The Assessor's Dilemma: Improving Bug Repair via Empirical Game Theory

Carlos Gavidia-Calderon, Federica Sarro, Mark Harman, and Earl T. Barr

Abstract—Priority inflation occurs when a QA engineer or a project manager requesting a feature inflates the priority of their task so that developers deliver the fix or the new functionality faster. We survey developers and show that priority inflation occurs and misallocates developer time. We are the first to apply empirical game-theoretic analysis (EGTA) to a software engineering problem, specifically priority inflation. First, we extract prioritization strategies from 42,620 issues from Apache's JIRA, then use TASKASSESSOR, our EGTA-based modelling approach, to confirm conventional wisdom and show that the common process of a QA engineer assigning priority labels is susceptible to priority inflation. We then show that the common mitigation strategy of having a bug triage team assigning priorities does not resolve priority inflation and slows development. We then use mechanism design to devise *assessor-throttling*, a new, lightweight prioritization process, immune to priority inflation. We show that assessor-throttling resolves 97% of high priority tasks, 69% better than simply relying on those filing tasks to assign their priorities. Finally, we present TheFed, a browser extension for Chrome that supports assessor-throttling.

Index Terms—Software Process, Game Theory, Bug Report, Priority Inflation

1 INTRODUCTION

Lack of time is actually lack of priorities.
—Timothy Ferriss

PEOPLE often break projects into tasks, then prioritise them. More important tasks, because they produce something of value or because other tasks depend on them, have higher priority. In an issue tracking system with shared prioritisation tooling, QA engineers, testers, or project managers assign a priority label to tasks; this label informs a project team about the fixes or features the next release should incorporate. *Priority inflation* occurs when an assessor increases the priority of an issue above their true assessment, so that tasks they care about are delivered more quickly [1], [2]. By undermining priority labels, priority inflation can misallocate developer time.

We contend that priority inflation hampers software development for three reasons: 1) Despite the fact that most teams would prefer to work on important, unclaimed tasks first, we found that teams using GitHub tend not to use its shared prioritisation tooling when its use is optional (Section 3.1); 2) We surveyed software development professionals who reported that priority inflation is frequent and significantly misallocates resources (Section 3.2); and 3) Industry leaders have deployed processes to triage bug reports and correct inflated priorities [3], [4].

In this paper, we use game theory to understand and fix priority inflation, of both new features and bug repair tasks. Game theory studies mathematical models of conflict and cooperation [5]. It has already been used in several disciplines, including artificial intelligence, e-commerce, and networking, because it provides a theoretical framework for interactions between entities with different and even conflicting interests [6].

• *C. Gavidia-Calderon, F. Sarro, M. Harman and E.T. Barr are with the Department of Computer Science, University College London, Gower Street, London WC2R 2LS, London, United Kingdom.*
E-mail: : {carlos.gavidia.15, f.sarro, mark.harman, e.barr}@ucl.ac.uk

Manuscript received April 19, 2005; revised August 26, 2015.

Standard game-theoretic models of real-world scenarios become intractable when dealing with many players or numerous strategies [7]. Having a dataset with hundreds of bug reporters and multiple strategies represents a challenge for classic game-theoretic approaches. To overcome this, we turn to an empirical game-theoretic analysis (EGTA), which combines empirically grounded game reduction with simulation to scale to real-world scenarios [7]. TASKASSESSOR realises our EGTA-based analysis of priority inflation; it is the first tool to apply EGTA to a software engineering problem.

TASKASSESSOR simulates a process with prioritised tasks. Given an EGTA model of that process, it computes the model's *Nash equilibrium* to diagnose problems. The Nash equilibrium is a stable outcome reached by rational and self-interested participants (players, in game theory). Its stability comes from the fact that each player enacts the strategy that benefits them the most: Any deviation from the equilibrium strategy decreases a player's pay-off. A task prioritisation process immune to priority inflation would produce a model with a single equilibrium where all the players — people filing tasks or reporting bugs — adopt a strategy under which players honestly prioritise tasks.

We first use TASKASSESSOR to model *distributed prioritisation*, which distributes prioritisation to the person filing a task or bug report [8]. To build our model, we use 42,620 issues collected from the JIRA issue tracker of the Apache Software Foundation. From them, we extract prioritisation strategies. A surprising finding here is the prevalence of priority deflation: 40% of the time, issue reporters deflate the priority label (Section 5.3). JIRA priority labels combine technical severity and business value or risk of an issue, end users often confuse and conflate the two [9]. Our focus on priority inflation means that we are, in particular, concerned with the proportion of resolved tasks that are, in fact, high priority. This is relevant to software development teams that seek to maximise the number of tasks resolved. So, we validate that the simulation component of TASKASSESSOR is sufficiently accurate with respect to the proportion of high priority tasks completed (Section 5.4).

After validating the model, we compute its Nash equilibrium and find that the equilibrium shows that the "Always Inflate" strategy is optimal (Section 6.1). In this way, we have used game theory to corroborate the conventional wisdom that distributed prioritisation is prone to priority inflation.

To combat priority inflation, development teams have incorporated *bug triage* into their prioritisation processes [1]. In this process, a team of *gatekeepers*, typically distinct from those who file or report issues, checks (and may reprioritise) each issue. Gatekeepers can be technical or business-focused employees. In Section 6.2, we show that gatekeeper processes reprise distributed prioritisation in one of two ways, which implies that they are also susceptible to priority inflation. We confirm this using TASKASSESSOR and find that, at equilibrium, task filers and QA engineers still have an incentive to inflate priorities in a gatekeeper process. We use a Jackson network, from queueing theory [10], to show that gatekeeping slows development, even in the presence of duplicate tasks or bug reports. Thus, while gatekeepers can improve the prioritisation of issues in terms of better matching priorities with their business value, we have used game theory to contradict the conventional wisdom showing that the gatekeeper process does not mitigate priority inflation and slows development.

Our game theoretic analysis, in short, shows that the current state of practice does *not* fix priority inflation and explains the extraordinarily low use of priority mechanisms in GitHub. To fix priority inflation, we turn to mechanism design, the branch of game theory concerned with designing games whose equilibrium strategies constrain players to behave in desirable ways. Using TASKASSESSOR to evaluate process interventions (or mechanisms), we devised a novel lightweight, prioritisation mechanism, which we call *assessor-throttling*, to tackle priority inflation (Section 7). When they have completed a task of fixed a bug, developers have also assessed its Jira priority along the way: they know its technical severity and often have acquired the expertise needed to evaluate its business value [11]. When they finish a task, assessor-throttling merely requires developers to record their assessment of the task's priority label. If the developer's assessment differs from the task's priority label, the offending reporter's reputation drops, which restricts the reporter's ability to submit tasks or bugs. Via simulation, we show that assessor-throttling matches an ideal gatekeeper in the completion of high priority tasks (Section 7.3). To help developers transition to assessor-throttling and thereby combat priority inflation, we realised it in TheFed, a browser plugin for Chrome (Section 7.5) that individual developers can easily download and install from our project page [12].

Our main contributions follow:

1) We are the first to use empirical game-theoretic analysis to analyse and improve an important software processes — bug repair and issue resolution (Section 5).
2) We show that processes in which task filers or QA engineers prioritise issues are inflationary (Section 6.1) and the common solution of interposing a gatekeeper — as advocated by some agile development models — does not prevent inflation and, in fact, reduces productivity (Section 6.2).
3) We propose assessor-throttling, a novel and lightweight task prioritisation process that is immune to priority inflation (Section 7).

Game theory has great potential to improve software processes beyond bug repair and issue resolution. Please join us to explore the possibilities at http://ttendency.cs.ucl.ac.uk/gametheory4se.

2 THE ASSESSOR'S DILEMMA

We now showcase game-theoretic modelling in a software development context and use it to illustrate priority inflation at Foo Inc, a small or medium-sized enterprise (SME).

Economic models represent behaviours where human motivation can be expressed as a function of price [13]. Neoclassical economists like Alfred Marshall also believe that money is also a suitable measure for intangibles like desires and aspirations. The magnitude of a person's preference towards a product or service can be approximated as the amount of money this person is willing to pay for it: this applies to both smartphones and to political platforms. The ability to approximate motivations — although imperfectly — with real numbers is what makes economics "the most exact of social sciences" [13]. In particular, game-theoretic models require expressing a player's payoffs as real numbers [14]. In our example below, we meet this requirement by assuming Foo Inc has a bonus policy tied to bug fixing measures, like the companies studied by Laplante and Ahmad [15].

Foo Inc. uses an Enterprise Resource Planning (ERP) system for its daily activities. Alice and Bob work in the Foo Inc's quality assurance team. They report bugs to a development team that cannot fix all known bugs, so Alice and Bob are competing for development time. Foo Inc wants its developers to fix more important bugs first, so it rewards QA engineers who report higher priority bugs that the developer team fixes with a higher bonus [15].

Foo Inc's finance manager tells Alice that the ERP system has two problems — the cash management module produces incorrect figures and the financial consolidation module is too slow. The first problem is severe and costs the company $10,000/day; the second is inconvenient, costing only $1,000/day. These figures are arbitrary, but consistent with the cost these bugs might impose on an SME, like Foo Inc. Crucially, we picked them to separate a severe bug from a trivial one by an order-of-magnitude. At the same time, Foo Inc's Human Resources Manager informs Bob that the payroll module crashes every day and that the learning module misplaces images when accessed from mobile devices. The payroll bug is high priority, costing $10,000/day; while the learning module bug is minor, costing only $1,000/day since Foo Inc does not yet widely use mobiles.

At Foo Inc, a fixed, high priority bug increases the reporting QA engineer's bonus by $100, while the resolution of a trivial bug increases it $50. We assume that Foo Inc has found that these values are sufficient incentive and within the bonus it is willing to pay. For the next release, the development team can only fix three of the four bugs that Alice and Bob report. Thus, they have the both the means and the motivation to inflate their bugs' priorities. Table 1 shows Alice's and Bob's expected bonuses, assuming that developers resolve higher priority bugs first and that bugs with the same priority have the same probability of being fixed. For example, when both Alice and Bob inflate their priorities, all four bugs are labelled high priority and have the same probability of getting fixed. In this case, Alice's and Bob's expected pay-off is

$$\frac{1}{2}\left(\frac{1}{2}\times\$100+\frac{1}{2}\times\$50\right)+\frac{1}{2}(\$100+\$50)=\$112.5.$$

Let us analyse Table 1 from Alice's perspective. If Bob accurately prioritise his bugs, Alice's best option is to inflate hers since she would obtain $150 instead of the $125 she would receive if she too were honest. If, instead, Bob inflates his bugs' priorities,

TABLE 1
Pay-off matrix for the assessor's dilemma: Each cell is the payoff Alice (A) and Bob (B) obtain, under the combination of actions each takes.

	Bob: accurate	*Bob: inflate*
Alice: accurate	$A = 125, B = 125$	$A = 100, B = 150$
Alice: inflate	$A = 150, B = 100$	$A = 112.5, B = 112.5$

Alice's best option remains inflating, since her bonus would be \$112.5 *vs.* \$100. Thus, Alice is better off inflating no matter what Bob does. This same analysis symmetrically holds for Bob. In game theory, this outcome is a *Nash equilibrium* of the game: no players have an incentive to change their actions in response to any other player's actions. Every game with a finite number of players and action profiles has at least one Nash equilibrium [16]. If players are rational and understand the game, we expect that repeated play will reach a Nash equilibrium. Empirical evidence in professional sports suggests this happens in practice [17], [18].

This Nash equilibrium is bad for Foo Inc: it represents a bug repair process that encourages testers to inflate priorities and misallocate developer time. Specifically, the Nash equilibrium entails 0.5 probability that one of the high priority bugs is not fixed in the next release. In monetary terms, the equilibrium scenario reduces costs only \$16,500/day, not the \$21,000/day that could have been achieved.

Also, Table 1 shows that, if Bob and Alice honestly reported priorities, they would be better off than if they took the actions leading to the equilibrium — \$125 *vs.* \$112.5. Rational play, however, dictates priority inflation; hence, Alice and Bob face a *dilemma*. We call this game the *Assessor's Dilemma*, since it is an instance of the *Prisoner's Dilemma* [19].

Fortunately, we can rely on game theory, not only to identify the assessor's dilemma, but to correct it. *Mechanism design* is the branch of game theory that designs games in a way that the behaviour of agents at equilibrium results in a specific output [20]. In Section 6, we demonstrate that current prioritisation processes suffer from priority inflation. In Section 7, we use mechanism design to devise a game, representing a task prioritisation process, that is immune to priority inflation.

3 Is Priority Inflation Real?

Prioritisation is challenging and important, since it drives how time, money and energy are spent [21]. Thus, we take for granted that prioritisation is essential for efficient and effective bug repair. Without prioritisation, one tends to fall into the trap of neglecting important tasks for the merely urgent[1]. Task prioritisation is also at the core of agile software development [23]. For example, Scrum development starts with a prioritised list of tasks created by the project sponsor, called the product backlog. In Extreme Programming, low-priority tasks — called slack — are included in each iteration to be discarded first in case of unexpected delays.

