A Tool for Generating Exceptional Behavior Tests With Large Language Models

Linghan Zhong
UT Austin (USA)
linghanz@cs.utexas.edu

Samuel Yuan
UT Austin (USA)
syuan@utexas.edu

Jiyang Zhang
UT Austin (USA)
jiyang.zhang@utexas.edu

Yu Liu
UT Austin (USA)
yuki.liu@utexas.edu

Pengyu Nie
UWaterloo (Canada)
pynie@uwaterloo.ca

Junyi Jessy Li
UT Austin (USA)
jessy@austin.utexas.edu

Milos Gligoric
UT Austin (USA)
gligoric@utexas.edu

Abstract

Exceptional behavior tests (EBTs) are crucial in software development for verifying that code correctly handles unwanted events and throws appropriate exceptions. However, prior research has shown that developers often prioritize testing "happy paths", i.e., paths without unwanted events, over exceptional scenarios. We present exLong, a tool that automatically generates EBTs to address this gap. exLong leverages a large language model (LLM) fine-tuned from CodeLlama and incorporates reasoning about exception-throwing traces, conditional expressions that guard throw statements, and non-exceptional behavior tests that execute similar traces. Our demonstration video illustrates how exLong can effectively assist developers in creating comprehensive EBTs for their project (available at https://youtu.be/Jro8kMgplZk).

CCS Concepts

• **Software and its engineering → Software testing and debugging**.

Keywords

Automatic test generation, exceptional behavior tests

ACM Reference Format:
Linghan Zhong, Samuel Yuan, Jiyang Zhang, Yu Liu, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. 2025. A Tool for Generating Exceptional Behavior Tests With Large Language Models. In *33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion '25), June 23–28, 2025, Trondheim, Norway.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3696630.3728608

```java
class Scheduler {
  ...
  public Job schedule(String nullableName,Runnable runnable,Schedule when){
    ...
    Job job = prepareJob(name, runnable, when);
    ...
    return job;}
  private Job prepareJob(String name, Runnable runnable, Schedule when)
    synchronized (indexedJobsByName) {
      Job lastJob = findJob(name).orElse(null);
      if(lastJob != null && lastJob.status() != JobStatus.DONE) {
        throw new IllegalArgumentException("A job is already scheduled with
               the name:" + name); }
      ...
      return job;}}}
```

(a) Method under test: `schedule`**.**

```java
@Test(expected = IllegalArgumentException.class)
public void reject_scheduling_a_job_with_same_name_but_different_runnable()
        {
    Scheduler scheduler = new Scheduler();
    Job j1 = scheduler.schedule("myJob", runnable1, now().plusSeconds(5));
    scheduler.schedule("myJob", runnable2, now().plusSeconds(6));}
```

(b) EBT generated by exLong.

Figure 1: Developer-oriented use case example.

1 Introduction

Many popular programming languages, including C#, Java, and Python, support exceptions [15, 17, 37]. Exceptions are thrown during program execution if an unwanted event happens, e.g., a method is invoked with an illegal argument value. Software developers write *exceptional behavior tests* (EBTs) to check that their code properly detects unwanted events and throws desired exceptions. Prior research studies on EBTs [2, 8, 14, 21, 24] have shown the importance of EBTs and developers' desire to improve the testing of exceptional behaviors. However, in practice, developers tend to focus on "happy paths" and have limited time to test exceptional behaviors. This results in a lower number of EBTs compared to non-EBTs in most projects.

Sadly, tool support for automatically generating EBTs is limited. Most existing analysis-based test generation tools (e.g., Randoop [28, 31] and EvoSuite [12]) and learning-based test generation tools (e.g., CAT-LM [30] and TeCo [26]) have no special settings for targeting EBTs and are primarily evaluated on non-EBTs. Random test generation tools can be guided by reinforcement learning to target exceptional behaviors [1], but the generation works only on the entire codebase, and not for a specific throw statement that a developer might select. Additionally, tests produced by analysis-based tools often lack readability [6, 7, 29].

