

PROJECT WORK CS6811
FIRST REVIEW

Federated Learning for Code Generation with Hallucination-Aware Fine Tuning

TEAM DETAILS

Abhinav P (2022503059),
Prabhakara Arjun R (2022503003),
Buvanes Srivardhan K (2022503037)

MENTOR

Dr. B. Thanasekhar,
Professor,
Computer Technology, MIT

DOMAIN: Large Language Models

INTRODUCTION

Large Language Models (LLMs) have remarkable capabilities in code generation, enabling automated solutions to a wide range of software engineering tasks. However, LLMs frequently suffer from hallucinations in code generation, producing syntactically valid but semantically incorrect programs, non-existent APIs, or logically flawed implementations. Existing approaches to mitigating code hallucinations primarily rely on prompt engineering, repeated sampling, or full model fine-tuning. Full fine-tuning is computationally expensive and impractical in low-resource or decentralized settings, and repeated sampling does not address the underlying causes or mitigation of hallucinations. In real-world development environments, code generation systems are often deployed across multiple decentralized clients where data cannot be centrally aggregated due to privacy and ownership constraints..

To address these limitations, we propose a hallucination-aware code generation framework that combines program analysis, targeted automatic program repair, and federated parameter-efficient fine-tuning using Low-Rank Adaptation (LoRA).

PROBLEM STATEMENT

Code generation using Large Language Models (LLMs) is currently hindered by hallucinations. Existing mitigation strategies typically rely on supervised fine-tuning or reinforcement learning from human feedback (RLHF), where models are trained to align with human preferences. While parameter-efficient fine-tuning methods such as LoRA reduce computational overhead compared to full fine-tuning, both approaches primarily optimize model behavior without explicit verification of program correctness. As a result, these methods may improve surface-level correctness but often fail to systematically detect and correct deeper structural, semantic, and execution-level errors in generated code.

Furthermore, in real-world scenarios, high-quality code and execution feedback are distributed across multiple organizations, making centralized data collection infeasible due to privacy concerns. Without a secure and decentralized learning mechanism, models are unable to effectively learn from diverse hallucination patterns observed across clients. This creates a critical gap for a unified framework that integrates explicit hallucination detection and automated program verification with privacy-preserving, parameter-efficient adaptation to improve the quality of code generation.

OBJECTIVES

- To design a hallucination-aware code generation framework that integrates static and dynamic program analysis.
- To detect and classify different types of hallucinations in generated code using oracle-based verification.
- To apply automatic program repair for correcting hallucinated code using knowledge-guided patching.
- To fine-tune large language models using parameter-efficient Low-Rank Adaptation (LoRA) in a federated learning setting.
- To evaluate the proposed framework on standard code generation benchmarks and analyse hallucination reduction.

LITERATURE SURVEY

S.NO	TITLE	NAME OF THE JOURNAL	METHODOLOGY	LIMITATIONS
[1]	Advancing LLM-Generated Code Reliability: A Hybrid Approach for Hallucination Detection	IEEE Transactions on Software Engineering (TSE), 2025	<ul style="list-style-type: none">• Proposes SDHD, a hybrid hallucination detection framework that integrates static analysis and dynamic execution for LLM-generated code.• This framework detects eight hallucination types and achieves superior precision, recall, and F1-score across multiple benchmarks.	<ul style="list-style-type: none">• Dynamic detection depends on test coverage quality, and incomplete coverage may miss certain logical failures.• The framework is currently validated only on Python, limiting cross-language generalizability.
[2]	LLM-Based Test-Driven Interactive Code Generation: User Study and Empirical Evaluation	IEEE Transactions on Software Engineering (TSE), 2024	<ul style="list-style-type: none">• TICODER is a test-driven interactive workflow that incrementally clarifies user intent through LLM-generated tests and lightweight user feedback.• The system ranks and prunes candidate code suggestions based on user-approved tests using two variants: PASS/FAIL validation and explicit output specification.	<ul style="list-style-type: none">• There is interactive overhead, and incorrect or noisy user feedback can prune correct solutions prematurely.• Effectiveness is contingent on the LLM's ability to generate high-quality tests.