Shared prioritization tooling (SPT) is a means for a team to share their assessments of task priority. The *shared* dimension of SPT requires the assessment to be public among team members, so they can both avoid overlapping and prioritise their work. By *tooling* we want to include only software solutions into this category. Mental prioritisation and pen-and-paper mechanisms do not constitute SPT solutions. Bug tracking systems — like Bugzilla and JIRA — have an SPT as part of their functionality. The SPT on bug tracking systems requires the inclusion of a measure of the importance of the bug filed. Bug importance has two dimensions: impact on the system functionality — called *severity* — and impact on the system value — called *priority* [24]. For example, a web application that crashes on Internet Explorer 5.0 has a high severity since functionality is lost, but low priority if the user base of such browser is minimal.

3.1 Shared Prioritization Tooling Adoption

SPT exploits collective intelligence to assess and focus work. To find out how widely SPT is used, we ask:

RQ1: *Do development teams adopt shared prioritisation tooling?*

The GitHub platform offers issue tracking functionality for the software projects, but unlike other issue trackers like JIRA, it does not assign priority labels to issues by default. Instead, GitHub offers a generic labelling system, that developers can use to "signify priority, category, or any other information you find useful" [25].

Thus, to answer RQ1, we performed an exploratory study over GitHub repositories. We collected GitHub projects and counted how many use GitHub's labelling system as SPT. To determine whether a project is using labels as SPT, we applied two heuristics to its label's text and colour: a project uses SPT 1) if the tokenisation and stemming of its label text snippets intersects a bag of priority related words or 2) if its color scheme suggests a priority ranking. For #1, we took the list of priority-related words from the field names and default priority rankings used by JIRA v6.3 (Section 5.1), JIRA v6.4 [26] and Bugzilla [27] . For #2, we used the semaphore colours (red, yellow and green) to identify repositories that colour-encode priorities, as suggested by industry practitioners [28] [29]. We evaluated our heuristics by applying them to 60 GitHub repositories, sampled uniformly. One of us manually assessed these project's use of SPT and found that our heuristics have an F_1 score of 0.8 for repositories using labels as SPT.

We applied these heuristics to the labels we extracted from the 600 most forked repositories created between January 2017 and April 2018. The GitHub development model requires contributors to first create a copy of the repository via a fork to then submit code contributions using pull requests [30]. The number of forks is a good indicator of project activity [31], as is highly correlated with number of contributors, number of commits, and number of branches [32]. We conservatively considered the most forked repositories are more likely to use SPT, as they are more likely to have active teams that would benefit from the coordination that SPT affords. Our finding is that developers on GitHub, a pre-eminent developer collaboration site, rarely use its prioritisation facilities:

> **Finding 1:** Only 6.3% of 60 uniformly sampled GitHub projects adopt shared prioritisation tooling.

To the extent to which GitHub generalises, development teams rarely use SPT when its use is optional. We argue that this is *not* evidence that shared prioritisation is unneeded, but rather evidence that existing SPT is not fit for purpose. In the next section, we describe a survey of developers who use SPT. The key finding is that priority inflation is, indeed, a problem, which may explain our initial finding that developers do not adopt SPT when given the option.

1. In a speech in 1954, Dwight D. Eisenhower said "I have two kinds of problems, the urgent and the important. The urgent are not important, and the important are never urgent.". This quote is the basis of the Eisenhower matrix, to which we refer here [22].

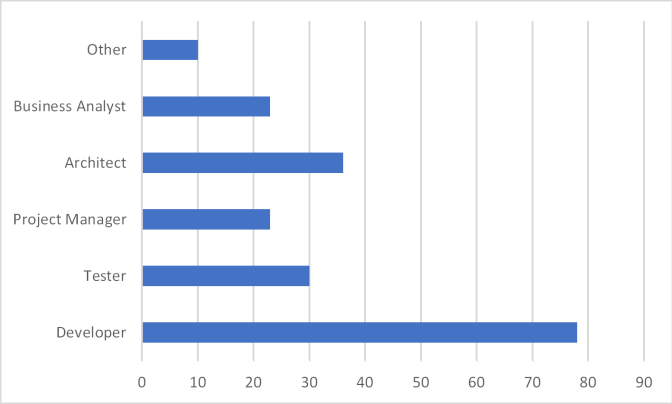

Fig. 1. Role in the development process: The horizontal axis represents the number of participants per role. The survey allowed the selection of multiple roles per participant. This figure represents the answers of 152 software professionals.

Threats to Validity: Our study faces the standard external validity threat: it generalises only to the extent GitHub does. We uniformly sampled the most active GitHub repositories to mitigate this threat. Since we rely on the number of forks as a proxy measure for projects more likely to use prioritisation, we also face a construct validity threat. This threat is mitigated by empirical studies that show the number of forks is correlated with project activity [31] [30] .

3.2 The Cost of Priority Inflation

Finding 1 showed developers generally tend to not to use an SPT when its adoption is optional. Here, we investigate whether priority inflation is the reason using a survey. We solicited survey participants from Apache Software Foundation contributors (Section 5.1) and software engineers in the authors' social networks, both in English and Spanish. 39 Apache contributors took our survey as did 113 software engineers from our social network. Convenience sampling[2] is appropriate given the exploratory nature of this study [33].

Figure 1 reports the roles covered in the software development process by the survey respondents in their organizations. The distribution of data in Figure 1 indicates that the sample is diverse, with an emphasis on the developer role. Since we did not know how many roles a respondent might play, we allowed them to select more than one role. Developer-architects are the largest group with 13 participants, followed by only-testers with 9 participants. Only 3 participants reported performing all the 5 roles included in the survey. We posted complete survey responses at our project page [12].

We surveyed using the questionnaire in Table 2. While question P6 is an open-ended question, the rest of the questionnaire is multiple-choice. The questions fall into three groups by role: bug reporting, fixing, and prioritisation. We instructed participants to only answer questions pertaining to roles they actually perform.

An indicator that priority inflation may have occurred is that the priority value filled by the original reporter was later corrected by another member of the team, like a software developer or a business analyst. We formulated the survey questions about bug reporting to investigate this behaviour. In the limit, as priority inflation becomes the rule, the priority field of issues becomes irrelevant, since developers will learn to ignore it. The survey's bug-fixing questions seek to elucidate how relevant is the priority field information for bug fixers, when compared with other fields included in the bug report. We asked survey participants about the usefulness of a list of bug report fields, including "steps to reproduce", "attached screenshot" and, of course, "priority". Survey participants can then indicate if they normally find useful information in each field, or if they find blank, incomplete, or incorrect information. Finally, the bug prioritisation questions ask developers directly how prevalent priority inflation (or deflation) is, how it impacts their work, and what measures are taken to alleviate it.

RQ2: *How does priority inflation impact software development teams?*

The bug prioritisation questions aim to discover whether the bug reporters assign priorities that differ from their true assessment. Reporters can dishonestly over or under state bug priority. As seen in Table 3, 25% of the participants reported working on projects where priority inflation is frequent while other 64% reported that priority inflation occurs occasionally. Regarding priority deflation, 15% work on projects where the bug report priorities are frequently understated, while 64.63% report that deflation occurs occasionally.

31% of those who answered P5 affirm that understated/overstated priorities have a significant impact on their daily duties, while 50% of them believe the impact is minimal. P5 is inadvertently ambiguous: we contend that most readers would interpret it to be one-sided and only about negative impact, but we recognise that some may interpret it as two-sided. To address this, we analysed P6 in depth and found that from the participants that include an impact description, 82% reported a negative impact, been resource misallocation the most popular response with 37%. Those numbers show that our participants found that priority inflation has a negative impact in their daily activities.

Question P6 of our survey is optional. Among 65 responses, the most popular measures were the following: (i) 34% reported a *gatekeeping* procedure, where a third-party verifies the priority included by the original reporter. This implies that the priority included by the reporter might be ignored depending only on the gatekeeper assessment; (ii) 12% mentioned *user training*, indicating the requisites and characteristics required by each level on the priority hierarchy. In summary:

> **Finding 2:** In a survey of 152 developers, 31% of respondents reported inaccurate priorities misallocated development effort, 25% stated that priority inflation occurs frequently in their projects, and 15% reported working on projects where priority deflation is frequent.

This finding suggests that priority inflation is a common problem in software teams adopting SPTs and that it has a negative impact in their daily activities. Using convenience sampling to recruit participants for our survey is a strong threat to this finding's external validity. It is, however, standard practice in an exploratory study, such as ours [33].

4 Modeling Software Processes as Games

This section is a precis of the game theoretic concepts on which this work rests. First, we present extended form games: a game representation suitable for process modelling. Extended form

2. In convenience sampling, the main selection criteria is ease of collection.

TABLE 2
The questionnaire presented to 152 software engineers.

Bug Reporting Duties
R1: When you create a bug report, which fields do you usually fill out?
R2: If a bug report changes, who changes it?
R3: When does your bug report need updating?
R4: In what percentage of your bug reports does the priority field change?
R5: When writing bug reports, how often do you overstate the priority to speed resolution?
Bug Fixing Duties
F1: How useful are bug reports for fixing bugs?
F2: In what percentage of your bug reports are the priority fields useful?
Bug Prioritization
P1: How many priority levels are typically supported by the bug reporting system(s) you use?
P2: How many priority levels do you think are needed for your current project(s)?
P3: Considering your current software project(s): How often is priority understated (or deflated) in bug reports?
P4: Considering your current software project(s): How often is priority overstated (or inflated) in bug reports?
P5: Is priority inflation/deflation affecting your work?
P6: If priority inflation/deflation is affecting your work, please detail how and what steps are being taken to address it.

TABLE 3
Survey responses to questions about the frequency of priority inflation and deflation in the respondent's current software project.

Anomaly	Question	Never	Occasionally	Frequently
Priority Inflation	P3	11%	64%	25%
Priority Deflation	P4	20%	65%	15%

games generate enormous game trees when approaching to real-world scenarios, so we introduce two game reduction techniques: Empirical game-theoretic analysis (Section 4.2) and the Twins Player Reduction (Section 4.3). These techniques bring software processes into the reach of our analysis.

4.1 Extended Form Games

Software processes are inherently temporal. In game theory, extensive form games (EFGs) represent players interacting over time. At their core, EFGs model multi-round games as trees, where some nodes represent nature and inject non-determinism into the game and the rest represent players. At each of a player's nodes, the player chooses an action, so the number of actions determines the out-degree of each player node. Formally, we have

Definition 4.1. (Extensive form game) A finite *perfect-information extensive form game* is a tuple (N, H, p, A, U) where:

- N is a set of n players;
- H is a set of histories, where a history is a possible sequence of actions. $H^T \subseteq H$ is the set of terminal histories.
- $p : H \setminus H^T \mapsto N$ is a function that assigns to each non-terminal history $h \in H \setminus H^T$ a player who must make a decision at h.
- For each $h \in H \setminus H^T$, $A(h)$ is the set of actions that player $p(h)$ may take after history h. A player can also randomize over these actions.
- $U = (u_1, \ldots, u_n)$ where $u_i : H^T \mapsto \mathbb{R}$ is the utility function of player i at a terminal history $\in H^T$ [34] [14].

The path to a node from a game's starting node is a *history*, since it captures all decisions all the players took to reach that node. *Terminal histories* correspond to completed games and contain each player's payoff.

Sazawal and Sudan's extensive-form game model of software evolution is an excellent example of Defintion 4.1 applied to a seminal software engineering problem [35]. In their game, end-users and a development team are players. The user's actions are to accept a design or make a change request; the development team actions are to respond to each request by accommodating the design, fully restructuring it, or simply ignoring the change request.

A game tree grows exponentially in the out-degree of each node with the number of rounds as the base. Naïve use of EFGs requires reasoning about astronomically huge trees. Sophisticated use of EFGs has an extensive literature that details various ways to employ abstraction to reduce game trees [36]. Among these approaches, we use two: empirical game theory analysis and the twins reduction.

4.2 Empirical Game-Theoretic Analysis

Empirical game-theoretic analysis (EGTA), proposed by Wellman [7], is a game theoretic framework that employs two techniques to reduce game size: sampling action sequences and simulation to reduce an EFG to a normal form game, in which all players move only once, simultaneously. It samples each player's action sequences to reduce the out-degree of player nodes and restricts the tree's height to the number of players, as shown in Figure 2. In the abstracted game, each player's "action" is to choose an action sequence from among that player's possible action sequences in the original game. This restriction of a player's actions to action sequences naturally restricts the height of the tree. EGTA compresses complex games into smaller representations, and simulation is key to accomplish this. EGTA simulates each terminal history of the reduced representation to compute the corresponding pay-off values per player.

