Abstract

Large language models can discuss system design fluently, yet their advice is fundamentally ungrounded: they suggest adding a Redis caching layer without checking whether the codebase already has one, or recommend Kubernetes for a 200-user internal tool. We introduce RepoDesign, a multimodal agent pipeline that accepts a product specification, an existing code repository, and (optionally) architecture diagrams, and produces a codebase-aware implementation plan—complete with real file paths, scale-appropriate technology choices, and executable ticket descriptions. Our core contributions are: (1) a structured Repo Intermediate Representation (Repo IR) that captures a repository's architecture through deterministic code analysis augmented by LLM summarization; (2) a multimodal fusion mechanism that grounds visual architecture diagrams against the Repo IR; and (3) a scale-aware design reasoning framework that conditions recommendations on project constraints (team size, user count, budget) rather than defaulting to enterprise-grade patterns. We fine-tune Qwen2-VL-7B-Instruct via QLoRA with supervised fine-tuning (SFT) and Direct Preference Optimization (DPO), and evaluate using a novel Repo Grounding Score that measures whether generated plans reference files and modules that actually exist in the target repository.

1. Introduction

System design is one of the most consequential activities in software engineering. A poor architectural decision made in week one can cost months of rework later. Yet today's AI-assisted development tools leave a critical gap in this process. Code-generation tools such as GitHub Copilot and Cursor excel at implementing individual functions but offer no guidance on what to build or where to put it within an existing codebase. Conversational AI assistants like ChatGPT and Claude can discuss system design concepts eloquently, but their recommendations are generic—decoupled from any actual repository, blind to existing infrastructure, and systematically biased toward over-engineered solutions.

This gap matters because real system design is always contextual. The right architecture for a 50-person engineering team serving 10 million users is radically different from the right architecture for a 2-person startup serving 500 users. As Kleppmann (2017) emphasizes, the value of a design checklist is not implementing everything but consciously deciding which items matter for your specific system. Current AI tools fail precisely at this contextual judgment: they cannot examine your codebase to see that you already use Django with PostgreSQL, they do not notice that your docker-compose.yml reveals a single-server deployment, and they will recommend sharding your database regardless of whether you have 1,000 or 10 million rows.

Broad Impact. A tool that produces repo-grounded, scale-appropriate design plans would democratize architectural expertise. Junior engineers at small companies—who lack access to senior architects—would receive guidance calibrated to their actual context rather than Silicon

Valley-scale assumptions. This could meaningfully reduce architectural debt, a leading cause of software project failure (Besker et al., 2018).

Intellectual Merit. This problem is technically challenging for several reasons. First, extracting a faithful representation of a repository's architecture requires fusing heterogeneous signals: dependency graphs, ORM models, API route decorators, infrastructure-as-code files, and architectural diagrams in various visual formats. Second, generating plans that are grounded—referencing real files, respecting existing patterns, and avoiding hallucinated paths—requires the model to reason over structured context in a way that current LLMs struggle with. Third, scale-aware reasoning demands that the model learn when not to recommend complexity, which runs counter to the bias in most training data toward large-scale systems. Finally, the multimodal aspect—understanding architecture diagrams alongside code and text—requires non-trivial cross-modal alignment.

2. Related Work

Prior work relevant to RepoDesign spans four major research themes: code-aware language models, automated software architecture, multimodal document understanding, and constraint-aware planning and preference alignment.

2.1 Code-Aware Language Models (and the long-context baseline)

Large language models trained or fine-tuned on source code have advanced rapidly. Early work such as CodeBERT (Feng et al., 2020) and CodeT5 (Wang et al., 2021) introduced pre-training objectives tailored to code understanding. StarCoder (Li et al., 2023) and Code Llama (Rozière et al., 2023) scaled this to fill-in-the-middle and long-context code generation. The field has since shifted decisively toward repository-level reasoning: RepoCoder (Zhang et al., 2023) uses iterative retrieval to generate code consistent with broader repository context, SWE-Agent (Yang et al., 2024) interacts with entire repositories to resolve real GitHub issues, and the SWE-bench benchmark (Jimenez et al., 2024) has become the standard evaluation for repo-level code agents. The open-source OpenHands framework (Wang et al., 2024b) and commercial systems like Devin (Cognition Labs, 2024) push this further toward autonomous multi-step software engineering. On the model side, Qwen2.5-Coder (Hui et al., 2024) and DeepSeek-Coder-V2 (Zhu et al., 2024) demonstrate that open-weight models can approach frontier performance on code tasks.

