Accepted: 17 December 2025
Revised: 17 December 2025
Received: 21 March 2025

SURVEY

Hot Fixing Software: A Comprehensive Review of Terminology, Techniques, and Applications

CAROL HANNA, University College London, London, U.K.

DAVID CLARK, University College London, London, U.K.

FEDERICA SARRO, University College London, London, U.K.

JUSTYNA PETKE, University College London, London, U.K.

Hot Fixing Software: A Comprehensive Review of Terminology, Techniques, and Applications

CAROL HANNA, University College London, United Kingdom
DAVID CLARK, University College London, United Kingdom
FEDERICA SARRO, University College London, United Kingdom
JUSTYNA PETKE, University College London, United Kingdom

A hot fix is an unplanned improvement to a specific time-critical issue deployed to a software system in production. While hot fixing is an essential and common activity in software maintenance, it has never been surveyed as a research activity. Thus, such a review is long overdue. In this paper, we conduct a comprehensive literature review of work on hot fixing. We highlight the fields where this topic has been addressed, inconsistencies we identified in the terminology, gaps in the literature, and directions for future work. Our search concluded with 140 articles on the topic between the years 1986 and 2024. The articles found encompass many different research areas such as log analysis, runtime patching (also known as hot patching), and automated repair, as well as various application domains such as security, mobile, and video games. We find that many directions can take hot fix research forward such as unifying existing terminology, establishing a benchmark set of hot fixes, researching costs and frequency of hot fixes, and researching the possibility of end-to-end automation of detection, mitigation, and deployment. We discuss these avenues in detail to inspire the community to systematize hot fixing as a software engineering activity.

CCS Concepts: • **Software and its engineering → Software maintenance tools**; **Error handling and recovery**; **Software testing and debugging**.

Additional Key Words and Phrases: Literature review, hot fix, hot patch

1 Introduction

Software maintenance is an essential activity in the software engineering life cycle [105]. After a software system is deployed, a lot of engineering effort is directed to maintain it. The maintenance activities ensure that both the functional and non-functional requirements of the system are upheld. These activities are especially important because ensuring that a system is correct under all possible conditions before it is deployed, is generally not feasible in practice. This is due to software testing being necessarily incomplete [34] as well as the tight release deadlines in modern enterprises [10, 11, 150]. Critical issues in production software incure very high costs to an enterprise. One such example is when Amazon had one hour of downtime on Prime Day which was reported to have cost it up to $100 million in lost sales [65]. Another example is when Google's search and maps availability was affected after a software update [48].

Not all of the issues in production will have the same priority. While most maintenance activities involve system upgrades and patches to improve the underlying software, some of the issues are critical and as such they require

Authors' Contact Information: Carol Hanna, carol.hanna.21@ucl.ac.uk, University College London, London, United Kingdom; David Clark, david.clark@ucl.ac.uk, University College London, London, United Kingdom; Federica Sarro, f.sarro@ucl.ac.uk, University College London, London, United Kingdom; Justyna Petke, j.petke@ucl.ac.uk, University College London, London, United Kingdom.

ACM 1557-7392/2025/12-ART
https://doi.org/10.1145/3786330

quick remediation. These critical issues are often ones that cause severe degradation in performance, security vulnerabilities, or serious functional defects visible to the end user. Patching these critical issues is often referred to as "hot fixing".

A traditional good fix for a software issue is expected to be correct, not break existing functionality, and not increase the technical debt of the system. However, in the case of hot fixes these criteria do not necessarily apply. Hot fixes need to remediate the unwanted symptoms of the critical issue as soon as possible [136]. In this case, a quick temporary fix is favored over a slower permanent fix that preserves the criteria [27]. There is less emphasis on correctness under all conditions and more on the time it takes to generate a plausible patch that hides the critical symptom without breaking the system. For these reasons, hot fixes for critical issues differ drastically from general patches.

The time criticality and temporary nature of hot fixing make the process of detecting when such fixes need to be applied, generating them, and deploying them not as systematic as other activities within software engineering. These are the issues that stakeholders are only concerned with at the time when it is urgently required. This is manifested in industrial practices and research activities on hot fixing, where the work is less established and the terminology is often inconsistent.

In this paper, we aim to aid in understanding the existing collective knowledge on hot fixing and in driving research in the area forward. We have used a rigorous search to conduct the first-ever comprehensive literature review on the topic. Such a review is long overdue. This paper makes the following contributions:

(1) A unified definition for *hot fix* to align the terminology used in research on the topic, while keeping consistent with the existing body of work (Definition 1).
(2) A comprehensive survey of the literature on hot fixing that encapsulates terminology (Section 2), publication trends (Section 7), techniques and applications (Section 9).
(3) A detailed research agenda with the current open challenges in the area of hot fixing (Section 12).

We hope that these contributions will help drive research in the area of hot fixing forward. Our website with a list of the included publications is available at https://carolhanna01.github.io/hotfixes.github.io/.

The rest of this paper is organised as follows: Section 2 details the terminology used in the literature, Section 4 outlines the scope of this study, Section 5 presents the research questions that we pose, Section 6 explains the methodology that we followed to conduct the literature review, Section 7 presents publication trends on the topic, Section 8 presents the current hot fixing practices published on hot fixing, Section 9 describes the techniques and applications, and finally we conclude with a discussion in Section 12, and threats to validity and conclusions in Sections 13 and 14.

2 Terminology

The term *hot fix* has had different usages in the literature [55]. Therefore, we first provide a brief history of the evolution of the term and associated definitions found in previous work [55], before presenting a unified definition to streamline future research in the area.

Definitions in the research literature on hot fixing can be divided into two main categories. The first regards a hot fix as a fix that needs to be deployed into a system in production at runtime without having to restart or reboot the system [13] [171]. This definition is less common as the term more often used for this is "hot patch". The second definition focuses on the criticality element of a hot fix. In this regard, a hot fix is a time-critical fix that is temporary, small in size, and targets a specific issue in a system in production [51] [7]. Some domain-specific definitions exist as well in the fields of operating systems and information retrieval [53] [103].

Which of these definitions came first remains unclear. However, the literature seems to suggest that the word *hot* in the term signified the "liveness" of the system into which the patch was being deployed. Thus, it is most likely that hot fixing was meant to describe the dynamic deployment of fixes into systems at run time (note that some work in the literature refer to this as hot patching). From there, developers responsible for hot fixing activities began to evolve the term's meaning in other directions. As a developer, when a fix needs to be deployed at run-time it must be an important and time-critical change, otherwise it would simply be deployed within the planned software release cycle. This interpretation with respect to timing has created two distinct definitions of the term, separating the literature into two fields. While there can be an overlap between them as hot patches can sometimes be hot fixes and vice versa, the cross-pollination between the two communities remains limited. An important point is that most commonly the term *hot fix* was associated with the time-criticality definition and *hot patch* with the run-time definition. We wish to follow the existing body of work and thus propose that the community use the term *hot fix* for the time-criticality definition and *hot patch* for the run-time definition moving forward.

The concept of time-criticality is nuanced and depends on the context. Some work considers the time-criticality of an issue to be indicated by its severity, priority, and even how often it is reopened [33]. This might be subjective in some instances. A time-critical issue might not be a breaking change necessarily but an issue that affects a specific important customer of the enterprise for example. Thus, we leave the time-criticality part of our definition open to be *templated*, based on the business needs. Instead of prescribing a fixed definition, the idea is to provide a flexible framework that can be adapted per company/project based on the context. For example, one company might define time-critical issues based on financial impact, while another might prioritize issues affecting regulatory compliance. By making this aspect of the definition "templated", the paper allows for a structured yet adaptable approach to defining time-critical bugs.

Hot fixing as a software engineering activity usually falls outside the traditional software engineering life cycle. As previously explained, this is due to the time constraint, which results in quick development of workarounds, skipping extensive testing, and having deployment not wait until the next scheduled release. From here, we can see that the criticality of the hot fix usually results in some form of exceptionality in its development process.

Hot patch refers to the runtime software patching activity, and is defined as follows by Islam et al. [66]: A *hot patch*, also referred to as a *runtime software patch*, "aims to update a given software system while preserving running processes and sessions" [66].

From here, one can consider a hot fix to be a *phenomenon* in software engineering. This phenomenon usually breaks the traditional software engineering life cycle to address emergency issues in the system. As for hot patching, this is more of a software *technique* or a set of techniques that relate to runtime patching. Thus, the two terms *hot patch* and *hot fix* cannot be directly compared and should also not be used interchangeably.

To streamline the literature on the topic, we propose to proceed with the following unified definition for the term *hot fix* [54]:

DEF. 1. *A* ***hot fix*** *is an unplanned improvement to a specific time-critical issue deployed to a software system in production.*

An unplanned improvement is a change initiated reactively in response to a disruptive production issue, rather than as part of a planned development effort. Such changes arise from incidents whose severity or impact creates a narrow window for mitigation, including crashes, vulnerabilities, outages, or urgent customer-facing failures. Although the exact duration of this window may vary across contexts due to factors such as financial risk, regulatory requirements, or service-level obligations, the defining characteristic of a hot fix is that it cannot be safely deferred and therefore demands rapid development and immediate deployment once a remedy is available. Further operational guidelines and illustrative examples, derived from the analysis/study of the literature, are provided in the terminology discussion (Section 12.1).

3 Motivation

Hot fixing is one of the most visible yet least systematically understood phenomena in software engineering. Unlike planned updates, hot fixes occur under urgent conditions when failures cannot wait for the next release cycle. They represent a class of interventions where stability, user trust, and business reputation are dependent on the ability to quickly diagnose and repair critical issues in production. Across domains, from gaming to cloud services to mobile ecosystems, hot fixes emerge as a recurring response to unexpected failures.

The gaming industry offers a vivid example of this urgency. Lin et al. provide one of the most striking demonstrations of the prevalence of hot fixing in practice through their study of urgent updates in the gaming industry [82]. Analyzing the most popular games on the Steam platform, they found that 80% required urgent updates, and nearly half of these were explicitly described by developers as hot fixes. These updates were not planned enhancements but emergency repairs applied under pressure to address crashes, gameplay malfunctions, or balance issues. Their study shows how hot fixing becomes a core part of sustaining user experience in a competitive, large-scale environment, and highlights the stress and improvisation inherent in these unplanned releases.

A similar story unfolds in the mobile ecosystem. Hassan et al. analyzed over 10,000 mobile apps and found more than 1,000 emergency updates, many triggered by seemingly simple but disruptive mistakes such as resource misconfigurations [56]. Despite their reactive nature, these fixes often endured longer than planned releases, lasting on average more than twice as long. At the same time, developers under pressure documented little about their rationale, and nothing at all in 63% of cases. These findings reveal the dual reality of hot fixes: they are rushed and improvised, yet they become permanent fixtures that shape software evolution. This underscores the importance in understanding them as a software engineering community.

Beyond examples from the literature, let us consider real-world issue reports that capture hot fixing in action. The Jira Align's documentation from the project Summer[1] explains that a hot fix was applied outside the normal release schedule to address an urgent issue affecting the product mappings UI [6]. The patch was deployed to bundled release environments on the evening of Monday, August 1st, 2025 and then integrated into the continuous-release track just a few days later, on Friday, August 5th, 2025. Such instances are common in enterprise software and highlight how urgently hot fixes must be deployed, bypassing planned workflows to safeguard production systems.

Hot fixing is not a marginal practice but a recurring, high-stakes necessity. Hot fixes highlight the tension between urgency and stability, improvisation and permanence, and they make clear why a systematic survey is needed: To unify terminology, consolidate practices, and map the tooling that supports this essential yet understudied phenomenon.

4 Survey Scope

This is the first survey dedicated to hot fixing in software systems. Hot fixing, as per Definition 1, refers to time-critical fixes applied in production under urgent conditions.

4.1 Relation to Prior Surveys

Islam et al. recently conducted a comprehensive study on runtime software patching, which addresses current gaps in the literature and provides insights into future directions [66]. Their survey details the state of the art in runtime patching with a scope limited to the deployment phase, i.e., the application of a patch into the end system. More specifically, their focus is on dynamic/runtime patching techniques for this deployment step and do not address critical patch fixing as a broader software engineering activity. Instead, they examine runtime patching as a technical mechanism. Since this survey covers prior work related to what we term the hot patch category of definitions (see Section 2), we exclude this line of work from our scope and instead focus specifically on hot

[1]https://www.atlassian.com/software/jira-align

fixing as defined in Definition 1. The overlap between our study and Islam et al.'s [66] survey is therefore minimal and covers only 8 papers out of 140. Since we explicitly exclude runtime patching and dynamic updating papers from being considered as standalone categories, they are not examined separately in our survey and thus we find that we have no overlap with other surveys on runtime patching and dynamic software updating [8, 64, 99].

4.2 Inclusion Criteria

Although our survey focuses specifically on hot fixing as per Definition 1, we do not strictly exclude papers that are not explicitly about hot fixing. If a study provides valuable insights into hot fixing processes or presents a tool that can be leveraged for hot fixing purposes, we consider it within scope. This allows us to capture relevant research that, while not directly framed as hot fixing, contributes to the broader understanding and practical implementation of hot fixing techniques. This includes papers that mention emergency handling, detection aimed at critical bugs, efficient patch and workaround generation and fast deployment.

It is important to notice that our scope can still include some articles that use runtime patching, but only if the runtime patching technique specifically addresses **critical software issues**. In that case, since the criticality property is met, the software activity would still be considered hot fixing and the technique used for it would be considered runtime patching. As such, articles of this kind are within our scope. Moreover, studies on monitoring the fix post-deployment are excluded from our scope. This is because monitoring for post-deployment issues is not specifically related to hot fixing. For monitoring, the context in which code was integrated into production is less relevant.

We summarize the scope of our survey as follows:

Scope: Previous work is in the scope of this survey if it

1. investigates the **detection** of critical software issues that hot fixes target; OR
2. investigates the **repair** of critical software issues through the generation of hot fixes; OR
3. investigates the **deployment** of hot fixes; OR
4. empirically analyzes hot fixes in software systems.

Included "detection" studies must:

- **Focus on criticality:** They should detect software issues that, by their nature, demand urgent remediation due to their impact or potential system compromise.
- **Relate to hot fixability:** The detection approach should explicitly contribute to identifying issues where a hot fix is a plausible next step.
- **Enable immediate remediation:** Studies should be specific to methods that drive towards urgent, corrective patching—such as techniques that work alongside remediation tools or integrate with hot fix deployment workflows.

We exclude papers on project management unless they provide empirical insights or introduce tooling relevant to hot fixing. Our focus is on studies that contribute concrete evidence, methodologies, or practical implementations rather than purely conceptual discussions on software project management.

5 Research Questions

Hot fixing software is an essential software engineering activity. In balancing the tradeoff between the cost of expensive testing and the cost of repair for production bugs, hot fixes are often unavoidable. The past, present, and future for hot fixing software remain unclear. Thus, in this paper we aim to compile the collective knowledge on the topic. We ask:

RQ1: [General Practices] What are the general practices for hot fixing?
The software engineering lifecycle encompasses various steps. Software engineering research focuses on optimising the various steps within this lifecycle so that software development processes can be more productive and software quality can be improved. Hot fixing is a software engineering step that often breaks this traditional lifecycle as it occurs as a result of a sudden unexpected critical isssue. As of now, there are no studies that compile the practices for hot fixing software. We aim to present the empirical work on hot fixing in the body of literature to shed light on past and current general practices for hot fixing.

- **RQ1.1** (Open-source Development): What are the general practices for hot fixing in open-source software projects?
- **RQ1.2** (Commercial development): What are the general practices for hot fixing in commercial software development?
- **RQ1.3** (Human Research): What are the perceptions of software practitioners about hot fixing practices?

RQ2: [Automation and Tooling] What are the existing tools and automated processes used for hot fixing software, and how do they impact the efficiency and reliability of the hot fix deployment process?
Automation is utilized to design software, develop it, and maintain it. Since hot fixing is a time-critical activity, there is great benefit in having tooling to aid the software developers and make the process less costly and more efficient. We aim to investigate the state-of-the-art in tooling for hot fixing. We hope that with this, researchers will be able to build upon existing tooling to improve automation for hot fixing.

- **RQ2.1** (Human-Assisted Tools): What are the existing helper tools that support practitioners in the hot fixing process?
- **RQ2.2** (Autonomous Tools): What are the existing autonomous tools used for hot fixing in software development?

RQ4: [Characteristics and Domains] What do the collected studies reveal about the characteristics of hot fixes and the domains in which they are most frequently applied?
Hot fixes differ in the types of issues they address (e.g., crashes, vulnerabilities, misconfigurations) and the contexts in which they are deployed (e.g., mobile, embedded, cloud). The characteristics and domains of hot fixes can be inferred from the tools designed to support them. By analyzing the surveyed tools, we identify the types of issues these tools target. This question situates tooling within the broader landscape of hot fixing, clarifying both the nature of the fixes and the environments where they are most relevant.
RQ4: [Open Challenges] What are the key unresolved challenges in the hot fixing process across different software development environments?
Understanding the bottlenecks in the state-of-the-art for hot fixing will ensure that future work in the field tackles the relevant issues. We aim to critically assess the existing literature on hot fixing to be able to identify gaps. These gaps are challenges that future work should be directed towards for achieving improved hot fixing processes and tooling.

6 Survey Methodology

We set a rigorous methodology to ensure a literature review that would result in a comprehensive identification and analysis of existing work on hot fixing. In this section, we detail this methodology and the results derived.

6.1 Primary Search

Table 1 presents the results of our primary search for keywords. We started off with the keyword "hot fix" and subsequently also searched the keyword "hot patch" as we found that sometimes the words *fix* and *patch* are used interchangeably in the literature. Variations of these two keywords including plurals and suffixes were included as well. We conducted our search using four computer science digital libraries: IEEE Xplore [4], ACM Digital

Table 1. Results of primary search for papers on hot fixes.

Keyword	‘hot fix’			Keyword	‘hot patch’		
Source	Filters	Papers found	Relevant papers	Source	Filters	Papers found	Relevant papers
IEEE Xplore	Full text & Metadata	187	27	IEEE Xplore	Full text & Metadata	140	26
ACM Digital Library	Anywhere	200	56	ACM Digital Library	Anywhere	120	25
DBLP Computer Science Bibliography	Default	32	2	DBLP Computer Science Bibliography	Default	26	12
ScienceDirect	Articles with this term	189	9	ScienceDirect	Articles with this term	291	3
Total number of papers		608	94	Total number of papers		577	66
Distinct number of papers			88	Distinct number of papers			56

Total number of distinct papers on hot fixing	**144**
Total in scope papers on hot fixing	**53**

Table 2. Results of snowballing search for papers on hot fixes.

Search step	Total papers found	Relevant papers
Round 1	1519	43
Round 2	1938	30
Round 3	1097	8
Round 4	260	1
Round 5	26	0
Total number of new papers found		**82**

Library [1], DBLP Computer Science Bibliography [2], and ScienceDirect [5]. For each of the search engines, the table presents the date on which the search was conducted, the filters used in the search, the total number of results given these filters, and finally the number of papers that we deemed as relevant. The search was not restricted to a specific time-frame. We looked at all papers published until the search date which is 19/11/2024. The earliest publication we could find was from the year 1986 and the latest was 2024. Thus, the time frame of this survey is 1986-2024. We used Mendeley [97] as a reference manager to collect and organize the publications throughout the search process as well as to de-duplicate the references.

Table 3. Summary of results of literature review on hot fixes.

Search step	Total papers found
Primary	53
Snowballing	82
Suggestions from authors	4
Suggestions from reviewers	1
Papers on hot fixes for software	140

At this stage in the review, we consider a paper to be relevant if it is in the domain of computer science and advances the knowledge on hot fixing software. Using this criterion, we found 88 distinct papers using the keyword "hot fix" and 56 using the keyword "hot patch". The primary search at this stage concluded in a total of 144 relevant distinct papers.

Given these 144 relevant papers, we assessed their content more closely to better understand the scope that they cover. We found that 14 of these papers strictly address hot fixing within the context of project management. Specifically, we found that these papers provide high-level tips for handling instances where hot fixing is required from the perspective of managing the full project. In 19 of the papers, a definition for "hot fix" or "hot patch" is provided but the focus of the paper is not on this topic and thus there is no added novelty in this regard. An additional 8 papers present scenarios for hot fixing a system. However, the scenarios in these papers are provided as mere examples and are not the main subject of the paper. For 25 of these papers, the term is used to refer to the runtime patching of bugs in software while not fulfilling the time-criticality criterium. As this point, it became apparent that there is a clear inconsistency with the terminology. Since runtime patching is not the topic of our survey these 25 papers were excluded as well. Finally, we also removed 24 papers that just motivate the need for hot fixes in the software development life cycle without additional knowledge that would make them core papers on the topic.

6.2 Snowballing Search

At this stage of the literature review, we were left with 53 papers from the primary search which we deem as core papers on the topic. The scope of our study was naturally born at this point which we present in Section 4. Following the primary search, we manually examined the bibliographies of all of the 53 papers that we deemed to fit the scope that we defined. This process of snowballing on all of the papers referenced from each of the papers within the scope that we set was repeated until no more new papers were found. The results of this snowball search are presented in Table 2. For each round of snowballing, we present the total number of papers found as well as the number of new distinct papers that fit within the scope of our survey. Our snowballing process thus concluded after 5 rounds.

