ADPTriage: Approximate Dynamic Programming for Bug Triage

Hadi Jahanshahi, Mucahit Cevik, Kianoush Mousavi, and Ayşe Başar

Abstract—Bug triaging is a critical task in any software development project. It entails triagers going over a list of open bugs, deciding whether each is required to be addressed, and, if so, which developer should fix it. However, the manual bug assignment in Issue Tracking Systems (ITS) offers only a limited solution and might easily fail when triagers are required to handle a large number of bug reports. During the automated assignment, there are multiple sources of uncertainties in the ITS, which should be addressed meticulously. In this study, we develop a Markov decision process (MDP) model for an online bug triage problem. In addition to an optimization-based myopic technique, we propose an ADP-based bug triage solution, called ADPTriage, which has the ability to reflect the downstream uncertainty in the bug arrivals and developers' timetables. Specifically, without placing any limits on the underlying stochastic process, this technique enables real-time decision-making on bug assignments while taking into consideration developers' expertise, bug type, and bug fixing time. Our result shows a significant improvement over the myopic approach in terms of assignment accuracy and fixing time. We also demonstrate the empirical convergence of the model and conduct sensitivity analysis with various model parameters. Accordingly, this work constitutes a significant step forward in addressing the uncertainty in bug triage.

Index Terms—Software engineering, bug triage, Reinforcement Learning, Approximate Dynamic Programming, software quality.

I. INTRODUCTION

Bug repositories and Issue Tracking Systems (ITS) are commonly used to track and address software issue reports, whether they are feature improvement requests or bugs arising during the testing or maintenance phase. To manage these requests, open-source software projects mainly rely on issue-tracking platforms, such as Bugzilla, Jira, and GitHub [1]. Bug triage task involves promptly prioritizing bugs and assigning them to appropriate developers. It is preceded by examining the validity of the bugs, determining the possible missing information in the bug report, checking possible duplicate bug reports, setting the severity of bugs, and assigning them to a proper developer [1], [2]. As such, bug triage is deemed as a challenging task that directly affects software quality. Since numerous bugs are reported to open-source software systems every day, the manual triage task in such projects is prone to subjective or erroneous decisions. Accordingly, our focus in this study is on the bug assignment task, which involves assigning bugs to the most appropriate developers at the right time.

Researchers proposed many bug triage approaches to overcome the issues of manually assigning bugs to developers. As bug triage is a multifaceted problem involving different sub-tasks, each study concentrates on a particular aspect of the problem. The majority of the studies in the literature are dedicated to improving the accuracy of the bug assignment by proposing appropriate developers based on the textual information of the bugs [3], [4], [5], [6], [7], [8]. On the other hand, some studies consider various other dimensions of the problem. For instance, [9] developed the CosTriage algorithm that, besides the accuracy of the assignment (i.e., assigning a bug to the right developer), considers the fixing cost of the bugs. CosTriage employs the content-based recommendation and collaborative filtering recommender to construct developers' profiles and approximate the fixing time of each bug type. [10] enhanced CosTriage by proposing an integer programming-based solution that incorporates the developers' suitability, bug fixing time, and software release dates. While [10]'s approach covered several important aspects of the bug triage, [11] proposed an extension of this work by incorporating the developers' schedules and the bug dependencies into bug triage. Additionally, [12] highlighted the importance of fair bug distribution among developers. Their method focuses on avoiding the assignment of bugs to developers who are overloaded by taking into account developers' expertise and current workloads.

Although previous works proposed different algorithms to accommodate various aspects of the bug triage, to the best of our knowledge, the uncertainty in the ITS is not yet explored in any model. For instance, the number of bugs reported to the system does not follow a particular pattern to be easily predicted [13]. Thus, there is no easy way to be certain about whether a severe bug will be reported in the upcoming days so that triagers/developers can plan ahead instead of being involved in less important open bugs in the system. Additionally, the developers' schedules are another source of uncertainty, which may constantly get updated in open-source software systems since not all developers are as dedicated as the ones in a proprietary software system. Therefore, the developer may be available...
the next day or not, depending on their schedule/availability. Accordingly, a triage model for the ITS should account for such uncertainties.

To the best of our knowledge, our study constitutes the first work that provides a stochastic model for the bug triage problem to optimize the long-run bug triage performance. The return is defined based on matching developers’ expertise and bug types and the corresponding action of either assigning the open bugs at the current timestamp or postponing them with the expectation of a better match for the upcoming bugs. We introduce a novel Markov Decision Process (MDP) model for the bug triage problem and leverage Approximate Dynamic Programming (ADP) as a solution technique to capture the uncertainty in the bug triage environment. Our proposed method, which we refer to as ADPTriage, not only assigns the bugs to the most appropriate developers or postpones them to the future but also determines the assignment timing according to the likelihood of having a particular bug type in the system and possible changes in developers’ schedules in the future.

The rest of the article is organized as follows. The background of the main approaches employed in our proposed model is briefly discussed in Section II. The ADPTriage technique and bug report datasets utilized in our research are described in Section III. The numerical experiments and comparisons with the baseline methods are presented in Section IV. The study limitations are discussed in Section V, which is followed by the literature review in Section VI. Finally, the article is concluded with a summary of the work and an elaborate discussion on potential future research directions in Section VII.

II. BACKGROUND

In this section, we provide background information on the methodological approaches utilized in our proposed model.

A. LDA for Categorizing Bug Reports

The Latent Dirichlet allocation (LDA) is an unsupervised, probabilistic topic modeling technique that uses word clusters and frequencies to identify topics in corpora [14]. LDA presumes that the document has \( n \) subjects, to one of which each word is assigned. LDA is commonly used in bug triage problems to determine the bug type given the bugs’ textual information, where the former relates to subjects and the latter to documents [15]. Hence, after removing the stop words, we extract bug descriptions and summaries from a bug report (as textual information) and construct a bag of words. For the sake of consistency, we follow the steps reported in previous studies for our LDA implementation [9], [10], [16], [17]. As [16] pointed out LDA is successful in topic modelling by identifying clusters of words that frequently co-occur in a corpus and representing them as topics. It is an effective unsupervised approach for discovering underlying themes and patterns within bug reports. Throughout this article, LDA category refers to the “bug type”. We also estimate the experience of each developer for each LDA category. We utilize Arun’s technique to determine the ideal number of LDA categories [18], and then, the average bug-fixing time of each developer given each category is computed. Finally, we employ a collaborative filtering recommender to approximate the missing values using the steps suggested by [9].

In the bug classification literature, there exist alternative techniques for identifying bug types [19]. One such approach is proposed by [20], which employs crowdsourcing to automatically classify software defects. This method involves using data labelled by multiple non-expert annotators to train classifiers, and the authors demonstrate how this approach can be effective in situations where expert knowledge is not readily available. In a recent study, [17] employ a deep learning model with a word embedding technique to learn bugs’ feature descriptions in the user manual and classify issue reports based on those features. Their approach achieves higher performance for classifying bug reports than the state-of-the-art methods. While we use LDA for topic modelling in our ADPTriage method, it can be configured to work with other alternative techniques for determining bug types.

B. Approximate Dynamic Programming

The general task of optimizing decisions over a time horizon extends to varied domains and backgrounds. Such problems are typically considered as sequential decision-making problems, and given sets of states, decisions, and rewards, they may be formulated as a Markov Decision Process (MDP) [21]. These models are commonly utilized in resource allocation problems, e.g., allocating fleets of vehicles [22], manpower planning problem [23], and assigning a task to an expert [24]. In typical MDP problems, an agent, according to its current state, takes an action which moves the agent to an intermediate state, called the post-decision state, where the agent receives new information from the environment, also called exogenous information, and accordingly, it proceeds to the next state (see Fig. 1).

Generally, MDP models consist of seven main components. The first two, the state and action space, \( s_t \in \mathcal{S}_t \) and \( a_t \in \mathcal{A}_t \), respectively, help to define the state of the system as well as the set of feasible decisions (or actions) at each time step \( t \). The next component, exogenous information, \( \xi_{t+1} \), independently arrives in the system and is used along with the two aforementioned components as input for moving the state of the system forward in time. Such forward movement is explicitly characterized through the use of a transition function, \( s_{t+1} = \text{state}_{\text{next}}(s_t, a_t, \xi_{t+1}) \), along with transition probabilities \( P(s_{t+1}|s_t, a_t) \), which probabilistically shift the system state based on its current state at time \( t \), the corresponding action(s) taken, and the exogenous information. The objective function looks to optimize the expected discounted reward/penalty and provides a policy. More specifically, a policy, \( \pi \in \Pi \) (where \( \Pi \) is the set of all decision policies), is a function
\( A_T^\pi(s_t) \) that prescribes a feasible action \( a_t \) to the agent at every time step \( t \) in a given state \( s_t \). The goal in solving MDP models is to find a policy that maximizes the expected total reward/cost received over a given time horizon. Specifically, optimal policy \( \pi \in \Pi \) solves

\[
\max_{\pi \in \Pi} \mathbb{E}^\pi \left\{ \sum_{t=1}^T \gamma^t R_t(s_t, A_T^\pi(s_t)) \right\},
\]

where \( \gamma \) is the discounting factor and \( R_t \) is the reward accrued in state \( s_t \) for taking action \( a_t \). Expectation term is required due to the information variable \( \xi_t \in \Xi \).

Backward Dynamic Programming (DP) is a well-known method for solving such discrete stochastic optimization problems. In backward DP, MDP problem is broken down into subproblems, and each sub-problem is solved through recursive equations which attempt to capture both the immediate and downstream value of being in a given state at the current decision epoch. Optimization problem in Equation (1) can be written using Bellman optimality equations as follows:

\[
V_t(s_t) = \max_{a_t \in A_t} \{ R_t(s_t, a_t) + \gamma \mathbb{E}[V_{t+1}(s_{t+1})|s_t] \},
\]

where \( V_t(s_t) \) is the value function and \( s_{t+1} = \text{state}_{\text{next}}(s_t, a_t, \xi_{t+1}) \). However, finding an exact solution to Bellman optimality equations (e.g., by using backward DP) proves to be difficult and/or intractable for large problems due to the three "curses of dimensionality", which are related to the presence of (i) a large multidimensional state space making value function approximation difficult, (ii) a large multidimensional action space hindering identification of optimal decisions, and (iii) a multidimensional outcome space impeding computing the expectation of future rewards [25].

