**An Exploratory Study of the Pull-based Software Development Model**

ABSTRACT

分布式版本控制系统的出现导致了分布式软件开发的新模式的发展；开发人员不再将修改推送到中央存储库，而是从其他存储库拉取修改并在本地进行合并。各种代码托管网站，特别是Github，通过提供工作流程支持工具，如代码审查系统和集成问题跟踪器，已经挖掘出促进基于拉动的开发的机会。在这项工作中，我们首先在GHTorrent语料库上，然后在291个项目中精心挑选的样本上，探索基于拉动的软件开发是如何进行的。我们发现，拉动请求模式提供了快速的周转，增加了社区参与的机会，减少了纳入贡献的时间。我们表明，相对较少的因素会影响合并拉动请求的决定和处理它的时间。我们还研究了拒绝拉动请求的原因，发现技术上的原因只占一小部分。

# INTRODUCTION

Pull-based development是分布式软件开发的一种新兴范式。随着越来越多的开发者对孤立的开发和分支的欣赏[7]，越来越多的项目，包括封闭源代码，特别是开放源代码，正在被迁移到代码托管网站，如Github和Bitbucket，支持基于拉动的开发[2]。这类网站的一个独特特点是，它们允许任何用户克隆任何公共资源库。克隆创造了一个属于克隆用户的公共项目，因此用户可以修改资源库而不需要成为开发团队的一部分。此外，这类网站通过拉动请求，自动将克隆的提交内容有选择地贡献给源代码。

拉动请求作为一种普遍的分布式开发模式，特别是由Github实现的，形成了一种分布式软件开发的新方法。其新颖之处在于，将开发工作与将开发成果纳入代码库的决定相分离。通过分离构建工件和整合变化的关注点，工作被干净地分配到贡献者团队和核心团队之间，贡献者团队提交（通常是偶尔的）考虑合并的变化，核心团队监督合并过程，提供反馈，进行测试，要求修改，并最终接受贡献。

以前的工作已经确定了分布式开发中通过补丁提交和接受的协作过程[23, 5, 32]。拉动请求的工作方式有许多相似之处；例如，出现了类似的工作团队结构，因为通常拉动请求会经过一个评估过程。此外，拉动请求所提供的是流程自动化和信息的集中化。通过拉动请求，代码不必离开修订控制系统，因此它可以跨库核查，而作者信息的维护也很省力。关于修改的交流是针对具体环境的，它以单个拉动请求为根基。此外，Github的重新查看机制还具有提高认知度的作用[9]；核心开发人员可以有效地访问与拉动请求有关的所有信息，并征求社区对合并决定的意见（"众包"）。

**如果拉动请求最终被接受，那么分布式开发工作流程就是有效的；如果拉动请求的时间尽可能短，那么分布式开发工作流程就是高效的。提高我们对拉动请求处理的有效性和效率的洞察力，对贡献者和开发者都有直接意义。**这项工作的目标是深入了解拉动请求的使用，并分析影响基于拉动的软件开发模式效率的因素。具体来说，我们试图回答的问题是：

**RQ1** 基于拉动的开发模式有多流行？

**RQ2** 拉动请求的生命周期特征是什么？

**RQ3** 哪些因素会影响合并拉动请求的决定和所需时间？

**RQ4** 为什么有些拉动请求没有被合并？

我们的研究基于Github合作开发平台的数据，这些数据是通过我们的GHTorrent项目[16]提供的。

利用它，我们首先探索了Github所有项目中近200万个拉动请求的使用情况。然后，我们检查了291个经过仔细检查的Ruby、Python、Java和Scala项目（总共有166,884个拉动请求），并使用定性和定量分析，确定影响拉动请求寿命、合并和拒绝的因素。

# BACKGROUND

(Read carefully)

自2001年出现以来，分布式版本控制系统（DVCS），特别是Git[8]，已经彻底改变了分布式软件开发的方式。在实用主义需求的驱动下，大多数DVCS从头开始设计，作为先进的补丁管理系统，而不是作为当时主流的版本控制模式的文件系统。在大多数DVCS中，一个文件是一组有序的变化，这些变化的连续应用导致了当前的状态。变更是由全球唯一的标识符标记的，它可以用来跟踪跨仓库的提交内容。当整合变化时，变化集可以来自本地文件系统或远程主机；工具有助于在本地镜像上获取和应用变化集。DVCS的分布式性质使得基于拉动的开发模式成为可能，在这种模式下，变更是通过项目分叉网络提供给项目库的；由库主决定接受或拒绝传入的拉动请求。

分布式开发的目的是使潜在的贡献者能够向由核心团队管理的软件项目提交一组修改。DVCS提供的开发模式是集中式版本控制环境中的超集[31, 6]。在接收和处理外部贡献方面，DVCs可以采用以下策略。

**共享数据库**。核心团队与贡献者共享项目的版本库，并有读写权限。为了工作，贡献者在本地克隆它，修改它的内容，可能会引入新的分支，并将他们的修改推回给核心团队。为了应对多个版本和多个开发者，大型项目通常采用分支模式，即在贡献者被合并到主要开发分支之前，用一种有组织的方式来检查和测试这些贡献者[7]。

**拉动请求**。项目的主资源库不在潜在的贡献者之间共享；相反，贡献者分叉（克隆）资源库，并使他们的修改彼此独立。当一组修改准备提交给主仓库时，他们会创建一个拉动请求，指定一个本地分支与主仓库的分支合并。然后，项目核心团队的成员负责检查这些修改，并把它们拉到项目的主分支。如果修改被认为不满意，可以要求更多的修改；在这种情况下，贡献者需要用新的提交来更新他们的本地分支。此外，由于拉取请求只指定可以拉取某些提交的分支，因此没有任何东西禁止它们在共享仓库的方法中使用（跨分支拉取请求）。图1是拉取请求流程的概述。

## 2.1 Pull requests in GitHub

Github支持上述所有类型的分布式开发；然而，拉动请求得到了特殊的对待。该网站经过调整，允许贡献者轻松地分叉项目，同时通过项目分支的自动比较促进拉动请求的产生。Github的拉动请求模型遵循上述的通用模式；此外，它还提供了上下文讨论和在线代码审查的工具。图2是Github上一个拉动请求的例子。

Github拉取请求包含一个分支（本地或其他仓库），核心团队成员应从该分支拉取提交。Github会自动发现要合并的提交，并在拉动请求中呈现。默认情况下，拉取请求会被提交到基础（Git术语中的 "上游"）仓库进行检查。检查要么是对拉动请求所提交的提交内容进行代码审查，要么是对拉动请求所引入的功能进行讨论。任何Github用户都可以参与这两种类型的检查。作为检查的结果，拉动请求可以用新的提交进行更新，或者作为多余的、不感兴趣的或重复的提交被关闭。在更新的情况下，贡献者在分叉的仓库中创建新的提交，而Github则自动更新显示的提交。然后可以在更新后的提交上重复进行代码检查。

