ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26nov573

A Review of Cross-Project and Just-in-Time Software Defect Prediction with Lessons from Real-World Failures

Radhalakshmi Rajagopalan[1*]; Dr. Radha V.[1]

[1]Research School of Physical Science and Computational Science, Avinashilingam Institute for Home Science and Higher Education for Women (Deemed to be University), Coimbatore, Tamil Nadu, India

[1*]ORCID: 0009-0006-5366-9631
[1]ORCID: 0000-0002-7590-9839

Corresponding Author: Radhalakshmi Rajagopalan[*]

Publication Date: 2026/11/21

Abstract: Software defects that escape into production can ground airline fleets, halt trading desks and disrupt essential services within minutes. This review connects the research on software defect prediction with the real cost of such failures. It first examines a set of widely reported incidents in aviation, airlines, finance and banking, drawn from official investigation reports, company filings and news coverage, and summarises their causes and documented losses. It then reviews two lines of defect prediction research that matter most in industrial settings. *Cross-project defect prediction* (CPDP) addresses the common situation in which a new or small project has too little history to train its own model, and must borrow knowledge from other projects through instance selection, feature transformation, transfer learning and deep domain adaptation. *Just-in-time defect prediction* (JIT-DP) scores each code change at the moment it is committed, so that risky changes can be reviewed before they reach production; deep learning models for JIT-DP learn directly from commit messages and code diffs. The review compares methods, datasets and evaluation practices in both lines, and then asks a practical question: for each incident, at which stage could a prediction model plausibly have raised a warning, and where were the causes outside its reach? It closes with a research agenda for deployment-aware defect prediction that combines cross-project knowledge, change-level risk scores and release-time safeguards.

Keywords: *Software Defect Prediction, Cross-Project Defect Prediction, Just-in-Time Defect Prediction, Transfer Learning, Deep Learning, Software Failures, Cost of Poor Software Quality, Airline IT Outages.*

How to Cite: Radhalakshmi Rajagopalan; Dr. Radha V. (2026) A Review of Cross-Project and Just-in-Time Software Defect Prediction with Lessons from Real-World Failures. *International Journal of Innovative Science and Research Technology*, 11(11), 1007-1015. https://doi.org/10.38124/ijisrt/26nov573

I. INTRODUCTION

Software failures are no longer confined to the IT department. When a defective update reached millions of Windows computers on 19 July 2024, airlines, banks, hospitals and broadcasters were disrupted on the same morning, and a single airline later reported losses of around US$500 million [N1], [N2]. Industry estimates put the overall cost of poor software quality in the United States at no less than US$2.41 trillion in 2022, with accumulated technical debt of about US$1.52 trillion [N7]. Even allowing for the uncertainty of such estimates, the scale makes defect prevention an economic question as much as a technical one.

Software defect prediction (SDP) aims to find fault-prone code before it is released, so that testing and review effort can be focused where risk is highest [1]. Two branches of SDP research are especially relevant to the failures that make headlines.

The first is cross-project defect prediction (CPDP). Classic models are trained on the history of the same project, but new projects, small teams and fast-changing products often lack enough labelled history. CPDP trains on data from other projects and transfers the knowledge to the target, which raises the problem of differences in data distribution between projects [2], [3].

The second is just-in-time defect prediction (JIT-DP), which scores each change at commit time rather than each file at release time [4]. Because modern companies deploy many times a day, a risk score that arrives while the change is still

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26nov573

fresh in the developer's mind, and before it reaches production, is far more useful than a file-level report weeks later. Deep learning has become the dominant approach for both branches.

This review makes three contributions. First, it summarises a set of well-documented real-world incidents in aviation, airlines, finance and banking, and the losses reported for them in official statements, regulatory decisions and news coverage. Second, it reviews CPDP and JIT-DP research, with emphasis on deep learning and transfer learning. Third, it maps the incidents back to the defect prediction pipeline, to show where prediction could plausibly have helped and where the causes lay in deployment, operations or human procedure, beyond the reach of any code-level model. The review is organised around four review questions:

- RQ1: What do documented real-world incidents reveal about the cost and nature of software defects in production?
- RQ2: How have cross-project defect prediction methods, especially transfer learning and deep learning, evolved, and how well do they work?
- RQ3: How have deep learning methods been applied to just-in-time (change-level) defect prediction, and what are their limitations?
- RQ4: Which gaps separate current research from the prevention of high-impact production failures?