ADP, which is a powerful algorithmic framework used to solve such large-scale discrete-time MDP problems, helps to address the aforementioned three curses of dimensionality. Specifically, to address (i), the concept of a post-decision state, \( s_t^{\text{post}} \), is introduced, which helps defining the state of the system after an action has been taken at time \( t \) but prior to the arrival of any exogenous information before transitioning to the next time step \( t+1 \). This allows us to break the transition function into two components, namely, \( s_t^{\text{post}} = \text{state}_{\text{post}}(s_t, a) \) which defines the transition to the post-decision state, and \( s_{t+1} = \text{state}_{\text{next}}(s_t^{\text{post}}, \xi_{t+1}) \) which characterizes the transition to the next state at time \( t+1 \) after the arrival of exogenous information. To address (ii), we estimate the values for the post-decision states via linear function approximation. To address (iii), sample paths are generated over the planning horizon, and a forward DP approach is adopted to solve the Bellman equations, stepping forward in time and repeating the process for multiple iterations. Accordingly, using the post-decision states, we can break down the Bellman equation into two parts:

\[
V(s_t) = \max_{a_t \in A_t} \{ R_t(s_t, a_t) + \gamma V^{\text{post}}(s_t^{\text{post}}) \},
\]

\[
V^{\text{post}}(s_t^{\text{post}}) = \mathbb{E}[V(s_{t+1})|s_t^{\text{post}}, \xi_{t+1}] .
\]

Equation (3) becomes deterministic, making it easier to solve, and Equation (4) is approximated and updated by stepping forward in time and observing sample realizations of exogenous information.

### III. Problem Formulation

We present an online bug triage method that assigns bugs to active developers in an ITS. In particular, we focus on open-source software projects, e.g., ECLIPSEJDT, GCC, and MOZILLA. In open-source software systems, bug arrivals and developers’ availabilities demonstrate a dynamic behaviour and they may evolve over time. We assume that bugs can be clustered into different subcategories according to their textual title and description using LDA. These bugs arrive at ITS according to a stochastic process, with an expected assignment deadline that may vary according to their priority/severity. It is desirable to triage a bug before its deadline. Hence, a late assignment cost is associated with those bugs remaining in the system longer than their due date. We assume a predefined project horizon, which we discretize into time intervals. We denote the discrete set of decision-making epochs by \( T := \{1, 2, \ldots, T\} \). Decisions are taken at the start of each time interval \( t \) while exogenous information is observed between two intervals. We assume that the time intervals between epochs, denoted by \( \Delta \), are equal. Without losing the generalizability, equispaced decision intervals may approach 0, making the model an online bug recommendation system. Similar to [10] and [11], we assume \( \Delta \) to be one day. Therefore, the model assigns bugs to the proper developers once a day. The granularity of this epoch length can be easily adjusted without violating the underlying model structure.

### A. Assumptions

We make the following assumptions to construct an ADP-based solution for the bug triage problem.

- We only evaluate active developers and omit inactive developers due to not having enough information about them [3], [10], [26]. Less active developers may visit the ITS infrequently, and little may be known about their schedule and availability. We define active developers as those whose bug fix number is higher than the interquartile range (IQR) of all developers’ bug fix numbers, using IQR as a measure of central distribution [26]. We acknowledge that in the agile software industry, the list of active developers must undergo periodic updates regularly as some may leave the company. We identify the number of active developers in ECLIPSEJDT, GCC, and MOZILLA as 16, 47, and 128, respectively.

- We define active developers based on the bug fix numbers from a historical dataset collected before the bug triaging process, ensuring that our definition does not incorporate any future information.

- A bug of type \( b \) has the fixing time of \( c_t^b \) if fixed by a developer with the experience \( d_{\text{exp}} \). Similar to the previous studies by [10] and [9], we utilize LDA for topic modeling and then find the average fixing time of each developer given the category. For instance, we have 6, 5, and 5 bug
types defined by Arun’s technique for ECLIPSEJDT, GCC, and MOZILLA, respectively.

• No more than one developer can fix a bug simultaneously.
• If developer $d$ agrees to fix bug $b$, they will be unavailable for the next $c_{df}^b$ epochs. During those times, no new bugs can be allocated to them.
• Each developer determines their vacations or off-days beforehand so that their schedules become updated accordingly.

### B. Dataset

We consider three large open-source software projects in our experiments, namely ECLIPSEJDT, GCC, and MOZILLA. Note that these well-established projects contain large number of bug reports. We collect the bug report data from the bug repositories using the Bugzilla REST API\(^1\), which provides both general bug attributes and bug metadata change history. Table I shows the details of the extracted datasets, including the total number of bugs reported, bug dependencies found, and relevant changes in the bugs’ history, as well as the mean, median, minimum, and maximum fixing time for each project. We use bug reports for the time period between 2010 and 2017 as the training set and those for the time period between 2018 and 2020 as the testing set.

Similar to the earlier research by [10] and [9], we only take into account bugs that match the following criteria:

- Similar to previous studies, we assume that META bugs imitate blocking bugs by linking to other bugs through the “depends on/blocks” mechanism [7]. META bugs aim to group similar bugs and they do not have their own test cases. Accordingly, we exclude them as they do not correspond to the actual bugs that users have reported to the system.
- There are cases where the exact date of fixing a bug is unknown or unavailable. This could happen because the bug is still open and has not been resolved yet or the fixing date was not recorded in the system’s history. In light of this, to ensure that we have enough reliable data, we only take into account the bugs that have been marked as FIXED or CLOSED and for which there is enough information available.
- In certain cases, the date when a bug was assigned to a developer is unknown or recorded after its resolution. Consequently, we classify such bugs as invalid and exclude those from our analysis. This measure is needed to ensure the accuracy and reliability of our bug-fixing time prediction model.
- We categorize developers as active if they have fixed a large number of bugs, which is defined as greater than the IQR of bug fix numbers of all developers. We only consider bugs assigned to active developers during the training phase to be used in the testing phase.
- In our numerical study, we consider bugs whose fixing time is within an acceptable range. We use a threshold of $Q3 + (1.5 \times IQR)$ to identify outliers. The interquartile range (IQR) is the difference between the third quartile (Q3) and the first quartile. The acceptable fixing time for ECLIPSEJDT, GCC, and MOZILLA is 21, 38.5, and 6 days, respectively. We exclude outlier bugs whose fixing time exceeds this threshold.

Table I shows how the dataset sizes change after applying the above criteria. Additionally, we preprocess textual information such as bug titles and descriptions by lemmatizing words, eliminating stop words, numbers, punctuations, and lengthy words (i.e., longer than 20 characters), similar to previous studies [10], [26], [27]. To further process the textual information of the bugs, we merge the titles and descriptions into one cohesive text. This step is necessary as the titles often provide a brief summary of the bug, while the descriptions provide more detailed information. By combining them, we have access to all relevant information about the bug in one place. After merging, we tokenize the text, which involves breaking it down into individual words or tokens.

### C. ADPTriage

We next provide the components of our MDP model and details of the ADP framework.

|                      | ECLIPSEJDT | GCC          | MOZILLA      |
|----------------------|------------|--------------|--------------|
|                      | Training   | Testing      | Training     | Testing      | Training     | Testing      |
| Total bugs reported  | 12,598     | 3,518        | 34,635       | 9,998        | 90,178       | 22,353       |
| Total bug dependencies found | 2,169     | 970          | 4,462        | 3,268        | 71,549       | 19,223       |
| Total relevant changes in the bugs' history | 55,109    | 15,505       | 138,580      | 42,117       | 410,010      | 114,778      |
| Mean and Median fixing time (days) | (41.2, 3) | (15.7, 1)    | (42.3, 3)    | (42.0, 3)    | (27.2, 5)    | (12.6, 4)    |
| Minimum and Maximum fixing time (days) | (1, 1,753) | (1, 423)     | (1, 2,396)   | (1, 681)     | (1, 2,172)   | (1, 550)     |

Table I: Summary Information for the Bug Datasets. The training phase is between Jan. 1st, 2010 and Dec. 31st, 2017, while the testing phase includes the data from Jan. 1st, 2018 to Dec. 31st, 2019.

---

1\(https://wiki.mozilla.org/Bugzilla/REST_API\)
1) State Variables: The state of the system is defined by open bugs and available developers at each epoch as follows:

- A developer is characterized by their state vector \( d = (d_{\text{exp}}, d_{\text{sch}}) \), representing the experience of a developer in solving specific types of bugs according to their LDA category and the schedule of the developer. The experience is estimated based on their history of fixing bugs and the LDA algorithm (see Section II-A). The schedule of a developer (sch) shows the number of epochs until the developer’s availability. For instance, \( \text{sch} = 5 \) implies that, after five epochs (in our case, five days), the developer becomes available. In the same way, \( \text{sch} = 0 \) implies that the developer is available in the current epoch. \( \mathcal{D}_t \) is the set of all developers with their associated attributes at epoch \( t \). As mentioned in Section III, in our formulation, we take each epoch to be one day; however, by appropriately adjusting the corresponding parameters, other epoch lengths can be considered in our formulation as well.

- A two-dimensional attribute vector, \( b = (b_{\text{lag}}, b_{\text{adv}}) \), describes an open bug in the system, capturing its LDA category and deadline. LDA category is used to estimate the fixing time \( c_{b}^d \) of bug \( b \) if assigned to developer \( d \). On the other hand, the deadline attribute corresponds to the acceptable number of epochs left to assign the bug to a developer for its on-time assignment. The deadline attribute belongs to the set \( \mathcal{U} := \{0, 1, \ldots, U\} \). Therefore, each bug has at most \( U \) epochs to be assigned (i.e., the maximum time of \( U \times \Delta \) for its assignment since being reported to the ITS). Exceeding the due date will incur a certain cost in the system. We assume that if a bug is reported to the system between decision epochs \( t-1 \) and \( t \), it enters the ITS at the beginning of epoch \( t \) with

\[
b_{\text{due}} = \min(T - t - 1, U)
\]

In other words, \( b_{\text{due}} \) is initialized by the number of epochs left until the end of the project horizon and the maximum assignment time of a bug. At the end of each epoch, we reduce \( b_{\text{due}} \) by 1 since it should always show the remaining time until the estimated due date. Therefore, if the due date is passed, \( b_{\text{due}} \) will become negative.

