My main research area is the fusion of software engineering (SE) and natural language processing (NLP), with a focus on improving developers' productivity during software development, testing, and maintenance. My research focuses on *execution-guided learning for evolving software*. Specifically, it covers developing learning-based software engineering techniques that leverage software *execution* [1, 2, 3, 4] and software *evolution* [5, 6, 7, 8, 9], as well as enhancing the utilization of *specifications* (comments, code contracts, and tests) [10, 11, 12, 13, 14, 15, 16, 17].

I have published 12 refereed conference papers at top-tier conferences in software engineering (FSE×2, ASE×2, ICSE, ISSTA, ICSEDemo), natural language processing (ACL×2), and programming languages (OOPSLA×2, IJCAR). I have won an ACM SIGSOFT Distinguished Paper Award (FSE'19) for my work on writing trigger-action comments in executable format [11]. My research has resulted in novel techniques and many open-source tools (ten GitHub repositories) for code and test completion, comment generation and maintenance, lemma naming, executable comments, executable code contracts, and inline testing. My research has been motivated by real-world SE challenges, which I observed by interacting with open-source and industry developers. During my internships at Facebook, I combined my research with industrial SE needs and contributed to several learning-based SE techniques used internally at the company.

1 Learning-Based Software Engineering

My research is among the first to develop learning-based software engineering techniques that leverage *code execution* and are *evolution-aware*. The success of ML (machine learning) in NLP motivated the applications ML on SE tasks to improve developers' productivity. However, existing ML solutions have limited effectiveness on SE tasks because of treating software as static natural language text that does not execute or evolve. I identified two important aspects of software and software development that must be taken into account to develop high-quality ML models. First, unlike text, code can be *executed*, and ML models should integrate code execution into their design and workflow. Second, software constantly evolves, and ML models should focus on *editing* and not just on generation.

1.1 Learning with Code Execution

Developers heavily rely on execution to understand and write code during software development and testing. However, existing ML models for SE rely on syntax-level representations of code. I was the first to combine ML models with code execution to improve their performance tasks that require deeper reasoning about code by leveraging runtime data, integrating execution into the workflow, and improving the model architecture. My research in this direction targets applications in domains where code execution is important, including testing, verification, and hardware design.

Test completion. Software testing is the most frequently-used technique in industry for checking software correctness. A large part of testing requires manually writing tests which is time-consuming. I developed TeCo [1] to aid developers in writing tests by completing the next statement given the prior test code statements and the code under test. TeCo exploits code semantics extracted from *test execution results* (e.g., local variable types) and *execution context* (e.g., last executed method). Given the candidate next statements predicted by the ML model, TeCo reranks them to prioritize the compilable and runnable candidates (which are likely functionally correct). Compared to the best existing ML model, TeCo improves the accuracy of generating the same developer-written next statement from 14% to 18%, and improves the chance of generating a runnable next statement (which simplifies debugging the output) from 19% to 29%.

Suggesting lemma names. Proof assistants are becoming popular for developing trustworthy software systems (e.g., compilers and file systems). Code written in proof languages (e.g., Coq) is difficult to understand, and thus using comprehensible names for lemmas is important. I developed Roosterize [2] for suggesting lemma names in Coq. Roosterize *executes* the lemma to extract its runtime representations from two phases of execution: syntax trees from the parsing phase and kernel trees from the elaboration phase. They are used jointly with the lemma as inputs to the ML model. Runtime representations contain more complete information (e.g., elaborated user-defined notations and implicit terms) for generating accurate names. Roosterize is deployed as a Visual Studio Code plugin [3]. A qualitative study conducted with a verification project maintainer found that 25% of the names suggested by Roosterize were of good quality.