We recently designed and developed exLong [44], a framework that utilized an instruction fine-tuned large language model (LLM) to automatically generate EBTs. Using CodeLlama [32] as its base, exLong is fine-tuned [34, 39, 40] with a novel task instruction dataset, designed specifically to embed the reasoning about the context which includes: (a) stack traces that lead to target throw statements, (b) guard expressions (i.e., conditional expressions that guard those throw statements), and (c) non-EBTs that execute similar traces. This context is used as the input to generate an EBT that

FSE Companion '25, Trondheim, Norway

ACM ISBN 979-8-4007-1276-0/2025/06
https://doi.org/10.1145/3696630.3728608

```
1 public Scheduler(SchedulerConfig config) {
2     if(config.getTimeProvider() == null) {
3         throw new NullPointerException("The timeProvider cannot be null"); }
4     ...
5 }
```

(a) Method under test: `Scheduler`**.**

```
@Test(expected = NullPointerException.class)
public void should_fail_if_timeProvider_is_null() {
    new Scheduler(SchedulerConfig.builder().maxThreads(1).timeProvider(null)
            .build());}
```

(b) EBT generated by exLong.

Figure 2: Machine-oriented use case example.

triggers the target throw statement. In figures 1 and 2, we show examples of EBTs generated by exLong.

This paper extends exLong by introducing a new command-line interface that simplifies the process of extracting the necessary context for EBTs generation and querying the fine-tuned LLM. We describe two use cases supported by exLong: (1) developer-oriented use case: developers select a method under test (e.g., `schedule` in Figure 1a), a target throw statement (e.g., line 12 in Figure 1a) and a destination test file. exLong then automatically generates an EBT that executes the target throw statement. (2) machine-oriented use case: developers employ exLong to automatically generate EBTs for their entire codebase, covering each existing throw statement, such as line 3 in `Scheduler` in Figure 2a. Additionally, to improve exLong's accessibility for typical users, we include an option to use a quantized [9, 42] version of the fine-tuned LLM, which reduces the memory usage by 75%. This optimization enables exLong to operate on machines with limited computational resources.

Our experiments demonstrate exLong's effectiveness in both supported use cases. For the developer-oriented use case, we compare our tool against a state-of-the-art test generation model (CAT-LM [30]) and a leading foundation LLM (GPT3.5 [27]). Results show that exLong generates 83.8% more executable EBTs than CAT-LM and 9.9% more than GPT3.5. After quantization, exLong can run on a local machine with a single GPU, with a relative small performance reduction resulting in the generation of 13.1% fewer executable EBTs. For the machine-oriented use case, we compare our tool against two popular analysis-based test generation tools: Randoop [28, 31] and EvoSuite [12]. While these tools complement each other (i.e., each tool can generate EBTs for some target throw statements that others cannot), our findings indicate that exLong outperforms both Randoop and EvoSuite. exLong is available on GitHub at https://github.com/EngineeringSoftware/exLong.

2 Technique and Implementation

Figure 3 [44] illustrates the workflow of exLong. Given a method under test (MUT), a target throw statement, and a destination test file, exLong collects stack trace, guard expression, and relevant non-EBTs using both static and dynamic program analyses (3). These components are then used to construct a prompt which encompasses both the task inputs and the relevant context (4). During training, a foundation LLM is fine-tuned to generate the EBT conditioned on the input p. During inference, exLong first prepares the necessary context to construct the prompt then the fine-tuned LLM generates EBTs given the prompt. We detail the design and implementation in the rest of this section.

2.1 Developer-oriented use case

Preparation. In this phase, exLong collects a set of stack traces from the execution of existing non-EBTs, that can reach methods that contain target throw statements in the repository.Using the example in Figure 1, exLong first identifies and instruments the throw statement in the method `prepareJob` to log the current stack trace upon the invocation of `prepareJob`. Then exLong executes the existing non-EBTs to log the stack traces and record the mapping between the non-EBTs and their invoked methods. Note that a developer only need to run this phase once for the repository they are working on.

