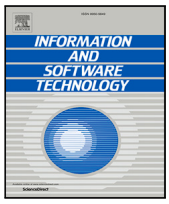
Information and Software Technology 195 (2026) 108133

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof



Architecting software monitors for control-flow anomaly detection through large language models and conformance checking☆

Francesco Vitale [a,*], Francesco Flammini [b,c], Mauro Caporuscio [d], Nicola Mazzocca [a]

[a] University of Naples Federico II, Via Claudio, 21, Naples, 80125, Italy
[b] University of Applied Sciences and Arts of Southern Switzerland, Via la Santa 1, Lugano, 6962, Switzerland
[c] University of Florence, Viale Morgagni, 67, Florence, 50134, Italy
[d] Linnaeus University, Universitetsplatsen, 1, Växjö, 35252, Sweden

ARTICLE INFO

Keywords:
Conformance checking
Software monitoring
Fuzzy runtime verification
Cyber–physical systems
Resilience
Railways

ABSTRACT

Context: Ensuring high levels of dependability in modern computer-based systems has become increasingly challenging due to their complexity. Although systems are validated at design time, their behavior can be different at runtime, possibly showing control-flow anomalies due to "unknown unknowns".
Objective: We aim to detect control-flow anomalies through software monitoring, which verifies runtime behavior by logging software execution and detecting deviations from expected control flow.
Methods: We propose a methodology to develop software monitors for control-flow anomaly detection through Large Language Models (LLMs) and conformance checking. The methodology builds on existing software development practices to maintain traditional V&V while providing an additional level of robustness and trustworthiness. It leverages LLMs to link design-time models and implementation code, automating source-code instrumentation. The resulting event logs are analyzed via conformance checking, an explainable and effective technique for control-flow anomaly detection.
Results: We test the methodology on a case-study scenario from the European Railway Traffic Management System/European Train Control System (ERTMS/ETCS), which is a railway standard for modern interoperable railways. The results obtained from the ERTMS/ETCS case study demonstrate that LLM-based source-code instrumentation can achieve up to 82.849% control-flow coverage of the reference design-time process model, while the subsequent conformance checking-based anomaly detection reaches a peak performance of 95.957% F1-score and 93.669% AUC.
Conclusion: Incorporating domain-specific knowledge to guide LLMs in source-code instrumentation significantly allowed obtaining reliable and quality software logs and enabled effective control-flow anomaly detection through conformance checking.

1. Introduction

The complexity of modern computer-based systems increases their exposure to several types of threats, including software defects, hardware faults, and malicious attacks [1,2]. The usage of model-based verification and formal methods can significantly help in fault avoidance and detection [3–5]; however, practical limitations in test coverage and scalability of formal methods, as well as changes and uncertainties in the system and its environments, make verification and validation extremely challenging. These limitations pose risks to computer-based systems' dependability [6,7]. We aim to demonstrate that a higher level of dependability can be achieved by software monitoring, which involves logging the software execution and detecting any deviations from the expected behavior [8,9]. Specifically, we focus on *control-flow anomaly detection*, which deals with the identification of misalignments between the expected flow of activities and their actual sequencing [10].

Software monitoring has been implemented through runtime verification [11–14]. Runtime verification is closely aligned with formal software specifications and can accurately identify errors that violate them, although its effectiveness depends on the completeness and correctness of the underlying models. In fact, runtime verification requires the synthesis of software monitors by specifying properties and assumptions about the execution, involving a thorough understanding of the software functionality, strong observability guarantees, and

☆ This article is part of a Special issue entitled: 'Software Architecture for AI-Driven Systems' published in Information and Software Technology.