Let \( \mathcal{B}_t \) denote the set of all open bugs at epoch \( t \). Also, let \( \mathcal{S}^{\text{bug}}_{t} \) and \( \mathcal{S}^{\text{dev}}_{t} \) be the number of open bugs with attribute vector \( b \) at time \( t \) and the number of available developers with attribute \( d \) at time \( t \), respectively. Then, the system state at \( t \in \mathcal{T} \) is defined as \( s_t := (\mathcal{S}^{\text{bug}}_{t}, \mathcal{S}^{\text{dev}}_{t}) \), incorporating state space of the open bugs \( \mathcal{S}^{\text{bug}}_{t} = \sum_{b \in \mathcal{B}_t} c_{b}^d \) and state space of the available developers \( \mathcal{S}^{\text{dev}}_{t} = \sum_{d \in \mathcal{D}_t} a_{d}^y \).

2) Decision Variables: Given the current system state \( s_t \), we have two different types of decisions at each decision epoch \( t \). The first one consists of bugs deferred to future decision epochs in the hopes of assigning those more cost-effectively (for example, allocating to a developer that becomes available in \( t+1 \) and is more expert in fixing this bug type). We define the variable \( p_{tb} \) for bugs, denoting the number of bugs with attribute \( b \) postponed to the next decision epoch. The second possible decision set would be the assignment to the available developers, consisting of the bugs to be fixed. Thus, \( y_{tdb} \) is the number of developers with attribute \( d \) assigned to bugs with attribute \( b \) at epoch \( t \).

With the above definition, we denote the decision tuple \( a_t = (y_t, p_t) \) at epoch \( t \), where \( y_t \) and \( p_t \) are decision vectors of variables \( y_{tdb} \) and \( p_{tb} \), respectively. We denote the feasible decision set \( \mathcal{A}_t(s_t) \) given the current state of the system \( s_t := (\mathcal{S}^{\text{bug}}_{t}, \mathcal{S}^{\text{dev}}_{t}) \). The tuple \( a_t \in \mathcal{A}_t(s_t) \) should satisfy the following constraints:

\[
\sum_{b \in \mathcal{B}_t} y_{tdb} + h_{td} = \mathcal{S}^{\text{dev}}_{td} \quad \forall d \in \mathcal{D}_t \tag{6}
\]

\[
\sum_{d \in \mathcal{D}_t} y_{tdb} + p_{tb} = \mathcal{S}^{\text{bug}}_{tb} \quad \forall b \in \mathcal{B}_t, \tag{7}
\]

where \( h_{td} \) can be considered as the slack of constraints (6). Constraints (6) ensure that the number of developers with attribute \( d \) assigned to all open bugs does not exceed the total number of available developers with that attribute type. Constraints (7) control the flow of the bugs, making sure that each bug with attribute \( b \) is either assigned to a developer or postponed to the next timestamp. Constraints (6) and (7) have a totally unimodular constraint matrix, implying that integer solutions can be found at all extreme points of the obtained feasible region, as such the integrality restrictions on the \( y_{tdb} \) and \( p_{tb} \) decision variables can be relaxed [28].

3) Cost Function: The total cost incurred as a result of actions \( a_t \) for the state \( s_t \) at epoch \( t \in \mathcal{T} \) is calculated as

\[
\text{Cost}(s_t, a_t) = \sum_{b \in \mathcal{B}_t} f(b_{\text{due}}, t)p_{tb} + \sum_{d \in \mathcal{D}_t} \sum_{b \in \mathcal{B}_t} c_{b}^d y_{tdb}. \tag{8}
\]

where \( c_{b}^d \) is the fixing cost associated with the experience of developer \( d \) in addressing bug of type \( b \). The postponement cost \( f(b_{\text{due}}, t) \) gives the incentive that a bug should be addressed as close as possible to its due date. We assume a linear cost for the postponement as follows:

\[
f(b_{\text{due}}, t) = \frac{T - b_{\text{due}}}{T}. \tag{9}
\]

Using this linear function, we aim to have a postponement penalty if the due date is not reached. When we arrive at the due date, the cost becomes 1, and after the due date has passed, it grows, imposing a higher cost for delaying overdue bugs. Attribute \( b_{\text{due}} \) of the bugs is a function of \( t \), starting from the maximum assignment time, \( U \), and reduced by one at each epoch if we decide to defer the bug. Moreover, \( T \) denotes the total number of epochs, normalizing all the bug due dates.

4) Exogenous Information: We have three types of exogenous information in our model. First, new bugs arrive randomly in the system following the underlying stochastic process. Their arrival times directly follow the actual data. We note that bug arrival time distributions might be different for the training and testing phases. We denote the exogenous information at epoch \( t \) as \( \xi_{t}^{\text{bug}} \), which determines the number of newly reported bugs of each type to the system at the end of epoch \( t \). Note that we do not have any presumption on the independence of bug arrival time in the system. Second, we may have a sudden change in the developers’ schedule. That way, a developer may give short notice of being absent/present in the
next epoch, which is expected in real-life scenarios, denoted by $\xi^{\text{sch-change}}_t$. That is, $\xi^{\text{sch-change}}_t$ specifies any unexpected change in the developers’ schedule. Third, after assigning a bug to a developer, they may accept/reject fixing the bug based on their preference. The parameter $\epsilon$ regulates the likelihood of assignment rejection in the system. In our model, however, we take advantage of the acceptance and rejection actions in the learning process to explore options other than the one recommended by the model. We further discuss the impact of exploration versus exploitation in Section IV.C.1. By defining the exogenous information of declining to fix an assigned bug as $\xi^{\text{rejection}}_t$, the exogenous information arrival vector for epoch $t \in \{1, \ldots, T-1\}$ between decisions epochs $t$ and $t+1$ is denoted as $\xi_t = (\xi^{\text{sch-change}}_t, \xi^{\text{rejection}}_t)$.

5) Transition Function: The transition function, which updates the system state forward in time, can be divided into two parts using the post-decision states. We define the two transition functions as follows:

$$S^{\text{bug-post}}_t = \text{statepost}(a_t, S^{\text{bug}}_t)$$

$$S^{\text{dev-post}}_t = \text{statepost}(a_t, S^{\text{dev}}_t).$$

(10)

Given the current state of the bugs and developers, as well as the tuple of actions, we update the bug and developer states forward in time to their respective post-decision states. These describe the states of bugs and developers after they finish performing the defined actions but before exogenous information is introduced in the next time step. Following that, we define the second set of transition functions as

$$S^{\text{bug-post}}_{t+1} = \text{statenext}(S^{\text{bug-post}}_t, \xi_t)$$

$$S^{\text{dev-post}}_{t+1} = \text{statenext}(S^{\text{dev-post}}_t, \xi_t).$$

(11)

which take the bug and developer post-decision states, together with the exogenous information from the next step time and move the system forward in time to the next set of states at epoch $t + 1$. Fig. 2 shows a typical transition of developers and bugs in the system. In this example, at epoch $t$, two out of three developers are available. Assume that ADPTriage recommends assigning $b_2$ to $d_2$. Accordingly, in the post-decision state, the assigned developer $d_2$ becomes unavailable for the next $c^{\text{sch}}_t$ epochs, whereas the other developer remains available. The unassigned bug will then move to the post-decision state while its $b^{\text{due}}_t$ attribute is reduced by 1. On the other hand, developer $d_3$ becomes available and is added back to the system as reflected in the post-decision state. Next, we observe the exogenous variable. Based on the new information, bug $b_1$ is reported to the system, developer $d_3$ accepts to work on the assigned bug, and there is no last-minute change in the developers’ schedules. Therefore, the system state at $t + 1$ includes two developers and two bugs, respectively.

Based on the aforementioned definition, in the post-decision state, we denote the number of bugs of type $b$ (i.e., those available after the assignment is made) and the number of developers as

$$S^{\text{bug-post}}_t(b_{\text{due}^{\text{post}}}, b_{\text{due}^{\text{post}}}-1) = p_t(b_{\text{due}^{\text{post}}}, b_{\text{due}^{\text{post}}}-1)$$

$$S^{\text{dev-post}}_t(d_{\text{exp}^{\text{post}}}, d_{\text{exp}^{\text{post}}}) = h_t(d_{\text{exp}^{\text{post}}}, d_{\text{exp}^{\text{post}}}) + S^{\text{dev}}_{t}(d_{\text{exp}^{\text{post}}}, d_{\text{exp}^{\text{post}}} = 1)$$

(12)

(13)

where the number of bugs in the post-decision state depends only on the number of postponed bugs with their updated $b^{\text{due}}$ attribute. Moreover, the number of available developers with attribute $d$ in the post-decision state is equal to the number of unassigned developers of the same type and the ones whose $d_{\text{sch}}$ attribute was 1 and is going to be available in epoch $t + 1$ (i.e., their $d_{\text{sch}}$ will become 0). 6) ADP Formulation: Even for medium-sized problem instances, the proposed model for the bug triage problem has intractability in decision outcomes, states, and action spaces. We develop an ADP framework to overcome these intractability issues. Specifically, we propose a look-ahead strategy, ADPTriage, that employs value function approximation to solve the bug triage problem. We leverage Linear Programming, Reinforcement Learning, and Natural Language Processing methodologies to construct ADPTriage.

a) Value function approximation: We employ a linear function approximation for the post-decision state value function as follows:

$$V^{\text{post}}_t(S^{\text{bug-post}}_t, S^{\text{dev-post}}_t) = \left( \sum_{b \in B} v^{\text{post}}_{tb} S^{\text{bug-post}}_t b \right)$$

$$+ \sum_{d \in D} v^{\text{post}}_{td} S^{\text{dev-post}}_t d.$$  

(14)

That is, we decompose the joint value function $V^{\text{post}}_t(S^{\text{bug-post}}_t, S^{\text{dev-post}}_t)$ into the value functions of individual bug and developer types since our system correlates rewards with available bugs’ and developers’ attributes. We sum over all postponed bugs in the first portion of Equation (14) with their expected future value estimate of $v^{\text{post}}_{tb}$. In the second part of Equation (14), we take the sum of all developers with attribute $d$ that are available at epoch $t$ and multiply it with
Algorithm 1: Forward-pass ADPTrage

1. Step 0: Initial post-decision values 
\( \bar{v}_{tb}^{\text{post},0}, \bar{v}_{td}^{\text{post},0} \), state \( s_{t}^{0} \), and step size \( \alpha^{0} \).

