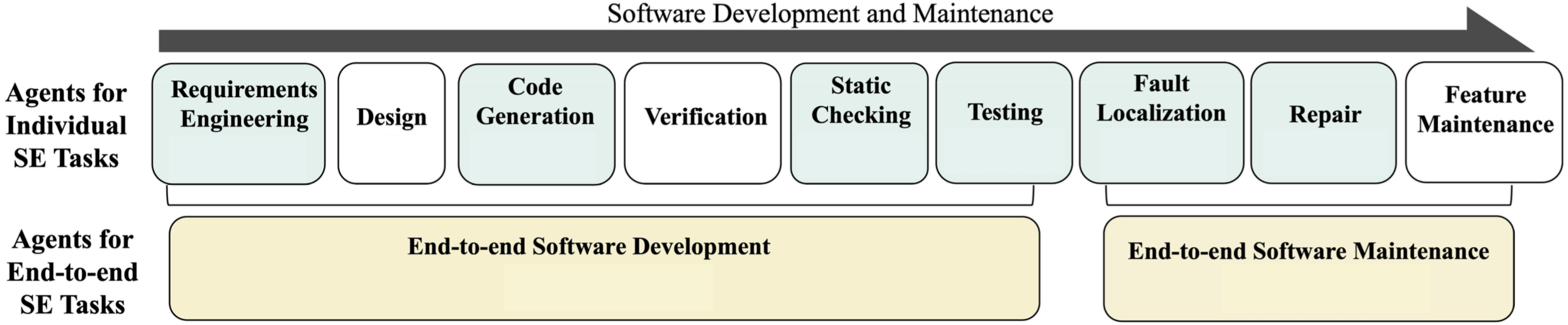
LLMs in Software Development

Contents

1. Code Generation
2. Code Refinement
3. Automated Testing
4. End-to-end Software Development
5. Copilots
6. Generated Code Evaluation
7. Further considerations: Reliability, Sustainability etc.
8. References

Code Generation



- Software development cycles involve **many repetitive and monotonous tasks** (e.g. boilerplate code, documentation, testing); manual handling of these tasks is time-consuming [1]
- LLMs can **automate** code generation, bug detection, and documentation, freeing developers for more creative and complex work => speed up development & increase productivity [1]
- Integrating LLMs into the workflow enables faster iterations and more agile responses to changing requirements [12]
- => Adoption of LLMs helps companies remain competitive in a rapidly evolving tech landscape [15]

Code Generation

- LLMs for code generation are trained on massive datasets of source code, docstrings, and natural language descriptions; they support multiple programming languages and can adapt to various coding styles and standards
- Fine-tuning may be performed on domain-specific codebases to improve accuracy and relevance for particular languages and tasks
- These models can generate code snippets, complete functions, write tests from natural language prompts and much more: e.g. OpenAI Codex, StarCoder, GitHub Copilot [13, 14, 16]
- However, utilizing raw LLMs (even code-specific) for software development is **not optimal**:
  - Standard decoding is optimized for token-likelihood, causing a disconnect between textual similarity and functional correctness in code generation [6]
  - Generated code may not be functionally correct or may contain vulnerabilities due to lack of deep understanding [4, 7, 8]
  - LLMs sometimes generate overly simplistic code: in [3], manual review of 50 generated PHP websites showed that 33 were not complex enough
  - Difficulty in capturing user intent and context can lead to irrelevant or incomplete code suggestion [1]

