eFish'nSea: Unity Game Set for Learning Software Performance Issues Root Causes and Resolutions

Andrew Quinlan
Stevens Institute of Technology
Hoboken, New Jersey, USA
andrewquinlan22@gmail.com

Ryan Mercadante
Stevens Institute of Technology
Hoboken, New Jersey, USA
rmercada@stevens.edu

Vincent Tufo
Stevens Institute of Technology
Hoboken, New Jersey, USA
vtufo@sbcglobal.net

Jonathan Morrone
Stevens Institute of Technology
Hoboken, New Jersey, USA
jdmorrone@comcast.net

Lu Xiao
Stevens Institute of Technology
Hoboken, New Jersey, USA
lxiao6@stevens.edu

ABSTRACT

Prior research has revealed eight recurring types of software performance issues, their root causes, and resolutions. People who do not have a strong programming background may benefit from learning about these concepts to form "performance-aware" thinking in general problem-solving and also prepare them for future studies in related fields. This paper intends to share a set of eight Unity games, named *eFish'nSea*, with the community to generate broader impacts. Each game maps to one of the eight performance issue types, conveying the technical concepts through fun and easy-to-capture game mechanisms, and providing feedback to players regarding how "efficient" they played the games. The authors have conducted stress testing and delivered several practical sessions of the games, showing great promise in the indicated intention.

1 INTRODUCTION

Software performance is one of the most critical quality attributes that determine the success of a software system [1–4], concerning the timeliness and resource consumption during system execution. Users are more likely to switch to competitors' products due to performance bugs compared to other general bugs [4].

Zhao et. al [5, 6] investigated a total of 570 real-world performance issue reports, revealing eight recurring root causes, and the respective resolutions, to performance issues in software systems. Practitioners who have rich experience in programming could benefit from this knowledge to facilitate performance engineering. Meanwhile, people who do not have a strong programming background may also benefit from learning about these concepts to form "performance-aware" thinking in general problem-solving and also prepare them for future studies in related fields. However, they may find it difficult to understand the related technical concepts due to having a lack of programming background, such as *API* and *loop*.

Research has proved the effectiveness of using computer games in learning [7–12]. This paper is motivated to help people who do not have a strong programming background gain an understanding of software performance issues from Zhao et. el's research [5, 6]. We introduce a set of eight Unity games, named *eFish'nSea*. Each game maps to one of the eight performance issue types, conveying the technical concepts through fun and easy-to-capture game mechanisms, and providing feedback to players regarding how "efficient" they played the games. The authors conducted stress testing to ensure that the games could provide a seamless user experience under expected levels of user traffic [13, 14]. The authors also delivered the games in various learning sessions, with approximately 350 students of various levels. We would like to share our games as a resource to the community to generate broader impacts. Educators who are interested in using our games can gain access here [1].

2 BACKGROUND

This section introduces the eight recurring performance issues and their respective resolutions revealed in Zhao et. al [5, 6]'s work. **1. Inefficient Iteration:** "The status of loop iterations remains the same and the iterations become useless." These issues are usually easily resolved by checking the loop condition, and *break* or *return* from the loop when a certain condition is satisfied [15]. **2. Inefficient Synchronization:** These problems "are caused by synchronization issues among multiple threads". More specifically, these issues can occur "because different threads have to access the same resource, and thus they have to wait for each other". The resolution is to employ an appropriate synchronization mechanism, such as ensuring that resources are accessed in a certain order by different threads. **3. Inefficiency under Special Cases:** A program runs well under normal circumstances, but it becomes extremely slow, or even crashes, in special cases. Special cases are usually triggered by inputs that are either null or exceed commonly expected values [16–18]. A typical resolution is to look out for special cases and design a special algorithm for treating such cases. **4. Inefficient API Usage:** In programming for the same function, different APIs are available, but some are more efficient than others in a certain context [19]. The sub-optimal choice of an API could significantly compromise the performance of a program. The typical resolution is to replace an inefficient API with a more efficient one for the usage scenario [20–22]. **5. Inefficient Data Structure:** Similarly, the choice of the appropriate data structure could also significantly impact the performance of a program [23], regardless of the programming language. So the typical resolution is to replace an inefficient data structure with a more efficient one for the computation scenario. **6. Repeated Computation:** A program repeatedly conducts the same process of computation and thus produces the same output, which does not change the status of the program. The resources for the computation go to waste and compromise the performance of the program. To resolve such issues, developers should 1) store the results of a complicated computation

