

Heterogeneous Agentic AI System Using Fine-Tuned SLMs

Arsh ^{1*}, Akshay Rajput¹ and Swarnendu Ghosh²

^{1*}Department of Computer Science and Engineering, Graphic Era
(Deemed to be University), Dehradun, India.

²AI Research Division, StartHack.IO Pvt Ltd, Dehradun, India.

*Corresponding author(s). E-mail(s): arsh03.2005@gmail.com;

Abstract

This paper introduces a heterogeneous agentic AI architecture that leverages fine-tuned Small Language Models (SLMs) for efficient, context-aware UI component generation. Unlike general-purpose LLM approaches that entail heavy computational loads and lack task specificity, the proposed model employs Gemma 4 E2B, a lightweight general-purpose SLM with 2.3B effective parameters, to tokenize user requests and transfer section-level generation tasks to Qwendean, our SLM fine-tuned from Qwen3-4B and specialized in the UI domain. A set of more than 900 UI components collected from ShadCN and TailwindCSS was augmented with synthetic prompt-completion pairs generated using DeepSeek models. After three augmentation passes the data increased to over 4100 samples and was released as `iamdyeus/ui-instruct-4k` on Hugging Face. Fine-tuning was performed using rsLoRA at 16-bit precision. The final pipeline, orchestrated by LangGraph, produces complete functioning React/TypeScript pages. Empirical results show that the heterogeneous design yields better cohesion, lower inference latency (3–5 s per section), near-zero API costs, and higher stylistic consistency compared to homogeneous LLM baselines. We further demonstrate that functional specialization—rather than model scale—is the key to efficient agentic AI.

Keywords: Agentic AI, Small Language Models, Fine-Tuning, rsLoRA, UI Code Generation, LangGraph, ShadCN, TailwindCSS, Heterogeneous AI Systems

1 Introduction

The agentic AI paradigm represents a new trend in artificial intelligence in which agents break down complex tasks into smaller ones and carry them out efficiently [1]. Unlike traditional generative AI that produces single-step outputs, agentic models incorporate planning and tool use for multi-step reasoning and execution [2]. While larger orchestrator models show higher competence in reasoning and task decomposition, they suffer from lower speeds and cost inefficiency. Narrow tasks such as creating an icon or a hero section do not require such heavy capabilities.

Small Language Models (SLMs) have recently become popular as alternatives to LLMs for specific operations within an agentic architecture. As noted in recent research [1], SLMs offer substantial advantages in heterogeneous systems: lower computation cost, faster inference, and the possibility of fine-tuning for precise applications. These benefits become crucial when latency, cost, and scalability are considered. Importantly, the advantage does not depend solely on model size but on functionality.

This paper argues—and demonstrates empirically—that a heterogeneous agentic architecture, where a general-purpose SLM handles orchestration and a domain-specialized fine-tuned SLM handles generation, outperforms homogeneous architectures where a single model performs both roles. Our architecture employs Gemma 4 E2B as a lightweight requirements gatherer and task planner, paired with Qwendean (fine-tuned Qwen3-4B) as a specialized UI code generator. The advantages go beyond cost: the specialized model produces outputs better aligned with the target domain in syntactic validity and stylistic coherence, while the general-purpose model is more effective at query decomposition without resorting to supercomputers.

We implement this framework for automated web UI component generation. Our key contributions are:

1. A systematic three-round iterative pipeline for collecting, normalising, annotating and augmenting UI component datasets, increasing from 900 to over 4100 high-quality training examples from open-source libraries.
2. A synthetic prompt generation methodology using DeepSeek-V3 to produce diverse, context-aware prompt-completion pairs in JSONL format, published as `iamdyeus/ui-instruct-4k` on Hugging Face.
3. Qwendean: a LoRA fine-tuned version of Qwen3-4B using rank-stabilized LoRA (rsLoRA) with 16-bit precision, proving that a 4B model fine-tuned on specific data can generate production-ready React/TypeScript/ShadCN/Tailwind components with significantly higher consistency than general-purpose models.
4. Demonstration that lightweight general-purpose models suffice for orchestration in scope-controlled agentic systems; diversity of specialisations, not model size, is responsible for efficiency.
5. A heterogeneous agentic pipeline orchestrated by LangGraph, where Gemma 4 E2B decomposes a user page request into section-level prompts, sends them to Qwendean for generation, and assembles the final page.
6. An iterative analysis of overfitting, dataset quality and training dynamics across three training rounds, offering practical insights for future SLM fine-tuning in code generation.

