Bridging Design and Implementation:
A Multimodal Multi-Agent Framework for Automated Front-End Generation

Caren Rizk

A Thesis

In the Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of

Master of Applied Science (Software Engineering)

at Concordia University

Montréal, Québec, Canada

December 2025

© Caren Rizk, 2025

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: Caren Rizk

Entitled: Bridging Design and Implementation : A Multimodal Multi-Agent Framework for Automated Front-End Generation

and submitted in partial fulfillment of the requirements for the degree of

Master of Software Engineering

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

____________________ Chair
Dr. Juergen Rilling

____________________ Examiner
Dr. Juergen Rilling

____________________ Examiner
Dr. Anil Ufuk Batmaz

____________________ Thesis Supervisor(s)
Dr. Emad Shihab

____________________ Thesis Supervisor(s)

Approved by ____________________
Dr. Denis Pankratov Chair of Department or Graduate Program Director

____
Dr. Mourad Debbabi Dean, Gina Cody School of Engineering and Computer Science

Abstract

Bridging Design and Implementation: A Multimodal Multi-Agent Framework for Automated Front-End Generation

Caren Rizk

Concordia University, 2025

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.

Acknowledgements

I would like to begin by expressing my sincere gratitude to my supervisor, **Dr. Emad Shihab**. His mentorship not only shaped my technical thinking but also encouraged me to believe in my capabilities during moments of doubt. I will always appreciate his reminders to stay calm and focused, and his advice has left a lasting impact on the way I approach challenges.

I am also deeply thankful to **Dr. SayedHassan Khatoonabadi** for his invaluable help, patience, and many discussions that guided me through the complexities of this thesis. His dedication and clarity made a tremendous difference.

My appreciation extends to the **NSERC CREATE program** for supporting my academic journey, and to our coordinator, **Lori**, for her kindness, reliability, and continuous assistance.

Finally, I want to thank my friends and colleagues - **Jasmine, Gideon, Chaima, Samuel, and Alor** for their support, motivation, and for sharing this journey with me, both academically and personally. Your presence made this experience meaningful and memorable.

And as my supervisor always says – keep it easy.

Dedication

This work is dedicated to my family – to my parents, **Adnan and Razan** – this thesis would not have been possible without your love and support. Thank you for always encouraging me to explore new things, to dream bigger, and to never be afraid of difficult paths. Your support during every tough moment-and your belief that I could succeed even when I doubted myself is the reason I had the courage to begin this Master's program in the first place.

To my brother, **Jad**, thank you for your positivity, humor, and for reminding me to take breaks, breathe, and stay grounded.

To my husband, **Wissam**, thank you for being my anchor throughout this journey my strength, my balance, and my constant source of encouragement. Thank you for standing by me through every late-night experiment, every deadline, and every moment of stress. Your support means more than words can express.

And to **Fadi and Kinda**, my second parents and lifelong role models – thank you for your guidance, positivity, and the unwavering belief that I could reach this goal.

This thesis is for all of you.

Table of Contents

List of Figures viii

List of Tables ix

1 Introduction and Research Statement 1

1.1 Introduction 1

1.2 Research Statement 4

1.3 Thesis Overview 4

1.4 Thesis Contributions 6

1.5 Related Publications 7

1.6 Thesis Organization 7

2 Background and Related Work 9

2.1 Background 9

2.2 Related Work 10

2.3 Chapter Summary 12

3 Approach 14

3.1 Agents 15

3.2 Tools 17

3.3 Architectures 18

3.4 Experiment Setup 21

3.5 Dataset 22

4 Results **25**

4.1 RQ1: How effectively can our multi-agent framework generate React applications that align with functional and visual user story requirements? 25

4.2 RQ2: How do different agent architectures affect token consumption and code quality? 28

4.3 RQ3: What types of failures occur in the generation process, and how can they be automatically detected or repaired? 31

5 Discussion **35**

5.1 Quality of Multimodal Front-End Generation (RQ1) 35

5.2 Architectural Impact on Cost and Efficiency (RQ2) 36

5.3 Pipeline Reliability and Failure Characteristics (RQ3) 36

5.4 Synthesis 37

6 Threats **38**

6.1 Internal Validity 38

6.2 External Validity 39

6.3 Construct Validity 40

6.4 Conclusion of Threats 40

7 Conclusion **41**

7.1 Future Work 42

Bibliography **45**

Appendices **53**

A.1 Algorithms 53

A.2 Prompt Templates 59

A.3 Dataset Example 63

List of Figures

3.1 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. 16
3.2 Overview of our agents coordination in the Supervisor (tool-calling) architecture 19
3.3 Overview of our agents coordination in the Hierarchical architecture 20
3.4 Overview of our agents coordination in the Custom architecture 21

4.1 Aggregate JSX cleanup operations by issue type and model family. Post-code text and code-fence artifacts were the most frequent cleanup operations across all models. 34

7.1 Figma frame associated with Story 8. 64

List of Tables

4.1 Functional Coverage and Visual Fidelity Results by Generator-Judge Configuration (Aggregated Across All Projects and Architectures) 26

4.2 Functional Coverage and Visual Fidelity Results by Project (Aggregated Across All Configurations and Architectures) 27

4.3 Average token consumption for Generators aggregated across all models (in millions). Lowest-cost architecture highlighted in bold. 29

4.4 Average token consumption for Judges aggregated across all models (in millions). Lowest-cost architecture highlighted in bold. 30

4.5 Functional coverage of the multi-agent framework across architectural strategies. Results show counts and percentages for user story evaluation. 30

4.6 Visual fidelity of the multi-agent framework across architectural strategies. Results show counts and percentages for Figma design alignment. 31

4.7 Summary of refusal-handling outcomes for GPT during code generation across all architectures. 33

List of Algorithms

1 Multimodal React Generation Pipeline 54
2 Figma-to-HTML Extraction Pipeline 56
3 Story-to-Component and Story-to-Figma Mapping Pipeline 57
4 Refusal Detection and Retry Strategy 58
5 JSX Cleanup Overview 58

Listings

A.1 Supervisor Builder Prompt Template 59
A.2 Enricher Prompt Template 59
A.3 Judge Prompt Template for Functional Coverage 60
A.4 Judge Prompt Template for Visual Fidelity 61
A.5 Fixer Prompt Template 62
A.6 Story-to-Component Mapping for Story 8 64
A.7 Story-to-Figma Mapping for Story 8 65
A.8 Excerpt of generated React component `ProfileUserInfo.jsx` mapped to house management stories 65

Chapter 1

Introduction and Research Statement

This chapter introduces the motivation and scope of the thesis. It outlines the challenges of generating front-end specifications (namely user stories and high-fidelity Figma designs) into executable applications. This chapter highlights recent advances in multimodal large language models that make such automation possible. The chapter defines the research problem, presents the core research questions, summarizes the thesis contributions, and provides an overview of the thesis organization to contextualize the remainder of the work.

1.1 Introduction

Modern user interface (UI) development follows a structured, multi-stakeholder process that transforms high-level business needs into functional applications [1]. In front-end workflows, stakeholders express features as user stories [2], UX designers translate these into visual mockups [3, 4], and engineers synthesize both artifacts into interactive code [5]. 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 [6, 7]. Automating this transition could significantly improve consistency and accelerate iteration cycles [8].

Recent progress in Multimodal Large Language Models (MLLMs)âĂŤmodels that can jointly process and reason over multiple data modalities such as text and imagesâĂŤhas opened new opportunities for automating such front-end tasks, including UI code generation from text or visual inputs [9]. MLLMs excel at context-rich reasoning [7, 10], yet most existing work handles only one input modality; design mockups [11, 12, 13, 14]

or textual descriptions [15, 16] 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 [17]. By isolating stages instead of modeling the full pipeline, prior systems cannot meet the functional and behavioral depth demanded by production code [9].

Moreover, there is little empirical evidence comparing LLM agent architectures on real software projects [18]. Most studies evaluate single-agent prompts or synthetic benchmarks [19, 20], leaving open questions about how coordination strategies affect quality and cost in realistic multimodal pipelines. This lack of empirical evaluation on real-world, multimodal tasks creates a gap in understanding how multi-agent architectures impact performance and efficiency in practice. Addressing this gap is critical for real-world adoption, where systems must balance output quality, scalability, and computational cost.

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 [5, 21]. We adopt Figma due to its wide use in collaborative UI/UX prototyping [22, 23]. 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 (Builder, Validator, and Fixer) performs a specialized task (layout parsing, component generation, enrichment, or validation) mirroring practical development pipelines [24]. We investigate how different architectural strategies influence scalability and output quality. Specifically, we compare Supervisor (tool-calling), Hierarchical, and Custom architectural patterns [25] 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. Answering these research questions is critical for determining whether multi-agent LLM systems can reliably generate front-end applications, how architectural choices impact quality and cost, and whether such systems are viable for real-world deployment. 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. We evaluate performance using two metrics: *functional coverage*, which measures whether the generated application correctly implements user story behaviors, and *visual fidelity*, which measures how closely the generated UI matches the corresponding Figma design. Each metric is assessed using categorical labels (*full match*, *partial match*, *fail*). 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 results are notable given the difficulty of jointly satisfying both functional and visual requirements in multimodal settings, which prior approaches typically do not address.

RQ2: How do different agent architectures affect token consumption and code quality? We compare three architectural strategies: *Supervisor (tool-calling)*, where a single controller dynamically invokes agent tools; *Hierarchical*, where a top-level controller delegates tasks to specialized sub-controllers; and *Custom*, a deterministic pipeline with predefined execution flow. All architectures are evaluated 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: the Custom architecture reduces token usage by 21–65% compared to the Hierarchical and Supervisor (tool-calling) configurations, without loss in output quality.

Furthermore, evaluation costs dominate overall computation: Judge agents consume on average 5.9× more tokens than generator agents, indicating that validation is the primary cost driver in the pipeline.

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.

1.2 Research Statement

Motivated by the growing complexity of modern front-end development workflows and the persistent gap between design intent, user-story requirements, and implementation the goal of this MASc thesis is to investigate whether multi-agent large language models (LLMs) can reliably automate the generation of functional and visually faithful front-end applications. We state our research statement as follows:

> Front-end development is a multi-stage and error-prone process requiring developers to interpret heterogeneous artifacts such as user stories and high-fidelity Figma designs. This thesis systematically evaluates the ability of multi-agent LLM architectures to automate this process. Specifically, it examines whether multimodal multi-agent systems can generate React applications that satisfy both functional requirements and visual design constraints, quantifies the trade-offs across different coordination architectures, and assesses their reliability, cost-efficiency, and failure modes for integration into end-to-end front-end generation workflows.

1.3 Thesis Overview

This section outlines the structure of the thesis and summarizes the core ideas developed throughout the remaining chapters. Each chapter contributes a different perspective on the design, implementation, and evaluation of multi-agent LLM architectures for automated front-end generation, collectively forming the foundations of this research.

Chapter 2 – Background and Related Work

This chapter reviews the key literature that shapes the problem addressed in this thesis. We begin with a brief look at how front-end applications are traditionally built: requirements are expressed as user stories, designers translate them into high-fidelity Figma mockups, and developers combine both to write interactive code. This separation between behavioral and visual specifications provides important context for why automated front-end generation is challenging.