We present the final results of our literature review in Table 3. We found 53 papers in the scope through the primary search and 82 additional papers in the snowballing search. We emailed the authors of 87 papers that were identified in the beginning stages of the study to ask for feedback on the first draft of the survey, we received 4 additional relevant suggestions for papers to be included in the survey. Finally, we got one suggestion from a reviewer. Overall, we ended up with 140 core papers which we describe in this survey.

6.3 Thematic Analysis

After the paper collection process, we conducted a thematic analysis [21] as used in qualitative analysis research to categorize the papers found and effectively organize this paper. We followed the six stages of thematic analysis which involved sifting through the papers, finding patterns among the collection, and naming the different categories.

In the first stage, we re-familiarized ourselves with the data. Prior to starting the thematic analysis, the search process itself required being familiar with the data to adequately filter relevant and irrelevant publications. However, after having collected the full corpus we needed to scan the content again and begin to notice patterns that might be relevant to our proposed research questions. We then began generating codes for each of the papers in our corpus. We did this directly within the Mendeley reference manager we were using by using 'tags' feature and adding the codes through tags to each of the papers. Given the tagged papers, we then sifted through and searched for themes keeping in mind our original research questions. This involved clustering codes together into clear themes. We next had a review process for the themes through a second assessor that checked for the quality of the codes and the themes that emerged from them. From here, we were confident to name each theme and begin structuring the answers to our research questions.

The themes found correspond with the sections that we outline to present the answers to RQ1 and RQ2. We initially divided the papers into two themes: general hot fixing practices and papers that present tooling. This allows us to distinguish between research that analyzes how hot fixing is performed and research that contributes practical solutions. General hot fixing practices explore workflows, processes, and challenges in applying hot fixes across different development contexts. These papers provide insights into industry trends, developer behaviour, and the factors influencing patching critical bugs. These papers help us answer RQ1. Tooling papers focus on practical automated or semi-automated solutions for hot fixing as used in industry or academia. These papers help us answer RQ2.

From there, we further refined the type of hot fixing practices depending on the data used: human studies, commercial data, and open-source data. As for the papers that cover tools, we refine them based on the type of tool: human-assisted tools and autonomous tools. Human-assisted tools include ones for system health monitoring, critical bug reporting and triage, and critical bug debugging. Autonomous tools include ones for detection, remediation, or deployment as well as end-to-end tools that cover these 3 phases. We explain this in Section 9 and present the refinement in Figure 4.

7 Publication Trends

The trend of publications on the topic of hot fixing software is plotted in Figure 1. The earliest relevant article on this topic that we were able to find was published in the year 1986. We observe that publications on this topic started gain to popularity after 2015. In more recent years, there has been a constant output of studies until the year 2024. We hypothesize that this is due to the growing complexity of software systems which makes fixing urgent issues in production a difficult task. As such, we can see that more research efforts have slowly started to be directed towards this software engineering activity. We observe a spike in the number of publications in the year 2020 with a total of 19 papers published on the topic that year. We hope that by reviewing the literature and streamlining what has already been done in tackling this topic, we can inspire future research in the area.

We then assess the types of venues that the papers found were published in. Since research on the topic spans multiple different fields as previously explained, we found a large variety of publication venues among the papers. We plot the most popular research areas that the papers pertain to in Figure 2. We found that the most popular were software engineering venues, venues for systems (cloud systems, etc), and venues for software security topics.

Following our thematic analysis, we were able to tag the papers into different categories. Within the papers that include tooling, we analyzed the trend of tooling types and present this in Figure 3. Interestingly, the earlier stages of the hot fixing pipeline had more tooling than the latter stages. We found that the most popular were tools for detecting critical issues, followed by tools for remediation of these critical issues, and finally around half

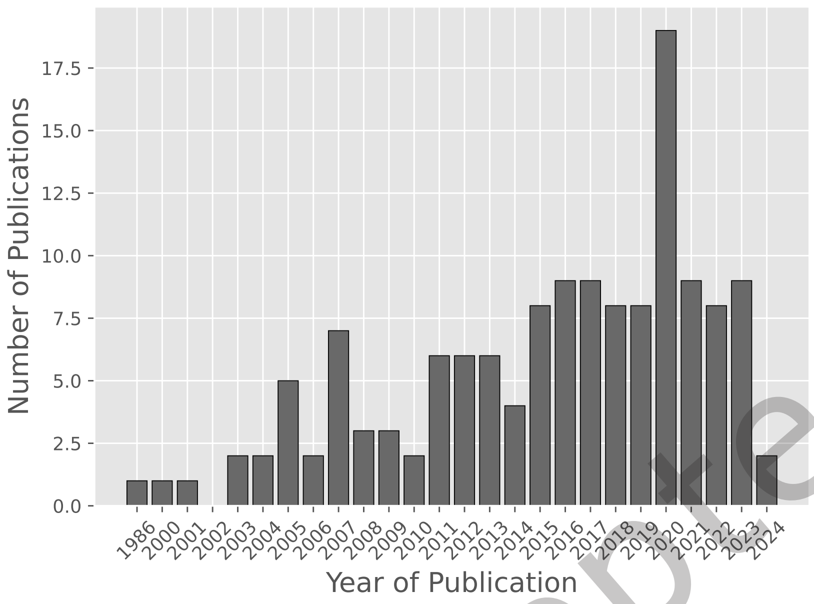

Fig. 1. Publications on hot fixing software (2000-2022) over the years.

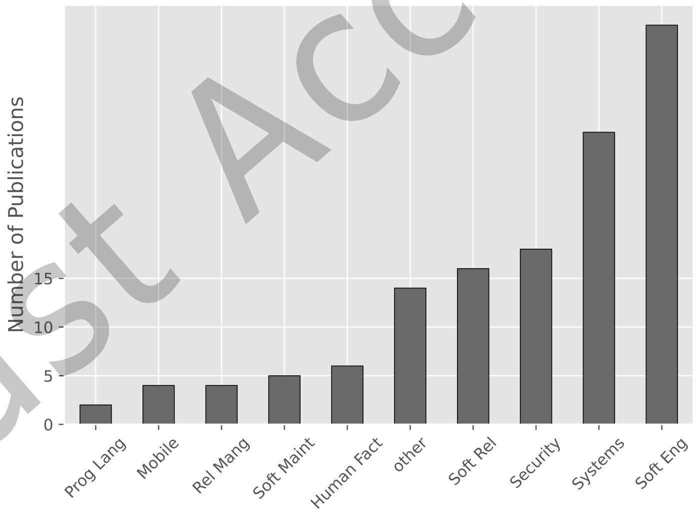

Fig. 2. Research areas the surveyed publications pertain to (Programming Languages, Mobile, Release Management, Software Maintenance, Human Factors, Software Reliability, Systems, Security, Software Engineering, and Other).

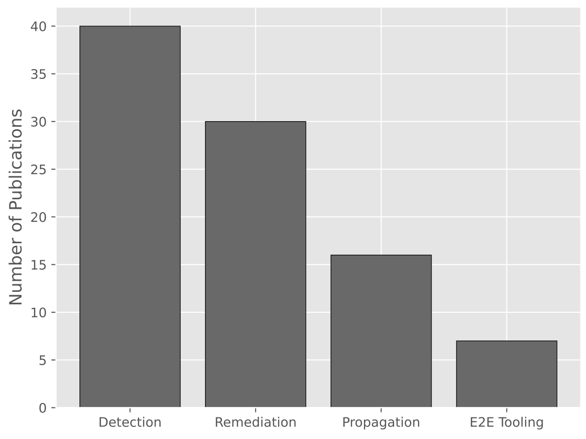

Fig. 3. Number of publications for the different tooling types for hot fixing software. This includes detecting critical issues, remediation techniques for these issues, deployment strategies for their hot fixes into the target system, as well as end-to-end tools that encapsulate all three aforementioned stages.

Table 4. Overview of Papers on General Hot Fixing Practices

Category	**Number of Papers**	**Publication Years**
Open-source Development	14	2008 - 2023
Commercial Development	14	1986 - 2023
Human Research	5	2004 - 2024

that number for tooling to deploy the hot fixes for these critical issues into the target system. We were only able to find a very limited number of end-to-end tooling that includes all three stages. From the tools that generate patches, there wasn't a very big gap between those that can be deployed at runtime (46.7%) and those that cannot (53.3%). However, the majority were tools that do not account for runtime deployment.

8 RQ1: General Hot Fixing Practices

In this section, we discuss existing hot fixing practices based on the body of work we reviewed to address RQ1. We categorize empirical research on hot fixing into commercial development, open-source development, and human research to distinguish between different development models and decision making processes. Commercial development focuses on proprietary software, where hot fixing tends to be driven by business priorities. open-source research examines public repositories and community driven patching, where transparency and decentralized decision making shape practices. Human studies explore how developers approach hot fixing, whether in open-source or commercial settings, through surveys, interviews, and experiments. This classification captures key differences in workflow, constraints, and motivations, while also allowing us to analyze commonalities. Table 4 presents an overview of the papers relevant to RQ1.

8.1 Open-source Development

Studies on open-source software development reveal unique characteristics in how bugs and updates are managed and the unique role that hot fixing plays in this process. Mockus et al. [98] conduct a general study on open-source software development. One of the conclusions they make is that in open-source systems, the response time to bugs reported by the customers is very quick. In contrast to commercial settings, they found that bugs are patched as soon as they are reported. Thus, an interesting takeaway from this work is that "hot fixes" might only be applicable to commercial systems that follow a stricter release schedule. As for the commit frequency, Kolassa et al. [77] conduct an empirical analysis in open-source software and find that it is common for authors to add two commits to the codebase in a short amount of time. One reason is splitting the contribution into multiple commits but the other reason is for hot fixing something that went wrong with their previous commit. Their analysis of open-source commit frequencies can be utilized to inform configuration management activities which are essential to the hot fixing process.

Additional studies analyze the characteristics of hot fixing, particularly in environments with shorter release cycles and security demands. Illes-Seifert et al. [63] consider hot fix the phase that makes up the first 5% of the total time between two releases. They find that the defect count of a file does not increase when it's modified in the context of a hot fix despite the usually inadequate testing of these changes. Khomh et al. [76] find that bugs are fixed faster when the length of the software release cycle is shorter, although a lower percentage of bugs are being fixed compared to longer release cycles. Zhao et al. [167] find that hot fixing is the most commonly used for change-induced incidents specifically induced by a data change. Malone et al. [94] critique the manner in which software patches are released. They find that only 1/4 of security vulnerability patches are disclosed on the National Vulnerability Database. Marconato et al. [95] empirically analyze the hot fixing of security vulnerabilities. An interesting takeaway from this study is that developers are relatively very reactive to security bugs. The average time for patching a vulnerability is around the 14-day mark. Moreover, the time between the discovery of a vulnerability and its patch release for operating systems decreases with the years. Finally, Gunawi et al. [49] investigate 597 unplanned outages that occurred from 2009 to 2015 and find 12 categories for their root causes: upgrades, network failures, bugs, misconfiguration, traffic load, cross-service dependencies, power outages, security attacks, human errors, storage failures, miscellaneous server and hardware failures, and external and natural disasters. In later work [87], hundreds of high-severity incidents from different Microsoft Azure services production runs are studied, and it is found that bugs are most often the root cause for them. Most commonly, these are data-format incidents, fault-related incidents, timing incidents, or constant-value setting incidents.

When it comes to mobile development, additional constraints must be considered for hot fixing. This is because changes in the mobile application itself requires deployment to app stores. This in itself may require an approval process. Additionally, re-installation from the users' end will also be needed. From a mobile application development perspective, Hassan et al. [56] study the phenomenon of emergency updates in the Google Play store. An emergency update in the context of this paper is an update published a short time after the previous update. They discover that while these updates are not well documented, they often end up being a permanent change to the app and that they receive a lower ratio of negative reviews. Moreover, they identify eight patterns for these types of updates. The commonality in all of the discovered patterns is that these updates are usually due to simple development mistakes. Shen et al. [125] also conduct an empirical study on the Google Play store to better understand how release planning can be optimized. In analyzing the trends for user rating, the authors conclude that hot fixes are encouraged and do not harm the app as long as they strictly target fixing bugs as intended (not feature updates).

Within the context of video games, Lin et al. [82] study urgent updates which are either those that developers describe as hot fixes or updates outside of the planned release cycle for the purpose of fixing critical bugs that

cannot wait until the next release. While 80% of the evaluated games in the study have urgent updates, they find that games that have a frequent update strategy are more likely to have a higher ratio of this kind of update. The urgent update also does not necessarily always address an issue from the immediate previous update. Truelove et al. [136] investigate which kinds of bugs are most frequently targeted by hot fixes. They find that crash bugs are the most severe, closely followed by Object Persistence and Triggered Event.
All that said, failure recovery can potentially cause more harm than the original issue if mishandled. Guo et al [50] explore this phenomenon and present a classification of failure recovery faults and recommendations for preventing them. The main takeaway from their work is that failure recovery should be mitigated conservatively and that the global system context should be considered as opposed to local reasoning about the failure.

Answer to RQ1.1 (open-source development): Open-source projects frequently apply quick, sequential commits to manage small bug fixes, contributing to flexible configuration management practices. Data-format incidents, fault-related incidents, timing incidents, or constant-value setting incidents are among the most common high-severity incident types. In security-critical contexts, studies reveal that response times to vulnerabilities are faster (around 14 days), though patch transparency is inconsistent, with only a quarter of security fixes disclosed in the National Vulnerability Database. Additionally, hot fixes in mobile applications though not well documented tend to be permanent changes to the application, while in video games, urgent updates address severe issues like crashes, with frequent updates leading to higher rates of hot fixing.

8.2 Commercial Development

Hot fixing in commercial software development is a critical practice that enables rapid response to urgent issues, balancing the need for immediate fixes with the constraints of structured release schedules.
Several studies examine the causes of critical incidents, the effectiveness of mitigation approaches, and the operational challenges associated with resolving them in large-scale systems. To better understand high-severity production incidents, Ghosh et al.[41] study hundreds of these incidents and their postmortems in the Microsoft-Teams cloud-based service. Their empirical study offers many valuable insights into the characteristics of high-severity incidents. They found that the root causes for 60% of these incidents stemmed from non-code issues such as infrastructure, deployment, and dependencies and that 80% were resolved without a code or configuration fix. They also found that 30% of these incidents had a delay in the mitigation process after the identification of the root cause due to poor documentation and manual steps which hints at the need for more automation and the possible opportunity of even automating the documentation itself given recent advances in large language models. The paper has 16 key findings, all of which are very applicable to the topic of this survey and worth reading more in-depth. Zhou et al. [170] found that only 3.8% of important customer issue reports at Microsoft are mitigated through hot fixes, further indicating that companies may avoid hot fixes even in high-stakes environments, opting instead for other mitigation approaches when possible. In 1986, administrators reported 41 critical mistakes in over 1300 years of operation [45] when referring to fault tolerant servers. However, as modern systems become increasingly complex, such historical metrics are less representative of contemporary challenges. With the rise of internet services replacing traditional fault-tolerant architectures, Oppenheimer et al. [104] highlight that operator errors are the leading cause of failures and the primary factor in repair time, with configuration mistakes being the most prevalent. They identify two critical challenges in the space: incomplete service/resource dependencies and imprecise resource health assessment. Holloway et al. [59] comment on hot patching for the Mars Science Laboratory Curiosity rover. Hot patches allow rapid, non-permanent modifications to the rover's software in RAM and must be re-applied at every reboot. As of October 2022, 13 hot patches were installed over 57,000 times on the rover flight software.

Studies on hot fix frequency give insights into the relationship between release cycles, deployment strategies, and the stability of software systems. Savor et al. [124] study continuous deployment practices at Facebook and OANDA. The authors in this paper use the number of hot fixes as a measure of software quality suggesting that hot fixes indicate low quality software. They find that increasing the number of deployments does not cause an increase in the number of hot fixes, such that rapid deployments can be achieved without compromising stability. Anderson et al. [12] touch on how development for hot fixing tends to happen on a separate branch and how this contributes to the large number of integration that happens during the initial stages of the release. Change is the leading cause for incidents in online systems. Change-induced incidents also tend to have higher severity and longer resolution times [152]. A study [152] 161 change-induced systems over a two year period from an online service system from Ant Group revealed the 4 challenges in change management: inadequate monitoring metrics, inadequate change monitoring, low business traffic, and inefficient abnormal change localization.
As for triaging these incidents, Chen et al. [24] investigate industrial practices through 20 online service systems at Microsoft. They find that up to 91.58% of incident reports are reassigned at least once which causes around a 10 time increase in the triage time of an incident. This frequent phenomenon is especially expensive when it comes to high-severity incidents, ones that require hot fixes. They also benchmark bug triage tooling for incident triage and find that while they work to a certain extent, they need to be improved to fit this context. Also at Microsoft, Chen et al. [30] study over two years of incident management practices.
As for the time required to hot fix critical issues, we found studies that analyze bug fixing times in commercial settings which shed light on this. At CA Technologies, they find that the bug fixing time for different bugs is an uneven long tail distribution [158]. An analysis of the impact of various bug features on this bug fixing time is conducted in which they account for bug priority and bug severity. In general, they find that bugs with the highest priority and highest severity are fixed faster. In the context of hot fixing, these are the type of bugs that hot fixing efforts would target. A study on Mandelbugs (ones that cause hard-to-reproduce failures due to their complexity) in real-world IT systems in production [135] creates estimates for several trends on hot fixing based on the authors' experiences. They disclose that a hot fix for a Mandelbug generally takes between a few minutes to three hours with the mean time for a hot fix being one hour. Additionally, after manual analysis of a detected Mandelbug, it has a 0.2 probability of requiring a hot fix. Finally, they estimate that the probability that such a hot fix would need just a reconfiguration and not a reboot is 0.9. In their continuation work [46], they describe a recovery model using a flowchart for four types of bugs: Restart-maskable Mandelbugs, Reboot-maskable Mandelbugs, Reconf-maskable Mandelbugs, and other (non-maskable such as Bohrbugs). They describe that for a restart, reboot, or reconfiguration-maskable Mandelbugs, a hot fix is usually sufficient to mask a failure by including system reinitialization or parameter adjustments (like resource thresholds). However, with Bohrbugs, a hot fix may not fully address the issue, and a longer-term solution, like a traditional bug fix, may be necessary, which can require additional development and testing time. The proposed flowchart can be adapted to develop a semi-Markov model that estimates recovery time from a failure for a generic IT system.
When looking at the effect of hot fixing on commercial software, Li and Long [81] raise an interesting point. They study the architectural degeneration of a commercial compiler system across two versions. They find that architecture degenerates over time and that correlated components are the main cause for this degeneration. As hot fixes are most commonly temporary workarounds for critical issues, we can assume that by their nature they contribute to this phenomenon.

Answer to RQ1.2 (commercial development): An increased number of deployments does not lead to more hot fixes, suggesting that rapid deployment can be stable. A study of high-severity incidents in Microsoft Teams found that 60% of root causes were non-code-related, such as infrastructure and deployment problems, highlighting the importance of addressing operational factors beyond code. This somewhat contradicts the study found in open-source research suggesting that the main cause of incidents is bugs. Moreover, only 3.8% of critical customer issues are resolved through hot fixes, indicating a preference for alternative mitigation strategies. Research from CA Technologies highlights that high-priority and high-severity bugs are fixed more quickly, aligning with the goals of hot fixing. Additionally, estimates regarding Mandelbugs show that hot fixes typically take between a few minutes and three hours, with a significant likelihood of requiring only reconfiguration.

8.3 Human Research

A few studies conduct surveys with users and system administrators which give insight into the practices and challenges in hot fixing software. An important perspective on hot fixing that must be considered in user-facing systems is the end-user experience in the installation process itself once the hot fix is released. Vaniea and Rashidi [143] surveyed 307 users. They found that users experience six stages when updating their system: awareness, decision to update, preparation, installation, troubleshooting, and post date. Sarabi et al. [122] studied user behaviour when it comes to software updating within the security domain by analyzing more than 400, 000 Windows machines. They looked at the relationship between the vendors and the users taking into account the rate of updating, vendor patch deployment, and patch installation practices.

The second category of individuals that must be considered are the system administrators who manage the machines of organizations and their software updates. To make sure that the released hot fixes actually get updated on the underlying infrastructure of an organization, we need to understand and cater to the needs of these individuals. Li et al. [79] found that system administrators go through five main stages: learning, deciding, preparing, deploying, and remedying. This was found through surveying 102 system administrators, 17 out of them in-depth. This study further identifies four pain points for system administrators: update information retrieval, update decision making, update deployment, and organizational culture that impedes regular update adoption. Barrett et al. [17] conducted field studies to further understand the problem-solving strategies of system administrators and conclude that available tooling does not support them in their practice, encouraging further work in this specific area. More recently, Jenkins et al. [67] surveyed 220 system administrators and found that the size of the organization greatly affects their patching processes such that larger organizations are more likely to have patching policies.