2. Step 1: for \( n \in \{1, \ldots, N\} \) do

3. Step 2: Choose a sample path \( (\bar{v}_{ni})_{i \in T} \) for exogenous information.

4. Step 3: for \( t \in \{1, \ldots, T\} \) do

5. Step 3.a: Solve the ADP equation in Equation (15), find the optimal decisions \( a_{t}^{n} \).

6. Step 3.b: Compute the marginal values of Constraints (6) and (7) to estimate \( \bar{v}_{tb}^{\text{Post},n} \) and \( \bar{v}_{td}^{\text{Post},n} \), respectively.

7. Step 3.c: Update post-decision value function approximation of the previous epoch, \( \bar{v}_{t-1,d}^{\text{Post},n-1} \) and \( \bar{v}_{t-1,b}^{\text{Post},n-1} \) using Equation (16).

8. Step 3.d: Move to the new state \( s_{t+1}^{n} \) after taking action \( a_{t}^{n} \) and observing the exogenous information \( \xi_{n}^{t} \) via Equations (10) and (11).

end

10 end

11 Step 4: Return the final approximation values of post-decision states, \( \bar{v}_{n}^{\text{Post},N} = (\bar{v}_{tb}^{\text{Post},N}, \bar{v}_{td}^{\text{Post},N}) \), for the future online decision making.

After approximating the post-decision state values, we update the post-decision value of the previous epoch for all elements in \( s_{t} \) using
\[
\bar{v}_{tb}^{\text{Post},n} = (1 - \alpha^{n})\bar{v}_{tb}^{\text{Post},n-1} + \alpha^{n}\bar{v}_{tb}^{\text{Post},n-1} + \alpha^{n}\bar{v}_{tb}^{\text{Post},n-1} + \alpha^{n}\bar{v}_{tb}^{\text{Post},n-1} \tag{16}
\]
where \( \alpha^{n} \) is the desired step size at iteration \( n \) (e.g., 0.5 for the constant case).

We continue to the next epoch after random information realization \( \xi_{n}^{t} \). By repeating the process for \( N \) iterations, the model returns the final post-decision value estimates, \( \bar{v}_{n}^{\text{Post},N} \), which can be employed for the bug assignment in online bug triage. Specifically, to obtain the bug assignments, we fix the calculated \( \bar{v}_{n}^{\text{Post},N} \) estimates, and use these to generate a policy for any given state of the system.

7) Enhancements: Updating the post-decision state values is crucial in finding the optimal solution. In the baseline ADP framework, we use a constant value (e.g., 0.5) for the step size \( \alpha \). However, in the initial iterations of the algorithm, our knowledge about the future values of the current state is inadequate. As we explore and revisit states, we better understand the long-run costs of a decision made. Therefore, we may adjust updating the step size according to the number of iterations, the number of times we visited a state or the difference between the current state value and its previous one. We explore two enhancements for updating step size, which are detailed below.

a) Harmonic step size update: The harmonic step size at iteration \( n \) is defined as
\[
\alpha^{n} = \max \left\{ \frac{\eta}{\eta + n - 1}, \alpha^{0} \right\} \tag{17}
\]
where we set \( \eta \) to 25, and \( \alpha^{0} \) to 0.05, similar to [29]. It is an arithmetically declining sequence with the limit point of \( \alpha^{0} \). Larger values for \( \eta \) slow down the decline and enhance the learning process.

b) BAKF step size update: The bias-adjusted Kalman filter (BAKF), which is bounded by \( 1/n \) is an efficient way of learning and converging the value function. [25] shows that it leads to a faster convergence compared to other step size updates. The BAKF step size balances observation variation and transient bias. As a result, it uses smaller step sizes for value function observations with a large variation and low bias, whereas it suggests larger step sizes for value function observations with a small variation and high bias.

Algorithm 2 shows the step size estimation via BAKF algorithm. In Step 0, we initialize the parameters of the models as follows: \( \theta^{0} = 0, \alpha^{0} = 1, \beta^{0} = 0, \nu^{0} = 0.01, \dot{v} = 0.2 \) and \( \delta^{0} = 0 \).

D. Myopic Policy

As a benchmark policy, we design an optimization-based myopic policy for the bug triage problem based on the typical intuition regarding the bug assignments. This policy does not take into account the future arrival of bugs or the potential schedule changes of the developers. The term “myopic” means that the cost of decisions made farther down the stream is simply ignored. As a result, it is comparable to the ADP method.
Algorithm 2: BAKF - Optimal Step size Algorithm

1 Step 0: Initialize $\theta^0$, $\alpha^0$, $\beta^0$, $\nu^0$, $\nu$ and $\delta^0$.
2 Step 1: Obtain the new observation $\tilde{\nu}_{\text{Post.}}$.
3 Step 2: Update the following parameters accordingly:
   \[
   \nu^n = \frac{\nu^{n-1}}{1 + \nu^{n-1} - \nu}
   \]
   \[
   \beta^n = (1 - \nu^n)\beta^{n-1} + \nu^n(\tilde{\nu}_{\text{Post.}} - \tilde{\nu}_{\text{Post.}}^{n-1})
   \]
   \[
   \delta^n = (1 - \nu^n)\delta^{n-1} + \nu^n(\tilde{\nu}_{\text{Post.}} - \tilde{\nu}_{\text{Post.}}^{n-1})^2
   \]
   \[
   (\sigma^n)^2 = \frac{\delta^n - (\beta^n)^2}{1 + \lambda^{n-1}}
   \]
4 Step 3: Evaluate the step sizes for the current iteration (if $n > 1$).
   \[
   \alpha^n & = 1 - (\sigma^n)^2
   \]
5 Step 3: Update the coefficient for the variance of the smoothed estimate.
   \[
   \lambda^n = \begin{cases} 
   (\alpha^n)^2, & \text{if } n = 1 \\
   (1 - \alpha^n)^2\lambda^{n-1} + (\alpha^n)^2, & \text{if } n > 1
   \end{cases}
   \]
6 Step 4: Smooth the value function estimate using the obtained step size $\alpha^n$ and Equations (16).

but without the learning component. We set the coefficient of postponement, $f(b_{\text{base}})$, in Equation (8) to a large number (e.g., the Big-M) to force the myopic approach to exploit the full capacity of the developers at each epoch without considering the potential benefits of the postponement. Hence, the myopic approach reduces the bug triage problem into a knapsack problem in which the aim is to reduce the fixing time while imposing constraints on developers’ schedules and capacities. This approach is first introduced by [10] as the release-aware bug triage method (RABT) and it was shown to outperform other baselines, e.g., content-based recommendation and cost-aware bug triage [10], [11]. As a result, we select RABT as the myopic baseline to be compared with our ADPTriage method.

E. Experimental Setup

We implement our approaches by following the three steps listed below.

(1) We leverage the Wayback Machine, introduced by [7], for the training period. Using the tool, we extract bugs’ LDA categories and assignment times. We need such information to understand the distribution of assignment deadlines and also cluster bugs to certain categories according to their textual information.

(2) We run the Wayback Machine once more to extract the distribution of bug arrival times per bug category and developers’ availability. The expertise of each developer in fixing different bug categories is also determined. Accordingly, we obtain a snapshot of the environment, which is essential to formulate an MDP problem.

(3) With bugs’ and developers’ information at hand, we run the ADPTriage according to Algorithm 1. Similar to the previous two steps, ADPTriage is implemented only for the training period. The output of this step is an optimal ADP policy that can be utilized during the testing phase.

After finding the ADP policy, we employ this policy for the bug assignment decisions in the testing phase and report the performance based on the statistics collected over the test data. We note that the LDA categories of the bugs in the testing phase are estimated based on the model we obtained during the training phase. That is, we do not perform any retraining afterwards.

IV. RESULTS

We conduct a detailed numerical study to compare ADPTriage and its enhancements against the myopic approach. We primarily focus on exploring whether ADPTriage postpones a bug to find a suitable developer in future decision epochs. We utilize Gurobi 9.5 in our ADP implementations and implement all of the algorithms in Python.

A. Performance Analysis

By defining the expected time to assign a bug, we aim to find a policy that optimizes the assignment time such that the most suitable developer (i.e., the developer with the shortest fixing time) is assigned to a bug. Therefore, by using the bug fixing time as a proxy for the suitability of a developer, the optimal policy requires shortening the fixing time through the proper assignments. Fig. 3 shows the boxplot of the bug fixing time distribution for all algorithms. We observe an almost 10% reduction in the fixing times of the bugs by assigning the developers via ADPTriage with BAKF update compared to the myopic one. We statistically test this observation to see whether the difference is significant [30], Friedman Aligned Ranks as a non-parametric test is selected to compare the algorithms. Given the same set of fixed bugs, the $p$-value of the test for the bug fixing time is equal to 1.05e$-9$, 8.85e$-23$, and 3.08e$-17$ for EclipseJDT, GCC, and MOZILLA, respectively. Hence, we conclude that differences in algorithms’ fixing times are statistically significant (for $\alpha = 0.05$).