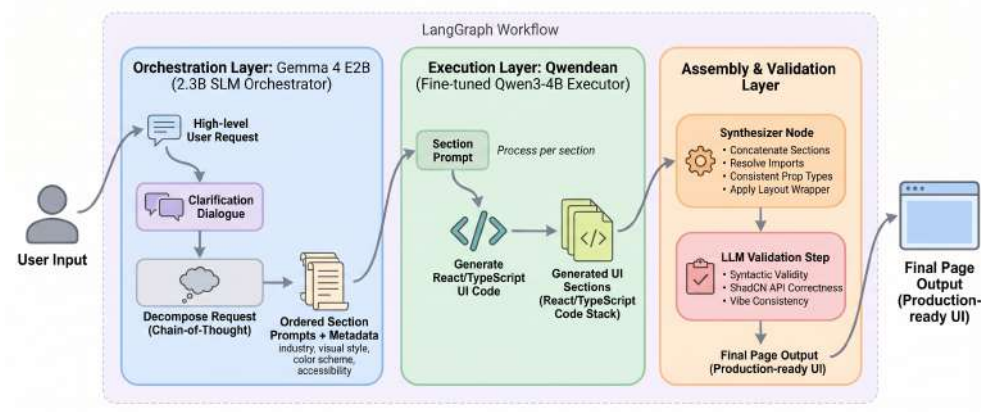


Fig. 1 Overall heterogeneous agentic architecture

2 Related Work

2.1 Agentic AI and Task Orchestration

Agentic AI extends generative models with autonomy and tool use. Frameworks such as LangGraph and AutoGen enable multi-agent architectures where specialised agents collaborate on complex tasks [3]. Key features include task decomposition, memory/-planning, and tool invocation. Recent studies highlight resource conservation when small models handle recurring tasks [1].

2.2 Small Language Models in Specialised Domains

Smaller models (under 10B parameters) can perform well on code generation and mathematical reasoning when properly fine-tuned [4]. Qwen models show strong results in code understanding and generation; Qwen3-4B offers a favourable balance between capacity and computational requirements [5]. Small general-purpose models can perform complex reasoning if properly scaffolded. Specialisation-based heterogeneity (generalist vs. domain expert) yields better efficiency than scale-based heterogeneity (large vs. small) for functionally decomposable tasks.

2.3 LoRA and Quantised Fine-Tuning

Low-Rank Adaptation (LoRA) [12] injects trainable low-rank matrices into transformer layers, reducing trainable parameters. QLoRA [13] quantises the base model to 4-bit NF4, further reducing memory. Rank-Stabilised LoRA (rsLoRA) [17] normalises the scaling factor by \sqrt{r} instead of r , stabilising training at higher ranks ($r = 32$ and above). We employ rsLoRA throughout because structured code generation benefits from increased rank capacity.

2.4 Synthetic Data Generation for Fine-Tuning

Recent approaches use synthetic prompt generation to create diverse, realistic prompts [6], preference optimisation (DPO) [7] to align outputs without reward models, and multi-stage pipelines with reasoning and critique models [8]. Our approach follows this paradigm, using DeepSeek-V3 for component description and synthetic prompt construction.

2.5 UI Component Libraries and Code Generation

Modular UI frameworks (ShadCN, TailwindCSS, Radix UI) provide reusable building blocks. Automating their selection and adaptation to specific contexts is an open challenge [9]. Existing systems either rely on monolithic LLMs (costly and slow) or brittle rule-based heuristics. Our heterogeneous approach bridges this gap by combining planning with specialised fine-tuned generation.