Analysis. exLong constructs a prompt from the developer-provided context and the information collected in the preparation phase. Taking Figure 1 as an example, exLong first searches the collected stack traces for one that begins with `schedule` and ends in `prepareJob`. An example of the resulting stack trace consisting of the `schedule` and `prepareJob` methods is shown in Figure 4a. While stack trace provides the sequence of method invocations that lead to the target throw statement, knowing only the names of the methods is insufficient for generating EBTs. exLong then constructs a guard expression to further aid the LLM's reasoning about system configurations that would lead to exceptional behaviors. A guard expression is a logical formula representing the constraints necessary to reach the target throw statement. An example of guard expression is shown in Figure 4b. Specifically, exLong collects guard-related AST nodes along the stack trace, including conditional expressions (line 11 in Figure 1) and assignments (line 10 in Figure 1). It then propagates symbolic variables, performing substitutions where necessary. The resulting formula is a conjunction of expressions guarding the target throw statement. Finally, exLong identifies relevant non-EBTs from the same repository to encourage the LLM to reason about the procedures to set up the object under test and to promote consistency between the newly generated code and existing code in terms of format and coding conventions. The non-EBT in figure 4c is identified as relevant since it invokes the target MUT `schedule`. To enhance the quality of the generated EBTs, exLong can optionally create multiple prompts by including different relevant non-EBTs and then select the best EBT based on its ability to compile, execute, and cover the target throw statement.

2.2 Machine-oriented use case

Preparation. exLong parses the repository to identify all target throw statements within public methods (line 3 in Figure 2). Similar to developer-oriented use case, it executes the existing non-EBTs to extract the coverage data. This is used to determine both the relevant non-EBTs and the destination test file.

Analysis. As shown in Figure 2a, for each target throw statement, the MUT is defined as the method containing the target throw statement (`Scheduler`). In this case, the stack trace only includes the MUT. The guard expression and relevant non-EBTs are extracted using the same approach as developer-oriented use case. The destination test file is selected using two heuristics similar to prior works [30]: (1) file name matching where given a code file named `Scheduler.java`, exLong searches for test file named `TestScheduler.java` or `SchedulerTest.java`, and (2) test coverage analysis in which if name matching fails, exLong searches for

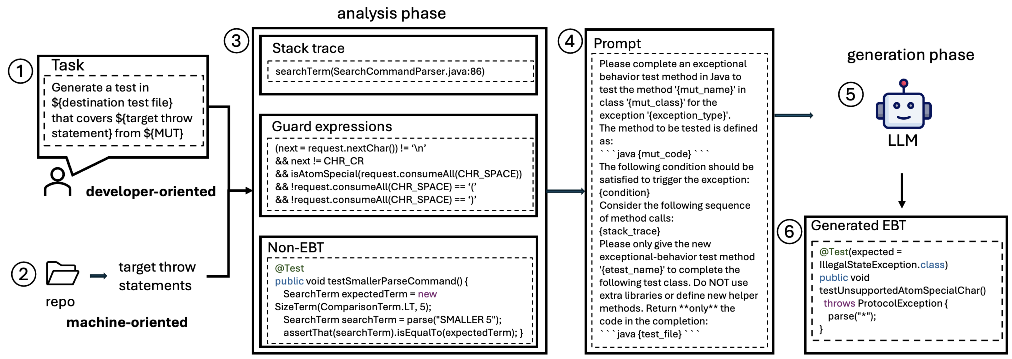

Figure 3: Overview of exLong.

```
schedule(Scheduler.java:186)
    ...
    Job job = prepareJob(name, runnable, when);
    ...
prepareJob(Scheduler.java:340)
    ...
    throw new IllegalArgumentException("A job is already scheduled with the
          name:" + name);
    ...
```

(a) Stack trace from MUT to target throw statement.

```
findJob(nullableName == null ? runnable.toString() : nullableName).orElse(
       null) != null && findJob(nullableName == null ? runnable.toString() :
        nullableName).orElse(null).status() != JobStatus.DONE
```

(b) Guard expression.

```
@Test
public void should_run_a_single_job() throws InterruptedException {
    Scheduler scheduler = new Scheduler();
    SingleJob singleJob = new SingleJob();
    scheduler.schedule("test", singleJob, Schedules.executeOnce(Schedules.
          fixedDelaySchedule(Duration.ofMillis(1))));
    waitOn(singleJob, () -> singleJob.countExecuted.get() > 0, 10000);
    scheduler.gracefullyShutdown();
    assertThat(singleJob.countExecuted.get()).isEqualTo(1);}
```

(c) non-EBT.