A post hoc analysis is employed to investigate the significance of the differences in fixing time reduction. The post hoc approach evaluates the pairwise differences between all algorithms in terms of the average ranking of absolute difference, enabling us to compare two models side by side [31]. Table II shows the $p$-values of pairwise comparisons using the Nemenyi post hoc test after finding the significance in the Friedman test. As the $p$-values of the myopic approach versus either of the ADP versions are much smaller than $\alpha = 0.05$ for all projects, we conclude that the fixing time (i.e., the suitability of the developers) for the ADP algorithms is significantly shorter than that of the myopic approach. Moreover, an effect size analysis was conducted to supplement our significance testing by computing Cohen’s $d$ values, which is a measure of the magnitude of difference in means. The computed Cohen’s $d$ values for Mozilla, GCC, and EclipseJDT were 0.087,
We define the most suitable person to address a bug as the one with the shortest fixing time, i.e., expert enough to handle the bug as fast as possible. Hence, we determine top-\(k\) developers. It indicates whether the proposed developer by a method is among top-\(k\) developers in terms of the fastest fixing time. Table III shows the models’ accuracy in terms of assigning the bugs to the appropriate developers. ADP algorithms demonstrate enhancement in accuracy even though they tend to postpone the bugs. This observation indicates that these postponements are made to find better developers rather than arbitrary deferrals. The improvement here may seem small; however, we note that according to the developers’ schedules and unpredictable availabilities, this result demonstrates the proper learning process of the ADP.

### B. Convergence Analysis

We demonstrate two performance measures of the ADP versions. First, we show the estimate \(\hat{V}_n^0(S_k)\) of the initial state \(S_0\) for different numbers of iterations \(n\). Second, for various numbers of iterations \(n\), we demonstrate the discounted rewards of employing the learned policy. Specifically, we run a secondary simulation on the side for a certain number of iterations \(n\). We ensure that the inner simulation maintains its characteristics for each iteration to make it comparable. Each of these inner simulations has \(O\) epochs, with the value function estimations fixed and the policy based on these values being followed (i.e., we follow the value function approximation from the previous \(n\) iterations and do not update the value function estimation during these \(O\) epochs). Accordingly, these inner validation epochs provide insights into the policy enhancements during the training phase. Every \(M\)th iteration, i.e., for \(n = 0, M, 2M, \ldots, N\), we run the inner simulation.

Fig. 5 show the realized rewards for this experiment for different ADP versions considering the parameters \(N = 20,000, M = 100\) and \(O = 30\). The initial value for the first iteration does not include any presumption about future costs. Therefore, it is equivalent to the myopic policy result. As the iterations continue, the model improves its policy by estimating the future

### Table II

**Comparison of Fixing Times of Algorithms Using Nemenyi Post Hoc Test**

|                | EclipseJDT |              |              | GCC       |              |              | MOZILLA   |              |              |
|----------------|------------|--------------|--------------|-----------|--------------|--------------|-----------|--------------|--------------|
|                | Myopic     | ADP          | ADP Harm     | ADP BAKF  | Myopic       | ADP          | ADP Harm   | ADP BAKF     | Myopic       |
| Myopic         | 1.000      | 0.001***     | 0.003**      | 0.001***  | 1.000        | 0.001***     | 0.001***  | 0.001***     | 1.000        |
| ADP Harm       | 1.000      | 0.900        | 0.806        | 0.900     | 1.000        | 0.072        | 0.900     | 1.000        | 1.000        |
| ADP BAKF       | 1.000      |              |              |           |              |              |           |              |              |

*Significance codes: 

- **p < 0.001**
- *p < 0.01*
- .01 *p < 0.05*
- .05 *p < 1*

0.072, and 0.108, respectively, indicating a small effect size. While these values suggest that the magnitude of the difference in bug fixing times between our ADP BAKF approach and the myopic approach is small, our Nemenyi post hoc test results demonstrate that this difference is statistically significant. This finding implies that despite the small magnitude, the difference in bug fixing times is consistent and reliable across all three projects. Additionally, we note that, in a practical context, a consistent 10% reduction in bug fixing times can lead to significant resource savings over the life of a project, demonstrating the practical importance of even a small effect size.

The intuitive interpretation for the significantly shorter bug fixing times of the ADP algorithms is that the ADP-based policy postpones bugs to find a better developer. Now, the important question is how much does the model sacrifice the timely fixing to achieve a better triage? Fig. 4 shows the distribution of the due date attribute of the bugs (\(\delta_{\text{due}}\)) in the assignment time. It indicates the number of days we had until on-time assignment. In other words, negative values correspond to late assignments, and positive values relate to the early ones. On average, ADPTriage approaches defer fixing the bugs by less than 2 days for EclipseJDT and almost 4 days for GCC and Mozilla to accomplish its objective of assigning them to a more suitable developer in upcoming epochs to postpone bugs, matching them with the most cost-effective developer. Nonetheless, the average extra postponement seems small compared to the project horizon and deadline distribution. To summarize, ADP can achieve considerable cost reductions compared to the myopic approach for about two to four more periods of deferral. This insight is crucial for the triagers, who may be interested in the magnitude of ADP’s postponements. We also observe a much greater variance in due dates of assigned bugs of the myopic algorithm than that of ADPTriage. Specifically, the BAKF version of ADP tends to triage bugs as close as possible to their due dates (i.e., close to 0). It indicates that the obtained policy of ADPTriage is more reliable in terms of timely resolution of the bugs.

![Fig. 3. The number of fixing days of the bugs during the testing phase.](image)
costs of a decision. As a result, we observe a decrease in the average discounted rewards, noting that we have a minimization problem that aims to reduce the bug-fixing time. It indicates a better bug assignment policy compared to the myopic policy since all the values are smaller than the initial value (the blue line). We observe a higher reduction in discounted rewards for the harmonic and BAKF stepizes compared to the ADP variant with fixed stepize. On the other hand, the fixed stepize variant demonstrates more fluctuation (i.e., instability) compared to the other two enhancements. We examine the convergence of the value estimation for each ADP algorithm in Figs. 5(b) and 5(d). The result indicates a faster convergence of the fixed and BAKF stepize ADP compared to the harmonic stepize one in ECLIPSEJDT. On the other hand, we do not observe such a behaviour for the GCC and MOZILLA projects. Nevertheless, if trained for a sufficient number of iterations, all the algorithms converge to the same value. We report the figure for the post-decision state of an arbitrary bug type and due date. We note that ADP with BAKF converges slightly faster to the optimal value and outperforms all other approaches during all iteration steps for the ECLIPSEJDT project.

### C. Hyperparameter Tuning

In this section, we investigate the sensitivity of the model to its various hyperparameters. We focus on three hyperparameters: exploration rate, cost function for postponement, and discounting factor of the Bellman equation.

1) **Exploration vs. Exploitation**: Although previous studies indicate a negligible value in exploration with high-dimensional state spaces [25], [32], we employ the $\epsilon$-greedy approach to observe whether it may help our model to obtain a better policy. The $\epsilon$-greedy policy lets the developer decline the assigned bug with probability $\epsilon$ or chooses the optimal action $\alpha$ following the policy $\pi$ with probability $1-\epsilon$. In practice, when a developer is CC’ed for a bug, they may start to fix the bug or discard it. This way, we provide the opportunity for the model to visit some states that normally cannot be visited if pure exploration is taken into account. Table IV shows how the performance of the models changes when we provide a likelihood of 75 percent for the exploration compared to the case with zero exploration. We observe an improvement in the fixing time and the accuracy of assignments for all methods when $\epsilon$-greedy approach is employed. We also investigated the algorithms during the training phase to see whether exploration helps the models decrease their discounted reward. As discussed in Section IV-B, the lower the long-run value for the objective function is, the better the model performance will be. All ADP versions demonstrate a decline (i.e., an improvement) in their discounted rewards when exploration is adopted. Accordingly, we use the $\epsilon$-greedy approach with $\epsilon = 0.75$ as the default parameter of our model in the numerical experiments.

2) **The Cost of Postponement**: We define two different cost functions for postponing the bugs (as shown in Fig. 6), namely Linear and Exponential:

\[ f(b_{\text{due}}, t) = 0.9^{b_{\text{due}}}, \quad (18) \]
\[ f(b_{\text{due}}, t) = \frac{T - b_{\text{due}}}{T}, \quad (19) \]

where $T$ is the project horizon. Using this exponential function (Equation (18)), we ensure that if we have a large enough number of epochs until the due date of a bug, the postponement cost is low. When we reach the due date, the cost becomes 1, and afterwards, it grows exponentially, restricting the model to further delay fixing the bugs whose due dates have passed. Considering the linear function (Equation (19)), the cost would be slightly higher for early assignments and much lower for the late ones compared to the exponential function. This way, we aim to see whether we need to impose on the model a high cost of delayed assignments or whether the model learns it through its value function estimation without such a cost.
Fig. 5. Finite horizon case: resulting realized rewards (top) and estimated $\bar{V}(s_0)$ (bottom) for an arbitrary post-decision state, using $N = 20,000$, $M = 100$, and $O = 30$.