* Corresponding author.
E-mail address: francesco.vitale@unina.it (F. Vitale).

https://doi.org/10.1016/j.infsof.2026.108133
Received 14 November 2025; Received in revised form 14 February 2026; Accepted 19 March 2026
Available online 24 March 2026

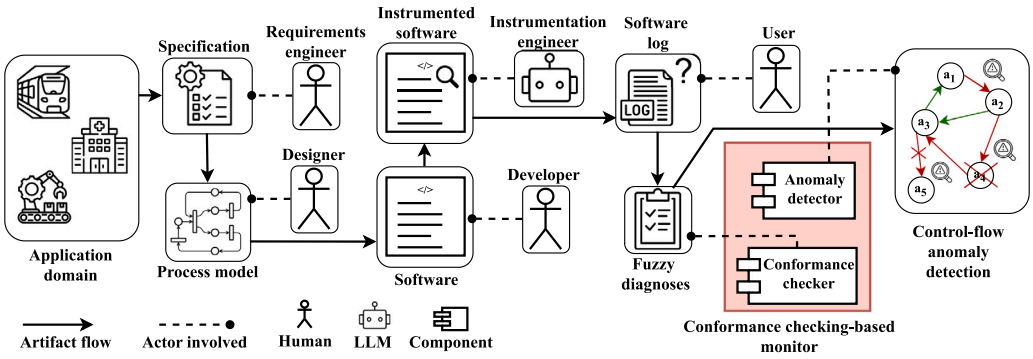


Fig. 1. High-level view of the methodology and the conformance checking-based monitor. Humans are involved in the generation of the specification of the application domain, the description of the process model capturing the software behavior, and the implementation of the software. The instrumented software is generated by an LLM by bridging the high-level design with the low-level implementation. Users interact with the instrumented software, which generates logs throughout its execution. These logs are automatically analyzed by a conformance checker, which provides fuzzy diagnoses. Finally, an anomaly detector uses the fuzzy diagnoses to verify the presence of control-flow anomalies.

control over the environment in which the software is deployed [15]. These challenges are particularly pronounced in modern software systems, where maintaining strict consistency between system models and their implementations is difficult due to system complexity and multi-developer settings [16–18]. Moreover, increasingly complex systems introduce incompleteness and uncertainty, including so-called "unknown unknowns" [19] and, more specifically, residual faults [20], which cause anomalous behavior that cannot be anticipated due to the lack of prior knowledge of their existence. Such emergent unwanted behavior can have multiple causes, such as system integration of different, possibly commercial-off-the-shelf components or lack of resources to evaluate all the possible normal and faulty scenarios of the system [21,22].

To account for unforeseen behaviors and environmental uncertainty at runtime, control-flow anomaly detection can exploit machine/deep learning approaches to characterize the observable normal behavior directly from log data and use such characterization to check for deviations at runtime [23–28]. While these approaches are more flexible and domain-agnostic, they suffer from significant and broadly documented limitations related to their explainability and trustworthiness, which hinder their deployment in critical scenarios [29]. Moreover, their data-driven and domain-agnostic nature loosens the relationship with the software specification, which makes connecting their diagnoses with high-level designs more challenging.

In this paper, we address (1) the limitations of traditional runtime verification in terms of its difficulties in capturing all possible properties and assumptions about the monitored system, and (2) the trustworthiness of machine/deep learning control-flow anomaly detection approaches and their lack of connection with high-level design. In particular, our proposal combines a new instrumentation technique capable of automatically connecting software implementations with high-level designs and conformance checking, an effective and explainable method for control-flow anomaly detection. The instrumentation technique is based on Large Language Models (LLMs). LLMs are advanced neural networks trained on vast amounts of code and textual data, enabling them to understand software structure and control flow [30]. By leveraging these capabilities, LLMs can automatically generate instrumentation code and allow obtaining high-quality software logs, which is a key requirement for quality conformance checking results [31–33].

Combining LLMs and conformance checking, we formulate the following two Research Questions (RQs):

- **RQ1 (Source-code instrumentation)**: How can LLMs be leveraged to bridge high-level design models and software implementations and generate high-quality logs for subsequent monitoring through conformance checking?
- **RQ2 (Control-flow anomaly detection)**: How effective is conformance checking-based control-flow anomaly detection in identifying software anomalies using LLM-enabled software instrumentation?

To address these RQs, we propose a methodology enabling the development of conformance checking-based monitors, whose high-level view is shown in Fig. 1. Based on the application domain, a requirements engineer develops a specification. This is used to drive the design of a prescriptive process model and the development of the software. The software is instrumented by an LLM, which bridges the high-level design with the low-level structure and control-flow of the software. The software log is generated when a user interacts with the system and compared with the process model through a conformance checker. This results in fuzzy diagnoses that an anomaly detector processes to find any control-flow anomalies. In conclusion, our proposal addresses the challenge of bridging software implementations with high-level software design through LLM-based software instrumentation, and allows for the flexible alignment and deviation identification using these designs through conformance checking.