Statement completion for hardware descriptions. Hardware description languages (HDLs) are used to describe the design of digital circuits in hardware. HDLs have unique syntax and semantics compared to popular imperative languages (e.g., Java). For example, to handle the parallelism inherent in hardware designs, the assignment statements are executed concurrently. To leverage this unique feature, I developed a statement completion ML model for VHDL (one of the most popular HDLs) where multiple prior assignment statements are concurrently used as inputs [4]. Our ML model outperforms the baseline code completion ML models, improving the accuracy from 14% to 19%.

1.2 Learning to Edit Software

Software constantly evolves to implement new features, fix bugs, improve documentation, etc. Prior work considered applying ML models on a single version of the software and entirely ignored valuable history readily available in repositories. The context from history is important for SE techniques as developers perform updates rather than write things from scratch. I was the first to study learning to *edit* software, initially on the task of updating code comments when code changes. Later, I generalized our ML model to a pretrained one that can support more software editing tasks and proposed a novel methodology for evaluating ML models.

Comment updating. Developers use API comments that accompany each method to document the method's intended behaviors and usages. Failure to co-evolve code and the associated API comments lead to confusion and introduce software bugs. To automate the co-evolution of code and comments, I developed an ML edit model for automatically updating a comment when its associated code changes [5]. The edit model generates an *edit sequence* that can be applied on the old comment to produce the new comment. Compared to existing ML models that generate from scratch, our approach produces more accurate comments by keeping the style and relevant information unchanged. Users accepted the outputs of our edit model (30%) more frequently than a generation model (12%).

Evolution-aware pretraining. To generalize to other software editing tasks beyond comment updating, I used the pretraining approach: train an ML model that leans to edit on a large dataset in an unsupervised manner, where the learned knowledge is transferrable to many downstream tasks. I developed the first pretrained ML model for code and natural language editing, CoditT5 [6]. To pretrain the ML model, I proposed a novel pretraining objective of generating edit sequences to recover random mutations mimicking developer edits. CoditT5 sets new state-of-the-art performance on several software editing tasks, including comment updating, bug fixing, and automated code review. For comment updating, CoditT5 improves the accuracy over our previous non-pretrained edit ML model from 33% to 45%. For bug fixing and automated code review, where strong pretrained generation ML models existed, CoditT5 also has 5–8% accuracy improvements.

Evolution-aware methodology. Data and metrics used to evaluate ML models for code summarization tasks ignored the order in which data was created in practice. To split the dataset into training and testing sets, prior work adopted either the mixed-project methodology (mixes data from all projects and then splits) or the cross-project methodology (splits data at the project level). A natural continuous-mode use case of ML models requires that data in the training set be available *before* data in the testing set, but the two existing methodologies are *not evolution-aware* and do not match this use case. I proposed the time-segmented evaluation methodology [7] which splits data based on timestamps and is *evolution-aware*. Our experiments found that prior approaches either underestimate or overestimate the prediction accuracy of ML models, so the time-segmented methodology should be used to evaluate ML models for evolving software.

2 Enhancing the Utilization of Specifications

Comments, code contracts, and tests form a large portion of software. I broadly define these code and natural language elements as *specifications* because they are used to describe software behaviors and development needs. These specifications help communicate development tasks and check software correctness, but their utilization has been limited due to the lack of formal connections between specifications and production code. To reduce the cost of writing and maintaining specifications and improve their benefits in software development, I develop novel techniques and domain specific languages to make specifications executable and easier to maintain.

Executable todo comments. Developers use todo comments to communicate future tasks and development needs. Todo comments in open-source projects frequently become dangling (i.e., forgotten or irrelevant) [10]. I developed TrigIt [11] for writing and automatically maintaining executable todo comments. TrigIt provides a domain specific language embedded in the host language to specify tasks that need to be performed (actions) when specific conditions hold on code-related artifacts (triggers). When a trigger evaluates to true, TrigIt executes the actions and removes the comment. I migrated dozens of natural language todo comments in ten open-source projects to executable todo comments, and reported six dangling todo comments found by using TrigIt to execute those comments.