Table IV

| Project | Method | $\epsilon$ | Top-1 | Top-3 | Top-5 Mean Fixing Time | Discounted Reward |
|---------|--------|------------|-------|------|------------------------|-------------------|
| ECLIPSEJDT | ADP | 28.2 | 66.7 | 82.5 | 3.93 | 423.43 |
| | ADP Harm | 28.6 | 66.7 | 82.7 | 3.90 | 417.23 |
| | ADP BAKF | 29.0 | 66.9 | 82.4 | 3.88 | 411.23 |
| | ADP | 28.7 | 65.0 | 80.8 | 3.97 | 423.42 |
| | ADP Harm | 27.8 | 65.7 | 81.8 | 3.96 | 420.90 |
| | ADP BAKF | 27.7 | 65.9 | 81.5 | 3.98 | 423.42 |
| | ADP | 25.6 | 49.7 | 58.0 | 3.67 | 851.20 |
| | ADP Harm | 25.4 | 48.7 | 58.1 | 3.76 | 926.66 |
| | ADP BAKF | 25.2 | 48.1 | 56.5 | 3.83 | 938.65 |
| GCC | ADP | 26.1 | 51.9 | 59.8 | 3.59 | 890.00 |
| | ADP Harm | 25.6 | 49.7 | 58.0 | 3.67 | 851.20 |
| | ADP BAKF | 25.5 | 50.5 | 58.5 | 3.62 | 872.10 |
| | ADP | 25.6 | 48.7 | 56.9 | 3.82 | 957.45 |
| | ADP Harm | 25.4 | 48.5 | 58.1 | 3.76 | 926.66 |
| | ADP BAKF | 25.2 | 48.1 | 56.5 | 3.83 | 938.65 |
| MOZILLA | ADP | 6.8 | 20.1 | 24.5 | 4.85 | 2329.33 |
| | ADP Harm | 6.0 | 19.7 | 24.9 | 4.95 | 2238.80 |
| | ADP BAKF | 6.2 | 19.7 | 24.2 | 4.84 | 2313.04 |
| | ADP | 6.7 | 19.9 | 25.1 | 5.05 | 2281.49 |
| | ADP Harm | 6.3 | 19.2 | 23.1 | 5.48 | 2242.95 |
| | ADP BAKF | 6.4 | 19.3 | 24.0 | 5.31 | 2255.41 |

Table V shows the comparisons with linear and exponential cost functions for the bug postponements. We observe that when we utilize the linear function in the objective function, we have a slight drop in the fixing time and a negligible improvement in the assignment accuracy. Therefore, no matter what cost function is selected, ADPTriage adapts itself to estimate the optimal values of the future rewards. Hence, it is not sensitive to either option. Moreover, there is no difference between the discounted reward of the training phase of the two options. As a result, we use the simpler linear function as the default cost function throughout our experiments.

3) Discount Factor: The discount factor $0 < \gamma < 1$ of the Bellman equation controls how we accumulate contributions over time. For $\gamma = 0$, we only consider the immediate reward and discard the future accumulated rewards. We explore the model sensitivity to two different settings for the discount factor: $\gamma \in [0.9, 0.99]$. Fig. 7 shows how the model learns to accumulate the rewards throughout the iterations. We note that for
TABLE V
COMPARING DIFFERENT COST FUNCTIONS FOR POSTPONEMENTS

| Project | Method | Cost function | Top-1 | Top-3 | Top-5 | Mean Fixing Time | Discounted Reward |
|---------|--------|---------------|------|------|------|-----------------|------------------|
| ECLIPSEJDT | ADP | Linear | 28.2 | 66.7 | 82.5 | 3.93 | 423.43 |
| ADP Harm | Linear | 28.6 | 66.7 | 82.7 | 3.90 | 417.23 |
| ADP BAKF | Linear | 29.0 | 66.9 | 82.4 | 3.88 | 411.23 |
| ADP | Exponential | 28.1 | 66.2 | 81.8 | 3.96 | 423.43 |
| ADP Harm | Exponential | 28.7 | 66.3 | 82.1 | 3.92 | 417.23 |
| ADP BAKF | Exponential | 29.0 | 66.7 | 82.4 | 3.90 | 411.23 |
| GCC | ADP | Linear | 26.1 | 51.9 | 59.8 | 3.59 | 890.00 |
| ADP Harm | Linear | 25.6 | 49.7 | 58.0 | 3.67 | 851.20 |
| ADP BAKF | Linear | 25.5 | 50.5 | 58.5 | 3.62 | 872.10 |
| ADP | Exponential | 25.7 | 50.6 | 59.0 | 3.63 | 890.00 |
| ADP Harm | Exponential | 25.0 | 49.0 | 57.6 | 3.70 | 851.20 |
| ADP BAKF | Exponential | 25.2 | 49.9 | 58.2 | 3.65 | 872.10 |
| MOZILLA | ADP | Linear | 6.8 | 20.1 | 24.5 | 4.85 | 2329.33 |
| ADP Harm | Linear | 6.0 | 19.7 | 24.9 | 4.95 | 2238.80 |
| ADP BAKF | Linear | 6.2 | 19.7 | 24.2 | 4.84 | 2313.04 |
| ADP | Exponential | 6.6 | 19.8 | 24.5 | 4.89 | 2329.33 |
| ADP Harm | Exponential | 6.1 | 19.7 | 24.8 | 4.97 | 2238.80 |
| ADP BAKF | Exponential | 6.2 | 19.8 | 24.4 | 4.87 | 2313.04 |

Fig. 6. Different postponement cost functions.

\( \gamma = 0.9 \), the ADP model underperforms the myopic approach as its discounted rewards are higher than that of the myopic method. As our objective function is a minimization, lower rewards/costs are desired. On the other hand, for \( \gamma = 0.99 \), after a few iterations, the model starts accumulating lower rewards/costs and learns how to estimate the future costs of a state. Therefore, we utilize \( \gamma = 0.99 \) based on its promising performance in this numerical experiments.

V. THREATS TO VALIDITY

In this section, we discuss the threats to validity of our analysis with regards to construct validity, internal validity and external validity.

A. Construct Validity

We use a train-test split to estimate the models’ performance. The first eight years are used as a training period and the latter two years as a test period. However, the results may be influenced by the bug repository’s ever-changing nature. Some developers may become less active over some time, leave the system, or become more focused on a single project component. As a result, the active developer’s definition may need to be revised yearly. We acknowledge that the definition of active developers can be subjective and may vary among different projects. However, in our study, we follow the common practice and definitions (i.e., interquartile range (IQR)) as a measure of central distribution to define active developers, which is a less subjective approach and has been used in previous studies [26]. While there are other approaches, such as setting an arbitrary threshold for active developers (e.g., six or more bug resolutions within six months to be considered active [10] or at least nine bug fixings in the last three months [3]), our method provides an objective way of defining active developers based on the distribution of bug-fix numbers. We acknowledge that some projects may have a pre-defined definition of active developers, and our approach may not be applicable to those projects. In such a case, domain expertise and project knowledge can be used to decide on the number of active developers.

We advocate a rolling strategy for the train-test split to avoid obsolete conclusions in future research. We note that the difference between the bugs’ and developers’ frequencies during the training and testing phases has been already reflected in the result. While we have used LDA topic modelling to define the number of bug types similar to previous studies [9], [10], [11], this approach may not be applicable to all projects, especially those with a limited number of bug types or incomplete textual information, which may require a different approach for bug type definition. Accordingly, we analyze textual information as our independent feature while finding the LDA category of the bugs; nonetheless, other bug characteristics such as components and keywords can be incorporated into the clustering models.

To the best of our knowledge, this is the first research to incorporate uncertainty in bug arrival and developer availability into the modelling of bug triage. As a result, comparing the performance of the ADPTriage against that of prior algorithms might not be feasible. We assume that the developers’ availability follows the same distribution as the one in the training and testing phases. We acknowledge the lack of data relating to the developers’ real schedule when it comes to the accuracy of constructing the modelling process based on the bugs’ history. The same threat exists in estimating the length of time it takes to assign a bug. In practice, that value can be estimated by the average time it takes for developers to assign a bug, which may vary among different projects.
determined according to a bug’s perceived priority and severity. While a developer can work on multiple bugs at once, this may prolong the time it takes for the developer to fix all of them. As a result, since our approach considers bug-fixing costs, assignment due dates, and capacities simultaneously, the problem of a developer’s availability is alleviated by a longer fixing time. These assumptions are flexible and can be replaced with actual values without compromising the validity of our model. On the other hand, assuming fixing one bug at a time does not violate the validity of the results, and it only changes the estimation of bug fixing times in practice. This point is further discussed in a study by [11], which found no significant difference in performance when considering multiple bug-fixing tasks versus disregarding them. Hence, incorporating developer capacity is a possible extension of our current work.

Bug triage is a multifaceted problem that involves many human-specific factors, especially in open-source software systems where the schedules and availabilities of developers are volatile and unpredictable. This makes bug triage even more challenging, and previous bug triage models may not be able to handle the uncertainty in bug arrival time and developer availability. The proposed ADPTriage model addresses this issue by incorporating downstream uncertainty in bug arrivals and developers’ schedules. However, in practice, the developers’ availability may not follow the same distribution as in the training and testing phases, which could affect the performance of the model. To mitigate this issue, we propose using a rolling strategy for the train-test split or reducing the training period to a shorter time frame. These measures may improve the model’s performance in real-world scenarios.

B. Internal Validity

We use the REST API to collect bug information from Bugzilla, including all bug data between January 2010 and December 2019. The API, however, is restricted to regular users, and access to some bugs is not possible. As a result, we extract all accessible bug reports. We ensure that all publicly accessible bugs are included in our work.

We exclude the severity and priority of bugs in our model because they are usually deemed subjective [33], [34], [35]. Our model can indirectly comprise these two factors by defining the due date for bug assignments. Accordingly, bugs with higher severity/priority may be given a shorter due date and fixed earlier in the system. On the other hand, in the event of a developer who is not an expert in a certain topic, we anticipate seeing high fixing costs (i.e., low suitability). As a result, the model may defer bugs according to their due dates until a more suitable developer becomes available in the future decision epochs. Another important consideration is the relationship between severity, priority, and resolution time. Previous works also indicate a shorter fixing time for bugs with higher severity and priority [37], [38]. Therefore, our model implies those factors by determining shorter assignment times for more severe bugs.

C. External Validity

This study examines EclipseJDT, GCC, and Mozilla, selected from the Bugzilla web-based bug-tracking system. These projects were selected based on their popularity in the open-source community, extensive and high-quality data availability, longevity, the variety of software types they represent, and their precedence in prior research [11], [12], [39], [40], [41], [42], [43]. While they are well-studied and long-lived systems, our findings may not necessarily apply to all other open software systems, emphasizing the need for caution in generalizing our results. However, these criteria make our project selection less likely to be biased. To further confirm the validity and
generalizability of our approach, we recommend replication of our work using different datasets that meet similar criteria. Additionally, we employ various performance metrics to accurately evaluate the benefits and drawbacks of our models. To enhance the external validity of the study, future work could explore the performance of ADPTriage using different topic modelling approaches (e.g., Latent Semantic Analysis (LSA), Non-negative Matrix Factorization (NMF), deep-learning approaches, or learning-from-crowd approaches) and investigate whether the results remain consistent.