当检查过程结束，拉动请求被认为是令人满意的，拉动请求就可以被合并了。一个拉动请求只能由核心团队成员合并。Git的多功能性使得拉动请求可以通过三种方式进行合并，以下是按照保留原始源代码属性的多少排序的:

**1. 通过Github工具**。Github可以自动验证一个拉动请求是否可以在不冲突的情况下合并到基础仓库中。当请求合并时，Github会自动应用拉动请求中的提交并记录合并事件。所有作者和历史信息都保留在合并后的提交中。

**2. 使用 Git 合并**。当拉动请求不能被干净地应用时，或者当项目相关政策不允许自动合并时，可以使用普通的Git工具来合并拉动请求，使用的技术如下:

- 分支合并: 分叉仓库中包含拉取请求提交的分支被合并到基本仓库的一个分支中。历史和作者信息都被保留，但Github无法检测到合并以记录合并事件[8，第3.2章]。

- 择优挑选: 合并不是合并所有的提交，而是从远程分支中挑选特定的提交，然后应用到上游分支中。唯一的提交标识符会发生变化，因此无法保持准确的历史记录，但作者的身份会被保留下来 [8, Chapter 5.3]。

与上述两种方法相辅相成的技术是提交简化（commit squashing）：如果项目对完整的历史不感兴趣，可以在拉取请求分支上将几个连续的提交合并成一个，然后将其合并到上游分支上。在这种情况下，提交的作者与申请提交的人是不同的[8，6.4章]。挑拣和压制提交都是Git支持重新排序提交（rebase）的副产品[8，第3.6章]。

**3. 提交补丁**。合并会在上游分支和拉取请求分支之间产生文字差异，然后她会将其应用于上游分支。历史和作者信息都会丢失。

由于项目分支是以分布式方式更新的，拉取请求中的修改可能会干扰到项目主分支中的新修改。合并这样的拉动请求会导致冲突。Github会自动检测冲突的拉动请求并将其标记为冲突。冲突可以由贡献者或核心团队成员来解决；然而，拉动请求的礼节要求贡献者负责将拉动请求恢复到没有冲突的状态。解决冲突的过程包括从项目的主库中拉出新的提交，进行修改以消除冲突，并通过消除冲突的提交来扩展拉动请求。

问题和拉动请求在Github上是双重的；对于每个拉动请求，都会自动打开一个问题。提交也可以附加到问题上，以将其转换为拉动请求（尽管是通过外部工具）。这种双重性使核心团队能够将拉动请求视为工作项目，可以使用用于问题的相同设施进行管理。此外，问题讨论可以包括拉动请求的链接，反之亦然，而特定的提交消息格式可以用来在提交合并到项目主分支时自动关闭问题或拉动请求。

Github的拉动请求的开放性使其适合各种使用模式。除了基本的补丁提交之外，拉动请求还可以作为需求和设计讨论工具1，或者作为项目发布的进度跟踪工具来使用。在第一种情况下，pull request可以作为一个讨论板，在实现新功能的同时征求其他开发者的意见。在第二种情况下，拉动请求与Github的问题跟踪器中的里程碑相关联。

# 3 RESEARCH DESIGN

本研究的重点是理解和解释项目如何使用拉动请求来实现协作。为了解决我们的研究问题，我们采用了一种顺序混合方法，即在一个研究过程的某个阶段收集、分析和整合定量和定性数据，以达到更好地理解问题的目的[17] 。对于具体的研究问题，我们首先对该领域进行定量探索，然后通过对案例的定性探索来突出感兴趣的案例。下面，我们介绍我们是如何处理每个研究问题的。

RQ1 为了评估基于拉动的开发模式的普及程度，我们提供并分析了Github中使用拉动请求的描述性统计数据。特别是，我们调查了这样的问题：有多少项目实际使用了拉动请求，有多少项目是原始仓库（相对于fork），以及拉动请求与Github的问题跟踪设施的关系。结果将在第5节介绍。

RQ2和RQ3 识别拉动请求的生命周期特征，并确定影响它们的因素，需要一个有足够长使用拉动请求历史的项目的专门数据集。这个数据集将在第4.2节中描述。

考虑到这个数据集，我们回答了RQ2和RQ3，通过咨询补丁提交、错误处理、代码审查和分布式协作等领域的相关工作，确定了一组合适的候选特征。然后，我们通过交叉相关分析对其进行清理，得到一组具有最大预测能力的特征。利用提取的特征数据，我们对拉动请求特征进行了详细的统计分析，以回答RQ2（第6节）。

接下来，我们使用机器学习来检索主要的特征。在运行分类算法之前，我们自动给每个拉动请求加上一个结果因子；在合并决定分类任务中，该标签标志着拉动请求是否已被合并。对于合并时间任务，我们首先过滤掉未被合并的拉动请求，然后根据合并拉动请求所需的时间，将剩余的数据点分成三类（一小时、一天、超过一天）。分割点的选择反映了RQ2的结果，并将可用的数据点分割成大致相同大小的仓。

在一个较高的水平上，检索两个分类任务的主要特征的过程的过程包括两个步骤。首先，我们通过6种分类算法运行每个数据集，即随机森林（randomforest）、逻辑回归（logregr）的变种（二进制用于合并决策任务，多进制用于合并时间任务）和奈何贝叶斯（naivebayes）、支持向量数学（svm）、决策树（dtree）和带决策树的AdaBoost（adaboost）。我们使用了这些算法，因为它们在大型数据集中这些算法在大数据集中表现良好[21]，并在以前涉及预测模型的工作中被使用[13]。我们没有对分类算法进行任何额外的调整。我们只报告前三个算法的结果，因为它们表现最好。然后，我们挑选出最佳的分类器，并应用分类器的特定过程，根据其在分类过程中的重要性对特征进行排序。

为了评估分类性能，我们使用准确度（ACC）和接受者操作特征曲线下面积（AUC）指标。为了选择合适的分类算法，我们运行了10-fold的随机选择交叉验证，并汇总每个分类指标的平均值。在每一次迭代中，该算法随机抽取一半的可用数据点，用90%的输入训练一个分类器，并用它来预测剩下的10%。10-fold的运行结果也使我们能够评估跨运行的指标稳定性（第7节）。