[1] https://efish-n-sea.github.io/

that tends to recur, and 2) add checking conditions before repeating the computation and retrieve the saved results when conditions are consistent[24]. As such, repeated computation can be eliminated, and time and resources can be saved. **7. Redundant Data Processing:** Redundant or tedious data processing, such as processing a large chunk of data in small units (bit by bit or pixel by pixel), leads to excessive and tedious data processing overhead. The typical resolution is to design a more efficient data processing strategy to avoid heavy overhead and process large chunks of data at once. **8. General Inefficient Computation:** These issues are caused by any other algorithmic inefficiency that cannot be captured by the above categories. The resolution to such problems is usually case-by-case, depending on the problem and program design. For example, the order of checking two conditions in an *and* clause could yield to significant difference in processing time, if one of the conditions is mostly true while the other the mostly false. As such, checking the *true* condition first can avoid checking the second condition to save time.

3 EFISH N' SEA GAME DESIGN

The games are named, *eFish n' Sea*, created as a pseudo-homophone to "efficiency". Each of the individual games follows this theme of fish and sea with their selection of characters and environments, deemed to be appropriate and appealing for the project's targeted younger audience. The artwork used in each of the games was acquired from various internet sources. Some artwork was generated using the OpenAI project DALL-E [2], and a few pieces of artwork were purchased from the Unity Asset Store [3].

3.1 *Loopy Fish*: Inefficient Iteration

(a) Game Initialization

(b) Cleaned Shell

Figure 1: *Loopy Fish*: Inefficient Iteration

The game *Loopy Fish* was designed to facilitate education about *Inefficient Iteration*. Figure 1a illustrates the initialization of this game. That is, the user is presented with a set of twenty seashells, each of which has a different level of "dirtiness". The more green a seashell is, the higher the level of "dirtiness", requiring more effort/time to clean it. As the game starts, a catfish enters the window and begins looping through the set of shells. Each time a shell is passed by the catfish, it gets cleaner by amount δ. Once a shell is completely cleaned, it is visually indicated by a sparkling effect, as shown in Figure 1b. Note that, due to the random distribution of dirtiness to each shell, some shells will be fully cleaned with fewer iterations than others. Additionally, it is critical to note that while the catfish is traveling along its set path, it moves substantially faster when there is no shell in its path and it does not attempt to clean a shell any further.

The objective for a player is to minimize the time used to clean all the seashells. Analogous to the *Inefficient Iteraction* root cause and resolution, there are two general strategies to play this game, which will yield a "zero-star" and "three-star" rating, respectively, reflecting the overall effectiveness of the player's ability to minimize needless iterations: 1) The "zero-star" (inefficient) strategy is to let the catfish repeatedly loop through every single seashell again and again, even after it has been completely cleaned—leading to useless loops; and 2) the "three-star" (optimal) strategy is to click on a completely cleaned shell to remove it from the loop cycle, similar to adding a checking condition to the loop to break from it. The first strategy takes approximately 45 seconds longer than the optimal strategy with *Inefficient Iteration* being eliminated in this game. The rating, which is linked directly to the total time spent cleaning all the shells, serves as performance feedback for the player.

3.2 *Macaroni Munchies*: Inefficient Synch.