VI. RELATED WORK

Due to the importance of bug triage in the literature, researchers conducted several related studies during the past decades. They adopted different techniques, e.g., machine learning, graph analysis, fuzzy set-based automatic bug triage, and deep learning techniques. However, the uncertainty in the issue tracking systems is yet to be explored. Specifically, in open-source projects, many freelancer developers, whose availability in the upcoming period is not easy to determine, may attempt to fix bugs. Moreover, the timing of bug reports may not follow a specific distribution, thus not being straightforward to predict the next bug report [44].

Previous studies mainly focused on improving the accuracy of the assignment in a static environment. In other words, knowing the bug attributes and previously assigned developers, how can we propose an appropriate developer for a particular bug. For instance, [45] suggested that using expectation-maximization and a combination of labeled and unlabeled bug reports can improve the performance of the Naive Bayes classifier in the bug assignment task. They used a small sample of labeled bug reports to train a classifier. In another study, [46] explored different traditional machine learning techniques using an industrial dataset and reported the accuracy of the assignments for each algorithm. These traditional methods were further improved using deep learning approaches [6], [47], whether adopting the same textual information or incorporating team labels or other attributes. These approaches are mainly prone to overspecialization, i.e., assigning numerous bugs to expert developers [10]. On the other hand, they do not consider the capacity and schedule of each developer while triaging the bugs. The dynamic nature of the ITS and the rate of incoming bug reports require defining an environment in which these features are captured.

[10] first proposed a simulation environment within which they can implement their release-aware bug triage method. That way, they guarantee that their method is applied in a system similar to the actual one. Moreover, [7] introduced their tool, called Wayback Machine, to reconstruct the actual bug arrival times and the historical decisions made in the ITS. Although both built a simulated environment to examine their algorithms, none considered the uncertainty in bug arrival times and developers’ availabilities. [48] emphasized the importance of developers’ activities in the bug triage process, i.e., whether a developer remains active in certain bug types during different project phases. They combined these developers’ engagements with the textual information of the bugs fed into a Convolutional Neural Network to enhance the bug assignment. However, the rate at which a bug type is reported to the ITS and the developers’ schedules are not included in their model. One of the sources of uncertainty, developers’ availability, is first considered by [11]. They proposed an IP solution for bug triage, incorporating bug dependencies, developers’ schedules, and bug fixing costs. Nevertheless, the IP model lacks getting updated over time and does not consider the future cost of each bug assignment. These semi-online methods made through simulation are not able to capture the uncertainty in the ITS.

In a more recent work by [49], for the first time, Reinforcement Learning (RL) is leveraged to propose an online solution for the bug triage problems. Based on the textual information of the bug reports, their method assigns them to developers with a likelihood of an “action” in RL. Accordingly, the feature information of bug reports is considered the “state” in (RL). The probability of a developer being chosen to fix a bug can be determined by analyzing the multidimensional properties; hence, the developer with the highest probability value can be identified. That is, the “reward” is defined based on the feedback of the assignments. To achieve the desired result, the model self-trains and adapts to the relationship between bug reports and developers. In their definition of the problem, exogenous information such as bug arrival or developer availability is disregarded. Therefore, the long-term reward may not represent the actual value of the current assignment. On the other hand, they do not consider the developers’ schedules. Overlooking such a constraint may lead to overspecialization. Hence, our model differs from the aforementioned studies in that it utilizes Approximate Dynamic Programming to comprise exogenous information in the ITS, uncertainty in this ever-evolving system, high-dimensional state-action pairs, and constraints on developers’ burdens. Our online ADPTriage method contributes to the literature in terms of novel bug triage formulation and improved assignment accuracy.

VII. DISCUSSIONS AND CONCLUDING REMARKS

In this article, we develop an online solution to bug triage using ADP, which takes into account uncertain bug arrivals and active developers of an open-source ITS. We note that there are many sources of uncertainty in an ITS. First of all, the number of bugs reported to the system does not follow a particular pattern that can be easily predicted. For example, in a real-world scenario, a severe bug may be reported unexpectedly, which requires immediate attention from triagers and developers, affecting the scheduling and assignment of other bugs in the system. This uncertainty can lead to delays in the bug triage process and impact the overall efficiency of the software development project. In addition, developers’ schedules and availability can be considered as another source of uncertainty, which may constantly get updated in open-source software systems. For instance, a developer may have planned to work on a particular bug the next day, but due to some unforeseen circumstances, they may not be able to work on it as planned. This could lead to delays in fixing the bug and result in additional time.
and resources being spent on it. Such uncertainties make it challenging to accurately plan and prioritize the assignment of bugs in the ITS.

We formulate an MDP model for the bug triage problem and present an ADP-based solution approach, called ADPTriage, based on the value function approximation of bugs and developers to incorporate the downstream uncertainty in the bug arrivals and developers’ schedules. In addition, we develop an optimization-based myopic strategy. While traditional machine learning-based methods for bug triage are also able to do real-time decision-making, they do not consider the always-evolving nature of the bug repository and the many possible constraints on the bug triage decisions. Unlike these models, our proposed ADPTriage method incorporates the downstream uncertainty in bug arrivals and developers’ timetables, which enables real-time decision-making while accounting for the changing environment. Moreover, despite the complexity of this ADP algorithm, once trained, our method can assign a bug to developers quickly in practice, taking less than a second to compute the optimal decision and so having a negligible inference time. In addition, we consider various enhancements to the ADP algorithm. To the best of our knowledge, this is the first time an ADP-based solution approach developed for the bug triage problem.

In our empirical analysis, we utilize bug reports of the ECLIPSEJDT, GCC, and MOZILLA projects between 2010 and 2020, which are the first years when they are taken as the training set, and the remaining ones are as the test set. Using elaborate modeling of the environment, we demonstrate the efficiency and effectiveness of the proposed ADP framework and the myopic strategy. We conduct a sensitivity analysis on model parameters and find that the ADP policy outperforms the myopic policy for online bug triage under various parameter configurations. We also demonstrate that ADPTriage makes intelligent decisions based on the system’s characteristics and expected future return.

This study serves as a prelude to various new research avenues in online bug triage decision-making. First, certain components of the ADPTriage can be modified to achieve better bug triage performance. For instance, LDA for bug type categorization can be replaced with alternative methods (e.g., other topic modeling techniques or deep learning-based bug type classification models) which can help achieve a more accurate parameter estimation procedure to obtain parameters that closely reflect the practice. Second, our work can be extended to proprietary software systems in which the developers’ schedules are more stable, and the chance of improvement over the myopic approach is higher because of such patterns. Third, the system state can be expanded to incorporate developers’ ability to address simultaneous bug-fixing tasks [11]. Since such a modification significantly increases the system state dimensionality, more advanced solution algorithms (e.g., Neural ADP) can be employed to overcome this issue [22]. Lastly, a deeper analysis of the collected data needs to be conducted, quantifying the influence of uncertainties such as the emergence of unexpected severe bugs or fluctuations in developer availability on bug-fixing times. This analysis is expected to further corroborate the evidence demonstrating the superiority of the proposed approach over the myopic method in real-world scenarios. Potential challenges associated with data availability and granularity are anticipated, and strategies to overcome these challenges should be sought in the course of this future work.

REFERENCES

[1] T. W. W. Aung, Y. Wan, H. Huo, and Y. Sui, “Multi-tri age: A multi-task learning framework for bug triage,” J. Syst. Softw., vol. 184, no. C, Feb. 2022, Art. no. 111133.
[2] S. Banerjee, Z. Syed, J. Helmick, M. Culp, K. Ryan, and B. Cukic, “Automated triaging of very large bug repositories,” Inf. Softw. Technol., vol. 139, pp. 1–13, Sep. 2017.
[3] A. Anvik, L. Hiew, and G. C. Murphy, “Who should fix this bug?,” in Proc. 28th Int. Conf. Softw. Eng., New York, NY, USA: ACM, 2006, pp. 361–370.
[4] S.-R. Lee, M.-J. Heo, C.-G. Lee, M. Kim, and G. Jeong, “Applying deep learning based automatic bug triager to industrial projects,” in Proc. 11th Intl. Meeting Found. Softw. Eng., New York, NY, USA: ACM, 2017, pp. 926–931.
[5] S. Xi, Y. Yao, X. Xiao, F. Xu, and J. Lu, “An effective approach for routing the bug reports to the right fixers,” in Proc. 10th Asia-Pacific Symp. Internetware, New York, NY, USA: ACM, 2018, pp. 1–10.
[6] S. Mani, A. Sankaran, and R. Aralikatte, “DeepTriage: Exploring the effectiveness of deep learning for bug triaging,” in Proc. ACM India Joint Int. Conf. Data Sci. Manage. Data, New York, NY, USA: ACM, 2019, pp. 171–179.
[7] H. Jahanshahi, M. Cevik, J. Navas-Suárez, A. Başar, and A. González-Torres, “Wayback machine: A tool to capture the evolutionary behavior of the bug reports and their triage process in open-source software systems,” J. Syst. Softw., vol. 189, no. C, Jul. 2022, Art. no. 111308.
[8] S. F. A. Zaidi, H. Woo, and C.-G. Lee, “A graph convolution network-based bug triage system to learn heterogeneous graph representation of bug reports,” IEEE Access, vol. 10, pp. 20677–20689, Feb. 2022.
[9] J.-w. Park, M.-W. Lee, J. Kim, S.-w. Hwang, and S. Kim, “CosTriage: A cost-aware triage algorithm for bug reporting systems,” in Proc. AAAI Conf. Artif. Intell., Aug. 2011, vol. 25, no. 1, pp. 139–144.
[10] Y. Kashivah and M. Ohira, “A release-aware bug triaging method considering developers’ bug-fixing loads,” IEICE Trans. Inf. Syst., vol. 103, no. 2, pp. 348–362, 2020.
[11] H. Jahanshahi and M. Cevik, “S-DABT: Schedule and dependency-aware bug triage in open-source bug tracking systems,” Inf. Softw. Technol., vol. 151, Nov. 2022, Art. no. 107025.
[12] A. Yadav, M. Baljon, S. Mishra, S. K. Singh, S. Saxena, and S. K. Sharma, “Developer load balancing bug triage: Developed load balance,” Expert Syst., May 2022, Art. no. e13006, doi: 10.1111/exsy.13006.
[13] J. Wang and H. Zhang, “Predicting defect numbers based on defect state transition models,” in Proc. ACM-IEEE Int. Symp. Empirical Softw. Eng. Meas., 2012, pp. 191–200.
[14] D. M. Blei, A. Y. Ng, and M. I. Jordan, “Latent Dirichlet allocation,” J. Mach. Learn. Res., vol. 3, pp. 993–1022, Jan. 2003.
[15] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshypany, and A. De Lucia, “How to effectively use topic models for software engineering tasks? An approach based on genetic algorithms,” in Proc. 35th Int. Conf. Softw. Eng. (ICSE), 2013, pp. 522–531, doi: 10.1109/ICSE.2013.6606598.
[16] G. Catifrá, F. Palomba, A. Zaidman, and F. Ferrucci, “Not all bugs are the same: Understanding, characterizing, and classifying bug types,” J. Syst. Softw., vol. 152, pp. 165–181, Jun. 2019, doi: 10.1016/j.jss.2019.03.002.
[17] H. Cho, S. Lee, and S. Kang, “Classifying issue reports according to feature descriptions in a user manual based on a deep learning model,” Inf. Softw. Technol., vol. 142, Feb. 2022, Art. no. 106743, doi: 10.1016/j.infsof.2021.106743.
[18] R. Arun, V. Suresh, C. Veni Madhavan, and N. Murthy, “On finding the natural number of topics with latent Dirichlet allocation: Some observations,” in Proc. Pacific-Asia Conf. Knowl. Discovery Data Mining, Hefei, China: Springer, 2010, pp. 391–402.
[19] M. Castelluccio, L. An, and F. Khomh, “An empirical study of patch uplift in rapid release development pipelines,” Empirical Softw. Eng., vol. 24, pp. 3008–3044, Oct. 2019.
[20] J. Hernández-González, D. Rodriguez, I. Inza, R. Harrison, and J. A. Lozano, “Learning to classify software defects from crowds: A
novel approach,” *Appl. Soft Comput.*, vol. 62, pp. 579–591, Jan. 2018, doi: 10.1016/j.asoc.2017.10.047.