We then survey prior work across four areas: text-to-code generation, multimodal UI synthesis from screenshots and Figma designs, commercial design-to-code tools, and multi-agent LLM frameworks. Together,

these studies highlight a common gap-most systems reason over either text or visuals, but rarely both in an integrated way. This gap motivates the unified multimodal, multi-agent approach developed in this thesis.

Chapter 3 – Approach

Chapter 3 introduces the end-to-end approach developed in this thesis. We begin by explaining how the system takes the two artifacts used in real front-end development-user stories and high-fidelity Figma designs and turns them into multimodal inputs for a multi-agent LLM pipeline. The chapter provides a high-level overview of the full workflow, from extracting structure from Figma frames, to generating and enriching React components, to validating and repairing the final application.

We then describe the three agents (Builder, Validator, Fixer), the tools they rely on, and how these components interact during generation and repair. Finally, the chapter explains how the same agent set is instantiated under three architectural strategies: Supervisor, Hierarchical, and Custom implemented using LangGraph. This sets the stage for the experiments by showing how the pipeline operates and how architectural differences affect coordination, efficiency, and output quality.

Chapter 4 – Results and Analysis

This chapter presents the core empirical findings of the thesis. We analyze the functional correctness and visual fidelity of generated applications across all model pairs and architectural strategies. Quantitative results are broken down by project, model family, and architectural configuration, revealing patterns of performance variation and consistency. We evaluate the cost-efficiency of each architecture through detailed token consumption statistics and compare generator and judge models to identify the dominant cost contributors. The chapter also provides an examination of errors encountered during generation, including LLM refusals, incomplete outputs, and JSX formatting artifacts, and evaluates the effectiveness of automated repair tools in resolving these issues.

Chapter 5 – Discussion

This chapter discusses the broader implications of the experimental findings. We explore why architectural differences yield modest improvements in output accuracy but large disparities in total token consumption. We analyze the challenges inherent in multimodal reasoning, particularly when integrating visual information

from Figma with textual requirements from user stories. The chapter also reflects on the nature of system failures observed during the pipeline, the reliability of LLM-based validators, and the practical trade-offs between architectural complexity, determinism, and cost. These insights contribute to a deeper understanding of how multi-agent LLM systems behave in structured front-end generation tasks.

Chapter 6 – Threats to Validity

This chapter examines potential threats to the validity of the study. We discuss internal threats such as the reliability of LLM-based judges, prompt sensitivity, and the assumptions built into agent coordination. External validity threats are also considered, including dataset representativeness, application domain limitations, model version drift, and the generalizability of results beyond front-end-only workflows. Construct validity is analyzed in terms of evaluation criteria, categorical scoring methods, and the constraints of multimodal visual-textual assessment. Together, these considerations provide a transparent view of the limitations and boundaries of the study.

Chapter 7 – Conclusion and Future Work

This chapter synthesizes the key findings of the thesis. It highlights the feasibility of using multimodal multi-agent architectures to generate functional and visually aligned front-end applications, while also noting the significant influence of architectural design on computational cost. The chapter discusses how automated refusal recovery, code cleanup, and repair tools contribute to pipeline stability. Finally, it identifies promising directions for future work, including extending the pipeline to full-stack applications, improving visual reasoning and component detection, developing more sophisticated repair agents, and exploring adaptive or self-improving agent workflows that leverage feedback over multiple iterations.

1.4 Thesis Contributions

Contributions. This thesis makes the following contributions:

- **A multimodal multi-agent framework** that integrates Figma designs and user stories to automatically generate complete React applications through coordinated generation, validation, and repair stages.
- **A comparative evaluation of three coordination architectures**: Supervisor (tool-calling), Hierarchical,

and Custom implemented using LangGraph, revealing their trade-offs in functional accuracy, visual fidelity, token efficiency, and pipeline stability.

- **A fully constructed and validated dataset** consisting of paired user stories and corresponding Figma frames from real-world open-source projects. This includes automated GitHub retrieval, filtering, design verification, Figma-frame extraction, and manual alignment, forming a reproducible benchmark for multimodal front-end generation research. The dataset and replication materials are publicly available as part of the replication package [26].
- **A detailed failure-mode analysis** of multimodal LLM behavior, identifying refusal patterns, JSX formatting artifacts, and architecture-specific instabilities, along with lightweight repair mechanisms that substantially improve completion rates.
- **Open-source replication package** containing all scripts, prompts, extracted Figma frames, dataset artifacts, and evaluation tools to ensure full reproducibility and support future work on automated front-end generation [26].

1.5 Related Publications

Parts of this thesis have been accepted for publication in the following paper:

- **Caren Rizk**, SayedHassan Khatoonabadi, and Emad Shihab. *Bridging Design and Implementation: A Study of Multi-Agent LLM Architectures for Automated Front-End Generation*. In *Proceedings of the International Conference on Mining Software Repositories (MSR'26)*, 2026. DOI: https://doi.org/10.1145/3793302.3793371. Available at: https://das.encs.concordia.ca/pdf/rizk_MSR2026.pdf.

1.6 Thesis Organization

This thesis is organized into several chapters that collectively develop, implement, and evaluate a multimodal multi-agent framework for automated front-end generation. Chapter 2 reviews foundational literature on multimodal LLMs, front-end generation, and multi-agent orchestration, establishing the context for this work. Chapter 3 presents the design of our framework, detailing the agents, tools, architectures, dataset construction, and experimental setup used throughout the study. Chapter 4 reports our empirical findings,

including functional accuracy, visual fidelity, token efficiency, and failure-mode analysis across models and architectures. Chapter 5 provides an in-depth discussion of the broader implications of these findings, focusing on architectural trade-offs, multimodal reasoning challenges, and system reliability. Chapter 6 examines threats to validity, addressing limitations related to evaluation reliability, dataset generalizability, and model behavior. Finally, Chapter 7 summarizes the key contributions of the thesis and outlines promising directions for future research.

Chapter 2

Background and Related Work

This chapter provides the conceptual and empirical foundation for the thesis. It first reviews the traditional front-end development workflow and explains the complementary roles of user stories and visual design artifacts. It then surveys prior research in text-to-code generation, image-based and multimodal UI synthesis, commercial design-to-code systems, and multi-agent LLM frameworks. By synthesizing these areas, the chapter identifies key gaps in existing work that motivate the multimodal, multi-agent approach proposed in this thesis.

2.1 Background

Front-end development follows a multi-stage workflow that integrates contributions from product managers, User Experience (UX) designers, and software engineers. This process typically begins with requirements engineering, where stakeholders articulate features as user stories (structured narratives that capture functionality from an end-user perspective e.g., "As a [user type], I want [feature] so that [benefit]") [2, 27]. User stories serve as behavioral specifications that communicate goals, constraints, and expected interactions in a developer-friendly format [28, 29].

UX designers then interpret these stories into high-fidelity visual mockups using tools such as Figma [11, 30, 31]. These mockups encode layout hierarchies, styling rules, interaction flows, and visual semantics that guide the structure and appearance of the final interface [3, 5]. Together, textual user stories and visual designs form complementary specifications: the former define what the system should do, while the latter define how it should look and behave.

Front-end engineers manually synthesize these heterogeneous artifacts into interactive code, wiring visual components to user-story-driven logic, state management, and routing [5]. Despite its structure, this workflow remains labor-intensive and error-prone, with misalignments frequently arising between requirements, design intent, and implemented behavior [32]. Automating this transformation from narrative requirements and design artifacts to executable front-end code has therefore been a long-standing goal in both software engineering and human-computer interaction.

Earlier attempts to automate User Experience (UI) generation relied on rule-based systems, model-driven engineering, and image-to-layout extraction [33, 14, 34]. More recent advances in large language models (LLMs) and multimodal models have revitalized this area by enabling systems that jointly reason over natural language, images, and source code [35, 36, 37]. These models have been used for text-to-code generation, image-based UI reconstruction, and hybrid multimodal pipelines that bridge structural, visual, and behavioral specifications.

Parallel to these developments, multi-agent LLM frameworks such as AutoGen [38], ChatDev [39], MetaGPT [40], and LangGraph [41] have introduced structured approaches to decomposing complex software tasks. These systems create specialized agents (e.g., planner, designer, coder, tester) and coordinate them through interaction patterns including supervisor-worker hierarchies, tool-supervised flows, and reactive graph-based control [42]. LangGraph, in particular, supports multiple orchestration strategies: (i) flat/reactive peer-to-peer flows, (ii) hierarchical supervisor-worker delegation, and (iii) tool-supervised execution with explicit action schemas. These patterns can be combined with features such as interrupts, human-in-the-loop overrides, and persistent memory to create robust, repeatable workflows. In our setting, this flexibility enables us to wire the same agent set into different architectural topologies and empirically isolate how control-flow design affects implementation completeness, visual fidelity, failure recovery, and token efficiency.

Taken together, these threads form the foundation for modern research on multimodal front-end code generation. The next section reviews prior work on UI generation from text, visuals, and combined multimodal inputs, highlighting limitations that motivate our multi-agent approach.

2.2 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.

2.2.1 Text-to-Code from Requirements

Text-to-code generation has advanced rapidly with models such as Codex and GPT [43], which synthesize UI logic directly from high-level natural language specifications [44]. GUIDE [45] uses retrieval-augmented prompting to decompose textual goals into UI components, while Kretzer et al. [46] 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.

2.2.2 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. [33] 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 [47] introduces a modular, three-stage process (grounding, planning, generation) to improve layout accuracy and interpretability. DesignCoder [48] adopts a hierarchy-aware, self-correcting pipeline to boost structural fidelity, while UICopilot [49] 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.

2.2.3 Commercial and Industrial Systems

Commercial tools like Builder.io [50] and Locofy.ai [51] 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 [52] 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.

2.2.4 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 [40] simulates a software company with role-based agents, AutoGen [38] supports conversational collaboration via tool use, and ChatDev [53] 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.

2.3 Chapter Summary

This chapter surveyed the major research areas that inform automated front-end generation using large language models. We reviewed prior work in text-to-code generation, highlighting approaches that translate natural language requirements into UI logic but often lack mechanisms for ensuring visual fidelity. We then examined image-based and multimodal UI generation, noting recent advances that leverage visual grounding to produce layout-accurate code, while still struggling to incorporate functional behaviors. Commercial design-to-code tools were discussed to illustrate the practical landscape and the limitations of proprietary, non-

transparent pipelines. Finally, we explored multi-agent LLM orchestration frameworks, which demonstrate the promise of role-specialized collaboration but are rarely evaluated on real-world, multimodal front-end tasks.

Together, these areas reveal a gap in existing literature: few systems jointly reason over *both* user stories and high-fidelity Figma designs, and even fewer evaluate how different multi-agent architectures affect code quality, visual alignment, cost efficiency, and reliability. This thesis addresses this gap by developing and empirically evaluating a multimodal multi-agent framework capable of generating, validating, and repairing complete React applications across a curated dataset of real-world projects.

Chapter 3

Approach

The goal of this thesis is to evaluate how different multi-agent LLM architectures generate, validate, and repair front-end applications when provided with two complementary inputs: textual user stories and high-fidelity Figma designs. Our approach follows a structured, end-to-end pipeline that mirrors real-world front-end development workflows shown in Figure 3.1. Before detailing the individual agents 3.1, tools 3.2, and architectural variations 3.3, we provide a high-level overview of the full multimodal system.

