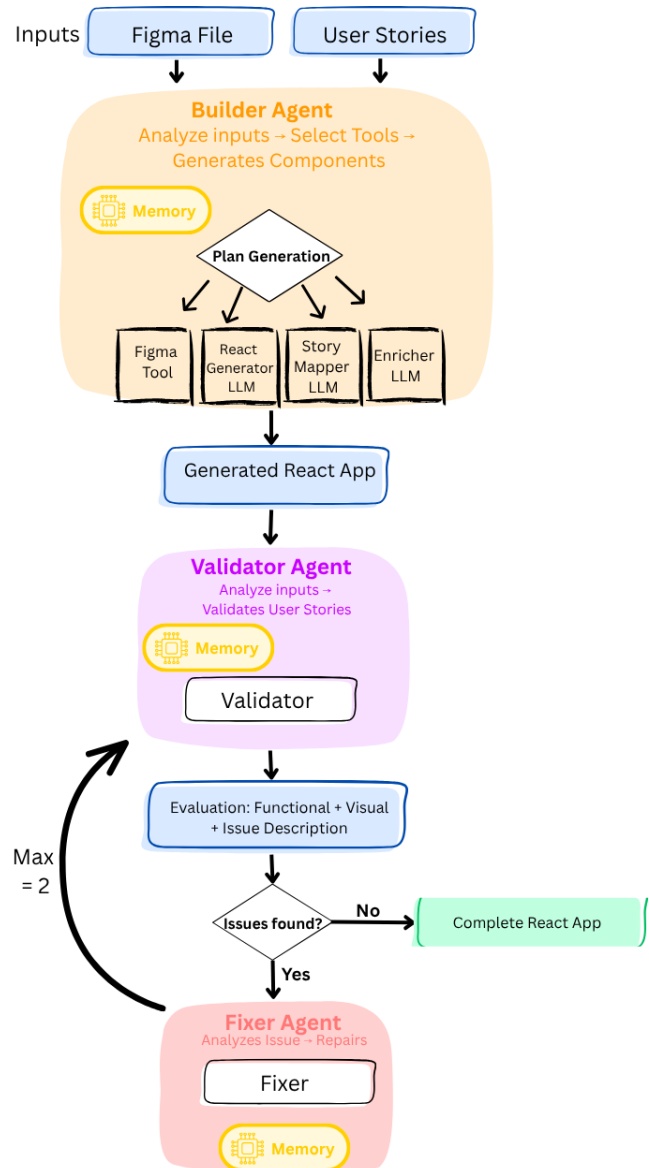
## 3 Approach

The goal of this study is to compare how different architectural strategies for MLLMs affect the quality and cost of front-end React application generation. An overview of the proposed multi-agent framework is shown in Figure 1.

### 3.1 Agents

The framework is organized around three autonomous agents **Builder**, **Validator**, and **Fixer**. These agents operate sequentially to generate, assess, and iteratively improve React applications. Each agent maintains a local memory and makes independent decisions about which tools to invoke based on prior context and outcomes. The specific tools used by each agent are detailed in Section 3.2, and all prompt templates and implementation details are available in the replication package [53].

- **Builder Agent:** is responsible for converting paired inputs from the Figma design and user stories into an initial runnable React application. It integrates information from visual layouts and textual requirements to produce structurally complete JSX components with routing and basic interactivity. The resulting application is then passed to the Validator agent.
- **Validator Agent:** this agent performs functional and visual evaluation of the generated application. Using its internal validation tool, it compares the generated components to their corresponding user stories and Figma references, producing a structured report that assigns each story a status label [full match] (all key elements are present, styled appropriately, and clearly communicate the intent of the user story), [partial] (some essential elements are present, but



**Figure 1: Multi-agent framework for React application generation with autonomous agents (Builder, Validator, Fixer) that use memory and decision-making to coordinate tool usage and iteratively refine code quality.**

others are missing, incorrect, or poorly aligned), or [fail] (the critical elements required to communicate the user story are absent) along with concise, file-scoped issue descriptions. The detailed evaluation criteria are included in the replication package [53]. These reports serve as actionable feedback for downstream repair.

- **Fixer Agent:** the fixer agent closes the feedback loop by applying targeted repairs based on the Validator's report. It leverages the internal fixer tool to automatically patch localized issues such as missing elements or unimplemented

event logic. Each round of fixes is re-evaluated by the Validator, with a maximum of two validation–repair cycles to prevent oscillation and ensure stable convergence.

All stages of the pipeline are fully automated. Failures such as missing files, blank renders, JSX compilation errors, and model refusals are detected programmatically and counted as failures if unrecoverable. No manual intervention or hand correction was applied during the experimental runs.

## 3.2 Tools

To ensure a fair comparison across architectures, we define a set of standardized tools for our framework. In Section 4, we explain how these tools are coordinated within each architectural strategy.

- **Figma Tool:** An automated, non-LLM tool that receives a Figma file and extracts inline-style HTML with structural hierarchies and styling information. The output is HTML with embedded style blocks that preserves layout semantics, component boundaries, and styling context before translation into code. Unlike other tools in the pipeline, this component is entirely rule-based and does not rely on any LLM. A hand-coded extractor was required to guarantee HTML preserving layout semantics, hierarchy, and style attributes exactly as designed. This ensures that downstream LLM agents operate on a faithful structural representation rather than on ambiguous or lossy textual descriptions of the design.
- **React Generator LLM:** Receives HTML output from the Figma Tool along with a frame screenshot and converts them into React JSX with inline CSS. The output is syntactically valid JSX components that provide a foundation for adding dynamic behavior and routing. This step mirrors the first action taken by front-end engineers in real-world workflows which is building static page structures that visually match the design, often using mock data or no data at all to simplify the process before wiring functionality. By isolating visual scaffolding from logic enrichment, this tool allows the system to separate purely structural correctness (layout, syntax, component hierarchy) from later behavioral synthesis, improving modularity and debugging transparency.
- **Story Mapper LLM:** Receives user stories and a component inventory, then matches each story to the relevant React component files. The output is a standardized JSON mapping of the form [component → stories], ensuring that functional logic is attached to the correct UI files for subsequent enrichment. Rather than passing the entire codebase to the LLM during the enrichment phase, this step strategically narrows the context to only those files relevant to each user story. This targeted mapping reduces token overhead, minimizes noise from unrelated files, and improves the precision of downstream enrichment, allowing the LLM to reason locally about functionality while maintaining global architectural consistency.
- **Enricher LLM:** Receives component JSX and user stories, then enriches components with dynamic behavior such as state management, event handlers, and routing logic derived from the stories. The output is enriched JSX code that transforms static components into interactive, functional elements

while preserving visual fidelity. This step mirrors the second pass typically performed by front-end engineers, who first scaffold pages visually and then integrate interactivity to ensure the application behaves as intended. Keeping enrichment separate from the initial generation is also beneficial to limit token consumption by reducing context size and prevent confusing the LLMs.