Figure 4: Context for exLong.

the test class covering the MUT or the class of the MUT. Finally, exLong constructs the prompt with all the available context. exLong can optionally create multiple prompts from different non-EBTs, generating and evaluating multiple EBTs then select the best one based on runtime evaluation.

3 Tool Installation

exLong generates EBTs for Java projects built using Maven. We require Maven 3.8.3+ and Java 8+. For quantized LLM inference, exLong leverages ollama [41], which can be installed following the instructions from ollama's official GitHub repository.

To get started with exLong, begin by cloning the repository:

```
$ git clone https://github.com/EngineeringSoftware/exLong.git
```

exLong is implemented in Python and requires version 3.10 or higher. For a smooth installation process, we recommend using Conda [5] to manage dependencies. Users can execute our provided script to set up exLong and its required components:

```
$ ./scripts/prepare_conda_env.sh
```

We also offer Docker-based installation options. The Docker image can be built and run with:

```
$ docker build -t exlong .
$ docker exec -it exlong /bin/bash
```

Furthermore, for integration with the ollama Docker image, the users can use our Docker Compose setup:

```
$ docker compose up -d
$ docker exec -it exlong-tool-1 /bin/bash
```

4 Tool Usage

In this section, we introduce how to use exLong for developer-oriented use case and machine-oriented use case.

4.1 Developer-oriented use case

For the developer-oriented use case, where exLong generates an EBT for a user-specified target throw statement, our tool's CLI requires the following parameters: the local path or remote link to the git repository, the path to the file containing the MUT, the line number of the beginning of MUT's definition, the path to the file containing the target throw statement, the line number of the target throw statement, and the path to the destination test file.

Additionally, exLong's CLI accepts the following optional parameters: a commit SHA (default: latest commit on the main branch), name of the test method to be written by exLong (default: none), whether exLong should used quantized LLM (default: true), whether exLong should sample multiple candidate EBTs and select the best test based on runtime evaluation (default: false), and the output file path for the generated EBT (default: `./output.java`).

An example command to invoke developer-oriented use case of exLong is as follows:

```
$ python -m etestgen.cli user_view \
    --repo_path=./Wisp \
    --mut_file_path=Scheduler.java \
    --mut_line=180 \
    --quant=true \
    --throw_file_path=Scheduler.java \
    --throw_line=340 \
    --test_context_path=SchedulerTest.java
    --sha="ce1d9f3cb1944115ad98b4428ea24b24ab3faf56" \
    --test_name=testSchedulerError \
    --pick_best=True \
    --output_file=./ExlongTest.java
```

Table 1: Results on developer-oriented use case with ground-truth EBT's name in the prompt.

Models	Compilable%	Runnable%	ThrowCov%
GPT3.5-few-shot	75.12	61.29	48.39
CAT-LM	71.83	36.64	30.03
exLong	**82.10**	**67.36**	**59.45**

4.2 Machine-oriented use case

In the machine-oriented use case, where exLong generates EBTs for the entire codebase. The only required parameter for exLong's CLI is the path or link to the git repository. The CLI also accepts commit SHA, option to sample multiple EBTs, option to use quantized LLM, time budget for exLong to finish, and path to output file as optional parameters.