II. REAL-WORLD SOFTWARE FAILURES AND THEIR LOSSES (RQ1)

This section summarises five incidents chosen because their causes and costs are documented in regulatory decisions, company statements or sustained news coverage. Figures are as reported by those sources; they are not independently audited and in some cases are disputed by the parties involved. Table 1 gives an overview.

Table 1 Selected High-Impact Software Incidents, their Reported Causes and Losses.

Incident	Year	Sector	Reported technical cause	Reported impact and losses	Defect category
CrowdStrike Falcon update outage [N1], [N2], [N11]	2024	Security software; airlines and many other sectors	A content configuration update that the sensor's rules engine could not interpret, released through a validator that did not catch it	About 8.5 million Windows devices crashed; Delta reported about US$500 million in losses, more than 7,000 cancelled flights and 1.3 million affected customers	Faulty update and insufficient validation
Southwest Airlines holiday meltdown [N3], [N4]	2022	Airline	Crew scheduling software could not re-plan crews at scale after a winter storm disrupted the network	More than 16,000 flights cancelled and over 2 million passengers stranded; more than US$600 million in refunds and reimbursements plus a US$140 million civil penalty	Legacy system capacity and design limits
FAA NOTAM system outage [N5], [N6]	2023	Aviation regulator	Contract staff unintentionally deleted files while correcting synchronisation between the primary and backup databases	First nationwide ground stop of US departures since 2001; more than 11,000 flights disrupted	Operational and maintenance error affecting data integrity
Knight Capital trading loss [N8]	2012	Finance	New code was not copied to one of eight servers; a repurposed flag re-activated obsolete "Power Peg" code whose safety counter had been moved years earlier	Millions of unintended orders in 45 minutes; loss of more than US$460 million on the resulting positions, and an SEC penalty of US$12 million	Dead code, repurposed flag and manual deployment
TSB Bank IT migration [N9], [N10]	2018	Banking	Migration of customer data to a new platform that regulators found was not adequately planned, tested or controlled	Up to 5.2 million customers affected; total cost reported at £330 million and regulatory fines of £48.65 million	Large-scale migration and testing failure

- *Aviation and Airlines*

Aviation shows how tightly coupled systems turn software problems into physical disruption. In the CrowdStrike case, the update was a configuration file rather than a code change, yet its effect was as severe as a code defect: affected machines entered reboot loops and many had to be fixed by hand [N1]. CrowdStrike told a US House subcommittee that the new detection configurations had been validated and sent out but were not understood by the sensor's rules engine [N11]. Delta's later lawsuit argued that testing the update on even one machine would have exposed the crash [N2]; CrowdStrike rejected Delta's claims and blamed the airline's slow recovery on its own IT [N1]. For this review the dispute matters less than the lesson: a defect that passes validation can propagate to millions of devices within minutes.

The Southwest meltdown had a different character. Winter weather triggered the disruption, but news and regulatory reports point to outdated crew scheduling software

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26nov573

as a major reason why the airline could not recover [N3]. This was less a single bug than a system that could not handle an extreme but foreseeable load. The FAA NOTAM outage, by contrast, began with an operational action: contract staff deleted files while repairing a database synchronisation problem, and the damaged data took a safety-critical system offline [N5].

➢ *Finance and Banking*

Knight Capital is the classic example of a deployment defect. According to the SEC's order, a technician failed to copy new code to one of eight servers, and a flag that had once activated an obsolete function was reused for the new feature [N8]. On the server with the old code, that flag woke the obsolete function, which no longer tracked filled orders and therefore never stopped trading. TSB's case was broader: regulators found failures in how the migration programme was organised, tested and overseen, including its reliance on a third-party supplier [N9].

➢ *What the Incidents Have in Common*

Three patterns recur. First, the failing artefact was often not ordinary application code but configuration, deployment state, data or a legacy component. Second, the defect reached production through a change process (an update, a release, a migration, a maintenance task) whose validation did not catch it. Third, the losses came mainly from scale and speed: millions of devices, thousands of flights or millions of orders were affected before humans could react. These patterns are revisited in Section 8.

III. REVIEW METHOD

This is a structured narrative review that combines two kinds of sources.

Academic literature. Studies on cross-project and just-in-time defect prediction were collected from IEEE Xplore, the ACM Digital Library, ScienceDirect, SpringerLink and Google Scholar, and by backward snowballing from existing surveys and benchmarks [2], [3], [5]. Search strings combined "cross-project", "cross-company", "heterogeneous", "transfer learning" or "just-in-time", "change-level", "commit" with "defect prediction", "fault prediction" and "deep learning". Studies were included if they proposed or evaluated a CPDP or JIT-DP method, provided a widely used dataset, or offered an empirical comparison of methods. The selection criteria follow the general guidance of Kitchenham and Charters [6], but the review does not claim to be exhaustive.