RQ4 为了研究为什么一些拉动请求没有被合并，我们对一组随机选择的非合并拉动请求进行了深入的定性分析。我们使用开放编码（一种基础理论工具）来得出一组关于拉动请求未被合并的原因，具体如下：第一作者阅读了Github上随机选择的拉动请求的讨论，并将关闭它们的原因总结为每个样本的一句话；在第二轮分析中，这些描述被汇总，代码被排除。为了验证所确定的代码，三位作者在不同的拉动请求上应用这些代码，比较结果，确定不一致的地方，并对最初选择的代码进行修改。最后一组代码被应用于第三个样本，我们用它来得出结果。抽样过程将在第4.3节中描述。

4. DATA  
4.1 Github data

我们使用通过GHTorrent项目[16]提供的Github数据，这是一个通过Github API提供的数据的离线镜像。Github API的数据有两种形式：流式数据流列出了实时发生在存储库上的事件，如分叉或创建拉动请求，而静态视图则包含实体的当前状态。为了获得对静态视图实体根部的引用，GHTorrent项目遵循事件流。从那里，它应用一个基于递归依赖关系的解析方法来获得通过API提供的所有数据。这些数据以未经处理的格式存储在MongoDB数据库中，而元数据则被提取并存储在MySQL关系型数据库中。GHTorrent数据集涵盖了Github上广泛的开发活动，包括拉动请求和问题。该项目自2012年2月以来一直在收集数据。截至2013年8月，已经收集了来自20多万个项目的190万个拉动请求。

4.2 Pull request project sample

**项目选择**。为了使分析切实可行，同时避免检查toy projects，我们使用了一个数据集，其中包括2012年2月至2013年8月期间GHTorrent记录了超过200个拉动请求的所有项目。最初的搜索产生了374个项目。然后应用以下标准，将项目从初始选择中排除。

项目应包括测试。为了测量测试对拉动请求接受度的影响，我们只能使用包括测试的项目，而且我们可以可靠地测量。为此，我们利用了Ruby（Gem）、Python、Java和Scala（均为Maven）语言生态系统中基于惯例的项目布局，因此我们的项目选择只限于这些语言。

项目应该至少有一个提交来自于拉动请求，以确保项目对外部贡献是开放的，拉动请求不仅仅是由项目内部的开发者使用。

项目应该是开发软件框架或应用，而不是文档或编程语言。我们排除了文档项目，因为我们对分布式软件开发感兴趣。我们排除了编程语言的实现项目，因为我们想避免开发的编程语言的核心库掩盖了实际实现的指标的情况。这对于Github上的5个Ruby实现来说尤其如此。

选定后，我们下载了所选项目的全部历史记录（包括拉动请求、问题和提交），并通过查询GHTorrent数据库和分析每个项目的Git仓库来提取特征值。此外，对于这些选定的项目，我们收集了所有的合并和所有因素的值，我们在机器学习实验中使用了这些因素，如下所述。

**Merge detection**.为了识别在Github之外合并的拉动请求，我们采用了以下启发式方法，这里按应用顺序列出:

**Pull request characteristics**:这些特征试图量化拉动请求对受影响代码库的影响。当考虑外部代码贡献时，补丁的大小会影响接受度和接受时间[32]。有各种衡量标准来确定补丁的大小，这些标准已经被重新搜索者使用：**代码流失**[24, 27]，**改变的文件[24]和提交的数量[11]**。在拉动请求的特殊情况下，开发人员报告说，**拉动请求中存在的测试**会增加他们对合并请求的信心[26]。为了研究这一点，我们将流失特征分成两个特征，即 src\_churn 和 test\_churn。参与者的数量已经被证明会影响到代码审查的过程时间[28]。最后，通过我们自己分析拉动请求的经验，我们发现在许多情况下，冲突是在拉动请求的评论中明确报告的，而在其他情况下，拉动请求包括与其他相关拉动请求的链接。

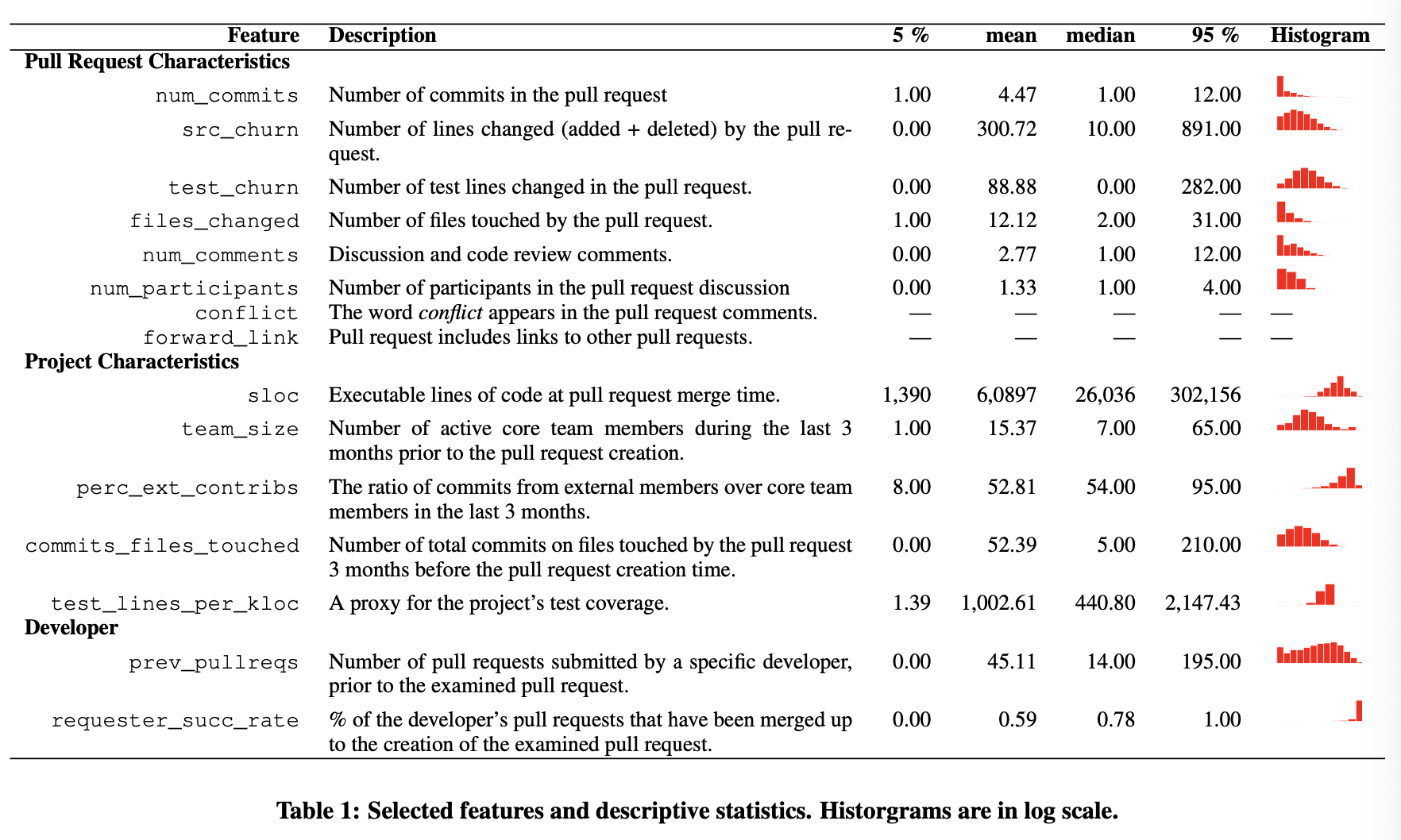
**project characteristics**.这些特征量化了项目对拉动请求的接受程度。如果项目的流程对外部贡献是开放的，那么我们希望看到外部贡献者的比例高于团队成员。项目的规模可能是一个不利于处理拉动请求速度的因素，因为它的影响可能更难评估。另外，传入的变化往往随着时间的推移而聚集（"昨天的天气 "的变化特征[15]），所以很自然地认为，影响系统中正在开发的部分的拉动请求将更有可能被合并。测试在处理速度方面发挥着作用；根据[26]，与不断变化的贡献者斗争的项目使用测试，手动或最好是自动测试，作为处理来自未知开发者贡献的安全网。