Code Refinement

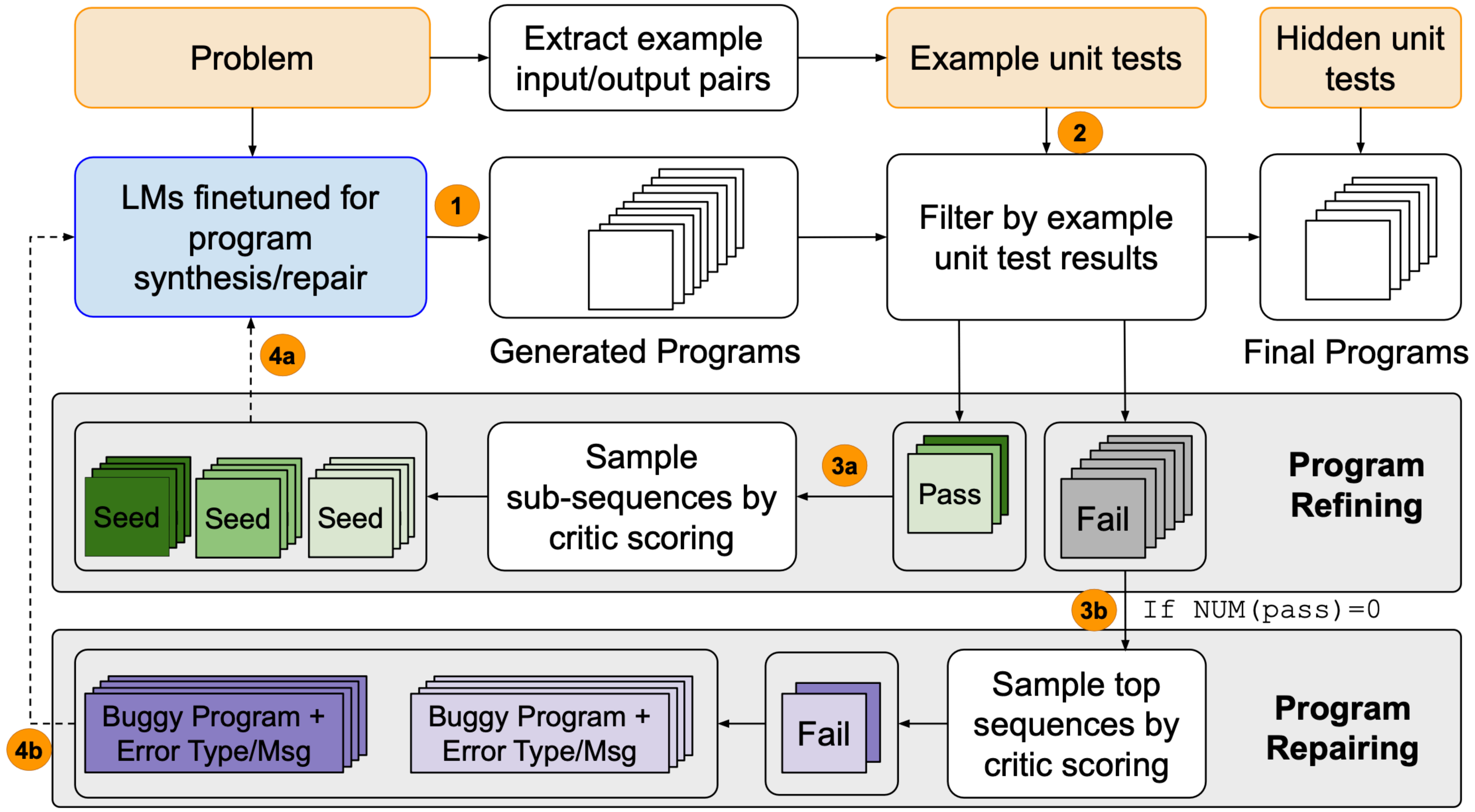
- *Iterative refinement* frameworks allow LLMs to improve code by incorporating feedback from execution or static analysis; this approach mimic real-world debugging cycles, enabling LLMs to fix errors and adapt code over multiple iterations (cf. iterative RAG)
- However, LLMs often **fail** to detect their own errors, especially in reasoning-heavy tasks, and performance can even **degrade** if refinements are inaccurate or unjustified [4]; moreover, simple iterative refinement would limit the LLM to a single COT which may be not beneficial for code generation [5]
- => Solution: coupled iterative refinement with *candidate selection*: several candidates are sampled and then evaluated to go on with the best one
- Simple variant: ask a critic model whether the suggested refinement is trustworthy and choose either the original solution or the suggestion [4]
- Still, that wouldn't bridge the gap between the highest token probabilities and true code functionality [2, 5]
- => Solution: let's incorporate signals from unit tests that correspond directly to the functional correctness [2, 4, 7]