Incident sources. Incidents were selected if (a) software, configuration, data or deployment played a central role, (b) the cause was described by a regulator, an official investigation or the company itself, and (c) the financial or operational impact was publicly reported. Facts were cross-checked across at least two independent news organisations or an official source. Incident sources are listed separately as [N1]–[N12].

➢ *Background: Three Settings for Defect Prediction*

Within-project defect prediction (WPDP) trains on earlier releases of a project and predicts defects in a later release of the same project. It works well when a project has a long, well-labelled history, but this is exactly what new projects lack.

Cross-project defect prediction (CPDP) trains on one or more *source* projects and predicts defects in a different *target* project. The central difficulty is that metric distributions differ between projects: a class with 500 lines may be typical in one system and an outlier in another. When source and target even use different metric sets, the problem becomes heterogeneous defect prediction (HDP) [7].

Just-in-time defect prediction (JIT-DP) works at the level of individual changes. Early work by Mockus and Weiss [8] and Kim et al. [9] showed that change properties predict whether a change will introduce a fault, and Kamei et al. [4] established the now-standard set of change metrics covering size, diffusion, history and developer experience. JIT-DP labels are mined with the SZZ algorithm [10], which links each bug fix back to the change that introduced the bug.

The settings combine naturally: a JIT model for a new project must usually be trained cross-project, a setting studied by Kamei et al. [11]. Figure 1 shows where each setting sits in the path from a commit to production, and where the incidents of Section 2 entered that path.

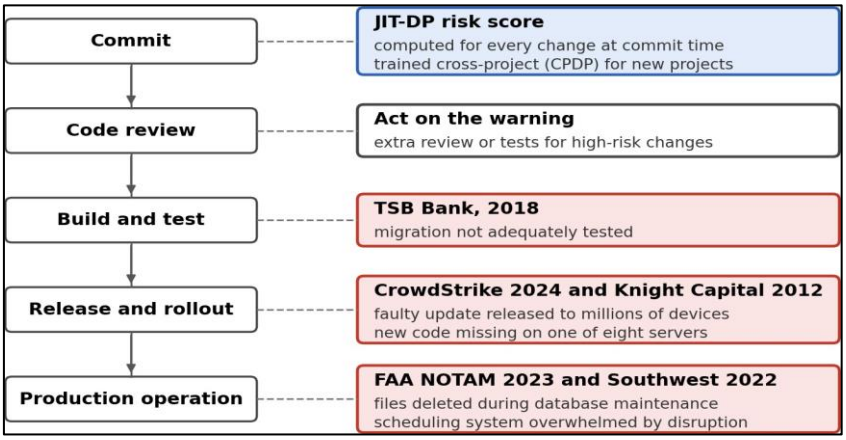


Fig 1 The Path from Commit to Production, with Where Prediction Acts and where the Incidents Entered.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26nov573

Only the earliest stages are covered by today's prediction models; four of the five incidents entered the pipeline at release or in production.

➢ *Cross-Project Defect Prediction (RQ2)*

- *The Difficulty of Borrowing Data*

Early results were discouraging. Zimmermann et al. [12] ran 622 cross-project predictions across open-source and Microsoft projects and found that only a small fraction performed well, and that a model trained on project A predicting B did not imply the reverse. Turhan et al. [13] showed that raw cross-company data produced high recall but very high false-alarm rates, which a nearest-neighbour filter that selected source instances similar to the target could reduce substantially. These two studies set the agenda for the next decade: CPDP works only if the difference between source and target data is handled explicitly.

- *Classical Transfer Approaches*

Three families of methods emerged.

✓ Instance selection and weighting keeps or up-weights source instances that resemble the target. Turhan's nearest-neighbour filter [13] is the best-known example; Transfer Naive Bayes by Ma et al. [14] weights source instances by their similarity to target data.
✓ Feature transformation maps source and target into a shared space where their distributions are closer. TCA+ by Nam et al. [15] applies transfer component analysis with automatically selected normalisation, building on the general transfer learning framework surveyed by Pan and Yang [16].
✓ Universal and heterogeneous models. Zhang et al. [17] built a universal model by clustering projects and normalising metrics within each cluster. Heterogeneous defect prediction [7] goes further and matches individual metrics between projects that use different metric sets.