- **Validator:** Executes automated functional and visual checks on the generated application. It outputs structured evaluation labels (full\_match/partial/fail) with brief issue descriptions, serving as a programmatic equivalent of a QA review. The Validator is directly invoked by the Validator Agent described in Section 3.1.
- **Fixer:** Applies targeted modifications to source files based on Validator feedback. It automatically repairs common artifacts, missing exports, or broken bindings and outputs corrected JSX files, closing the loop between generation and evaluation. The Fixer tool is used internally by the Fixer Agent described in Section 3.1 during the repair stage.

## 3.3 Experiment Setup

To evaluate our multi-agent framework fairly across different architectural strategies, we employ three state-of-the-art MLLMs: GPT-4o, Gemini-2.5-Pro, and Claude-Sonnet-4.5. Each model was selected to represent distinct strengths in the current LLM landscape: GPT-4o offers robust vision capabilities and strong instruction-following [47]; Gemini-2.5-Pro provides competitive multimodal reasoning and enhanced coding [19]; and Claude-Sonnet-4.5 excels at nuanced code generation and detailed adherence to specifications [4]. These model families also dominate leading public and research leaderboards: for example, the Vellum AI Leaderboard ranks Gemini 2.5 Pro, Claude 4.5 Sonnet, and GPT-4o among the top models for reasoning and coding tasks [2]; and SWE-bench an academic benchmark for software engineering tasks which confirms that GPT, Gemini, and Claude families achieve state-of-the-art results across issue resolution and repair tasks [28, 60]. By evaluating across three models, we mitigate model-specific artifacts and assess whether architectural improvements generalize across different LLM bases.

**LLM as a Judge.** Similar to prior work [41, 55] we used LLM-as-a-Judge to evaluate how well the generated front-end applications align with both functional user stories and Figma-based visual designs. Each Judge receives both the generated source code and a rendered screenshot of the application, which it compares against the original Figma frame and user story. Critically, we separate the role of code generation from judgment to isolate architectural effects. In each run, one model serves as the React code generator (executing the builder agents), while a different model serves as the final Judge (performing functional and visual evaluation). Prior studies have shown that using the same model for both roles can introduce “self-preference” bias where models tend to rate their own outputs more favorably [38, 48]. To mitigate this, our internal Validator and final evaluation Judge use the same structured prompt and rubric, but are executed on different models (e.g., GPT-4o-generator with Claude-judge, or Gemini-generator with GPT-4o-judge). This separation ensures that no model evaluates its own

outputs. The full Judge prompt can be found in the replication package [53].

### 3.4 Dataset

To evaluate our multi-agent pipeline under realistic front-end development conditions, we curated a dataset consisting of paired user stories and their corresponding visual design representations. The dataset creation involved several stages, combining automated retrieval with manual verification to ensure quality and alignment.

- **Source Collection:** We used GitHub’s advanced search to retrieve issues labeled as "user-story" within JavaScript-based open-source repositories. The query used was:  
label: "user-story" language: JavaScript  
This search returned 1.2k issues, covering 83 unique repositories.
- **Filtering Criteria:** We applied the following filtering rules to refine the dataset:
  - Excluded issues written in non-English languages to ensure consistent semantic grounding.
  - Discarded repositories containing fewer than 5 user stories, as these offered insufficient grounding for generating or evaluating a multi-component application.
- **Repository Selection and Validation:** For each remaining repository, we manually reviewed its README, documentation, and linked assets to verify the availability of visual design resources. Repositories were retained only if they contained a publicly accessible Figma link, allowing direct extraction of design frame which resulted in 4 repositories.

**Final Dataset Composition.** The resulting dataset comprises 75 user stories across four open-source front-end projects. Every story–design pair was manually inspected to ensure it represented a reliable input–output mapping for both *functional validation* (based on the user story) and *visual matching* (based on the design). This dataset serves as the foundation of our evaluation.

The four projects vary in complexity and domain:

- Vox-Box [17]: a volunteer coordination platform (2 stars, 34 user stories).
- Transcriber [18]: an audio transcription interface (3 stars, 23 user stories).
- Urban-Calendar [21]: a collaborative event-scheduling app (3 stars, 9 user stories).
- House-Hunting [20]: a property listing interface (2 stars, 9 user stories).

Overall, the evaluation includes 450 independent experimental instances, combining 4 projects (75 stories) with 3 architectural configurations and 3 model pairs (serving as generator–judge combinations). Specifically, each user story is *generated six times* (3 architectures  $\times$  2 model pairs per architecture) and *evaluated six times* (by its paired evaluator). This setup enables robust descriptive analysis across architectures, assessing consistency and variance in functional correctness, visual fidelity, and cost efficiency under diverse architectural strategies.

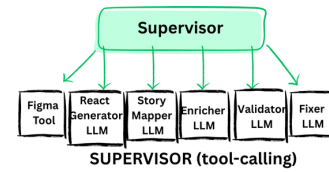


Figure 2: Overview of our agents coordination in the Supervisor (tool-calling) architecture

## 4 Agent Architectures

To orchestrate the full lifecycle of code generation from visual design and textual specifications, we adopt LangGraph [31] as the orchestration substrate. LangGraph models an application as a stateful directed graph whose nodes are agents/tools and whose edges encode control flow (including conditional routing and parallel branches). This lets us compose the full lifecycle: generation (builder agents), assessment (functional/visual validation), and targeted repair (fixers) under one reproducible workflow with checkpoints and retries.

LangGraph defines five architectural patterns for composing multi-agent systems: *Supervisor*, *Supervisor (Tool-Calling)*, *Hierarchical*, *Network*, and *Custom* [34]. Each offers distinct trade-offs in coordination, flexibility, and reproducibility that will be discussed below.

### 4.1 Supervisor

A single coordinator dynamically selects which specialized agent to invoke based on the evolving conversation state. This enables flexible routing but can become unstable or inefficient as context grows, since the supervisor must reason about all agents and outputs in natural language [36, 49, 59]. Because this approach relies heavily on unstructured text-based reasoning, we excluded it from our experiments in favor of the *Supervisor (Tool-Calling)* variant, which offers greater determinism and reproducibility.