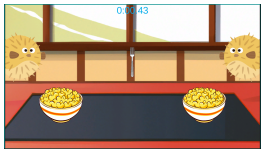


Figure 2: *Macaroni Munchies*: Inefficient Synchronization

This game was developed to convey the concept of *Inefficient Synchronization*, which was chosen to be represented by cooperation between two players. As shown in Figure 2, the game contains three key elements: two pufferfish—i.e. the two players; two bowls of macaroni—i.e. tasks to be accomplished by the two players; and a single fork—i.e. the shared resource that needs to be synchronized for the tasks. The two players can use the Q and P keys on the keyboard to move the fork to either the left or right pufferfish to feed it a bite of macaroni. Each bowl has a total of *five* bites before it is emptied. Critical elements of the game to convey this concept of synchronization are that 1) it takes a pufferfish *ten* seconds to finish chewing a bite, 2) a pufferfish cannot take another bite before finishing chewing, and 3) the fork will be "locked" by a pufferfish and forced to wait for it to finish its current chewing cycle if it is brought to an actively chewing pufferfish, thereby making it impossible for the other pufferfish to utilize it during this time.

The two pufferfish need to effectively synchronize their combined utilization of the fork, so as to finish their entire bowls of macaroni in the minimum amount of time. The strategies to play this game that reflect the root cause and resolution to *Inefficient Synchronization* correspond to both the "zero-star" and "three-star" ratings, respectively, given at the end of the game to provide feedback to the players. More specifically, 1) the "zero-star" strategy is for one pufferfish (i.e. player) to constantly maintain the fork until it finishes its bowl, after which time the fork is finally released to the other pufferfish. There is no synchronization between the two players in this strategy. 2) The "three-star" strategy is that the two

[2] https://openai.com/dall-e-2
[3] https://assetstore.unity.com/

pufferfish rotate the fork (shared resource) with the best possible synchronization—i.e. one pufferfish grabs the fork and feeds itself a bite while the other is chewing. The time needed to finish the game is reduced to half when using the optimal strategy, reflecting the resolution of *Inefficient Synchronization* compared to the "zero-star" solution.

3.3 *Fishing Frenzy*: Inefficiency in Special Cases

(a) Stage 1

(b) Stage 2

Figure 3: *Fishing Frenzy*: Inefficiency under Special Cases

The *Fishing Frenzy* game was developed to convey the concept of *Inefficiency under Special Cases*. As shown in Figure 3, the game has two main stages. In the first stage, shown in Figure 3a, the player is tasked with selecting up to five buckets for an upcoming fishing trip in a bait and tackle shop. There are four types of buckets: *fish*, *shark*, *turtle*, and *trash*, which can hold five fish, one shark, one turtle, and ten pieces of trash, respectively, and this information is conveyed to the user through the clerk's dialogue. The user can select between 0 and 5 total buckets and is not limited by the number of each type of bucket. The player can subsequently press the "Checkout" button to move on to the next stage of the game, shown in Figure 3b, which takes place on an underwater fishing dock. The buckets from the first stage are shown on the dock, just behind the fisherman. In the second stage, the fisherman rapidly catches a variety of objects from the water and flings each toward the buckets on the dock behind him. If there is an open bucket of the matching type, the caught object will fly directly into that bucket, and award the user respective points—fish, sharks, turtles, and trash award the user 100, 250, 250, and 50 points, respectively. However, if there is no available matching bucket (i.e. not selected at all, or the appropriate buckets are full), the caught object will land on the dock, fall back into the water, and award the user 0 points. The game is designed such that the fisherman will always catch exactly ten fish, ten pieces of trash, and either one shark or one turtle. Notably, the game actually makes the decision of giving a turtle or a shark only semi-randomly; particularly, if the player chooses a shark bucket but not a turtle bucket, only a turtle will ever be caught, and vice versa. As a result, being prepared for one scenario but not the other will still result in a loss of points.

The objective for the player is to earn the maximal possible points, which requires them to prepare for every possible scenario—including special cases, such as the case of catching a shark or of catching a turtle. Thus, the player needs to observe the "profile" of the caught objects and prepare an optimal combination of buckets—i.e. two fish buckets, and one of each other type of bucket—in order to receive the highest score (1,750) and a three-star rating. This reflects the resolution to *Inefficiency under Special Cases*. In comparison, the sub-optimal strategy would be to only prepare certain types of buckets, thereby losing points when special objects, such as a turtle or a shark, are caught, leading to a lower score and zero-, one-, or two-star rating.