**Developer**。基于开发者的特征量化了创建拉动请求的人对合并决定和处理时间的影响。特别是，创建补丁的开发者已被证明会影响补丁的接受决定[19]。为了在不同开发者的项目中抽象出结果，我们加入了量化开发者记录的特征[9]，即以前拉动请求的数量和接受率；前者被认为是拉动请求质量的一个有力指标[26]。Bird等人[5]提出的证据表明，社会声誉对补丁是否会被合并有影响；在我们的数据集中，Github上的追随者数量可以被视为声誉的代表。

所有的特征都是在拉动请求被关闭或合并时计算的，以评估由于随后的讨论而对拉动请求进行中间更新的效果。在计算中包含时间维度的特征（例如，team\_size或commits\_on\_files\_touched）是在拉动请求开启之前的三个月时间内计算的。

最初的选择包含25个特征。为了检查所选特征是否足够独立，我们使用Spearman等级相关（⇢）指标对所有特征进行了一对一的相关分析。我们设定的阈值为⇢=0.7，超过这个阈值的特征就被剔除。使用这个阈值，我们删除了2个特征，即asserts\_per\_kloc和test\_cases\_per\_kloc，因为它们与包含的test\_lines\_per\_kloc特征有很强的相关性（p> 0.92）。我们还删除了在拉动请求完成时无法可靠计算的特征（追随者和星星）。最后，我们合并了类似的特征（例如，doc\_files和src\_files被合并到files\_changed）。

后期处理阶段给我们留下了15个特征，可以在表1中看到。一般来说，很少有特征的相关值⇢>0.2，而只有两个，src\_churn和files\_chan- ged，是强相关的p=0.63。虽然相关性很强，但它低于我们的阈值，而且也不是确定的；因此我们没有从数据集中删除这两个特征。所有结果都具有统计学意义（n = 166, 884, p < 0.001）。



4.3 Qualitative data

为了研究拉动请求在实践中是如何使用的，以及为什么一些拉动请求没有被合并，我们对拉动请求的随机样本进行了深入的检查，然后进行编码和主题分析。第一个编码员最初使用了100个拉动请求，以确定关闭拉动请求的离散原因（引导样本），而另一组100个拉动请求则由所有三个编码员使用，以验证所确定的类别（交叉验证样本）。在交叉验证之后，两个数据集被合并，另外150个随机选择的拉动请求被添加到引导样本中，以构建最终分析的数据集，总共有350个拉动请求。

## 5. POPULARITY OF PULL-BASED DEVELOPMENT

截至 2013 年 8 月，Github 报告了超过 700 万个存储库和 400 万用户。然而，并非所有这些项目都处于活跃状态：在 2012 年 2 月至 2013 年 8 月期间，GHTorrent 数据集捕获了由（大约）2,281,000 名用户发起的影响 4,887,500 个存储库的事件。大多数注册的存储库是其他存储库的分支，托管用户网页程序配置文件的特殊存储库和用于评估 Git 的临时存储库。在 GHTorrent 数据集中，不到一半（1,877,660 或 45%）的活动存储库是原始存储库。

在 Github 上打开的所有存储库默认启用拉取请求；然而，并非所有项目都使用它们进行协作。在 2012 年 2 月至 8 月期间，315,522 个原始存储库收到了一次提交。其中，53,866 (17%) 收到了至少一个 pull request，而 54,205 (18%) 使用了共享存储库方法，收到了多个开发人员的提交并且没有 pull request。 2013年同期情况类似；在收到单个提交的 1,157,625 个存储库中，120,104 (10%) 个存储库收到了拉取请求，而 124,316 (11%) 个存储库专门使用共享存储库方法。在这两年中，14% 的活动存储库使用拉取请求。虽然拉取请求的使用总体上在增加，部分反映了 Github 的增长，但使用拉取请求模型的存储库的相对数量略有减少。几乎相同数量的项目使用拉取请求和共享存储库进行分布式协作。

对于 2013 年收到拉取请求的项目，每个项目的拉取请求平均数相对较低，为 8.1（中位数：2，百分位数：5%：1，95%：21）；然而，项目中拉取请求数量的分布是高度倾斜的。存在的项目，例如 Ruby on Rails 和 Homebrew 包管理器，具有超过 5,000 个拉取请求。从 2013 年开放的拉取请求中，73.07% 已经使用 Github 工具合并，从而表明拉取请求原则上可以作为获取外部贡献的手段。此外，尽管人们可能认为收到最多拉取请求的是知名项目，但我们的数据仅适度支持这一点：项目的明星数量与其拉取请求数量之间的 Spearman 等级相关性收到的是p= 0.36 (p < 0.001, n = 239, 131)。