2.6 Gaps in Current Literature

Few works combine LLMs and agentic systems in an end-to-end, empirically rigorous manner. Specifically, the literature lacks:

1. Analysis of data size and source heterogeneity effects on LLM-generated code quality through iterative training.
2. Investigation of overfitting during fine-tuning on small code datasets.
3. Integration of fine-tuned LLMs as specialised agents in graph-based orchestration platforms.

This paper addresses all three gaps.

3 Problem Statement

Traditional automatic code/content creation suffers from fundamental shortcomings that hinder production-level use, regardless of whether large LLMs or general-purpose SLMs are employed.

Absence of Domain Specialisation

General-purpose models (70B LLMs or 7B SLMs) cannot specialise output to domain-specific constraints. When asked to create a React component, there is no guarantee that ShadCN primitives, Tailwind classes, or project-specific guidelines will be used. Even general-purpose SLMs lack an adequate understanding of idiomatic syntax unless explicitly prompted.

Computational Inefficiency of Monolithic Architecture

Systems where a single model handles both orchestration and execution cannot optimise separately for these two activities. Orchestration (logical reasoning in a limited domain) suits lightweight general-purpose models, while coding requires deep specialisation (internal patterns, naming conventions, library-specific APIs). Coupling both forces them into the computational constraints of the latter.

Wasted Computational Resources in Latent Operations

Using one large model for all tasks wastes compute resources. Even using one general-purpose model for both roles yields poorer results than using a separate generalist planner and fine-tuned domain specialist—not only due to computation but because of output quality differences.

Context Loss and Inconsistency

General-purpose models produce outputs that depend on exact prompt wording; different phrasings lead to different styles, structures, and inconsistencies. A fine-tuned domain specialist produces consistent style, predictable structure, and conformance to library idioms.

4 Methodology

4.1 Dataset Construction: Iterative Development

The dataset evolved across three rounds (Fig. 2), each motivated by observed training behaviour.

4.1.1 Initial Collection (900+ components)

We collected UI components from ShadCN UI and TailwindCSS using automated scraping, extracting over 900 distinct components (hero sections, feature lists, navbars, footers, testimonials, cards, modals). A standardisation process (code formatting, dependency management, metadata extraction, quality assessment) produced more than 900 validated components. Initial fine-tuning on this corpus led to severe overfitting—the model memorised training samples but produced redundant or gibberish responses.

4.1.2 Prompt Augmentation (2,400+ samples)

To address overfitting, we generated multiple prompt variants per existing component using DeepSeek-V3 [6]. Each component’s code and metadata produced a baseline natural-language query, which was elaborated via chain-of-thought reasoning incorporating industry context, visual vibe, and accessibility. This yielded about 2,400 training rows. Overfitting decreased, and the model began generating more diverse outputs, but quality remained suboptimal (incorrect API merges, outdated Tailwind syntax, poor prompt alignment).

4.1.3 Multi-Library Expansion (4,100+ samples)

We expanded the source corpus with publicly available UI libraries (Aceternity UI, Magic UI, etc.) that follow the ShadCN design language, sourced from GitHub based on stars and activity. This added new archetypes, improved React/TypeScript idioms, and increased style diversity without diverging from ShadCN. The expanded corpus went through the same two-stage synthetic prompt pipeline, producing a final dataset of approximately 4,100 samples (90/10 train/validation split), released as

iamdyeus/ui-instruct-4k. Training on this dataset substantially improved output coherence, idiomatic correctness, and semantic alignment.