We test the methodology and the conformance checking-based monitor on a case-study scenario from the European Rail Traffic Management System/European Train Control System (ERTMS/ETCS), a railway standard setting the specification for software development and testing of modern interoperable railways [34]. The ERTMS/ETCS requirements specification document, SUBSET-026, prescribes the behavior of train components in all reference operational scenarios.[1] The results obtained from the ERTMS/ETCS case study demonstrate that LLM-based source-code instrumentation can achieve up to 84.755% control-flow coverage, while the subsequent conformance checking-based anomaly detection reaches a peak performance of 96.610% F1-score and 93.515% AUC.

In summary, the novel contributions of this paper are:

- A fuzzy and explainable conformance checking-based runtime monitor capable of detecting control-flow anomalies from software logs generated by LLM-instrumented code.
- A methodology to guide LLM-enabled source-code instrumentation and development of the fuzzy and explainable conformance checking-based monitor for software monitoring of computer-based systems.

[1] https://www.era.europa.eu/era-folder/1-ccs-tsi-appendix-mandatory-specifications-etcs-b4-r1-rmr-gsm-r-b1-mr1-frmcs-b0-ato-b1

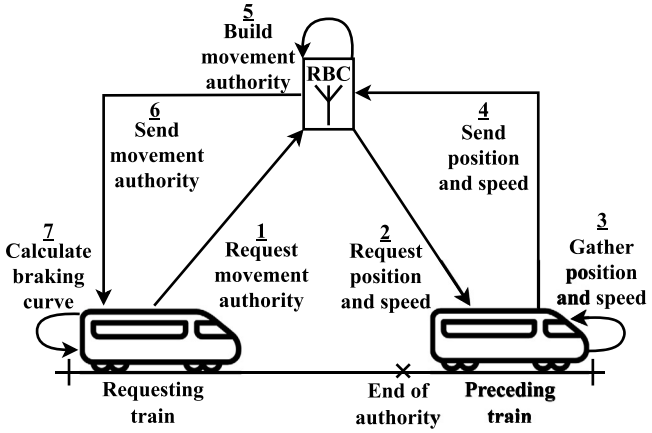


Fig. 2. Request of the movement authority and calculation of the braking curve by a train in ERTMS/ETCS systems.

- The application of the methodology and the evaluation of the monitor capabilities referencing the ERTMS/ETCS standard, a real-world case study documenting the specification of European railways.

The rest of the paper is organized as follows. Section 2 motivates the research work through a detailed description of ERTMS/ETCS systems. Section 3 presents the related works on software logging and anomaly detection. Section 4 describes the conformance checking-based monitor architecture. Section 5 describes the methodology driving the engineering and use of conformance checking-based software monitoring. Section 6 describes the case-study scenario from the ERTMS/ETCS standard, the techniques employed for control-flow anomaly detection, and the experimental factors and metrics. Section 7 presents the results. Section 8 discusses the results as responses to the research questions, and presents the threats to validity. Section 9 concludes and hints about future research directions.

2. Motivation

Modern computer systems will run within increasingly variable and unpredictable environments. A running example are railway systems, which are characterized by many sources of uncertainty, such as environmental conditions, railway traffic, and connection issues between on-board and trackside equipment [35]. In this paper, we target the safety-critical ERTMS/ETCS railway systems, which control the operation of trans-European railway traffic.