[3]	A Federated Adaptive Large Language Model Fine-Tuning Framework for Software Development	IEEE Transactions on Services Computing, 2025	<ul style="list-style-type: none"> The framework integrates LoRA-based parameter-efficient fine-tuning within a federated setting, where each organization locally trains lightweight adapter modules and shares only adapter to the central server. A FedAvg-style is used to combine adapters based on local data size and an L2-norm constraint to manage gradient divergence 	<ul style="list-style-type: none"> Experimental results show degraded performance on unrelated datasets. Although communication costs are reduced, the framework still requires multiple communication rounds, which may introduce latency in large-scale federations.
[4]	Towards Efficient Fine-Tuning of Language Models With Organizational Data for Automated Software Review	IEEE Transactions on Software Engineering 2024	<ul style="list-style-type: none"> Proposes CodeMentor, a framework for few-shot learning that utilizes heuristic rules and weak supervision to construct datasets from internal organizational data It employs Low-Rank Adaptation (LoRA) for parameter-efficient fine-tuning and integrates Reinforcement Learning from Human Feedback (RLHF) to incorporate domain expertise. 	<ul style="list-style-type: none"> The framework depends on the quality of organizational data, which may introduce domain bias and limit generalization across different coding lang. RLHF requires computational resources and expert involvement, bottleneck scenario during scaling.
[5]	The Impact of Prompt Programming on Function-Level Code Generation	IEEE Transactions on Software Engineering (TSE), 2025	<ul style="list-style-type: none"> Empirical study to evaluate the impact of prompt programming techniques on function-level code generation. CodePromptEval, a dataset of 7,072 prompts derived from 221 real-world Python functions, covering five prompt techniques: few-shot learning, chain-of-thought, persona, function signature, and package context, including all 32 possible combinations. 	<ul style="list-style-type: none"> The study is restricted to Python function-level tasks, limiting generalizability to other languages and higher-level program synthesis. Rely on benchmark-driven evaluation, which may not fully capture real-world scenario. Prompt techniques are evaluated in a fixed ordering, excluding effects of alternative prompt sequencing.

[6]	An Adaptive Framework Embedded With LLM for Knowledge Graph Construction	IEEE Transactions on Multimedia, Vol. 27, 2025	<ul style="list-style-type: none"> KG construction happens in three stages: Information Extraction (IE) for open triple extraction, Additional Relation (AR) for enriching triples with semantic information, and Knowledge Graph Normalization (KGN) for schema-level alignment using vector similarity search. Adaptive Prompt Generation (APG) module automatically generates, evaluates, and ranks prompts to guide LLM behavior across stages, enabling multi-domain KG construction from text, speech, and images. 	<ul style="list-style-type: none"> High computational overhead due to multi-stage LLM inference, embedding generation, and vector similarity matching. The framework relies heavily on prompt engineering, which,, is not fully autonomous and may still suffer from hallucination and semantic inconsistency.
[7]	Fight Fire With Fire: How Much Can We Trust ChatGPT on Source Code-Related Tasks?	IEEE Transactions on Software Engineering (TSE), Vol. 50, No. 12, 2024	<ul style="list-style-type: none"> Evaluating self-verification capability of GPT for: code generation, code completion, and program repair. Direct question, guiding question, and test report generation prompting is used assess whether ChatGPT can reliably validate its own outputs. Effectiveness is measured using accuracy, precision, recall, and F1-score, with additional analysis of self-contradictory hallucinations. 	<ul style="list-style-type: none"> Results show that ChatGPT frequently overestimates the correctness of its own outputs, leading to low recall in direct self-verification. While guiding questions and test reports improve recall, precision is reduced. Generated test reports often contain incorrect explanations, especially for code generation and failed repairs.