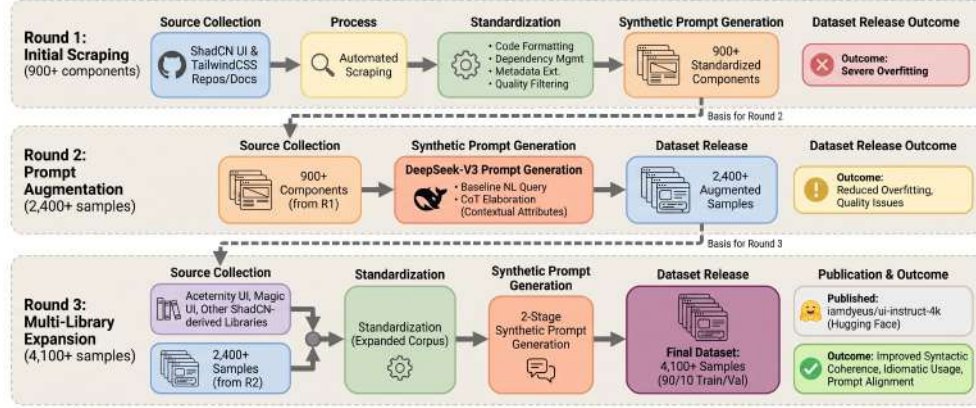


Fig. 2 Dataset evolution pipeline across three rounds

4.2 JSONL Dataset Format

All training data is stored in JSONL format (standard for Hugging Face, Unsloth):

```
{"prompt": "<DESCRIBE THE COMPONENT>", "completion": "<GENERATED CODE OR OUTPUT>"}
```

4.3 Model Selection and Fine-Tuning Configuration

We selected Qwen3-4B as the base SLM for:

- Competitive code generation performance.
- 4B parameter scale (efficient on consumer GPUs, 8–10GB VRAM for FP16 inference).
- Open-source (Apache 2.0) via Hugging Face.
- Dual reasoning modes (thinking mode disabled to produce clean code-only output).

The model was loaded in 16-bit (bfloat16) precision without quantisation, using Unsloth’s FastLanguageModel with gradient checkpointing. Although QLoRA [13] was evaluated, it introduced syntax-level artifacts in code generation and was not adopted.

LoRA adapters were configured with rsLoRA:

- Rank $r = 32$, $\alpha = 32$, dropout = 0.0.
- Target modules: `q_proj`, `k_proj`, `v_proj`, `o_proj` (attention) and `gate_proj`, `up_proj`, `down_proj` (MLP).
- rsLoRA enabled: scaling factor normalised by \sqrt{r} for stability.

Training hyperparameters:

- Batch size: per device 2, gradient accumulation 8, effective batch size 16.
- Epochs: 3.
- Learning rate: 1×10^{-4} , linear decay, warmup 16 steps.
- Optimizer: AdamW 8-bit, weight decay 0.01.
- Max sequence length: 4096 tokens.
- Evaluation every 50 steps, early stopping patience 3, best checkpoint loaded.
- Packing disabled to avoid mid-code truncation artifacts.

Training used an NVIDIA RTX 6000 PRO. After training, LoRA adapters were merged into the base model and pushed to Hugging Face in merged_16bit precision, with additional GGUF quantisations for Ollama/LM Studio.

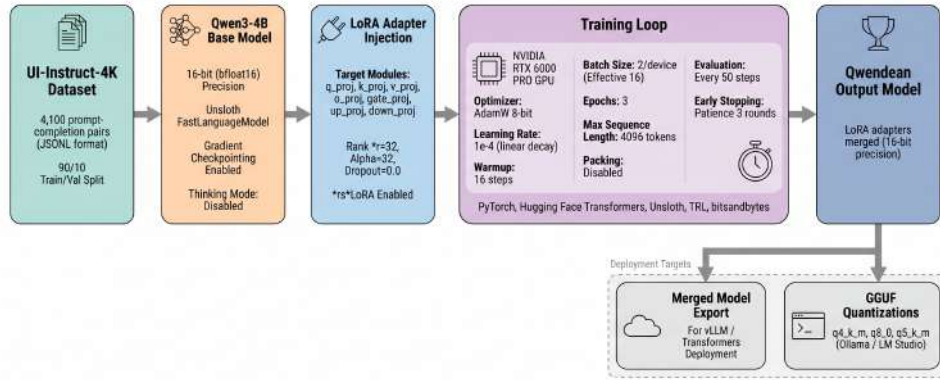


Fig. 3 Training pipeline

4.4 Agent Architecture System

We designed a heterogeneous agentic system for UI page generation with three logical layers: Orchestration, Execution, and Assembly/Validation, implemented using LangGraph for graph-based workflow management.