An ERTMS/ETCS railway system involves the digital elaboration of data generated by trackside and on-board equipment [36]. Essential on-board components are the Driver Machine Interface (DMI), European Vital Computer (EVC), Balise Transmission Module (BTM), and the Radio Transmission Module (RTM): the DMI enables the driver to engage with ERTMS/ETCS procedures and control the train; the EVC implements the needed logic for safely managing data flow within the on-board subsystem and between on-track and on-board communications; the BTM reads information through Eurobalises; and the RTM handles all the needed train-to-infrastructure communication logic. On the trackside, the RBC supervises trains in its area and handles several functionalities through the interlocking system to ensure efficient and safe train operation. The type of processing depends on the level of operation, from level 1 to level 3, which defines the degree of automation and the needed trackside and on-board equipment. At level 1, the system uses active Eurobalises to transmit signals to the train, providing discontinuous signaling. Level 2 introduces continuous radio communication between the train and trackside equipment, enabling more efficient operation. Level 3 adds moving block signaling and train integrity check equipment for even higher capacity. Let us consider an example ERTMS/ETCS critical task: the acquisition of the movement authority by a train. The movement authority provides information about the distance from other trains to the EVC of a train. The EVC must calculate the braking curve based on such information and on the static speed profiles to ensure the safety of the passengers and infrastructure. Fig. 2 shows the acquisition of the movement authority in ERTMS/ETCS level 2 and level 3, which enables the calculation of the braking curve of the requesting train by letting the RBC collect the train positions wirelessly.

The case-study ERTMS/ETCS scenario that we target in our experiments is the Start of Mission (SoM) procedure. This procedure is rather complex and critical, as its goal regards obtaining a movement authority at the start of a journey. The procedure includes: validating the driver's identity; registering to the closest RBC for train supervision; reporting the train position; obtaining railway traffic information; and obtaining the movement authority.

Although SoM is described in the ERTMS/ETCS system requirements specification, it is mostly specified in natural language and can be subject to interpretation ambiguity among different manufacturers of system components. In fact, extensive field integration testing involving different companies is usually required, which however cannot exclude interaction anomalies in unspecified situations and edge cases. Software monitoring can identify these anomalies by tracing and checking key information regarding software executions. In the next section, we are exploring related works on software monitoring and outlining the key novelty of this work's proposal in comparison to existing approaches.

3. Related work

This section provides an overview of the existing work regarding software monitoring, which we split into software logging and software anomaly detection.

3.1. Software logging

Binary-code instrumentation. literature has proposed many binary-code instrumentation frameworks. Engelke and Schulz [42] presented Instrew, which is similar to Valgrind, but optimizes instrumentation intrusiveness by leveraging the LLVM compiler infrastructure. Mu et al. [45] developed FlexInstru, which introduces a hardware-independent dynamic instrumentation framework. By leveraging a novel process attachment/detachment mechanism and an instrumentation sampling strategy, FlexInstru allows for flexible control over instrumentation duration, effectively balancing profiling accuracy with runtime overhead. While binary-code instrumentation is very flexible and can be optimized to reduce its overhead regarding software execution time and program size, source-code instrumentation typically leads to more interpretable and higher-quality logs. This is pivotal to obtaining meaningful descriptions of software executions.

Source-code instrumentation. Briand et al. [37] proposed the instrumentation of Java programs to record objects' interactions and log the sequences of messages exchanged. Their objective was to reverse engineer the program and automatically obtain a sequence diagram to enhance the understanding of the software's internals. Cinque et al. [11] introduced the rule-based source-code instrumentation technique, which involves placing logging instructions in specific places in the program according to a system representation developed at design time. This allows for the collection of high-quality logs linked to the entities of the system and their interactions. Zhang et al. [38] proposed a declarative approach to tag the structures of Java programs and insert logging instructions corresponding to those structures. The approach requires querying the program and specifying instrumentation commands, which can include logging instructions.

Table 1
Comparison of the reviewed literature in terms of instrumentation of software logging and anomaly detection. The symbol "—" indicates that the reference did not detail or include the related functionality. ML = Machine Learning, RV = Runtime Verification, PM = Process Mining.