3.4 *Fintastic Toast*: Inefficient API Usage

(a) Indoor

(b) Outdoor

Figure 4: *Fintastic Toast*: Inefficient API Usage

The *Fintastic Toast* game was designed to convey the concept of *Inefficient API Usage*, as shown in Figure 4. First, the player is presented with a kitchen with clickable cooking implements to use in the task of making toast, shown in Figure 4a. The options in the kitchen include a oven, a box of matches, a waffle iron, and a toaster. Additionally, the user has the option to go outdoors, and explore other options such as a grill, a campfire, a flamethrower, and even the sun. It is quite straightforward for the player to figure out the optimal tool to use—clearly, the toaster would be optimal, with the oven being a potential secondary choice—leading to a three- and two-star rating respectively. In particular, although the oven can toast the bread, it could potentially take more energy and a longer time to produce the same result, compared to a toaster. Meanwhile, there are no stars awarded for choosing a "tool" that is clearly unrealistic, such as the sun or flamethrower. ***The key takeaway from this game is that one must always consider what is the most effective tool for the task at hand, much like how a software developer should consider which API is most practical for the project.*** If one were cooking a steak, the optimal "API" would be the grill. Just like how changing the item being cooked changes the optimal tool for the job, different scenarios within the context of software development require different APIs for the most efficient results.

3.5 *Moving Day*: Inefficient Data Structures

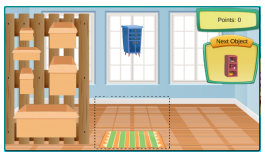


Figure 5: *Moving Day*: Inefficient Data Structures

The game *Moving Day* was designed to facilitate the teaching of *Inefficient Data Structures*. As shown in Figure 5, the player is presented with the scenario of moving out of a house and packing up 20 different items of varying sizes. The player is responsible for identifying the appropriately sized box to package each respective item. In the game setting, 1) if an item is put in the appropriate sized box, the player gets full points for this item; 2) if the box is

too big for the item, the user gets half the amount of points; and 3) if the user selects a box that is too small for the item, the box will be crushed and the player receives no points.

In this setting, the boxes represent the "data structures", the choice of which impacts the efficiency of the move. Having boxes that are too big will cause the move to take longer due to the extra number of trips needed. Conversely, having boxes that are too small will leave items not properly protected during transit. Therefore, the best strategy for the player is to carefully select the most appropriate box for each given item to receive a three-star rating. Meanwhile, choosing boxes which are too big or too small will compromise the scores, accordingly leading to two-, one-, or zero-star ratings at the end. This is analogous to improper data structure choice potentially resulting in inefficiency of storage and/or data processing, or potentially even data loss.

3.6 *Crabsworth's Cave*: Repeated Computation

(a) Initialization **(b) Lock** **(c) Notepad**

Figure 6: *Crabsworth's Cave*: Repeated Computation

Crabsworth's Cave was designed to convey the concept of *Repeated Computation*. The game revolves around the player navigating a maze-like space shown in Figure 6, in which the doors that lead from one room to the next are locked until the player enters the correct code for the simple puzzle, shown in Figure 6b. Many of the puzzles in the rooms ask the player to recall the solutions to earlier puzzles and repeat and/or modify the solutions to move on to the next. As such, the player may have to return (i.e. analogous to recomputation) to a previous room to retrieve the solution. Alternatively, the game also offers the player use of a notepad (see Figure 6c) for the player to record previously found solutions so that, whenever needed again, they can be looked up in the notebook, as opposed to retracting back to previous rooms (thereby eliminating the repeated "computation").

The objective for the player is to navigate the entire maze while making as few repeated visits to rooms as possible. The principle is that by "caching" the previous solutions in the notebook, the player can refer to and reuse the previous answers without needing to "recompute" them by returning to the previous rooms. This allows the player to be more efficient and thereby get a better rating. Particularly, to get a three-star rating, the player must clear the maze in 14 room transitions or less, which is possible even in the first exposure to the game assuming the player can properly track all of the information needed. Larger numbers of room transitions will accordingly receive lower ratings, with 18 being the cutoff for two stars and 25 being the cutoff for one star.