[8]	FedALoRA: Adaptive Local LoRA Aggregation for Personalized Federated Learning in LLM	IEEE Internet of Things Journal, Vol. 12, No. 24, December 2025	<ul style="list-style-type: none"> Proposes FedALoRA, a personalized federated learning framework that integrates LoRA-based PEFT with adaptive local aggregation to address severe non-IID data in federated LLM training. FedALoRA uses an Adaptive LoRA Aggregation to merge global and local parameters via locally learned weights. This selective, element-wise fusion focused on higher transformer layers preserves domain expertise while capturing shared knowledge without increasing communication costs. 	<ul style="list-style-type: none"> FedALoRA introduces additional local optimization complexity due to adaptive weight learning in ALoRA modules. The method requires careful tuning of the hyperparameter l, which controls the number of transformer layers involved in adaptive aggregation, and improper settings can degrade performance.
[9]	No Need to Lift a Finger Anymore? Assessing the Quality of Code Generation by ChatGPT	IEEE Transactions on Software Engineering (TSE), Vol. 50, No. 6, June 2024	<ul style="list-style-type: none"> Evaluation framework for assessing ChatGPT-based code generation. The authors construct standardized prompts for 728 LeetCode algorithm problems and 54 CWE security scenarios across five programming languages. Generated code is evaluated using automated judgment platforms (LeetCode for functional correctness) and static analysis tools (CodeQL for vulnerability detection). A multi-round fixing process is designed, where execution or analysis feedback is iteratively fed back 	<ul style="list-style-type: none"> ChatGPT performs significantly better on problems published before 2021, indicating reliance on memorized patterns rather than reasoning. The dialog-based fixing process has limited effectiveness, for logically complex problems . Generated code often contains security vulnerabilities and non-deterministic behavior.

[10]	Knowledge Enhanced Program Repair for Data Science Code	Proceedings of the 47th IEEE/ACM International Conference on Software Engineering (ICSE), 2025	<ul style="list-style-type: none"> API KG Construction is done using a custom ontology and RDF triples to model API attributes and dependencies for seven data science libraries. SPARQL is used for knowledge retrieval from on buggy code. AST level Bug Knowledge Enrichment, which executes code node-by-node to localize runtime and assertion errors at the AST node level. 	<ul style="list-style-type: none"> DS-KG is version-specific and requires continuous updates to track evolving API documentation, making long-term maintenance costly and error-prone. KG-based knowledge retrieval introduces higher latency than plain-text search
[11]	THINK: Tackling API Hallucinations in LLMs via Injecting Knowledge	IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2025	<ul style="list-style-type: none"> THINK proposes a two-phase API hallucination mitigation framework consisting of Pre and Post execution Optimization. Constructs an API knowledge database (Javadoc + validated code examples) and retrieves task-relevant APIs using task decomposition, similarity search, and LLM-based API labeling to guide code generation. Error Classification into Seven API Error Types for correction 	<ul style="list-style-type: none"> It may introduce irrelevant API knowledge, occasionally degrading performance for tasks already solvable by strong LLMs. Both retrieval results and ReAct-based repair depend on LLM generation, leading to non-deterministic behavior and reduced reproducibility across runs

[12]	EvoAPR: Enhancing Large Language Models for Automatic Program Repair with Genetic Algorithm and Dynamic LoRA	IEEE International Conference on Web Services (ICWS), 2025	<ul style="list-style-type: none"> Genetic evolution of buggy hunks under tri-objective fitness: syntax compliance, bug-hunk masking rate, repair-location masking rate. Dynamic LoRA swaps non-overlapping rank-r adapters per bug-index/project for on-demand, project-specific parameter retrieval. Base + LoRA models generate candidate patches; token-level entropy scorer picks the least uncertain fix as the final patch. 	<ul style="list-style-type: none"> Genetic template evolution and dynamic LoRA fine-tuning introduce significant computation and implementation complexity compared to direct prompting-based APR methods. The effectiveness of both stages relies on accurate bug localization and rich bug context.
[13]	CodeHalu: Investigating Code Hallucinations in LLMs via Execution Based Verification	Proceedings of the AAAI Conference on Artificial Intelligence (AAAI), 2025	<ul style="list-style-type: none"> Executes LLM-generated code against test cases under time and memory constraints, recording execution failures, unexpected outputs, infinite loop behaviors to detect hallucination states. Aggregates execution states across multiple LLMs and applies a two-stage heuristic induction process to classify hallucinations into four main types (Mapping, Naming, Resource, Logic) and eight subcategories. 	<ul style="list-style-type: none"> The study focuses exclusively on Python, and hallucination behaviors in other programming languages are not evaluated. CodeHalu targets functional and logical correctness; identifying or mitigating security vulnerabilities is outside its scope.