Answer to RQ1.3 (human research): Surveys of users identified six stages they experience during updates and an analysis of over 400,000 Windows machines shed light on user behaviour in the context of security updates. For system administrators, it was found that there are five stages they navigate during updates and four key pain points that hinder effective hot fix adoption (namely update information retrieval, update decision making, update deployment, and organizational culture), emphasizing the need for better tools and support to address these challenges especially in smaller companies that may not have patching policies.

8.4 Summary of Common Practices

Despite differences in context, several practices emerge consistently across open-source, commercial, and human-centered studies. All three perspectives highlight the tension between speed and stability. In open source, developers frequently push hot fixes immediately after a faulty commit [77]. In commercial settings, rapid fixes often require separate branches [12] or reconfigurations to contain incidents [135], and for administrators and end-users, the pressure to apply updates quickly is tempered by pain points such as troubleshooting, deployment friction, or organizational culture [79, 143]. They also underscore that hot fixing extends beyond the technical patch itself: Poor documentation delays fixes in commercial systems [41], open-source practices highlight the role of configuration management [77], and administrator surveys emphasize the need for better tools to support decision-making and deployment [17, 30]. Together, these commonalities point to hot fixing as an activity that consistently balances urgency, reliability, and human processes across domains. Another recurring pattern is that hot fixes frequently serve as temporary workarounds rather than permanent solutions. In open source, developers often push quick corrective commits to patch earlier mistakes [77]. In commercial systems, hot fixes are used to mask failures or reconfigure systems even though they may accelerate architectural degeneration [81, 135], and from the perspective of developers and administrators, hot fixes are often applied quickly but leave behind troubleshooting or maintenance burdens [79, 143]. This reinforces the view of hot fixing as a short-term emergency response designed to restore service rapidly, with more durable solutions deferred to later development cycles. Finally, they highlight the impact of release cadence and deployment models on hot fixing practices. In open-source projects, shorter release cycles are associated with quicker hot fixes but a lower proportion of bugs being fixed overall [63, 75]. Commercial studies show that rapid continuous deployment can reduce the need for hot fixes without compromising stability [124], while in mobile and organizational contexts, app store approval processes and patching policies impose external constraints that shape how quickly fixes can be applied [56, 125, 166].

9 RQ2: Automation and Tooling

In this section, we detail the techniques and applications that we found by examining the literature on hot fixing to address RQ2. First, we present work on human-assisted tooling, i.e., helper tools for system administrators and software developers for hot fixing at different stages and granularities. These tools do not fully automate any of the stages of the hot fixing process. Instead, they aid the human effort in doing so through reporting, debugging, etc. We then dive into fully autonomous tools that automate at least one of the stages of hot fixing: detecting the need for a hot fix, generating the hot fix, or deploying it into the target system. We outline the taxonomy of existing work on hot fix tooling in Figure 4. In Table 5 we provide an overview of the tools within the different categories. Side-by-side comparisons of the tools within each category based on domain, evaluation, and effectiveness are presented at the end of each category section.

9.1 Human-Assisted Tools

We begin by exploring human-driven techniques for hot fixes, which involve semi-automated or human-assisted tools designed to support and streamline the hot fixing process. These techniques include system administrator tools that help facilitate hot fixing activities, bug reporting mechanisms that simplify the identification and understanding of critical underlying issues, and debugging assistance tools that aid in diagnosing and resolving problems efficiently. Each of these tools enhances the speed and accuracy of addressing critical software issues, relying on human expertise alongside automation.

9.1.1 System Health Monitoring. In this section, we will present tools designed for real-time monitoring and alerting system administrators to critical issues.
System administrators are often the individuals responsible for handling critical crises in enterprises. They usually play the most critical role in mitigating time-sensitive system disruptions such as security attacks, performance degradation, and system unavailability. While there has been abundant research on tooling for software developers, less is known about the methods that the system administrators follow. These individuals and the tools that they adopt are vital for hot fixing systems. In this section, we present papers that propose tooling for system administrators that could be applicable for the scenario of hot fixing unplanned time-critical bugs.

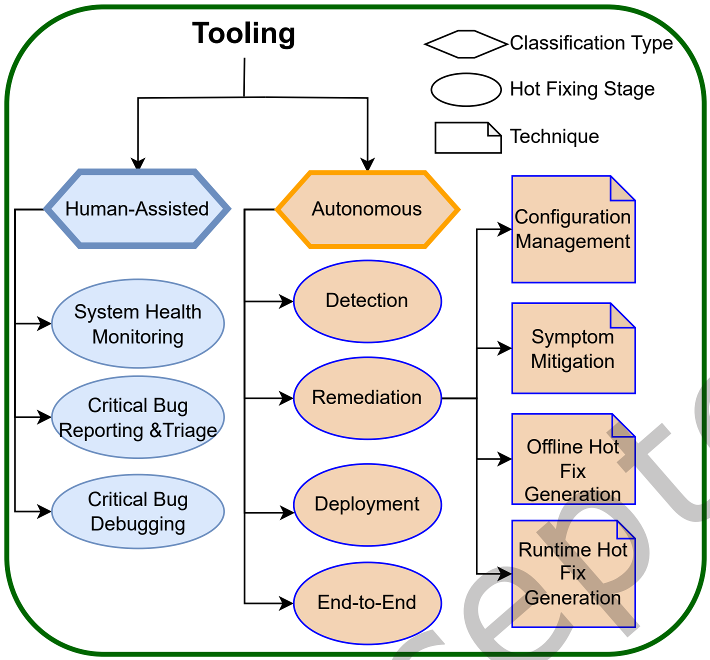

Fig. 4. Taxonomy of existing work on tooling for hot fixing activities.

Table 5. Overview of Hot Fixing Tooling

Category	Sub-category		Number of Papers	Publication Years
Human-Assisted	Monitoring		8	2006 - 2022
	Reporting & Triage		19	2009 - 2022
	Debugging		8	2011 - 2023
Autonomous	Detection		21	2003 - 2023
	Remediation	Configuration Management	3	2008 - 2015
		Symptom Mitigation	3	2005 - 2016
		Offline Hot Fix Generation	11	2007 - 2020
		Online Hot Fix Generation	14	2007 - 2024
	Deployment		16	2004 - 2021
	End-to-End		9	2005 - 2017

Bodik et al. [19] propose two new tools for system administrators at Amazon. The first aids in tracing dependencies between the different components of the system and allows for tracking the overall health of the system. The second monitors the actions of administrators to enable the identification of recurring problems and automate suggestions for their resolution. The usefulness of these helper tools in improving the efficiency of system administrators is not evaluated in this paper. They do, however, highlight that it will be essential to properly train them to convince them that these tools will indeed be useful. In turn, their tools will need to be extended to monitor system administrator actions. These actions and their outcomes should be incorporated in the tool suggestions, and the tool could also receive feedback from the system administrators themselves to improve its accuracy in the future. In a continuation paper [20], they develop a technique for crisis classification based on summarising the data center's state in a representation called a *fingerprint*. The fingerprint is a summary of hundreds of performance metrics which can be used to identify whether a specific crisis has been seen before so that its known solution can be applied. Under realistic conditions, their approach enables initiation of recovery actions with 80% accuracy in an average of 10 minutes which is 50 minutes earlier than the deadline required. However, it is not clear if these tools are effective in cloud computing environments or across different applications used by the system administrators.

Tools for system management and system health monitoring have been developed to automate updates, detect issues, and manage memory errors effectively. TJOSConf [148] is a platform that assists with system management by automating system updates safely. It interacts with services to verify the impact of an update and ensure that unexpected failures are recognized quickly. This tool was utilized in Alibaba and the authors claim that were no service breakdowns as a cause of system updates. For diagnosing and patching memory management bugs, First-aid [40] is a runtime tool for diagnosing and patching memory management bugs and the memory objects that trigger them. This technique relies on rolling back programs to previous checkpoints, which may not always be feasible in environments where maintaining state consistency is critical. AUDIT [92] focuses on troubleshooting recurring transient errors in cloud systems specifically. Using lightweight triggers, it can identify the first occurrence of a problem and then rank the software methods according to their likelihood of being involved in the root cause of the problem. Similar work was done by Yuan et al. [155] which is based on statistical learning techniques that classify system call sequences. Nair et al. [101] propose a hierarchical monitoring system that is composed of low-level detectors, structure discovery models for identifying relationships among the variables in the system, learning for better detection, and human interaction for further refining machine learning outcomes by allowing expert adjustments. The system was deployed at Microsoft and at the time of publication was reported to detect 19 customer-impacting issues in three months. Finally, to reduce the manual effort of on-call engineers for system monitoring and health checks, SoftNER [127] was introduced. It is a tool based on unsupervised machine learning and deep learning, dedicated to extracting structured knowledge (e.g. error messages, resource IDs, etc) from service incidents.

We present a summary and side by side comparison of all tools mentioned in Table 6.

Answer to RQ2.1 (Human-assisted tools for system health monitoring): System administrators are vital in managing critical crises such as security attacks and system failures in enterprises, making tooling for helping them monitor system health essential. Amazon aims to enhance system administration through dependency tracing and monitoring of administrator actions. Tools like TJOSConf, First-Aid, AUDIT, and SoftNER have been introduced to automate system updates, diagnose memory management bugs, troubleshoot transient errors, and reduce manual efforts in system health checks, each with their own limitations and specific contexts of application.

Table 6. Summary of Monitoring Tools Applicable for Hot Fixing

Domain	Function	Benchmark	Tool	Effectiveness
General problem diagnosis	root cause identification: use system behavior information to build correlations with known problems	dataset of system call sequences created through fault injection	Yuan et al.[155]	accuracy of root cause detection is nearly 90%
Performance	classification and identification of performance crises	4 months of trouble-ticket data from a production datacenter	Fingerprint [20]	80% correctness in an average of 10 minutes
Memory	surviving common memory management bugs and preventing their reoccurance	3 server applications (Apache, Squid, and CVS) and 4 desktop applications (Pine, Mutt, M4, and BC)	First-aid [40]	provides quick failure recovery and thereby hides program failures from user
Online service systems	system management platform for safe system updates	application in Alibaba	TJOSConf [148]	application in Alibaba
Online Service Systems	monitoring system to detect and diagnose service issues	data collected from 11 instances of the Storage Manager component of the service	Nair et al.[101]	deployed in production
Cloud	troubleshooting transiently-recurring problems	5 mature open source and commercial applications	AUDIT [92]	AUDIT identified previously unknown issues
Cloud	knowledge extraction from service incidents	more than 2 months of cloud incidents at Microsoft	SoftNER [127]	precision of 0.96, deployed at Microsoft
Large-scale internet services	2 tools: visualize system health and suggests solutions to operators to recurring problems	Deployed within Amazon	Maya [19]	Maya: useful but overwhelming to users. Recommender-useful metrics appeared near the top of the suggestions

9.1.2 Critical Bug Reporting and Triage. An efficient reporting and triage process for critical issues in production is essential to facilitate fast mitigation and hot fixing. Ensuring timely and accurate diagnosis allows teams to respond quickly, minimizing system downtime and user impact.

Several studies tackle this topic by proposing automated tooling for quicker incident diagnosis through optimised bug reporting processes. Zhang et al. [163] developed a tool called "Onion" that automatically locates incident-indicating logs. Their solution uses log clustering techniques and has been used practically on the cloud system at Microsoft. Log3C [57] is a tool that also uses log clustering for distinguishing relevant logs but with the addition of utilizing system KPIs (key performance indicators). Also based on KPIs, Decaf [16] utilizes service logs to triage KPI issues and has successfully diagnosed 10 out of 31 issues on two large-scale cloud services. Decaf, Onion, and Log3C have been successfully applied in the industry.

Other techniques aim to utilize the repeated occurrence of the same issues to improve reporting, diagnosis, and intervention. Woodard et al. [151] identify crises in complex distributed systems through automatic recognition of recurrences. Using model-based clustering based on a Dirichlet process mixture, they are able to distinguish previous similar crises for which the root cause is known. Similarly, Saha et al. [117] also utilize past incident investigation reports that include root cause information and propose an Incident Causation Analysis (ICA) engine that uses state-of-the-art NLP techniques to extract structured causal knowledge from them.

When production incidents occur, alerts are generated, triaged, and assigned to the responsible team. There is often a large load of alerts that overwhelms on-call engineers which must be prioritized. Interestingly, when looking at 18 real-world online service systems at Microsoft, it was found that on average, 50.32% of incidents are not high priority and do not need immediate remediation [26]. DeepIP [26] is a tool that aims to identify these types of incidents. To prioritize the alerts generated from incoming incidents, Zhao et al. [166] propose *AlertRank* which is a framework for identifying severe alerts. It works by extracting alert features and utilizes the XGBoost ranking algorithm to triage the incoming alerts achieving an F1 score of 0.89 average surpassing baselines and reducing the required manual effort. Warden [80] is another tool in this problem space but it differs in that it aims to detect broader incidents by aggregating alerts across multiple services using a balanced random forest model. RAPID [88] takes a different approach by focusing on security-related alerts, particularly those from intrusion detection systems (IDS) while optimizing for space and time. To further help with triage, it may be useful to predict the bug fixing time. TTMPred [147] is a tool designed for this purpose. It functions within continuous triage, utilizing temporal information from ongoing discussions. TTMPred is based on a deep learning approach that uses both the semantics from the textual data as well as this temporal information.

Another key challenges in incident management is the fragmentation of monitor-generated Incident reports and customer incident reports. LinkCM [47] is a tool that aims to link these reports by formulating the linking problem as a binary classification problem, and adopting a neural network to solve it. When evaluated on 7 industrial cloud service systems in Microsoft, they find that on average in 77.70% of scenarios, the system incidents affecting customer products can be detected before the customer reports problems and that an improvement in the efficiency of CI triage can be achieved when linking the reports. Also to make parsing customer reports more efficient, iDice [85] is a tool that links issue reports that might indicate a new widespread problem by finding patterns in report attribute combinations.

Assigning the appropriate team to handle an incident is another crucial aspect of triage. DeepCT [25] is a deep learning-based approach that leverages incident discussions for continuous triage. Upon evaluation on 14 large-scale online service systems in Microsoft, it was able to correctly identify the responsible team in 64.1% of the instances on average given one discussion item. A later tool named DeepTriage [109] combines multiple machine learning techniques such as gradient boosted classifiers, clustering methods, and deep neural networks and achieves a 82.9% F1 score on incidents in Microsoft Azure.

At Microsoft, the Windows Error Reporting (WER) has processed error reports collected from a billion machines over the course of 10 years [42]. This system classifies the collected reports into buckets based on similarity which allow for developer effort prioritization and more immediate, user-focused development. Another technology at Microsoft is the Service Analysis Studio [90] [84] [91] which is a data-driven solution that tackles incident management from a software analytics perspective. More recently, an approach named Oasis [73] has been deployed at Microsoft which assesses the impact scope of cloud outages using relevant incidents to generate human-readable summaries of the outage by leveraging LLMs.

We present a summary and side by side comparison of all tools mentioned in Table 7.

Table 7. Summary of Reporting and Triage Tools Applicable for Hot Fixing

Domain	Function	Benchmark	Tool	Effectiveness
Windows machines debugging (large scale)	automates the processing of error reports to prioritize developer effort and report fixes	large scale data- error reports collected from Windows machines worldwide	WER [42]	effective at reporting and recovering from errors even for rare Heisenbugs due to large scale
General system	identify impactful system problems by utilizing log sequences and system KPIs	log data from an online service system at Microsoft	Log3C [57]	average precision of 0.877 and average recall of 0.883
Cloud	locating incident-indicating logs	Apache Spark Examples	Onion [163]	average F1-score of 0.95
Cloud	detects the occurrence of incidents from a global perspective	data collected in an 18-month period from 26 major services	Warden [80]	deployed in Microsoft Azure
Cloud	assess and summarize the impact scope of outages	18 real-world cloud systems	Oasis [73]	deployed at Microsoft
Cloud	root cause analysis for incidents	in-house collected dataset of 2000 incidents.	Saha et al.[117]	built at Salesforce, shown to be effective
Cloud	incident transfer service	real incidents in Microsoft Azure	DeepTriage [109]	deployed in Azure; achieves 82.9% F1 score
Cloud	customer Incident Triage via Linking with System Incidents	collected datasets from 7 production cloud service systems in Microsoft	LinkCM [47]	LinkCM significantly outperforms its 2 competitors.

Continued on next page

Domain	Function	Benchmark	Tool	Effectiveness
Performance	automated diagnosis and triaging of KPI issues using service logs.	2 large scale cloud services in Microsoft	DeCaf [16]	successfully diagnosed 10 known and 31 unknown issues
Performance + Distributed computing	monitors performance metrics, applies online clustering to identify patterns, and detects "crisis" events	simulated data and data from a production computing center	Woodard et al.[151]	validated in production
Online Service Systems	framework for identifying severe alerts	datasets from a top global commercial bank	AlertRank [166]	F1-score of 0.89
Online Service Systems	continuous Incident Triage	14 large-scale online service systems in Microsof	DeepCT [25]	average accuracy identifying the responsible team precisely is 0.641-0.729
Online Service Systems	data-driven techniques to improve incident management at Microsoft	deployed at Microsoft	Service Analysis Studio (SAS) [84, 90, 91]	deployed to worldwide product datacenters and widely used by on-call engineers for incident management
Online Service Systems	prioritizing incidents based on a large amount of historical incident data	real-world incident data from Microsoft	DeepIP [26]	AUC of DeepIP achieves 0.808
Online Service Systems	incident time-to-mitigate prediction	4 large-scale online service systems in Microsof	TTM-Pred [147]	improves upon baseline by 25.66% on average in terms of MAE
Online Service Systems	problem Identification for Emerging Issues	service at Microsoft and a synthetic generated dataset of issue reports	iDice [85]	deployed at Microsoft

Continued on next page

Domain	Function	Benchmark	Tool	Effectiveness
Security	real-Time Alert Investigation with Context-aware Prioritization	1TB dataset from DARPA Transparent Computing (TC) program with 411 million events	RAPID [88]	reduce the time of alert provenance analysis to discover all the major attack traces by up to 99%

Answer to RQ2.1 (Human-assisted tools for critical bug reporting and triage): Studies on bug reporting propose automated tools that streamline incident diagnosis by optimizing bug reporting processes. Tools like Onion and Log3C utilize log clustering, with the latter incorporating system KPIs to improve log relevance, and both have seen successful industry use. Other methods, focus on identifying recurring crises in complex systems to aid rapid diagnosis and response. As for reducing manual triage efforts, tools such as AlertRank aid with alert prioritization and tools such as DeepCT aid with assigning them to the responsible team.

9.1.3 Critical Bug Debugging. Although better bug-reporting methodologies and tooling can accelerate the process of identifying critical issues, additional assistance with debugging is often required to resolve them. IDRA_MR [96] is an online debugger for Map/Reduce applications that works by removing the debugging session to an external process. Wolverine [144] is an end-to-end debugging tool that allows for stepping through the execution of a program, the visualization of its states, and the synthesis of repair patches. Wolverine is able to integrate the synthesized patches into the running program without requiring a restart. This tool can aid with the debugging process as well as suggest resolutions without the overhead of system downtime. The debugging process can be made more efficient when outages in production clouds stem from recurring issues. COT [149] is an outage triage approach that learned from historical outages to infer the root cause of emerging outages.
On-call engineers often utilize troubleshooting guides (TSGs) whenever available for an incident at hand to aid with quicker resolution times. On average, developers spend about 36.3% of the total mitigation time just on locating the desired TSG [70]. DEEPRMD [70] is a tool that recommends the appropriate TSG to the developer given an incident using deep learning technique and the textual similarity between incident description and its corresponding TSG. AutoTSG [128] is a framework that takes this a step further and makes use of existing TSGs to turn them into executable workflows. They evaluate their approach on 50 TSGs and show promising initial results in parsing the TSGs for execution.
The rise of large language models presents a unique opportunity for enhancing debugging tools, as root cause prediction can be paired with explanations that aid in the debugging process. RCACopilot [28] is one such system that leverages large language models to offer efficient root cause analysis for cloud incidents at Microsoft. Ahmed et al. [9] experiment with a more direct approach that utilized large language models for the purpose of finding the root cause and mitigation for production incidents. They investigate the performance of existing models on over 40K incidents at Microsoft and find that it is a promising research direction. Finally, XPERT [71] is a tool that automates the domain-specific language (DSL) queries that on-call engineers have to generate to analyze telemetry data. It leverages large language models and historical incident data to generate custom queries for new incidents which helps accelerate the debugging process.
We present a summary and side by side comparison of all tools mentioned in Table 8.