Figure 2 models a two-player three-round rock-paper-scissors game under EGTA: From a full game tree of height 6 and 364 nodes we obtained an abstract game of height 2 and 7 nodes. Instead of having actions at the round level, now a player's action is to select an *action sequence*. Player 1 has two available sequences: Rock, paper, and scissors (RPS) or a stochastic sequence where playing scissors, then paper and finally rock has a probability of 0.3 and the probability of playing paper, rock, and scissors is 0.7 (30% SPR / 70% PRS). Player 2 can choose between rock-rock-paper (RRP) and paper-paper-scissors (PPS). The terminal histories contain the

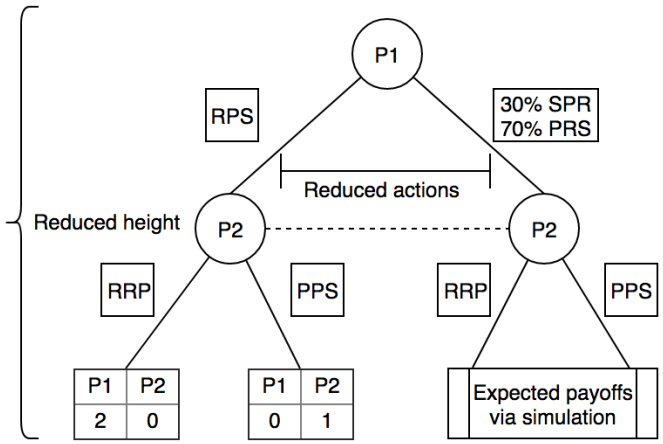

Fig. 2. EGTA-abstracted game for a 2-player-3-round rock-paper-scissors game: Each round victory is rewarded with 1 point and draws give no points to any player.

pay-off values per player. For example, when player 1 selects RPS and player 2 selects RRP, player 1 wins twice, producing a 2-0 score. However, in the terminal histories that involve a stochastic sequence (like 30% SPR / 70% PRS) the expected pay-off values must be obtained by averaging simulation results.

The first realization of EGTA, due to Walsh *et al.*, predates the definition of EGTA itself [37]. Walsh *et al.* use *heuristic strategies*, defined as "policies that govern the choice of individual actions" [37]. This definition is very general: if we are using a game tree representation like the one in Figure 2, any program that traverses it qualifies as a heuristic strategy. We do not need this degree of generality, so, in this work, heuristic strategies are probability distributions over the actions at a decision node, guarded by a condition over the game state. A game analyst defines heuristic strategies based on their understanding of the game, to test hypotheses about player behaviour, or by interviewing experts or participants.

For example, consider Assessor's Dilemma in Table 1 (Section 2) again. The primitive actions in this game are "inflate" and "accurate". An extended form version of this game would permit Alice and Bob to repeatedly choose one of these actions over a sequence of bugs. A possible heuristic strategy for this extended game might be "inflate only if there are more than 10 high priority bugs in the development queue". We call the set of heuristic strategies for a empirical game its *strategy catalogue*.

Normal form games are those games in which all players move only once, simultaneously, like a single round of rock-paper-scissors. Many game representations can be reduced to a normal-form, so it is considered "arguably the most fundamental in game theory" [14]. Walsh *et al.* represent empirical games using the normal-form, whose heuristic pay-off table is obtained via simulation [37]. Once the heuristic payoff table is ready, several algorithms are available for obtaining its Nash Equilibrium [38]. In this paper, we rely on the algorithm implementations provided by Gambit, a software tool extensively used in research [39]. Gambit has proven to be an invaluable resource, as it allowed us to focus our efforts in game-theoretic analysis instead of algorithm implementation.

When building this pay-off table, Walsh *et al.* assume a *symmetric game* [37]. In symmetric games, pay-off values are independent of the players' identities and depend instead only on players' actions. Consider the scenario in Section 2: Bob and Alice have the same action set and their pay-offs depend only on the action played. A symmetric game with a player set N and an action set S needs to compute $\binom{|N|+|S|-1}{|N|}$ entries for its pay-off table, instead of the $|S|^{|N|}$ entries required for an asymmetric game.

4.3 The Twins Player Reduction

EGTA is not enough to bring many interesting software processes into computational reach. Game representation size using the normal form and a pay-off matrix grows exponentially with the number of players and strategies [40]. Our bug repair and issue resolution data, for instance, contains 235 players. Even when we restrict each player to 7 actions, the resulting games are infeasibly large. This problem is not new. The EGTA community has already proposed several *player reduction* techniques to address it [41], [42], [43].

An intuitive player reduction approach is to cluster players by their payoffs and strategies. Modelling all the players in a cluster as a single decision maker, however, ignores the fact that players within a cluster may act differently because of the lockstep actions of the other players within the cluster. For example, imagine a simplified priority inflation game with a cluster with 10 QA engineers, where 9 of them adopt the accurate action. Clearly, the remaining QA engineer has a strong incentive to deviate and start inflating. Thus, the Nash equilibria computed for games using naïve clustering can be inaccurate.

To solve this problem, Ficici *et al.* proposed the *Twins Player Reduction* approach [43]. Given a set of pure (*i.e.* deterministic) strategies, Ficici *et al.* compute a feature vector for each player whose components are the average payoff for each strategy over a set of sample game instances, then cluster them through k-means. In Ficici *et al.*'s nomenclature, players in the same cluster have the same *strategic view*. To support a reduced game that permits a player to deviate from their cluster's strategy, Ficici *et al.* represent each cluster with two players, the eponymous *twins*, in the reduced game. In a twins game, assume Player 1 selects Strategy A and his twin, Player 2, selects Strategy B: Player 1's payoff corresponds to an agent who plays Strategy A in the full game while all other agents play Strategy B and player 2's payoff corresponds to an agent who plays Strategy B in the full game while all other agents play Strategy A.

Twins Player Reduction applies to both symmetric and asymmetric games, but is especially powerful when applied to symmetric games. When the game is symmetric, all players have the same expected payoffs for all strategies, hence they all have the same strategic view and fall into the same cluster. As we explained above, Walsh's EGTA (Section 4.2) assumes symmetry, so combining it with the Twins Player Reduction reduces the number of players to two and improves the scalability of our analysis. Although a twins game allows a twin to defect, Ficici *et al.* chose to restrict their analysis to Twin Symmetric Nash Equilibria (TSNE), a subset of Nash equilibria in which both twins adopt the same strategy. Ficici *et al.* proved that all games have a TSNE. We have followed them here: From the equilibria produced by the game solver, we keep only TSNE. Ficici *et al.*'s obtained pay-offs from a linear regression model trained with actual game data or simulation outputs. However, Wiedenbeck and Wellman [42] obtained better results via direct simulation, so this is the approach we adopt in this paper.

TABLE 4
The TASKASSESSOR Corpus of Issues extracted from JIRA and GitHub.

Project Name	Drive-by R.	Engaged R.	Issues	Non-default
OFBIZ	151	95	5120	51.5%
CASSANDRA	281	116	7417	53.7%
CLOUDSTACK	115	99	7463	47.9%
MAHOUT	54	25	1044	36.8%
ISIS	15	5	1125	67.3%
SPARK	35	14	1330	44.4%

5 TASKASSESSOR: MODELLING BUG REPAIR

Figure 3 describes TASKASSESSOR, our approach for modelling software processes. There are three inputs. Two involve strategies, which players use to decide which actions to take. The game analyst must, of course, find sufficient process data from which to extract empirical strategies, strategies we observe players following in the data, and inputs for the process simulator. The analyst also generates heuristic strategies from domain experts or to test hypotheses about how participants interact in the game. The analyst must also generate a reduced game to make the analysis tractable. We merge the empirical and heuristic strategies, then feed them, along with the relevant process data and the reduced game, to the simulator to compute the payoff matrix. Finally, we compute the Nash equilibria.

With Nash equilibria in hand, we compare it with the goals of the process we are analysing. Usually, we will find mismatches between the desired equilibrium and a player's equilibrium strategies, because we will rarely employ this analysis on effective processes. We will diagnose how a player's action set and incentives cause these mismatches, then use mechanism design to consider changes to the action set and rewards to reduce or eliminate these mismatches.

This section introduces and validates TASKASSESSOR, our simulator that models bug repair and issue resolution as a game. Section 5.1 describes the bug repair and issue resolution corpus with which we built and validated TASKASSESSOR. Modelling task prioritization requires deciding what it is relevant and important to capture and what not. Section 5.2 details those decisions and Section 5.3 describes how we built TASKASSESSOR. Section 5.4 validates TASKASSESSOR, then Section 5.5 discusses the treats to TASKASSESSOR's validity. Section 5.6 closes by describing how to use TASKASSESSOR.

5.1 Bug Repair and Issue Resolution Corpus

We collected bug repair and issue resolution data from open source projects in the Apache Software Foundation JIRA Repository (version 6.3.4) [44], using its public REST API [45]. JIRA manages *issues*, which represent software artefacts, such as bugs, feature requests, or tasks. From these data sources, we built a corpus of issue lifecycle data. In our corpus, 53% of the issues are bug reports. We dropped JIRA projects that we could not match with a Git repository [46], because we use Git commits to determine whether or not an issue was resolved. This gave us 15 projects.

Our game-theoretic model of bug prioritisation is suitable for scenarios where 1) teams resolve bugs according to their assigned priority and 2) QA engineers are interested in obtaining fixes for their reported bugs and therefore compete for developer time and attention. Such projects are subject to the assessor's dilemma. Despite the cost of developer time, many projects do not use the priority field, so we exclude them. For us, a project is not actively using priorities if the proportion of issues with non-default priorities is less than 30.0%, the rounded median of non-default priority usage over our dataset. This project-using-priority filter left us with 6 projects and their issues, as shown in Table 4. When building and evaluating TASKASSESSOR, we consolidated the issues across these projects to maximise the total data available.

Reporters that participate sporadically in bug repair and issue resolution are not really involved our task assessment game and will not learn from or respond to changing rewards; they are not acting as QA engineers. We define an engaged reporter as one who files at least 10 different issues in 10 different days. Under this definition, 53.3% of reporters are engaged. We deem the rest to be drive-by, unengaged reporters, and discard their issues. Under this engaged-reporter, *i.e.* QA engineering, filter, we extracted 23,499 issues from these 6 projects, reported between May 2006 and November 2015 and involving 354 reporters. The code to extract our corpus from JIRA is available at our project page [12].

We applied our using-priorities and engaged-reporter filters to focus on people and projects actively using JIRA's shared task prioritisation tooling. They can, of course, also introduce bias. Ablation showed that removing these filters just slows experimentation without changing the results. Game generation for distributed prioritisation under reduced bandwidth (Section 6.1) takes 96% more time without the filters. When only the engaged-reporter filter is active, game generation takes 44% more time. This figures in 28% when the only filter active is using-priorities. All three of these scenarios produce the same Nash equilibria.

Out-of-the-box, JIRA supports five priority labels: Blocker, Critical, Major, Minor, Trivial [47]. JIRA defines these labels, but few developers know JIRA's definitions and rely instead on their meanings in ordinary language. These meanings naturally split these five labels in two: {Blocker, Critical, Major} and {Minor, Trivial}. Within each subset, distinctions can be hard to make: is Blocker worse than Critical? Further, different definitions and rankings will emerge in different projects, especially in a corpus that is not Google-scale. For these reasons, we reduce these labels to two, mapping Blocker, Critical, Major to High and Minor and Trivial to Low. This boosts signal and allows us to focus on harmful mislabelling of priority (whether inflation or deflation).

5.2 TASKASSESSOR as a Game

Developers are expensive; their attention is a scarce resource for which new features and bug fixes compete [48]. Some software processes rely on "Quality Assurance" (QA) engineers to report issues, monitor their progress, and verify their resolution. We model such processes as a tragedy of the commons in which QA engineers — the players — compete with each other for the shared commons of developer time.

Our focus is priority inflation in shared prioritisation tooling, so, in our game, QA engineers can inflate, deflate, or honestly report an issue's priority. In line with the competent programmer hypothesis [49], we assume QA engineers are competent and usually know the ground truth priority of an issue. In this work, we are using classic game theory, in which players behave rationally. Thus, QA engineers seek to maximise the number of their issues that developers resolve. Later sections show that this simple model is sufficient to capture actual issue prioritisation behaviour and to

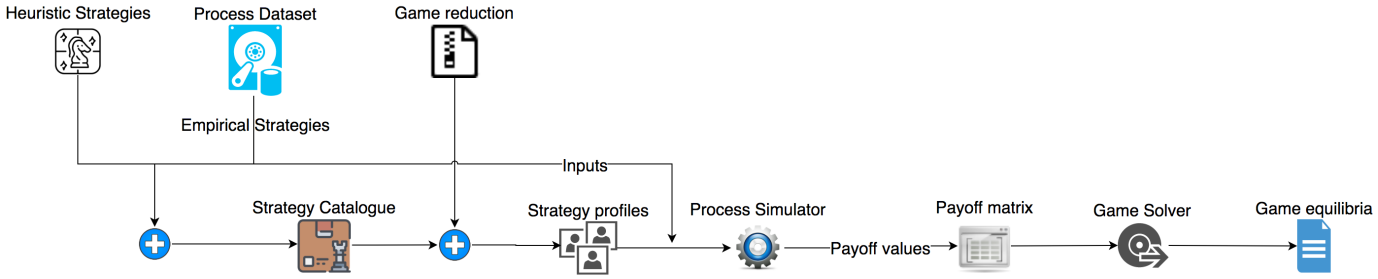

Fig. 3. TASKASSESSOR: Empirical game design for modeling software processes.

provide a solid foundation for a mechanism design solution to the problem of priority inflation. Considering behavioural game theory is future work [50].

In the TASKASSESSOR game, a QA engineer's strategy is their propensity to change an issue's ground truth priority. Let P_g be a random variable that denotes the ground truth priority of an issue, H_q be a random variable for the QA engineer q reporting an issue as high and L_q denote reporting an issue as low priority. A QA engineer's *inflation probability* is $P_I = P(H_q|P_g = L)$; it is a QA engineer's conditional probability to inflate a low priority task. A QA engineer's *deflation probability* is $P_D = P(L_q|P_s = H)$; it is a QA engineer's conditional probability to deflate a high priority task. Hence, the probability for a QA engineer to honestly assess an issue is $1 - P_I$ and $1 - P_D$. A QA engineer is "honest" if s/he never knowingly misprioritises an issue, *i.e.* $P_I = P_D = 0$ for her/him, and as "dishonest" if s/he always inflates or deflates, *i.e.* $P_I = P_D = 1$ for her/him.