2025+ shift: long-context repo ingestion as the “obvious” baseline. Reviewers will reasonably ask: *why build a Repo IR when models can ingest extremely long contexts?* Recent evaluations show that long-context competence—especially for code and repositories—is not solved by window size alone. LongBench v2 explicitly includes code repository understanding with contexts up to massive lengths and highlights persistent failures in using far-context evidence even when present (LongBench v2, 2024/2025). Additional work specifically targets training models to fully utilize long repository context (e.g., CoLT, 2025), indicating that “full-repo prompting” can still suffer from attention diffusion, structural underuse, and high inference cost.

This motivates RepoDesign's design choice: **Repo IR is not competing with long-context; it is a structural scaffold.**Instead of relying on the model to discover architectural signals across thousands of files, Repo IR deterministically surfaces high-signal structure (routes, ORM models, dependency topology, infra configs) and compresses it into a stable representation that downstream planners can reliably attend to. Despite rapid progress, existing systems still overwhelmingly focus on code generation and bug fixing—writing or editing code—rather than architectural reasoning—deciding what to build, where to put it, and at what level of complexity. RepoDesign targets this upstream planning step that precedes code generation.

2.2 Automated Software Architecture and Agentic SWE Workflows ("flow engineering")

Several lines of work attempt to automate aspects of software architecture. Early approaches used formal architecture description languages (ADLs) and model-driven engineering (Medvidovic & Taylor, 2000). More recently, Chen et al. (2023) explored using LLMs to generate UML diagrams from natural language requirements, while ChatDev (Qian et al., 2024) and MetaGPT (Hong et al., 2024) simulate multi-agent software development teams that produce design documents and code. However, these systems generate architectures de novo without awareness of an existing codebase. They also uniformly target greenfield projects, ignoring the more common (and harder) scenario of extending an existing system.

In parallel, a newer thread emphasizes **structured agentic workflows**—often described as *flow engineering*—where models iteratively plan, generate, and verify outputs (e.g., via tests or execution). AlphaCodium formalizes this approach for code generation by combining structured decomposition with verification loops (AlphaCodium, 2024/2025). In 2025, the ecosystem has expanded toward scalable data and environments for training SWE agents and verifiers (e.g., SWE-smith; SWE-Gym, 2025), and scalable benchmark generation frameworks (e.g., SWE-Bench++, 2025). These advances largely optimize for *patch correctness* (does the code compile/pass tests) rather than *design correctness* (is the plan grounded in an existing architecture and appropriate for constraints).

RepoDesign specifically addresses this gap by grounding design decisions in the Repo IR of an existing codebase and by explicitly conditioning recommendations on project scale. Conceptually, RepoDesign's "verifier" is not a unit test suite but the repository architecture itself (via Repo Grounding Score) and a constraint rubric (via Scale Appropriateness and Constraint Satisfaction).

2.3 Multimodal Document, Diagram Understanding, and Grounding

Vision-language models have made significant progress in understanding technical documents and diagrams. Models like Qwen2-VL (Wang et al., 2024a) and LLaVA (Liu et al., 2023) can interpret charts, diagrams, and screenshots. In the software domain, specific work has explored understanding UML diagrams (Kimura et al., 2023) and translating whiteboard sketches to structured representations. However, no prior work has attempted to ground visual architecture diagrams against an actual codebase—connecting a box labeled "User Service" in a diagram to the actual src/services/user/ directory in the repository.