对拉取请求的审查可以针对整个拉取或单个提交，从而类似于代码审查。平均而言，每个拉取请求会收到 2.89（分位数：5%：0，95%：11，中位数：1）讨论和代码审查评论。即使任何 Github 用户都可以参与审查过程，但通常是项目社区成员这样做：只有 0.011% 的拉取请求评论来自尚未提交到项目存储库的用户。在 2012 年收到拉取请求的项目中，35% 还在 Github 问题跟踪器上收到了错误报告（不是基于拉取请求），这表明项目和项目社区都大量使用 Github 的协作工具。

RQ1: 14%的存储库在Github上使用拉动请求。拉动请求和共享仓库在项目中的使用情况相同。尽管使用拉动请求的仓库比例略有下降，但拉动请求的使用在绝对数量上却在增加。

6. PULL REQUEST LIFECYCLE

**Lifetime of pull requests:** 在提交之后，拉动请求可以处于两种状态：合并或关闭（因此不合并）。在我们的数据集中，大多数拉动请求（84.73%）最终都被合并了。这一结果高于我们为Github计算的总体结果；我们将此归因于这样一个事实，即除了Github设施发生的合并之外，数据集的生成过程还采用了启发式方法来检测合并情况。

对于合并的拉动请求，一个重要的属性是处理和合并它们所需的时间。合并的时间分布是高度倾斜的，绝大部分的合并发生得非常快。以天为单位，95%的拉动请求在26天内被合并，90%在10天内被合并，80%在3.7天内被合并。30%的拉动请求在一小时内被合并；大多数这样的拉动请求（60%）来自社区，而他们的源代码流失率明显低于主要团队成员的拉动请求（中位数：5行和13行）。如果我们比较正在合并的拉动请求和未合并的拉动请求的处理时间，我们可以看到，已经合并的拉动请求比未合并的拉动请求（中位数：434分钟）关闭得更快（中位数：2250分钟）。非成对的Mann-Witney测试（P<0.001）结果表明，这种差异在统计上是有意义的，其影响大小适中（Cliff's : 0.32）。这意味着拉动请求要么被快速处理，要么在被关闭之前被长时间搁置。

基于这些观察，我们检查了拉动请求是否比来自外部贡献者的请求处理得更快。为了回答这个问题，我们对每个组的拉动请求的合并时间进行了非配对的Mann-Witney测试。结果是，虽然两组的差异有统计学意义（n1 = 51, 829, n2 = 89, 454, p < 0.001），但明显的差异是可以忽略的（Cliff's : 0.09）。这意味着合并后的拉动请求没有得到任何特殊待遇，无论它们是来自核心团队成员还是来自社区。

在每个项目的基础上，如果我们计算合并一个拉动请求的平均时间，我们看到在大多数项目中（65%），合并一个拉动请求的平均时间少于7天。合并的平均时间与项目的规模（p= -0.05）和项目的测试覆盖率（p= 0.27）没有关联。然而，它与贡献者的记录密切相关（p= -0.69）：开发人员向同一项目提交的拉动请求越多，处理每个请求的时间就越短。此外，项目并没有因为处理更多的拉动请求而变得更快；合并的平均时间与项目收到的拉动请求的数量之间的相关性很弱（p= -0.23, n = 291, p < 0.01）。

**Sizes of pull requests.**一个拉动请求将一组提交捆绑在一起；一个拉动请求的提交数量通常少于10个（95%的百分位数：12，90%的百分位数：6，80%的百分位数：3），中位数为1。拉动请求所改变的文件数量一般少于20个（95%百分位数：36，90%百分位数：17，80%百分位数：7），中位数为2。拉动请求所改变的总行数平均低于500（95%百分位数：1227，90%百分位数：497，80%百分位数：168），中位数为20。

**Tests and pull requests.** 除了项目的源代码，拉动请求也会修改测试代码。在我们的样本中，33%的拉动请求包括对测试代码的修改，而4%只修改了测试代码。在包括修改测试代码的拉动请求中，83%被合并，这与平均水平相似。这似乎与Pham等人[26]的研究结果相悖，受访的开发者认为拉动请求中测试的存在是他们接受的主要因素。拉动请求中是否有测试似乎并不影响合并时间：非配对的Mann-Witney测试显示，虽然包含测试的拉动请求的合并时间的平均值存在差异（tests=45，488，no\_tests=95，980，p<0.001），但影响非常小（sigma=0.11）。

**Discussion and code review.** 一旦一个拉动请求被提交，它就会被公开讨论，直到它被合并或关闭。讨论通常是简短的。95%的拉取请求收到12条或更少的评论（80%少于4条评论）。同样，参与讨论的人数也很低（95%的拉动请求是由少于4个人讨论的）。讨论中的评论数量与合并拉动请求的时间（p= 0.48, n = 141,468）和关闭非合并拉动请求的时间（p= 0.37, n = 25,416）适度相关。

代码审查被整合到拉动请求过程中。虽然拉动请求的讨论可以被认为是一种隐性的代码审查形式，但在我们的样本中，有12%的拉动请求也经过了显性的代码审查，即在包括提交的源代码行上收到评论。代码审查似乎并没有增加拉动请求被合并的概率（84%的审查过的拉动请求被合并），但它们确实减缓了拉动请求的处理：在合并审查过的（中位数：2719分钟）和未审查过的（中位数：295分钟）拉动请求所需时间之间的非配对Mann-Witney测试给出了统计上的显著差异，影响大小（= 0.41）。采用代码审查的项目比不采用的项目具有更大的代码库和更大的团队规模。

任何Github用户都可以参与任何拉动请求的讨论。通常，讨论发生在试图理解拉动请求所带来的变化的核心团队成员和解释它的社区成员（通常是拉动请求的创建者）之间。在大多数项目中，一半以上的参与者是社区成员。然而，评论的数量并非如此；在大多数项目中，大部分评论来自于核心团队成员。人们可能会认为，拉动请求的外部评论者的比例越大，项目就越开放，因此外部贡献的比例就越高；Spearman测试表明，这不是真的（p= 0.22,n = 291,p < 0.05）。

RQ2：大多数拉动请求的长度小于20行，并且在不到1天的时间内得到处理（合并或丢弃）。讨论平均跨越了3个评论，而代码审查影响了合并拉动请求的时间。包含测试代码并不影响合并拉动请求的时间和决定。无论拉动请求是来自贡献者还是核心团队，都不会受到特殊对待。