This design is motivated by the need to decompose complex multimodal reasoning into structured stages that isolate visual generation, functional enrichment, and validation, improving determinism, interpretability, and error recovery.

The pipeline begins by extracting structural and visual information from Figma designs using a custom-built rule-based extractor that converts each design frame into inline-style HTML and a corresponding screenshot. In parallel, user stories are processed to identify the functional requirements and map them to the appropriate UI components. These two specification sources (visual layout (HTML + screenshot) and textual behavior (user stories)) form the multimodal inputs for the rest of the system.

The next stage is code generation. A Builder agent synthesizes React components that visually match the extracted HTML structure while maintaining a modular project layout. This stage focuses on structural fidelity (component hierarchy, styling, and layout). This mirrors the first pass a human front-end engineer would complete before adding behavior.

Once structural scaffolding is produced, a Story Mapper identifies which components correspond to which user stories. The Enricher then injects functional logic such as state management, routing, form handling, and event processing into the relevant components. This decomposed workflow of visual scaffolding first,

functional enrichment second, reduces hallucination, improves determinism, and limits token overhead by localizing reasoning.

After generation, the Validator agent assesses each component along two dimensions: (1) functional coverage, based on the extent to which the enriched components satisfy user-story requirements, and (2) visual fidelity, measuring alignment between the generated React code and the original Figma design. Validation uses a structured rubric enforced through a dedicated LLM-as-a-judge model to ensure consistent scoring.

Finally, the Fixer agent closes the loop by applying targeted repairs based on Validator feedback. This includes adjusting missing behaviors, correcting JSX artifacts, and refining layout elements. The system allows up to two validation-repair cycles to avoid oscillation while maximizing improvement. The entire pipeline is executed under three architectural strategies: Supervisor (tool-calling), Hierarchical, and Custom implemented using LangGraph. These architectures differ only in control-flow and coordination logic; the underlying agents and tools remain identical. This design allows us to isolate the effects of orchestration strategy on accuracy, token consumption, and pipeline stability.

To evaluate the effectiveness of the generated applications, we define two metrics: *functional coverage*, which measures the extent to which generated components satisfy user story requirements, and *visual fidelity*, which measures alignment between the generated UI and the original Figma design. Both metrics are assessed using categorical labels (*full match*, *partial match*, *fail*), enabling consistent evaluation across architectures and model configurations.

Together, these stages form a multimodal, multi-agent workflow capable of generating dynamic and visually grounded front-end applications. The remainder of this chapter details the specific agents, tools, datasets, and architectural strategies that operationalize this approach.

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 [26].

- **Builder Agent**: is responsible for converting paired inputs from the Figma design and user stories

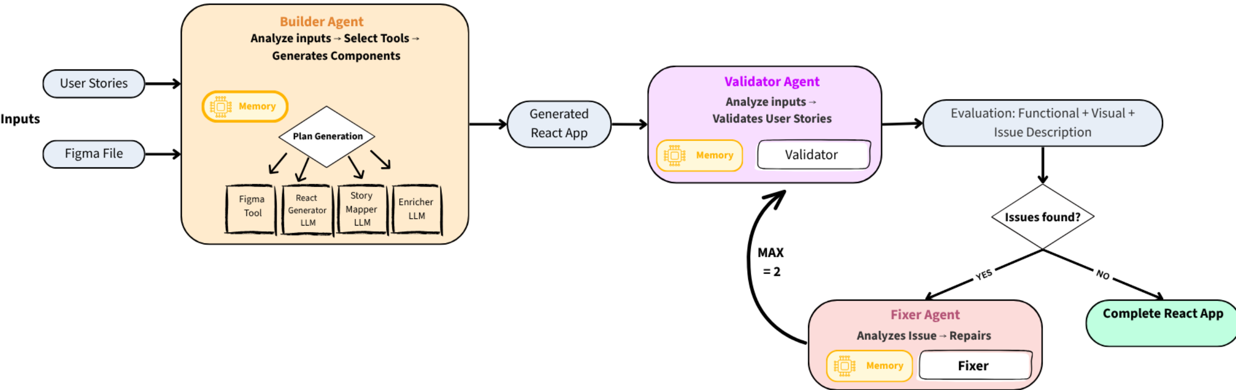

Figure 3.1: Multi-agent framework for React application generation with autonomous agents (Builder, Validator, Fixer) that use memory and decision-making to coordinate tool usage and iteratively refine code quality.

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 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 [26]. 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.

3.2 Tools

In this context, a *tool* refers to a callable moduleâĂŤeither LLM-based or rule-basedâĂŤthat performs a well-defined task (e.g., generation, mapping, validation) and is invoked by agents as part of the pipeline execution.

To ensure a fair comparison across architectures, we define a set of standardized tools for our framework. These tools encapsulate the core functional operations used throughout the pipeline, allowing different architectural strategies to coordinate the same underlying components. In Section 3.3, we explain how these tools are orchestrated 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 Architectures

To orchestrate the full lifecycle of code generation from visual design and textual specifications, we adopt LangGraph [54] 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* [25]. Each offers distinct trade-offs in coordination, flexibility, and reproducibility that will be discussed below.

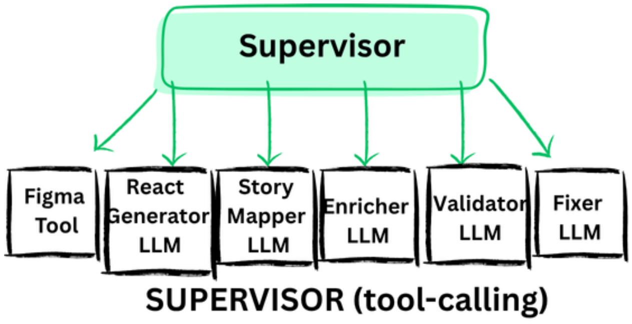

Figure 3.2: Overview of our agents coordination in the Supervisor (tool-calling) architecture

3.3.1 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 [55, 42, 38]. 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. Specifically, in the tool-calling variant, agent actions are represented as structured function calls with predefined schemas rather than free-form natural language. This reduces ambiguity in decision-making, constrains the output space, and ensures that similar inputs lead to consistent execution paths. As a result, the system exhibits more deterministic behavior and produces reproducible execution traces across runs.

3.3.2 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 [56]. 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 3.2).

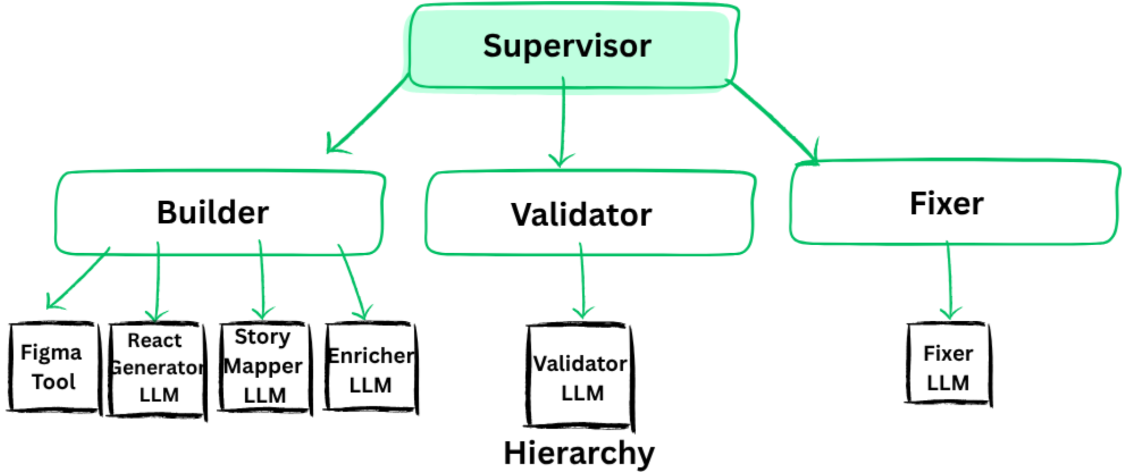

Figure 3.3: Overview of our agents coordination in the Hierarchical architecture

3.3.3 Hierarchical

A multi-level architecture in which a top-level supervisor delegates subtasks to intermediate supervisors, each managing its own specialized agents. This tree-based organization localizes context, shortens prompts, and scales coordination, making it well-suited for complex workflows [57]. 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 3.3).

3.3.4 Network

The Network pattern connects agents as a general graph with free-form communication and no fixed sequence or hierarchy. LangGraph recommends this design for problems that lack clear dependency structure or execution order [58]. Since our workflow follows a strict dependency chain (*Build* $\rightarrow$ *Validate* $\rightarrow$ *Fix*), adopting the Network pattern would introduce unnecessary routing complexity and undermine determinism. Therefore, it was excluded from our evaluation.

3.3.5 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

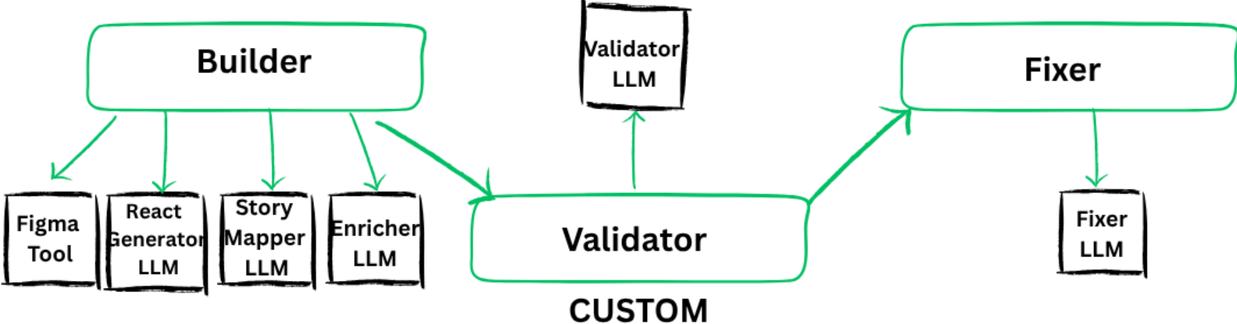

Figure 3.4: Overview of our agents coordination in the Custom architecture

them are deterministic, allowing precise measurement of cost, performance, and agent interactions [59]. 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 3.4).

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.

3.4 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 [43]; Gemini-2.5-Pro provides competitive multimodal reasoning and enhanced coding [60]; and Claude-Sonnet-4.5 excels at nuanced code generation and detailed adherence to specifications [61]. These model families also dominate leading public and research leaderboards: for example, the Vellum AI Leaderboard ranks Gemini 2.5 Pro, Claude 4 Sonnet, and GPT-4o among the top models for reasoning and

coding tasks [62]; 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 [63, 64]. 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 [65, 66] 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. 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 [67, 68]. 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 [26].

3.5 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 [69]: a volunteer coordination platform.
- Transcriber [70]: an audio transcription interface.
- Urban-Calendar [71]: a collaborative event-scheduling app.
- House-Hunting [72]: a property listing interface.

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 us to provide descriptive statistics across architectures, assessing consistency and variance in functional correctness, visual fidelity, and cost efficiency under diverse architectural strategies.