ARCHITECTURE DIAGRAM OF THE PROPOSED SYSTEM

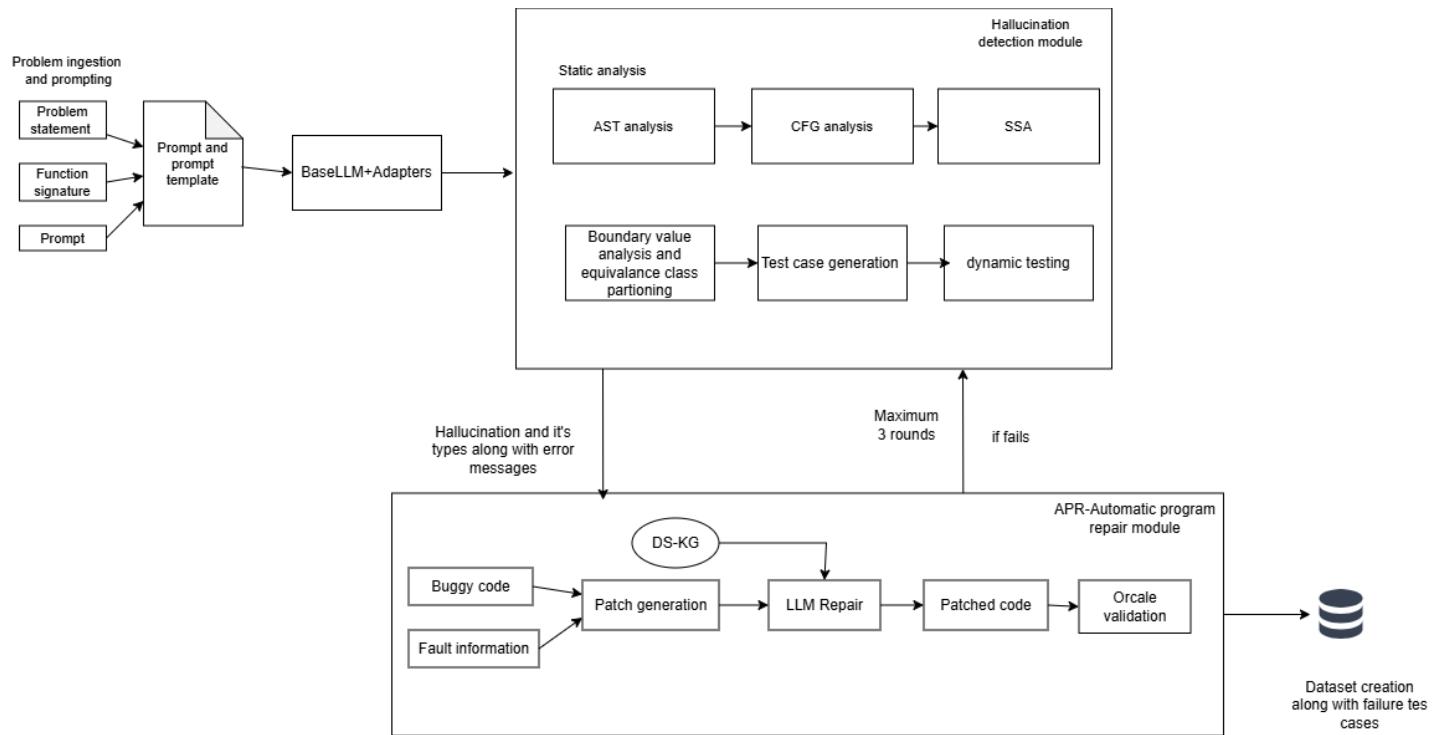


Fig.1 Client side hallucination detection and mitigation framework

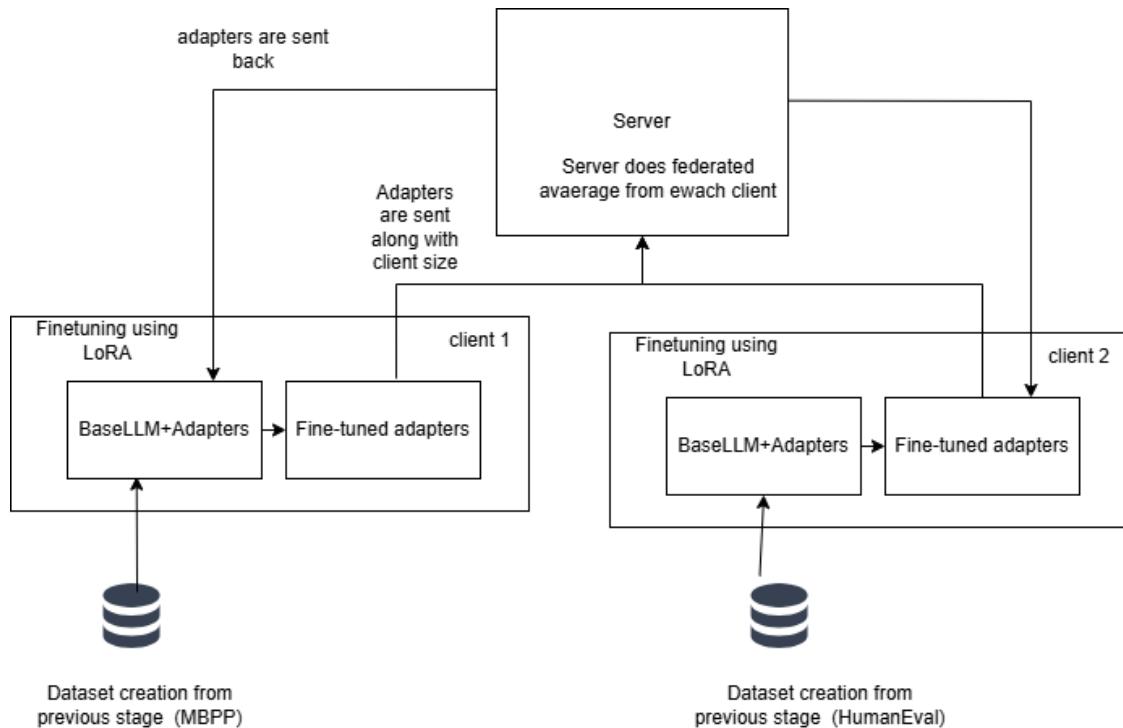


Fig.2 Federated learning + LoRA for fine-tuning

IDENTIFICATION OF PROJECT MODULES, TOOLS & PLATFORMS

S.NO	MODULES	SUB - MODULES	TOOLS & PLATFORMS
1.	Problem ingestion and code generation	<ul style="list-style-type: none"> • Dataset selection . • Client-wise dataset partitioning . • Single-shot prompting. • Pretrained LLM selection and loading. • Baseline inference using frozen LLM 	Hugging Face Datasets, Transformers, Python, Pandas, NumPy, Google Colab.
2.	Base Model Selection and Configuration	<ul style="list-style-type: none"> • Static analysis using AST ,CFG analysis and SSA. • Test case generation using BVA and ECP. • Structured hallucination labeling. 	Python `ast` module, Graphviz (CFG), Pytes.
3.	Automatic Program Repair (APR)	<ul style="list-style-type: none"> • Knowledge extraction • RDF triple representation • knowledge graph construction. • Patch-based repair • Bounded repair attempts. • Ground-truth verification and hard hallucination labeling. 	Numpy ,pandas, sklearn , scipy, statsmodel , Qwen2.5-Coder (Repair), Python.
4.	Fine tuning using LoRA	<ul style="list-style-type: none"> • Hallucination-aware dataset construction • Low-rank decomposition ($r = 8$). • Injection of LoRA modules into Query and Key projections. • Parameter isolation with frozen backbone. • Adapter checkpointing. 	PyTorch, Hugging Face PEFT, Transformers.