Large benchmarks temper the optimism. Herbold et al. [3] re-implemented 24 CPDP approaches and evaluated them under the same conditions, finding that many published methods did not outperform simple baselines. A meta-analysis by Hosseini et al. [2] concluded that CPDP generally still trails within-project prediction, although the gap depends strongly on the approach and on how performance is measured.

- *Deep Learning and Deep Transfer*

Deep learning changes the CPDP problem in two ways. First, models that learn from source code rather than metrics may capture project-independent patterns. Wang et al. [18] showed that semantic features learned by a Deep Belief Network from AST token sequences improve cross-project as well as within-project prediction, and Dam et al. [21] evaluated their tree-LSTM model in both settings. Chen et al. [19] went further, rendering code as images and fine-tuning networks pre-trained on natural images, in effect transferring knowledge from a completely different domain.

Second, deep networks allow *domain adaptation* to be built into training. Domain-adversarial training [20] adds a component that tries to tell source from target examples, and trains the feature extractor so that it cannot; the resulting features are informative about defects but uninformative about which project they came from. This idea, together with fine-tuning of code models pre-trained on very large corpora, is the main direction of recent CPDP work. A caution applies: pre-trained models may have seen the target projects during pre-training, so cross-project evaluations should check for such overlap.

Table 2 Main Families of Cross-Project Defect Prediction Methods.

Family	Representative work	Core idea	Main limitation
Raw transfer	Zimmermann et al. [12]	Train on another project as-is	Rarely works; asymmetric
Instance selection	Turhan et al. [13]	Keep source instances near the target	Needs target metrics; discards data
Instance weighting	Ma et al. [14]	Weight source instances by similarity	Sensitive to metric scaling
Feature transformation	Nam et al. (TCA+) [15]	Map projects to a shared space	Assumes the same metric set
Universal model	Zhang et al. [17]	Cluster and normalise across many projects	Coarse; may blur project specifics
Heterogeneous matching	Nam and Kim [7]	Match different metrics across projects	Matching can be unstable
Deep semantic features	Wang et al. [18], Dam et al. [21]	Learn features from code structure	Needs source code; costly to train
Deep transfer and adaptation	Chen et al. [19]; adversarial methods [20]	Pre-train, fine-tune or align domains	Hard to tune; risk of data overlap

IV. JUST-IN-TIME (REAL-TIME) DEFECT PREDICTION (RQ3)

JIT-DP is the closest that defect prediction comes to real time. A model scores each commit as it arrives, and high-risk changes can be routed to extra review or testing before they are merged or deployed.

Because the unit is small and recent, developers can act on the warning while the change is fresh in their minds [4].

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26nov573

➢ *From Change Metrics to Learned Features*

Kamei et al. [4] showed that logistic regression on 14 change metrics, grouped into diffusion, size, purpose, history and experience, could identify a large share of defect-inducing changes while inspecting a fraction of the changed lines. Yang et al. [22] were among the first to apply deep learning, using a Deep Belief Network to build more expressive combinations of the same metrics.

Later models learned from the content of the change. DeepJIT [23] encodes the commit message and the added and removed code with convolutional networks, learning features end-to-end without manual design. CC2Vec [24] learns vector representations of code changes that respect their hierarchical structure (files, hunks, lines) and showed that these representations improve DeepJIT.

➢ *Do Deep JIT Models Beat Simple Ones?*

Zeng et al. [25] re-examined DeepJIT and CC2Vec on larger and more realistic data and found that a simple logistic regression using only the number of added lines matched or exceeded both deep models in many settings. JITLine [26] responded with a lightweight model that combines a bag of code tokens with change metrics, trains far faster than the deep models, and also ranks the lines within a change by risk. JIT-Fine [27] then combined deep semantic features with expert change metrics in one model and reported better results at both change and line level than either kind alone. The lesson is consistent with Section 5: deep models add value mainly when they complement, rather than replace, strong expert features.

➢ *Real-Time Deployment Challenges*

Running JIT-DP in a live pipeline raises problems that offline studies often ignore.

- Concept drift. The properties that predict risky changes shift as a project, its team and its practices evolve. McIntosh and Kamei [28] found that fix-inducing change patterns are a moving target, so JIT models need regular retraining.
- Verification latency. A change can only be labelled defect-inducing after its bug is found and fixed, which may take months. Cabral et al. [29] showed that this delay, together with class imbalance that also changes over time, strongly affects online JIT models.
- Cold start. New projects have no labelled history; cross-project JIT models [11] are the usual answer, which links Sections 5 and 6.
- Beyond source code. Most JIT models look only at source code diffs. Configuration files, data and deployment scripts, which played a central role in several incidents of Section 2, are rarely represented.