4.4.1 Orchestration Layer

Gemma 4 E2B (2B parameters) acts as the requirements gatherer and orchestrator. Given a high-level user command (e.g., “Create a SaaS landing page for a fintech product for enterprise customers”), the orchestrator engages in a conversational dialogue to resolve ambiguities (visual style, target audience, sections, colour scheme, accessibility). After clarification, it performs task decomposition using chain-of-thought

prompting, producing an ordered list of section IDs with metadata-enriched prompts (industry, vibe, colour theme, accessibility). The orchestrator’s lightweight nature validates that logical reasoning in a bounded domain does not require a large model.

4.4.2 Execution Layer

Each section prompt is routed to Qwendean, the fine-tuned Qwen3-4B domain specialist. Qwendean receives the prompt along with a system prompt instructing clean React/TypeScript code using ShadCN and Tailwind, with thinking mode disabled (code-only output). LangGraph routes via conditional edges: the orchestrator node updates the shared state with the section list and maps each section to the Qwendean node sequentially.

4.4.3 Assembly and Validation Layer

A synthesizer node concatenates the generated sections, resolves import deduplication, ensures consistent prop types across shared components, and applies a final layout wrapper. A rubric-based validation step checks syntactic validity, correct ShadCN API usage, Tailwind class correctness, and vibe consistency. Sections that fail are flagged for regeneration (max three attempts) or send back to the orchestrator for prompt refinement.

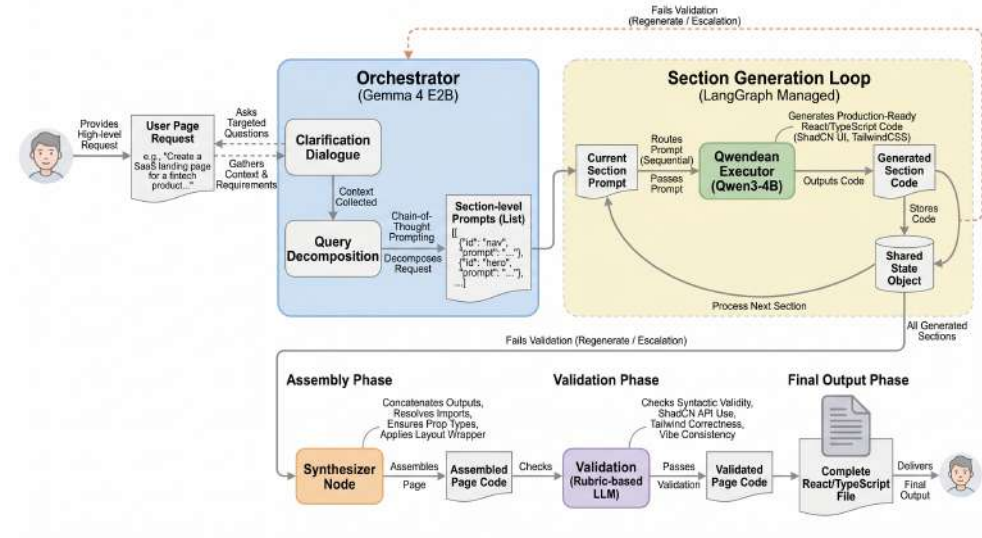


Fig. 4 Detailed agentic architecture flow (LangGraph implementation)

5 Results and Discussion

5.1 Training Dynamics Across Iterations

Three rounds clearly showed that dataset diversity is more important than raw size.