3.7 *Load 'Em Up*: Redundant Data Processing

The *Load 'Em Up* scenario, as showcased in Figure 7, commences by presenting players with an array of cardboard boxes, symbolizing

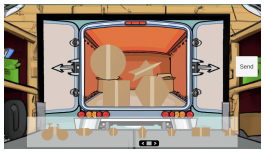


Figure 7: *Load 'Em Up*: Redundant Data Processing

a data set of arbitrary pieces of information, that need to be loaded into a moving truck for transportation. There are thirty boxes in total, with the shapes ranging from basic shapes such as circles, squares, and stars to complex shapes such as a bike or toilet. ***The objective is to optimize the transportation process by minimizing the number of trucks used.*** The act of loading boxes into trucks mirrors data processing operations. Players are granted the freedom to select boxes, determine their placement within the trucks, and decide when to dispatch the trucks. The most efficient strategy hinges on meticulously planning how to maximize the utilization of each truck's capacity before initiating the transportation. When executed adeptly, this approach leads to a three-star rating achieved through a maximum of three truck deployments. It is imperative to underscore that the quantity and dimensions of the boxes are deliberately crafted to foster the belief that accomplishing the task within three attempts is feasible. However, a lack of strategic planning frequently results in a surplus box requiring a fourth truck for transport, yielding a two-star rating. If the number of trucks exceeds four, the outcome is a one-star rating.

3.8 *Shrimplock Holmes*: GIC.

(a) Start: Random Order **(b) Sorted: Color Order**

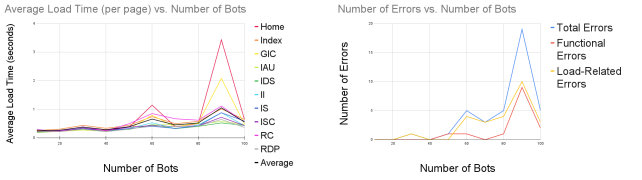
Figure 8: *Shrimplock Holmes*: General Inefficient Comp.

The objective of is to locate and select as many correct criminal fish in the given amount of time by optimizing your searching process. The game begins by prompting the player with a message telling them that they must assist Shrimplock Holmes in finding the criminal fish amongst a large set of fish of varying species, colors, and names. The player then has 2 minutes to locate as many fish criminals as they can. At the top of the screen is a box that contains an image of the criminal fish to show its color and species, as well as a text box that contains its name. On the bottom of the screen are four buttons, three of which are sorting methods for color, species, and name, and the other is to go to the next set of fish, since only so many can fit on the screen. In the beginning, all the fish are sorted randomly and are swimming on and off the screen, as shown in 8a. During this, the player can still hover over the fish to view its name and click on a fish to choose it as the criminal. Once a sorting button is selected the fish will swim off-screen and then swim back on screen in a sorted order, as shown

in 8b. The game is designed to represent how various methods of sorting can be optimal in different cases to find an object in a list of objects. This is ideally made more apparent by the added time constraint needed to find the name of a fish as it is the only unique identifier, however, it takes extra time to access. To achieve a 3-star rating, the player must find at least 10 fish, the other ratings are at other arbitrary values between 0 and 10, with only 0 achieving no stars.

4 STRESS TESTING

We conducted stress testing to ensure that under expected levels of user traffic our servers would have no impact on user experience. We used *LoadTester* [4], which is a commercial tool for web-based stress testing with bots [25, 26]. We vary the number of bots from ten to one hundred, with continuous tests being performed at each increment of ten bots. An increment of ten bots would provide insights into the effects that increasing levels of stress could have on various key metrics. Each test session lasts for three minutes with three phases. First, the test ramped up aggressively for one minute with bots rapidly increasing from zero to the target number (i.e. 10, 20, ..., and up to 100). Then, the test remained at peak load for one minute. Finally, the test ramped down reluctantly for one minute. During a test, a bot would first navigate to the home page, then click on the thumbnail for a game; wait for the game to load, and then return to the home page. Each bot repeats this process for each of the eight games.