An example command to invoke developer-oriented use case of exLong is as follows:

```
$ python -m etestgen.cli machine_view  \
       --repo_link= \
           "https://github.com/Coreoz/Wisp.git" \
       --sha="ce1d9f3cb1944115ad98b4428ea24b24ab3faf56" \
       --timeout=1000
```

5 Evaluation

Following prior work [26], we collect our dataset from Java projects in CodeSearchNet [19], which are available on GitHub. We evaluate exLong's performance with full precision LLM under both developer-oriented use case and machine-oriented use case. For developer-oriented use case, we benchmark exLong on a subset of 434 examples from which we are able to extract stack traces. For machine-oriented use case, we evaluate exLong on 649 examples, filtering out data for which our heuristic failed to locate the corresponding destination test file.

We evaluate EBTs generated by exLong using the percentage of generated EBTs that can be compiled (Compilable%), can be executed (Runnable%), and those that are semantically valid and are targeting the throw statement specified by developers (ThrowCov%). We compare exLong against a widely used foundation model, GPT3.5, and a specialized test-generating LLM, CAT-LM. Our results are shown in Table 1. We observe that exLong outperforms all the baselines on all metrics. exLong achieves higher performance for both generating executable EBTs (Runnable%) and EBTs that cover the target throw statements (ThrowCov%). Specifically, exLong outperforms GPT3.5 by 9.9% and 22.8% on Runnable% and ThrowCov%, respectively. Similarly, exLong outperforms CAT-LM by 83.8% and 98.0% on Runnable% and ThrowCov%, respectively.

For machine-oriented use case, we evaluate the tool's ability to cover throw statements within a given repository with ThrowCov%, which measures the percentage of target throw statements covered by the generated EBTs. We benchmark exLong against two widely-used analysis-based test generation tools: Randoop [28, 31] and EvoSuite [12]. Our results, illustrated in Figure 5, indicates that exLong covers the most target throw statements. For more details of our evaluation, refer to the full paper [44].

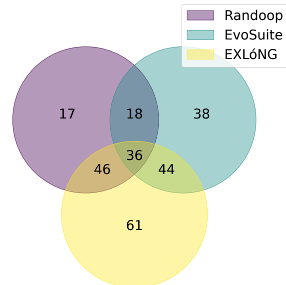

Figure 5: Venn diagram of target throw statements coverage by exLong, Randoop, and EvoSuite on all 30 projects.

6 Related Work

Recent studies have leveraged transformer models for test generation [10, 20, 25, 26, 30, 35, 36, 38, 43]. Some approaches use conditions to guide the generation process [3, 4, 33], while others utilize existing test cases as context [10, 26, 30, 36]. Our work uniquely combines non-exceptional tests with stack traces and guard expression to guide exceptional test generation.

Non-LLM test generation approaches include random-based [28, 31], search-based [12, 16, 22, 23], and constraint-based [11, 13, 18] strategies. While tools like Randoop and EvoSuite can generate tests for exceptional behaviors, they neither guarantee coverage of specific exceptional paths nor consistently produce readable test cases due to their random nature.

7 Conclusion

We presented exLong, a novel command-line tool that leverages large language models to generate exceptional behavior tests (EBTs). exLong offers two practical use cases: developer-oriented use case, which generates an EBT for a specific method and target throw statement, and machine-oriented use case, which automatically creates tests for all target throw statements in a repository. To make exLong accessible to general users, we provide an option which uses a quantized fine-tuned LLM to reduce the computational cost of running inference. We believe that exLong targets an important task in software testing and demonstrated strong performance. By simplifying the process of generating tests, exLong enables developers to more easily create comprehensive test suites that cover exceptional behaviors.

Acknowledgments

We thank Nader Al Awar, Jayanth Srinivasa, Aditya Thimmaiah, Zijian Yi and Zhiqiang Zang for their feedback. We acknowledge the Texas Advanced Computing Center at The University of Texas at Austin and Digital Research Alliance of Canada for providing HPC resources that have contributed to the research results reported within this paper. This work is partially supported by the US National Science Foundation under Grant Nos. CCF-2107291, CCF-2217696, CCF-2313027, CCF-2403036; as well as AST-2421782 and Simons Foundation MPS-AI-00010515 (NSF-Simons AI Institute for Cosmic Origins – CosmicAI). This work was in part supported by Cisco Research. Any opinions, findings and conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of Cisco Research.