Table 3 Representative Just-in-Time Defect Prediction Approaches.

Approach	Year	Input	Model	Main finding
Kamei et al. [4]	2013	14 change metrics	Logistic regression	Effort-aware JIT is practical at scale
Yang et al. [22]	2015	Change metrics	Deep Belief Network	Learned metric combinations help
DeepJIT [23]	2019	Commit message and diff	CNN	End-to-end learning from changes
CC2Vec [24]	2020	Hierarchical diff	Hierarchical attention	Reusable change embeddings
Zeng et al. [25]	2021	Added lines count	Logistic regression	Simple baselines rival deep models
JITLine [26]	2021	Code tokens and change metrics	Random forest with explanation	Fast, explainable, line-level ranking
JIT-Fine [27]	2022	Semantic and expert features	Pre-trained code model plus metrics	Combining both improves change and line level

➢ *Datasets and Evaluation Practices*

Table 4 Datasets Commonly Used for Cross-Project and Just-in-Time Defect Prediction.

Dataset	Setting	Unit	Content	Notes
PROMISE (Jureczko & Madeyski) [30]	CPDP, WPDP	Class	Object-oriented metrics for many releases of Java projects	The most common CPDP benchmark
NASA MDP [31], [32]	CPDP, WPDP	Module	Static code metrics for NASA systems	Known duplicate and inconsistent records [32]
AEEEM / Bug Prediction Dataset [33]	CPDP, HDP	Class	Product, process and change metrics for five Java systems	Different metric set from PROMISE; used in HDP
Kamei et al. JIT data [4]	JIT-DP	Commit	Change metrics for open-source and commercial projects	Defined the standard JIT metric set
Qt and OpenStack [23], [28]	JIT-DP	Commit	Commits with code review data	Used by DeepJIT, CC2Vec and JITLine
ApacheJIT [38]	JIT-DP	Commit	Large set of commits from Apache projects labelled with SZZ	Supports deep JIT models at scale
Line-level defect data [34]	JIT and file level	Line	Defective lines for 32 releases of 9 Java projects	Used for line-level ranking

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26nov573

- *Metrics*

Accuracy is misleading because defects are rare. Studies increasingly report MCC, AUC and F1 for the defective class, and *effort-aware* measures such as recall when inspecting the riskiest 20% of changed lines [4]. For deployment decisions, the cost of a false alarm (an unnecessary review) must be weighed against the cost of a miss; Section 2 shows that the cost of a miss can be several orders of magnitude larger.

- *Validation*

Random cross-validation lets models learn from the future. For JIT-DP, time-ordered splits are essential, and verification latency must be simulated so that the model is trained only on labels that would have been known at the time [29]. Tantithamthavorn et al. [35] showed that the choice of validation technique alone can change conclusions about which model is best.

- *Label Noise*

SZZ can wrongly blame refactoring or cosmetic changes for bugs [37], and its output depends on how bug-fixing commits are identified. Noise of this kind affects JIT-DP directly, because every training label is an SZZ decision.

- *Comparability*

Shepperd et al. [36] found that the research group behind a study explains more of the variation in reported results than the classifier used. Benchmarks such as Herbold et al. [3], which re-run many approaches under one protocol, are therefore more informative than isolated comparisons.

➢ *Lessons from the Incidents: Where Prediction Could and Could Not Help (RQ4)*

It would be wrong to claim that a defect prediction model would have prevented any of the incidents in Section 2. Their causes were complex, and several lay outside source code altogether. It is useful, however, to ask at which stage of the change pipeline each problem entered, and whether a risk signal at that stage was plausible. Table 5 gives this assessment; it is the authors' interpretation of the public record, not a finding of the cited investigations.

Table 5 Mapping Incidents to the Change Pipeline.

Incident	Where the problem entered	Could CPDP or JIT-DP plausibly flag it?	Safeguards that address it more directly
CrowdStrike update [N1], [N11]	A content configuration update passing through an automated validator	Only if JIT models covered configuration and content changes, which current models do not	Testing on real hosts before release; staged (canary) rollout; validator testing; ability to roll back remotely
Southwest meltdown [N3]	Long-standing capacity limits of a legacy scheduling system	No; the problem was not a recent change	Load and stress testing under extreme scenarios; modernisation of legacy systems
FAA NOTAM outage [N5]	A manual maintenance action on a production database	No; no code change was involved	Two-person rules for maintenance (adopted afterwards [N12]); automated backup integrity checks
Knight Capital [N8]	A deployment that left old code on one server, plus a reused flag and dead code	Partly: the 2005 change that moved the safety counter, and the reuse of an old flag, are the kind of risky change JIT-DP and code review aim to catch	Automated deployment verification; removal of dead code; independent risk limits
TSB migration [N9]	A large migration programme with inadequate testing and oversight	Partly, for the code changed within the programme; not for the programme-level decisions	Phased migration; rehearsal on realistic data; supplier oversight

Three lessons follow for defect prediction research.