Unifying imperative code and declarative code contracts. Code contracts, including invariants, pre-conditions, and post-conditions, are used to formally describe software behavior. They are traditionally checked during executing production code and recently can be also executed alone for mocking and prototyping. Code contracts are usually written in declarative languages different from the mainstream imperative language, which can be hard to learn and use. I developed Deuterium [12] for writing and executing code contracts entirely in Java as a combination of imperative and declarative code. Deuterium helps write type-safe code contracts and speeds up the execution of code contracts by utilizing their imperative parts.

Inline testing. Existing testing frameworks target unit-level (i.e., a single method), which is ill-suited for testing individual statements. Previous studies showed that single-statement bugs are common and often missed by unit tests. I proposed a novel granularity of testing, inline testing [13, 14], for checking the correctness of individual statements. Inline testing supplements existing unit testing to provide more confidence that software is correct. The inline tests we wrote for 100 statements in open-source projects helped find two bugs not covered by unit tests. Because inline tests are written next to the statement under test, they are easier to co-evolve with code.

3 Future Directions

In the short term, I will focus on improving learning-based SE techniques with software execution and evolution. I will apply the improved techniques to more tasks, such as test generation based on bug reports, and more domains, such as to support the development of ML code. In the long term, I will design the future ML-assisted software development workflow with the help of specifications.

Learning from evolving code execution data. I will combine learning from execution and learning to evolve with the goal of developing high-quality ML models that understand code semantics. One possible combination is to consider the diff of execution data when software evolves, which may be more informative than the diff of code itself. Another possible combination is to consider the evolution of the program state when code executes, because execution data (e.g., runtime values) can be too large to be used by ML models, but the delta execution data (e.g., runtime values changed during execution) is more succinct. I will leverage these new sources of data in large-scale pretraining, which has promising potential in building high-quality ML models for code (e.g., Codex, AlphaCode). Obtaining large-scale software execution and evolution dataset for pretraining is challenging, which I will tackle by augmenting real-world software data via mutation and generating synthetic data based on programming languages' grammars.

Learning to generate bug-revealing tests. Having bug-revealing tests that expose software bugs is a prerequisite for fixing bugs (either manually or using automated program repair techniques), but writing such tests can be challenging especially under the pressure of fixing bugs in a short time (e.g., bugs in deployed software). I propose to develop a learning-based technique to help generate bug-revealing tests given the bug report and code under test. The technique will extract test inputs and oracles from the bug report and leverage test execution to refine the inputs and oracles. For software with existing tests, the technique will learn to reuse an existing test with edited inputs and oracles instead of always generating from scratch.

Learning-based SE for ML. The amount of code for ML frameworks and algorithms is rapidly growing, but SE support for ML is lacking. For instance, many ML projects are not well-documented or tested. Despite being written in popular programming languages (e.g., Python), ML code uses specialized "dialects" for unique features like tensor operations and optimized training. Thus, existing (learning-based) SE techniques are ill-suited for ML code. I propose to develop learning-based SE techniques adapted for ML code, to (1) generate and maintain comments for ML code, leveraging the external knowledge of the research papers accompanying the code; (2) generate tests for ML code to check correctness, robustness, and runtime performance, leveraging the execution (training and inference) of ML models.

ML-Assisted Software Development Workflow. Current software development relies on developers to design, implement, and maintain code, but a new software development workflow is needed in the next few years when learning-based techniques become accurate enough to automate some parts of the workflow, e.g., implementing code. I propose that specifications, including comments, code contracts, and tests, should be the interface between developers and ML models. For example, in the initial stages of software development, ML models can generate code with specifications given the high-level requirements provided by developers. Then, developers can modify the specifications, and ML models should update the code accordingly. Towards this direction, I will iteratively design better formats of specifications, develop learning-based techniques for the specifications, and deploy the techniques in real-world software development workflow to collect feedback. The design of specifications will consider the concerns from both developers' perspectives (e.g., being executable and concise) and ML models' perspectives (e.g., correlation with code).