### 4.2 Supervisor (Tool-Calling)

A structured variant of the Supervisor architecture that represents individual agents as callable tools. In this setup, a single decision-making LLM (the supervisor) determines which agent tool to invoke and which arguments to pass, reducing free-form meta-reasoning and improving interpretability and determinism [37]. It maintains centralized control but executes through explicit tool interfaces (builder, validator, fixer), ensuring clearer routing and consistent coordination. In our implementation, the supervisor manages the pipeline by invoking each agent as a registered tool and interpreting feedback after every stage to decide the next step. This design supports adaptive branching (e.g., retrying the fix stage when visual fidelity remains partial) while maintaining a transparent, centralized decision process. However, since all reasoning occurs through a single agent, long context windows can reduce consistency on larger projects (Figure 2).

### 4.3 Hierarchical

A multi-level architecture in which a top-level supervisor delegates subtasks to intermediate supervisors, each managing its own

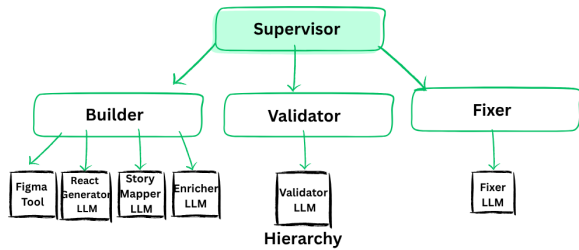


Figure 3: Overview of our agents coordination in the Hierarchical architecture

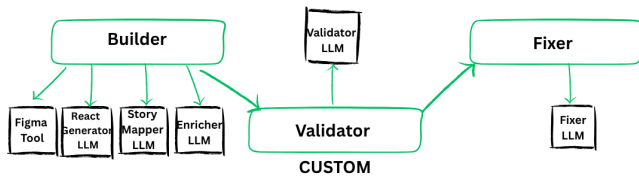


Figure 4: Overview of our agents coordination in the Custom architecture

specialized agents. This tree-based organization localizes context, shortens prompts, and scales coordination, making it well-suited for complex workflows [33]. In our configuration, the top-level controller orchestrates build–validate–fix loops through three sub-supervisors: a *Builder Team*, a *Validator Team*, and a *Fixer Team*. Each sub-supervisor autonomously manages its team’s logic and reports summarized results upward, confining reasoning to local contexts and improving efficiency as project scope grows (Figure 3).

#### 4.4 Network

The Network pattern connects agents as a general graph with free-form communication and no fixed sequence or hierarchy. Lang-Graph recommends this design for problems that lack clear dependency structure or execution order [35]. Since our workflow follows a strict dependency chain (*Build* → *Validate* → *Fix*), adopting the Network pattern would introduce unnecessary routing complexity and undermine determinism. Therefore, it was excluded from our evaluation.

#### 4.5 Custom

The Custom architecture provides explicit control over the workflow graph, defining both the sequence and conditions under which agents are invoked. Agents are represented as nodes, and transitions between them are deterministic, allowing precise measurement of cost, performance, and agent interactions [32]. In our implementation, the workflow is encoded as a finite-state pipeline (*Build* → *Validate* → *Fix*), where each transition is automatically triggered by the agents output. This removes high-level reasoning entirely, resulting in a reproducible and auditable process that supports fine-grained measurement of token and image usage per stage (Figure 4).

Even single-agent subgraphs such as *Validator* and *Fixer* in the Hierarchical and Custom architectures are wrapped in supervisory layers to maintain a consistent prompting and tool interface

across all subgraphs. Without these lightweight supervisors, each single-agent stage would require its own prompt and routing logic, reducing modularity and increasing implementation divergence between architectures.

Although all three architectures operate within the same overall development workflow, they differ in how agent coordination and decision-making are managed. Each experiment executes the same sequence of specialized agents but under different architectural schemes.

## 5 Results

In this section, we present the experimental findings addressing our three research questions. Each subsection follows a consistent structure: *Motivation* (explaining why the question matters), *Approach* (describing how it was investigated), and *Results* (summarizing key observations and quantitative outcomes). Together, these analyses evaluate generation quality and token efficiency across models and architectures.

### 5.1 RQ1: How effectively can our multi-agent framework generate React applications that align with functional and visual user story requirements?

**Motivation.** Automated front-end generation requires models to jointly reason over visual layouts and functional requirements; a challenge that tests their ability to bridge design intent with executable logic. In this RQ, our goal is to assess how effectively the framework integrates functional and visual specifications to produce dynamic applications.

**Approach.** To answer this RQ, we generated complete React applications from paired user stories and Figma designs using all three multi-agent architectures (Supervisor (tool-calling), Hierarchical, and Custom) as discussed in Section 4. Each output was assessed along the following two dimensions:

- (1) Functional coverage: whether the generated components implemented the behaviors described in user stories (e.g., event handling, routing, state updates).
- (2) Visual fidelity: whether the generated UI matched the Figma design in layout, structure, and style.

We utilized a *Judge agent* as described in Section 3.3. The Judge rated each output as [full match] (all key elements are present, styled appropriately, and clearly communicate the intent of the user story), [partial] (some essential elements are present, but others are missing, incorrect, or poorly aligned), or [fail] (the critical elements required to communicate the user story are absent).

To validate the reliability of our *LLM-as-a-Judge* setup, the first author conducted a manual audit on 30 user stories per judge, assessing both functional coverage and visual fidelity. This corresponds to 180 random judgments (30 stories × 2 criteria × 3 judges) sampled from the total 450 evaluated cases. Each manually reviewed sample was cross-compared against the model-generated labels to measure inter-rater agreement using Cohen’s  $\kappa$  [12] to assess agreement between the LLM judges and manual ratings. Manual annotation was used exclusively to validate the reliability of the LLM judges and was never used to compute or adjust the reported functional

**Table 1: Functional Coverage and Visual Fidelity Results by Generator–Judge Configuration (Aggregated Across All Projects and Architectures)**