A closely related 2025+ line of work is **GUI grounding**, where models map visual UI elements to structured substrates like DOM trees or accessibility trees. Systems such as Ferret-UI and OSWorld-style grounding benchmarks emphasize pixel-to-structure alignment and tool-usable grounding. RepoDesign’s diagram grounding can be seen as the architectural analogue: mapping visual nodes/edges in architecture diagrams to the repository’s structural representation (Repo IR). This framing places RepoDesign within a contemporary grounding paradigm: **visual-to-structured alignment enabling downstream action**, rather than diagram understanding in isolation.

2.4 Constraint-Aware Planning and Preference Alignment (scale as a first-class constraint)

The challenge of grounding LLM outputs in factual context has been extensively studied through retrieval-augmented generation (Lewis et al., 2020) and tool-augmented LLMs (Schick et al., 2024). Planning with LLMs has been explored in robotics (SayCan, Ahn et al., 2022) and task decomposition (Wang et al., 2023). However, for system design, correctness is not only about feasibility; it is about **appropriateness under constraints**—especially scale.

Recent 2025 work in alignment provides a useful lens: preference optimization and reward modeling are increasingly used to enforce **constraint satisfaction and process quality**, not just outcome correctness (e.g., process reward modeling lines of work such as SP-PRM, 2025). RepoDesign’s scale-aware reasoning fits naturally into this framing: the goal is to align the model’s design process with a constraint rubric (team size, expected users, budget, timelines) and penalize both over-engineering and under-engineering. Our use of DPO over contrastive “scale-tier” pairs operationalizes this: plans are judged not only by whether they could work, but whether they are *the right level of complexity* for the stated scale.

Summary of Gaps. Existing work either (a) generates code without architectural reasoning, (b) reasons about architecture without codebase awareness, (c) understands diagrams without grounding them in code, or (d) ignores scale as a conditioning variable (or treats it implicitly). RepoDesign addresses all four gaps in a unified multimodal pipeline.

3. Research Ideas

Based on the identified gaps, we propose three research ideas, one per team member, which converge into the unified RepoDesign system.

Research Idea 1: Repo IR Extraction and Grounding (Member A)

Idea: Develop a structured Intermediate Representation (Repo IR) that captures a repository's architecture by combining deterministic static analysis (dependency graphs via pipreqs/madge, API route extraction via AST parsing, ORM model parsing, infrastructure config parsing) with LLM-based summarization of architectural patterns. The key research question is whether a fine-tuned 7B model can produce faithful, structured Repo IRs that enable downstream grounding—i.e., whether implementation plans that reference the Repo IR contain significantly fewer hallucinated file paths than plans generated without it.

Novelty: While repository-level retrieval has been explored for code generation (RepoCoder), no prior work has defined a canonical structured representation of repository architecture specifically designed for downstream design tasks. The Repo IR is a contribution in itself as a benchmark artifact.

Research Idea 2: Multimodal Architecture Diagram Fusion (Member B)

Idea: Extend the pipeline to accept architecture diagrams (PlantUML renders, Mermaid diagrams, hand-drawn whiteboard photos, cloud console screenshots) as an additional input modality. The model must parse visual diagrams, identify components and their relationships, and align them with entities in the Repo IR. For example, a box labeled "Auth Service" in a diagram should be mapped to the src/auth/ directory, and a line between "API Gateway" and "Database" should correspond to the actual call chain in the code.

Novelty: Prior multimodal work on software diagrams focuses on diagram generation (code → diagram) or diagram understanding in isolation. We propose diagram grounding: aligning visual architectural representations with real code artifacts. This is a genuinely multimodal task that requires cross-modal reasoning between visual structure and code structure.

Research Idea 3: Scale-Aware Design Reasoning (Member C)

Idea: Train the model to condition its architectural recommendations on project scale constraints (expected user count, team size, budget, timeline). A scale-aware model should recommend a simple monolithic deployment for a 500-user internal tool, but propose a microservices architecture with load balancing for a 5M-user consumer product—even when the functional requirements are identical. We operationalize this by creating contrastive training pairs where the "preferred" plan matches the scale and the "rejected" plan over-engineers or under-engineers for the given constraints.