- Widen the notion of a "change". In several incidents the risky change was a configuration file, a feature flag, a data operation or a deployment step. JIT models that analyse only source code diffs cannot see these. Extending JIT-DP to configuration, infrastructure-as-code and deployment scripts is a natural next step.
- Weigh misses by impact, not count. Standard metrics treat every missed defect alike. The incidents show that a few defects in widely deployed or safety-critical components account for most of the cost. Impact-weighted evaluation, for example weighting by deployment reach or by component criticality, would better reflect what companies lose.
- Predict at the boundary to production. The window between merge and deployment is where a risk score is most valuable, because it can still stop or slow a release. Combining JIT risk scores with staged rollouts, so that riskier changes reach fewer users first, links prediction directly to damage limitation.

➢ *Open Challenges and Research Agenda*

- Cold-start JIT models through transfer. New services need JIT risk scores from their first commit. Combining cross-project training [11] with deep domain adaptation [20] and pre-trained code models offers a path, but it has rarely been evaluated in a realistic, time-ordered online setting.
- JIT prediction for configuration and deployment artefacts.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26nov573

Datasets and models are needed for changes to configuration files, feature flags, infrastructure-as-code and data migrations, the artefacts involved in several of the incidents in Section 2.

- Impact-aware evaluation. Defect prediction should be evaluated against the cost of what it misses. Weighting defects by component criticality, deployment reach or downstream dependencies would align research metrics with business losses.
- Online learning under drift and latency. JIT models in production must update continuously while labels arrive late [28], [29]. Methods that estimate and correct for missing labels, and that detect drift early, are still immature.
- Linking prediction to release control. A risk score is most useful when it changes what happens next: extra review, more tests, or a slower, staged rollout. Studies that measure how such policies change failure rates in practice would close the gap between prediction accuracy and avoided losses.
- Explanations developers trust. Line-level ranking [26], [27] and model-agnostic explanations [39] help developers act on a warning. Their usefulness should be tested with practitioners, not only on benchmarks.
- Incident datasets for research. Public post-incident reviews, regulatory decisions and court filings contain rich information about how failures arise. Curating them into a structured dataset linked to code and configuration changes, where available, would let researchers test prediction methods against failures that actually cost money.

V. CONCLUSION

The incidents reviewed here, from a faulty security update that grounded airline fleets to a trading system that lost hundreds of millions of dollars in under an hour, show that the cost of software defects is concentrated in a few failures that spread quickly through large systems. Defect prediction research has developed strong tools for two settings that matter in industry: cross-project prediction for projects without their own history, and just-in-time prediction that scores changes as they are made. Deep learning has improved both, especially when learned features are combined with expert metrics, but simple baselines remain competitive and evaluations often overlook drift, label delay and the unequal cost of misses.

The main gap is one of scope. The failures that hurt companies most often began in configuration, deployment or data rather than in the source code that current models analyse. Extending cross-project and just-in-time prediction to these artefacts, evaluating models by the impact of what they miss, and linking risk scores to release decisions are the steps most likely to turn defect prediction research into avoided losses.

ACKNOWLEDGEMENTS

[Add funding sources and acknowledgements.]

➢ *Conflict of Interest*

The authors declare no conflict of interest.

REFERENCES

➢ *Academic Sources*

[1.] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A systematic literature review on fault prediction performance in software engineering," *IEEE Trans. Softw. Eng.*, vol. 38, no. 6, pp. 1276–1304, 2012.

[2.] S. Hosseini, B. Turhan, and D. Gunarathna, "A systematic literature review and meta-analysis on cross project defect prediction," *IEEE Trans. Softw. Eng.*, vol. 45, no. 2, pp. 111–147, 2019.

[3.] S. Herbold, A. Trautsch, and J. Grabowski, "A comparative study to benchmark cross-project defect prediction approaches," *IEEE Trans. Softw. Eng.*, vol. 44, no. 9, pp. 811–833, 2018.

[4.] Y. Kamei et al., "A large-scale empirical study of just-in-time quality assurance," *IEEE Trans. Softw. Eng.*, vol. 39, no. 6, pp. 757–773, 2013.