Configuration (Generator–Judge)	Functional Coverage			Visual Fidelity		
	Full Match	Partial	Fail	Full Match	Partial	Fail
Gemini–GPT	124 (63.6%)	58 (29.7%)	13 (6.7%)	166 (51.7%)	139 (43.3%)	16 (5.0%)
GPT–Gemini	74 (33.3%)	56 (25.2%)	92 (41.4%)	0 (0%)	0 (0%)	0 (0%)
Gemini–Claude	140 (62.2%)	20 (8.9%)	65 (28.9%)	178 (68.7%)	23 (8.9%)	58 (22.4%)
Claude–Gemini	119 (52.9%)	56 (24.9%)	50 (22.2%)	0 (0%)	0 (0%)	0 (0%)
Claude–GPT	134 (58.5%)	70 (30.6%)	25 (10.9%)	117 (50.6%)	91 (39.4%)	23 (10.0%)
GPT–Claude	122 (55.0%)	40 (18.0%)	60 (27.0%)	124 (62.0%)	16 (8.0%)	60 (30.0%)
<b>Total</b>	<b>713 (54.1%)</b>	<b>300 (22.8%)</b>	<b>305 (23.1%)</b>	<b>585 (57.9%)</b>	<b>269 (26.6%)</b>	<b>157 (15.5%)</b>

*Note.* Visual scores show 0% for GPT–Gemini and Claude–Gemini due to systematic evaluation failures (ERR).

or visual scores. Visual Fidelity achieved the highest consistency, with  $\kappa$  values ranging from 0.66 for Claude to 0.81 for GPT, and 0.90 for Gemini. Functional Coverage showed slightly lower agreement, ranging from 0.55 for Claude to 0.69 for GPT reaching 0.70 for Gemini. Overall, agreement was consistently higher for visual fidelity than for functional coverage, indicating that layout and style judgments were more consistently interpreted than behavior-oriented assessments. **Across all judges, the  $\kappa$  values indicate substantial to almost-perfect reliability, supporting the robustness of the evaluation rubric.**

**Results.** Table 1 presents the functional coverage and visual fidelity results averaged across all projects (Vox-Box, Transcriber, Urban-Calendar, House-Hunting) and architectures (Supervisor (tool-calling), Hierarchy, Custom), grouped by model pairs. In each pair, the first model acts as the *Generator* (responsible for executing the entire pipeline) and the second model as the *Judge* (evaluating the output) for example if we take the first row [Gemini–GPT], the generator is Gemini and the Judge would be GPT.

On average, **models achieved 54.1% full functional coverage and 57.9% full visual fidelity. When including partial cases, these rates increase to 76.9% (functional coverage) and 84.9% (visual fidelity).** Among all model families, Gemini achieved the highest overall performance, leading in both functional accuracy and visual alignment, indicating stronger multimodal reasoning and grounding in design context. Overall, these results show that **the generated applications are often functionally and visually aligned with user requirements**, with partial cases contributing over 20% additional coverage this suggests that most failures involve localized or fixable issues rather than complete generation errors.

We next examine the variation across individual projects shown in Table 2. *Vox-box* achieved the highest success rates (66.1% functional coverage, 72.4% visual fidelity). By contrast, *Transcriber*, an audio processing application with specialized AI-related UI elements, proved most challenging (26.6% functional coverage, 29.8% visual fidelity). We observed performance variations across different UI complexities and application domains, suggesting that **non-standard component logic and domain-specific interactions reduce consistency among models and architectures.**

**RQ1 Summary:** The evaluation results show that our multi-agent framework can often generate functionally and visually aligned React applications. Full-match accuracy averaged 54.1% for functional coverage and 57.9% for visual fidelity, rising to 76.9% and 84.9% when partial matches are included. Comparing results across architectures we observed performance variations across different UI complexities and application domains suggesting that non-standard component logic and domain-specific interactions reduce consistency among models and architectures.

## 5.2 RQ2: How do different agent architectures affect token consumption and code quality?

**Motivation.** Automated front-end generation pipelines differ not only in output quality but also in computational efficiency. Understanding how architectural choices influence token consumption and performance is essential for scalable deployment, especially in multimodal multi-agent setups where repeated reasoning and image processing can significantly inflate cost. In this RQ, our goal is to examine how different architectures balance efficiency and quality; quantifying their impact on total token usage (cost) and overall cost-performance ratio.

**Approach.** To answer this RQ, we quantify how architectural strategies impact *token consumption* and *quality* under identical tasks. Using the same stories–design pairs as in RQ1, we run all three architectures (Supervisor (Tool-Calling), Hierarchical, Custom) across the same model families (GPT, Gemini, Claude). We separate roles into generator (builder pipeline) and judge (functional/visual evaluation). To isolate architectural efficiency, in Tables 3 and 4, we averaged token usage across all models (GPT, Claude, Gemini) for each architecture. This aggregation removes model-specific variance and highlights the relative overhead introduced by architectural design alone.

For every run, we log the following metrics and report per-architecture aggregates:

- **Total tokens** – the sum of all input and output tokens consumed during a full generation cycle, representing the overall computational cost.
- **Prompt tokens** – the number of tokens provided to the model as input context (including prior messages, instructions, and architectural coordination data).

**Table 2: Functional Coverage and Visual Fidelity Results by Project (Aggregated Across All Configurations and Architectures)**

Project	Functional Coverage			Visual Fidelity		
	Full Match	Partial	Fail	Full Match	Partial	Fail
home-hunting	96 (59.6%)	49 (30.4%)	16 (9.9%)	43 (51.2%)	42 (50.0%)	3 (3.6%)
vox-box	405 (66.1%)	99 (16.2%)	108 (17.6%)	420 (72.4%)	93 (16.0%)	67 (11.6%)
urban-calendar	102 (61.4%)	37 (22.3%)	27 (16.3%)	49 (44.1%)	48 (43.2%)	14 (12.6%)
transcriber	110 (26.6%)	115 (27.8%)	159 (38.5%)	73 (29.8%)	86 (35.1%)	86 (35.1%)

Note. Visual totals differ from functional results due to systematic evaluation failures in the GPT–Gemini and Claude–Gemini configurations.

**Table 3: Average token consumption for Generators aggregated across all models (in millions). Lowest-cost architecture highlighted in bold.**

Architecture	Prompt Tokens (M)	Completion Tokens (M)	Avg. Total Tokens (M)
Supervisor (tool-calling)	2.17	0.40	3.10
Hierarchical	1.57	0.40	2.47
<b>Custom</b>	<b>0.87</b>	<b>0.33</b>	<b>1.40</b>

**Table 4: Average token consumption for Judges aggregated across all models (in millions). Lowest-cost architecture highlighted in bold.**

Architecture	Prompt Tokens (M)	Completion Tokens (M)	Avg. Total Tokens (M)
<b>Supervisor (tool-calling)</b>	9.87	0.17	<b>10.20</b>
Hierarchical	19.47	0.33	19.93
Custom	10.33	0.17	10.77

- **Completion tokens** – the number of tokens generated by the model as output (i.e., newly produced code, reasoning steps, or judgments).