While our dataset contains 75 paired user stories and Figma frames which is modest in size compared to large synthetic UI datasets such as RICO (66k screens) or Screen2Code (10k samples), it fills a critical gap in current research resources. To the best of our knowledge, no existing publicly available dataset jointly provides:

- real user stories (behavioral specifications),

- real high-fidelity Figma frames (visual specifications)

This distinction is important because real-world front-end development requires the integration of both functional requirements and visual design artifacts. Without datasets that jointly capture these modalities, prior work is limited to evaluating systems on isolated tasks, preventing a realistic assessment of end-to-end front-end generation and multimodal reasoning capabilities.

Prior datasets are either image-only (e.g., RICO [73], Pix2Code [14]), text-only (e.g., natural-language-to-code benchmarks), or proprietary (e.g., commercial design-to-code systems). None integrate both modalities in a unified format. As a result, our dataset is the only multimodal UI-generation dataset that mirrors real-world development workflows by providing aligned textual and visual inputs. This unique structure enables controlled evaluation of systems that must jointly reason over design intent and functional requirements; an ability not captured by existing datasets.

Chapter 4

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.

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

4.1.1 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. Evaluating this capability is essential for determining whether such systems can be reliably used in practice, where both functional correctness and visual fidelity are required for usable applications. In this RQ, our goal is to assess how effectively the framework integrates functional and visual specifications to produce dynamic applications.

4.1.2 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 3.3. Each output was assessed along the following two dimensions:

Table 4.1: Functional Coverage and Visual Fidelity Results by Generator-Judge Configuration (Aggregated Across All Projects and Architectures)

Configuration	Functional Coverage			Visual Fidelity		
(Generator–Judge)	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%)
Total	**713 (54.1%)**	**300 (22.8%)**	**305 (23.1%)**	**585 (57.9%)**	**269 (26.6%)**	**157 (15.5%)**

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

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 3.4. 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 $\times$ 2 criteria $\times$ 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 κ [74] to assess agreement between the LLM judges and manual ratings. Visual Fidelity achieved the highest consistency, with κ 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 κ values indicate substantial to almost-perfect reliability, supporting the robustness of the evaluation rubric.**

Table 4.2: 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 due to systematic evaluation failures in GPT-Gemini and Claude-Gemini configurations.

4.1.3 Results.

Table 4.1 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)**.

Here, a `full match` indicates that all key functional or visual elements are correctly implemented, while `partial` denotes that most elements are present but some aspects are missing, incorrect, or misaligned.

These results are significant given the complexity of the task, which requires jointly satisfying both behavioral requirements and visual design constraints in a multimodal setting. In contrast to prior approaches that typically address only a single modality or simplified benchmarks, achieving over 50% full alignment demonstrates that the system can produce fully correct end-to-end outputs in a substantial portion of cases.

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 4.2. *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 accross 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.

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

4.2.1 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.

4.2.2 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 4.3 and 4.4, we averaged token usage across all models (GPT, Claude, Gemini) for each architecture. This aggregation removes model-specific variance and highlights the relative overhead

Table 4.3: 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
Custom	0.87	0.33	**1.40**

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

4.2.3 Results

Table 4.3 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 4.4 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 4.3). Averaged

Table 4.4: 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)**
Supervisor (tool-calling)	9.87	0.17	**10.20**
Hierarchical	19.47	0.33	19.93
Custom	10.33	0.17	10.77

Table 4.5: 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%)
Average	**51.4%**	**22.6%**	**26.0%**

across architectures, this is about 5.9× more overall. Judges required 10.2M-19.9M tokens per run, with Hierarchical incurring the highest cost (19.9M).

Tables 4.5 and 4.6 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.

Table 4.6: 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%)
Average	**56.2%**	**29.5%**	**14.2%**

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

4.3.1 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.

4.3.2 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):

- `i'm sorry`
- `i cannot help`
- `i can't help`
- `i'm unable to`

- `as an ai`
- `i need more information`
- `please provide more details`
- `could you please provide`
- `i need.*specific user stor(y|ies)`

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.

4.3.3 Results

Table 4.7 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

Table 4.7: 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%
Total attempts	**144**	**100%**

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.**

Figure 4.1 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.

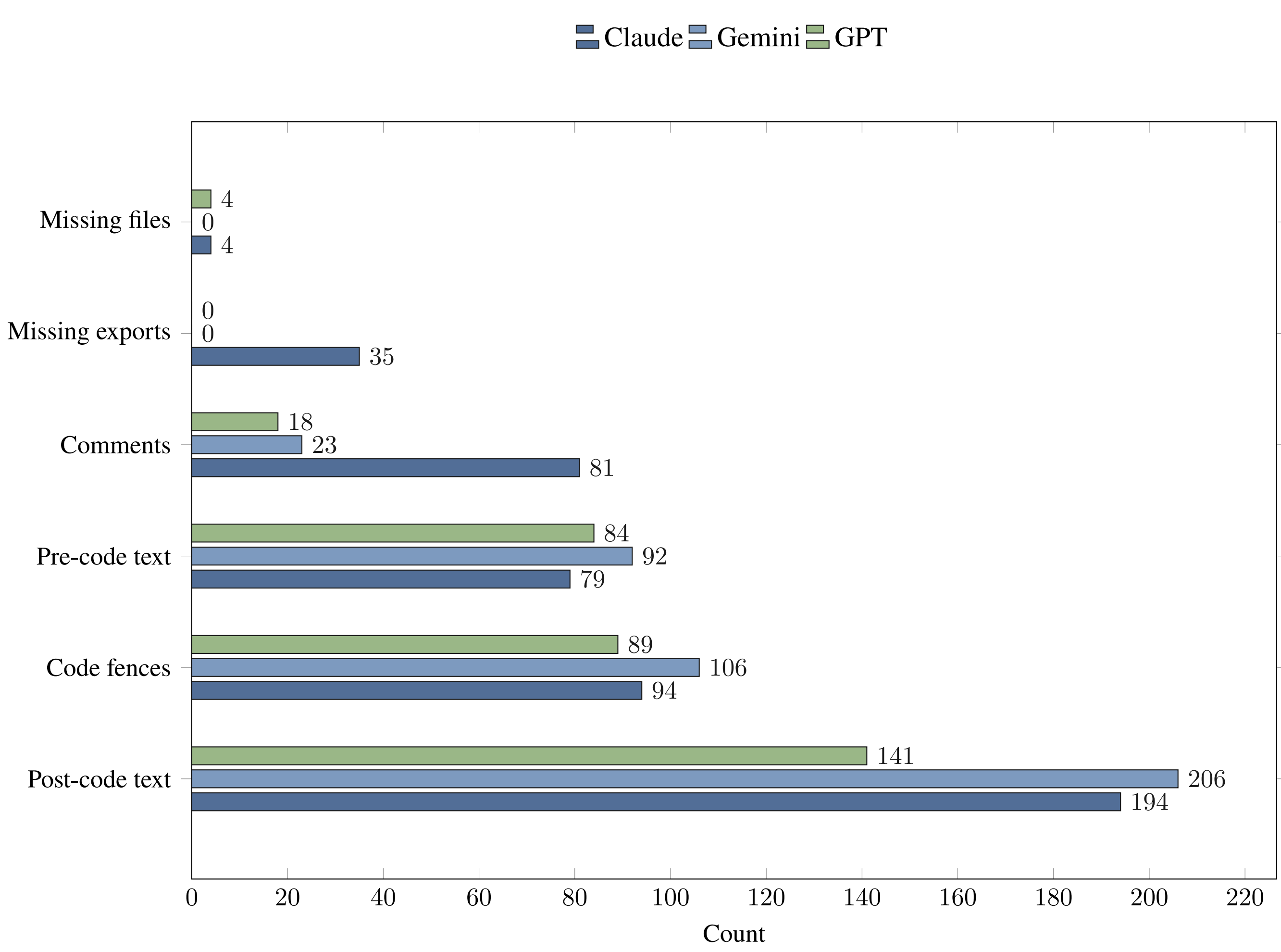

Figure 4.1: Aggregate JSX cleanup operations by issue type and model family. Post-code text and code-fence artifacts were the most frequent cleanup operations across all models.

Chapter 5

Discussion

Our evaluation across RQ1–RQ3 provides a comprehensive view of how large multimodal models behave within a multi-agent pipeline for automated front-end generation. The findings highlight not only the strengths of current models in grounding UI code in textual and visual specifications but also the bottlenecks that emerge when these systems are deployed in multi-step, multimodal workflows.

5.1 Quality of Multimodal Front-End Generation (RQ1)

Results from RQ1 demonstrate that modern multimodal models can frequently produce React applications that are both functionally correct and visually aligned with design intent. Full-match accuracy across models averaged 54.1% for functional coverage and 57.9% for visual fidelity, increasing to 76.9% and 84.9% when partial matches are included. These outcomes show that most failures are not catastrophic but instead reflect localized issues such as missing event handlers, misaligned layout elements, or incomplete styling that remain correctable through automated repair.

Variation across projects further reveals that model performance correlates with UI complexity. Applications such as *Vox-Box*, with relatively standard components and interactions, achieved the highest success rates, while domain-specific applications such as *Transcriber* saw markedly lower performance. This suggests that front-end generators generalize best to familiar interaction patterns, whereas specialized or AI-centric UIs introduce novel affordances that current models struggle to interpret. The high inter-rater agreement between LLM judges and manual audits reinforces that these differences reflect true variation in generation quality rather than inconsistencies in evaluation.

5.2 Architectural Impact on Cost and Efficiency (RQ2)

RQ2 shows that architectural coordination strategies substantially influence token consumption and overall computational cost. Although all architectures perform similar amounts of actual code generation, the Supervisor (tool-calling) and Hierarchical configurations incur substantial overhead due to repeated context passing and complex decision orchestration. In contrast, the Custom architecture achieves the lowest token usage 1.4M tokens on average, representing a 21–65% reduction compared to alternative strategies while also achieving the highest accuracy.

These findings challenge the assumption that more sophisticated or multi-layered coordination leads to higher-quality outputs. Instead, the results indicate that deterministic and streamlined orchestration can outperform more complex strategies both in cost and accuracy. Across all conditions, judges remain the dominant cost driver, consuming roughly 5.9× more tokens than generators. This suggests that future work seeking to improve scalability should focus on more efficient evaluation mechanisms, such as pruning visual inputs or compressing story-design context before judgment.

5.3 Pipeline Reliability and Failure Characteristics (RQ3)

RQ3 reveals that the reliability issues in multimodal generation pipelines stem primarily from predictable and surface-level formatting inconsistencies rather than deep reasoning failures. GPT was the only model to exhibit refusal behaviors during generation, but these were largely mitigated by an automated retry mechanism, which recovered nearly 30% of initially failed outputs.

The remaining errors such as code fences, explanatory text, missing exports, residual comments, and incomplete files were consistently detected and corrected by a lightweight JSX cleanup script. Importantly, these artifacts did not reflect misunderstandings of design intent or user-story semantics; instead, they arose from formatting tendencies common in conversational LLM interfaces. The success of simple post-processing heuristics indicates that reliability in multimodal pipelines can be dramatically improved through targeted, low-cost sanitization.