5.	Federated Aggregation and Distribution	<ul style="list-style-type: none"> Secure client-side adapters Federated Averaging (FedAvg) with sample-size-based weighting. Global adapter construction 	Flower ,PyTorch, NumPy
6.	Evaluation and analysis	<ul style="list-style-type: none"> Evaluation on MBPP, HumanEval, and DS-1000 benchmarks. Functional correctness using Pass@1. Hallucination detection metrics (Precision, Recall, F1). Code similarity analysis using CodeBLEU. Cross-client validation. Before–after performance comparison. 	CodeBLEU, Matplotlib, Seaborn, Scikit-learn

METHODS / MODELS

1. Problem ingestion and code generation

Input:

Benchmark code-generation datasets.

Output:

Initial LLM-generated code solutions (potentially containing hallucinations).

Methods:

- Dataset Loading:** Benchmark code-generation datasets including MBPP, HumanEval, and DS-1000 are loaded using the Hugging Face library.
- Client-wise Dataset Allocation:** Each federated client is assigned a distinct dataset or dataset subset to simulate heterogeneous behaviour.
- Problem Specification Construction:** For each task, structured inputs consisting of the problem statement, function signature, constraints is constructed.
- Single-Shot Prompting:** Code generation is performed using a single-shot prompting strategy, where the model receives the task specification without iterative refinement.
- Baseline Inference:** The frozen Qwen2.5-Coder-1.5B pretrained model is used to generate initial code outputs, which may contain hallucinations and serve as baseline for downstream analysis.

2. Hallucination detection

Input:

LLM-generated code obtained from the code generation module.

Output:

Detected hallucination types along with structured diagnostic information.

Methods:

- **AST Construction and Analysis:** Abstract Syntax Tree (AST) is constructed and analyzed to detect syntax errors, undefined variables, and invalid language constructs.
- **CFG Analysis:** Control Flow Graph (CFG) is constructed to analyze control-flow structures, including branching behavior, unreachable code paths, and missing return statements.
- **SSA Checking:** Single Static Assignment(SSA) checks use-before-definition errors and conflicting variable reassignments.
- **Test Case Generation:** Test cases are generated using Boundary Value Analysis (BVA) and Equivalence Class Partitioning (ECP) to capture edge cases
- **Dynamic Analysis:** The generated test cases are executed on the code to detect runtime errors and semantic hallucinations.
- **Hallucination Labeling:** Detected errors are categorized into different hallucination types, and the resulting information is recorded for dataset construction during fine-tuning.

3. Automatic program repair module

Input:

Buggy code, detected hallucination type, fault hints, and a domain knowledge graph.

Output:

Corrected code validated against ground-truth oracles or hard hallucination label.

Methods:

- **Knowledge Extraction:** library, class, method, and dependency information is extracted from dir,help,inspect .
- **RDF Representation:** The extracted data is represented using RDF triples.
- **KG construction :** KG constructed using our ontology design and RDF triplets.
- **Patch-Based Repair:** Applying localized patches to the hallucinated code from fault information from previous modules.

- **Fault-Guided Prompting:** The detected hallucination type, fault hints, and relevant knowledge graph entries are provided to LLM for code correction.
- **Bounded Repair Attempts:** program repair is performed with a bounded number of attempts to ensure controlled behavior.
- **GT Verification:** Each candidate is then verified with Hallucination detection module. If verification fails after bounded attempts, it is marked as hard hallucination.

4. Fine tuning using LoRA

Input:

Hallucination-aware code samples, including buggy code, diagnostic information, and repair outcomes, along with a frozen pre-trained LLM.

Output:

Client-specific LoRA adapters trained on hallucination aware datasets.

Methods:

- **Dataset Construction:** Construct structured dataset using problem description, buggy code, hallucination type, repairability flag and corrected code.
- **Low-Rank Decomposition:** Weight updates are parameterized using two low-rank matrices A and B, where the rank r is set to 8.
- **Attention Layer Injection:** LoRA modules are injected into the Query and Key projection matrices of each Transformer layer.
- **Parameter Isolation:** Original model weights remain frozen, and only LoRA parameters are made trainable.
- **Adapter Checkpointing:** Only trained LoRA adapter weights are saved and transmitted to the server, ensuring lightweight communication and privacy preservation.