Answer to RQ2.1 (Human-assisted tools for critical bug debugging): IDRA_MR and Wolverine provide online debugging capabilities, including external session handling and repair patch synthesis without

Table 8. Summary of Debugging Tools Applicable for Hot Fixing

Domain	**Function**	**Benchmark**	**Tool**	**Effectiveness**
Big Data applications	online debugging	2 applications: poll analysis and blockchain analysis	IDRAMR [96]	demonstrated successful online debugging scenarios
Heap manipulating programs	debug-Repair environment: visualize program and generate hot patches	1600 buggy programs generated using fault injection	Wolverine [144]	could repair all the buggy instances in less than 5 seconds in most instances
Online Service Systems	automated troubleshooting guide recommender system	18 online service systems	DeepRmd [70]	can recommend the correct TSG as the Top 1 returned result for 80.3% incidents
Cloud	automation of troubleshooting guides to executable workflows	50 troubleshooting guides	AutoTSG [128]	precision 0.94 in parsing guides to execution
Cloud	root Cause Analysis by LLMs	year's worth of incidents from Transport service in Microsoft	RCACopilot [28]	accuracy up to 0.766
Cloud	outage triage approach that considers the global view of service correlations	real-world dataset containing one year of data collected from Microsoft Azure	COT [149]	acuracy 82.1% 83.5%
Cloud	query recommendation framework for incident management.	deployed in production at Microsoft	Xpert [71]	deployed in production at Microsoft
Cloud	root cause and mitigation for production incidents using LLMs	40,000 incidents from Microsoft	Ahmed et al.[9]	exploratory study-not a fully deployed tool

requiring a restart. Other techniques use historical outage data to expedite the root cause analysis of recurring production issues. Troubleshooting guides are common in assisting the debugging process, and some tools such as DEEPRMD and AutoTSG leverage deep learning to recommend and automate them. Finally, the integration of large language models in tools like RCACopilot and XPERT enhances root cause prediction and automates query generation for faster debugging.

9.2 Autonomous Tools

Next, we review the applications of techniques that fully automate at least one of the stages for hot fixing. This includes tooling for detection, remediation, and deployment. Finally, we dedicate a section for tools we found that automate this pipeline end-to-end.

9.2.1 Detection Tooling. We begin with an overview of the tools that we found for detecting the need for a hot fix. This includes tooling for detecting underlying functional and non-functional critical software issues.
Degradation of performance is often a problem that requires immediate attention through a hot fix. Zhang et al. [162] address the specific scenario of service level objectives violations. They introduce an approach that identifies likely causes of performance problems within this context using several Bayesian network models that adapt to changing workloads. Their approach is low cost which makes it feasible for practical application. Fu et al. [39] also propose a solution for detecting critical performance issues. Their technique uses system metric data mining to diagnose problems in online service systems. Performance anomaly alerting based on trace data often has the issue of generating false alarms or having limited explainability due to reliance on deep learning models. TraceArk [157] aims to address this issue by incorporating a small amount of engineer feedback.
Looking at problem diagnosis from the perspective of functional problems, Triage [137] is an automated tool for onsite failure diagnosis. This tool diagnoses a failure at the moment that it occurs and reports in detail the nature of the failure, the conditions that trigger it, as well as the fault propagation chain. Triage mimics the steps a human takes in debugging a failure by employing several existing diagnosis techniques as well as a new one proposed by the authors called *delta analysis*. Khomh et al. [75] address triaging crashes from the vantage point of prioritizing crash types. This process helps direct development efforts toward critical crashes that might require hot fixing. By grouping similar crash reports into crash types, they are able to use entropy region graphs to capture the distribution of their occurrences among system users. IFeedback [169] is a tool that leverages user feedback for detecting issues. Rather than conducting exhaustive text mining of feedback reports, it focuses on fast online issue detection by extracting word combination-based indicators from feedback texts. Instead of relying on user feedback, LogFlash [69] is a real-time anomoly detection tool that utilizes logs. It has been evaluated on lab settings, therefore its performance on complex real-world logs is yet to be evaluated. Alternatively, another approach leverages a static analyzer, Aspirator [156], which is tool that relies on a set of three rules derived from analyzing past critical bugs to automatically detect issues in error handling code. Finally, ctests [168] a type of test that links configurations in the production system to software tests. This allows changes to the configurations to be tested before any failure-inducing changes, dormant software bugs, and misconfigurations reach production. Ctests are generated from existing tests with reasonable manual effort by instrumenting the configuration APIs of the system (parameterizing an existing test to run against different system configurations while ensuring test validity for the included configuration parameters).
From the perspective of memory issues, RESIN [89] employs low-overhead monitoring and a bucketization-based pivot scheme to efficiently detect memory leaks. It then automatically attempts to mitigate leaks to minimize service impact by isolating affected processes and performing targeted reboots, ensuring minimal disruption to the overall system. At the time of publication, RESIN had been running in production in Azure for over 3 years.
Finally, from a security perspective, there have been several works that address diagnosing vulnerabilities. Qin et al. [113] propose an automated mechanism *Dataflow Analysis for Known Vulnerability Prevention System*. Through tracing the vulnerability context and its spread path, they are able to detect the exploit and generate the vulnerability filter and its respective hot fix with minimal overhead. To protect vulnerable programs in the cloud, vPatcher [160] examines network packets to detect vulnerable processes. Araujo et al. [14] present a cross-stack sensor framework for cyber security allowing for booby-trap insertion at multiple network layers. FIBER [159] is a tool that detects security vulnerabilities in software distributions by testing the presence of security patches.

FIBER does this by first analyzing open-source security patches, generating fine-grained signatures, and using these signatures to search against the target system.
Techniques that are based on prediction exist as well. Sahoo et al. [118] leverage various system measurements to build a prediction system for large clusters. They tested various prediction algorithms and found that rule-based classification algorithms were able to predict critical events with 70% accuracy which is an encouraging avenue of exploration. In contrast to this work that predicts failures through detecting abnormal signals from monitors, AirAlert [29] is a tool for predicting critical failures in cloud service systems, which they refer to as outages, from these failure signals. They examine how outages correlate with alerting signals through Bayesian networks and use a robust gradient boosting tree classifier for outage prediction. Their approach deals with outages not only within a single service but from different parts of the whole cloud system. Similarly, NARYA [78] is a system that has been deployed in Microsoft Azure that predicts VM failures before they occur by utilizing multi-layer system signals, online experimentation, and reinforcement learning. As for forecasting when an incident in an online service will happen, eWarn [165] utilizes historical data based on alert data in real-time. CRANE [32] is a tool from Microsoft for failure prediction, change risk analysis, and test prioritization. Its deployment helped Microsoft engineers identify tests that are likely to uncover problems. Later research work at Microsoft on post-release defect prediction models utilizes various text execution metrics [58]. Finally, Shihab et al. [129] propose prediction models that specifically focus on high-impact defects for customers and practitioners. These tend to be breakage defects and surprise defects respectively. While they are somewhat successful in their mission, they conclude that more specialized models that take into account a defect's type instead of just its location are needed for practical adoption.
We present a summary and side by side comparison of all tools mentioned in Table 9.

Table 9. Summary of Detection Tooling Applicable for Hot Fixing

Domain	**Function**	**Benchmark**	**Tool**	**Effectiveness**
General software	triaging Field Crashes	10 beta releases of Firefox 4	Khomh et al.[75]	improves triaging used by Firefox teams
General software	failure prediction, change risk analysis and test prioritization system	analyzed all fixes made to Windows for one year	CRANE [32]	for some builds, CRANE reduced tests run by 50%
General software	prediction models for identifying files that have high-impact breakages and surprises	5 releases of a large commercial software system	Shihab et al.[129]	building specialized prediction models is valuable for making defect prediction adoptable in practice
General software	failure diagnosis protocol that mimics the steps a human takes in debugging	10 real software failures from 9 open source applications	Triage [137]	accurately diagnoses with overhead of under 5%

Continued on next page

Domain	Function	Benchmark	Tool	Effectiveness
General software	predict pre- and post-release bugs using test metrics	test metrics collected during Windows 8 development	Herzig et al.[58]	explorative study assessing the general suitability of test failure metrics for defect prediction models
Security	a cross-stack threat sensing framework that injects deceptive sensors at the network to dettect attackers	enterprise-grade environments	INSIDER [13, 14]	detected multiple classes of attacks with low false positives
Security	detects and blocks exploits by performing dynamic taint analysis	N/A	DA-VPS [113]	no in-depth experimentation
Security	test presense of security patch	107 real-world security patches and 8 Android kernel images from 3 different mainstream vendors	FIBER [159]	average accuracy of 94% with no false positives
Security	anomaly detection and diagnosis	Hadoop, Spark, Flink	LogFlash [69]	reduces over 5 times of training and detection time
Security + Cloud	data patching technique based on virtual machine introspection	6 vulnerable programs (httpdx, xchat...)	VPatcher [160]	experimental results show feasibility
Cloud	forecast the occurrence of outages and diagnose root cause	outage dataset collected from a Microsoft cloud system	AirAlert [29]	effective on dataset
Cloud	averting VM failures via prediction and mitigation	running in production at Microsoft;	NARYA [78]	running in production at Microsoft; reduces VM interruptions by 26%

Continued on next page

Domain	Function	Benchmark	Tool	Effectiveness
Cloud	tests for detecting failure inducing configuration changes to prevent production failures.	real-world failure-inducing configuration changes, injected misconfigurations, and deployed configuration files from public Docker images.	ctests [168]	shown effective on dataset
Cloud; memory leaks	system for memory leak detection, diagnosis, and mitigation	production in Microsoft Azure	RESIN [89]	running in production in Microsoft Azure and reports 24 memory leaks monthly on average with high accuracy
Distributed systems	critical Event Prediction for Proactive Management	a specific event id in a 350 node cluster	Sahoo et al.[118]	70% accuracy on the evaluated event
Distributed systems	static checker that implements 3 simple rules for discovering catastrophic failures.	9 distributed systems	Aspirator [156]	located 143 bugs
Performance	diagnosis of performance issues	3-tier system running a web-accessible Internet service based on Java 2 Enterprise Edition	Zhang et al.[162]	showed that collecting instrumentation, inducing models, and maintaining the ensemble of models is inexpensive enough to do in (soft) real time
Performance + Online Service Systems	performance issue diagnoosis	2 systems: TPC-W and a production system	Fu et al.[39]	36% average accuracy on beacon identification
Performance + Online Service Systems	performance anomaly detection	dataset of Microsoft Exchange service and an anomaly injection dataset collected from an open-source project	TraceArk [157]	running in production at Microsoft; improvement in F1 is 50.47% and 20.34% on the two datasets, respectively

Continued on next page

Domain	Function	Benchmark	Tool	Effectiveness
Online Service Systems	real-time issue detection based on user feedback texts	application in systems in production	iFeedback [169]	successfully applied in tens of large-scale online service systems
Online Service Systems	utilizes historical data to forecast whether an incident will happen in the near future based on alert data in real time	11 real-world online service systems from a large commercial bank	eWarn [165]	successfully deployed in 2 large commercial banks

Answer to RQ2.2 (Autonomous tools for critical bug detection): Performance issues are often identified through adaptable, cost-effective models or data mining, which diagnose problems based on system metrics. Functional failures are managed by automated onsite tools that diagnose and report failures in real time, while other approaches prioritize critical crash types to guide development focus. In security, automated mechanisms detect vulnerabilities by tracing exploit paths or analyzing network packets, and predictive tools also assist in prioritizing testing and identifying high-impact defects. However, specialized models for defect types are still needed for broader practical use.

9.2.2 Remediation Tooling. Remediation for hot fixes takes many forms depending on the software issue. Many critical issues are due to software configuration incompatibilities which can be fixed with reconfiguration rather than code-level patching. Other issues require code changes but are extremely time-critical. Thus, in these cases, instead of adding patches for the root cause which might be time consuming, workarounds might be sufficient so that the symptoms of the issue can be hidden as quickly as possible. In other cases, generating actual patches that address the underlying issues is needed. Those can either be deployed offline or during runtime depending on the availability requirements of the target system.

2a) Configuration Fix Software configuration management is essential for achieving rapid changes to modern software systems. Especially when developing hot fixes for software, the turnaround time must be fast. Configuration management makes it possible to monitor the changes made, package them, and make sure that they are error-free prior to delivery. Karale et al. [74] propose a configuration management framework that automates the tasks of the configuration manager. For hot fixing, their framework automates all the required steps for hot fix packaging which reduces the time required from 33 minutes to 25 minutes. To support applying configuration changes at runtime, Rasche et al. developed a new algorithm called ReDAC [115] which ensures the consistency of application data while the reconfiguration happens. ReDAC specifically supports distributed and multi-threaded component-based applications that have cyclic dependencies. They specifically mention the case of hot fixing where dynamic reconfiguration is necessary to achieve high reliability and availability.

At Microsoft, *ConfSeer* [110] is a system used in the Operations Management Suite for remediating configuration issues. ConfSeer leverages a knowledge base of natural language descriptions of configuration issues and their fixes. It first starts by taking a snapshot of the user configuration files. It then extracts the parameter names and values and cross-references them with the Knowledge Base. From here, if a match is found the pinpointed error and corresponding Knowledge Base article are presented to the user as a suggested fix. Confseer is reported

Table 10. Summary of Configuration Tools Applicable for Hot Fixing

Domain	Function	Benchmark	Tool	Effectiveness
General Software	configuration Manager	experiment on 60 developers	[74]	8 minutes reduction in hotfix packaging time
General software	misconfiguration Detection by leveraging a knowledge base	100K random queries taken from configuration snapshots	ConfSeer	running in production at Microsoft for 1 year to find misconfigurations on tens of thousands of servers
Distributed component-based applications	dynamic reconfiguration of component software during runtime	PaintDotNet + a lab test env	ReDAC	dynamically updated PaintDotNet with no overhead

to be 80% - 100% accurate across four large customer deployments. Its efficacy however, is fully reliant on the comprehensiveness of the knowledge base that it builds upon.
We present a summary and side by side comparison of all tools mentioned in table 10.
2b) Symptom Mitigation Often in the case of hot fixing, the bug is time-critical and there is not enough time to either identify the cause of the problem or generate a proper patch for it. Thus, a common way to remediate critical bugs is through symptom mitigation. The problematic side effect of the critical bug is removed, however, the root issue is not resolved. This often comes at the cost of lost functionality or deteriorated performance.
Qin et al. proposed Rx [112], a technique implemented on top of Linux. The idea behind it is that software failures can be mitigated by simply changing the environment in which the program executes. The technique rolls the program back to the checkpoint where the program was in a stable state and then re-executes it in a modified environment. Similarly, Sweeper [138] is a software security system that efficiently scans for suspicious requests. After an attack is detected, Sweeper re-executes the suspicious code to apply analysis techniques. Once the analysis is complete, Sweeper is then able to quickly recover the system and generate antibodies to prevent similar attacks from happening in the future. Another security mitigation technique is Security Workarounds for Rapid Response (SWRRs) which has been implemented into a system called Talos [61]. These techniques are designed to secure the system against vulnerabilities at the cost of lost functionality. Unlike configuration workarounds which have a similar effect, SWRRs require minimal developer effort and knowledge of the system by utilizing existing error-handling code.
We present a summary and side by side comparison of all tools mentioned in Table 11.
2c) Offline Hot Fix Generation In this section, we expand on automated offline hot fix generation techniques. We focus on techniques for generating hot fixes that require the system to be rebooted when the patch is applied. In other words, these are techniques that generate hot fixes but do not take into account their runtime deployment into the system later.
Ding et al. utilize this approach by generating unique signatures from log data to identify and resolve similar issues, significantly reducing service restoration time in systems with millions of users. Similarly, Pozo et al. present a protocol that quickly hot-fixes link failures within time-triggered schedules, achieving recovery within just a few milliseconds

Table 11. Summary of Symptom Mitigation Tools Applicable for Hot Fixing

Domain	**Function**	**Benchmark**	**Tool**	**Effectiveness**
General software	recovering from failures through rollbacks	4 server applications: Apache httpd, Squid, MySQL, CVS that include 6 failures	Rx	Rx can survive all the tested software failures
Security	worm analysis and recovery post attack	4 servers: Apache1/2, CVS, Squid and 3 vulnerabilities recorded by US-CERT/NIST	Sweeper	can detect an attack and generate antibodies in under 60 milliseconds
Security	neutralize security vulnerabilities in a timely manner	5 popular Linux server application + 11 real-world software vulnerabilities	Talos	can neutralize 75.1% of all potential vulnerabilities

Mining historical data to suggest solutions for recurring software issues has proven effective in improving response times. Ding et al. [35] [36] utilize this approach and can match issues by generating signatures for issues from their corresponding logs. Their system was evaluated on a real system that compromises millions of users and was able to successfully provide resolutions that reduced the mean time to restore of the service. Similarly, at Facebook, Lin et al. [83] propose a framework for predicting hardware failure remediations by utilizing similar closed repair tickets. In improving response time for link failures in time-triggered schedules, Pozo et al. [111] propose a protocol that hot fixes these link failures within a few milliseconds.
Various techniques target offline hot fix generation for software vulnerabilities. MacHiry et al. [93] argue that most security patches are safe patches meaning that they can be applied without disrupting the functionality of the program and thus require no testing. They find that most patches in the CVE database are indeed safe patches which they hope will encourage project maintainers to apply them without a delay. However, when such a preventative approach fails and zero-day attacks are discovered, ShieldGen [31] is a technique that is able to generate a patch given its instance. Subsequently, it is able to generate additional potential attack instances to determine whether given the patch the system can still be exploited.
Several tools are designed to patch vulnerabilities in specific scenarios. For containerized applications specifically, Tunde-Onadele et al. [139] [140] propose tooling, Self-Patch, for security attack containment that performs both exploit identification and vulnerability patching. They report accurate detection and classification for 81% of attacks as well as an 84% reduction in patching overhead. For out-of-bound vulnerabilities, AutoPaG [86] catches the violation, and based on the data flow analysis is able to identify the root cause and generate a patch automatically. The most impressive part of this work is that the vulnerability patch can be generated within seconds. For Android, we found AppSealer [161] which automates fix generation for component hijacking vulnerabilities. Finally, Aurisch et al. [15] propose using Mobile Agents for vulnerability and patch management. We present a summary and side by side comparison of all tools mentioned in Table 12.
2d) Runtime Hot Fix Generation In this section, we focus on hot fix generation techniques that account for the ability of the generated hot fix to be integrated into the system during runtime. Generating this specific type of hot fix has been researched for multiple specific use cases.
While runtime hot fixing techniques aim to apply patches without interrupting system availability, they may risk leaving the system in an inconsistent state. Katana [114] is a tool for hot fixing ELF binaries that aims to

Table 12. Summary of Offline Hot Fix Generation Tools Applicable for Hot Fixing

Domain	Function	Benchmark	Tool	Effectiveness
Time-Triggered network scheduling	repairs time-triggered schedules at runtime after link failures	synthetic network with 8 switches, 8 end systems, 54 links, and traffic from 50–250 frames.	Pozo et al.[111]	patching heals link failures in up to 2.5 ms
Security	detection and repair of out-of-bound vulnerability	18 buffer overflows	AutoPaG [86]	patches for 15 out of 18 vulnerabilities.
Security	mobile agents to detect, distribute, and apply vulnerability-handling tasks	simulation of a military network	Aurisch et al.[15]	high vulnerability-handling completion rates
Security	automatic Data Patch Generation for Unknown Vulnerabilities	25 vulnerabilities for which Microsoft has issued security bulletins between 2003 and 2006.	ShieldGen [31]	successfully generated effective patches for the majority of tested vulnerabilities
Security	enabling Fast Patch Propagation in Related Software Repositories	41,767 patches from 32 large repositories + 809 CVE patches	SPIDER [93]	able to identify 67,408 sps (safe patches) and that most of the CVE patches are sps
Security + Containerized Applications	security attack containment through detection and patching	31 real world security vulnerability exploits in 23 commonly used server applications.	Opatch [139]	can accurately detect and classify 81% vulnerability exploits
Security + Containerized Applications	runtime attack detection and dynamic targeted patching for security protection	31 real world vulnerability attacks in 23 commonly used server applications	Self-patch [140]	detect and classify 81% of attacks and reduce patching overhead by up to 84%
Mobile + Security	patch generation to prevent known vulnerabilities	16 real-world vulnerable Android apps	AppSealer [161]	successful on all vulnerabilities in dataset
Online Service Systems	suggest an appropriate healing action for a given new issue	243 issues of a large-scale product online service	Ding et al.[35, 36]	87% accuracy for healing action suggestion
Datacenters	predicts the required remediations for undiagnosed hardware failures	production repair tickets from Facebook datacenters.	Lin et al.[83]	deployed at Facebook

reduce this risk. By introducing a new file format called a Patch Object, they can provide more information about the structure and implications of patches. Jeong et al. [68] propose a newer tool designed for this same purpose. However, they focus on adaptive resource management. Their platform prioritizes system resilience and resource optimization by enabling imprecise computing and dynamic resource allocation in mixed-criticality systems which allows for different behaviour in an emergency hot fixing scenario. Another attempt to encourage users to use runtime patching is binary quilting [119]. This technique creates an entirely new reusable binary. With this, users can apply the minimum patch required without the unwanted side effects. Finally, also based on binary runtime injection techniques, band-aid patching [130] is a technique for testing hot fixes so that their deployment can be accelerated. This technique alters binaries so that upon reaching a patched code segment, the system initiates two execution threads that run both the modified and original code. It then retroactively selects the best execution path by evaluating factors like detected errors, past outcomes, and user feedback.

Some techniques have been proposed which are security vulnerability-specific. ProbeGuard [18] balances performance and security by hot fixing more powerful defenses when probing attacks occur. For string interpolation vulnerabilities, DEXTERJS [106] is a low-overhead technique that automatically generates patches to place on vulnerable sites. Xu et al. [153] propose a source code and binary level vulnerability detection and patching framework. It learns from the source code the function inputs that trigger the vulnerability and builds a filter to block them.