References

[1] Hussein Almulla and Gregory Gay. 2020. Learning how to search: Generating exception-triggering tests through adaptive fitness function selection. In *International Conference on Software Testing, Verification, and Validation*. 63–73.

[2] Rafael Di Bernardo, Ricardo Sales Jr., Fernando Castor, Roberta Coelho, Nelio Cacho, and Sergio Soares. 2011. Agile testing of exceptional behavior. In *Brazilian Symposium on Software Engineering*. 204–213.

[3] Arianna Blasi, Alessandra Gorla, Michael D Ernst, and Mauro Pezzè. 2022. Call me maybe: Using NLP to automatically generate unit test cases respecting temporal constraints. In *Automated Software Engineering*. 1–11.

[4] Islem Bouzenia and Michael Pradel. 2023. When to say what: Learning to find condition-message inconsistencies. In *International Conference on Software Engineering*. 868–880.

[5] Conda. 2024. Conda. https://docs.conda.io/projects/conda/en/stable.

[6] Ermira Daka. 2018. *Improving readability in automatic unit test generation*. Ph. D. Dissertation. University of Sheffield.

[7] Ermira Daka, José Miguel Rojas, and Gordon Fraser. 2017. Generating unit tests with descriptive names or: Would you name your children thing1 and thing2?. In *International Symposium on Software Testing and Analysis*. 57–67.

[8] Francisco Dalton, Márcio Ribeiro, Gustavo Pinto, Leo Fernandes, Rohit Gheyi, and Baldoino Fonseca. 2020. Is exceptional behavior testing an exception? An empirical assessment using Java automated tests. In *International Conference on Evaluation and Assessment in Software Engineering*. 170–179.

[9] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. 2024. LLM.int8(): 8-bit matrix multiplication for transformers at scale. In *International Conference on Neural Information Processing Systems*.

[10] Khalid El Haji, Carolin Brandt, and Andy Zaidman. 2024. Using GitHub Copilot for test generation in Python: An empirical study. *International Workshop on Automation of Software Test*, 45–55.

[11] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of computer programming* 69, 1-3, 35–45.

[12] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic test suite generation for object-oriented software. In *International Symposium on the Foundations of Software Engineering*. 416–419.

[13] Patrice Godefroid. 2012. Test generation using symbolic execution. In *Annual Conference on Foundations of Software Technology and Theoretical Computer Science*.

[14] Alberto Goffi, Alessandra Gorla, Michael D. Ernst, and Mauro Pezzè. 2016. Automatic generation of oracles for exceptional behaviors. In *International Symposium on Software Testing and Analysis*. 213–224.

[15] James Gosling. 2000. *The Java language specification*. Addison-Wesley Professional.

[16] Mark Harman and Phil McMinn. 2009. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *Transactions on Software Engineering* 36, 2, 226–247.

[17] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. 2003. *C# language specification*. Addison-Wesley Longman Publishing Co., Inc.

[18] Josie Holmes, Iftekhar Ahmed, Caius Brindescu, Rahul Gopinath, He Zhang, and Alex Groce. 2020. Using relative lines of code to guide automated test generation for Python. *Transactions on Software Engineering and Methodology* 29, 4, 1–38.

[19] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.

[20] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. 2023. CodaMosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *International Conference on Software Engineering*. 919–931.

[21] Luan P. Lima, Lincoln S. Rocha, Carla I. M. Bezerra, and Matheus Paixao. 2021. Assessing exception handling testing practices in open-source libraries. *Empirical Software Engineering* 26, 5.

[22] Yu Liu, Pengyu Nie, Anna Guo, Milos Gligoric, and Owolabi Legunsen. 2023. Extracting Inline Tests from Unit Tests. In *International Symposium on Software Testing and Analysis*. 1–13.

[23] Yu Liu, Aditya Thimmaiah, Owolabi Legunsen, and Milos Gligoric. 2024. ExLi: An Inline-Test Generation Tool for Java. In *International Symposium on Software Testing and Analysis*. 1–5.