Ref.	Logging			Anomaly detection		
	Type	Flexible	Log semantics	Type	Fuzzy	Explainable
[37]	Source code	No	High-level (object interactions)	–	–	–
[11]	Source code	No	High-level (service interactions)	RV	No	Yes (intrinsic)
[38]	Source code	No	High-level (object interactions)	–	–	–
[12]	Indirect	Yes	Low-level (system calls)	RV	No	Yes (intrinsic)
[23]	–	–	–	ML	Yes	Yes (intrinsic)
[24]	–	–	–	ML	Yes	Partly (post-processing)
[39]	ML	Yes	High-level (explicit messages)	–	–	–
[40]	–	–	–	PM	Yes	Yes (intrinsic)
[13]	Source code	No	Low-level (software variables)	RV	No	Yes (intrinsic)
[41]	–	–	–	PM	Yes	Yes (intrinsic)
[42]	Binary code	Yes	Low-level (system calls, memory access)	–	–	–
[25]	–	–	–	ML	Yes	No
[26]	–	–	–	ML	Yes	No
[27]	–	–	–	ML	Yes	Partly (post-processing)
[43]	ML	Yes	High-level (explicit messages)	–	–	–
[44]	ML	Yes	High-level (explicit messages)	–	–	–
[28]	–	–	–	ML	Yes	Partly (post-processing)
[10]	–	–	–	PM + ML	Yes	Yes (intrinsic + post-processing)
[45]	Binary code	Yes	Low-level (branches, instructions)	–	–	–
This work	ML	Yes	High-level (design-time model activities)	PM + ML	Yes	Yes (intrinsic + post-processing)

Machine learning-based instrumentation. Recent research has also leveraged machine learning to automate the placement and generation of log statements. Jia et al. [39] utilized random-forest classifiers to identify necessary logging points within exception blocks. Addressing content generation, Mastropaolo et al. [43] employed T5 transformers to automatically produce complete log statements based on the surrounding source code. Most recently, Wang et al. [44] applied LLMs via in-context learning to both locate logging opportunities and generate descriptive messages, achieving higher accuracy than previous deep learning approaches.

3.2. Software anomaly detection

Runtime verification. approaches to identifying any anomalous behavior in software involve runtime verification of declarative specifications. Cinque et al. [11] developed rule-based source-code instrumentation to verify whether the modeled software interactions between objects and internal services failed due to missing variable initialization, wrong value assigned to a variable, or missing function call. Cimatti et al. [13] presented an LTL-based runtime verification framework where a reference model is used to define assumptions and constraints on the software's behavior. Kejstová et al. [12] extended the idea of runtime verification by proposing runtime model-checking frameworks. The framework involves collecting the software state during its execution to restrict the state space of the reference software model. In this way, model-checking of declarative specifications becomes feasible at runtime due to the reduced time needed to check the specification properties. It is worth noting that the mentioned works may exploit some form of software instrumentation (e.g., source-code instrumentation) or monitor the software indirectly through, e.g., its system calls.

Machine learning-based anomaly detection. runtime verification is effective for verifying software behavior, it requires accurate software specifications and models. Hence, the literature put forward several data-driven solutions for software anomaly detection. For example, Singh et al. [23] proposed a rule-based framework for anomaly detection that involves the selection of an optimized set of software features and the extraction of fuzzy rules that discriminate correct behaviors from anomalous ones. Denaro et al. [25] proposed a neural network-based unsupervised anomaly detection framework. The solution employs deep autoencoders trained using software key performance indicators to discriminate whether the software execution is correct. It is worth noting that domain-agnostic data-driven methods grounded in deep learning with extension to the software domain have also been proposed. For example, Du et al. [24] have proposed DeepLog, a method based on long–short term memory networks, which are a type of recurrent neural network. The architecture is able to capture deviations between the predicted and observed events, and detect anomalies in system execution. Extensions to recurrent neural networks are those integrating the reconstruction-based anomaly detection paradigm through autoencoders, such as the method proposed in [27]. Recently, LLMs have also been adopted for anomaly detection in system logs. Han et al. [26] introduced LogGPT, which is an LLM-based framework that involves LLM pre-training, reinforcement learning-based fine-tuning, and online inference to predict whether new log sequences are normal. Zhang et al. [28] extended this paradigm to include a human-in-the-loop explanation process.