Code Refinement

- [5] suggest to **balance** exploitation and exploration:
  - Sample several program candidates
  - Evaluate them with unit tests
  - Focus on the best candidates, but still try to refines several failed ones
  - After some number of iterations, select the most successful candidate
- Planning-Guided Transformer Decoding (PG-TD) [6] uses lookahead search and test cases to guide token selection:
  - Samples multiple token paths
  - Evaluates each with public unit tests
  - The best continuation is selected based on **both** token likelihood and performance
- Human-in-the-Loop Agent [8] integrates human feedback as its signal from the environment
  - The framework is integrated to JIRA, has access to the codebase, tickets etc.
  - It generates functioning source code e.g. from GitHub issues
  - The developers are involved after each major phase, from planing to testing

Code Refinement

- CodeRL [2] introduces a critical sampling strategy, where candidate programs are evaluated using unit tests and a critic network, and the best samples are regenerated for higher correctness
- During the inference, N programs are generated for each input
- Critic sampling:
  - The programs that are predicted to fail the unit tests are repaired with a separate LM from the last token that is predicted to pass the tests
  - The ones that are predicted to pass the tests are nevertheless refined: they are sub-sampled, and M best sub-samples are used as the seed to precondition even better variants of these programs



Automated Testing

- Automated testing is yet another challenging task due to the limited code coverage and compilation/runtime errors from hallucinated tests [10, 11]
- LLMs are increasingly used to automate unit test generation, reducing manual effort and improving coverage
- ChatUniTest [9] includes only the most relevant code context and utilizes a generation-validation-repair loop
- TestART [10] utilizes the ART [4] principle to test generation and achieves an 18% improvement in pass rate and 20% higher coverage compared to baselines, with fewer test cases needed than EvoSuite
- Meta deployed an LLM-based system for improvement of the existing tests and reported [11] the following:
  - 75% of generated tests built successfully.
  - 25% of generated tests led to increased code coverage
  - 73% of test suggestions were accepted by engineers **into production**

End-to-end Software Development

- LLM agents are now being designed for end-to-end software engineering, simulating entire development teams with roles like project managers, requirement analysts, designers, developers, and QA experts [1]
- Systems such as CodeS decompose code generation not onto roles but onto tasks: different agents manage repository, file, and method layers [1]
- Maintenance is addressed by agents that can analyze, update, and refactor codebases, supporting long-term project evolution and bug fixing [1]
- Specialized LLM agents can automate environment setup and deployment, such as generating Docker environments and managing containerized workflows [17]
- Such multi-agent frameworks enable not just code writing but also requirements analysis, design, testing, and documentation—mirroring real-world team workflows [1]
- => Conclusion: modern LaMAs are capable enough to be able to support (if not overtake) the most processes of software engineering routine in real production. Or is it?

Copilots

- „Tab, Tab, Tab…“: modern developers cannot now imagine their work without *copilots* — AI-based assistants that integrate with development environments to provide real-time code suggestions, autocompletion, and contextual help throughout the software development process
- **Codex**: a cloud-based software engineering agent powered by the codex-1 model from OpenAI; it is capable of not only writing code, but also answering questions about it, fixing bugs etc., supports **parallel** task handling
- **GitHub Copilot**, powered by the latest OpenAI Codex models, can generate code, refactor functions, write tests, and assist with documentation directly in popular IDEs [14]
- Research [15] shows that GitHub Copilot increases developer productivity—tasks are completed up to 55% faster, and 60–75% of users report **higher satisfaction** and reduced frustration
- Cursor [16] goes even further: it is an AI-powered code editor that seamlessly integrates large language models directly into the development environment; it is highly **aware** about everything in your project and can answer questions about your code, suggest context-aware changes, and reference files or docs for more accurate assistance