**Results.** Table 3 shows token consumption across the three architectural configurations. The **Custom architecture achieved the lowest average token cost** (1.4M), representing a 43% reduction compared to Hierarchical (2.47M) and a 55% reduction compared to Supervisor (tool-calling) (3.1M). Prompt tokens dominated total usage (60–70% of all tokens), confirming that most architectural overhead arises from context duplication rather than extended reasoning or output generation.

Completion tokens (actual code generation) represented only 13–24% of total token usage across all architectures, with prompt tokens (context reading) dominating at 60–70%. Since completion ratios were stable, all architectures performed similar amounts of actual code generation. The token efficiency differences arose from how much context each architecture needed to duplicate and re-read, not from differences in reasoning or generation complexity.

Table 4 shows average Judge token usage. Judges are the dominant cost drivers, consuming roughly 3.3× (Supervisor), 8.1× (Hierarchical), and 7.7× (Custom) more tokens than Generators (Table 3). Averaged across architectures, this is about 5.9× more overall.

**Table 5: Functional coverage of the multi-agent framework across architectural strategies. Results show counts and percentages for user story evaluation.**

Architecture	Full Match	Partial	Fail
Supervisor (tool calling)	220 (49.5%)	93 (21.0%)	131 (29.5%)
Hierarchical	230 (51.3%)	112 (25.0%)	106 (23.7%)
Custom	236 (53.3%)	97 (21.9%)	110 (24.8%)
<b>Average</b>	<b>51.4%</b>	<b>22.6%</b>	<b>26.0%</b>

**Table 6: Visual fidelity of the multi-agent framework across architectural strategies. Results show counts and percentages for Figma design alignment.**

Architecture	Full Match	Partial	Fail
Supervisor (tool calling)	208 (56.8%)	110 (30.0%)	48 (14.2%)
Hierarchical	150 (53.0%)	89 (31.5%)	44 (15.5%)
Custom	197 (58.1%)	93 (27.4%)	49 (14.5%)
<b>Average</b>	<b>56.2%</b>	<b>29.5%</b>	<b>14.2%</b>

Judges required 10.2M–19.9M tokens per run, with Hierarchical incurring the highest cost (19.9M).

Tables 5 and 6 compare token usage against accuracy and reveal a weak or even inverse correlation. The Hierarchical architecture consumed 26–126% more tokens than the Custom architecture but achieved lower functional and visual accuracy (51.3% vs. 53.3%; 53.0% vs. 58.1%). In contrast, the **most efficient setups (Custom architectures) achieved the highest accuracy at the lowest cost**, confirming that increased coordination complexity does not necessarily translate into better generation quality.

**RQ2 Summary:** Across all models and roles, Custom architectures delivered the best cost–performance ratio, reducing token consumption by 21–65% relative to Hierarchical and Supervisor (tool-calling) setups *without degrading output quality*. Judges consistently dominated total cost, using on average 5.9× more tokens than generators. Overall, streamlined and deterministic pipelines achieved superior generation quality per token spent, confirming that minimizing context duplication is key for scalable multimodal multi-agent generation.

### 5.3 RQ3: What types of failures occur in the generation process, and how can they be automatically detected or repaired?

**Motivation.** Incomplete or invalid responses such as non-code outputs, partial files, or malformed JSX in code generation disrupt the pipeline and require manual correction. While conceptually simple, these issues reduce automation reliability and increase post-processing cost, making it essential to understand their frequency and impact to improve workflow stability.

**Approach.** To answer this RQ, we systematically analyzed all generation failures and recovery events across models and architectures.

**LLM refusals** occurred when a model declined to produce code or returned a meta-response instead of executable output. These behaviors were observed exclusively when GPT acted as a generator, while Claude and Gemini exhibited none. Each refusal automatically triggered a retry mechanism that resent the same prompt up to three times before discarding the attempt. Refusal detection relied on a case-insensitive regex filter that matched the following canonical refusal cues (a response was flagged if it matched *any* pattern):

- \bi'm sorry\b
- \bi cannot help\b
- \bi can't help\b
- \bi'm unable to\b
- \bas an ai\b
- \bi need more information\b
- \bplease provide more details\b
- \bcould you please provide\b
- \bi need.\*specific user stor(yies)

**Code generation artifacts** represented another major source of failure, often blocking compilation or rendering prior to evaluation. To mitigate these issues, we developed `JSX_cleanup` a lightweight automated repair tool that detects and corrects common LLM-specific artifacts through pattern-based sanitization. The script was executed after each Enricher LLM run and applied the following transformations across 144 files per model:

- (1) **Code Fence Artifacts:** Removal of markdown code fences (e.g., “`jsx . . .`”) that break JSX syntax.
- (2) **Pre-Code Text:** Truncation of textual explanations preceding the first valid `import` statement.
- (3) **Missing Export Statements:** Automatic inference and insertion of `export default` declarations for unexported components.
- (4) **Post-Code Text:** Removal of descriptive text following the final export.
- (5) **Excessive Comments:** Cleanup of overly annotated JSX via single-line and block comment stripping.
- (6) **Missing Project Files:** Automatic scaffolding of commonly omitted files (`index.html`, `App.jsx`, `index.js`, `index.css`).
- (7) **Image Path and package.json Fixes:** Repair of broken imports and incomplete configurations using template-based patching.

**Table 7: Summary of refusal-handling outcomes for GPT during code generation across all architectures.**

Outcome Category	Count	Percentage
Passed on first try	89	61.8%
Recovered after retries	42	29.2%
Unrecovered after 3 retries	13	9.0%
<b>Total attempts</b>	<b>144</b>	<b>100%</b>

**Table 8: Aggregate JSX cleanup operations by issue type and model family.**

Issue Type	Claude	Gemini	GPT
Code fences	94	106	89
Pre-code text	79	92	84
Comments	81	23	18
Missing exports	35	0	0
Post-code text	194	206	141
Missing files	4	0	4

**Results.** Table 7 summarizes the outcomes of the refusal-handling mechanism for GPT during code generation. Out of 144 generation attempts, **61.8%** succeeded on the first try, **29.2%** initially refused but were successfully recovered after one or more retries, and only **9.0%** remained unrecoverable after three retries. **This indicates that the retry strategy substantially improved completion stability.**

Table 8 reports aggregate cleanup operations across the three models. Gemini exhibited the highest number of cleanup events overall, particularly in *code fences* (106) and *pre-code text* (92), as well as the largest amount of residual *post-code text* (206). Although Gemini produced fewer missing files than the other models, the high number of structural artifacts indicates greater inconsistency in formatting and code boundary control.