[5.] G. Giray, K. E. Bennin, Ö. Köksal, Ö. Babur, and B. Tekinerdogan, "On the use of deep learning in software defect prediction," *J. Syst. Softw.*, vol. 195, 2023.

[6.] B. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in software engineering," Tech. Rep. EBSE-2007-01, Keele University and Durham University, 2007.

[7.] J. Nam and S. Kim, "Heterogeneous defect prediction," in *Proc. ESEC/FSE*, 2015, pp. 508–519.

[8.] A. Mockus and D. M. Weiss, "Predicting risk of software changes," *Bell Labs Tech. J.*, vol. 5, no. 2, pp. 169–180, 2000.

[9.] S. Kim, E. J. Whitehead, and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Trans. Softw. Eng.*, vol. 34, no. 2, pp. 181–196, 2008.

[10.] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proc. MSR*, 2005.

[11.] Y. Kamei, T. Fukushima, S. McIntosh, K. Yamashita, N. Ubayashi, and A. E. Hassan, "Studying just-in-time defect prediction using cross-project models," *Empir. Softw. Eng.*, vol. 21, no. 5, pp. 2072–2106, 2016.

[12.] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: A large scale experiment on data vs. domain vs. process," in *Proc. ESEC/FSE*, 2009.

[13.] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano, "On the relative value of cross-company and within-company data for defect prediction," *Empir. Softw. Eng.*, vol. 14, no. 5, pp. 540–578, 2009.

[14.] Y. Ma, G. Luo, X. Zeng, and A. Chen, "Transfer learning for cross-company software defect prediction," *Inf. Softw. Technol.*, vol. 54, no. 3, pp. 248–256, 2012.

[15.] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *Proc. ICSE*, 2013.

[16.] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Trans. Knowl. Data Eng.*, vol. 22, no.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26nov573

10, pp. 1345–1359, 2010.
[17.] F. Zhang, A. Mockus, I. Keivanloo, and Y. Zou, "Towards building a universal defect prediction model," in *Proc. MSR*, 2014.
[18.] S. Wang, T. Liu, J. Nam, and L. Tan, "Deep semantic feature learning for software defect prediction," *IEEE Trans. Softw. Eng.*, vol. 46, no. 12, pp. 1267–1293, 2020.
[19.] J. Chen et al., "Software visualization and deep transfer learning for effective software defect prediction," in *Proc. ICSE*, 2020, pp. 578–589.
[20.] Y. Ganin et al., "Domain-adversarial training of neural networks," *J. Mach. Learn. Res.*, vol. 17, no. 59, pp. 1–35, 2016.
[21.] H. K. Dam et al., "Lessons learned from using a deep tree-based model for software defect prediction in practice," in *Proc. MSR*, 2019, pp. 46–57.
[22.] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep learning for just-in-time defect prediction," in *Proc. IEEE QRS*, 2015.
[23.] T. Hoang, H. K. Dam, Y. Kamei, D. Lo, and N. Ubayashi, "DeepJIT: An end-to-end deep learning framework for just-in-time defect prediction," in *Proc. MSR*, 2019.
[24.] T. Hoang, H. J. Kang, D. Lo, and J. Lawall, "CC2Vec: Distributed representations of code changes," in *Proc. ICSE*, 2020.
[25.] Z. Zeng, Y. Zhang, H. Zhang, and L. Zhang, "Deep just-in-time defect prediction: How far are we?" in *Proc. ISSTA*, 2021, pp. 427–438.
[26.] C. Pornprasit and C. Tantithamthavorn, "JITLine: A simpler, better, faster, finer-grained just-in-time defect prediction," in *Proc. MSR*, 2021.
[27.] C. Ni, W. Wang, K. Yang, X. Xia, K. Liu, and D. Lo, "The best of both worlds: Integrating semantic features with expert features for defect prediction and localization," in *Proc. ESEC/FSE*, 2022.
[28.] S. McIntosh and Y. Kamei, "Are fix-inducing changes a moving target? A longitudinal case study of just-in-time defect prediction," *IEEE Trans. Softw. Eng.*, vol. 44, no. 5, pp. 412–428, 2018.
[29.] G. G. Cabral, L. L. Minku, E. Shihab, and S. Mujahid, "Class imbalance evolution and verification latency in just-in-time software defect prediction," in *Proc. ICSE*, 2019.
[30.] M. Jureczko and L. Madeyski, "Towards identifying software project clusters with regard to defect prediction," in *Proc. PROMISE*, 2010.
[31.] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Trans. Softw. Eng.*, vol. 33, no. 1, pp. 2–13, 2007.
[32.] M. Shepperd, Q. Song, Z. Sun, and C. Mair, "Data quality: Some comments on the NASA software defect datasets," *IEEE Trans. Softw. Eng.*, vol. 39, no. 9, pp. 1208–1215, 2013.
[33.] M. D'Ambros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: A benchmark and an extensive comparison," *Empir. Softw. Eng.*, vol. 17, pp. 531–577, 2012.
[34.] S. Wattanakriengkrai, P. Thongtanunam, C. Tantithamthavorn, H. Hata, and K. Matsumoto, "Predicting defective lines using a model-agnostic technique," *IEEE Trans. Softw. Eng.*, vol. 48, no. 5, pp. 1480–1496, 2022.
[35.] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "An empirical comparison of model validation techniques for defect prediction models," *IEEE Trans. Softw. Eng.*, vol. 43, no. 1, pp. 1–18, 2017.
[36.] M. Shepperd, D. Bowes, and T. Hall, "Researcher bias: The use of machine learning in software defect prediction," *IEEE Trans. Softw. Eng.*, vol. 40, no. 6, pp. 603–616, 2014.
[37.] E. C. Neto, D. A. da Costa, and U. Kulesza, "The impact of refactoring changes on the SZZ algorithm: An empirical study," in *Proc. SANER*, 2018.
[38.] H. Keshavarz and M. Nagappan, "ApacheJIT: A large dataset for just-in-time defect prediction," in *Proc. MSR*, 2022.
[39.] J. Jiarpakdee, C. Tantithamthavorn, H. K. Dam, and J. Grundy, "An empirical study of model-agnostic techniques for defect prediction models," *IEEE Trans. Softw. Eng.*, vol. 48, no. 1, pp. 166–185, 2022.