As previously explained, there is great value in hot fixing during runtime. However, many of the official patches on CVE do not allow for it. VULMET [154] is a tool that learns from existing official patches and generates hot fixes that can be deployed during runtime by using weakest precondition reasoning. Similarly, EMBROIDERY [164] transforms official CVE patches into hot fixes for a broad spectrum of Android devices. The main contribution of this tool is that it can be used to maintain obsolete Android systems and devices that often do not receive patches. A major hurdle with quick hot fixing for mobile devices is that Android partners require lengthy compatibility testing for the patches. InstaGuard [27] tackles this problem by bypassing the testing requirement through avoiding the injection of new code and relying on rule generation instead.

As for memory management, systems have evolved to address runtime vulnerabilities by identifying and patching memory errors in real-time. AutoPatch [121] is the first automated technique for hot fixing embedded devices with high availability on the fly. It performs an automatic analysis of the original patch, extracting its underlying semantics by leveraging predicate abstraction techniques. It then uses this information to generate a *hotpatch*—a patch that is semantically equivalent to the official version but designed to be applied seamlessly in a live production environment. Within memory management, for example, Exterminator [102] uses randomization to find memory errors in C/C++ programs and derive runtime patches for them. It works through three key components to detect, isolate, and correct memory errors. Firstly, they implement DieFast, a probabilistic debugging allocator, which detects errors by randomizing heap allocations. Secondly, a probabilistic error isolation algorithm analyzes heap snapshots to locate buffer overflows and dangling pointers, pinpointing the exact allocation and deletion sites. Finally, a correcting allocator then applies targeted runtime patches, adding padding to prevent overflows and delaying object deallocation to prevent dangling pointers. These runtime patches are tailored to each error's specifics and are applied both immediately in the running program and stored for future executions, continuously improving application reliability First aid [40] is also a system tailored for treating memory management bugs. It came after Exterminator to reduce the space and time overhead and allow for scaling. By rolling the program back to previous checkpoints, it is able to diagnose memory bugs. Following the diagnosis, First-aid then generates patches that prevent the memory bug and applies them during runtime.

More recently, in the era of large language models, we also see auto-remediation techniques utilizing them. Sarda et al. [123] propose a tool for the runtime automatic remediation of Ansible playbooks using in-context learning on pre-trained LLMs based on their custom Ansible remediation dataset. They achieve an average correctness of 98.86%, which hints at the applicability of these technologies to this problem space.

We present a summary and side by side comparison of all tools mentioned in Table 13.

Table 13. Summary of Runtime Hot Fix Generation Tools Applicable for Hot Fixing

Domain	**Function**	**Benchmark**	**Tool**	**Effectiveness**
Security	patching unsafe string interpolation	Alexa Top 1000 Websites	DEXTERJS [106]	patch hundreds of exploitable DOM-XSS vulnerabilities with a reasonable performance overhead
Security	monitors applications for signs of probing	SPEC CPU2006 benchmarks, Nginx web server, ApacheBench	Probe-Guard [18]	stops all tested probing-based exploit after the first detectable probe
Security	vulnerability detection and patching framework for source code and binary	each component evalauted separately existing existing work	Xu et al. [153]	can find 97 new recurring vulnerabilities for src and 71.0% accuracy for binaries
Security + Mobile	generating hot patches from official patches	373 Android kernel CVEs	Vulmet [154]	correct hot patches for 55/373 CVEs
Security + Mobile	transplants official patches of known vulnerabilities to different Android devices	two cross-platform Linux CVEs and vulnerabilities from Stagefright library	Embroidery [164]	tool fixes vulnerabilities in dataset (dataset is small though)
Security + Mobile	instant deployment of security no-code patches for mobile devices	CVEs from 2016 Android Security Bulletins	Insta-Guard [27]	can handle all evaluated critical CVEs
Security + real-time embedded devices	automatic hotpatching approach for real-time embedded devices	3 real CVEs from Zephyr and PicoTCP	AutoPatch [121]	can automatically generate hotpatches correctly based on the official patches
General software	software patch testing	N/A	Band-aid Patching [130]	preliminary proof-of-concept

Continued on next page

Domain	Function	Benchmark	Tool	Effectiveness
Microservice Applications	automatic remediation through generation of Ansible playbooks	custom-made Ansible-based remediation dataset	Sarda et al. [123]	average correctness 98.86%
Memory	automatically corrects heap-based memory error	SPECint2000 benchmark suite	Exterminator[102]	Squid buffer overflow identified and fixed; Mozilla heap overflow identified
Memory	surviving common memory management bugs and preventing their reoccurance	3 server applications (Apache, Squid, and CVS) and 4 desktop applications (Pine, Mutt, M4, and BC)	First-aid [40]	provides quick failure recovery and thereby hides program failures from user
Elf binaries	hot patching	N/A	Katana [114]	prototype presentation with no in-depth eval
ELF binaries	hot-patching tool	implemented a custom application	Jeong et al. [68]	show feasibility with exploratory scenarios
Linux on x86	apply the minimum patch for the targeted bug to avoid side effects	10 open source utility programs	Peanut [119]	reduced patch size by up to 90%

Answer to RQ2.2 (Autonomous tools for critical bug remediation): Hot fix remediation varies widely. Well-managed configuration processes are crucial for reliable and responsive software maintenance such as automation frameworks to improve efficiency and mechanisms for maintaining data integrity and stability during runtime changes. When time is critical, symptom mitigation techniques like re-execution in modified environments or rapid security workarounds can help contain issues quickly, though often at the cost of loss of functionality or performance. Offline hot fix generation techniques enhance service reliability by rapidly generating patches that apply upon reboot. Techniques range from using historical data to address recurring issues, to specialized tools for specific vulnerabilities. Runtime hot fix generation techniques focus on applying patches without disrupting system availability. These methods address a variety of use cases, from memory management and security vulnerabilities to embedded devices and mobile systems.

9.2.3 Deployment Tooling. An important consideration for hot fixing critical bugs is how the hot fix can ultimately be deployed efficiently after it is developed. There have been a number of works that tackle this problem from different perspectives. The techniques differ greatly depending on the target system into which the hot fix will be deployed i.e. Android systems, IoT infrastructures, cloud platforms, etc. In this section, we cover papers on the deployment of hot fixes into different types of production systems.

To allow software fixes and even features to be deployed more quickly, Van der Storm [141] [142] proposes a solution for automating the delivery of components in component-based software. When deploying software fixes to systems in production, the binary is modified and thus a restart is often required. This leads to downtime of the system which is often disruptive, especially in user-facing and mission-critical systems. There has been a lot of research into how patches can be integrated during run-time without causing the system to become unavailable as a result of the update.

As explained in Section 2, this activity is often referred to as hot patching. As discussed, since a survey on hot patching exists, our scope only includes work that hot patches time-critical issues, according to our definition of hot fix. Payer et al. [107] call this a problem of "as soon as possible" (ASAP) repair. They investigate the feasibility of runtime patching by investigating whether patches for critical bugs from the Apache web server can be dynamically applied. They find that dynamic update mechanisms are feasible and highly effective. Katana [114] is one such dynamic update mechanism. Katana specifically hot patches ELF binaries by creating *patch objects* which the user can apply to running processes. The authors of this paper also address an important concern that many owners have regarding hot patching. As this type of updating mechanism is less common, these techniques are often viewed as at risk of leaving the system in an inconsistent state. Russinovich et al. [116] present an optimization to live migrations of virtual machines that does not require turn space, requires minimal CPU, and no network while preserving VM state and causing minimal VM blackout.

Users are sometimes reluctant to update their system once a hot fix has been deployed due to the risk of software releases becoming incompatible and breaking their system. Saieva et al. [120] propose alleviating this friction through a technique called *Binary Patch Decomposition*. This technique provides the users with extra context around the distributed fixes which allows them to select and integrate the compatible pieces of the update. To increase the robustness of upgrades but in the context of rolling upgrades on cloud platforms is an approach named R^2C [132]. This approach offers early error detection of rolling upgrades as well as risk and time completion predictions.

Several works address the deployment of hot fixes for Android systems specifically. InstaGuard [27] is a hot patching approach for Android that bypasses the slow compatibility testing processes of Android device partners to facilitate immediate patching for critical security vulnerabilities. InstaGuard makes this possible by avoiding the injection of new code in the patch and enforcing instantly updatable rules instead. Ford et al. [38] discuss two tools that tackle the same problem as Instaguard by bypassing the traditional mobile app patching lifecycle. They find that while both of these tools do enable quick updates, they expose the users to many security vulnerabilities. PatchDroid [100] is a tool that distributes and applies Android security patches. It enables safe in-memory patching in a scalable way such that a patch can be written once and deployed to all affected versions. Socio-Temporal Opportunistic Patching (STOP) [133] also enables the delivery of security hot fixes to mobile devices. It is a two-tier system that collects co-location data of mobile devices then targets the delivery of hot fixes to a subset of these devices. From there, the patch can be spread by these devices opportunistically. LEONORE [146] [145] is a large-scale IoT deployment framework that allows for the provisioning of components on resource-constrained edge devices. Araujo et al. [13] propose a patch management model for rapid deployment of security patches during runtime. Their approach includes patch testing and recovery in the case of an incompatible patch.

In complete contrast with the idea of patching during runtime, Candea et al. [22] highlight the importance of recursive restartability. In other words, the ability of the system to handle restarts at multiple levels. This is because many nondeterministic bugs do not necessarily need to be patched. They can be solved by simply rebooting. In their paper, they highlight the required properties for recursive restartability and outline steps for beginning to adopt this in software systems. In a continuation paper [23], they discuss microrebooting. The idea is to recover the faulty application components without affecting the rest of the application. We present a summary and side by side comparison of all tools mentioned in Table 14.

Table 14. Summary of Deployment Tools Applicable for Hot Fixing

Domain	**Function**	**Benchmark**	**Tool**	**Effectiveness**
IoT	elastic app provisioning on constrained, heterogeneous edge devices in large-scale IoT deployments	real-world IoT deployment from one of their industry partners	LEONORE [146]	able to elastically provision large numbers of devices using a testbed based on a real-world industry scenario.
Security	rapid deployment of JIT security patches	6 server applications: ApacheHTTP, nginx, bind, sendmail, samba, vsftpd	INSIDER [14]	Insider is stable and incurs a small overhead
Mobile + Security	instant deployment of security no-code patches for mobile devices	CVEs from 2016 Android Security Bulletins	InstaGuard [27]	can handle all evaluated critical CVEs
Mobile + Security	distribute and apply third-party security patches for Android	multiple Android devices running different Android versions that contain a number of known vulnerabilities.	PatchDroid [100]	effectively fixes security vulnerabilities on legacy Android devices without noticeable performance overhead
Mobile + Security	method for users to receive immediate updates	no evaluation	JSPatch/ Rollout.io [38]	no evaluation
Mobile + Security	patching of Short Range Mobile Malware	3 traces of real mobile device contacts carried by human	STOP [133]	STOP can contain malware in a finite time in three different types of environments
Elf executables	hot patching	no evaluation	Katana [114]	no evaluation
Web servers	on-the-fly update system that provides ASAP repair	software updates released for Apache 2.2 between Dec 1st, 2005 and Feb 18, 2013	Payer et al.[107]	patching 45 of 49 bugs at runtime
Online service systems	robust rolling upgrade in clouds	testing in AWS and through a simulation	R2C [132]	early error detection is accurate

Continued on next page

Domain	Function	Benchmark	Tool	Effectiveness
General software	introduces recursive restartability to tolerate restarts at multiple levels	N/A	Candea et al.[22]	describes building RR systems in a systematic way
General software	workflow that allows developers to validate prospective patches and users to select which updates they would like to apply	constructed dataset of 21 bugs in widely used C/Linux programs	AT-TUNE [120]	successfully validated the real developer patches in both the developer and operator environments for 19 bugs and failed for 2
Cloud	minimizing application downtime while updates, including zero-day patches.	deployed in Microsoft Azure	VM-PHU [116]	deployed in Microsoft Azure
Component-based systems	incremental, binary updates for component-based software systems	Asf+Sdf Meta-Environment	Storm et al.[142]	show feasibility/POC
Component-based systems	agile and automatic release of software components	medium-sized software system, the Asf+Sdf Meta-Environment.	Van et al.[141]	the releases produced are correct with respect to the integration predicate
Component-based systems	recovering faulty application components, without disturbing the rest of the application	internet auction system running on an application server.	Microreboot [23]	fixed majority of failures in the evaluation system

Answer to RQ2.2 (Autonomous tools for hot fix deployment): Efficient deployment of hot fixes in production systems requires tailored approaches depending on the platform, such as Android, IoT, or cloud environments. Techniques like runtime patching and dynamic updates enable critical patches without downtime. Meanwhile, tools like InstaGuard and PatchDroid focus on bypassing slow testing processes in Android, facilitating faster deployment. Binary patch decomposition and rolling upgrade strategies offer additional flexibility, providing users with context to select compatible fixes and enhancing the robustness of deployments across diverse systems.

9.2.4 End-to-End Tooling. Our literature review concludes with a few works that outline end-to-end approaches to automate the detection, remediation, and deployment of hot fixes for critical software issues.
The earliest work we could find that tackles this is Huang et al. in 2005 [60]. The authors propose an automated hot fixing framework in which they are able to reason about the cause of a fault, apply simple remediation

patching, and do this during runtime without affecting the system's availability. They evaluate their technique on a small-scale within the domain of web-based applications.

For security, Data flow Analysis for Known Vulnerability Prevention System [113] is an end-to-end technique for protecting a target system at risk from security attacks. It detects attacks through dataflow analysis, examines the spread of the attack, and generates an appropriate vulnerability filter and hot fix which can be deployed at runtime.

As for Mobile applications, Gomez et al. [43] present a new vision for app stores. This new generation of app stores aims to automate several functionalities in mobile app maintenance to improve their quality and reduce human intervention. By utilizing user reviews, ratings, execution traces, and crash reports they aim to monitor and analyze the app and then automatically be able to generate, validate, and deliver patches. This new generation of app stores would work in a feedback loop model such that the newly delivered patches would result in new input data that they can then use to produce even more patches. They begin to realize their vision for Android devices in a continuation paper [52]. The framework was tested minimally on a single app in which they were able to patch a user-reported crash.

Wolverine [144] is an interesting tool that addresses the three phases of detection, repair, and deployment. However, it does this within the context of debugging sessions. By allowing the developer to step through the program states and visualize them, Wolverine aids with detection. It then implements a repair algorithm that synthesizes patches. To avoid having to abort the debug session, it allows for hot fixing suggested patches during runtime. It has been evaluated on somewhat small programs (student submissions and programs that implement known data structures).

Other tools have made it possible to create and apply targeted fixes directly within production environments, even without pre-existing test cases. These tools target specific bug types. Itzal [37] is an automated software technique that generates patches directly in the production environment. The novelty of this tool is that it removes the requirement of having a failing test case. Instead, it accesses the system state at the point of failure to conduct regression testing and patch search. The paper discusses a proof of concept prototype implementation which was evaluated on null dereference failures specifically. Such tooling can accelerate the process of generating hot fixes as it not only automates the patch generation process but is designed to work directly in production. AFix is another tool that automates this whole process [72] only targeting a specific bug type. AFix targets single-variable atomicity violations. It is first able to detect these bugs from bug reports. Using static analysis, AFix constructs suitable patches for the detected bugs. Finally, it attempts to combine multiple patches to improve performance and readability. It provides customized testing for each patch for validation. Sidiroglou et al. [131] also only look at a specific set of bugs, those that are recurring within the system. Using an instruction-level emulator before instruction execution, they are able to check for recurring faults and recover a safe control flow for the program. Finally, ClearView [108] is a five-step automated system for patching systems with high availability requirements. It learns invariants of the program behaviour, monitors for failures, identifies failures through invariant violations, generates patches to uphold the invariants by changing the state or flow of control, and observes the execution after patches are applied to select the most successful patch. As for evaluation, an external Red Team generated 10 code injection exploits to attack an application protected by ClearView. ClearView was able to block all attacks. It was able to generate patches that correct the behaviour of the system under attack in 7 out of the 10 cases.

These tools are inspiring, as they demonstrate the feasibility of automation in this domain and the implications that this can have on software development.

We present a summary and side by side comparison of all tools mentioned in Table 15.

Table 15. Summary of End-to-End Tools Applicable for Hot Fixing

Domain	**Function**	**Benchmark**	**Tool**	**Effectiveness**
Mobile	app store monitoring to automatically be able to generate, validate, and deliver patches	N/A	App store 2.0 [52]	paper presents a vision
Mobile	envision a new generation of app stores	PocketTool app	Smart App Store [43]	prototype initial results
General threaded programs	fixing single-variable atomicity violations from bug reports	8 real-world open-source multithreaded application	AFix [72]	fixed 97 out of 105 known atomicity violations
Windows x86 binaries	online patching of deployed software	10 code-injection exploits	ClearView [108]	generated patches for 7/10 exploits – detected all 10
Web applications	hot patching web apps	simulated ATM web app	Huang et al.[60]	proof-of-concept mini experiment
Security	detects and blocks exploits by performing dynamic taint analysis	no evaluation	DA-VPS [113]	no in-depth experimentation
Heap manipulating programs	debug-Repair environment: visualize program and generate hot patches	1600 buggy programs generated using fault injection	Wolverine [144]	could repair all the buggy instances in less than 5 seconds in most instances
Specific fault classes	detection and containment of specific faults: buffer overflows, illegal memory dereferences, and division-by-zero	set of exploits against popular server applications (Apache, OpenSSH, and Bind)	Sidiroglou et al.[131]	prevented recurrence of over 88% of tested software faults
General system	generate patches on-the-fly directly in production	N/A	Itzal [37]	prototype with limited evaluation

Answer to RQ2.2 (Autonomous end-to-end tools): End-to-End hot fixing tools demonstrate promising advancements in runtime patch generation, sometimes even without pre-existing test cases. The strengths of these tools include their innovative approaches to maintaining system availability during hot fixing and their effectiveness in addressing narrowly defined issues, such as security attacks or specific bug types, making them highly efficient in specific contexts. However, their main limitations are their narrow focus, with many tools only targeting specific bug types (e.g. atomicity violations or null dereferences) or small-scale applications (limited evaluation of the tools), and their limited scalability across diverse systems. These challenges suggest that further research is required to enhance the versatility and applicability of these tools for broader use in complex and varied software environments.

10 RQ3: Hot Fix Characteristics and Domains

To answer RQ3, we employed the surveyed tools as a lens to understand the characteristics of hot fixes and the domains in which they are applied.

The evidence shows that hot fixes most often address crashes, performance failures, memory errors, and security vulnerabilities. They are expected to be urgent and lightweight, enabling their application in runtime or near-runtime contexts. At the same time, they often rely on rollback or containment mechanisms so that systems can continue operating while a fix is applied. This is either through neutralizing an attack, masking a failure, or recovering execution without downtime.

Looking across domains, hot fixes appear most critical in high-availability, user-facing environments where downtime has significant consequences. These include cloud and large-scale online service systems (e.g., Azure, Microsoft Exchange, Facebook datacenters), where interruptions affect millions of users. Additionally in mobile ecosystems, where diverse devices must be quickly secured against vulnerabilities. Hot fixes also play a vital role in security-sensitive environments, where rapid mitigation of exploits and CVEs is essential. Beyond these, they are applied in embedded/IoT systems, datacenters, and enterprise software, showing that hot fixing is not tied to a single platform but consistently arises wherever resilience and continuity are paramount.

From the 107 tools we analyzed (spanning early 2000s to 2023), further characteristics of hot fixes emerge. They are typically urgent, localized, and domain-specific rather than broad or generic. The scarcity of end-to-end frameworks (fewer than 10, mostly published between 2005–2015 and minimally evaluated) illustrates that holistically automating hot fixing remains difficult. Instead, support for hot fixes has evolved around incremental, modular interventions, rapid detection, effective triage, targeted patch generation, and safe deployment rather than monolithic solutions. Hot fixes are also almost never fully autonomous due not only to technical difficulty but also to the required oversight for this class of high-stakes patching. Most approaches remain human-in-the-loop, requiring developer oversight during triage, validation, or rollback, which highlights the balance struck between urgency and caution.

Another characteristic that emerges from our study is how hot fixes are benchmarked and validated. We observed a strong divide. The first category that many tools demonstrated is that they are evaluated directly in production environments or on industry datasets. We saw error reports from Windows machines worldwide, incident logs from Microsoft Azure, Facebook datacenter repair tickets, or CVE vulnerabilities in Android and Linux. The second category of papers exist only as proof-of-concept prototypes with minimal or no real-world validation (e.g., App Store 2.0, Katana, Itzal). Even when synthetic benchmarks are used, they often attempt to reproduce high-stakes operational contexts, such as fault injection into Apache/MySQL servers or stress testing in distributed clusters. This divide between production-grade deployments and prototype-level work highlights both the practical urgency of hot fixing in industry settings and the technical difficulty of developing fully validated, end-to-end solutions for hot fixing.