To determine our players' empirical strategies, we look to the data. To learn an engineer's strategy is to learn his P_I and P_D. To do this from data, we need the ground truth. Using labelled data is a possibility, but we adopt a different strategy: the expedient of 3rd party assessment. In our data set, 3rd party assessment manifests itself as a report whose priority label was changed by a 3rd party; 254 bug reporters filed such a report. Section 5.5 discusses the construct threat this proxy for inflation poses.

To extract empirical strategies from our corpus, we cluster observed strategies. We used the k-means algorithm implementation from *scikit-learn* [51] to cluster the players and infer these strategies. In Table 5, the rows whose Origin is "Apache data" show the empirical strategies obtained. We are also very interested in assessing how the honest and always inflate ($P_I = 1 \neq P_D = 0$) strategies perform in the assessor's dilemma because we want to encourage honest prioritisation and discourage inflation. Thus, we add these two heuristic strategies (Section 4.2) to the empirical strategies we mined; Table 5, as a whole, is TASKASSESSOR's strategy catalogue.

Surprisingly, deflation dominates inflation in three of the five empirical strategies in Table 5; indeed, the persistent deflaters deflate *all* high priority issues that pass through their hands. Just over 40.0% of all reporters are deflaters. Clearly, a large portion of reporters are focusing on reducing the number of high priority issues that developers see, rather than merely maximizing the number of their issues that developers fix. A payoff function that counts all of a QA engineer's issues, would implicitly penalize deflators and fail to explain their behaviour. Thus, our payoff function is simply the count of issues that a QA engineer files as high priority that the developer team fixes:

$$payoff(r) = \sum_{f \in F} h(r, f), \tag{1}$$

where F is the set of all fixes or features the development team implements and h returns 1 if r files f with high priority.

A classical game consists of players, actions, strategies, and payoff function (Section 2). Here, we have described such a game. Unfortunately, classical game theory does not scale.

5.3 TASKASSESSOR under Twins and EGTA

Now, we discuss how we reduce game size to make the analysis of our priority inflation game tractable. It requires two major changes. Working top down, we reduce the number of players by clustering them following the Twins Player Reduction (Section 4.3), then implement our payoff function as a simulation model to handle temporality, as required by EGTA (Section 4.2),

Table 6 describes TASKASSESSOR's parameters. N_D is the size of the development team. TASKASSESSOR uses the queueing discipline of the development queue to model whether developers consider the reported priority. M_{dev} = Priority specifies total trust in priority labels; M_{dev} = FIFO specifies total mistrust. When M_{dev} = Priority, tie breaking is FIFO. A simulation run stops when the development team fixes N_f bugs.

We also assume that bug fixes and new features are independent from each other and can be resolved with a single commit made, since evidence suggests this happens in the majority of cases [52]. QA engineers file reports in the tracking system in batches after executing a group of test cases. Hence, we model report arrival with two random variables: the time between batches T_{IA} and the number of reports per batch N_b.

Each report has a ground truth JIRA priority P_g and is assigned to QA engineer R. The JIRA project observed that its users tended to confuse severity, the technical difficulty of a task, with its priority, its value to an enduser [9]. Thus, they decided to only keep one field — priority — whose purpose is to define "the order in which engineers should work on issues" [53]. Since we use JIRA data, we use JIRA priorities. A QA engineer's assessment strategy S_r governs the priority they assign to a task. Tasks require different time to be fixed, which depends on their JIRA priority; T_{rp} determines, for each priority p, the amount of time to resolve a task. Developers ignore some reports; Q_p captures this probability. To set these parameters to TASKASSESSOR, we build empirical probability distributions based on a linear interpolation between sample quantiles [54]. We define $\bar{I}$ to be a tuple of settings bound to all the parameters in Table 6. We treat $\bar{I}$ as an associative array and use $\bar{I}[name]$ to access its components.

TABLE 5
Strategy catalogue S for our task prioritisation game. P_I and P_D represent the conditional probabilities that a QA engineer inflates or deflates an issue.

Strategy	P_I	P_D	Cluster Size	Origin	Description
Honest	0.00	0.00	–	Heuristic	Players adopting this strategy always report their priority assessment
Always Inflate	1.00	0.00	–	Heuristic	Players adopting this strategy report every bug discovered as high priority
Empirically Honest	0.05	0.01	50.39%	Apache data	Empirical strategy with the lowest probabilities for dishonesty
Empirically Inflater	0.19	0.02	9.06%	Apache data	Empirical strategy with the highest probability for priority inflation
Persistent Deflater	0.08	1.00	7.87%	Apache data	Empirical strategy with the highest probability for priority deflation
Regular Deflater	0.04	0.58	16.54%	Apache data	Empirical strategy with a significant probability for priority deflation
Occasional Deflater	0.06	0.26	16.14%	Apache data	Empirical strategy with a high probability for priority deflation

TABLE 6
PlayGame's input variables. Random variables are sampled during a run until the number of bug fixed equals N_f. The nonrandom variables are constant during a simulation run.

Nonrandom Parameters	
R	The set of QA engineers (reporter).
N_D	Size of the developer team.
M_{dev}	Queuing discipline of the development queue (FIFO or Priority).
N_r	Number of simulation runs.
N_f	Number of bugs to fix in a simulation run.
Random Parameters	
T_{IA}	Interarrival time of report batches.
N_b	Count of bug reports contained in a report batch.
P_g	Ground-truth priority of a bug (High or Low).
R	QA engineer (reporter) who filed a report or issue.
$\{S_r\}$	Set of mixed strategies over inflate/deflate/honest each reporter r adopts.
T_{rp}	Resolution time of issues/bug reports with JIRA priority p.
Q_p	Probability the developer team ignores a bug with priority p.

Algorithm 1 [TASKASSESSOR] This algorithm uses TWINS, Algorithm 2, to construct a payoff matrix for our priority inflation game.

Input: S, The strategy catalogue, defined Table 5.
$\bar{I}$, tuple of PlayGame's simulation parameters in Table 6.
Output: *payoffMatrix*, TASKASSESSOR's payoff matrix.
1: **for all** $(s_i, s_j) \in S \times S$ **do**
2: $payoffMatrix[i,j] := \text{TWINS}(s_i, s_j, \bar{I})$
3: **return** *payoffMatrix*

Under the Twins Player Reduction, ours is a symmetric, two player game. From $|S|$ and $\bar{I}$, TASKASSESSOR, as defined in Algorithm 1, forms the payoff matrix in Table 7. The coordinates of each cell is a pair of actions, *i.e.* an action profile. Each cell contains $\text{TWINS}(s_1, s_2, \bar{I})$, the payoff for each player under that action profile.

Algorithm 2, which defines TWINS, manifests TASKASSESSOR's use of the Twins players reduction, in the context of a symmetric game: it binds one action to a distinguished player and binds the other action to all the other players on line 4, then swaps those bindings on line 6.

PlayGame, defined in Algorithm 3, lies at Algorithm 2's core. Algorithm 2 calls *PlayGame* N_r times and returns the average of the results of the payoffs of each run. When $\bigcup A_i = A$, the set of

TABLE 7
Pay-off matrix TASKASSESSOR builds: Since is symmetric, the game has only two players ($Twin_1$ and $Twin_2$) and both player has $|S|$ actions. Algorithm 2, TWINS, computes the payoff for each pair of actions for each cell.

	$Twin_2$: s_1	$\cdots$	$Twin_2$: s_n
$Twin_1$: s_1	$u_1, u_2 = \text{TWINS}(s_1, s_1, \bar{I})$	$\cdots$	$u_1, u_2 = \text{TWINS}(s_1, s_n, \bar{I})$
$\cdots$	$\cdots$	$\cdots$	$\cdots$
$Twin_1$: s_n	$u_1, u_2 = \text{TWINS}(s_n, s_1, \bar{I})$	$\cdots$	$u_1, u_2 = \text{TWINS}(s_n, s_n, \bar{I})$

Algorithm 2 [TWINS] This algorithm uses symmetric twins player reduction to estimate the payoff of a action profile in a symmetric twins game (Section 4.3) via the *PlayGame* simulation.

Input: s_1, $Twin_1$'s action.
s_2, $Twin_2$'s action.
$\bar{I}$, tuple of *PlayGame*'s simulation parameters in Table 6.
Output: Average payoffs for $Twin_1$ and $Twin_2$.
1: $r := choose\ \bar{I}[R]$
2: $U_1, U_2 := \{\}, \{\}$
3: **for** $i = 1$ to $\bar{I}[N_r]$ **do**
4: $payoffs := PlayGame(\{(\{r\}, s_1), (\bar{I}[R] \setminus \{r\}, s_2)\}, \bar{I})$
5: $U_1 := U_1 \cup \{payoffs(r)\}$
6: $payoffs := PlayGame(\{(\{r\}, s_2), (\bar{I}[R] \setminus \{r\}, s_1)\}, \bar{I})$
7: $U_2 := U_2 \cup \{payoffs(r)\}$
8: **return** $\frac{1}{\bar{I}[N_r]} \sum_{u \in U_1} u,\ \frac{1}{N} \sum_{u \in U_2} u$

agents, $PlayGame(\{(A_i, s_i)\}, \bar{I})$ runs our issue resolution and bug repair game among the players in A, using the action s_i for the agents in A_i and the simulation parameters in $\bar{I}$. It returns the payoff function defined in Equation 1.

Adapting TASKASSESSOR to a new task prioritisation process requires only redefining *PlayGame*, a simple but extremely general task prioritisation simulation. In this work, we use three different definitions to model the three different processes we discuss in later sections. For each of these prioritisation processes, we were able to reuse large parts of the *PlayGame* algorithm.

Finally, TASKASSESSOR passes the resulting payoff matrix to game solver. Although we are using Gambit [55] in this paper, TASKASSESSOR is solver agnostic. TASKASSESSOR will produce payoff values for each cell of the payoff matrix, that can be then organized in the format required by a specific solver. The solver computes one or more probability distributions over each player actions (the rows or columns in Table 7, corresponding to heuristic strategies), or a *mixed strategy* per player in game-theoretic terms. This map of players to strategies is called *strategy profile*, and

Algorithm 3 [PlayGame] This algorithm corresponds to the discrete-event queueing simulation used to obtain the number of fixes per QA engineer.

Input: $\bar{I} = \langle M_{dev}, N_D, N_f, T_{IA}, N_B, P_g, P_R, T_{rp}, \{S_r\}, Q_p \rangle$, tuple of PlayGame's simulation parameters in Table 6.
Output: *Fixes*, map containing fixes per reporter.

```
time := 0
IssueQueueDev := createQueue(M_dev)
Devs := initDeveloperTeam(N_D)
Fixes := {}
while notFinished(Fixes, N_f) do
    if newBatch(T_IA, time) then
        Batch := generateBatch(N_B, P_g, P_R, T_rp)
        for all issue ∈ Batch do
            reportedPriority := assignPriority(issue, {S_r})
            enqueue(issue, reportedPriority, IssueQueueDev)
    for all dev ∈ Devs do
        devIssue := dev.currentIssue
        if done(devIssue, time) then
            Fixes[devIssue.reporter] += payoff(devIssue)
            if notIgnore(IssueQueueDev, Q_p) then
                dev.currentIssue := dequeue(IssueQueueDev)
    time += 1
return Fixes
```

each strategy profile produced by the solver corresponds to a Nash equilibrium. According to the TSNE definition (Section 4.2), TASKASSESSOR only considers strategy profiles in which both twin players perform the same mixed strategy. There are various ways to interpret the probability distributions TASKASSESSOR returns. We adopt the learning interpretation: the probability associated to each action is the fraction of the time this action is adopted in the limit, when the game is played multiple times [14].

5.4 Validating TASKASSESSOR

There is no point in diagnosing or fixing a process using an inaccurate model; useful models capture a phenomenon under study with sufficient accuracy to support decisions. Thus, validation is key to assuring stakeholders that our model effectively reflects their software development process and, therefore, is a solid test bed for evaluating the effects of the mechanism design decisions, like those in Section 7.

To validate TASKASSESSOR (Section 5.3), we assess the ability of its core simulator, *PlayGame*, to produce output indistinguishable from the process it is modelling. Specifically, *PlayGame* outputs f_l, the number of low, and f_h, high priority issues resolved per player. We start by splitting our dataset into training, validation and testing. We obtain *PlayGame*'s parameters from the training dataset (Table 6). We calibrate *PlayGame* on the validation dataset. As usual, we reserve the test data to measure the quality of *PlayGame*'s simulation. We selected 60% of our data for training-validation purposes and the other 40% for testing.