Generated Code Evaluation

- HumanEval [18, 19] is a standard benchmark with Python programming tasks evaluated via functional correctness—generated code must pass hidden unit tests. Key metrics are *Pass@k,* and code correctness
- APPS (Automated Programming Progress Standard) is a large-scale benchmark with problems ranging from beginner to competition level. It uses execution-based metrics such as Pass@k and test case accuracy, challenging models with both short and long-form code generation
- LLM as Evaluator: recent works propose using LLMs themselves to assess code quality. For example, the CodeJudge framework [20] prompts LLMs to perform detailed "slow thinking" evaluations, scoring code on semantic correctness, style, and robustness, **even in the absence** of test cases. This approach helps better evaluate not only the correctness, but whether the code actually aligns with the initial human intent

Further considerations: Reliability, Sustainability etc.

- Even though LLM-based systems are now intelligent and performant, it turns out that they might still be not good enough to blindly transfer all the routine on them
- It turns out, existing LLMs frequently generate code containing vulnerabilities, often neglecting security best practices [22]:
  - [21] shows that out of 1200+ coding questions that cover 24 Java API, even for advanced models like GPT-4, 62% of generated code samples contain API misuses
  - [3] finds that GPT-4 generated PHP-website with vulnerabilities in 11.16% of the whole dataset (2500 entries), and from the sites with file upload functionality, 78% were vulnerable according to static analysis
- => We should use such systems in tandem with professionals who will notice vulnerabilities or misuse or order to fix them
- Moreover, LLMs are not as green as manual coding (yet) [23]

References

[1] Large Language Model-Based Agents for Software Engineering: A Survey, Fudan University, Nanyang Technological University & University of Illinois at Urbana-Champaign

[2] CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning, Salesforce Research

[3] LLMs in Web Development: Evaluating LLM-Generated PHP Code Unveiling Vulnerabilities and Limitations, University of Oslo

[4] The ART of LLM Refinement: Ask, Refine, and Trust, ETH Zurich & Meta AI

[5] Code Repair with LLMs gives an Exploration-Exploitation Tradeoff, Cornell, Shanghai Jiao Tong University & University of Toronto

[6] Planning with Large Language Models for Code Generation, MIT-IBM Watson AI Lab et al.

[7] A Real-World WebAgent with Planning, Long Context Understanding, and Program Synthesis, Google DeepMind & The University of Tokyo

[8] Human-In-the-Loop Software Development Agents, Monash University, The University of Melbourne & Atlassian

[9] ChatUniTest: A Framework for LLM-Based Test Generation, Zhejiang University & Hangzhou City University

[10] TestART: Improving LLM-based Unit Testing via Co-evolution of Automated Generation and Repair Iteration, Nanjing University & Huawei Cloud Computing Technologies

[11] Automated Unit Test Improvement using Large Language Models at Meta, Meta

[12] Design and evaluation of AI copilots -- case studies of retail copilot templates, Microsoft

[13] Introducing Codex, OpenAI (blog post)

[14] GitHub Copilot, GitHub (product page)

[15] Research: quantifying GitHub Copilot’s impact on developer productivity and happiness, GitHub (blog post)

[16] Cursor: The AI Code Editor, Cursor (product page)

[17] An LLM-based Agent for Reliable Docker Environment Configuration, Harbin Institute of Technology & ByteDance

[18] Evaluating Large Language Models Trained on Code, OpenAI

[19] Code Generation on HumanEval, OpenAI (leaderboard)

[20] CodeJudge: Evaluating Code Generation with Large Language Models, Huazhong University of Science and Technology & Purdue University

[21] Can ChatGPT replace StackOverflow? A Study on Robustness and Reliability of Large Language Model Code Generation, UC San Diego

[22] Enhancing Large Language Models for Secure Code Generation: A Dataset-driven Study on Vulnerability Mitigation, South China University of Technology & University of Innsbruck

[23] Learn to Code Sustainably: An Empirical Study on LLM-based Green Code Generation, TWT GmbH Science & Innovation et al.