The surge of recent activity in cloud, mobile, and security-related highlights that hot fixes are most necessary in environments where uptime and user trust are non-negotiable. Thus, a further recurring property is the focus on minimizing cost and overhead—whether through smaller patch sizes, low-latency deployment, or lightweight runtime monitoring so that fixes can be applied without disrupting running systems. Historically, earlier work in the 2000s emphasized rollback and fault tolerance, whereas recent research has shifted toward automation, AI/ML-driven detection, and large-scale cloud and mobile environments, showing how hot fixing evolves with deployment models. At the same time, tooling remains highly fragmented and domain-specific, with few general-purpose frameworks and limited standardized benchmarks, suggesting that effective hot fixes are inherently context-bound and that progress depends heavily on collaboration with industry environments.

Answer to RQ3 (Hot Fix Characteristics and Domains): Hot fixes are urgent, localized, and domain-specific, most often addressing crashes, vulnerabilities, memory errors, or performance failures in high-availability environments such as cloud, mobile, and online services. They rely on modular interventions: detection, triage, patch generation, and deployment rather than end-to-end automation, not only due to the technical challenge but also to keep humans in the loop to balance urgency with caution. Evaluations reveal a split between industry-grade production deployments and academic proofs-of-concept, underscoring that hot fixes are inherently context-bound and resource-intensive.

11 RQ4: Open Challenges

In this section, we answer RQ3 by identifying open challenges on the basis of the literature reviewed. We provide a summary of such challenges in Table 16, and describe each of them in detail below. These challenges are not explicitly enumerated in prior work. They have been inferred from the recurring gaps, absences, and limitations we observed across the studies we surveyed.

The Hot Fixing Vocabulary Challenge: Using an unified terminology to aid understanding of the state-of-the-art in the area and ease building upon existing work. In our previous work [55], we provided a detailed analysis of the inconsistencies in terminology found in the literature, within papers that explicitly define the term. In covering hot fixing papers in this study, we highlight that they rely on these conflicting definitions. One such example is the tool Wolverine [144] that defines hot patches as patches applied during debugging sessions, while AutoPatch [37] refers to them as automated patches designed to be semantically equivalent to official security patches but can deployed directly in the production environment. This difference can lead to confusion when comparing the two tools. This inconsistency complicates research comparisons, hinders progress in the field, and limits the development of universal tools or methodologies A unified terminology would make it easier to identify commonalities and distinctions, helping researchers and developers build on each other's work more systematically.

The Hot Fixing Benchmarking Challenge: Building a benchmark of hot fixes to aid research comparisons and ease progress in the area. A clear challenge that emerged through compiling and analyzing the existing work in this space is how difficult it was to compare the different tools. As it stands, there are no hot fix benchmarks that can be utilized by researcher to evaluate their tools against previous work. Existing tools are typically tested within isolated or specific environments (e.g., ClearView [108] with code injection exploits and Itzal [37] with null dereference failures). These limited benchmarks prevent a clear comparison across tools. Across several tools, the absence of standardized benchmarking prevents a clear and fair comparison of their effectiveness, especially when considering factors like error types, platform compatibility, real-time performance, scalability, and impact. These inconsistencies in benchmarking hinder progress in the field by making it difficult to assess the overall state of the art in automated hot fixing. Developing a standardized benchmark for evaluating hot-fixing tools could enable objective comparisons, accelerating advancements by providing a clear framework for performance metrics and comparison.

The Hot Fixing Taxonomy Challenge: Creating a taxonomy for critical bug classes such that remediation strategies can effectively target them to drive research toward more advanced remediation tooling. Many tools

Table 16. Open challenges in hot fixing.

Open Challenge	Priority	Impact
The Hot Fixing Vocabulary Challenge	1	Unified definitions will ensure that future empirical work, benchmarks, tooling, etc address the correct problem.
The Hot Fixing Benchmarking Challenge	2	Specialized benchmarks are needed to understand the current state-of-the-art and eventually to evaluate the performance of hot fixing tooling that will be built.
The Hot Fixing Taxonomy Challenge	3	Outlining a taxonomy for critical software bugs that hot fixing targets will ensure that tooling is sufficient and complete.
The Hot Fixing Industrial Practice Challenge	4	Understanding the current industrial practices will aid in understanding the pain points and help in building useful tooling. Additionally, candid conversations with developers who work on hot fixing are essential to creating a body of work that is useful to them.
The Hot Fixing Distribution Challenge	4	Gaining a deeper understanding of hot fixing patterns will help systematize this activity.
The Hot Fixing Impact Challenge	4	Critical issues are the targets of hot fixes. Thus, studying their impact is integral to developing successful hot fixing tooling.
The Hot Fixing Tooling Challenge	5	There are many directions for future work in developing hot fixing tooling. Such tooling can make hot fixes faster to develop and deploy and lower its risk on the production environment.
The Hot Fixing Predictability Challenge	6	Accelerate the process of identifying critical issues and thus accelerating their hot fixing process.

target distinct types of bugs (e.g., AFix [72] for atomicity violations, Itzal [37] for null dereference failures), but there's no cohesive taxonomy of critical software bugs that would allow for connecting critical bug classes to remediation strategies. This absence makes it difficult for researchers to generalize findings and identify tools best suited for particular bug types. A taxonomy could drive more focused tool development by clarifying which types of bugs are critical and mapping each to effective remediation methods. In this paper, we provide a taxonomy of the hot fixing studies found in the literature. However, a comprehensive taxonomy detailing the critical bug classes such that hot fixes can address them is still missing. Such a taxonomy would allow us to more accurately identify gaps in tooling and match specific tools to the relevant critical bug categories.

The Hot Fixing Industrial Practice Challenge: Conducting more empirical studies on industrial practices in terms of automated strategies and manual intervention for detecting, remediating, and deploying hot fixes. Most tools are evaluated on small-scale or controlled cases (e.g. app store model on a single app [52]), meaning there's limited knowledge of how these techniques would fare in real-world, large-scale industry settings. Additionally, Conducting questionnaires and surveys with industrial practitioners can play a crucial role in ensuring that hot fixing tools are developed for the correct use cases by providing direct insights into real-world practices, needs, and challenges. Industrial practitioners deal with a wide variety of issues that may not be captured in academic research (e.g. specific high-priority use cases, resource constraints, and availability requirements). They can highlight gaps in their workflow and provide valuable feedback on future tool development and integration, ensuring that the tools are not only technically effective but also align with the needs and processes of current industry standards

The Hot Fixing Distribution Challenge: Providing details on the cost of hot fixing in software enterprises and the average frequency of hot fix releases to aid future research and understand the hot fixing cost in practice. The costs associated with hot fixing in enterprise environments, such as resource allocation or frequency of hot fix releases, are rarely addressed in current research. Since tools like Exterminator [102] and First-aid [40] highlight the runtime and space overheads of their approaches, understanding the cost trade-offs in real-world settings could guide tool development to balance performance with minimal impact on resources. While tools focus on specific bug types and patching methodologies, they tend to overlook the operational costs involved in applying patches at scale. Issues such as the time to generate and apply patches, resource consumption during patching, and the impact on system performance are less clear making the advantage of using automation over standard manual processes less convincing. As the research community develops these tools further, a more detailed exploration of the costs involved in hot fixing would help in creating tools that minimize disruptions to live systems and optimize the use of resources.
The Hot Fixing Impact Challenge: Measuring the impact of a critical issue and predicting the best remediation strategy for it as a first step towards automating hot fixing. This includes automation for quantifying the time allotted to find the fix, the issue's direct impact on users, and the desired type of resolution. We have found that the impact of an issue on end-users, as well as the time-sensitivity of a resolution, is not fully captured when considering the business-criticality aspect of hot fixing. A framework that evaluates an issue's urgency and impact on users before suggesting a remediation approach could improve the prioritization of hot fixes in dynamic production environments.
The Hot Fixing Tooling Challenge: Developing specialized tooling for hot fixing for all of the phases that we identified in this survey: identifying critical issues, remediating these issues, and finally deploying the hot fixes as quickly as possible. Automation and the creation of end-to-end specialized tooling here opens up many directions for future work. Existing hot fixing tools generally excel at isolated phases of the process (e.g. Triage [137] for failure diagnosis or InstaGuard [27] for immediate deployments in Mobile app stores). However, a truly comprehensive hot fixing tool would need to cover the entire lifecycle—detecting critical issues, implementing precise fixes, efficiently deploying patches in real-time, and measuring their post-deployment impact to ensure stability and effectiveness. While we have presented some tools that attempt at integrating multiple phases (detection, repair, and deployment), they remain limited to specific bug patterns lack the full spectrum of capabilities needed for production-scale systems. For instance, Exterminator [102] specializes in memory errors, while AFix [72] focuses on crafting patches for atomicity violations. Building more robust, end-to-end tooling could streamline workflows for engineers and reduce both manual intervention and potential delays, ultimately enhancing the reliability and responsiveness of production systems. Expanding tooling to include post-deployment monitoring for impact assessment, in particular, would provide valuable insights into hot fix effectiveness, making it easier to iterate on solutions.
The Hot Fixing Predictability Challenge: Exploring the power of existing defect and vulnerability prediction tooling on the set of bugs that have been hot fixed, to understand their ability to detect such bugs. Defect prediction tools have not been widely tested on bugs specifically requiring hot fixes. By examining how well these tools can predict bugs typically addressed by hot fixes, the industry could better allocate resources toward bugs with a high likelihood of needing urgent remediation, leading to more proactive issue resolution.
Although these open challenges may appear to be fundamental recommendations, they have yet to be fully explored in the context of hot fixing, further highlighting the need for more research in the area. This makes these challenges even more paramount as we find that many of the building blocks for facilitating advancements in research within the area remain untapped. We hope that these recommendations highlight the need for this foundational research to the research community.

Answer to RQ4 (Open challenges): We identified 8 open challenges in hot fixing research from our literature review. These challenges include establishing consistent terminology, creating robust benchmarks, clarifying the taxonomy of critical bugs that hot fixes target, conducting more industrial studies, understanding both system and financial impacts, developing comprehensive end-to-end tooling, and predicting the need for hot fixes. For each, we outline key motivations from existing work, assign a priority level, and assess its expected impact on the community.

12 Discussion and Reflections

In this section we expand on the aforementioned challenges through a detailed discussion and reflection on the existing body of work. We also point out directions for future work.

12.1 Terminology

It is very evident from this review that the terminology used in different papers is sometimes conflicting. This has created a disjointed body of research. A first step in advancing the research on hot fixing is unifying the terminology and reflecting on how all of the papers that address this topic fit together – our aim with this comprehensive review. Moreover, some of the existing work included in our review is applicable to hot fixing, since it addresses the treatment of critical bugs, even though it does not explicitly mention any hot fixing terminology. Thus, we urge the community to not only produce more research directed at hot fixing but also to pay careful attention to consistency of terminology so as to streamline the acquisition of collective knowledge. To facilitate this effort, we have proposed a unified definition for hot fix (see Definition 1). This definition guided the scope of search for our survey. During our literature search and screening process, we reviewed a large number of papers that referenced or described the hot fixing phenomenon. Across this material, three conditions consistently emerged as the clearest way to operationalize our definition in practice, as follows.
An update in production qualifies as a hot fix **if and only if** it satisfies **all** of the following conditions:

(1) **Trigger condition:** The change is initiated reactively in response to an unplanned disruptive, high-impact issue in production.
(2) **Development condition:** The issue creates a severely constrained time window for mitigation, limiting the extent of engineering activities normally performed.
(3) **Deployment condition:** Immediate release requirement: deployed at the earliest operationally viable moment and is prioritized above current tasks or other defects, due to its time-critical nature.

A software update that does not satisfy any one of these three conditions is not a hot fix, regardless of domain or release cadence. These conditions provide concrete guidance for researchers and practitioners to draw the boundary of when an update is a hot fix. Table 17 illustrates the boundary of the definition by demonstrating updates that qualify as hot fixes and updates that fail one or more of these conditions. Each example reflects a real software-engineering scenario and shows precisely why it does or does not qualify as a hot fix, reinforcing the necessity of satisfying all three conditions jointly. These conditions distinguish the impact and severity (condition 1), responsiveness and pressure (condition 2), and the release immediacy (condition 3), ensuring that only changes requiring rapid development and rapid deployment due to a disruptive production incident qualify as hot fixes. These complement our intentionally broad definition while preserving its ability to capture the phenomenon across domains.

Table 17. Examples of updates that **qualify** and **do not qualify** as hot fixes.

Update	**Failed condition**	**Hot fix because...**
Faulty configuration flag causes production crash	N/A	Triggered by a disruptive failure, fixed under severe time pressure, and deployed immediately to restore service.
Zero-day security critical vulnerability	N/A	Reactively addresses a high-impact security incident, prepared rapidly, and released as soon as operationally viable.
Severe performance regression under peak load	N/A	Production users experience critical degradation; engineers produce a minimal targeted fix quickly and deploy it right away.
		Not a hot fix because...
Feature release to meet customer deadline	(1)	Urgency is driven by a customer deadline rather than high-impact issue.
Slow-progress data corruption in production	(2)	A disruptive production failure exists, but its gradual progression allows development to occur without a constrained time window.
Urgent fix delayed for marketing	(3)	A severe issue is fixed urgently, but deployment is paused for marketing-led incident messaging, effectively removing the urgency.
Immediate routine enhancement	(1) & (2)	Not incident-driven and not urgent, yet deployed immediately.
Deadline-driven feature change in batched release	(1) & (3)	Not incident-driven and not deployed immediately, though developed urgently.
Severe issue delayed pending resource availability	(2) & (3)	A disruptive failure exists, but development does not begin immediately and deployment is postponed due to resource constraints.
Routine feature update	(1) & (2) & (3)	No incident, no urgency, and not deployed immediately.

12.2 Benchmarks

From here, an essential requirement for driving research on a specific topic forward is reliable benchmarks. In our review, we were not able to find any such large-scale benchmarks on hot fixing. We believe that having a collection of real-world hot fixing instances can help in better understanding this software engineering activity as observed in production. As explained in section 9.2.2, we were able to identify multiple remediation techniques depending on the type of critical issue. Thus, a larger benchmark is needed here to realistically capture the different hot fix types depending on the granularity (e.g. build, configuration, source code) as well as the robustness of the fix (e.g. workaround, root cause fix). Such a benchmark can help in determining the effectiveness of existing tooling so that research efforts can be directed more productively. Tens of millions of public repositories exist on GitHub[3] which can be utilized in creating it. Taking this a step further, it would be valuable to collaborate with industrial stakeholders to create benchmarks from larger-scale products. As the size of the product grows, prioritizing the issues and detecting the most critical ones inevitably becomes a more difficult task. Generating a hot fix for these issues becomes more complex as there are likely more dependencies that would need to be taken into account.

Thus, we hypothesize that the scale of the target system would have some effect on what constitutes a hot fix and what the classes of hot fixes would be.

12.3 Hot Fixing Practices

In open-source empirical research on hot fixing, mobile applications and video games have been the most studied assessing the types of updates and the bugs that they target (Section 8.1). Assessing these properties in different domains, such as security, cloud, web, and machine learning models, can be further explored in the literature. We believe that this is essential for drawing wider conclusions from the empirical results.
Multiple smaller-scale studies have been conducted in industrial settings. There has been industry-partnered research in which hot fixing is mentioned but is not the central topic of the work [158]. In the future, we hope to see larger-scale studies on hot fixing efforts in real-world applications. Moreover, we found that there has not been large-scale industrial adoption for hot fix automation tooling as of yet. We hope that future research on hot fixing creates a positive feedback loop in industrial settings.
The literature lacks detail on the cost of hot fixing in software enterprises, the average frequency of hot fix releases, and a structured taxonomy on the critical bug types that they target. This quantitative information is important for incentivizing companies to invest in hot fix automation tooling and further understanding the level of automation that is actually required to optimize productivity and user satisfaction.
Finally, while surveys and interviews have been conducted with end users [143] and system administrators [79] there is still a gap in conducting such human empirical work with the software developers. Understanding patterns in hot fixing activities from their perspectives can bring the required insight for understanding the prevention of bugs targeted by hot fixing.

12.4 Semi-Automated Tooling

Semi-automated tooling whose purpose is to aid system administrators with treating critical crises needs to mimic the user journey of the administrator. In this section, we break down the user journey into its different stages and address possible future research directions given existing work as presented in Section 9.1.
The administrator needs to get notified of the crisis in a user-friendly way to increase productivity. Techniques for localizing logs that demonstrate the incident via log clustering [163] and system KPIs [57] make this possible. They then need to know the impact of the crisis. This includes quantifying the number of users that it affects, whether backups exist, the criticality of the features/projects affected, and so on. This will inform the amount of time that they have to resolve the crisis. We found tooling for tracing dependencies [19], summarizing a data center's state using system metrics [20], and effort prioritization given a collection of error reports [42]. We believe that there remains potential here in taking this a step further. We envision a tool that brings these tools together. Given the dependency tracing, the system's current state, and effort prioritization, we need to automate quantifying the time allotted to find the fix, the issue's direct impact on users, and the desired type of resolution (compromising features, finding the root cause solution, securing the system, etc.), assigning the task to the most qualified administrator for the task, as well as providing the required data for the following step.
From here, they will begin resolving the crisis. The resolution will be affected by the criticality of the crisis and the time allotted to resolve it. As such, for crises that need immediate resolutions, workarounds will be implemented whereas for crises that allow for more time, a true resolution might be achieved. Automated tools that detect recurring crises [151] [19] and suggest resolutions for them can greatly help in increasing the turnaround time as well as increase productivity so that they do not have to resolve the same issue multiple times. Automated repair techniques exist [44], however, none target the needs of system administrators directly. In the future, we would like to see a clear taxonomy of the types of bugs that are observed by these individuals to help drive research

forward. We expect that the different classes of critical bugs require vastly different remediation strategies. Given the taxonomy, we will hopefully begin to see more specialized automated repair tooling for system administrators. Once a resolution has been reached, it must be deployed as quickly as possible ideally without causing any downtime to the target system. We were able to find a tool for measuring the impact of an update for better system update safety [148]. From here, verifying that the resolution did indeed resolve the crisis is required. There is a gap here in the literature. We envision an automated tool here dedicated to managing recently deployed resolutions. Specifically in tooling for verifying the satisfaction with the deployed fix, measuring its impact post-deployment, and specifying whether it needs to be improved at some point into a permanent fix, when this fix would need to take place, and who would be responsible for undertaking it.

The Microsoft Service Analysis Studio [90] is the closest tool we found in this domain to more advanced system administrator tooling. However, this paper was published over 10 years ago. Since the internal process for software engineering tasks are ever-changing and advancing with the growing size and complexity of systems, we believe bringing such a study up to date can be extremely valuable. Thus, we encourage the community to publish research in this area so that we can begin to see a new generation of such tools targeted specifically at time-critical software issues. In a follow-up paper [91], the authors reflect on their experience with this system and the lessons learned along the way. To demonstrate the value of the system early, the team took a step-by-step approach to manage engineering resources rather than building out the service all at once. They emphasize the importance of maintaining the system's robustness, performance, and availability and discuss the algorithms that they adopted in order to do so.

12.5 Detection Tooling

We were able to find detection of critical software issues in the context of performance degradation, functional defects, and security vulnerabilities. We specifically only wanted to include tools that mention identifying ritical issues. Our search concluded with only 26 papers, suggesting that more research in this area is needed.

None of the studies that we found in the context of detecting critical issues specifically target prediction. There is abundant research on vulnerability [126] and defect prediction [134]. However, through this review, we were not able to see a clear correlation between those works and hot fixing. We believe that there is untapped potential in the applicability of these prediction techniques for this context.

12.6 Remediation Tooling

For the purpose of hot fixing, there is no one-size-fits-all solution. A fully optimized remediation framework for critical software issues must sit on top of all of the remediation strategies that we mention in Section 9.2.2: reconfiguration, symptom mitigation (workarounds), as well as offline and online hot fix generation. Depending on the nature of the critical issue, the correct remediation strategy must be applied. This will depend on the type of bug, its impact, the complexity of the resolution, the availability requirements of the system that the issue resides in, and so on. The major gap in the literature that we see here is not necessarily in advancing each of the remediation strategies. We believe that the first step that needs to be tackled here is deepening the collective understanding of what strategy needs to be applied and when. This can then be unified in a switch case-like system that automates the assignment of each critical issue to the correct strategy and launches that corresponding automated remediation technique.

12.7 Deployment Tooling

Despite the abundant existing research on runtime patching of software, these techniques are not often applied in practice. One of the major reasons is that software developers and maintainers view them as high-risk for breaking the target systems. Thus, we think that more human-studies are required to understand the needs of these individuals in order to build tools that they can put their trust in. More empirical research on the success of these techniques can help in shifting their perspective as well. Finally, simulation environments can help validate the runtime patch before deployment.

12.8 End-to-End Tooling

One of the main takeaways we wish to leave you with in this survey is that end-to-end tooling for automating hot fixes is lacking. We were only able to find four tools that address all three phases of treating critical software issues. However, these tools are either within a specific domain (i.e., mobile [43], web apps [60]), or target a specific kind of critical software issue (i.e. single variable atomicity violations [72], online debugging [144]). We believe that there is a big gap in the literature here. This task of hot fixing critical software bugs is needed in every system in production. Directed research on this specific task and how it can be partially/fully automated will have an immense impact on the software engineering community and a wide variety of applications in industrial settings. We encourage researchers to further expand the existing research into different use cases and domains. As explained in the survey methodology (Section 6), our scope encapsulates work that targets hot fixing activities specifically. However, much bug detection, remediation, and deployment tooling exists that are not targeted at critical software issues or hot fixing. Comparative studies that analyze the efficacy and efficiency of this more general tooling but in our use case, as opposed to hot fixing-specific tooling, would be useful to better understand the current state of the art.