5.4 Synthesis

Taken together, these findings show that automated front-end generation is not limited by a model's ability to reason over multimodal inputs but by the orchestration and stabilization mechanisms surrounding it. Models are capable of producing semantically and visually coherent applications; the primary challenges lie in mitigating predictable formatting errors, optimizing architectural overhead, and reducing the significant cost associated with visual and functional evaluation.

The emergent picture is that full automation of front-end generation is feasible, but its scalability depends on: (1) selecting cost-efficient orchestration strategies, (2) integrating robust retry and repair mechanisms, and (3) advancing the reliability of multimodal judges.

These insights offer a pathway for developing more efficient, generalizable, and stable pipelines for multimodal UI generation in future systems.

Chapter 6

Threats

We identify several internal, external, and construct-related factors that may threaten the validity of our results. For each threat, we describe its potential impact on our findings and outline the mitigation strategies applied in our study. While the experimental design aims to minimize these risks, inherent limitations remain due to the evolving nature of multimodal LLMs and the complexity of multi-agent pipelines.

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 risk of judgment bias, including hallucinations or inconsistent interpretations of fine-grained behaviors and layout details. Such bias may inflate or deflate performance estimates. To mitigate this threat, all judges were prompted with fixed templates, no model evaluated its own outputs, and we conducted a manual audit over 180 randomly sampled judgments to measure inter-rater reliability using Cohen's κ. Agreement scores between 0.55 and 0.90 indicate substantial to almost-perfect reliability, yet some inconsistency remains inherent to automated evaluators.

Prompt Sensitivity and Instruction Drift. LLMs are highly sensitive to phrasing, position of instructions, and formatting of multimodal inputs. Minor variations can lead to divergent outputs, potentially confounding architectural differences with prompt-induced variance. We minimized this threat by standardizing prompts across all experiments, freezing model versions, and ensuring deterministic formatting of Figma-derived HTML and user-story inputs. Full prompt templates are included in the replication package for reproducibility.

Pipeline-Level Dependencies. Our multi-agent pipeline involves sequential interactions between Builder,

Validator, Fixer, and Judge agents. Errors or drift early in the pipeline may propagate through subsequent stages, influencing final outcomes. While automated cleanup and retry mechanisms reduce cascading failures, this dependency chain remains a source of internal variability. We addressed this by running identical inputs across all architectures and tracking all repair events through structured logs.

Multimodal Interpretation Variability. Because Figma designs are processed as both images and HTML-like structural representations, mismatches between visual perception and extracted structure could bias system performance. Differences in how models interpret screenshots versus DOM-style HTML may cause inconsistent grounding. We mitigate this by using a unified Figma-to-HTML extractor with consistent node ordering, but multimodal ambiguity remains an intrinsic risk.

6.2 External Validity

Dataset Scope and Representativeness. Our experiments were conducted on four open-source projects (75 user stories). Although these projects include realistic UI patterns and multi-component layouts, their domain diversity is still limited compared to large enterprise systems with complex state management, custom design frameworks, or tight backend integration. We selected production-grade OSS projects to approximate real-world conditions, but conclusions may not fully generalize to all web development contexts.

LLM Version Dependence and Model Drift. Our findings reflect the behavior of specific checkpoints of GPT-4o, Claude 3.5, and Gemini 1.5. These models are frequently updated, and future versions may behave differently in terms of generation quality, refusal tendencies, or cost-efficiency. To mitigate this threat, we document all model identifiers, API versions, and configuration parameters. However, full replication may be affected by upstream model changes outside our control.

Frontend-Only Evaluation. Our framework evaluates front-end React application generation without backend logic, server state, or API interaction. This restriction may limit generalization to full-stack applications where correctness depends on dynamic data flows, authentication, or asynchronous network operations. Nevertheless, front-end generation provides a representative multimodal reasoning challenge and captures the core difficulties of grounding UI code in visual and textual specifications.

Generalization Across Design Tools. The study focuses exclusively on Figma as the design source. While Figma is industry-standard, other design tools (Sketch, Adobe XD) may exhibit different structural conventions or metadata formats. Cross-tool generalization therefore remains untested.

6.3 Construct Validity

Evaluation Metrics and Granularity. Our evaluation uses categorical labels (`full match`, `partial`, `fail`) that summarize functional and visual correctness. While interpretable, these categories may overlook subtle distinctions such as minor spacing differences or near-correct event logic. Partial matches group together a wide range of cases, from almost-correct implementations to those missing essential elements. We mitigated this limitation through manual validation and detailed examples in the appendix.

LLM-as-a-Judge as a Measurement Instrument. Although LLMs provide scalable evaluation, they are not perfect substitutes for human experts. Their understanding of nuanced interaction logic or layout alignment may still be imperfect. We attempt to strengthen construct validity by using cross-family judging (e.g., Gemini judging GPT outputs) and performing human verification on a large subsample, but fully eliminating judge-related uncertainty is not possible.

Token Cost as a Proxy for Efficiency. Token usage is used as a measurement of computational efficiency, but it does not directly measure latency, runtime cost on GPU clusters, or inference-time parallelism. Future work could integrate energy consumption or server throughput as complementary efficiency metrics.

6.4 Conclusion of Threats

Taken together, these threats reflect the challenges inherent to evaluating complex multimodal, multi-agent LLM systems. While our design includes multiple safeguards such as prompt standardization, version control, manual auditing, and open replication artifacts, some limitations remain unavoidable due to model drift, dataset scope, and multimodal ambiguity. Nonetheless, the consistency of results across architectures and projects suggests that the core findings are robust and replicable, within the boundaries defined above.

Chapter 7

Conclusion

This thesis examined how multimodal, multi-agent LLM architectures generate front-end React applications grounded in both user stories and high-fidelity Figma designs. By evaluating three orchestration strategies Supervisor (tool-calling), Hierarchical, and Custom across four real-world projects and multiple model families, we provided one of the first systematic analyses of design-to-implementation pipelines using modern multimodal LLMs.

Across all configurations, our findings show that current models are capable of producing front-end applications that are both functionally meaningful and visually aligned with design intent. Full-match rates averaged 54.1% for functionality and 57.9% for visual fidelity, increasing to 76.9% and 84.9% when partial matches were included.

These results are significant given the complexity of the task, which requires end-to-end generation of applications that simultaneously satisfy both behavioral requirements and visual design constraints. Achieving over 50% full correctness indicates that models can produce fully valid outputs in a substantial portion of cases, while the high rate of partial matches suggests that many remaining issues are incremental and localized rather than fundamental failures.

Overall, this indicates that current models have a strong understanding of the core mapping between textual requirements and visual layout structure, but still require refinement to consistently achieve production-level quality.

Architectural choice was not a major determinant of output quality but strongly influenced computational efficiency. The Custom architecture consistently provided the best cost-performance ratio, reducing token consumption by 21-65% relative to more complex strategies while maintaining or improving accuracy. By

contrast, Supervisor (tool-calling) and Hierarchical architectures introduced substantial overhead due to repeated context passing and elaborate coordination logic. Judges emerged as the primary cost driver across all experiments, consuming on average 5.9× more tokens than generators, highlighting evaluation as a critical bottleneck in multimodal code generation pipelines.

Finally, our analysis of reliability issues revealed that most generation failures stemmed from predictable and superficial structural artifacts, such as code fences, missing exports, or explanatory text. GPT was the only model to produce refusal responses, but 91% of these were successfully recovered using an automated retry mechanism. Nearly all remaining artifacts were resolved automatically through a lightweight cleanup pipeline. These findings show that multimodal front-end generation is highly amenable to stabilization through simple, targeted repair strategies.

Overall, this thesis demonstrates that streamlined and deterministic orchestration-paired with automated repair and careful prompt design-provides a scalable path toward reliable multimodal front-end generation. As multimodal models continue to improve, the primary limitations are shifting away from model capability and toward evaluation efficiency, dataset availability, and system-level engineering choices.

7.1 Future Work

This work investigates how three multi-agent LLM architectures perform on the task of multimodal front-end generation. While the findings highlight clear trends in efficiency, robustness, and cost, several promising research directions remain.

7.1.1 Exploring Alternative Multi-Agent Architectures Beyond LangGraph

This thesis evaluated three architectural strategies within the LangGraph ecosystem: Supervisor (tool-calling), Hierarchical, and Custom. While these configurations enabled controlled comparisons, they represent only a subset of possible coordination models. Future work should examine *architectures that do not rely on LangChain or LangGraph*, and frameworks such as AutoGen, SPADE, or Agentverse. These alternatives employ fundamentally different assumptions regarding autonomy, communication, and control flow. Comparing them against the same multimodal generation pipeline would reveal whether the efficiency and stability patterns observed in this thesis generalize across frameworks or are specific to LangGraph's execution model.

7.1.2 Evaluating Architectural Effects Across Different Software Engineering Tasks

The findings presented in this thesis capture architectural behavior for a single task: multimodal front-end generation. However, different software engineering activities stress different reasoning capabilities. Tasks such as API design, automated testing, documentation alignment, refactoring, or back-end endpoint generation may exhibit distinct sensitivities to orchestration strategy. A promising direction for future work is to *apply the same architectural comparisons across multiple tasks* to determine whether the patterns observed here such as the superior efficiency of Custom architectures persist more broadly or are specific to the front-end domain. This cross-task analysis would offer a deeper understanding of when architectural design plays a critical role and when simple coordination strategies may suffice.

7.1.3 Extending the Pipeline to Back-End Generation for End-to-End Evaluation

Another natural extension of this work is to incorporate an automatically generated back-end layer, enabling a full-stack evaluation of multi-agent LLM pipelines. Even a minimal service layer for example, a REST API would allow investigation of whether the pipeline can maintain consistency across front-end and back-end components, including shared data models, validation rules, and authentication flows. Studying cross-layer coupling would make it possible to assess whether architectural choices influence end-to-end correctness and robustness, thereby better reflecting real-world software development workflows.

7.1.4 Constructing a Larger and More Diverse Dataset

The dataset used in this thesis consists of 75 user-story–Figma pairs across four open-source projects. Although it is, to the best of our knowledge, the only publicly available dataset that pairs real user stories with their corresponding Figma designs, its size limits the statistical power of large-scale architectural analysis. A valuable direction for future work is to expand the dataset significantly. One potential approach is to mine a broader set of open-source UI-centric repositories, extract their implemented web interfaces, and use tools such as DOM-to-Figma converters, automated layout extractors, or screenshot-to-design reconstruction systems to generate approximate Figma frames from live websites. This strategy would enable the creation of a substantially larger multimodal benchmark without requiring manual designer intervention. A richer dataset would support more rigorous comparisons, improved generalization analysis, and stronger empirical grounding for multimodal front-end generation research.

Summary

In summary, this thesis shows that multimodal multi-agent LLM systems are capable of producing usable front-end applications with strong design alignment and reasonable functional correctness. The main barriers are not conceptual but practical: cost, orchestration efficiency, and reliability. The results highlight the importance of purpose-built datasets, stable evaluation workflows, and lightweight repair mechanisms-laying a strong foundation for the next generation of automated front-end development tools.

Bibliography

[1] Samar Al-Saqqa, Samer Sawalha, and Hiba AbdelNabi. Agile software development: Methodologies and trends. *International Journal of Interactive Mobile Technologies*, 14(11), 2020.

[2] Mrs Rupali M Chopade and Nikhil S Dhavase. Agile software development: Positive and negative user stories. In *2017 2nd International Conference for Convergence in Technology (I2CT)*, pages 297–299. IEEE, 2017.