➢ *Incident, Regulatory and News Sources*

[40.] [N1] A. Culafi, "Delta sues CrowdStrike over IT outage fallout," *TechTarget*, Oct. 2024. https://www.techtarget.com/cybersecurity/news/366614745/Delta-sues-CrowdStrike-over-IT-outage-fallout
[41.] [N2] Reuters, "Delta sues CrowdStrike over software update that prompted flight disruptions," *Business Insurance*, Oct. 2024. https://www.businessinsurance.com/delta-sues-crowdstrike-over-software-update-that-prompted-mass-flight-disruptions/
[42.] [N3] NPR, "Southwest will pay a $140 million fine for its meltdown during the 2022 holidays," Dec. 2023. https://www.npr.org/transcripts/1219906471
[43.] [N4] Reuters, "Southwest fined $140 million for last year's holiday meltdown," *NBC DFW*, Dec. 2023. https://www.nbcdfw.com/news/business/money-report/southwest-fined-140-million-for-last-years-holiday-meltdown/3414112/
[44.] [N5] NBC News, "FAA says problem that grounded US flights last week was unintentional," Jan. 2023. https://www.nbcnews.com/news/us-news/faa-says-problem-grounded-us-flights-last-week-was-unintentional-rcna66649
[45.] [N6] Fox 2 Detroit, "FAA reveals what caused computer outage prompting ground stop," Jan. 2023. https://www.fox2detroit.com/news/faa-what-caused-january-computer-outage-prompting-ground-stop
[46.] [N7] CISQ, "CISQ publishes the Cost of Poor Software Quality in the US: A 2022 Report," press release, Dec. 2022. https://www.it-cisq.org/press-releases/12-06-22
[47.] [N8] U.S. Securities and Exchange Commission, "In the Matter of Knight Capital Americas LLC," Release No. 34-70694, Oct. 2013. https://www.sec.gov/litigation/admin/2013/34-

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26nov573

70694.pdf
[48.] [N9] Financial Conduct Authority, "TSB fined £48.65m for operational resilience failings," press release, Dec. 2022. https://www.fca.org.uk/news/press-releases/tsb-fined-ps4865m-operational-resilience-failings
[49.] [N10] ITPro, "TSB bank fined £48 million over botched IT upgrade programme," Dec. 2022. https://itpro.com/business-strategy/it-infrastructure/369753/tsb-bank-fined-48-million-over-botched-it-upgrade
[50.] [N11] Reuters, "CrowdStrike executive apologizes before US Congress for software glitch behind July Microsoft outage," *Deccan Herald*, Sep. 2024.
[51.] [N12] AIN Online, "FAA bolsters safeguards for NOTAM maintenance, database," Jan. 2023. https://www.ainonline.com/aviation-news/air-transport/2023-01-31/faa-bolsters-safeguards-notam-maintenance-database.