5. Federated Aggregation and Distribution

Input:

Client-side LoRA adapter weights, along with sample size.

Output:

Global LoRA adapter distributed to all participating clients.

Methods:

- **Adapter Collection:** LoRA adapter weights are collected from all participating clients after local fine-tuning.
- **Federated Averaging:** Client-side adapters are aggregated using the Federated Averaging (FedAvg) algorithm, where clients with larger sample sizes contribute proportionally higher weights to the global adapter.
- **Global Adapter Construction:** A global LoRA adapter is constructed by averaging the weighted client adapters while keeping the base model parameters frozen.
- **Adapter Distribution:** The aggregated global LoRA adapter is redistributed to all clients and combined with the shared base model for subsequent inference or further training rounds.

6. Evaluation and analysis

Input:

Code generated by the base model and the federated LoRA-enhanced model across benchmark datasets.

Output:

Benchmark-wise evaluation.

Methods:

- **Functional Correctness (Pass@1):** The Pass@1 metric is used to measure functional correctness if code passes all unit tests on the first attempt.
- **Hallucination Detection Metrics:** Precision, Recall, and F1-score are computed based on oracle-validated hallucination labels to assess the accuracy of hallucination detection.
- **Code Similarity Analysis:** CodeBLEU is used as a secondary metric to measure syntactic and structural similarity between generated code and reference implementations.
- **Cross-Client Validation:** Cross-validation is performed across federated clients to evaluate generalization and robustness under heterogeneous data distributions.
- **Before–After Comparison:** Model performance is compared before and after LoRA fine-tuning to quantify improvements in correctness and hallucination reduction.