[3] Jenifer Tidwell. *Designing interfaces: Patterns for effective interaction design*. " O'Reilly Media, Inc.", 2010.

[4] Figma Design. Figma: the collaborative interface design tool.(2017). *Retrieved September*, 17:2017, 2017.

[5] José Matías Rivero, Julián Grigera, Gustavo Rossi, Esteban Robles Luna, Francisco Montero, and Martin Gaedke. Mockup-driven development: providing agile support for model-driven web engineering. *Information and Software Technology*, 56(6):670–687, 2014.

[6] Thomas Chau and Frank Maurer. Knowledge sharing in agile software teams. In *Logic versus approximation: essays dedicated to Michael M. Richter on the occasion of his 65th birthday*, pages 173–183. Springer, 2004.

[7] Yi Gui, Zhen Li, Yao Wan, Yemin Shi, Hongyu Zhang, Bohua Chen, Yi Su, Dongping Chen, Siyuan Wu, Xing Zhou, et al. Webcode2m: A real-world dataset for code generation from webpage designs. In *Proceedings of the ACM on Web Conference 2025*, pages 1834–1845, 2025.

[8] Kevin Moran, Carlos Bernal-CÃąrdenas, Michael Curcio, Richard Bonett, and Denys Poshyvanyk.

Machine learning-based prototyping of graphical user interfaces for mobile apps. *IEEE Transactions on Software Engineering*, 46(2):196–221, 2020. doi:10.1109/TSE.2018.2844788.

[9] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*, 2024.

[10] Ting Zhou, Yanjie Zhao, Xinyi Hou, Xiaoyu Sun, Kai Chen, and Haoyu Wang. Bridging design and development with automated declarative ui code generation. *arXiv preprint arXiv:2409.11667*, 2024.

[11] Figma, Inc. Figma: Collaborative interface design tool. https://www.figma.com/, 2016. Accessed: 2025-08-15.

[12] Yuxuan Wan, Chaozheng Wang, Yi Dong, Wenxuan Wang, Shuqing Li, Yintong Huo, and Michael R Lyu. Automatically generating ui code from screenshot: A divide-and-conquer-based approach. *arXiv preprint arXiv:2406.16386*, 2024.

[13] Chunyang Chen, Ting Su, Guozhu Meng, Zhenchang Xing, and Yang Liu. 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*, pages 665–676, 2018.

[14] Tony Beltramelli. pix2code: Generating code from a graphical user interface screenshot. In *Proceedings of the ACM SIGCHI symposium on engineering interactive computing systems*, pages 1–6, 2018.

[15] Jason Wu, Eldon Schoop, Alan Leung, Titus Barik, Jeffrey P Bigham, and Jeffrey Nichols. Uicoder: Finetuning large language models to generate user interface code through automated feedback. *arXiv preprint arXiv:2406.07739*, 2024.

[16] Dongyang Liu, Shitian Zhao, Le Zhuo, Weifeng Lin, Yi Xin, Xinyue Li, Qi Qin, Yu Qiao, Hongsheng Li, and Peng Gao. Lumina-mgpt: Illuminate flexible photorealistic text-to-image generation with multimodal generative pretraining. *arXiv preprint arXiv:2408.02657*, 2024.

[17] Alan Cooper, Robert Reimann, David Cronin, and Christopher Noessel. *About face: the essentials of interaction design*. John Wiley & Sons, 2014.

[18] Yihong Dong, Xue Jiang, Jiaru Qian, Tian Wang, Kechi Zhang, Zhi Jin, and Ge Li. A survey on code generation with llm-based agents. *arXiv preprint arXiv:2508.00083*, 2025.

[19] Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. Agentbench: Evaluating llms as agents. *arXiv preprint arXiv:2308.03688*, 2023.

[20] Mahmoud Mohammadi, Yipeng Li, Jane Lo, and Wendy Yip. Evaluation and benchmarking of llm agents: A survey. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2*, pages 6129–6139, 2025.

[21] Anna Beatriz Marques, Alex Felipe Costa, Ismayle Santos, and Rossana Andrade. Enriching user stories with usability features in a remote agile project: a case study. In *Proceedings of the XXI Brazilian Symposium on Software Quality*, pages 1–10, 2022.

[22] Savvas Petridis, Michael Terry, and Carrie Jun Cai. 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*, pages 1–6, 2023.

[23] Tianyi Huang. Fead: Figma-enhanced app design framework for improving ui/ux in educational app development. *arXiv preprint arXiv:2412.06793*, 2024.

[24] Pekka Abrahamsson, Outi Salo, Jussi Ronkainen, and Juhani Warsta. Agile software development methods: Review and analysis. *arXiv preprint arXiv:1709.08439*, 2017.

[25] LangChain AI. Langgraph multi-agent systems, 2025. URL https://langchain-ai.github.io/langgraph/concepts/multi_agent/. Low-level orchestration framework for long-running, stateful agents.

[26] Anonymous. Bridging design and implementation: A study of multi-agent llm architectures for automated front-end generation, July 2025. URL https://zenodo.org/records/17429375.

[27] Axel Van Lamsweerde. Requirements engineering in the year 00: A research perspective. In *Proceedings of the 22nd international conference on Software engineering*, pages 5–19, 2000.

[28] Lan Cao and Balasubramaniam Ramesh. Agile requirements engineering practices: An empirical study. *IEEE software*, 25(1):60–67, 2008.

[29] Garm Lucassen, Fabiano Dalpiaz, Jan Martijn EM van der Werf, and Sjaak Brinkkemper. Improving

agile requirements: the quality user story framework and tool. *Requirements engineering*, 21(3): 383–403, 2016.

[30] Fabio Staiano. *Designing and Prototyping Interfaces with Figma: Learn essential UX/UI design principles by creating interactive prototypes for mobile, tablet, and desktop*. Packt Publishing Ltd, 2022.

[31] Mohamad Yusril Aldiana Mahendra. *THE USE OF DRAW IO AS DIGITAL MIND MAP TO IMPROVE STUDENTSâĂŹCREATIVITY AND STUDENTSâĂŹCONCEPT MASTERY IN LEARNING HUMAN INFLUENCE ON ECOSYSTEM*. PhD thesis, Universitas Pendidikan Indonesia, 2021.

[32] Gail C. Murphy, David Notkin, and Kevin J. Sullivan. Software reflexion models: Bridging the gap between design and implementation. *IEEE Transactions on Software Engineering*, 27(4):364–380, 2001.

[33] Oksana Nikiforova, Kristaps Babris, and Farshad Mahmoudifar. Automated generation of web application front-end components from user interface mockups. In *Proceedings of International Conference on Software Technologies*, volume 1, pages 100–111, 2024.

[34] Tuan Anh Nguyen and Christoph Csallner. Reverse engineering mobile application user interfaces with remaui (t). In *2015 30th IEEE/ACM international conference on automated software engineering (ASE)*, pages 248–259. IEEE, 2015.

[35] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[36] Chenglei Si, Yanzhe Zhang, Ryan Li, Zhengyuan Yang, Ruibo Liu, and Diyi Yang. Design2code: Benchmarking multimodal code generation for automated front-end engineering. *arXiv preprint arXiv:2403.03163*, 2024.

[37] Mark Chen. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[38] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, et al. Autogen: Enabling next-gen llm applications via multi-agent conversations. In *First Conference on Language Modeling*, 2024.

[39] Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, et al. Chatdev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15174–15186, 2024.

[40] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, et al. Metagpt: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*, 2023.

[41] Langgraph: Framework for building multi-agent applications. https://langchain-ai.github.io/langgraph/, 2024. Accessed 2025-02-01.

[42] Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th annual acm symposium on user interface software and technology*, pages 1–22, 2023.

[43] OpenAI. Hello GPT-4o. *OpenAI Blog*, 5 2024. URL https://openai.com/index/hello-gpt-4o/. Introduces GPT-4o's multimodal capabilities and unified text–vision–audio reasoning.

[44] Bingyang Wei. Requirements are all you need: From requirements to code with llms. In *2024 IEEE 32nd International Requirements Engineering Conference (RE)*, pages 416–422. IEEE, 2024.

[45] Kristian Kolthoff, Felix Kretzer, Christian Bartelt, Alexander Maedche, and Simone Paolo Ponzetto. Guide: Llm-driven gui generation decomposition for automated prototyping. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 1–4. IEEE, 2025.

[46] Felix Kretzer, Kristian Kolthoff, Christian Bartelt, Simone Paolo Ponzetto, and Alexander Maedche. 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*, pages 1–19, 2025.

[47] Yilei Jiang, Yaozhi Zheng, Yuxuan Wan, Jiaming Han, Qunzhong Wang, Michael R Lyu, and Xiangyu Yue. Screencoder: Advancing visual-to-code generation for front-end automation via modular multimodal agents. *arXiv preprint arXiv:2507.22827*, 2025.

[48] Yunnong Chen, Shixian Ding, YingYing Zhang, Wenkai Chen, Jinzhou Du, Lingyun Sun, and Liuqing Chen. Designcoder: Hierarchy-aware and self-correcting ui code generation with large language models. *arXiv preprint arXiv:2506.13663*, 2025.

[49] Yi Gui, Yao Wan, Zhen Li, Zhongyi Zhang, Dongping Chen, Hongyu Zhang, Yi Su, Bohua Chen, Xing Zhou, Wenbin Jiang, et al. Uicopilot: Automating ui synthesis via hierarchical code generation from webpage designs. In *Proceedings of the ACM on Web Conference 2025*, pages 1846–1855, 2025.

[50] Builder.io. Builder.io: Visual headless cms for react and more. https://www.builder.io/, 2025. Accessed: 2025-06-22.

[51] Locofy.ai. Convert figma designs to code. https://www.locofy.ai/, 2021. Accessed: 2025-06-22.

[52] Bolt.new. Ai-powered ui builder, 2025. URL https://www.bolt.new/.

[53] Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, et al. Chatdev: Communicative agents for software development. *arXiv preprint arXiv:2307.07924*, 2023.

[54] LangChain AI. Langgraph, 2025. URL https://langchain-ai.github.io/langgraph/. Low-level orchestration framework for long-running, stateful agents.

[55] LangChain AI. Langgraph supervisor, 2025. URL https://langchain-ai.github.io/langgraph/concepts/multi_agent/#supervisor. Low-level orchestration framework for long-running, stateful agents.

[56] LangChain AI. Langgraph supervisor tool calling, 2025. URL https://langchain-ai.github.io/langgraph/concepts/multi_agent/#supervisor-tool-calling. Low-level orchestration framework for long-running, stateful agents.

[57] LangChain AI. Langgraph hierarchical, 2025. URL https://langchain-ai.github.io/langgraph/concepts/multi_agent/#hierarchical. Low-level orchestration framework for long-running, stateful agents.

[58] LangChain AI. Langgraph network, 2025. URL https://langchain-ai.github.io/langgraph/concepts/multi_agent/#network. Low-level orchestration framework for long-running, stateful agents.

[59] LangChain AI. Langgraph custom, 2025. URL https://langchain-ai.github.io/langgraph/concepts/multi_agent/#custom-multi-agent-workflow. Low-level orchestration framework for long-running, stateful agents.