Claude showed similar artifact diversity but slightly lower total counts. It generated 35 *missing export statements* which is not something observed in Gemini or GPT.

GPT produced the cleanest outputs overall, with fewer *code fences* (89), minimal *comments* (18), and relatively low *pre-code text* (84). However, it occasionally omitted project files (4 cases), similar to Claude.

In summary, most detected issues were surface-level artifacts such as code fences, misplaced text, and missing exports, all of which were deterministic and easily corrected through automated cleanup. These results suggest that **generation failures in multimodal pipelines largely stem from predictable formatting inconsistencies.** Lightweight post-processing therefore provides a reliable and low-cost means of standardizing outputs without requiring manual intervention.

**RQ3 Summary:** Across all models, most reliability issues stemmed from GPT refusals and syntax artifacts produced during code generation. The automated retry mechanism successfully recovered 29% of initially failed GPT generations, substantially improving completion stability. Post-processing analysis revealed that generation failures in multimodal pipelines largely stem from predictable formatting inconsistencies. Together, these results show that lightweight recovery and repair strategies, such as automated retries and pattern-based cleanup, can effectively stabilize multimodal code generation pipelines with minimal cost overhead.

## 6 Threats to Validity

We identify several internal and external factors that may threaten the validity of our results. For each, we discuss the potential impact and the mitigation strategies we applied.

### 6.1 Internal Validity

**Judge Agent Bias.** Our evaluation relies on LLM-based Judge agents to assess functional coverage and visual fidelity. This introduces a threat of judgment bias or hallucination, as models may misinterpret subtle behaviors or visual details. Such bias could overestimate or underestimate actual quality. To mitigate this, we used fixed prompts across all judges, ensured that no model evaluated its own outputs, and conducted a manual audit on 180 random cases to measure inter-rater reliability using Cohen’s  $\kappa$ . Agreement scores ranged from 0.55 to 0.90, indicating substantial to almost-perfect consistency.

**Prompt Sensitivity.** Prompt phrasing and ordering can strongly influence LLM behavior, potentially affecting generation quality or validation accuracy. This variability could lead to unintentional prompt-induced bias rather than true architectural differences. We addressed this by standardizing all prompts, freezing model versions, and releasing full templates in the replication package [53] to ensure reproducibility.

### 6.2 External Validity

**Dataset Scope.** All experiments were conducted on four open-source Figma projects (75 user stories). While these projects include diverse interaction types and UI complexities, their open-source nature could limit generalization to large-scale or proprietary enterprise systems. However, we selected mature, production-grade OSS projects to ensure realistic code structures and design fidelity.

**LLM Version Dependence.** Our findings reflect the behavior of specific LLM checkpoints (GPT-4o, Claude 4.5, Gemini 1.5). Model updates or retraining may alter generation or judgment consistency, which could affect replication. To mitigate this, we fixed all API versions during data collection and document model identifiers and parameters in the replication package for reproducibility.

**Frontend-Only Evaluation.** The framework evaluates only front-end React code generation, assuming no backend or data-binding layers. This constraint may limit generalization to full-stack applications with dynamic data, authentication, or complex state management. However, front-end generation remains a representative and high-impact subtask that directly tests multimodal reasoning and UI fidelity.

## 6.3 Construct Validity

**Evaluation Metrics.** Our metrics are based on categorical match labels (full match, partial, fail), which provide interpretable summaries but may overlook finer-grained quality distinctions. This could underestimate near-correct outputs or visual subtleties. To reduce this limitation, we supplemented automatic labels with manual verification.

## 7 Conclusion

This study investigated how multi-agent LLM architectures generate front-end applications grounded in both user stories and Figma designs. Across three coordination strategies (Supervisor (tool-calling), Hierarchical, and Custom) results show that all architectures achieve comparable quality, with average full-match rates of 54.1% for functionality and 57.9% for visual fidelity. When including partial matches, over 75% of generated outputs were functionally and visually usable, demonstrating that multimodal grounding enables strong alignment between design intent and implementation.

Architectural choice had limited impact on generation quality but substantial impact on cost. The Custom architecture consistently provided the best cost–performance ratio, reducing token use by 21–65% compared to more complex designs without quality loss. Judges were the dominant cost drivers, consuming on average 5.9× more tokens than generators, emphasizing the need for efficient evaluation workflows.

We also identified and mitigated key failure modes. GPT occasionally refused complex generation prompts, though 91% of these cases were resolved automatically through a retry mechanism. Most remaining issues stemmed from predictable structural artifacts such as missing exports or invalid syntax. These were automatically repaired through a lightweight regex-based cleanup pipeline, restoring compilable, runnable code in nearly all cases.

Overall, our findings highlight that streamlined, deterministic coordination achieves the best balance of quality, efficiency, and reliability. Increased architectural complexity offers little benefit while substantially increasing token and image processing costs.

## References

- [1] Pekka Abrahamsson, Outi Salo, Jussi Ronkainen, and Juhani Warsta. 2017. Agile software development methods: Review and analysis. *arXiv preprint arXiv:1709.08439* (2017).
- [2] Vellum AI. 2025. LLM Leaderboard 2025 – Vellum AI. <https://www.vellum.ai/llm-leaderboard>. Accessed: 2025-10-19.
- [3] Samar Al-Saqqa, Samer Sawalha, and Hiba AbdelNabi. 2020. Agile software development: Methodologies and trends. *International Journal of Interactive Mobile Technologies* 14, 11 (2020).
- [4] Anthropic. 2025. Introducing Claude Sonnet 4.5. *Anthropic News*. <https://www.anthropic.com/news/claude-sonnet-4-5> Details Claude Sonnet 4.5’s improvements in reasoning, code generation, and context length..
- [5] Tony Beltramelli. 2018. pix2code: Generating code from a graphical user interface screenshot. In *Proceedings of the ACM SIGCHI symposium on engineering interactive computing systems*. 1–6.
- [6] Bolt.new. 2025. *AI-powered UI Builder*. <https://www.bolt.new/>
- [7] Builder.io. 2025. Builder.io: Visual Headless CMS for React and More. <https://www.builder.io/>. Accessed: 2025-06-22.
- [8] Thomas Chau and Frank Maurer. 2004. Knowledge sharing in agile software teams. In *Logic versus approximation: essays dedicated to Michael M. Richter on the occasion of his 65th birthday*. Springer, 173–183.
- [9] Chunyang Chen, Ting Su, Guozhu Meng, Zhenchang Xing, and Yang Liu. 2018. From ui design image to gui skeleton: a neural machine translator to bootstrap mobile gui implementation. In *Proceedings of the 40th International Conference on*