References

[1] Pengyu Nie, Rahul Banerjee, Junyi Jessy Li, Raymond J. Mooney, and Milos Gligoric. Learning deep semantics for test completion. In *ICSE*, 2023. To appear.

[2] Pengyu Nie, Karl Palmskog, Junyi Jessy Li, and Milos Gligoric. Deep generation of Coq lemma names using elaborated terms. In *IJCAR*, pages 97–118, 2020.

[3] Pengyu Nie, Karl Palmskog, Junyi Jessy Li, and Milos Gligoric. Roosterize: Suggesting lemma names for Coq verification projects using deep learning. In *ICSEDemo*, pages 21–24, 2021.

[4] Jaeseong Lee, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. On the naturalness of hardware descriptions. In *FSE*, pages 530–542, 2020.

[5] Sheena Panthaplackel, Pengyu Nie, Milos Gligoric, Junyi Jessy Li, and Raymond J. Mooney. Learning to update natural language comments based on code changes. In *ACL*, pages 1853–1868, 2020.

[6] Jiyang Zhang, Sheena Panthaplackel, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. CoditT5: Pretraining for source code and natural language editing. In *ASE*, pages 1–12, 2022.

[7] Pengyu Nie, Jiyang Zhang, Junyi Jessy Li, Raymond J. Mooney, and Milos Gligoric. Impact of evaluation methodologies on code summarization. In *ACL*, pages 4936–4960, 2022.

[8] Pengyu Nie, Karl Palmskog, Junyi Jessy Li, and Milos Gligoric. Learning to format Coq code using language models. In *The Coq Workshop*, 2020.

[9] Jiyang Zhang, Sheena Panthaplackel, Pengyu Nie, Junyi Jessy Li, Raymond J. Mooney, and Milos Gligoric. Leveraging class hierarchy for code comprehension. In *CAP Workshop*, 2020.

[10] Pengyu Nie, Junyi Jessy Li, Sarfraz Khurshid, Raymond J. Mooney, and Milos Gligoric. Natural language processing and program analysis for supporting todo comments as software evolves. In *NL4SE Workshop*, pages 775–778, 2018.

[11] Pengyu Nie, Rishabh Rai, Junyi Jessy Li, Sarfraz Khurshid, Raymond J. Mooney, and Milos Gligoric. A framework for writing trigger-action todo comments in executable format. In *FSE*, pages 385–396, 2019.

[12] Pengyu Nie, Marinela Parovic, Zhiqiang Zang, Sarfraz Khurshid, Aleksandar Milicevic, and Milos Gligoric. Unifying execution of imperative generators and declarative specifications. In *OOPSLA*, pages 217:1–217:26, 2020.

[13] Yu Liu, Pengyu Nie, Owolabi Legunsen, and Milos Gligoric. Inline tests. In *ASE*, pages 1–13, 2022.

[14] Yu Liu, Zachary Thurston, Alan Han, Pengyu Nie, Milos Gligoric, and Owolabi Legunsen. pytest-inline: An inline testing tool for Python. In *ICSEDemo*, 2023.

[15] Ahmet Celik, Pengyu Nie, Christopher J. Rossbach, and Milos Gligoric. Design, implementation, and application of GPU-based Java bytecode interpreters. In *OOPSLA*, pages 177:1–177:28, 2019.

[16] Pengyu Nie, Ahmet Celik, Matthew Coley, Aleksandar Milicevic, Jonathan Bell, and Milos Gligoric. Debugging the performance of Maven's test isolation: Experience report. In *ISSTA*, pages 249–259, 2020.

[17] Yu Liu, Jiyang Zhang, Pengyu Nie, Milos Gligoric, and Owolabi Legunsen. Leveraging code-change information for faster regression test selection in projects with limited histories. In *ISSTA*, 2023.