In discrete-event simulation, validation techniques range from hypothesis testing to human assessment. Hypothesis testing can be too strict and rule out simulation models that are sufficiently precise for decision making [56], which in the context of TASKASSESSOR is process diagnosis. The approach we adopted evaluates if the simulation output and the real system are close enough to ensure stakeholder trust via confidence intervals. To this end, we build a confidence interval from the simulation output, obtain the best-case and worst-case error of the interval with respect to the measure in the testing dataset, and accept or reject the simulation model by comparing the errors obtained with a threshold ε. The value of ε should be "small enough to allow valid decisions" [56]. We set $\varepsilon = 20\%$ to ensure *PlayGame* is *at most* 20% wrong when predicting the percentage of reported bugs that were fixed. Despite this imprecision, our subsequent results show that *PlayGame* captures the influence of priorities in bug fixing while keeping the *PlayGame*'s model simple, easy to understand and quick to execute.

Results: In the testing dataset, 16.1% of low priority bugs were fixed on average and 33.8% of high priority ones. When $M_{dev} = Priority$, the 95.0% confidence interval for f_l, obtained from 1,000 simulation runs, is $[18.1\%, 20.4\%]$. When the tested value falls outside the confidence interval as ours does, the best-case error is $18.1\% - 16.1\% = 2.0\%$ and the worst-case error is $20.4\% - 16.1\% = 4.0\%$. Under our validation procedure [56, Chapter 10, p.326], the validation of *PlayGame* for f_l succeeds because its worst-case error is $4.0\% \leq 20.0\% = \varepsilon$. By similar reasoning, validation succeeds for f_h as well, since the worst-case error is $17.0\% \leq 20\% = \varepsilon$, despite the fact that $f_h 33.8\% \notin [45.6\%, 50.1\%]$.

5.5 Threats to Validity

TASKASSESSOR faces the standard threat to its external validity: its results generalise only to the extent to which its corpus is representative. It is drawn from JIRA projects, filtered for use of git, use of prioritisation, and engaged reporters. These filters can introduce bias not already present in the JIRA projects. As we showed in Section 5.1, however, the last two filters do not change Nash equilibria we compute. To extract empirical strategies from our corpus, we used the number of third-party corrections to indicate dishonest reporting. Their use to proxy inflation or deflation rates represent a construct validity threat. Of course, a third-party may reprioritise a report for reasons other than a dishonest initial assessment, including honest mistakes and new information. However, under our assumptions, a dishonest QA engineer benefits from an inflated report while a third-party assessor does not. Thus, we think it is reasonable to assume that third-party assessment is more likely to be accurate. Mistakes or logic errors in TASKASSESSOR's design or implementation are the main internal validity threat to this work. We mitigate this threat in two ways. First, we have detailed TASKASSESSOR's construction so that readers can themselves assess its logical validity. Second, we validated TASKASSESSOR output using state of practice techniques from the simulation community as described in Section 5.4.

5.6 Using TASKASSESSOR

TASKASSESSOR is a diagnosis tool for task prioritisation processes, tailored to a specific process by redefining its *PlayGame* simulator. When modelling a process that is immune to priority inflation, TASKASSESSOR outputs a single equilibrium with a probability of 1.0 for the Honest Strategy. Such output means that at equilibrium every task has a reliable priority. On contrast, processes susceptible to priority inflation have a positive probability for inflationary strategies — where $P_I > 0$ like Always Inflate or Empirically Inflater in Table 5 — at one of its equilibria. A non-zero probability for those strategies means inflated reports at equilibrium. The worst-case scenario is a single equilibrium where Always Inflate has a

probability of 1.0: that means that in the limit *all* bug reports have an inflated priority. It is also possible that TASKASSESSOR finds multiple, opposing equilibria for a task prioritization process: Like Always Inflate with a probability of 1.0 in one equilibrium and Honest with a probability of 1.0 in another. As stated in Section 1, a Nash equilibrium is *stable*: once reached, players have no incentive to deviate, so learning dynamics could settle in any one of multiple equilibria.

One needs to discuss TASKASSESSOR's equilibrium results with stakeholders to validate whether they explain a prioritisation process. Process modelling is hard: usually you will need to discuss several models before stakeholder acceptance. If stakeholders reject TASKASSESSOR's results, revise the simulation model: over-simplification or over-engineering can distort pay-off calculations. Also, ensure that the strategy catalogue does not obviate common or impactful prioritisation behaviour. Once stakeholders agree with TASKASSESSOR's results, it can also evaluate process interventions to improve a prioritisation process, as we do in Section 7.3.

We implemented TASKASSESSOR in Python[3]. We implemented TASKASSESSOR's Algorithm 2 component in Simpy, a discrete-event simulation library [57]. You issue

```
taskAssessor.py -r 100 -d 50 -n 50 -o equilibria.csv
```

to generate a payoff matrix for 100 QA engineers and 50 developers, running until $N_t = 50$ bugs are fixed and storing the equilibria in `equilibria.csv`.

6 DIAGNOSING CURRENT PRACTICE

Figure 4 describes our understanding of task prioritization using the Software Process Engineering Metamodel (SPEM) notation to represent its roles, tasks and work products [58]. Three tasks, coloured blue, are common to all the task prioritization processes under analysis: reporting, prioritising, and resolving tasks.

Three different task prioritization processes are superimposed in Figure 4. In *distributed prioritization*, the person adopting the reporter role both files and prioritises tasks. In Section 6.1, we show that distributed prioritization is susceptible to priority inflation (Finding 3). To correct distributed prioritisation's tendency to priority inflation, development teams have taken prioritisation away from reporters and given it to a *gatekeeper* (light red in Figure 4). In Section 6.2, we present two findings. First (Finding 4), we make an argument from queueing theory that gatekeeping *only slows* task resolution. Second (Finding 5), we show that even a perfect gatekeeper that correctly prioritizes all reports does not remove the incentive for inflating priorities. In short, we first confirm the conventional wisdom about distributed bug prioritization, then we contradict the conventional wisdom that gatekeepers improve bug repair.

We are especially interested in the impact of developer bandwidth on priority inflation: intuitively, scarce development time will magnify the reward for inflating priorities. Hence, we compute Nash equilibria for the following two scenarios: In *full bandwidth D*, all the developers actively remove reports from the development queue; in *reduced bandwidth* $\frac{D}{2}$, only half of them are active. In all scenarios, we assume the developers consider a tasks's priority label when deciding whether to work on the task. Without this assumption, we cannot distinguish the inflationary propensity of the two processes we analyze below.

3. TASKASSESSOR is available on our project page [12].

6.1 Distributed Bug Prioritization

A bug prioritization process can assign the reporting and prioritization of bugs to the QA engineer role. Such processes are common in both FOSS and industry projects. A company, for instance, adopts such a process when they decide to outsource the development services to an external IT provider. In this scenario, the outsourcing company fixes the bug that contracted company reports and prioritizes [8]. A QA role appears in some agile settings, where the team member that discovers a bug is in charge of logging a bug and assigning its priority [59].

We call bug prioritization process involving a QA engineer *distributed bug prioritization*, because it distributes bug prioritization to QA engineers, or bug reporters. Common knowledge suggests that this process encourages priority inflation as is already reported by practitioners [2] [1]. To verify this, we ask

RQ3: *Is distributed bug prioritization susceptible to priority inflation?*

We built game-theoretic models by applying our approach (Section 5) to each bandwidth scenario D and $\frac{D}{2}$, finding a single equilibrium where the probability of the always inflate strategy is 1.0 and the probability of the other 6 strategies is 0.0. This differs from the desired outcome of an equilibrium with a probability of 1.0 for honest. This is an important finding since it corroborates conventional wisdom and validates the accuracy of our approach. In summary:

> **Finding 3:** Distributed bug prioritization *is* susceptible to priority inflation.

6.2 Do Gatekeepers Prevent Priority Inflation?

A standard approach to address priority inflation is to appoint a *bug triage* team, who inspects and corrects bug reports, including the reported priority [1]. We call a process that incorporates such a team, a *gatekeeper process*, because these teams act as gatekeepers who control access to developer time. We show that a gatekeeper process does *not* prevent priority inflation and, in fact, can slow development.

Not all software organisations necessarily use gatekeepers, but several have reported on their use. For example, Microsoft reports that their gatekeeper is a cross-discipline team [4], while Google reports that some teams delegate this task to a tester-developer pair [3]. In open-source projects, developers establish a gatekeeping rota, or they rely on volunteers from the community to perform gatekeeping duties [60], [61]. Teams adopting an agile process can also include bug triage tasks, where the product owner, a developer, or a customer representative performs the gatekeeper role [62], [63]. Our survey also shows that a gatekeeper is a common a way to tackle priority inflation (34% of responses, as reported in Section 3.2). Although developers can be part of the gatekeeping process, in our experience gatekeepers are usually not software developers. Given the high salaries of developers, we believe teams prefer to invest their time in building features than in gatekeeping tasks. The findings of this section are independent of any other roles held by the gatekeeper.

In a gatekeeper process, QA engineers place their issues into the gatekeeper's queue, not the development team's queue. This process reprises distributed prioritization in one of two ways. In the first one, gatekeepers face priority distortion instead of developers; while in the second one gatekeepers are the ones distorting priorities for the development team. Two justifications are usually advanced

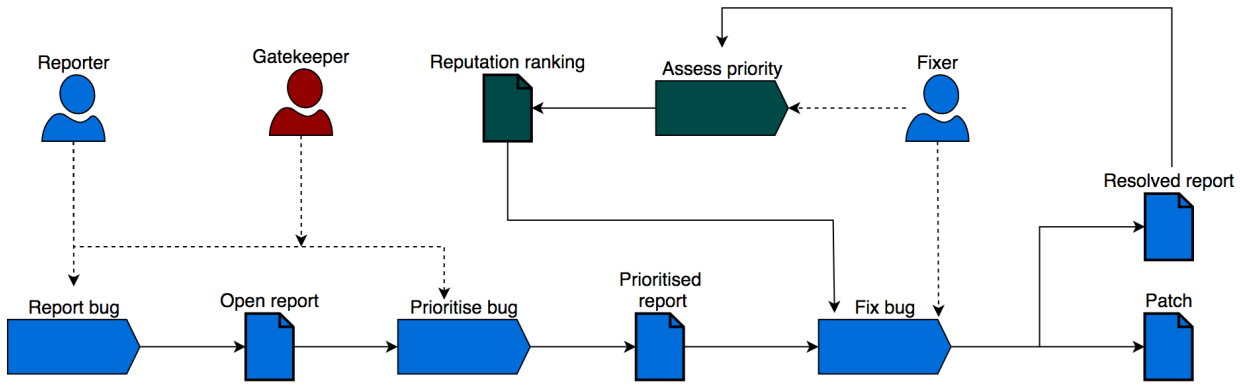

Fig. 4. Bug prioritization processes: The blue components are shared between the three processes (including distributed prioritization), the red ones are exclusive to the gatekeeper process and the green ones to the assessor-throttling process. The solid lines represent the input/output of an activity, and the dashed lines link an activity its performing role.

for adopting a gatekeeper process. One is to involve business expertise in task prioritization to ensure that priorities correctly reflect business value. The other is economic: gatekeepers are usually cheaper than developers [48] and, since they focus on issue/bug report quality, become more efficient at that task than developers. These two justifications are often in tension because of the cost of business expertise.

When an organization opts for business expertise, it assigns product managers, business analysts, or even clients to the gatekeeper role. These stakeholders can potentially be scarce, expensive, and busy even before taking on a gatekeeper role. As gatekeepers, they tend to make development slower and more costly. The economic justification breaks down when gatekeeping is a role that developers or product managers play. Recall that Jira priorities, which we perforce use in this work, combine technical severity, *i.e.* difficulty of resolution, and value, including risk, to end-users. Regardless of who fills the gatekeeper role, gatekeepers do not learn their technical severity, since they do not actually resolve issues. When gatekeepers are drawn from technical employees, like testers, they are no better or worse at learning to assess business value than developers.

To represent gatekeeping, we added two elements to our simulation model in Section 5: A bug triage queue, with a queuing discipline of $M_{gk} = FIFO$, and a gatekeeping team G, whose members take R time to triage a task with an error rate of A_{gk}. The gatekeeping queue protects developers from poor issue descriptions, but can be a bottleneck, especially if the gatekeeping team lacks resources.

RQ4: *What is the impact of a gatekeeper on issue resolution?*

Researchers have successfully used queuing systems with Poisson arrivals and exponential service times to model real-world bug repair processes [64], [65]; we follow their lead. We assume QA engineers file issues under a Poisson distribution, a gatekeeper takes exponentially distributed time to review them, and the development team take exponentially distributed time to fix them. When $\mathbf{E}(W_G)$ is the mean sojourn time of an issue in the gatekeeper (or triage) queue, $\mathbf{E}(W_D)$ is the mean sojourn time of an issue in the development team queue, and $\mathbf{E}(W_{GP})$ is the mean, end-to-end, sojourn time of an issue/bug report in a gatekeeping process, we have

Finding 4: Over a sequence of issues, a gatekeeper can only slow issue resolution: $\mathbf{E}(W_{GP}) = \mathbf{E}(W_G) + \mathbf{E}(W_D)$.

Under our assumptions of Poisson arrivals and exponential service times, the gatekeeper process is a Jackson network using tandem queues (a triage queue for gatekeepers and a developer queue) [10]. In Jackson networks, the mean sojourn time of the whole system at steady state — that is, the time an issue/bug spends in the system from reporting to fixing — is the sum of the mean sojourn times of each individual queue of the system [66].