- Software Engineering*. 665–676.
- [10] Yunnong Chen, Shixian Ding, YingYing Zhang, Wenkai Chen, Jinzhou Du, Lingyun Sun, and Lixiang Chen. 2025. DesignCoder: Hierarchy-Aware and Self-Correcting UI Code Generation with Large Language Models. *arXiv preprint arXiv:2506.13663* (2025).
  - [11] Mrs Rupali M Chopade and Nikhil S Dhavase. 2017. Agile software development: Positive and negative user stories. In *2017 2nd International Conference for Convergence in Technology (I2CT)*. IEEE, 297–299.
  - [12] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20, 1 (1960), 37–46.
  - [13] Alan Cooper, Robert Reimann, David Cronin, and Christopher Noessel. 2014. *About face: the essentials of interaction design*. John Wiley & Sons.
  - [14] Figma Design. 2017. Figma: the collaborative interface design tool.(2017). Retrieved September 17 (2017), 2017.
  - [15] Yihong Dong, Xue Jiang, Jiaru Qian, Tian Wang, Kechi Zhang, Zhi Jin, and Ge Li. 2025. A survey on code generation with llm-based agents. *arXiv preprint arXiv:2508.00083* (2025).
  - [16] Figma, Inc. 2016. Figma: Collaborative Interface Design Tool. <https://www.figma.com/>. Accessed: 2025-08-15.
  - [17] Gelilaa and Contributors. 2024. Vox-Box. <https://github.com/gelilaa/VoxBox>. Accessed: 2025-10-19, 2 stars, 34 user stories.
  - [18] Tim Goalen. 2024. Transcriber Frontend. <https://github.com/timgoalen/transcriber-frontend>. Accessed: 2025-10-19, 3 stars, 23 user stories.
  - [19] Google DeepMind. 2025. Gemini 2.5: Our most intelligent AI model—reasoning, coding, multimodal. *Google Blog*. <https://blog.google/technology/google-deepmind/gemini-model-thinking-updates-march-2025/> Describes Gemini 2.5 Pro’s reasoning, coding, and long-context capabilities..
  - [20] GSG-G9. 2024. House Hunting App. <https://github.com/GSG-G9/house-hunting-app>. Accessed: 2025-10-19, 2 stars, 9 user stories.
  - [21] GSG-K3. 2024. Urban Calendar. <https://github.com/GSG-K3/urban-calendar>. Accessed: 2025-10-19, 3 stars, 9 user stories.
  - [22] Yi Gui, Zhen Li, Yao Wan, Yemin Shi, Hongyu Zhang, Bohua Chen, Yi Su, Dongping Chen, Siyuan Wu, Xing Zhou, et al. 2025. Webcode2m: A real-world dataset for code generation from webpage designs. In *Proceedings of the ACM on Web Conference 2025*. 1834–1845.
  - [23] Yi Gui, Yao Wan, Zhen Li, Zhongyi Zhang, Dongping Chen, Hongyu Zhang, Yi Su, Bohua Chen, Xing Zhou, Wenbin Jiang, et al. 2025. UICoPilot: Automating UI synthesis via hierarchical code generation from webpage designs. In *Proceedings of the ACM on Web Conference 2025*. 1846–1855.
  - [24] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zizuan Lin, et al. 2023. MetaGPT: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*.
  - [25] Tianyi Huang. 2024. FEAD: Figma-Enhanced App Design Framework for Improving UI/UX in Educational App Development. *arXiv preprint arXiv:2412.06793* (2024).
  - [26] Juyong Jiang, Fan Wang, Jiashi Shen, Sungju Kim, and Sunghun Kim. 2024. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515* (2024).
  - [27] Yilei Jiang, Yaozhi Zheng, Yuxuan Wan, Jiaming Han, Qunzhong Wang, Michael R Lyu, and Xiangyu Yue. 2025. ScreenCoder: Advancing Visual-to-Code Generation for Front-End Automation via Modular Multimodal Agents. *arXiv preprint arXiv:2507.22827* (2025).
  - [28] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world GitHub Issues?. In *The Twelfth International Conference on Learning Representations (ICLR 2024)*. <https://openreview.net/forum?id=VTF8yNQM66>
  - [29] Kristian Kolthoff, Felix Kretzer, Christian Bartelt, Alexander Maedche, and Simone Paolo Ponzetto. 2025. GUIDE: LLM-Driven GUI Generation Decomposition for Automated Prototyping. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 1–4.
  - [30] Felix Kretzer, Kristian Kolthoff, Christian Bartelt, Simone Paolo Ponzetto, and Alexander Maedche. 2025. Closing the loop between user stories and GUI prototypes: an LLM-based assistant for cross-functional integration in software development. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*. 1–19.
  - [31] LangChain AI. 2025. *LangGraph*. <https://langchain-ai.github.io/langgraph/> Low-level orchestration framework for long-running, stateful agents.
  - [32] LangChain AI. 2025. *LangGraph Custom*. [https://langchain-ai.github.io/langgraph/concepts/multi\\_agent/#custom-multi-agent-workflow](https://langchain-ai.github.io/langgraph/concepts/multi_agent/#custom-multi-agent-workflow) Low-level orchestration framework for long-running, stateful agents.
  - [33] LangChain AI. 2025. *LangGraph hierarchical*. [https://langchain-ai.github.io/langgraph/concepts/multi\\_agent/#hierarchical](https://langchain-ai.github.io/langgraph/concepts/multi_agent/#hierarchical) Low-level orchestration framework for long-running, stateful agents.
  - [34] LangChain AI. 2025. *LangGraph Multi-Agent Systems*. [https://langchain-ai.github.io/langgraph/concepts/multi\\_agent/](https://langchain-ai.github.io/langgraph/concepts/multi_agent/) Low-level orchestration framework for long-running, stateful agents.
  - [35] LangChain AI. 2025. *LangGraph Network*. [https://langchain-ai.github.io/langgraph/concepts/multi\\_agent/#network](https://langchain-ai.github.io/langgraph/concepts/multi_agent/#network) Low-level orchestration framework for long-running, stateful agents.
  - [36] LangChain AI. 2025. *LangGraph Supervisor*. [https://langchain-ai.github.io/langgraph/concepts/multi\\_agent/#supervisor](https://langchain-ai.github.io/langgraph/concepts/multi_agent/#supervisor) Low-level orchestration framework for long-running, stateful agents.