7. FACTORS INFLUENCING MERGING AND MERGE TIME

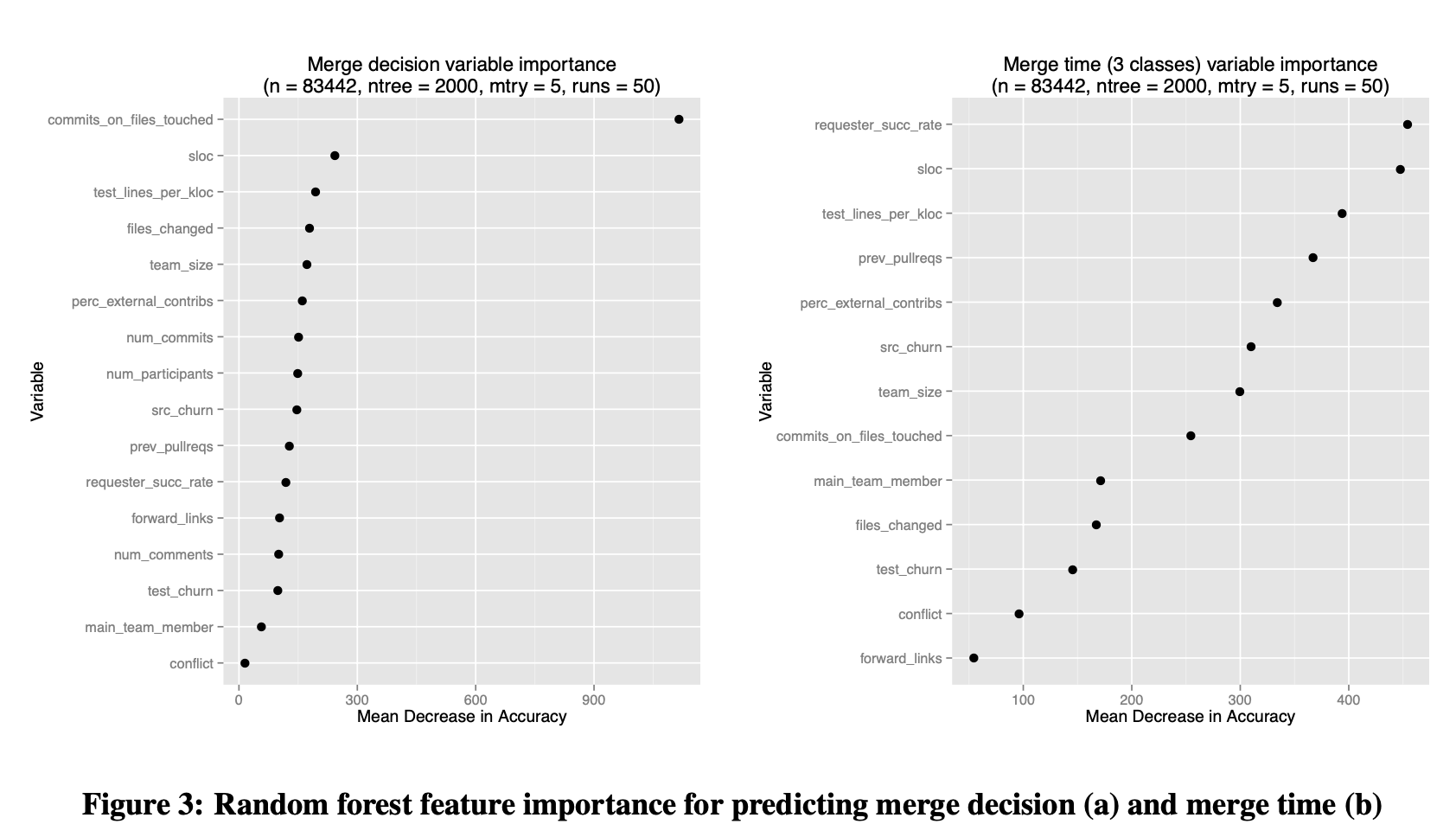
为了了解哪些因素会影响合并的决定以及做出这一决定所需的时间，我们根据第3节中规定的方法运行分类程序。每个分类器都试图根据表1中的特征来预测因变量（合并决定，合并时间类）。对于合并时间实验，我们排除了num\_comments、num\_commits和num\_participants的特征，因为它们无法在pull request到达时测量。基于表2中的结果，我们为两个实验选择了随机森林分类算法。对于合并时间实验，Randomforest的AUC为0.73，对于每一个小时、一天和超过一天的类别，其先验概率分别为31%、35%和34%。对于mergedeci-sion实验，主导类的先验概率为84%，这使得该算法获得了接近完美的分数。在这两种情况下，AUC指标在不同褶皱中的稳定性都很好（mergetime：AUC = 0.019，mergedecision：AUC = 0.008）。

为了提取对每个分类任务都很重要的特征，我们采用了Genuer等人[12]建议的程序。具体来说，我们在随机选择的n=83442个项目的样本上运行了50次算法，使用了大量的生成树（2000个），并在每次分割时尝试5个随机变量。然后，我们使用随机森林算法的R实现所报告的50次运行的平均准确率下降指标，来评估每个特征的重要性。结果可以在图3中看到。

最后，为了验证我们的特征选择，我们重新运行了10-fold交叉验证过程，从每个案例中最重要的一个开始增加预测特征的数量。在每个迭代步骤中，我们将下一个最重要的特征添加到模型中。当每个任务的平均AUC指标与表2中的数值相差不超过2%时，我们停止该过程。所选的特征集应该足以以合理的精度预测分类结果，因此可以被描述为重要的特征[12]。

对于合并决策任务，特征的重要性结果被commits\_on\_files\_touched特征所主导。通过重新运行交叉验证过程，我们得出结论，使用commits\_on\_files\_touched、sloc和files\_changed特征来预测一个拉动请求是否会被合并就足够了（AUC: 0.94, ACC: 0.86）。因此，我们可以得出结论，合并一个拉动请求的决定受到以下因素的影响：它是否触及了系统的急性开发部分（"昨天的天气 "假说的一个变种），项目的源代码库有多大，拉动请求改变了多少文件。