Finding 4 matters even when an organization adopts gatekeeping to detect and remove duplicate issues before they reach developers. In general, a gatekeeper must observe and consider several issues before 1) determining that they are duplicates and 2) learning to quickly identify and drop them. Let k denote the expected number of issues one needs to view before one realizing that they are duplicate and that each gatekeeper filters reports for a team of n developers. A developer is at least as good as a gatekeeper at detecting duplicate bug reports, but each developer learns independently, so, collectively, they will need to see nk duplicates before they all can quickly drop them. Both gatekeepers and developers can learn in parallel; so, given enough duplicates, the expected time needed to learn recognize duplicates for both a gatekeeper and a developer team is the same. For m duplicate reports, there are three cases. If $m < k$, gatekeeping does not remove duplicates before they reach developers. If $m > kn$, then all the developers will have learned to identify and remove them. In this case, the gatekeeper provides no advantage in the limit. It is only when $k \leq m \leq kn$ that gatekeepers remove duplicates at less cost than simply asking developers to do it.

Under Finding 4, whenever an issue's mean sojourn on the triage queue exceeds zero, a gatekeeper reporting process slows the delivery of bug fixes. Despite the overhead the gatekeeper process imposes, if it reduces or eliminates priority inflation, it might be worth adopting, so we ask

RQ5: *Is gatekeeper prioritisation susceptible to priority inflation?*

For the equilibrium analysis, we consider a team of $G = 2$ gatekeepers that spend $R = 20$ minutes assessing the priority of an issue. We set $G = 2$ because Ayewah reported that Google used this number [67]; we set $R = 20$ because Page reported 20 minutes as the approximate triage effort per bug at Microsoft [4]. We also explored three performance profiles: a fallible gatekeeper with an

error rate of $A_{gk} = 50\%$, an expert gatekeeper with an error rate of $A_{gk} = 10\%$ and an ideal gatekeeper with an error rate of $A_{gk} = 0\%$.

In both bandwidth scenarios D and $\frac{D}{2}$, the fallible gatekeeper ($A_{gk} = 50\%$) has a single equilibrium with probability 1.0 for inflating priorities, the always inflate strategy. This is the expected result: a fallible gatekeeper leaves the door open to QA engineers profiting from inflating their reports. The expert gatekeeper ($A_{gk} = 10\%$) does no better: in both D and $\frac{D}{2}$, the expert gatekeeper also has a single equilibrium profile with probability 1.0 for the always inflate strategy: even 90% prioritization accuracy is insufficient. What about perfect accuracy? Under the ideal gatekeeper ($A_{gk} = 0\%$), each scenario produces multiple equilibria: 3 under $\frac{D}{2}$ bandwidth and 4 under D, not our desired result: a single equilibrium with probability 1.0 for honest prioritization[4]. These equilibria do not rule out priority inflation. The reason is, under the ideal gatekeeper, each QA engineer's pay-off is the same regardless of their prioritization decisions. In summary,

Finding 5: Gatekeeper prioritisation *is* susceptible to priority inflation.

7 The Assessor-Throttling Process

The gatekeeper process slows issue resolution and bug repair and does not prevent priority inflation. In this section, we present assessor-throttling: a novel task prioritization process that, unlike gatekeeping, is lightweight and removes priority inflation.

We first present mechanism design (Section 7.1), then apply it to priority inflation to define and model assessor throttling, our novel task prioritization process (Section 7.2). In Section 7.3, we validate that assessor-throttling *prevents* priority inflation. In Section 7.4, we compare distributed prioritization, gatekeeping, and assessor-throttling with respect to the expected proportion of high priority issues resolved. In closing, Section 7.5 presents *TheFed*, a tool to support teams who adopt assessor-throttling.

7.1 Mechanism Design

In game theory, the discipline that handles game design in order to obtain a specific behaviour is called *mechanism design*. Mechanism design is also called "inverse game theory": While in game theory, we try to predict the outcome of a strategic interaction or suggest to players a course of action, in mechanism design, we wish to design a strategic interaction that produces the player's behaviour we desire. Using game-theoretic terminology, the goal of mechanism design is to design a game whose equilibrium maximizes the global utility, independent of player preference [68]. Since we are focussed on modelling software processes, the global utility depends not only on the development team — the players — but especially on the project stakeholders. In a bug repair context, software quality is a key stakeholder requirement.

A *mechanism* has three elements [20]: the strategies players are able to perform within a game, the mechanism centre that keeps track of players behaviour, and an outcome rule. The mechanism centre uses the outcome rule to assign a pay-off value to each player, according to the strategies they all perform in the game. This outcome rule must be known to all players.

A mechanism should ideally be *strategy-proof* and *incentive compatible*. We called a mechanism *strategy-proof* if all the agents have a dominant strategy. These mechanisms have the advantage that players do not need to consider the other players' actions to maximize their pay-offs [20]. A mechanism is *incentive compatible*, or *truthful*, when it requires players to reveal their private information — in the game context— at equilibrium. For example, when modelling an auction an incentive compatible mechanism requires the bidders to bid their private value for the auctioned item. According to the *revelation principle* of mechanism design, every mechanism, as defined above, can be transformed to an equivalent mechanism that is incentive compatible, where all players reveal their preferences [40]. A mechanism designer's goal is to find such transformations.

7.2 Modelling Assessor-Throttling

Assessor-throttling (AT) rests on the insight that developers naturally assess tasks while resolving them. Currently, this developer assessment is wasted. We can use it to assess the priority assigned to a task by the bug reporter or QA engineer who filed it. To construct an incentive compatible mechanism from this developer assessment of task priority, we rate each QA engineer's assessment accuracy; this rating becomes a QA engineer's reputation. A QA engineer's reputation then controls their access to the developer team in two ways: 1) we use reputation to control how many issues a QA engineer can add to the development queue when it is under contention and 2) we display the QA engineer's reputation when developers are considering whether to take up a task from the work queue. Honest QA engineers tend to get more developer time; dishonest (or incompetent) ones will get less, and eventually no, access.

The assumption that underlies this reputation mechanism is that developers can accurately assess a task's or bug's priority. JIRA priorities combine technical severity and business value. Developers necessarily overcome a task's technical severity when they resolve it. While developers vary in expertise and some might find a bug more difficult, and thus more severe, than other developers, they are better placed to assess severity than testers or triage teams. Assessing business value is harder. The ground-truth assessment of business value relies on the role that generates software requirements, like the customer, a business analyst, or the business owner. Nonetheless, software engineers can estimate the business value when they work in the same domain long enough, they can eventually qualify as domain experts [11].

Like distributed prioritisation, assessor-throttling decentralises prioritisation: AT does not introduce a second queue, in contrast to gatekeeping, and requires *both* QA engineers and developers assess task priority. We model developer assessment mistakes with A_{dev}. The developer's assessment is the green task in Figure 4. QA engineers and task assessors have a reputation R^r. When a developer finishes a task, they consider its assigned priority. If the developer agrees with that priority, she rewards the assessor by increasing R^r by T_+; if she disagrees, she penalises the assessor by decreasing R^r in T_-. To allow an assessor to recover from a mistake, we have $T_+ > 0$; to discourage priority distortion, $T_+ \ll T_-$. Assuming that developer assessment approximates the ground-truth, this behaviour penalises both dishonest and incompetent QA engineers. AT does not need to rely on the QA engineer's intentions to improve bug prioritisation: both dishonest and incompetent prioritisation are indistinguishable and discouraged under our model, under the weak assumption that QA engineers learn over time. Under AT, as noted above, R^r controls the rate at which they can submit tasks to the developer's work queue and likelihood a developer selects their tasks.

4. These equilibria are available at our project page [12].

TABLE 8
Pay-off matrix for the assessor's dilemma using assessor throttling.

	Bob: accurate	*Bob: inflate*
Alice: accurate	$A = 125, B = 125$	$A = 125, B = 100$
Alice: inflate	$A = 100, B = 125$	$A = 112.5, B = 112.5$

Under AT, developers take action: they reward or penalise QA engineers. This action implies that we should model them as strategic agents, or players, rather than a commons, as TASKASSESSOR currently does. Indeed, developers might abuse their power to rank QA engineers who prioritise tasks they like. As we explain in Section 5.3, however, adding developers as players to our game-theoretic model would produce an unmanageably large game tree and break the symmetry assumption on which a number of the game reduction techniques (Section 4.2) we use depend. Thus, treating developers as players is future work.

Simulation in Section 7.3 demonstrates the effectiveness of our mechanism. This is unsurprising, because it aligns with previous work: Restricted to bug repair, Guo *et al.* found that the bug reporter reputation is a key factor in the likelihood of a bug to be fixed [69]. They define reputation as proportion of reported bugs that are fixed, following Hooimeijer and Weimer [70]. Neither of these prior works propose a new process that exploits reputation. AT is a novel task prioritization process that does exploit a reporter's reputation and permits developers to change a QA engineer's reputation. Further, AT defines as agreement with a task resolver, not the proportion of bugs reported and fixed. Integrated into AT's reward system, this operationally changes the definition of reputation into a measure of honest reporting.

7.3 Assessor-Throttling Prevents Priority Inflation

Let us see how throttling works in the situation described in Section 2. If Alice reports the accurate priorities of her bugs and Bob inflates his reports, we have two possible outcomes: Alice gets only her high priority report fixed — Bob's inflation was detected after all his patches were delivered — or she gets two fixes, because Bob's inflated trivial bug was fixed first so he was marked as an offender. Hence, now Alice expected pay-off is $0.5 \times 100 + 0.5 \times (100 + 50) = 125$ and offender Bob obtains $0.5 \times (100 + 50) + 0.5 \times 50 = 100$.

If we update the original pay-off matrix with the throttling pay-off values, we obtain the matrix in Table 8. The Nash Equilibrium of the new matrix has both Alice and Bob reporting the accurate priorities, which is in *Foo Inc.'s* best interest. Assessor-throttling appears to prevent priority inflation. To confirm, we ask

RQ6: *Is the assessor-throttling susceptible to priority inflation?*

Assessor-throttling depends on two parameters: the error rate of the development team (A_{dev}) for detecting dishonesty and the penalty they apply when they detect such behaviour (T_{-}). In our experiments, the development team error rate was fixed to $A_{dev} = 5\%$: it is a more palatable value than $A_{dev} = 50\%$ — that makes the priority field irrelevant — or $A_{dev} = 100\%$ — that would necessarily lead to an equilibrium with dishonest prioritization. Regarding inflation penalty T, it needs to have a value big enough so that the expected benefits from inflation are less than the expected penalty due to reputation loss. But for the sake of deployability, we want T_{-} to be small since big penalties can face resistance. We start with $T_{-} = 3\%$ and progressively augment it until obtaining an equilibrium with honest prioritization. Each value of T_{-} was analysed under the 2 bandwith scenarios used in Section 6.

When applying TASKASSESSOR to assessor-throttling with $T_{-} = 3\%$, the $\frac{D}{2}$ bandwidth scenario produced a single equilibrium where the probability of the always inflate strategy was 1.0. As explained in Section 5.6, that result identifies a process susceptible to priority inflation. This lead us think that the value of the penalty with respect to the inflation reward is too low to discourage this behaviour.

If the dishonesty penalty is set to $T_{-} = 10\%$, TASKASSESSOR produced one TSNE equilibria for each bandwidth scenario. In the $\frac{D}{2}$ bandwidth scenario, TASKASSESSOR outputs the desired equilibrium where the honest strategy has a probability of 1.0, while in the D bandwidth scenario this probability is 0.97. Although close, for the D bandwidth scenario we still do not obtain priority inflation immunity. By increasing the penalty value, we now observe that equilibria with high probability for honest behaviour start to appear.

When applying TASKASSESSOR to an assessor-throttling process with $T_{-} = 20\%$, we obtain the same result in both scenarios: a single equilibrium where the honest strategy has a probability of 1.0. We obtain the same result with a dishonesty penalty of $T_{-} = 22\%$. Having the expected equilibrium with such a low penalty value is an indicator of the actionability of the assessor throttling process. This also reflects that penalty calibration is a key factor for the effectiveness of it. In summary:

Finding 6: Assessor-throttling prevents priority inflation.

After calibrating the penalty-value parameter, assessor throttling produces an equilibrium where the honest strategy has a probability of 1.0, which implies honest bug prioritization.

7.4 Racing to the Fixes

We now compare each of the task prioritization processes presented under the mean percentage of high priority bugs that were fixed $\mathbf{E}(g_h)$. We show that assessor-throttling always performs at least as well as the other two prioritization processes in the bandwidth scenarios used in Section 6. In fact, assessor-throttling is statistically indistinguishable from an ideal gatekeeper. The comparison is made in terms on impact on software quality, so our last research question is

RQ7: *What is the impact of the adopted task prioritization process in the quality of the software product?*

Several techniques are available in the simulation and operations research communities for finding the best simulated system design according to an expected performance measure [71]. The two-stage Bonferroni procedure proposed by Nelson and Matejcik [72] is one of the many techniques that rely on an *indifference zone*. Indifferent zone techniques are known to be statistically conservative: they guarantee a lower bound to the probability of selecting the best system $1 - \alpha$ given that this system is at least ε better than the rest of the systems [73]. In our context, the system designs under comparison are the prioritization processes at their equilibria (Table 9) and the performance measure is $\mathbf{E}(g_h)$. The two-stage Bonferroni procedure takes three parameters: ε, $1 - \alpha$, and the first-stage sample size R_0. We execute R_0 iterations on each task prioritization process at equilibrium, which is then used as an input to obtain the second-stage sample sample size R. When $R > R_0$, two-stage Bonferroni requires to execute additional $R - R_0$ iterations. From the simulation iterations, we obtain $\mathbf{E}(g_h)$ per task prioritization process at equilibrium. From the simulation results,

TABLE 9
Performance comparison of task prioritization processes in the $\frac{D}{2}$ bandwidth scenario.