- **Round 1 (900 samples):** Training loss dropped to near zero, validation loss rose after 60 steps → severe overfitting. Outputs were often gibberish.
- **Round 2 (2,400 samples):** Train-validation gap narrowed; overfitting postponed. However, systematic mistakes remained: incorrect ShadCN props, confusion between Tailwind v3 and v4 classes, valid structure but wrong content.
- **Round 3 (4,100 samples):** Validation loss declined smoothly over 3 epochs; early stopping fired later, indicating continued meaningful learning. Qualitative evaluation showed correct ShadCN API usage, idiomatic Tailwind classes, and clean assembly without manual fixes.



Fig. 5 Training loss decreasing over steps (Round 3)

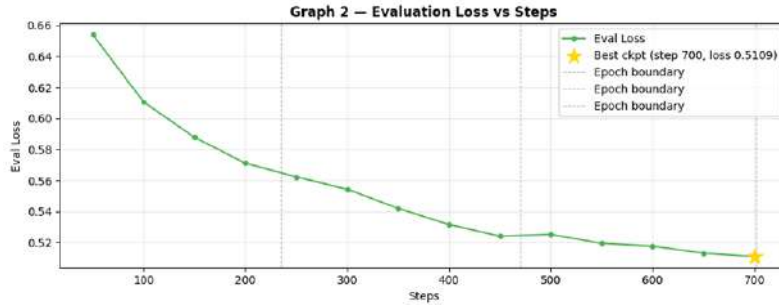


Fig. 6 Training and validation loss comparison

5.2 Model Configuration Insights

rsLoRA at rank 32 was crucial. Preliminary experiments with standard LoRA (rank 16) showed unstable gradient norms during the first epoch. Switching to rsLoRA produced stable gradient norms, a smooth learning rate decay, and a minimum validation loss of 0.5109 at step 700.

Training in 16-bit precision (instead of 4-bit QLoRA) improved output quality at the cost of higher VRAM (24–32GB total). The 4-bit approach introduced quantisation noise that caused syntactic issues in long completions. Sequence length 4096 covered the 99th percentile of training samples; packing was disabled to avoid truncated partial completions.

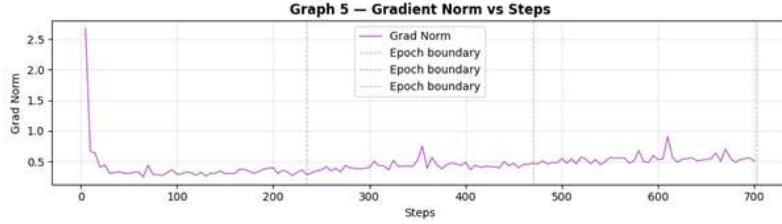


Fig. 7 Gradient norm stabilisation with rsLoRA



Fig. 8 Learning rate decay profile (linear warmup + decay)

5.3 Agentic System Performance

End-to-end evaluation compared our heterogeneous pipeline with a homogeneous baseline (GPT-4o performing both decomposition and section generation).

Output consistency: Qwendean sections showed much higher consistency in component API usage, naming conventions, and Tailwind idioms than GPT-4o sections, which varied with prompt phrasing. A fine-tuned SLM internalises a consistent coding style from its training corpus, whereas a general-purpose LLM samples from a broader distribution.

Latency and cost: Section generation on a local RTX 4060 Laptop GPU took 3–5s per section vs. 8–16s for GPT-4o API calls—a 3–5 \times reduction. API cost for Qwendean is effectively zero beyond infrastructure cost.

Table 1 Homogeneous vs. heterogeneous system performance

Metric	Homogeneous (GPT-4o)	Heterogeneous (Ours)
Output consistency	Varies with prompt phrasing	Consistent API/naming/styling
Inference latency (per section)	8–16s	3–5s
API cost	Non-trivial per token	Zero (local inference)
Orchestration latency	N/A	<4s per decomposition

Orchestration quality: Gemma 4 E2B consistently produced well-structured section lists with appropriate metadata for e-commerce, SaaS, and portfolio use cases. The dialogue resolved ambiguities, and decomposition outputs were logical and correctly scoped.