(a) Load Time **(b) Errors**

Figure 9: Stress Testing Results

We collected two metrics, including the *load time* and the *number of errors*. The former measures the average time to load a page, including the page for each game, as well as the overall average. The latter counts the number of issues encountered during each test. Errors are further distinguished as *load-related errors*, which are caused by high user traffic, and *functional errors*, which are other errors. For example, a game still loading after 15 seconds is reported as a *load-related error*; while *"could not find the Home button..."* is a *function error* likely caused by incompatibility with different screen sizes. Figure 9 shows how the two metrics trend in the stress testing. Figure 9a shows that the average *load time* of the games remains within 1 second, with up to 100 bots. Though the *GIC* game and the home page are the two outliers, they still load within 4 seconds with up to 80 bots. Figure 9b shows that the number of errors is kept within 5 with up to 80 users but increases to a total of 20 with above 80 users. Thus, we recommend a session of 80 students as the maximum capacity.

[4] https://loadster.app/

5 PRACTICAL SESSIONS

We have offered the games in learning sessions to different audiences. In each session, two authors opened up with a quick, interactive lesson on the general concept of efficiency. The students then played the games, followed by a group discussion to share their learning outcomes. The first two sessions were held at a local elementary school for a total of 40 4th-grade students, who were very engaged and enthusiastic. The next six sessions were conducted at a pre-college summer program with a total of around 300 students. The final batch of sessions took the form of focus groups consisting of six college students. The games were generally received well by the two later groups but with higher variations in the level of interest and engagement compared to the first group.

6 LIMITATIONS AND FUTURE WORK

We acknowledge that we have not conducted a thorough and scientific evaluation of the games. The practical sessions described above only serve as anecdotal evidence of the feasibility of the games. The learning effectiveness of the games, in particular how the game concepts transfer to software development tasks, has not been systematically measured. The objective of this paper is to share the games with the community to explore their potential.

In our future work, we plan to conduct a more rigorous and systematic evaluation of the games. In particular, we aim to evaluate and measure whether and how to best use these games to truly help students of different levels grasp relevant concepts in programming tasks. Also, we aim to understand how the learning effectiveness of the eight games differ, as such we can further improve the games. Lastly, we also aim to investigate who are the most appropriate audiences of the games—as we observed a much higher level of engagement with the 4th-grade students.

7 CONCLUSION

This paper shared a set of eight Unity games, named *eFish'nSea*, intended to help young audiences form "performance-aware" thinking and prepare them for future studies in related fields. Each game maps to one of the eight performance issue types, conveying the technical concepts through fun and easy-to-capture game mechanisms, and providing feedback to players regarding how "efficient" they played the games. The authors have conducted stress testing and delivered several practical sessions of the games, showing great promise in the indicated motivation. Educators who are interested in using the games are welcome to use them.

ACKNOWLEDGEMENT

This work was supported in part by the U.S. National Science Foundation (NSF) under grants CCF-2044888.

REFERENCES

[1] Connie U Smith and Lloyd G Williams. *Performance solutions: a practical guide to creating responsive, scalable software*, volume 1. Addison-Wesley Reading, 2002.
[2] Connie U Smith and Lloyd G Williams. *Software performance engineering*. Springer, 2003.
[3] Murray Woodside, Greg Franks, and Dorina C Petriu. The future of software performance engineering. In *Future of Software Engineering (FOSE'07)*, pages 171–187. IEEE, 2007.

[4] Shahed Zaman, Bram Adams, and Ahmed E Hassan. A qualitative study on performance bugs. In *2012 9th IEEE working conference on mining software repositories (MSR)*, pages 199–208. IEEE, 2012.

[5] Yutong Zhao, Lu Xiao, Andre B Bondi, Bihuan Chen, and Yang Liu. A large-scale empirical study of real-life performance issues in open source projects. *IEEE Transactions on Software Engineering*, 49(2):924–946, 2022.