TIMELINE FOR COMPLETION OF MODULES

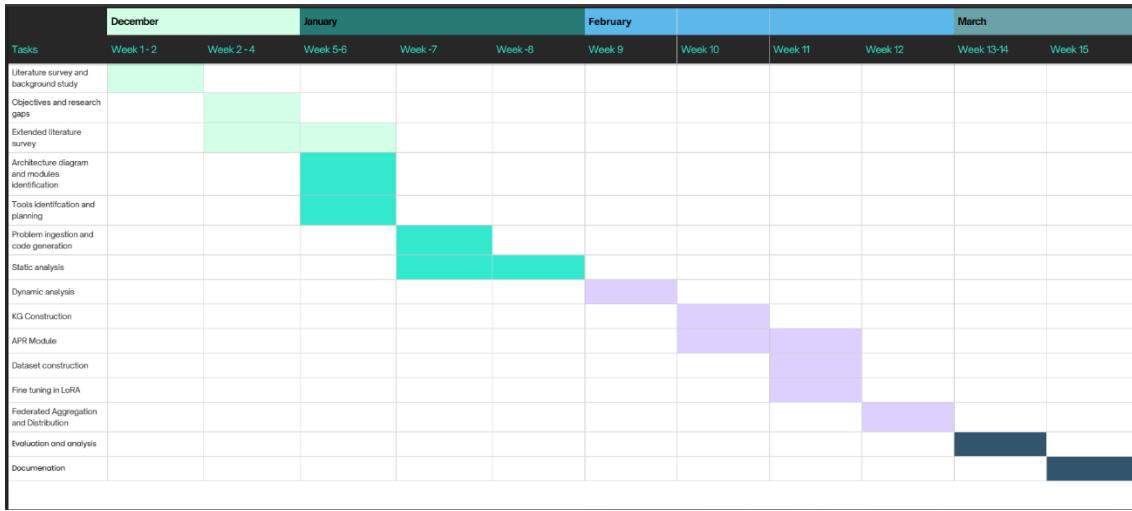


Fig.3 Federated learning + LoRA for fine-tuning

REFERENCES

- [1] B. Yang, J. Dang, H. Liu and Z. Jin, "Advancing LLM-Generated Code Reliability: A Hybrid Approach for Hallucination Detection" in *IEEE Transactions on Software Engineering*, vol. , no. 01, pp. 1-17, PrePrints 5555, doi: 10.1109/TSE.2025.3640641.
- [2] S. Fakhouri, A. Naik, G. Sakkas, S. Chakraborty and S. K. Lahiri, "LLM-Based Test-Driven Interactive Code Generation: User Study and Empirical Evaluation," in *IEEE Transactions on Software Engineering*, vol. 50, no. 9, pp. 2254-2268, Sept. 2024, doi: 10.1109/TSE.2024.3428972.
- [3] J. Chen, Z. Cai, W. Chen, W. Wang, Z. Zheng and P. S. Yu, "A Federated Adaptive Large Language Model Fine-Tuning Framework for Software Development," in *IEEE Transactions on Services Computing*, doi: 10.1109/TSC.2025.3623626
- [4] M. Nashaat and J. Miller, "Towards Efficient Fine-Tuning of Language Models With Organizational Data for Automated Software Review," in *IEEE Transactions on Software Engineering*, vol. 50, no. 9, pp. 2240-2253, Sept. 2024, doi: 10.1109/TSE.2024.3428324
- [5] R. Khojah, F. G. de Oliveira Neto, M. Mohamad and P. Leitner, "The Impact of Prompt Programming on Function-Level Code Generation," in *IEEE Transactions on Software Engineering*, vol. 51, no. 8, pp. 2381-2395, Aug. 2025, doi: 10.1109/TSE.2025.3587794

- [6] Q. Wang, C. Li, Y. Liu, Q. Zhu, J. Song and T. Shen, "An Adaptive Framework Embedded With LLM for Knowledge Graph Construction," in *IEEE Transactions on Multimedia*, vol. 27, pp. 2912-2923, 2025, doi: 10.1109/TMM.2025.3557717
- [7] X. Yu, L. Liu, X. Hu, J. W. Keung, J. Liu and X. Xia, "Fight Fire With Fire: How Much Can We Trust ChatGPT on Source Code-Related Tasks?," in *IEEE Transactions on Software Engineering*, vol. 50, no. 12, pp. 3435-3453, Dec. 2024, doi: 10.1109/TSE.2024.3492204
- [8] X. Yi, C. Hu, B. Cai, H. Huang, Y. Chen and K. Wang, "FedALoRA: Adaptive Local LoRA Aggregation for Personalized Federated Learning in LLM," in *IEEE Internet of Things Journal*, vol. 12, no. 24, pp. 51854-51865, 15 Dec. 15, 2025, doi: 10.1109/JIOT.2025.3582427
- [9] Z. Liu, Y. Tang, X. Luo, Y. Zhou and L. F. Zhang, "No Need to Lift a Finger Anymore? Assessing the Quality of Code Generation by ChatGPT," in *IEEE Transactions on Software Engineering*, vol. 50, no. 6, pp. 1548-1584, June 2024, doi: 10.1109/TSE.2024.3392499
- [10] S. Ouyang, J. M. Zhang, Z. Sun and A. M. Penuela, "Knowledge-Enhanced Program Repair for Data Science Code," in 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE), Ottawa, ON, Canada, 2025, pp. 898-910, doi: 10.1109/ICSE55347.2025.00246.
- [11] J. Liu, Y. Zhang, D. Wang, Y. Li and W. Dong, "THINK: Tackling API Hallucinations in LLMs via Injecting Knowledge," 2025 *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Montreal, QC, Canada, 2025, pp. 229-240, doi: 10.1109/SANER64311.2025.00029.
- [12] H. Zhang, Q. Yan, W. Min, C. Zhang, L. Kuang and Y. Xia, "EvoAPR: Enhancing Large Language Models for Automatic Program Repair with Genetic Algorithm and Dynamic LoRA," 2025 *IEEE International Conference on Web Services (ICWS)*, Helsinki, Finland, 2025, pp. 946-956, doi: 10.1109/ICWS67624.2025.00123.
- [13] X. Yi, C. Hu, B. Cai, H. Huang, Y. Chen and K. Wang, "FedALoRA: Adaptive Local LoRA Aggregation for Personalized Federated Learning in LLM," in *IEEE Internet of Things Journal*, vol. 12, no. 24, pp. 51854-51865, 15 Dec. 15, 2025, doi: 10.1109/JIOT.2025.3582427