5.4 Orchestration with Lightweight General-Purpose Models

Employing Gemma 4 E2B (2B parameters) for orchestration validates the hypothesis that requirements gathering and structured task decomposition are not compute-intensive when the problem domain is well-scoped. The 2B model successfully performed logical reasoning, contextual question formulation, and metadata synthesis. Orchestration latency was under 4s per decomposition, and handoff to Qwendean was seamless.

This result has far-reaching implications: functional specialisation, not parameter count, is the source of heterogeneity advantage. A 2B generalist orchestrator + 4B specialist executor outperforms homogeneous architectures of any scale. The efficiency gain comes from matching model capacity to task requirements.

5.5 Limitations

- Quantitative baselines (pass@k, human preference scores) will be addressed in future work; current evaluation is primarily qualitative.
- Dataset size (4100 samples) is sufficient for proof-of-concept but not for covering all possible UI components.
- Validation currently uses LLM-based rubric checks; AST/compiler-level verification is not yet integrated.

6 Future Work

Several directions are identified:

- **Dataset scaling:** Expand to 30,000–40,000 high-quality samples from additional open-source libraries, developer-collected prompts, and adversarial examples.
- **Base model upgrades:** Fine-tune next-generation models with better code generation benchmarks.
- **MCP server integration:** Expose Qwendean as a Model Context Protocol (MCP) server, allowing any MCP-compatible LLM client (Claude, GPT-4o, Gemini) to invoke it as a tool for UI section generation.

- **Code verification layers:** Integrate AST parsing, TypeScript type checking (`tsc`), and LLM-based reflection nodes for targeted regeneration.
- **Preference optimisation:** Apply Direct Preference Optimisation (DPO) using human annotations on code readability, reusability, and accessibility.
- **Domain generalisation:** Validate the heterogeneous architecture for API integration code, database schema design, and automated test case generation.
- **Electron demonstration app:** A full desktop application built with Electron has been developed to showcase the end-to-end functionality of the heterogeneous agentic pipeline, allowing users to interactively generate UI pages and export React code.

7 Conclusion

We have demonstrated that heterogeneous agentic AI architectures—combining general-purpose model orchestration with domain-specific fine-tuned generation—offer clear advantages over homogeneous architectures for constrained generation problems. For automated UI code generation, our system uses Gemma 4 E2B as a lightweight requirements elicitation and task planner and Qwendean (fine-tuned Qwen3-4B) as a domain-specific generator. Qwendean produces consistent, idiomatic, and prompt-compliant code with much lower inference variance than general-purpose models.

Crucially, an effective heterogeneous architecture does not require scale-based heterogeneity. A 2B orchestrator paired with a 4B executor, specialised in its domain, outperforms homogeneous architectures of any scale for well-defined problems. Orchestrators rely on logical reasoning within a bounded space, which general-purpose SLMs can handle; code generation requires deep knowledge of syntax, libraries, and idioms, which can be acquired via fine-tuning.

The development process from 900 to 4,100 multi-library samples across three training iterations provides practical insights into how dataset quality and diversity affect fine-tuning dynamics for code generation. The validated configuration (rsLoRA, rank 32, 16-bit, 3 epochs, 4096 context) is justified by the observed trajectories and quality evaluations. The LangGraph-based pipeline with Qwendean is a practical, extensible architecture for other domain-specific heterogeneous AI systems. As LLMs evolve toward tool-assisted, multi-agent systems, fine-tuned SLMs will become increasingly essential for production-level agentic AI.

Declarations

Funding

Not applicable.

Conflict of interest

The authors declare no competing interests.

Ethics approval and consent to participate

Not applicable.

Consent for publication

All authors consent to publication.