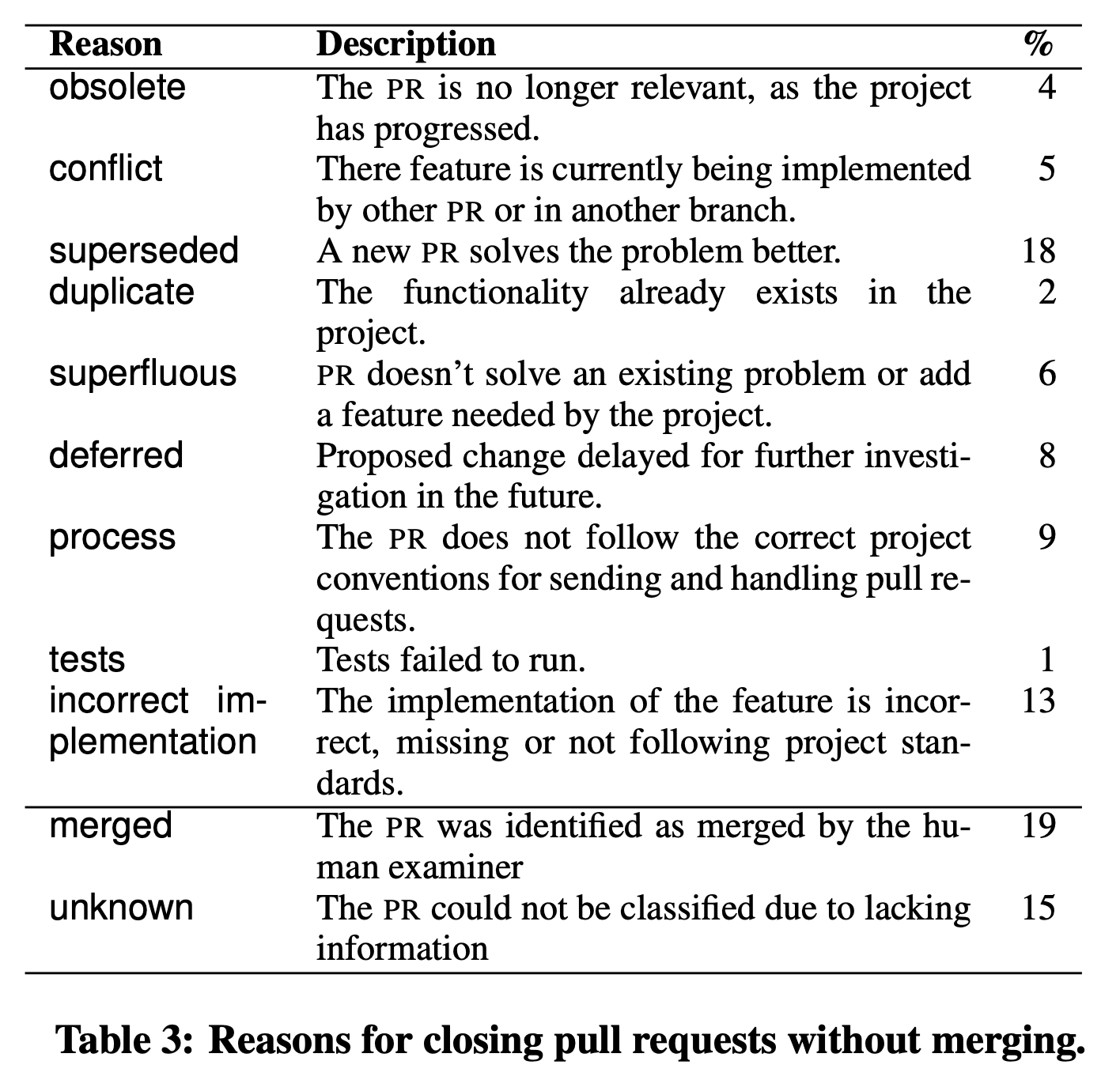
对于合并时间的任务，没有主导的特征；分类模型的重新运行显示，至少需要6个特征来预测一个拉动请求的合并速度。分类精度适中（AUC：0.74，ACC：0.59），但仍比随机选择有所提高。这些结果证明，设计者以前的记录、项目的规模和测试覆盖率以及项目对外部贡献的开放性似乎对拉动请求被接受的速度起着重要作用。



*RQ3: 合并拉动请求的决定主要受拉动请求是否修改了最近修改的代码的影响。合并的时间受开发者以前的记录、项目的规模和测试覆盖率以及项目对外部贡献的开放程度的影响。*

# 8. UNMERGED PULL REQUESTS

由于大多数拉动请求确实被合并了，探讨一下为什么有些拉动请求没有被合并是很有意思的。出于这个原因，我们手动调查了350个拉动请求，并按照第4.3节中描述的原因进行了分类。在一组不同的拉动请求上进行的交叉验证表明，所确定的类别足以对关闭拉动请求的所有原因进行分类，即使编码者之间存在差异。结果列于表3。



结果显示，没有明显突出的关闭拉动请求的原因。然而，如果我们把具有时间维度的关闭原因（过时、冲突、被取代obsolete, conflict, superseded ）归纳在一起，我们看到27%的未合并的拉动请求是由于项目分支中的代码被修改而关闭的。另外16%（多余的、重复的、推迟的(superfluous, duplicate, deferred) ）被关闭的原因是贡献者没有正确识别项目的方向，因此提交了不感兴趣的修改。10%的贡献被拒绝，原因是与项目过程和质量要求（过程，测试）有关；这可能是一个过程没有被很好地沟通或严格的代码审查过程的指标。最后，还有13%的拉动请求被拒绝，因为代码审查发现了实现中的错误。

此外，对于15%的拉动请求，人工审查员无法确定不合并的原因。这通常意味着在关闭拉动请求之前没有进行讨论，或者拉动请求是由外部工具自动发起和管理的；顺便说一下，在我们的随机样本中，除了一个项目有这样的拉动请求，其他项目都没有使用Github的问题跟踪设施。因此，这些项目对拉动请求的使用可能是肤浅的，只是对由其他工具管理的流程的部分补充。最后，即使自动启发式方法不能识别，人类审查员也能识别19%的拉动请求为合并请求；这意味着本研究报告中的合并拉动请求比例（84%）可能被略微低估了，因为非包容性启发式方法。通过推断，合并的拉动请求的总数可能高达90%。

值得注意的是，只有13%的贡献是由于技术问题而被拒绝的，这也是代码审查的主要原因，而总共有53%的贡献是由于与拉动请求过程的分布式性质（当前修改）或项目处理项目目标和实践沟通的方式有关的原因而被拒绝的。这可能意味着基于拉动的模式（或者至少是Github实现它的方式）对于项目的核心团队来说可能是透明的[10]，但对于潜在的贡献者来说却不是那么回事。人类审查员即使在人工审查后也无法理解拉动请求被拒绝的原因，这一事实进一步支持了这一假设。

RQ4：53%的拉动请求被拒绝的原因与基于拉动开发的分布式性质有关。只有13%的拉动请求是由于技术原因而被拒绝的。

# 9. DISCUSSION

## 9.1 The pull-based development model

**Development turnover.** 拉动请求模式的承诺之一是快速开发周转，即从拉动请求的次级任务到它被项目主库接受的时间。在对Apache和Mozilla等项目的补丁提交过程的各种研究中，研究人员发现，将50%的贡献提交到项目主库的时间从几个小时到不到3天不等。我们的研究结果显示，大多数（80%）的拉动请求在4天内被合并，60%在一天内被合并，而30%在一小时内被合并（与项目规模无关）。这些数字表明，通过拉动请求的拉动式开发可能比传统的基于电子邮件的补丁更有效率。另外，影响周转时间的是项目相关因素，而不是拉动请求本身的特征。这意味着主要是由项目来调整其流程（特别是测试范围和流程的开放性），以加快周转。