[60] Google DeepMind. Gemini 2.5: Our most intelligent ai model—reasoning, coding, multimodal. *Google Blog*, 3 2025. URL https://blog.google/technology/google-deepmind/gemini-model-thinking-updates-march-2025/. Describes Gemini 2.5 Pro's reasoning, coding, and long-context capabilities.

[61] Anthropic. Introducing claude sonnet 4.5. *Anthropic News*, 4 2025. URL https://www.anthropic.com/news/claude-sonnet-4-5. Details Claude Sonnet 4.5's improvements in reasoning, code generation, and context length.

[62] Vellum AI. Llm leaderboard 2025 - vellum ai. https://www.vellum.ai/llm-leaderboard, 2025. Accessed: 2025-10-19.

[63] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations (ICLR 2024)*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

[64] 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. SWE-bench multimodal: Do ai systems generalize to visual software domains? In *The Thirteenth International Conference on Learning Representations (ICLR 2025)*, 2025. URL https://openreview.net/forum?id=riTiq3i21b.

[65] Weixi Tong and Tianyi Zhang. Codejudge: Evaluating code generation with large language models. *arXiv preprint arXiv:2410.02184*, 2024.

[66] Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. G-eval: Nlg evaluation using gpt-4 with better human alignment. *arXiv preprint arXiv:2303.16634*, 2023.

[67] Arjun Panickssery, Samuel Bowman, and Shi Feng. Llm evaluators recognize and favor their own generations. *Advances in Neural Information Processing Systems*, 37:68772–68802, 2024.

[68] Dawei Li, Renliang Sun, Yue Huang, Ming Zhong, Bohan Jiang, Jiawei Han, Xiangliang Zhang, Wei Wang, and Huan Liu. Preference leakage: A contamination problem in llm-as-a-judge. *arXiv preprint arXiv:2502.01534*, 2025.

[69] Gelilaa and Contributors. Vox-box. https://github.com/gelilaa/VoxBox, 2024. Accessed: 2025-10-19, 2 stars, 34 user stories.

[70] Tim Goalen. Transcriber frontend. https://github.com/timgoalen/transcriber-frontend, 2024. Accessed: 2025-10-19, 3 stars, 23 user stories.

[71] GSG-K3. Urban calendar. https://github.com/GSG-K3/urban-calendar, 2024. Accessed: 2025-10-19, 3 stars, 9 user stories.

[72] GSG-G9. House hunting app. https://github.com/GSG-G9/house-hunting-app, 2024. Accessed: 2025-10-19, 2 stars, 9 user stories.

[73] Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. Rico: A mobile app dataset for building data-driven design applications. In *Proceedings of the 30th annual ACM symposium on user interface software and technology*, pages 845–854, 2017.

[74] Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46, 1960.

Appendices

A.1 Algorithms

A.1.1 Multimodal Front-End Generation Pipeline

This appendix details the end-to-end multimodal pipeline used to generate, enrich, and validate React applications from Figma designs and textual user stories. The pipeline combines HTML extraction, LLM-based code generation, multimodal reasoning with screenshots, functional and visual evaluation, and automatic repair.

A. Algorithm Overview

Algorithm 1 Multimodal React Generation Pipeline

Require: Figma file F, set of frames $f_1 \dots f_n$, user stories S, LLM model M

1: Extract raw Figma node tree for each frame f_i

2: Render a high-resolution thumbnail image I_i for each frame

3: Convert each node tree into a structured HTML fragment H_i with inline styles

4: **for** each frame f_i **do**

5: Build prompt containing:

- Inline-style HTML H_i
- Frame screenshot I_i
- Structural rules for React + Tailwind

6: Call LLM tool `react_generator_llm` to produce JSX component C_i

7: Clean the component using the JSX Cleanup module:

- remove code fences and prose
- strip preambles
- enforce a single default export

8: **end for**

9: Aggregate all generated components into a runnable React project P

10: Use Story Mapper to compute mappings:

- Story → Component(s)
- Story → Figma screenshot(s)

11: **for** each user story $s \in S$ **do**

12: Identify relevant components using mapping

13: Provide s + relevant components + screenshot(s) to `enricher_llm`

14: Receive updated JSX implementing dynamic behavior

15: Apply fixes to project P

16: **end for**

B. Notes

- The pipeline supports both GPT, Claude, and Gemini multimodal models.
- Image tiles from Figma thumbnails are sent through the multimodal API.
- Refusal detection and retry logic ensure robustness across models.
- All interactions are logged with token usage and cost estimates.

A.1.2 Figma-to-HTML Extraction Pipeline

To translate visual designs into a form suitable for LLM-based code generation, we implement a Figma-to-HTML extraction pipeline. The goal of this module is to convert Figma frames into (i) a hierarchical HTML-like structure that preserves layout and styling, and (ii) PNG thumbnails used for multimodal grounding during generation and enrichment.

The pipeline interacts directly with the Figma REST API, retrieves full node trees, extracts relevant layout and styling information from each node, and recursively converts the structure into semantically meaningful HTML tags. When image fills are detected, the pipeline downloads the corresponding assets to ensure they are available during code generation.

Algorithm 2 summarizes the main steps of this process.

A.1.3 Story-to-Component and Story-to-Figma Mapping Pipeline

The Story Mapper is responsible for linking each user story to (i) one or more React component files produced by the generator, and (ii) one or more Figma frames used during visual grounding. This mapping provides the Enricher with the correct file-level context when inserting dynamic behavior, and it enables the Judge agent to evaluate functional and visual alignment.

The pipeline first loads all available user stories, generated component filenames, and extracted Figma screenshots. It then issues two LLM queries: one to map stories to React components, and another to map stories to Figma frames. The LLM returns two JSON structures that are saved to disk and used throughout the generation and evaluation pipeline.

Algorithm 3 summarizes this process.

Algorithm 2 Figma-to-HTML Extraction Pipeline

```
Require: Figma file key F, optional page name p
Ensure: Set of extracted frames 𝒦, PNG thumbnails, and HTML-like representations
 1: doc ← fetch_figma_file(F)
 2: 𝒦 ← extract_all_frames(doc, p)
 3: for all k ∈ 𝒦 do
 4:     node_id ← k.id
 5:     thumb ← get_frame_thumbnail(F, node_id)            ▷ PNG screenshot
 6:     html ← figma_node_to_html(k)                       ▷ Recursive HTML-like rendering
 7:     store (k, thumb, html)
 8: end for
 9: function FIGMA_NODE_TO_HTML(node)
10:     tag ← semantic_tag_for_node(node)
11:     style ← extract_styles(node)                       ▷ size, color, typography, borders
12:     if node contains image fill then
13:         src ← download_image(F, node.id)
14:         return <img src="$src" style="$style" />
15:     end if
16:     children ← concatenation of FIGMA_NODE_TO_HTML(c) for all c in node.children
17:     if node.type = TEXT then
18:         return <p style="$style">node.characters</p>
19:     else
20:         return <$tag style="$style">$children</$tag>
21:     end if
22: end function
23: return 𝒦 with (html, thumbnails)
```

A.1.4 Refusal Detection and Retry Strategy

During code generation and evaluation, large language models may occasionally return meta-responses instead of executable code (e.g., apologies, requests for more information, or generic disclaimers). To reduce manual intervention and improve the robustness of the pipeline, we implement a refusal detection and retry mechanism within the shared LLM wrapper.

The mechanism relies on a small set of refusal patterns (e.g., "I'm sorry", "I cannot help", "as an AI", "I need more information") that are checked against every model response. When a refusal is detected, the event is logged together with the model name, tool identifier, and matched patterns, and the call is retried up to a fixed maximum number of attempts. For multimodal generation, the system prompt is also strengthened on subsequent attempts to explicitly discourage further refusals and encourage the model to make reasonable assumptions.

Algorithm 4 summarizes the core procedure.

Algorithm 3 Story-to-Component and Story-to-Figma Mapping Pipeline

Require: User stories S, component files C, Figma screenshots $\mathcal{F}$
Ensure: JSON mappings: story → components, story → figma frames

1: $S \leftarrow$ load_stories() ▷ Read `.txt` files
2: $C \leftarrow$ load_component_names() ▷ React component filenames
3: $\mathcal{F} \leftarrow$ load_figma_screenshots()
4: Build story descriptions string D from S
5: Construct prompt P_{comp}:

- Input: list of component filenames C and user stories D
- Output: JSON mapping `component_file.jsx âĘŠ [story_ids]` plus `"__unmapped__"`

6: $M_{comp} \leftarrow \text{LLM}(P_{comp})$ ▷ Story → component mapping
7: Save M_{comp} to disk
8: Construct prompt P_{fig}:

- Input: list of Figma screenshots $\mathcal{F}$ and user stories D
- Output: JSON mapping `story_id âĘŠ [figma_frames]`

9: $M_{fig} \leftarrow \text{LLM}(P_{fig})$ ▷ Story → Figma mapping
10: Save M_{fig} to disk
11: **return** (M_{comp}, M_{fig})

A.1.5 JSX Cleanup Algorithm

To ensure that LLM-generated projects remain compilable and runnable, we apply a post-processing step that removes common generation artifacts and scaffolds missing core files. The `JSX_cleanup` routine operates over all React components and entry files in a generated project and performs a series of deterministic transformations. These transformations target issues such as Markdown code fences, preamble text, excessive comments, missing `export` statements, trailing prose after exports, and missing project skeleton files.

Algorithm 5 summarizes the procedure.

Algorithm 4 Refusal Detection and Retry Strategy

Require: Tool identifier T, prompt or messages X, model name M, maximum attempts N
Ensure: Final model response R or unrecovered failure

1: Define list of refusal patterns $\mathcal{P}$ (e.g., “i’m sorry”, “i cannot help”, “as an ai”, “i need more information”)
2: **for** $k \leftarrow 1$ to N **do**
3:     Call provider-specific API with (M, X) to obtain response R_k and usage statistics U_k
4:     $r \leftarrow$ LOOKS_LIKE_REFUSAL(R_k, M, T) ▷ Check if any pattern in $\mathcal{P}$ matches
5:     Log summary of this response via LOG_ALL_RESPONSE(M, T, r, U_k)
6:     **if** r = True **then**
7:         LOG_REFUSAL($M, T, \mathcal{P}'$) ▷ $\mathcal{P}' \subseteq \mathcal{P}$ that matched
8:         **if** $k < N$ **then**
9:             **if** multimodal generation and M is GPT **then**
10:                 Strengthen system instructions to:
“Do not ask for more information. Make reasonable assumptions and return ONLY valid React code.”
11:                 Replace system message in X and continue to next attempt
12:             **else**
13:                 Wait briefly and retry with the same input X
14:             **end if**
15:         **else**
16:             **return failure** ▷ Unrecovered refusal after N attempts
17:         **end if**
18:     **else** ▷ Non-refusal response
19:         **if** JSON is expected **then**
20:             Extract JSON block from R_k (e.g., remove code fences, parse object)
21:             LOG_RESPONSE(M, R_k, T, U_k)
22:             **return** parsed JSON
23:         **else**
24:             LOG_RESPONSE(M, R_k, T, U_k)
25:             **return** R_k
26:         **end if**
27:     **end if**
28: **end for**