13 Threats to Validity

The relevance filtering of the papers in the search was conducted by one author of this paper. As this can be subject to judgment bias, a second author of this paper acted as a second judge on a sample of the papers to measure agreement. A 10% random sample of the irrelevant papers and 10% random sample of the relevant ones were selected and shuffled. The second judge then manually assessed and filtered the papers based on the scope of the literature review.

The second author's judgment matched that of the first judge in all but two instances. There were two papers that were included in the scope of the survey that the second author deemed as irrelevant. These were papers that did not explicitly mention hot fixing or addressing critical software issues. Instead, they present transferable techniques that can be applicable to the problem as they automate the detection of bugs and the deployment of patches into the end system. Thus, we conclude that the judgment of the author who collected the papers does not miss any relevant work. However, they were more lenient with their inclusion criteria which might have resulted in more papers getting included in this survey.

In our primary search, we use two keywords: *hot fix* and *hot patch*. We attempted to look for synonyms of these keywords in the IEEE thesaurus [62] but were unable to find any. Thus for due diligence, we conducted a primary search on keywords we see as related to the scope of this literature review. We conducted this search on one search engine only to gauge relevance which was IEEE Xplore. We experimented with the following keywords: *critical bug fix*, *critical bug patch*, *critical fix*, and *critical patch*. We found that the number of search results returned was very small (3, 0, 14, and 51 respectively). In addition, only 5 papers were relevant to our scope and out of these 5 papers none were key papers that would add substantial additional knowledge to the contents of this review.

Finally, the categorization of the papers is subject to bias. We mitigate this by following the thematic analysis [21] procedure used in qualitative research to conduct this in a more structured manner.

14 Conclusions

We have presented an overview of existing work on hot fixing for software systems. Hot fixing is a fundamental software engineering task for systems in production. However, we found that the number of publications on the topic remains limited and the terminology is yet to be streamlined. We believe that hot fixing both in industry and research has the potential to become a more systematic and established software engineering activity, and less of an afterthought than it is viewed today. With this paper, we wish to ignite more research in the area, especially toward automation. In the future, we hope to see community effort in building benchmarks and conducting both quantitative and qualitative empirical studies on current practices. Following a deeper understanding of the state of the art and current challenges, we then hope to start seeing the process become more automated.

Acknowledgments and Copyright

We would like to thank those authors who provided comments and feedback on earlier drafts of this paper. We would also like to thank Prof. Mark Harman for his valuable input at various stages of this work. This work was supported by the UKRI EPSRC Fellowship EP/P023991/1 and the ERC Advanced Grant No.741278. For the purpose of open access, the authors have applied a Creative Commons Attribution (CC BY) license to any accepted manuscript version arising. ChatGPT was utilized to improve the clarity and readability of certain sections of the manuscript. However, all ideas, arguments, and conclusions presented were entirely developed by the authors of this paper.

References

[1] ACM Digital Library. https://dl.acm.org/. Accessed: 2023-11-17.

[2] dblp: computer science bibliography. https://dblp.org/. Accessed: 2023-11-17.

[3] Github: Let's build from here. https://github.com/. Accessed: 2023-11-17.

[4] IEEE Xplore. https://ieeexplore.ieee.org/Xplore/home.jsp. Accessed: 2023-11-17.

[5] ScienceDirect. https://www.sciencedirect.com/. Access: 2023-11-17.

[6] Hotfix notes for version 10.109.3.13. https://help.jiraalign.com/hc/en-us/articles/8249094221716-Hotfix-Notes-for-10-109-3-13, 2025. Accessed: 2025-08-27.

[7] Agarwal, A., and Garg, N. K. Effective test strategy model for ensuring ftr of hot-fix. *ICROIT 2014 - Proceedings of the 2014 International Conference on Reliability, Optimization and Information Technology* (2014), 40–43.

[8] Ahmed, B. H., Lee, S. P., Su, M. T., and Zakari, A. Dynamic software updating: a systematic mapping study. *IET Software 14*, 5 (2020), 468–481.

[9] Ahmed, T., Ghosh, S., Bansal, C., Zimmermann, T., Zhang, X., and Rajmohan, S. Recommending Root-Cause and Mitigation Steps for Cloud Incidents using Large Language Models. *Proceedings - International Conference on Software Engineering* (2023), 1737–1749.

[10] Alshahwan, N., Ciancone, A., Harman, M., Jia, Y., Mao, K., Marginean, A., Mols, A., Peleg, H., Sarro, F., and Zorin, I. Some challenges for software testing research (invited talk paper). In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (New York, NY, USA, 2019), ISSTA 2019, Association for Computing Machinery, p. 1–3.

[11] Alshahwan, N., Harman, M., and Marginean, A. Software testing research challenges: An industrial perspective. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)* (2023), pp. 1–10.

[12] Anderson, J., Salem, S., and Do, H. Striving for failure: An industrial case study about test failure prediction. *ICSE* (2015).

[13] Araujo, F., and Taylor, T. Improving cybersecurity hygiene through jit patching. *ESEC/FSE* (11 2020), 1421–1432.

[14] Araujo, F., Taylor, T., Zhang, J., and Stoecklin, M. P. Cross-stack threat sensing for cyber security and resilience. *Proceedings - 48th Annual IEEE/IFIP Int. Conf. on Dependable Systems and Networks Workshops, DSN-W 2018* (7 2018), 18–21.

[15] Aurisch, T., and Jacke, A. Handling vulnerabilities with mobile agents in order to consider the delay and disruption tolerant characteristic of military networks. *Int. Conf. on Military Communications and Information Systems* (6 2018), 1–7.

[16] Bansal, C., Renganathan, S., Asudani, A., Midy, O., and Janakiraman, M. DeCaf: Diagnosing and triaging performance issues in large-scale cloud services. *Proceedings - International Conference on Software Engineering* (jun 2020), 201–210.

[17] Barrett, R., Kandogan, E., Maglio, P. P., Haber, E. M., Takayama, L. A., and Prabaker, M. Field studies of computer system administrators: analysis of system management tools and practices. *CSCW '04: Proceedings of the 2004 ACM Conf. on Computer supported cooperative work* (11 2004), 388–395.

[18] Bhat, K., Kouwe, E. V. D., Bos, H., and Giuffrida, C. Probeguard: Mitigating probing attacks through reactive program transformations. *Int. Conf. on Architectural Support for Programming Languages and Operating Systems - ASPLOS* (4 2019), 545–558.

[19] Bodík, P., Fox, A., Jordan, M. I., Patterson, D., Banerjee, A., Jagannathan, R., Su, T., Tenginakai, S., Turner, B., Ingalls, J., Lab, R., Berkeley, U. C., and University, S. Advanced tools for operators at amazon.com. *Hot Topics in Autonomic Computing (HotAC)* (2006).

[20] Bodík, P., Goldszmidt, M., Fox, A., Woodard, D. B., and Andersen, H. Fingerprinting the datacenter: Automated classification of performance crises. *EuroSys'10 - Proceedings of the EuroSys 2010 Conf.* (2010), 111–124.

[21] Braun, V., and Clarke, V. Thematic analysis. *APA handbook of research methods in psychology, Vol 2: Research designs: Quantitative, qualitative, neuropsychological, and biological.* (3 2012), 57–71.

[22] Candea, G., and Fox, A. Recursive restartability: Turning the reboot sledgehammer into a scalpel. *Proceedings of the Workshop on Hot Topics in Operating Systems - HOTOS* (2001), 125–130.

[23] Candea, G., Kawamoto, S., Fujiki, Y., Friedman, G., and Fox, A. Microreboot-a technique for cheap recovery.

[24] Chen, J., He, X., Lin, Q., Xu, Y., Zhang, H., Hao, D., Gao, F., Xu, Z., Dang, Y., and Zhang, D. An Empirical Investigation of Incident Triage for Online Service Systems. *Proceedings - 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP 2019* (may 2019), 111–120.

[25] Chen, J., He, X., Lin, Q., Zhang, H., Hao, D., Gao, F., Xu, Z., Dang, Y., and Zhang, D. Continuous incident triage for large-scale online service systems. *Proceedings - 2019 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019* (nov 2019), 364–375.

[26] Chen, J., Zhang, S., He, X., Lin, Q., Zhang, H., Hao, D., Kang, Y., Gao, F., Xu, Z., Dang, Y., and Zhang, D. How Incidental are the Incidents? Characterizing and Prioritizing Incidents for Large-Scale Online Service Systems. *Proceedings - 2020 35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020* (sep 2020), 373–384.

[27] Chen, Y., Li, Y., Lu, L., Lin, Y.-H., Vijayakumar, H., Wang, Z., and Ou, X. Instaguard: Instantly deployable hot-patches for vulnerable system programs on android. *Network and Distributed System Security Symposium* (2018).

[28] Chen, Y., Xie, H., Ma, M.-J., Kang, Y., Gao, X., Shi, L., Cao, Y., Gao, X., Fan, H., Wen, M., Zeng, J., Ghosh, S., Zhang, X., Zhang, C., Lin, Q., Rajmohan, S., and Zhang, D. Empowering Practical Root Cause Analysis by Large Language Models for Cloud Incidents. *arXiv.org* (2023).

[29] Chen, Y., Yang, X., Lin, Q., Zhang, D., Dong, H., Xu, Y., Li, H., Kang, Y., Zhang, H., Gao, F., Xu, Z., and Dang, Y. Outage prediction and diagnosis for cloud service systems. *The Web Conference 2019 - Proceedings of the World Wide Web Conference, WWW 2019* (2019), 2659–2665.

[30] Chen, Z., Kang, Y., Li, L., Zhang, X., Zhang, H., Xu, H., Zhou, Y., Yang, L., Sun, J., Xu, Z., Dang, Y., Gao, F., Zhao, P., Qiao, B., Lin, Q., Zhang, D., and Lyu, M. R. Towards intelligent incident management: why we need it and how we make it. *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (nov 2020), 1487–1497.

[31] Cui, W., Peinado, M., Wang, H. J., and Locasto, M. E. Shieldgen: Automatic data patch generation for unknown vulnerabilities with informed probing. *Proceedings - IEEE Symposium on Security and Privacy* (2007), 252–266.

[32] Czerwonka, J., Das, R., Nagappan, N., Tarvo, A., and Teterev, A. Crane: Failure prediction, change analysis and test prioritization in practice - experiences from windows. *Proceedings - 4th IEEE Int. Conf. on Software Testing, Verification, and Validation, ICST 2011* (2011), 357–366.

[33] D'Ambros, M., Lanza, M., and Pinzger, M. A bug's life visualizing a bug database. *Int. Workshop on Visualizing Software for Understanding and Analysis* (2007), 113–120.

[34] Dijkstra, E. W. Structured programming.

[35] Ding, R., Fu, Q., Lou, J. G., Lin, Q., Zhang, D., Shen, J., and Xie, T. Healing online service systems via mining historical issue repositories. *ASE* (2012), 318–321.

[36] Ding, R., Fu, Q., Lou, J. G., Lin, Q., Zhang, D., and Xie, T. Mining historical issue repositories to heal large-scale online service systems. *Proceedings of the Int. Conf. on Dependable Systems and Networks* (9 2014), 311–322.

[37] Durieux, T., Hamadi, Y., and Monperrus, M. Production-driven patch generation. *ICSE* (6 2017), 23–26.

[38] Ford, S., and Olmsted, A. Security vulnerabilities in javascript hotpatching in ios with a commercial and open-source tool. *Int. Conf. on Information Society 2018-January* (5 2018), 108–110.

[39] Fu, Q., Lou, J. G., Lin, Q. W., Ding, R., Zhang, D., Ye, Z., and Xie, T. Performance issue diagnosis for online service systems. *Proceedings of the IEEE Symposium on Reliable Distributed Systems* (2012), 273–278.

[40] Gao, Q., Zhang, W., Tang, Y., and Qin, F. First-aid: Surviving and preventing memory management bugs during production runs. *Proceedings of the 4th ACM European Conf. on Computer Systems, EuroSys'09* (2009), 159–172.

[41] Ghosh, S., Shetty, M., Bansal, C., and Nath, S. How to Fight Production Incidents? An Empirical Study on a Large-scale Cloud Service. *SoCC 2022 - Proceedings of the 13th Symposium on Cloud Computing* (nov 2022), 126–141.

[42] Glerum, K., Kinshumann, K., Greenberg, S., Aul, G., Orgovan, V., Nichols, G., Grant, D., Loihle, G., and Hunt, G. Debugging in the (very) large: Ten years of implementation and experience. *SOSP'09 - Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles* (2009), 103–116.
[43] Gomez, M., Martineza, M., Monperrus, M., and Rouvoy, R. When app stores listen to the crowd to fight bugs in the wild. *ICSE 2* (8 2015), 567–570.
[44] Goues, C. L., Pradel, M., and Roychoudhury, A. Automated program repair. *Communications of the ACM* (2019).
[45] Gray, J. Why Do Computers Stop and What Can Be Done About It? *Symposium on Reliability in Distributed Software and Database Systems* (1986).
[46] Grottke, M., Kim, D. S., Mansharamani, R., Nambiar, M., Natella, R., and Trivedi, K. S. Recovery from software failures caused by mandelbugs. *IEEE Transactions on Reliability 65* (3 2016), 70–87.
[47] Gu, J., Wen, J., Wang, Z., Zhao, P., Luo, C., Kang, Y., Zhou, Y., Yang, L., Sun, J., Xu, Z., Qiao, B., Li, L., Lin, Q., and Zhang, D. Efficient customer incident triage via linking with system incidents. *ESEC/FSE 2020 - Proceedings of the 28th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2020), 1296–1307.
[48] Guardian, T. Google outage: tech giant apologises after software update causes search engine to go down | google | the guardian, 2022.
[49] Gunawi, H. S., Hao, M., Suminto, R. O., Laksono, A., Satria, A. D., Adityatama, J., and Eliazar, K. J. Why does the cloud stop computing? Lessons from hundreds of service outages. *Proceedings of the 7th ACM Symposium on Cloud Computing, SoCC 2016* (oct 2016), 1–16.
[50] Guo, Z., McDirmid, S., Yang, M., Zhuang, L., Zhang, P., Luo, Y., Bergan, T., Musuvathi, M., Zhang, Z., and Zhou, L. Failure Recovery: When the Cure Is Worse Than the Disease, 2013.
[51] Gupta, M., Banerjee, S., Agrawal, M., and Rao, H. R. Security analysis of internet technology components enabling globally distributed workplacesa framework. *ACM Transactions on Internet Technology (TOIT) 8* (10 2008).
[52] Gómez, M., Adams, B., Maalej, W., Monperrus, M., and Rouvoy, R. App store 2.0: From crowdsourced information to actionable feedback in mobile ecosystems. *IEEE Software 34* (3 2017), 81–89.
[53] Han, S., Baby, D., and Mendelev, V. Residual adapters for targeted updates in rnn-transducer based speech recognition system. *2022 IEEE Spoken Language Technology Workshop, SLT 2022 - Proceedings* (2023), 160–166.
[54] Hanna, C., Elliman, D., Emmerich, W., Sarro, F., and Petke, J. Behind the hot fix: Demystifying hot fixing industrial practices at zühlke and beyond. ACM.
[55] Hanna, C., and Petke, J. Hot patching hot fixes: Reflection and perspectives. *ASE* (9 2023).
[56] Hassan, S., Shang, W., and Hassan, A. E. An empirical study of emergency updates for top android mobile apps. *Empirical Software Engineering 22* (2 2017), 505–546.
[57] He, S., Lin, Q., Lou, J. G., Zhang, H., Lyu, M. R., and Zhang, D. Identifying impactful service system problems via log analysis. *ESEC/FSE 18* (10 2018), 60–70.
[58] Herzig, K. Using pre-release test failures to build early post-release defect prediction models. *Proceedings - Int. Symposium on Software Reliability Engineering, ISSRE* (12 2014), 300–311.
[59] Holloway, A., Denison, J., Patel, N., Maimone, M., and Rankin, A. Six Years and 184 Tickets: The Vast Scope of the Mars Science Laboratory's Ultimate Flight Software Release. *IEEE Aerospace Conference Proceedings 2023-March* (2023).
[60] Huang, H., Tsai, W. T., and Chen, Y. Autonomous hot patching for web-based applications. *Proceedings - Int. Computer Software and Applications Conf. 2* (2005), 51–56.
[61] Huang, Z., Dangelo, M., Miyani, D., and Lie, D. Talos: Neutralizing vulnerabilities with security workarounds for rapid response. *Proceedings - IEEE Symposium on Security and Privacy* (8 2016), 618–635.
[62] IEEE. July 2023 ieee thesaurus version 1.02 created by the institute of electrical and electronics engineers (ieee).
[63] Illes-Seifert, T., and Paech, B. Exploring the relationship of a file's history and its fault-proneness: An empirical study. *Testing: Academic and Industrial Conf. Practice and Research Techniques* (2008), 13–22.
[64] Ilvonen, V., Ihantola, P., and Mikkonen, T. Dynamic software updating techniques in practice and educator's guides: a review. In *2016 IEEE 29th International Conference on Software Engineering Education and Training (CSEET)* (2016), IEEE, pp. 86–90.
[65] Insider, B. Amazon prime day issues estimated to cost 72*millionto*99 million - business insider, 2018.
[66] Islam, C., Prokhorenko, V., and Babar, M. A. Runtime software patching: Taxonomy, survey and future directions. *Journal of Systems and Software 200* (6 2023), 111652.
[67] Jenkins, A., Wolters, M., Liu, L., and Vaniea, K. Not as easy as just update: Survey of System Administrators and Patching Behaviours. *Conference on Human Factors in Computing Systems - Proceedings* (may 2024).
[68] Jeong, H., Kang, K., and An, J. Hot-patching Platform for Executable and Linkable Format Binary Application for System Resilience. *Proceedings of the ACM Symposium on Applied Computing* (mar 2023), 1301–1304.
[69] Jia, T., Wu, Y., Hou, C., and Li, Y. LogFlash: Real-time Streaming Anomaly Detection and Diagnosis from System Logs for Large-scale Software Systems. *Proceedings - International Symposium on Software Reliability Engineering, ISSRE 2021-Octob* (2021), 80–90.

[70] Jiang, J., Lu, W., Chen, J., Lin, Q., Zhao, P., Kang, Y., Zhang, H., Xiong, Y., Gao, F., Xu, Z., Dang, Y., and Zhang, D. How to mitigate the incident? an effective troubleshooting guide recommendation technique for online service systems. *ESEC/FSE 2020 - Proceedings of the 28th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (nov 2020), 1410–1420.

[71] Jiang, Y., Zhang, C., He, S., Yang, Z., Ma, M., Qin, S., Kang, Y., Dang, Y., Rajmohan, S., Lin, Q., and Zhang, D. Xpert: Empowering Incident Management with Query Recommendations via Large Language Models. *Proceedings - International Conference on Software Engineering* (dec 2023), 1121–1133.

[72] Jin, G., Song, L., Zhang, W., Lu, S., and Liblit, B. Automated atomicity-violation fixing. *Conf. on Programming Language Design and Implementation* (2011), 389–400.

[73] Jin, P., Zhang, S., Ma, M., Li, H., Kang, Y., Li, L., Liu, Y., Qiao, B., Zhang, C., Zhao, P., He, S., Sarro, F., Dang, Y., Rajmohan, S., Lin, Q., and Zhang, D. Assess and Summarize: Improve Outage Understanding with Large Language Models. *ESEC/FSE 2023 - Proceedings of the 31st ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2023), 1657–1668.

[74] Karale, S. V., and Kaushal, V. An automation framework for configuration management to reduce manual intervention. *ACM Int. Conf. Proceeding Series 12-13-August-2016* (8 2016).

[75] Khomh, F., Chan, B., Zou, Y., and Hassan, A. E. An entropy evaluation approach for triaging field crashes: A case study of mozilla firefox. *Working Conf. on Reverse Engineering* (2011), 261–270.

[76] Khomh, F., Dhaliwal, T., Zou, Y., and Adams, B. Do faster releases improve software quality? an empirical case study of mozilla firefox. *IEEE Int. Working Conf. on Mining Software Repositories* (2012), 179–188.

[77] Kolassa, C., Riehle, D., and Salim, M. A. The empirical commit frequency distribution of open source projects. *Proceedings of the Int. Symposium on Open Collaboration* (2013).

[78] Levy, S., Yao, R., Wu, Y., Dang, Y., Huang, P., Mu, Z., Zhao, P., Ramani, T., Govindaraju, N., Li, X., et al. Predictive and adaptive failure mitigation to avert production cloud {VM} interruptions. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (2020), pp. 1155–1170.

[79] Li, F., Chetty, M., Rogers, L., Mathur, A., and Malkin, N. Keepers of the machines: Examining how system administrators manage software updates. *Fifteenth Symposium on Usable Privacy and Security (SOUPS 2019)* (2019), 273–288.