**Managing pull requests**. Dabbish等人[10]的受访者认为管理拉动请求是最重要的项目活动。Dabbish等人提到，项目经理 "根据代码的风格、效率、彻底性（例如，是否包含测试）和提交者的记录来推断代码的质量"。项目经理提到的一些激励点（测试拉动请求中的代码，跟踪记录）也被包括在我们的分类模型中，但它们似乎对合并决策过程影响不大。然而，开发人员的跟踪记录对处理拉动请求的速度很重要。此外，我们发现，从被拒绝的拉动请求中，几乎有53%是由于基于拉动开发的分布式性质而被拒绝的。虽然从项目经理的角度来看，拉动请求的过程是透明的（Dabbish等人的访谈对象也对此表示称赞），但我们的发现表明，从潜在贡献者的角度来看，它并不透明。

**Attracting contributions** .Pham等人[26]提到，pull requests通过一种通常被称为 "驱动式提交 "的机制，使临时贡献变得简单。由于在Github上分叉一个软件库的相对成本可以忽略不计（54%的软件库是分叉的），开发者分叉其他软件库来进行随意提交的情况并不少见。这种提交可以被识别为包含来自尚未加入项目社区的用户的单一提交的拉动请求，占2012年拉动请求总数的7%。此外，3.5%的分叉创建的唯一目的是为了创建一个开车提交。为了准确定义和评估偷渡提交的影响，我们还需要做更多的工作。

**Crowd sourcing the code review.** 对一个开源项目的贡献过程的一个重要部分是对所提供的代码进行审查。Rigby和German[29]报告说，80%的核心团队成员也参与了补丁的代码审查，这个数字也与Mockus等人[23]的早期发现相一致。在我们的数据集中，我们发现所有项目的核心团队成员都至少参与了一次拉动请求的讨论。此外，我们发现，在我们的数据集中的所有项目中，讨论拉动请求的社区实际上比核心团队成员更大。

**Democratizing development** **.**这项工作的关键发现之一是，拉动请求并没有因为其来源而被区别对待；核心团队成员和外部开发者在相同的时间范围内都有同等的机会让他们的拉动请求被接受。事实上，即使是我们建立的分类模型也对相应的功能赋予了低重要性。在我们看来，这是对开源开发方式的彻底改变。在拉动请求之前，大多数项目都采用了成员资格晋升策略[18]，以促进感兴趣的第三方开发者进入核心团队。有了拉动请求，开发者可以向任何资源库做出贡献，而不会失去作者的信息。通过拉动请求，这些贡献被接受的几率更高；在Github，超过70%的外部贡献被合并（其他研究中为40%[29, 32]）。特别的网站，如Ohloh和CoderWall，追踪开发者的活动，并帮助开发者宣传他们的专业知识。我们相信，开发工作的民主化将导致一个更强大的共享生态系统；这还有待进一步研究验证。

## 9.2 Implications

**Contributors**.潜在的项目贡献者希望他们的贡献能被接受。我们的研究表明，那些影响到项目中最近经常被修改的部分（即 "热门"）的拉动请求很可能会被合并。此外，80%的合并拉动请求修改了三个或更少的文件，并且包括长度在100行以下的补丁。因此，我们对寻求增加一个特定功能或修复一个错误的贡献者的建议是 "保持简短"。如果贡献者的目的是为了让项目社区认识他，那么影响一个热门的项目领域也是有益的。

**Core team.**核心团队的主要工作是评估拉动请求列表并决定是否应用它们。为了确保拉动请求被及时处理，明显的策略是投资于一个全面的测试套件，并确保项目过程足够公开和透明。为了避免与开发并发有关的问题，核心团队成员可以要求contributors通过打开一个问题来传达他们的缩进变化，然后用代码来增加，并转化为拉动请求。项目应该在项目的Github页面的显著位置清楚地说明对拉动请求的期望，例如测试或本地化的变化。

我们结果的一个直接应用是构建工具来帮助核心团队确定工作的优先次序；由于我们可以非常准确地预测一个拉动请求是否会被合并，一个潜在的工具可能会建议哪些拉动请求在到达时可以被合并而无需进一步检查。其他工具可能会在贡献者的网站上检查拉动请求的质量，并基于项目的概况，提供自动化的改进建议（例如，更多的测试、文档）.

# 11. CONCLUSION

这项工作的目标是深入了解基于拉动的软件开发模式，该模式用于Github上许多重要的开源项目。为此，我们对数以百万计的拉动请求进行了统计分析，并对来自积极使用拉动模式的项目的数十万个拉动请求进行了仔细的组合。

我们的主要发现有以下几点:

1. 基于拉动的模式并不像我们预想的那样流行。只有14%的活跃项目使用拉动请求，但这个数字与使用共享资源库方法的项目数量相当（第5节）。

2.大多数拉动请求只影响几十行代码，60%的拉动请求在不到一天的时间内被处理（合并或丢弃）。合并决定不受测试代码存在的影响。核心成员和外部开发者有同等的机会使他们的拉动请求被接受（第6节）。

3. 合并的决定主要受拉动请求是否修改了最近修改的代码的影响。合并的时间受到各种因素的影响，包括开发者的记录，以及项目的测试覆盖率。

4. 53%的非合并拉动请求被拒绝，原因与基于拉动的开发的分布式性质有关。只有13%的拉动请求是由于技术原因而被拒绝。

我们的发现有以下意义:

1. 基于拉动的模式要求我们修改目前对开源软件发展的一些理解。有趣的研究方向可能包括团队和管理层次的形成、新的代码审查实践以及开发者在高度透明的工作空间工作的动机。

2.寻求吸引外部贡献者和加快贡献合并的团队，不仅可以通过提供明确的拉动请求处理指南，还可以通过纳入高覆盖率的测试套件来实现。

3.任务阐述不充分似乎是造成工作浪费（未合并）的最重要原因。设计新的方法将任务协调纳入基于拉动的模型是一个有希望的进一步研究领域。

最后但同样重要的是，我们的数据集提供了丰富的开源软件开发信息。数据集以及定制的Ruby和R分析工具都可以在Github仓库gousiosg/pullreqs上找到，同时还有如何使用它们的说明。