Algorithm 5 JSX Cleanup Overview

1: Ensure required project files exist: `index.html`, `App.jsx`, `index.js`, `index.css`
2: Load all JSX files from generated directories
3: **for all** files **do**
4:     Remove code fences (“‘`jsx`) if present
5:     Strip preamble text before first `import`
6:     Remove JS comments (block and single-line)
7:     Infer missing `export default` statements
8:     Trim trailing text after final export
9:     Save cleaned file and update counters
10: **end for**
11: Log per-run and aggregated cleanup statistics

A.2 Prompt Templates

A.2.1 Builder / Generator Prompt

```
You are a supervisor deciding which builder tool to run next in a React generation
    pipeline.

Options:
- figma_tool: Build from Figma before creating the react app
- react_generator_llm: Create React App from HTML
- story_mapper_llm: Matches user stories to component files. Run this before running
    the next steps
- enricher_llm: Enriches React components with user story logic

Return only the name of the next tool to run.
```

Listing A.1: Supervisor Builder Prompt Template

A.2.2 Enricher Prompt

```
You are a senior React frontend engineer. Your job is to enrich a React component by
    implementing dynamic behavior, user interactivity, and state logic based on
    specific user stories.
Preserve the existing layout, colors, and visual appearance shown in the screenshot.
Do not introduce unnecessary visual changes. Focus on wiring up logic, adding hooks,
    and supporting user actions.

IMPORTANT:
 - If the component represents a standalone page (e.g., Login, Home, Profile), output
    a first line comment like '/* @route: /login */' with a kebab-cased path. Otherwise
     output '/* @route: none */'.
 - After that first line, output ONLY the full, valid React component code. No extra
    commentary.

Below is a list of user stories that describe expected functionality for this
    component.
Your task is to modify the given React component to satisfy these requirements.
```

```
Rules:
- Add state variables, event handlers, or React hooks as needed.
- Keep the JSX structure and styling visually unchanged unless strictly required.
- Preserve the existing layout and styling from the screenshot.
- Output only valid React code, with the FIRST LINE being a comment marker for routing
    : Either ‘/* @route: /some-path */‘ when this component should be a page route, or
    ‘/* @route: none */‘ when it should not be routable.
- Use only hardcoded values. No props.

User Stories:
{stories}

React Component:
{component_code}
```

Listing A.2: Enricher Prompt Template

A.2.3 Functional Judge Prompt

```
You are a frontend code reviewer evaluating a **frontend-only** React project.

This project has **no backend**. All features are implemented **at the UI level**,
    using local state, React context, mock data, or placeholders.

Your goal is to judge whether the React code **fully satisfies the user story at the
    frontend level**.
Do not penalize the implementation for missing backend logic, APIs, or persistent
    storage.

Evaluation Criteria:
- "full_match": Every requirement in the user story is clearly and explicitly
    implemented in the code at the UI level (even if mock logic is used).
- "partial": Some but not all requirements are implemented.
- "fail":Core elements of the user story are missing, broken, or implied without code.

Rules:
```

```
- Simulated logic using mock data, local state, or placeholder functions counts as
    valid.
- Do not reduce the score for lack of backend/persistence.
- Only what is explicitly implemented in code should be considered.
- Do not guess or assume missing features.

Here is the user story:
---
{user_story}
---

Here is the code:
---
{code}
---

Return your judgment in this JSON format:
{
  "implements_story": "true" | "partial" | "false",
  "explanation": "Step-by-step explanation of whether each user story requirement is
    implemented.",
  "component_names": ["Component1.jsx", "Component2.jsx"]
}
```

Listing A.3: Judge Prompt Template for Functional Coverage

A.2.4 Visual Judge Prompt

```
You are a **visual QA expert** evaluating whether a rendered React component visually
    fulfills a given user story, based solely on comparison to a Figma design.

Scope of Evaluation (Strict Visual Focus Only):
- Compare only the screenshots -- do not evaluate interactivity, behavior, or code.
- Focus on: presence/absence of elements, visual hierarchy, layout, colors, and
    spacing.
- Ignore: animations, responsiveness, hover effects, functionality.
```

```
Visual Match Levels:
- "full_match": Most or all key visual elements are present, styled appropriately, and
     clearly communicate the intent of the user story.
- "partial": Some essential visual elements are present, but others are missing,
    incorrect, or poorly aligned.
- "fail": Critical visual elements required to communicate the user story are absent
    or incorrect.

Instructions:
Compare the following:
- Figma design image: {figma_path}
- Rendered component image: {component_path}

Here is the user story:
"{user_story}"

Be fair and practical. Use "full_match" when the layout is mostly accurate and clearly
     supports the user story, even if not visually perfect.

Return ONLY a valid JSON object with the structure below:

{
  "user_story": "{user_story}",
  "match_level": "complete" | "partial" | "no",
  "issues": [
    "brief description of visual mismatches"
  ],
  "explanation": "Short paragraph explaining how well the visual layout supports the
    user story."
}
```

Listing A.4: Judge Prompt Template for Visual Fidelity

A.2.5 Fixer Prompt

```
You are a frontend QA engineer. Your task is to fix a broken React component based on
    a list of issues found during prior evaluation.
```

```
If the component already addresses all listed issues, return it unchanged. If it does
    not, return the fully corrected code.
Only fix the problems described - do not make unrelated changes. Maintain the
    component's original visual layout as shown in the screenshot.
Important rules:
- Do not return Markdown code fences (e.g., '''jsx).
- Do not return JSON or explanations.
- Do not add comments, summaries, or surrounding text.
- Your response must be the complete and valid JSX code âĂŤ and nothing else.
- Do not change any image imports. Available image files are listed below.
```

Listing A.5: Fixer Prompt Template

A.3 Dataset Example

This example demonstrates how a single user story from the dataset is represented across text, design, and code artifacts, as well as how it is mapped to Figma frames and React components.

A.3.1 User Story

Story ID: 8

Text: I need a control panel to manage my houses.

As a user I can:

- Preview my houses
- Add a new house
- Delete a house from the list
- Edit my house
- Filter and sort houses

A.3.2 Associated Figma Frame

The story is visually grounded in a Figma design showing the user profile and house-management interface.

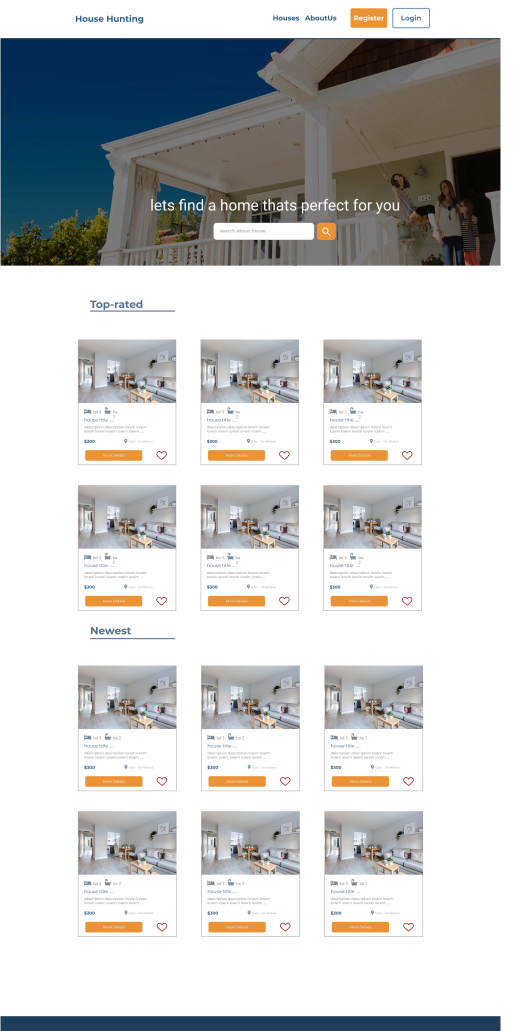

Figure 7.1: Figma frame associated with Story 8.

A.3.3 Story-to-Component Mapping

The Story Mapper tool identified that this user story relates to the React component `profileuser-info.jsx`:

Listing A.6: Story-to-Component Mapping for Story 8

```
{
```

```
"profileuserinfo.jsx": ["11", "17"]
}
```

Story 17 corresponds to the same conceptual feature group (profile and house management).

A.3.4 Story-to-Figma Mapping

Listing A.7: Story-to-Figma Mapping for Story 8


```
{
  "17": ["profile-userinfo.png"]
}
```


This links story 17—which groups logically with story 8—to the correct Figma frame representing the house management panel.

A.3.5 Generated React Component (Excerpt)

```
import React, { useState } from "react";

const ProfileUserInfo = () => {
  const [isEditing, setIsEditing] = useState(false);
  const [userInfo, setUserInfo] = useState({
    username: "Muhammad",
    email: "muh@muh.com",
    mobile: "0599000000",
  });
  const [formData, setFormData] = useState({
    username: "Muhammad",
    email: "muh@muh.com",
    mobile: "0599000000",
    password: "",
  });

  const handleEditClick = () => {
    setFormData({ ...userInfo, password: "" });
    setIsEditing(true);
```

```
};

const handleSaveClick = () => {
  const { password, ...updatedUserInfo } = formData;
  setUserInfo(updatedUserInfo);
  setIsEditing(false);
};

const handleInputChange = (e) => {
  const { name, value } = e.target;
  setFormData((prev) => ({ ...prev, [name]: value }));
};

return (
  <div style={{ width: "1440px", height: "1483px", position: "relative" }}>
    {/* ... layout and styling markup omitted for brevity ... */}

    <p style={{ fontSize: "36px", fontWeight: 700, color: "rgb(47,92,132)" }}>
      User Information
    </p>

    {/* Username */}
    {isEditing ? (
      <input
        type="text"
        name="username"
        value={formData.username}
        onChange={handleInputChange}
      />
    ) : (
      <p>{userInfo.username}</p>
    )}

    {/* Email */}
    {isEditing ? (
      <input
```

```
          type="email"
          name="email"
          value={formData.email}
          onChange={handleInputChange}
        />
      ) : (
        <p>{userInfo.email}</p>
      )}

      {/* Mobile */}
      {isEditing ? (
        <input
          type="tel"
          name="mobile"
          value={formData.mobile}
          onChange={handleInputChange}
        />
      ) : (
        <p>{userInfo.mobile}</p>
      )}

      {/* Password (masked when not editing) */}
      {isEditing ? (
        <input
          type="password"
          name="password"
          value={formData.password}
          onChange={handleInputChange}
          placeholder="Enter new password"
        />
      ) : (
        <p>***************</p>
      )}

      {/* Action buttons */}
      {isEditing ? (
```

```
        <button onClick={handleSaveClick}>Save Changes</button>
      ) : (
        <button onClick={handleEditClick}>Edit Information</button>
      )}

      <button style={{ borderColor: "rgb(166,11,11)", color: "rgb(177,25,25)" }}>
        Delete my account
      </button>
    </div>
  );
};

export default ProfileUserInfo;
```

Listing A.8: Excerpt of generated React component `ProfileUserInfo.jsx` mapped to house management stories

A.3.6 Summary

This example illustrates how a single user story links together:

- textual requirements describing house management,
- one or more Figma frames showing the intended UI,
- generated React components implementing the UI logic,
- and automated mappings used by the multi-agent pipeline.

These cross-modality links form the foundation of the benchmark used in this thesis for evaluating functional coverage and visual fidelity.