Process mining-based anomaly detection. data-driven approaches have proven effective, they require laborious feature extraction through ad-hoc selection of software metrics and/or trace encodings. In addition, the limited trustworthiness and explainability of advanced machine learning limit its use. A promising area of research that addresses these issues is process mining, which combines traditional model-based analyses with data-driven insights. Cinque et al. [40] evaluated the capability of process mining to discover the normal behavior of the Apache web server as Petri nets with several algorithms, including the α-miner, inductive miner, and integer linear programming-based miner. Subsequently, the authors performed a sensitivity analysis for the threshold to assign to the fitness metric for discriminating normal and anomalous behavior. Similarly, Pecchia et al. [41] evaluated the anomaly detection capability of process mining. The sensitivity analysis related to thresholding fitness assessed that higher fitness values lead to improved detection performance. However, the authors also remark that the quality of the model being discovered is critical to quality detection performance. In fact, they correlated noise in event logs handled by process discovery with negative effects on performance. In particular, as noise increases in event logs, there is a higher chance of false negatives, i.e., more misclassification of anomalies. Vitale et al. [10] developed a domain-agnostic process mining-based framework that performs feature extraction through conformance checking diagnoses and integrates an explainable machine learning component that exploits such diagnoses for control-flow anomaly detection.

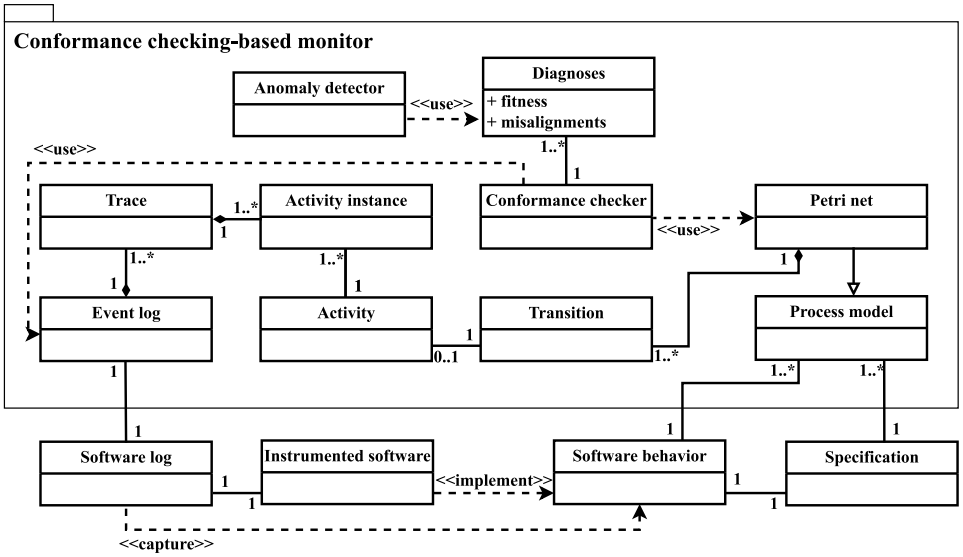


Fig. 3. Conceptual UML class diagram of the conformance checking-based monitor and other related methodological artifacts.

3.3. Our approach

Table 1 compares our work to the reviewed literature with respect to both software logging and anomaly detection. From the comparison, the main novelty of our approach is enabling conformance checking-based control-flow anomaly detection through LLM-based instrumentation. In fact, by utilizing LLMs to align code logic with design-time model semantics, we automate the generation of logging lines that are not only syntactically correct but also semantically meaningful and highly interpretable. This allows extending the process mining-based control-flow anomaly detection framework proposed in [10] to the software domain.

4. The conformance checking-based monitor architecture

The conformance checking-based monitor connects well-known concepts of the process mining community and software artifacts produced in traditional software development processes. On the one hand, process mining includes the concepts of process models and event logs, the two fundamental inputs to conformance checking. On the other hand, the software artifacts allow generating these inputs, hence their presence is pivotal to the application of conformance checking. Fig. 3 shows a conceptual UML class diagram of the overall architecture. In this section, we are interested in describing the monitor architecture. The other concepts will be explored in detail in Section 5.

The main goal of the monitor is to generate diagnoses through the **conformance checker** and use these diagnoses to detect control-flow anomalies through the **anomaly detector**.