  - [37] LangChain AI. 2025. *LangGraph Supervisor Tool Calling*. [https://langchain-ai.github.io/langgraph/concepts/multi\\_agent/#supervisor-tool-calling](https://langchain-ai.github.io/langgraph/concepts/multi_agent/#supervisor-tool-calling) Low-level orchestration framework for long-running, stateful agents.
  - [38] Dawei Li, Renliang Sun, Yue Huang, Ming Zhong, Bohan Jiang, Jiawei Han, Xiangliang Zhang, Wei Wang, and Huan Liu. 2025. Preference leakage: A contamination problem in llm-as-a-judge. *arXiv preprint arXiv:2502.01534* (2025).
  - [39] Dongyang Liu, Shitian Zhao, Le Zhuo, Weifeng Lin, Yi Xin, Xinyue Li, Qi Qin, Yu Qiao, Hongsheng Li, and Peng Gao. 2024. Lumina-mgpt: Illuminate flexible photorealistic text-to-image generation with multimodal generative pretraining. *arXiv preprint arXiv:2408.02657* (2024).
  - [40] Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. 2023. Agentbench: Evaluating llms as agents. *arXiv preprint arXiv:2308.03688* (2023).
  - [41] Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. 2023. G-eval: NLG evaluation using gpt-4 with better human alignment. *arXiv preprint arXiv:2303.16634* (2023).
  - [42] Localy.ai. 2021. Convert Figma Designs to Code. <https://www.localy.ai/>. Accessed: 2025-06-22.
  - [43] Anna Beatriz Marques, Alex Felipe Costa, Ismayle Santos, and Rossana Andrade. 2022. Enriching user stories with usability features in a remote agile project: a case study. In *Proceedings of the XXI Brazilian Symposium on Software Quality*. 1–10.
  - [44] Mahmoud Mohammadi, Yipeng Li, Jane Lo, and Wendy Yip. 2025. Evaluation and benchmarking of llm agents: A survey. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2*. 6129–6139.
  - [45] Kevin Moran, Carlos Bernal-Cárdenas, Michael Curcio, Richard Bonett, and Denys Poshyvanyk. 2020. Machine Learning-Based Prototyping of Graphical User Interfaces for Mobile Apps. *IEEE Transactions on Software Engineering* 46, 2 (2020), 196–221. doi:10.1109/TSE.2018.2844788
  - [46] Oksana Nikiforova, Kristaps Babris, and Farshad Mahmoudifar. 2024. Automated generation of Web application front-end components from user interface Mock-ups. In *Proceedings of International Conference on Software Technologies*, Vol. 1. 100–111.
  - [47] OpenAI. 2024. Hello GPT-4o. *OpenAI Blog*. <https://openai.com/index/hello-gpt-4o/> Introduces GPT-4o’s multimodal capabilities and unified text–vision–audio reasoning..
  - [48] Arjun Panickssery, Samuel Bowman, and Shi Feng. 2024. Llm evaluators recognize and favor their own generations. *Advances in Neural Information Processing Systems* 37 (2024), 68772–68802.
  - [49] Joon Sung Park, Joseph O’Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. 2023. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th annual acm symposium on user interface software and technology*. 1–22.
  - [50] Savvas Petridis, Michael Terry, and Carrie Jun Cai. 2023. PromptinFuser: Bringing user interface mock-ups to life with large language models. In *Extended Abstracts of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–6.
  - [51] Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, et al. 2023. Chatdev: Communicative agents for software development. *arXiv preprint arXiv:2307.07924* (2023).
  - [52] José Matías Rivero, Julián Grigera, Gustavo Rossi, Esteban Robles Luna, Francisco Montero, and Martin Gaedke. 2014. Mockup-driven development: providing agile support for model-driven web engineering. *Information and Software Technology* 56, 6 (2014), 670–687.
  - [53] Caren Rizk. 2025. *Bridging Design and Implementation: A Study of Multi-Agent LLM Architectures for Automated Front-End Generation*. doi:10.5281/zenodo.17429375
  - [54] Jenifer Tidwell. 2010. *Designing interfaces: Patterns for effective interaction design*. " O’Reilly Media, Inc."
  - [55] Weixi Tong and Tianyi Zhang. 2024. Codejudge: Evaluating code generation with large language models. *arXiv preprint arXiv:2410.02184* (2024).
  - [56] Yuxuan Wan, Chaozheng Wang, Yi Dong, Wenxuan Wang, Shuqing Li, Yintong Huo, and Michael R Lyu. 2024. Automatically generating UI code from screenshot: A divide-and-conquer-based approach. *arXiv preprint arXiv:2406.16386* (2024).
  - [57] Bingyang Wei. 2024. Requirements are all you need: From requirements to code with llms. In *2024 IEEE 32nd International Requirements Engineering Conference (RE)*. IEEE, 416–422.
  - [58] Jason Wu, Eldon Schoop, Alan Leung, Titus Barik, Jeffrey P Bigham, and Jeffrey Nichols. 2024. Uicoder: Finetuning large language models to generate user interface code through automated feedback. *arXiv preprint arXiv:2406.07739* (2024).
  - [59] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, et al. 2024. Autogen: Enabling

- next-gen LLM applications via multi-agent conversations. In *First Conference on Language Modeling*.
- [60] John Yang, Carlos E. Jimenez, Alex L. Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R. Narasimhan, Diyi Yang, Sida I. Wang, and Ofir Press. 2025. SWE-bench Multimodal: Do AI Systems Generalize to Visual Software Domains?. In *The Thirteenth International Conference on Learning Representations (ICLR 2025)*. <https://openreview.net/forum?id=riTiq3i21b>
- [61] Ting Zhou, Yanjie Zhao, Xinyi Hou, Xiaoyu Sun, Kai Chen, and Haoyu Wang. 2024. Bridging design and development with automated declarative ui code generation. *arXiv preprint arXiv:2409.11667* (2024).