Data availability

The dataset `iamdyeus/ui-instruct-4k` and the fine-tuned model `iamdyeus/qwendean-4b-GGUF` are publicly available on Hugging Face. Code is available at <https://github.com/iamDyeus/qwendean>.

Materials availability

All materials are open source.

Code availability

The fine-tuning and inference code is available at the GitHub repository above.

Author contribution

Arsh: conceptualisation, methodology, fine-tuning experiments, training pipeline implementation, LangGraph orchestration, integration of the agentic system, development of the Electron demonstration application, manuscript writing and revision. **Swarnendu Ghosh:** dataset construction, synthetic prompt augmentation (expansion from 900 to 4100 samples), initial LangGraph design, validation support. **Akshay Rajput:** supervision, mentorship, project guidance, and reviewing the manuscript.

References

- [1] P. Belcak and R. Wattenhofer, “Small Language Models are the Future of Agentic AI,” *arXiv preprint arXiv:2506.02153*, Jun. 2025.
- [2] J. Wei, X. Wang, D. Schuurmans, et al., “Chain-of-Thought Prompting Elicits Reasoning in Large Language Models,” in *Proc. NeurIPS*, Lake Tahoe, NV, USA, Dec. 2022, pp. 1–21.
- [3] LangChain, “LangGraph: Multi-Agent Orchestration,” [Online]. Available: <https://python.langchain.com/docs/langgraph/>. Accessed: Nov. 2025.
- [4] T. Pöppel, M. Eisenschlos, J. Nijkamp, et al., “On the Transfer Learning Capabilities of Code Models,” *arXiv preprint arXiv:2409.15895*, Sep. 2024.
- [5] Qwen Team, “Qwen3 Technical Report,” *arXiv preprint arXiv:2505.09388*, May 2025.

- [6] DeepSeek-AI, “DeepSeek-V3 Technical Report,” *arXiv preprint arXiv:2412.19437*, Dec. 2024.
- [7] R. Rafailov, S. Sharma, H. Zhang, et al., “Direct Preference Optimization: Your Language Model is Secretly a Reward Model,” *arXiv preprint arXiv:2305.18290*, May 2023.
- [8] S. Wang, Y. Jiang, H. Xu, Z. Liu, C. C. Elkan, and S. Jain, “Want To Reduce Labeling Cost for Your Dataset: Read This,” in *Proc. NeurIPS Datasets and Benchmarks Track*, 2021, pp. 1–15.
- [9] D. D. Ruiz, D. D. García, and J. J. Gómez, “Automating Web UI Design: Current Approaches and Future Challenges,” *IEEE Trans. Softw. Eng.*, vol. 49, no. 6, pp. 3150–3170, Jun. 2023.
- [10] ShadCN UI Documentation. [Online]. Available: <https://ui.shadcn.com/docs>. Accessed: Nov. 2025.
- [11] TailwindCSS Documentation. [Online]. Available: <https://tailwindcss.com>. Accessed: Nov. 2025.
- [12] E. J. Hu, Y. Shen, P. Wallis, et al., “LoRA: Low-Rank Adaptation of Large Language Models,” *arXiv preprint arXiv:2106.09685*, Jun. 2021.
- [13] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, “QLoRA: Efficient Finetuning of Quantized LLMs,” *arXiv preprint arXiv:2305.14314*, May 2023.
- [14] Unsloth, “Unsloth: Optimized LLM Fine-Tuning,” [Online]. Available: <https://github.com/unslothai/unsloth>. Accessed: Nov. 2025.
- [15] Hugging Face, “Transformers Library,” [Online]. Available: <https://huggingface.co/docs/transformers>. Accessed: Nov. 2025.
- [16] PyTorch Foundation, “PyTorch Documentation,” [Online]. Available: <https://pytorch.org>. Accessed: 2025.
- [17] D. Kalajdzievski, “A Rank Stabilization Scaling Factor for Fine-Tuning with LoRA,” *arXiv preprint arXiv:2312.03732*, Nov. 2023.