[80] Li, L., Zhang, X., Zhao, X., Zhang, H., Kang, Y., Zhao, P., Qiao, B., He, S., Lee, P., Sun, J., Gao, F., Yang, L., Lin, Q., Rajmohan, S., Xu, Z., and Zhang, D. *Fighting the Fog of War: Automated Incident Detection for Cloud Systems.* 2021.

[81] Li, Z., and Long, J. A case study of measuring degeneration of software architectures from a defect perspective. *Proceedings - Asia-Pacific Software Engineering Conf., APSEC* (2011), 242–249.

[82] Lin, D., Bezemer, C. P., and Hassan, A. E. Studying the urgent updates of popular games on the steam platform. *Empirical Software Engineering 22* (8 2017), 2095–2126.

[83] Lin, F., Davoli, A., Akbar, I., Kalmanje, S., Silva, L., Stamford, J., Golany, Y., Piazza, J., and Sankar, S. Predicting remediations for hardware failures in large-scale datacenters. *Proceedings - 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks: Supplemental Volume, DSN-S 2020* (jun 2020), 13–16.

[84] Lin, Q., Lou, J. G., Zhang, H., and Zhang, D. How to tame your online services. *Perspectives on Data Science for Software Engineering* (2016), 63–65.

[85] Lin, Q., Lou, J.-G., Zhang, H., and Zhang, D. idice: Problem identification for emerging issues. In *Proceedings of the 38th International Conference on Software Engineering* (2016), pp. 214–224.

[86] Lin, Z., Jiang, X., Xu, D., Mao, B., and Xie, L. Autopag: Towards automated software patch generation with source code root cause identification and repair. *eProceedings of the 2nd ACM Symposium on Information, Computer and Communications Security, ASIACCS '07* (2007), 329–340.

[87] Liu, H., Lu, S., Musuvathi, M., and Nath, S. What bugs cause production cloud incidents? *Proceedings of the Workshop on Hot Topics in Operating Systems, HotOS 2019* (may 2019), 155–162.

[88] Liu, Y., Shu, X., Sun, Y., Jang, J., and Mittal, P. RAPID: Real-Time Alert Investigation with Context-aware Prioritization for Efficient Threat Discovery. *ACM International Conference Proceeding Series* (dec 2022), 827–840.

[89] Lou, C., Chen, C., Huang, P., Dang, Y., Qin, S., Yang, X., Li, X., Lin, Q., and Chintalapati, M. *RESIN: A Holistic Service for Dealing with Memory Leaks in Production Cloud Infrastructure.* 2022.

[90] Lou, J. G., Lin, Q., Ding, R., Fu, Q., Zhang, D., and Xie, T. Software analytics for incident management of online services: An experience report. *ASE* (2013), 475–485.

[91] Lou, J. G., Lin, Q., Ding, R., Fu, Q., Zhang, D., and Xie, T. Experience report on applying software analytics in incident management of online service. *Automated Software Engineering 24*, 4 (2017), 905–941.

[92] Luo, L., Nath, S., Sivalingam, R., Musuvathi, M., and Ceze, L. Troubleshooting transiently-recurring errors in production systems with blame-proportional logging troubleshooting transiently-recurring problems in production systems with blame-proportional logging. *USENIX Annual Technical Conf. (USENIX ATC 18)* (2018), 321–334.

[93] MacHiry, A., Redini, N., Camellini, E., Kruegel, C., and Vigna, G. Spider: Enabling fast patch propagation in related software repositories. *Proceedings - IEEE Symposium on Security and Privacy 2020-May* (5 2020), 1562–1579.

[94] Malone, M., Wang, Y., Snow, K., and Monrose, F. Applicable micropatches and where to find them: Finding and applying new security hot fixes to old software. *Proc. - 2021 IEEE 14th Int. Conf. on Software Testing, Verification and Validation, ICST 2021* (4 2021), 394–405.

[95] Marconato, G. V., Nicomette, V., and Kaâniche, M. Security-related vulnerability life cycle analysis. *Int. Conf. on Risks and Security of Internet and Systems* (2012).

[96] Marra, M., Polito, G., and Boix, E. G. A debugging approach for live big data applications. *Science of Computer Programming 194* (8 2020).

[97] Mendeley. Mendeley, 2025.

[98] Mockus, A., Fielding, R. T., Herbsleb, J., Labs, B., and Blvd, S. A case study of open source software development: The apache server. *ICSE* (2000).

[99] Mugarza, I., Parra, J., and Jacob, E. Analysis of existing dynamic software updating techniques for safe and secure industrial control systems. *International journal of safety and security engineering 8*, 1 (2018), 121–131.

[100] Mulliner, C., Oberheide, J., Robertson, W., and Kirda, E. Patchdroid: Scalable third-party security patches for android devices. *ACM Int. Conf. Proceeding Series* (2013), 259–268.

[101] Nair, V., Raul, A., Khanduja, S., Bahirwani, V., Shao, Q., Sundararajan, S., Keerthi, S., Herbert, S., and Dhulipalla, S. Learning a hierarchical monitoring system for detecting and diagnosing service issues. *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining 2015-Augus* (aug 2015), 2029–2038.

[102] Novark, G., Berger, E. D., and Zorn, B. G. Exterminator: Automatically correcting memory errors with high probability. *Conf. on Programming Language Design and Implementation* (2007), 1–11.

[103] Oosterhuis, H., and Rijke, M. D. D. Robust generalization and safe query-specializationin counterfactual learning to rank. *The Web Conference 2021 - Proceedings of the World Wide Web Conference, WWW 2021* (4 2021), 158–170.

[104] Oppenheimer, D., Ganapathi, A., and Patterson, D. A. Why Do Internet Services Fail, and What Can Be Done About It?, 2003.

[105] Pamunuwa, V., Deraniyagala, D., Kulasekara, V., Thennakoon, R., and Lankasena, B. Investigating the impact of software maintenance activities on software quality: Case study.

[106] Parameshwaran, I., Budianto, E., Shinde, S., Dang, H., Sadhu, A., and Saxena, P. Auto-patching dom-based xss at scale. *ESEC/FSE* (8 2015), 272–283.

[107] Payer, M., and Gross, T. R. Hot-patching a web server: A case study of asap code repair. *Annual Conf. on Privacy, Security and Trust* (2013), 143–150.

[108] Perkins, J. H., Kim, S., Larsen, S., Amarasinghe, S., Bachrach, J., Carbin, M., Pacheco, C., Sherwood, F., Sidiroglou, S., Sullivan, G., Wong, W. F., Zibin, Y., Ernst, M. D., and Rinard, M. Automatically patching errors in deployed software. *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles* (2009), 87–102.

[109] Pham, P., Jain, V., Dauterman, L., Ormont, J., and Jain, N. DeepTriage: Automated Transfer Assistance for Incidents in Cloud Services. *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2020), 3281–3289.

[110] Potharaju, R., Chan, J., Hu, L., Nita-Rotaru, C., Wang, M., Zhang, L., and Jain, N. ConfSeer: leveraging customer support knowledge bases for automated misconfiguration detection. *Proceedings of the VLDB Endowment 8*, 12 12 (2015), 1828–1839.

[111] Pozo, F., and Rodriguez-Navas, G. A semi-distributed self-healing protocol for run-time repairs of time-triggered schedules. *IEEE Int. Conf. on Emerging Technologies and Factory Automation, ETFA 2019-September* (9 2019), 1399–1402.

[112] Qin, F., Tucek, J., Sundaresan, J., and Zhou, Y. Rx: Treating bugs as allergies - a safe method to survive software failures. *Proceedings of the 20th ACM Symposium on Operating Systems Principles, SOSP 2005* (2005), 235–248.

[113] Qin, L., Li, Y., and Yue, C. Dataflow analysis for known vulnerability prevention system. *2008 IEEE Int. Conf. on Cybernetics and Intelligent Systems, CIS 2008* (2008), 1032–1035.

[114] Ramaswamy, A., Bratus, S., Smith, S. W., and Locasto, M. E. Katana: A hot patching framework for elf executables. *ARES 2010 - 5th Int. Conf. on Availability, Reliability, and Security* (2010), 507–512.

[115] Rasche, A., and Polze, A. Redac - dynamic reconfiguration of distributed component-based applications with cyclic dependencies. *Proceedings - IEEE Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing* (2008), 322–330.

[116] Russinovich, M., Govindaraju, N., Raghuraman, M., Hepkin, D., Schwartz, J., and Kishan, A. Virtual machine preserving host updates for zero day patching in public cloud. *Proceedings of the European Conf. on Computer Systems 21* (4 2021), 114–129.

[117] Saha, A., and Hoi, S. C. H. Mining Root Cause Knowledge from Cloud Service Incident Investigations for AIOps. 197–206.

[118] Sahoo, R. K., Oliner, A. J., Rish, I., Gupta, M., Moreira, J. E., Ma, S., Vilalta, R., and Sivasubramaniam, A. Critical event prediction for proactive management in large-scale computer clusters. *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2003), 426–435.

[119] Saieva, A., and Kaiser, G. Binary quilting to generate patched executables without compilation. *ACM Workshop on Forming an Ecosystem Around Software Transformation* (11 2020), 3–8.

[120] Saieva, A., and Kaiser, G. Update with care: Testing candidate bug fixes and integrating selective updates through binary rewriting. *Journal of Systems and Software 191* (9 2022), 111381.

[121] Salehi, M., and Pattabiraman, K. Poster autopatch: Automatic hotpatching of real-time embedded devices. *Proceedings of the ACM Conf. on Computer and Communications Security* (11 2022), 3451–3453.

[122] Sarabi, A., Zhu, Z., Xiao, C., Liu, M., and Dumitraş, T. Patch me if you can: A study on the effects of individual user behavior on the end-host vulnerability state. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 10176 LNCS* (2017), 113–125.

[123] Sarda, K., Namrud, Z., Litoiu, M., Shwartz, L., and Watts, I. Leveraging Large Language Models for the Auto-remediation of Microservice Applications: An Experimental Study. *FSE Companion - Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering* (jul 2024), 358–369.

[124] Savor, T., Douglas, M., Gentili, M., Williams, L., Beck, K., and Stumm, M. Continuous deployment at Facebook and OANDA. *ICSE* (5 2016), 21–30.

[125] Shen, S., Lu, X., Hu, Z., and Liu, X. Towards release strategy optimization for apps in google play. *ACM Int. Conf. Proceeding Series Part F130951* (9 2017).

[126] Shen, Z., and Chen, S. A survey of automatic software vulnerability detection, program repair, and defect prediction techniques. *Security and Communication Networks* (2020).

[127] Shetty, M., Bansal, C., Kumar, S., Rao, N., Nagappan, N., and Zimmermann, T. Neural knowledge extraction from cloud service incidents. *Proceedings - International Conference on Software Engineering* (2021), 218–227.

[128] Shetty, M., Bansal, C., Upadhyayula, S. P., Radhakrishna, A., and Gupta, A. AutoTSG: learning and synthesis for incident troubleshooting. *ESEC/FSE 2022 - Proceedings of the 30th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2022), 1477–1488.

[129] Shihab, E., Mockus, A., Kamei, Y., Adams, B., and Hassan, A. E. High-impact defects: A study of breakage and surprise defects. *SIGSOFT/FSE* (2011), 300–310.

[130] Sidiroglou, S., Ioannidis, S., and Keromytis, A. D. Band-aid patching. *Workshop on Hot Topics in System Dependability* (2007), 102–106.

[131] Sidiroglou, S., Locasto, M. E., Boyd, S. W., and Keromytis, A. D. Building a reactive immune system for software services. *Proceedings of the 2005 USENIX Annual Technical Conf.* (2005), 149–161.

[132] Sun, D., Fekete, A., Gramoli, V., Li, G., Xu, X., and Zhu, L. R2c: Robust rolling-upgrade in clouds. *IEEE Transactions on Dependable and Secure Computing 15* (9 2018), 811–823.

[133] Tang, J., Kim, H., Mascolo, C., and Musolesi, M. Stop: Socio-temporal opportunistic patching of short range mobile malware. *2012 IEEE Int. Symposium on a World of Wireless, Mobile and Multimedia Networks, WoWMoM 2012 - Digital Proceedings* (2012).

[134] Thota, M. K., Shajin, F. H., and Rajesh, P. Survey on software defect prediction techniques. *Int. Journal of Applied Science and Engineering 17* (2020), 331–344.

[135] Trivedi, K. S., Mansharamani, R., Kim, D. S., Grottke, M., and Nambiar, M. Recovery from failures due to mandelbugs in it systems. *Proceedings of IEEE Pacific Rim Int. Symposium on Dependable Computing, PRDC* (2011), 224–233.

[136] Truelove, A., de Almeida, E. S., and Ahmed, I. We'll fix it in post: What do bug fixes in video game update notes tell us? *ICSE* (5 2021), 736–747.

[137] Tucek, J., Lu, S., Huang, C., Xanthos, S., and Zhou, Y. Triage: Diagnosing production run failures at the user's site. *Operating Systems Review (ACM)* (2007), 131–144.

[138] Tucek, J., Lu, S., Huang, C., Xanthos, S., Zhou, Y., Newsome, J., Brumley, D., and Song, D. Sweeper: A lightweight end-to-end system for defending against fast worms. *Operating Systems Review (ACM)* (2007), 115–128.

[139] Tunde-Onadele, O., Carolina, N., Lin, Y., He, J., and Gu, X. Toward just-in-time patching for containerized applications. *Proceedings of the 7th Symposium on Hot Topics in the Science of Security* (2020).

[140] Tunde-Onadele, O., Lin, Y., He, J., and Gu, X. Self-patch: Beyond patch tuesday for containerized applications. *Int. Conf. on Autonomic Computing and Self-Organizing Systems* (8 2020), 21–27.

[141] Van Der Storm, T. Continuous release and upgrade of component-based software. *Proceedings of the 12th Int. Workshop on Software Configuration Management, SCM 2005* (2005), 43–57.

[142] Van Der Storm, T. Binary change set composition. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 4608 LNCS* (2007), 17–32.

[143] Vaniea, K., and Rashidi, Y. Tales of software updates: The process of updating software. *Conf. on Human Factors in Computing Systems - Proceedings* (5 2016), 3215–3226.

[144] Verma, S., and Roy, S. Synergistic debug-repair of heap manipulations. *FSE Part F130154* (8 2017), 163–173.

[145] Vögler, M., Schleicher, J. M., Inzinger, C., and Dustdar, S. A scalable framework for provisioning large-scale iot deployments. *ACM Transactions on Internet Technology 16* (3 2016).

[146] Vögler, M., Schleicher, J. M., Inzinger, C., Nastic, S., Sehic, S., and Dustdar, S. Leonore - large-scale provisioning of resource-constrained iot deployments. *Proceedings - IEEE Int. Symposium on Service-Oriented System Engineering 30* (6 2015), 78–87.
[147] Wang, W., Chen, J., Yang, L., Zhang, H., Zhao, P., Qiao, B., Kang, Y., Lin, Q., Rajmohan, S., Gao, F., Xu, Z., Dang, Y., and Zhang, D. How Long Will it Take to Mitigate this Incident for Online Service Systems? *Proceedings - International Symposium on Software Reliability Engineering, ISSRE 2021-Octob* (2021), 36–46.
[148] Wang, Y., Jiang, S., and Cui, B. Tjosconf: Automatic and safe system environment operations platform. *ACM Int. Conf. Proceeding Series 2022* (2 2022), 21–28.
[149] Wang, Y., Li, G., Wang, Z., Kang, Y., Zhou, Y., Zhang, H., Gao, F., Sun, J., Yang, L., Lee, P., Xu, Z., Zhao, P., Qiao, B., Li, L., Zhang, X., and Lin, Q. Fast outage analysis of large-scale production clouds with service correlation mining. *Proceedings - International Conference on Software Engineering* (may 2021), 885–896.
[150] Weiß, C., Premraj, R., Zimmermann, T., and Zeller, A. How long will it take to fix this bug? *Proceedings - ICSE 2007 Workshops: Fourth International Workshop on Mining Software Repositories, MSR 2007* (2007).
[151] Woodard, D. B., and Goldszmidt, M. Online model-based clustering for crisis identification in distributed computing. *Journal of the American Statistical Association 106* (3 2012), 49–60.
[152] Wu, Y., Chai, B., Li, Y., Liu, B., Li, J., Yang, Y., and Jiang, W. An Empirical Study on Change-induced Incidents of Online Service Systems. *Proceedings - International Conference on Software Engineering* (2023), 234–245.
[153] Xu, Z. Source code and binary level vulnerability detection and hot patching. *ASE* (2 2020), 1397–1399.
[154] Xu, Z., Zhang, Y., Zheng, L., Xia, L., Bao, C., X-Lab, B., Wang, Z., Liu, Y., Longri, B. X.-L., Baidu, Z., Liangzhao, X.-L., Baidu, X., Chenfu, X.-L., Baidu, B., and Wang, X.-L. Z. Automatic hot patch generation for android kernels. *Proceedings of the USENIX Conf. on Security Symposium* (2020).
[155] Yuan, C., Ma, W. Y., Wen, J. R., Li, J., Zhang, Z., and Wang, Y. M. Automated known problem diagnosis with event traces. *ACM SIGOPS Operating Systems Review 40* (4 2006), 375–388.
[156] Yuan, D., Luo, Y., Zhuang, X., Rodrigues, G. R., Zhao, X., Zhang, Y., Jain, P. U., and Stumm, M. *Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems.* 2014.
[157] Zeng, Z., Zhang, Y., Xu, Y., Ma, M., Qiao, B., Zou, W., Chen, Q., Zhang, M., Zhang, X., Zhang, H., Gao, X., Fan, H., Rajmohan, S., Lin, Q., and Zhang, D. TraceArk: Towards Actionable Performance Anomaly Alerting for Online Service Systems. *Proceedings - International Conference on Software Engineering* (2023), 258–269.
[158] Zhang, H., Gong, L., and Versteeg, S. Predicting bug-fixing time: An empirical study of commercial software projects. *ICSE* (2013), 1042–1051.
[159] Zhang, H., and Qian, Z. Precise and accurate patch presence test for binaries. *27th USENIX Security Symposium* (2018).
[160] Zhang, H., Zhao, L., Xu, L., Wang, L., and Wu, D. Vpatcher: Vmi-based transparent data patching to secure software in the cloud. *TrustCom* (1 2015), 943–948.
[161] Zhang, M., and Yin, H. Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications. *NDSS 14* (2014), 23–26.
[162] Zhang, S., Cohen, I., Goldszmidt, M., Symons, J., and Fox, A. Ensembles of models for automated diagnosis of system performance problems. *Proceedings of the Int. Conf. on Dependable Systems and Networks* (2005), 644–653.
[163] Zhang, X., Xu, Y., Qin, S., He, S., Qiao, B., Li, Z., Zhang, H., Li, X., Dang, Y., Lin, Q., Chintalapati, M., Rajmohan, S., and Zhang, D. Onion: Identifying incident-indicating logs for cloud systems. *ESEC/FSE 21* (8 2021), 1253–1263.
[164] Zhang, X., Zhang, Y., Li, J., Hu, Y., Li, H., and Gu, D. Embroidery: Patching vulnerable binary code of fragmentized android devices. *Int. Conf. on Software Maintenance and Evolution* (11 2017), 47–57.
[165] Zhao, N., Chen, J., Wang, Z., Peng, X., Wang, G., Wu, Y., Zhou, F., Feng, Z., Nie, X., Zhang, W., Sui, K., and Pei, D. Real-time incident prediction for online service systems. *ESEC/FSE 2020 - Proceedings of the 28th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2020), 315–326.
[166] Zhao, N., Jin, P., Wang, L., Yang, X., Liu, R., Zhang, W., Sui, K., and Pei, D. Automatically and Adaptively Identifying Severe Alerts for Online Service Systems. *Proceedings - IEEE INFOCOM 2020-July* (jul 2020), 2420–2429.
[167] Zhao, Y., Jiang, L., Tao, Y., Zhang, S., Wu, C., Wu, Y., Jia, T., Li, Y., and Wu, Z. How to Manage Change-Induced Incidents? Lessons from the Study of Incident Life Cycle. *Proceedings - International Symposium on Software Reliability Engineering, ISSRE* (2023), 264–274.
[168] Zheng, L., Berkeley, U., Jia, C., Sun, M., Wu, Z., Group, A., Yu, C. H., Haj-Ali, A., Wang, Y., Yang, J., Zhuo, D., Sen, K., Gonzalez, J. E., and Stoica, I. *Testing Configuration Changes in Context to Prevent Production Failures.* 2020.
[169] Zheng, W., Lu, H., Zhou, Y., Liang, J., Zheng, H., and Deng, Y. IFeedback: Exploiting user feedback for real-time issue detection in large-scale online service systems. *Proceedings - 2019 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019* (2019), 352–363.
[170] Zhou, H., Lou, J.-G., Zhang, H., Lin, H., Lin, H., and Qin, T. An empirical study on quality issues of production big data platform. *ICSE* (2015).

[171] Zhou, L., Zhang, F., Liao, J., Ning, Z., Xiao, J., Leach, K., Weimer, W., and Wang, G. Kshot: Live kernel patching with smm and sgx. *Proceedings - 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2020* (6 2020), 1–13.