Novelty: To our knowledge, no prior work explicitly trains or evaluates LLMs on scale-appropriate system design. This is a novel evaluation axis and a practical differentiator. We operationalize scale-awareness using a comprehensive system design decision framework derived from Kleppmann (2017) and established engineering practice. This framework decomposes system design into 20 decision dimensions—including performance requirements, consistency models, caching strategy, replication topology, partitioning schemes, security posture, and failure resilience—and defines, for each dimension, what the appropriate recommendation looks like at four scale tiers: hobby (<1k users, 1 developer), startup(1k–50k users, 2–5 developers), growth (50k–1M users, 10–30 developers), and enterprise (1M+ users, 30+ developers). For example, at the hobby tier, the correct caching recommendation is often "none—your database is fast enough"; at growth tier, it might be "Redis with cache-aside pattern"; at enterprise tier, it involves multi-layer caching with CDN, application-layer, and database query caching with stampede prevention. This structured decision framework serves both as training signal (ground-truth for contrastive pairs) and evaluation rubric (does the model's recommendation match the tier-appropriate choice for each applicable dimension?).

4. Planned Execution

4.1 Datasets

Repo IR Training Data. We will curate approximately 200 well-structured open-source GitHub repositories spanning multiple scales: ~70 small projects (single developer, <5k users), ~70 medium projects (small team, 5k–100k users), and ~60 large projects (large team, 100k+ users). For each repository, we will run deterministic extraction tools and generate silver-label Repo IR JSON using GPT-4/Claude, then manually correct 50–100 examples to serve as gold-standard validation data.

Implementation Plan Training Data. For each repository, we will generate 2–3 synthetic PRDs (product requirement documents) reflecting features at appropriate scale. Using GPT-4/Claude, we will generate gold-label implementation plans, then create negative examples (hallucinated file paths, over-engineered solutions for small projects, under-engineered solutions for large projects) to form preference pairs for DPO training.

Architecture Diagram Data. We will collect real architecture diagrams from repositories that include .puml, .mmd(Mermaid), or architecture.md files. We will supplement this with rendered versions of these diagrams and synthetically degraded versions (simulating whiteboard photos) to create a diverse visual input set. We estimate ~100–150 diagram-repo pairs.

Scale-Awareness Contrastive Pairs. For each of 50+ scenarios, we will generate pairs of plans at different scale tiers with the same functional requirements but different non-functional constraints. Using the 20-dimension system design framework as ground truth, the "preferred" plan makes tier-appropriate choices for each dimension (e.g., recommending SQLite + single server for a hobby project, or multi-region replication + CDN for an enterprise platform), while the "rejected" plan systematically over-engineers (e.g., Kubernetes + microservices + Kafka for a 200-user app) or under-engineers (e.g., single-server monolith for a 2M-user global service). This framework ensures the preference signal is principled rather than arbitrary.

4.2 Evaluation Metrics

Repo Grounding Score (RGS). The percentage of file paths mentioned in the generated plan that actually exist in the target repository. This is our primary automated metric and is fully deterministic—no LLM-as-judge subjectivity.

Plan Completeness Score. A checklist-based score derived from a 20-dimension system design framework (covering problem scope, functional requirements, performance, availability, consistency, durability, API design, data modeling, caching, replication, partitioning, networking, message queues, compute architecture, batch/stream processing, security, privacy, failure resilience, monitoring, and deployment). For each generated plan, we evaluate which applicable dimensions are addressed. Critically, this metric is scale-conditioned: a plan for a 500-user tool should not be penalized for omitting a sharding strategy, but a plan for a 5M-user platform should. We weight dimensions by relevance to the spec's stated scale tier and requirements.

Scale Appropriateness Score. An automated metric that checks whether the recommended technologies and patterns match the stated scale tier. For example, recommending Kubernetes for a project with team_size: 2 and expected_users: 500 would be penalized; recommending a single-server deployment for expected_users: 5M would also be penalized. We will define a rubric mapping scale tiers to acceptable/unacceptable patterns, derived from the Kleppmann-based system design checklist.