[6] Yutong Zhao, Lu Xiao, Xiao Wang, Lei Sun, Bihuan Chen, Yang Liu, and Andre B Bondi. How are performance issues caused and resolved?-an empirical study from a design perspective. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, pages 181–192, 2020.

[7] Gunilla Svingby and Elisabet M Nilsson. Research review: Empirical studies on computer game play in science education. *Handbook of research on improving learning and motivation through educational games: Multidisciplinary approaches*, pages 1–28, 2011.

[8] Nergiz Ercil Cagiltay. Teaching software engineering by means of computer-game development: Challenges and opportunities. *British Journal of Educational Technology*, 38(3):405–415, 2007.

[9] Wong Yoke Seng and Maizatul Hayati Mohamad Yatim. Computer game as learning and teaching tool for object oriented programming in higher education institution. *Procedia-Social and Behavioral Sciences*, 123:215–224, 2014.

[10] Nicola Jane Whitton. *An investigation into the potential of collaborative computer game-based learning in higher education.* PhD thesis, 2007.

[11] Nicola Whitton. Motivation and computer game based learning. *Proceedings of the Australian Society for Computers in Learning in Tertiary Education, Singapore*, pages 1063–1067, 2007.

[12] Michael Begg, David Dewhurst, and Hamish Macleod. Game-informed learning: Applying computer game processes to higher education. *Innovate: Journal of Online Education*, 1(6), 2005.

[13] Martin Čihák. Introduction to applied stress testing. 2007.

[14] Myrvin H Ellestad. *Stress testing: principles and practice.* Oxford University Press, 2003.

[15] Linhai Song and Shan Lu. Performance diagnosis for inefficient loops. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 370–380. IEEE, 2017.

[16] Du Shen, Qi Luo, Denys Poshyvanyk, and Mark Grechanik. Automating performance bottleneck detection using search-based application profiling. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 270–281. ACM, 2015.

[17] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. Perffuzz: Automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 254–265. ACM, 2018.

[18] Marc Brünink and David S Rosenblum. Mining performance specifications. In *Proceedings of the 24th International Symposium on Foundations of Software Engineering*, pages 39–49. ACM, 2016.

[19] David Kawrykow and Martin P Robillard. Detecting inefficient api usage. In *Proceedings of the 31st International Conference on Software Engineering*, pages 183–186. IEEE, 2009.

[20] Qiao Huang, Xin Xia, Zhenchang Xing, David Lo, and Xinyu Wang. Api method recommendation without worrying about the task-api knowledge gap. In *Proceedings of the 33rd International Conference on Automated Software Engineering*, pages 293–304. ACM, 2018.

[21] Zhenmin Li and Yuanyuan Zhou. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the ACM SIGSOFT Software Engineering Notes*, pages 306–315. ACM, 2005.

[22] David Kawrykow and Martin P Robillard. Improving api usage through automatic detection of redundant code. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 111–122. IEEE Computer Society, 2009.

[23] Milind Chabbi and John Mellor-Crummey. Deadspy: a tool to pinpoint program inefficiencies. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 124–134, 2012.

[24] Luca Della Toffola, Michael Pradel, and Thomas R Gross. Performance problems you can fix: A dynamic analysis of memoization opportunities. In *Proceedings of the ACM SIGPLAN Notices*, pages 607–622. ACM, 2015.

[25] Linda Erlenhov, Francisco Gomes de Oliveira Neto, Riccardo Scandariato, and Philipp Leitner. Current and future bots in software development. In *2019 IEEE/ACM 1st International Workshop on Bots in Software Engineering (BotSE)*, pages 7–11. IEEE, 2019.

[26] Nataliya Yatskiv, Solomiya Yatskiv, and Anatoliy Vasylyk. Method of robotic process automation in software testing using artificial intelligence. In *2020 10th International Conference on Advanced Computer Information Technologies (ACIT)*, pages 501–504. IEEE, 2020.