[24] Diego Marcilio and Carlo A. Furia. 2021. How Java programmers test exceptional behavior. In *International Working Conference on Mining Software Repositories*. 207–218.

[25] Pengyu Nie. 2023. *Machine learning for executable code in software testing and verification*. Ph. D. Dissertation. The University of Texas at Austin.

[26] Pengyu Nie, Rahul Banerjee, Junyi Jessy Li, Raymond J. Mooney, and Milos Gligoric. 2023. Learning deep semantics for test completion. In *International Conference on Software Engineering*. 2111–2123.

[27] OpenAI. 2024. GPT-3.5-turbo. https://platform.openai.com/docs/models/gpt-3-5-turbo.

[28] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed random test generation. In *International Conference on Software Engineering*. 75–84.

[29] Annibale Panichella, Sebastiano Panichella, Gordon Fraser, Anand Ashok Sawant, and Vincent J Hellendoorn. 2022. Test smells 20 years later: Detectability, validity, and reliability. *Empirical Software Engineering* 27, 7, 170.

[30] Nikitha Rao, Kush Jain, Uri Alon, Claire Le Goues, and Vincent J. Hellendoorn. 2023. CAT-LM: Training language models on aligned code and tests. In *Automated Software Engineering*. 409–420.

[31] Brian Robinson, Michael D Ernst, Jeff H Perkins, Vinay Augustine, and Nuo Li. 2011. Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs. In *Automated Software Engineering*. 23–32.

[32] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code Llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

[33] Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. 2024. Code-Aware prompting: A study of coverage guided test generation in regression setting using LLM. In *International Symposium on the Foundations of Software Engineering*. 951–971.

[34] Victor Sanh, Albert Webson, Colin Raffel, Stephen H Bach, Lintang Sutawika, Zaid Alyafeai, Antoine Chaffin, Arnaud Stiegler, Teven Le Scao, Arun Raja, et al. 2021. Multitask prompted training enables zero-shot task generalization. *arXiv preprint arXiv:2110.08207*.

[35] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. An empirical evaluation of using large language models for automated unit test generation. *Transactions on Software Engineering*.

[36] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2020. Unit test case generation with transformers and focal context. *arXiv preprint arXiv:2009.05617*.

[37] Guido VanRossum and Fred L Drake. 2010. *The Python language reference*. Vol. 561. Python Software Foundation Amsterdam, The Netherlands.

[38] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software testing with large language models: Survey, landscape, and vision. *Transactions on Software Engineering*.

[39] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2023. Self-Instruct: Aligning Language Models with Self-Generated Instructions. In *Annual Meeting of the Association for Computational Linguistics*. 13484–13508.

[40] Jason Wei, Maarten Bosma, Vincent Y Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M Dai, and Quoc V Le. 2021. Finetuned language models are zero-shot learners. *arXiv preprint arXiv:2109.01652*.

[41] Michael Yang, Jeffrey Morgan, Daniel Hiltgen, Bruce MacDonald, Matt Williams, Patrick Devine, Blake Mizerany, Michael, Jesse Gross, Josh, royjhan, Jeremy, frob, Dane Madsen, Parth Sareen, Eva H, Mark Ward, James Braza, Arne Müller, Hernan Martinez, Ikko Eltociear Ashimine, Rapid Architect, Sam, Timothy Jaeryang Baek, slouffka, Alexander F. Rødseth, tusharhero, Eli Bendersky, Lei Jitang, and Mraiser. 2025. *Ollama*. https://github.com/ollama/ollama

[42] Zhewei Yao, Reza Yazdani Aminabadi, Minjia Zhang, Xiaoxia Wu, Conglong Li, and Yuxiong He. 2022. ZeroQuant: Efficient and affordable post-training quantization for large-scale transformers. In *Conference on Neural Information Processing Systems*.

[43] Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. 2023. No more manual tests? Evaluating and improving ChatGPT for unit test generation. *arXiv preprint arXiv:2305.04207*.

[44] Jiyang Zhang, Yu Liu, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. 2025. exLong: Generating exceptional behavior tests with large language models. In *International Conference on Software Engineering*.