Constraint Satisfaction Rate. The percentage of hard constraints from the spec (e.g., "must use PostgreSQL," "no new infrastructure") that the generated plan respects.

Format Compliance. Whether the output parses as valid JSON conforming to our defined schemas.

Human Evaluation. We will recruit 10–20 software engineers (graduate students and industry contacts) for a comparative study. Each evaluator will see two implementation plans for the same repo + spec (one from our model, one from a baseline GPT-4 prompt) and answer: (a) "Which plan would you execute Monday morning?" (b) "Which better matches the project's scale?" (c) "Which references real code more accurately?"

4.3 Experimental Setup

Research Questions and Hypotheses:

RQ1 (Grounding): Does the Repo IR improve grounding compared to naive RAG over raw repository files? **Hypothesis:** Plans generated with Repo IR will achieve ≥30% higher Repo Grounding Score than plans using raw file retrieval.

RQ2 (Multimodal Fusion): Does incorporating architecture diagrams as input improve plan quality compared to text-only specs? **Hypothesis:** For repositories that have architecture diagrams, including the diagram will improve Plan Completeness Score and reduce hallucinated components.

RQ3 (Scale Awareness): Does DPO training with scale-contrastive pairs produce more scale-appropriate recommendations? **Hypothesis:** After DPO, the model will recommend simpler architectures for small-scale projects at ≥2x the rate of the SFT-only baseline.

RQ4 (End-to-End): Do engineers prefer RepoDesign's output over prompted GPT-4?
Hypothesis: In human evaluation, RepoDesign will be preferred ≥60% of the time on the "execute Monday morning" question.

Baselines:

- GPT-4 with a detailed system design prompt (no repo awareness)
- GPT-4 with naive RAG over repository files (no structured Repo IR)
- SFT-only model (no DPO, to isolate the effect of preference tuning)

Model and Training: We will fine-tune Qwen2-VL-7B-Instruct using QLoRA via Unsloth. Training proceeds in two stages: (1) SFT on Repo→IR extraction and Spec+IR→Plan generation, and (2) DPO on good-vs-bad plan preference pairs, with emphasis on scale-appropriateness and grounding.

Risks and Mitigations:

- **Risk:** Repo IR extraction is noisy for poorly structured repositories. **Mitigation:** Focus evaluation on well-structured repos; report performance stratified by repo quality.
- **Risk:** 200 repos may be insufficient for robust fine-tuning. **Mitigation:** Use data augmentation (multiple synthetic PRDs per repo) and fall back to a RAG-based approach if fine-tuning underperforms.
- **Risk:** Multimodal diagram data is scarce. **Mitigation:** Supplement with synthetic diagram renders from Mermaid/PlantUML definitions found in repos, and evaluate multimodal contribution as a delta over text-only baseline.
- **Risk:** Human evaluation sample size is small. **Mitigation:** Use paired comparisons (more statistically efficient) and report confidence intervals.

5. Task Division and Timeline

Team Roles

Member A (Repo IR & Data Pipeline): Owns the deterministic repo analysis tools, Repo IR schema definition, repo scraping pipeline, and Repo Grounding Score evaluation. Responsible for curating the 200-repo dataset.

Member B (Multimodal & Training): Owns the diagram extraction pipeline, multimodal input formatting, QLoRA/SFT/DPO training runs, and model evaluation. Responsible for the diagram-repo alignment task.

Member C (Scale Reasoning & Evaluation): Owns the scale-awareness rubric, contrastive pair generation, human evaluation study design, and demo/UI. Responsible for the Spec Normalizer and scale-tier classification.

All members will collaboratively write the final report and contribute to experiment design. Code reviews will be cross-functional to ensure shared understanding of the full pipeline.