Reporting process	# equilibria	$\mathbf{E}(g_h)$ at best
Distributed Prioritization	1	28%
Gatekeeper $A_{gk} = 50\%$	1	52%
Gatekeeper $A_{gk} = 10\%$	1	97%
Gatekeeper $A_{gk} = 0\%$	3	97%
Assessor-throttling $T = 3\%$	1	22%
Assessor-throttling $T = 10\%$	1	97%
Assessor-throttling $T = 20\%$	1	97%

we build confidence intervals. From them, we can conclude that a task prioritization process is either inferior to the best performer or statistically indistinguishable from it.

Due to computational costs, we simulate each scenario for $R_0 = 120$ iterations. We would also like a 95% confidence of obtaining the best process, given that the best differs from the second best by at least $g_h = 5\%$. That translates to $1 - \alpha = 0.95$ and $\varepsilon = 0.05$. Table 9 presents the $\mathbf{E}(g_h)$ for each task prioritization process in a $\frac{D}{2}$ bandwidth scenario: In case the task prioritization process has more than one equilibrium we report the one the best performance. The statistically indistinguishable best repair processes are gatekeeper with $A_{gk} = 10\%$ error rate, gatekeeper with a $A_{gk} = 0\%$ error rate, assessor-throttling with $T_{-} = 20\%$ dishonesty penalty and assessor-throttling with $T_{-} = 10\%$ dishonesty penalty. The best performer has a performance value of $\mathbf{E}(g_h) = 0.97$, which is significantly better than the ones marked as inferior.

Under D, when all developers are available, the best performer is one of the equilibriums of the gatekeeper with $A_{gk} = 0\%$ error rate with a performance value of $\mathbf{E}(g_h) = 0.96$. Distributed prioritization with $\mathbf{E}(g_h) = 0.57$ and assessor throttling with a $T_{-} = 3\%$ dishonesty penalty and $\mathbf{E}(g_h) = 0.16$ are inferior, while the rest are statistically indistinguishable from the best performing equilibrium of gatekeeper with $A_{gk} = 0\%$. Given a sufficiently strong penalty value, we find

Finding 7: Developers fix as many high priority bugs under assessor-throttling as under an ideal, error-free gatekeeper.

7.5 TheFed: Tool Support for Assessor-Throttling

Assessor-throttling relies on a reputation system for task assessors, so we built a software tool to track assessor reputation and support the teams who want to adopt assessor-throttling. Our tool, the *TheFed*[5], is a Chrome extension for JIRA. *TheFed* is open source [74] and has these key features: *TheFed*

1) Tracks each QA engineer's reputation.
2) Allows developers to penalize inflation.
3) Rewards honest QA engineers.

In assessor-throttling, developers and task assessors make decisions based on the reputation score R^r, so *TheFed* must calculate and store these values. To this end, *TheFed* uses JIRA's REST API [75] to obtain how many times a QA engineer incorrectly prioritised an issue, relative to the priority assessment of the developer who resolved the issue. R^r is a function of the number of these "infractions", so developers need to update it when a priority assignment is inaccurate. To support this feature, *TheFed* relies on JIRA's existing functionality for updating priority; issue resolvers use this functionality to state that they disagree with a QA engineer's priority assessment. Assessor-throttling aims to give high-reputation task assessors a higher probability that their bugs will be fixed than low-reputation assessors. To this end, *TheFed* provides a prioritized inbox. This inbox shows open and unassigned bugs sorted by R^r: honest QA engineers will have their bugs listed on top. Also, *TheFed* displays QA engineers, ranked by R^r.

We hope that development teams adopt assessor-throttling. This cannot happen if its adoption disrupts existing practices and tools. To maximize the deployability, *TheFed*

1) Is compatible with existing toolkits;
2) Tackles priority inflation immediately after installation; and
3) Deploys to clients, not servers.

The penalty value per infraction is a key parameter of assessor-throttling. As is shown in Section 7.3, a penalty value that is too low might not produce the desired output of a single equilibrium with the honest strategy probability of 1.0. *TheFed* allows this parameter to be configurable. To this end, we allow the users of *TheFed* to customize its behaviour by using the options page functionality available for Chrome extensions. Some of the parameters available are `penalty_per_infraction`, `issues_in_inbox` and `JIRA_project`.

Tools drive software development: consider text editors, IDE's and bug tracking systems, to name a few. *TheFed* integrates easily with existing tools. The current version of *TheFed* is a Chrome extension that relies on JIRA's REST API to access issue data. Chrome extensions are also compatible with the Opera browser and *TheFed* can be extended to other bug tracking systems that expose a REST API, like Bugzilla [76]. From the moment it is installed, *TheFed* provides value: it immediately tackles priority inflation; it calculates R^r using previous priority updates accessed through JIRA's API. From this useful starting point, *TheFed* only improves through network effects. Developers directly and immediately benefit from *TheFed* since it helps them better prioritize their work, but it does not directly benefit system administrators, who tend to work on different tasks. Crucially, developers can install *TheFed* locally. This allows individual team developers to adopt it without requiring support from a JIRA system administrator, unlike a JIRA plugin.

8 Related Work

Game Theory was made popular in 1940 when John von Neumann used it for economic analysis [77], and since then it has been applied to many fields including Computer Science [6]. While we are not the first to apply it to software engineering, this paper is the first end-to-end process improvement approach driven by empirical game-theoretic analysis. In this section, we review previous applications of game theory to software engineering, describing their proposals and how we differ.

Grechanik *et al.* proposed that the conflict of interests between project management, the customer organisation and the development team devise a game [78]. Since these conflicting interests can produce undesired outcomes to the development organisation, he proposed game theory for analysing these strategic interactions. He modelled the software development process as a game, but he

5. *TheFed* stands for Federal Reserve: the central banking system of USA. As such, it defines target inflation rates.

did not validate his model nor calculate its Nash equilibrium. Oza proposed a two-player game for offshore development, between the customer organisation and an offshore software development team [79]. He also claimed that this game is an instance of the Prisoner's Dilemma and discussed issues to consider when designing a model. For example, Oza suggests to model the power a vendor acquires while providing value for the client, and mutual perceptions before contract negotiation. Both papers, like us, identify the importance and relevance of game-theory for software engineering scenarios, but not formulate or analyse explicit game-theoretic models. Hata *et al.* [80] used game theory to model contributor behaviour in open source projects. Using a game in extensive form, they analysed if any improvement to the development environment setup — like documentation — would translate to code contributions. They later verify their findings with data from open-source projects in GitHub. Their model only considers two players with two actions each, and they recognise it can be an oversimplification. Our EGTA-based approach does not suffer this limitation.

Other publications focus on modelling agile practices. For example, instead of considering the software development team as a monolithic player, Hazzan and Dubinsky explore the interactions between software development team members using game theory: They proposed that the cooperation dynamics inside a team is an instance of the Prisoner's Dilemma [81]. They also described how cooperation scenarios relate to extreme programming (XP) practices, but no formal game-theoretic analysis methods were applied. Hasnain *et al.* studied the role of communication in developing trust among agile development team members [82]. The authors used an experimental game-theoretic approach, using students and practitioners to play several instances of a simulated *work-shrink game* — an instance of Prisoner's Dilemma — to see the impact of stand-up meetings in the development of trust, finding a positive correlation among them. Being a study with human participants, the scope of the work is limited: only 28 iterated games instances were used. In this work we rely on simulation and equilibrium calculation to be able to analyse scenarios with multiple players and iterations while keeping experimentation cost-effective.

Software Testing is another activity within the software process that has been analysed from a game-theoretic perspective. Feijs modelled the whole software testing process as a strategic-form game played between the development team and the testing team, where their action set is defined by their performance at work (i.e. do a good job or do a bad job) [83]. According to the pay-off values given by their assumptions, he concluded that the *Testing Game* is an instance of the Prisoner's Dilemma. As a way of validating the proposed model the author applied it to the *text copy testing game (TCTG)*, an artificial testing scenario. In TCTG, the specification is to transcribe an input string. Both the actions for the developer and the tester are different performance levels. In the case of the developer, performance is a function of characters wrongly transcribed and for the tester performance is a function of the number of characters inspected. Also in the testing domain, Kukreja *et al.* proposed a game-theoretic approach for randomising test case execution [84]. Due to resource constraints, an entire regression test suite often cannot be executed, so the authors propose modelling test execution as a *security game* between testers and developers, where developers try to introduce code without being tested and testers try to force them to test all code. They demonstrate their method on a toy example and compare it with uniform randomization, leaving a more exhaustive evaluation to future work.

In the same fashion as this work, there have been previous attempts for software process improvement through mechanism design. Yilmaz *et al.* proposed to model a software process as a mechanism and provided an example, but the validation of this approach was relegated to future work [85]. Bacon *et al.* [86] approached software estimation as a mechanism design problem, exploring the characteristics of scoring systems that would lead developers to share information with managers — who estimate taks— and to perform their best during implementation. Bacon *et al.* [86] test their scoring systems via simulation but, unlike TASKASSESSOR, their models are not based on process data and rely on assumptions. A key benefit of TASKASSESSOR is its data-driven approach, which makes models closer to the process under study.

9 Conclusion and Future Work

Improving software processes is challenging: initiatives for change face resistance and several barriers [87]. Game theory can help bring down these barriers, because games are a way to describe, investigate and optimize processes. We have shown how to conceptually and pragmatically recognise and prevent priority inflation using game theory. We believe problems like priority inflation occur all too often in software development and that game theory can diagnose and fix them, as readily as it led us to a solution for priority inflation. You can join us in this effort at http://ttendency.cs.ucl.ac.uk/gametheory4se.

Acknowledgments

The authors would like to thank the anonymous professionals who responded the survey presented in this study. This research is funded in part by the Dynamic Adaptive Automated Software Engineering Programme Grant (EP/J017515).

References

[1] P. Butcher, *Debug it! : find, repair, and prevent bugs in your code*. Raleigh, N.C: Pragmatic Bookshelf, 2009.
[2] M. Doar, *Practical development environments*. Sebastopol, Calif: O'Reilly, 2005.
[3] J. A. Whittaker, J. Arbon, and J. Carollo, *How Google tests software*. Addison-Wesley, 2012.
[4] A. Page, K. Johnston, and B. Rollison, *How we test software at Microsoft*. Microsoft Press, 2008.
[5] R. Myerson, *Game theory: Analysis of Conflict*. Harvard University Press, 1991.
[6] Y. Shoham, "Computer science and game theory," *Communications of the ACM*, vol. 51, no. 8, p. 74, Aug. 2008.
[7] M. P. Wellman, "Methods for empirical game-theoretic analysis," in *Proceedings, The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference, July 16-20, 2006, Boston, Massachusetts, USA*, 2006, pp. 1552–1556.
[8] J. Watkins and S. Mills, *Testing IT: an off-the-shelf software testing process*. Cambridge University Press, 2010.
[9] "Bug tracking for JIRA server," accessed: 2019-03-22. [Online]. Available: https://jira.atlassian.com/browse/JRASERVER-886
[10] D. Gross, *Fundamentals of queueing theory*. John Wiley & Sons, 2008.
[11] S. Kelly, *Domain-specific modeling : enabling full code generation*. Hoboken, N.J: Wiley-Interscience IEEE Computer Society, 2008.
[12] "Improving software processes via empirical game theory," accessed: 2019-02-14. [Online]. Available: http://ttendency.cs.ucl.ac.uk/gametheory4se
[13] S. Brue and R. Grant, *The Evolution of Economic Thought*. Cengage Learning, 2012.

[14] K. Leyton-Brown and Y. Shoham, *Essentials of Game Theory: A Concise Multidisciplinary Introduction*, ser. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2008. [Online]. Available: https://doi.org/10.2200/S00108ED1V01Y200802AIM003

[15] P. A. Laplante and N. B. Ahmad, "Pavlov's bugs: Matching repair policies with rewards," *IT Professional*, vol. 11, no. 4, pp. 45–51, 2009. [Online]. Available: https://doi.org/10.1109/MITP.2009.80

[16] J. Nash, "Non-cooperative games," *Annals of mathematics*, pp. 286–295, 1951.

[17] I. Palacios-Huerta, "Professionals play minimax," *The Review of Economic Studies*, vol. 70, no. 2, pp. 395–415, 2003.

[18] M. Walker and J. Wooders, "Minimax play at wimbledon," *The American Economic Review*, vol. 91, no. 5, pp. 1521–1538, 2001.

[19] R. Axelrod, *The Evolution of Cooperation: Revised Edition*. Basic Books, 2009.

[20] P. Norvig and S. Russell, *Artificial Intelligence: A Modern Approach*. Pearson Education, 2011.

[21] R. Banfield, *Product leadership : how top product managers Launch awesome products and build successful teams*. Beijing: O'Reilly, 2017.

[22] M. Krogerus and R. Tschäppeler, *The Decision Book: Fifty models for strategic thinking (New Edition)*. Profile, 2017.