[21] W. B. Powell, *Reinforcement Learning and Stochastic Optimization: A Unified Framework for Sequential Decisions*, vol. 1. Hoboken, NJ, USA: Wiley, 2022.

[22] S. Shah, M. Lowalekar, and P. Varakantham, “Neural approximate dynamic programming for on-demand ride-pooling,” in *Proc. AAAI Conf. Artif. Intell.*, 2020, vol. 34, no. 1, pp. 507–515.

[23] A. E. Bradshaw, “United States Air Force officer manpower planning problem via approximate dynamic programming,” *Air Force Institute of Technology, Wright-Patterson Air Force Base, Ohio, USA, Tech. Rep.*, 2016. [Online]. Available: https://scholar.ati.edu/etd/355/

[24] A. Erdelyi and H. Topaloglu, “Approximate dynamic programming for dynamic capacity allocation with multiple priority levels,” *IIE Trans.*, vol. 43, no. 2, pp. 129–140, 2010.

[25] W. B. Powell, *Approximate Dynamic Programming: Solving the Curses of Dimensionality*, vol. 703. Hoboken, NJ, USA: Wiley, 2011.

[26] J.-w. Park, M.-W. Lee, J. Kim, S.-w. Hwang, and S. Kim, “Cost-aware triage ranking algorithms for bug reporting systems,” *Knowl. Inf. Syst.*, vol. 48, no. 3, pp. 679–705, 2016.

[27] J. Liu et al., “A multi-source approach for bug triage,” *Int. J. Softw. Eng. Knowl. Eng.*, vol. 26, no. 0910, pp. 1593–1604, 2016.

[28] L. A. Wolsey and G. L. Nemhauser, *Integer and Combinatorial Optimization*, vol. 55. Hoboken, NJ, USA: Wiley, 1999.

[29] M. R. Mes and A. P. Rivera, “Approximate dynamic programming by practical examples,” in *Markov Decision Processes in Practice*, Springer, 2017, pp. 63–101. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-47766-43

[30] S. Garcia, A. Fernández, J. Luengo, and F. Herrera, “Advanced nonparametric tests for multiple comparisons in the design of experiments in computational intelligence and data mining: Experimental analysis of power,” *Inf. Sci.*, vol. 180, no. 10, pp. 2044–2064, 2010.

[31] J. Demsar, “Statistical comparisons of classifiers over multiple data sets,” *J. Mach. Learn. Res.*, vol. 7, pp. 1–30, 2006.

[32] K. Mousavi, “Approximate dynamic programming for crowd-shopping with in-store customers,” Ph.D. dissertation, Univ. of Toronto, Toronto, ON, Canada, 2022.

[33] S. Akbarinasaji, C. Kavaklioglu, A. Başar, and A. Neal, “Partially observable markov decision process to generate policies in software defect management,” *J. Syst. Softw.*, vol. 163, May 2020, Art. no. 110518.

[34] V. Gupta, D. Kumar, and P. K. Kapur, “Optimizing the defect prioritization in enterprise application integration,” in *Software Engineering*, M. N. Hoda, N. Chauhan, S. M. K. Quadril, and P. R. Srivastava, Eds., Singapore: Springer Singapore, 2019, pp. 585–597.

[35] Y. Tian, D. Lo, X. Xia, and C. Sun, “Automated prediction of bug report priority using multi-factor analysis,” *Empirical Softw. Eng.*, vol. 20, no. 5, pp. 1354–1383, 2015.

[36] G. Yang, S. Baek, J.-W. Lee, and B. Lee, “Analyzing emotion words to predict severity of software bugs: A case study of project topics,” in *Proc. Symp. Appl. Comput.*, New York, NY, USA: ACM, 2017, pp. 1280–1287.

[37] R. Sepahvand, R. Akbari, and S. Hashemi, “Predicting the bug fixing time using word embedding and deep long short term memories,” *IET Softw.*, vol. 14, no. 3, pp. 203–212, 2020.

[38] H. Zhang, L. Gong, and S. Versteeg, “Predicting bug-fixing time: An empirical study of commercial software projects,” in *Proc. 35th Int. Conf. Softw. Eng. (ICSE)*, San Francisco, CA, USA: IEEE, 2013, pp. 1042–1051.

[39] T. Zhang, J. Chen, G. Yang, B. Lee, and X. Luo, “Towards more accurate severity prediction and fixer recommendation of software bugs,” *J. Syst. Softw.*, vol. 117, pp. 166–184, Jul. 2016, doi: 10.1016/j.jsyssoft.2016.02.034.

[40] X. Xia, D. Lo, Y. Ding, J. M. Al-Kofahi, T. N. Nguyen, and X. Wang, “Improving automated bug triaging with specialized topic model,” *IEEE Trans. Softw. Eng.*, vol. 43, no. 3, pp. 272–297, Mar. 2017, doi: 10.1109/TSE.2016.2576454.

[41] A. Yadav, S. K. Singh, and J. S. Suri, “Ranking of software developers based on expertise score for bug triaging,” *Inf. Softw. Technol.*, vol. 112, pp. 1–17, Aug. 2019, doi: 10.1016/j.infsof.2019.03.014. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584919300709

[42] Y. Tan, S. Xu, Z. Wang, T. Zhang, Z. Xu, and X. Luo, “Bug severity prediction using question-and-answer pairs from stack overflow,” *J. Syst. Softw.*, vol. 165, Jul. 2020, Art. no. 110567, doi: 10.1016/j.jss.2020.110567. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121220300480

[43] N. K. Nagwani and J. S. Suri, “An artificial intelligence framework on software bug triaging, technological evolution, and future challenges: A review,” *Int. J. Inf. Manage. Data Insights*, vol. 3, no. 1, 2023, Art. no. 100153, doi: 10.1016/j.jimide.2022.100153.

[44] H. Jahanshahi, M. Cevik, and A. Başar, “Predicting the number of reported bugs in a software repository,” in *Proc. Adv. Artif. Intell.*, C. Goutte and X. Zhu, Eds., Cham, Switzerland: Springer International Publishing, 2020, pp. 309–320.

[45] J. Xuan, H. Jiang, Z. Ren, J. Yan, and Z. Luo, “Automatic bug triage using semi-supervised text classification,” 2017, arXiv:1704.04769. [Online]. Available: https://arxiv.org/abs/1704.04769

[46] P. Oliveira, R. M. C. Andrade, I. Barreto, T. P. Nogueira, and L. Morais Bueno, “Issue auto-assignment in software projects with machine learning techniques,” in *Proc. IEEE/ACM 8th Int. Workshop Softw. Eng. Res. Ind. Pract.*, 2021, pp. 65–72.

[47] C. A. Choquette-Choo, D. Sheldon, J. Proppre, J. Alphonso-Gibbs, and C. A. Choquette-Choo, D. Sheldon, J. Proppre, J. Alphonso-Gibbs, and H. Gupta, “A multi-label, dual-output deep neural network for automated bug triaging,” in *Proc. 18th IEEE Int. Conf. Mach. Learn. Appl. (ICMLA)*, 2019, pp. 937–944.

[48] S. Guo et al., “Developer activity motivated bug triaging: Via convolutional neural network,” *Neural Process. Lett.*, vol. 51, no. 3, pp. 2589–2606, 2020.

[49] Y. Liu, X. Qi, J. Zhang, H. Li, X. Ge, and J. Ai, “Automatic bug triaging via deep reinforcement learning,” *Appl. Sci.*, vol. 12, no. 7, pp. 1–19, 2022.