Timeline (Biweekly Milestones)

Weeks 1–2 (Foundation): Define JSON schemas for Spec, Repo IR, and Implementation Plan. Build Repo IR extraction pipeline v1 (deterministic tools). Curate initial list of 200 target repos stratified by scale. Build Spec Normalizer (prompt-based). Write diagram mining script. Set up training infrastructure (Unsloth, QLoRA).
Weeks 3–4 (Data Generation): Run Repo IR extraction on all 200 repos. Generate silver-label Repo IRs using GPT-4/Claude. Manually validate 50+ Repo IRs. Generate synthetic PRDs and gold-label implementation plans. Create scale-contrastive preference pairs. Collect and render architecture diagrams from repos.
Weeks 5–6 (Training & Initial Evaluation): First SFT training run (Repo→IR). Second SFT training run (Spec+IR→Plan with multimodal diagram input). Implement Repo Grounding Score, Plan Completeness, and Scale Appropriateness automated metrics. Evaluate SFT model, identify failure modes, iterate on data quality.
Weeks 7–8 (DPO & Refinement): Build DPO preference dataset from SFT model outputs + scale-contrastive pairs. Run DPO training. Compare SFT vs. DPO on all automated metrics. Build demo UI (Gradio/Streamlit). Begin human evaluation recruitment.
Weeks 9–10 (Evaluation & Demo): Run full automated evaluation harness across all baselines. Conduct human evaluation study (10–20 participants). Prepare before/after demo examples. Compile results and begin writing final report.
Weeks 11–12 (Final Report & Presentation): Complete final report. Prepare presentation with live demo (real repo + spec → grounded plan). Submit code, data, and trained model artifacts to GitHub repository.

References

Ahn, M., et al. (2022). Do As I Can, Not As I Say: Grounding Language in Robotic Affordances. arXiv:2204.01691.
Besker, T., Martini, A., & Bosch, J. (2018). Managing architectural technical debt: A unified model and systematic literature review. JSS, 135, 1–16.
Chen, M., et al. (2023). Towards automated UML generation from natural language requirements using LLMs. ASE Workshop.
Cognition Labs. (2024). Devin: The First AI Software Engineer. Blog post.
Feng, Z., et al. (2020). CodeBERT: A Pre-Trained Model for Programming and Natural Languages. EMNLP.
Hong, S., et al. (2024). MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework. ICLR.
Hui, B., et al. (2024). Qwen2.5-Coder Technical Report. arXiv:2409.12186.
Jimenez, C. E., et al. (2024). SWE-bench: Can Language Models Resolve Real-World GitHub Issues? ICLR.
Kimura, T., et al. (2023). Understanding UML diagrams with vision-language models. ICSE

Companion.
Kleppmann, M. (2017). Designing Data-Intensive Applications. O'Reilly.
Lewis, P., et al. (2020). Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. NeurIPS.
Li, R., et al. (2023). StarCoder: May the Source Be With You. arXiv:2305.06161.
Liu, H., et al. (2023). Visual Instruction Tuning. NeurIPS.
Medvidovic, N., & Taylor, R. N. (2000). A classification and comparison framework for software architecture description languages. IEEE TSE, 26(1), 70–93.
Qian, C., et al. (2024). ChatDev: Communicative Agents for Software Development. ACL.
Rozière, B., et al. (2023). Code Llama: Open Foundation Models for Code. arXiv:2308.12950.
Schick, T., et al. (2024). Toolformer: Language Models Can Teach Themselves to Use Tools. NeurIPS.
Wang, X., et al. (2024b). OpenHands: An Open Platform for AI Software Developers as Generalist Agents. arXiv:2407.16741.
Wang, Y., et al. (2021). CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. EMNLP.
Wang, P., et al. (2024a). Qwen2-VL: Enhancing Vision-Language Model's Perception of the World at Any Resolution. arXiv:2409.12191.
Wang, L., et al. (2023). Plan-and-Solve Prompting: Improving Zero-Shot Chain-of-Thought Reasoning by Large Language Models. ACL.
Yang, J., et al. (2024). SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. arXiv:2405.15793.
Zhang, F., et al. (2023). RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation. EMNLP.
Zhu, Q., et al. (2024). DeepSeek-Coder-V2: Breaking the Barrier of Closed-Source Models in Code Intelligence. arXiv:2406.11931.