[23] A. Stellman and J. Greene, *Learning agile: Understanding scrum, XP, lean, and kanban*. " O'Reilly Media, Inc.", 2014.

[24] R. Black, *Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing*. Wiley, 2013.

[25] "About labels - user documentation," accessed: 2018-10-19. [Online]. Available: https://help.github.com/articles/about-labels/

[26] "What is an issue (v6.4) - atlassian documentation," accessed: 2018-10-19. [Online]. Available: https://confluence.atlassian.com/jira064/what-is-an-issue-720416138.html

[27] "Bugzilla/fields - mediawiki," accessed: 2018-10-19. [Online]. Available: https://www.mediawiki.org/wiki/Bugzilla/Fields

[28] "Github labels for better workflows - yoast," accessed: 2018-10-19. [Online]. Available: https://yoast.com/dev-blog/github-labels/

[29] "How we use labels on github issues at mediocre laboratories," accessed: 2018-10-19. [Online]. Available: https://mediocre.com/forum/topics/how-we-use-labels-on-github-issues-at-mediocre-laboratories

[30] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. Germán, and D. E. Damian, "The promises and perils of mining github," in *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India*, P. T. Devanbu, S. Kim, and M. Pinzger, Eds. ACM, 2014, pp. 92–101. [Online]. Available: https://doi.org/10.1145/2597073.2597074

[31] V. Cosentino, J. L. C. Izquierdo, and J. Cabot, "A systematic mapping study of software development with github," *IEEE Access*, vol. 5, pp. 7173–7192, 2017. [Online]. Available: https://doi.org/10.1109/ACCESS.2017.2682323

[32] O. Jarczyk, B. Gruszka, S. Jaroszewicz, L. Bukowski, and A. Wierzbicki, "Github projects. quality analysis of open-source software," in *Social Informatics - 6th International Conference, SocInfo 2014, Barcelona, Spain, November 11-13, 2014. Proceedings*, ser. Lecture Notes in Computer Science, L. M. Aiello and D. A. McFarland, Eds., vol. 8851. Springer, 2014, pp. 80–94. [Online]. Available: https://doi.org/10.1007/978-3-319-13734-6_6

[33] P. Lavrakas, *Encyclopedia of Survey Research Methods: A-M.*, ser. A SAGE reference publication. SAGE Publications, 2008.

[34] N. McCarty and A. Meirowitz, *Political game theory: an introduction*. Cambridge University Press, 2007.

[35] V. Sazawal and N. Sudan, "Modeling software evolution with game theory," in *International Conference on Software Process*. Springer, 2009, pp. 354–365.

[36] T. Sandholm, "Abstraction for solving large incomplete-information games," in *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA.*, 2015, pp. 4127–4131.

[37] W. E. Walsh, R. Das, G. Tesauro, and J. O. Kephart, "Analyzing complex strategic interactions in multi-agent systems," in *AAAI-02 Workshop on Game-Theoretic and Decision-Theoretic Agents*, 2002, pp. 109–118.

[38] N. Nisan, T. Roughgarden, E. Tardos, and V. Vazirani, *Algorithmic Game Theory*. Cambridge University Press, 2007.

[39] R. D. McKelvey, A. M. McLennan, and T. L. Turocy, "Gambit: Software Tools for Game Theory, Version 14.1.0." [Online]. Available: http://www.gambit-project.org

[40] E. Tardos and V. V. Vazirani, "Basic solution concepts and computational issues," *Algorithmic game theory*, pp. 3–28, 2007.

[41] M. P. Wellman, D. M. Reeves, K. M. Lochner, S. Cheng, and R. Suri, "Approximate strategic reasoning through hierarchical reduction of large symmetric games," in *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*, 2005, pp. 502–508.

[42] B. Wiedenbeck and M. P. Wellman, "Scaling simulation-based game analysis through deviation-preserving reduction," in *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*. International Foundation for Autonomous Agents and Multiagent Systems, 2012, pp. 931–938.

[43] S. G. Ficici, D. C. Parkes, and A. Pfeffer, "Learning and solving many-player games through a cluster-based representation," in *UAI 2008, Proceedings of the 24th Conference in Uncertainty in Artificial Intelligence, Helsinki, Finland, July 9-12, 2008*, 2008, pp. 187–195.

[44] "JIRA- issue and project tracking software," accessed: 2018-10-21. [Online]. Available: https://www.atlassian.com/software/jira

[45] "System dashboard - ASF JIRA," accessed: 2018-10-21. [Online]. Available: https://issues.apache.org/jira/

[46] "Github - the apache software foundation," accessed: 2018-10-21. [Online]. Available: https://issues.apache.org/jira/

[47] "What is an issue (v6.3) - atlassian documentation," accessed: 2018-10-21. [Online]. Available: https://confluence.atlassian.com/jira063/what-is-an-issue-683542485.html

[48] J. Spolsky, *Joel on software : and on diverse and occasionally related matters that will prove of interest to software developers, designers, and managers, and to those who, whether by good fortune or ill luck, work with them in some capacity*. Berkeley, CA: Apress, 2004.

[49] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[50] K. Holyoak and R. Morrison, *The Oxford Handbook of Thinking and Reasoning*, ser. Oxford Library of Psychology. OUP USA, 2012.

[51] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[52] M. Rath, J. Rendall, J. L. C. Guo, J. Cleland-Huang, and P. Mäder, "Traceability in the wild: automatically augmenting incomplete trace links," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, Eds. ACM, 2018, pp. 834–845. [Online]. Available: https://doi.org/10.1145/3180155.3180207

[53] "Organizing issues with priority to optimize delivery," accessed: 2019-03-24. [Online]. Available: https://www.atlassian.com/blog/jira-software/organizing-issues-priority-optimize-delivery

[54] G. Fishman, *Discrete-Event Simulation : Modeling, Programming, and Analysis*. New York, NY: Springer New York, 2001.

[55] "Gambit documentation," accessed: 2018-11-29. [Online]. Available: https://media.readthedocs.org/pdf/gambitproject/latest/gambitproject.pdf

[56] J. Banks, *Discrete-event system simulation*. Upper Saddle River, NJ: Pearson Prentice Hall, 2005.

[57] K. G. Müller and T. Vignaux. (2011) Simulation with simpy - in depth manual. [Online]. Available: https://pythonhosted.org/SimPy/Manuals/Manual.html

[58] S. OMG and O. Notation, "Software & systems process engineering meta-model specification." *OMG Std., Rev*, vol. 2, 2008.

[59] M. Lacey, *The Scrum field guide : practical advice for your first year*. Addison-Wesley Professional, 2012.

[60] H. Naguib, N. Narayan, B. Brügge, and D. Helal, "Bug report assignee recommendation using activity profiles," in *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*, T. Zimmermann, M. D. Penta, and S. Kim, Eds. IEEE Computer Society, 2013, pp. 22–30. [Online]. Available: https://doi.org/10.1109/MSR.2013.6623999

[61] J. Anvik and G. C. Murphy, "Reducing the effort of bug report triage: Recommenders for development-oriented decisions," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 3, pp. 10:1–10:35, 2011. [Online]. Available: https://doi.org/10.1145/2000791.2000794

[62] L. Crispin, *Agile testing : a practical guide for testers and agile teams*. Upper Saddle River, NJ: Addison-Wesley, 2009.

[63] E. Brechner, *Agile project management with Kanban*. Redmond, WA: Microsoft Press, 2015.

[64] S. S. Gokhale and R. E. Mullen, "Queuing models for field defect resolution process," in *17th International Symposium on Software Reliability*

Engineering (ISSRE 2006), 7-10 November 2006, Raleigh, North Carolina, USA, 2006, pp. 353–362.
[65] B. Luong and D.-B. Liu, "Resource allocation model in software development," in *Annual Reliability and Maintainability Symposium. 2001 Proceedings. International Symposium on Product Quality and Integrity (Cat. No.01CH37179)*, 2001, pp. 213–218.
[66] S. Bose, *An Introduction to Queueing Systems*. Boston, MA: Springer US Imprint Springer, 2002.
[67] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Evaluating static analysis defect warnings on production software," in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. ACM, 2007, pp. 1–8.
[68] Y. Shoham and K. Leyton-Brown, *Multiagent systems: Algorithmic, game-theoretic, and logical foundations*. Cambridge University Press, 2008.
[69] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy, "Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, 2010, pp. 495–504.
[70] P. Hooimeijer and W. Weimer, "Modeling bug report quality," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 2007, pp. 34–43.
[71] S.-H. Kim and B. L. Nelson, "Selecting the best system," *Handbooks in operations research and management science*, vol. 13, pp. 501–534, 2006.
[72] B. L. Nelson and F. J. Matejcik, "Using common random numbers for indifference-zone selection and multiple comparisons in simulation," *Management Science*, vol. 41, no. 12, pp. 1935–1945, 1995.
[73] K. Inoue, S. E. Chick, and C.-H. Chen, "An empirical evaluation of several methods to select the best system," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 9, no. 4, pp. 381–407, 1999.
[74] C. Gavidia-Calderon, "TheFed at GitHub," accessed: 2019-03-10. [Online]. Available: https://github.com/cptanalatriste/inflation-tracker-extension
[75] Atlassian, "The JIRA Cloud Platform REST API," accessed: 2019-03-10. [Online]. Available: https://docs.atlassian.com/jira/REST/cloud/
[76] Bugzilla, "WebService API Reference," accessed: 2019-03-10. [Online]. Available: http://bugzilla.readthedocs.io/en/latest/api/
[77] S. Stahl, *A gentle introduction to game theory*. Providence, R.I: American Mathematical Society, 1999.
[78] M. Grechanik and D. E. Perry, "Analyzing software development as a noncooperative game," in *26th International Conference on Software Engineering - W9L Workshop "Sixth International Workshop on Economics-Driven Software Engineering Research (EDSER-6)"*, 2004, pp. 29–34.
[79] N. V. Oza, "Game theory perspectives on client: vendor relationships in offshore software outsourcing," in *Proceedings of the 2006 international workshop on Economics driven software engineering research*. ACM, 2006, pp. 49–54.
[80] H. Hata, T. Todo, S. Onoue, and K. Matsumoto, "Characteristics of sustainable oss projects: A theoretical and empirical study," in *Proceedings of the Eighth International Workshop on Cooperative and Human Aspects of Software Engineering*. IEEE Press, 2015, pp. 15–21.
[81] O. Hazzan and Y. Dubinsky, "Social perspective of software development methods: The case of the prisoner dilemma and extreme programming," in *International Conference on Extreme Programming and Agile Processes in Software Engineering*. Springer, 2005, pp. 74–81.
[82] E. Hasnain, T. Hall, and M. J. Shepperd, "Using experimental games to understand communication and trust in agile software teams," in *6th International Workshop on Cooperative and Human Aspects of Software Engineering, CHASE 2013, San Francisco, CA, USA, May 25, 2013*, 2013, pp. 117–120.
[83] L. Feijs, "Prisoner's dilemma in software testing," *7e Nederlandse Testdag*, p. 65, 2001.
[84] N. Kukreja, W. G. Halfond, and M. Tambe, "Randomizing regression tests using game theory," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2013, pp. 616–621.
[85] M. Yilmaz, R. V. O'Connor, and J. Collins, "Improving software development process through economic mechanism design," in *European Conference on Software Process Improvement*. Springer, 2010, pp. 177–188.
[86] D. F. Bacon, D. C. Parkes, Y. Chen, M. Rao, I. Kash, and M. Sridharan, "Predicting your own effort," in *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*. International Foundation for Autonomous Agents and Multiagent Systems, 2012, pp. 695–702.
[87] B. W. Boehm and R. Turner, "Management challenges to implementing agile processes in traditional development organizations," *IEEE Software*, vol. 22, no. 5, pp. 30–39, 2005.

Carlos Gavidia-Calderon is a PhD. student in Software Engineering at University College London, under the supervision of Dr Earl T. Barr, Dr Federica Sarro, and Prof. Mark Harman. His research interests are game-theoretic applications to software engineering, reinforcement learning, and agent-based models.

Federica Sarro is an Associate Professor at the University College London. Her research covers Predictive Analytics for Software Engineering (SE), Empirical SE and Search-Based SE, with a focus on software effort estimation, software sizing, software testing, and mobile app store analysis. On these topics she has published more than 65 peer-reviewed conference and journal papers. She has also received several international awards, including three best paper awards, the ICSE'19 ACM distinguished paper award, a GECCO-HUMIES award, and the ICSE'18 ACM distinguished reviewer award.

Mark Harman is an engineering manager at Facebook London, where he manages a team working on Search Based Software Engineering (SBSE), a rapidly growing field he co-founded. He is also a part time professor of Software Engineering at University College London, where he directed the CREST centre for ten years (2006-2017) and was Head of Software Systems Engineering (2012-2017). He is known for work on source code analysis, software testing, app store analysis and empirical software engineering. In 2019, he won Harlan D. Mills award for fundamental contributions throughout software engineering.

Earl T. Barr is a senior lecturer (associate professor) at the University College London. He has won international awards, including three ACM distinguished paper awards, the ten year most influential paper at the Mining Software Repositories conference, a research highlight in the Communications of the ACM, and two GECCO-Humies awards. His research interests include bimodal software engineering, testing and analysis, and computer security. His recent work focuses on engineering software development processes using game theory, optional typing, and using machine learning to solve programming problems. He dodges vans and taxis on his bike commute in London.