4.1. Event log

Conformance checking algorithms use event logs, which are structured data that collect uniquely identified activity instances of a reference process [46]. In this paper, we focus on sequences of activity instances, i.e., traces; thus, we consider the following definition:

Definition 4.1 (*Event Log*)**.** Let $\mathcal{A}$ denote the universe of activities. Let $\mathcal{A}^*$ denote the universe of traces. An example trace is $\langle \sigma_1, \sigma_2, \dots, \sigma_n \rangle$. Let $\mathcal{B}(\mathcal{A}^*)$ indicate the set of multisets of traces. An event log is an element of $\mathcal{B}(\mathcal{A}^*)$, i.e., $L \in \mathcal{B}(\mathcal{A}^*)$. An example event log is

$$L = [\langle \sigma_1, \dots, \sigma_\alpha \rangle^a, \langle \sigma_1, \dots, \sigma_\beta \rangle^b, \langle \sigma_1, \dots, \sigma_\gamma \rangle^c],$$

where $\sigma_\alpha, \sigma_\beta, \sigma_\gamma$ indicate the last activity instance within the corresponding trace, and a, b, c indicate the number of times the three traces appear in the event log.

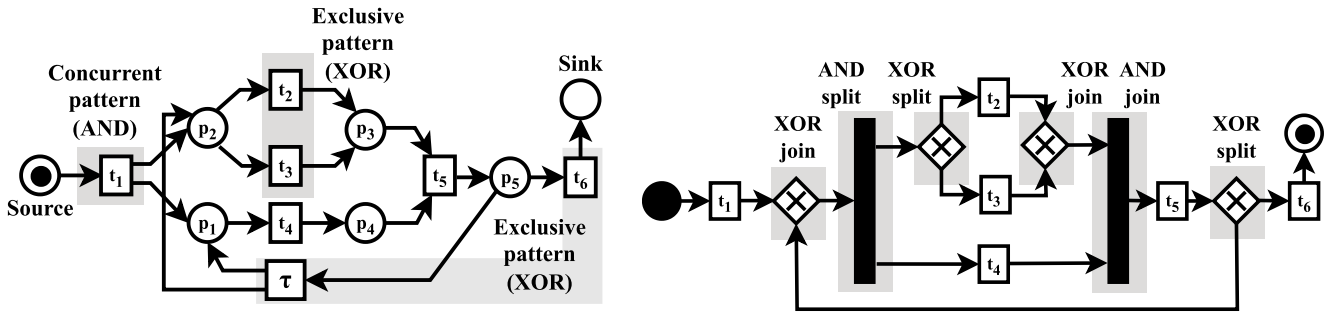
The event log is associated with a software log recorded from the execution of the software instrumented through LLMs. The criticality lies in maintaining a reliable association between the activities of the event log and the process model. This will be explored in more detail in the instrumentation phase of the methodology detailed in Section 5.

4.2. Process model

Event logs are checked against a process model, which is a behavioral model that captures control-flow relationships between activities. The most common formalism employed by process mining is the Petri net, and, in particular, we focus on the labeled accepting Petri net variant [46].

Definition 4.2 (*Labeled Accepting Petri Net*)**.** Let P and Tr be two sets of nodes of a bipartite graph such that $P \cap Tr = \emptyset$, and $F \subseteq (P \times Tr) \cup (Tr \times P)$ a set of directed arcs. A Petri net is the triple (P, Tr, F). Its marking $M \in \mathcal{B}(P)$ is a multiset of tokens and encodes the current state. Petri nets that have an initial marking M_0 and a final marking M_f, i.e., an initial and final state, are called accepting Petri nets. Furthermore, let $A \subseteq \mathcal{A}$ be a set of activities and $l_{Tr} : Tr \to A \cup \{\tau\}$ a function that associates to elements of Tr either an activity of A or the "silent" label τ, which represents unobservable activities that add legitimate behaviors not directly visible. A *labeled accepting Petri net* is the tuple $(P, Tr, F, M_0, M_f, A, l_{Tr})$. $\mathcal{N}$ denotes the universe of all labeled accepting Petri nets, hereafter referred to as Petri nets.

The formal semantics of Petri nets include the ability to *fire* a transition. Given the number of incoming arcs of a transition, if the current marking is such that there is at least a token in each place connected to the transition, it can fire, i.e., the transition consumes one token from each incoming place and generates one token for each outgoing arc. This firing dynamic enables the Petri net to evolve its state, allowing for the above-mentioned conformance checking and simulation. In addition, it is worth noting that there is a special class of Petri nets, namely workflow Petri nets, which have two specific places: a source place and a sink place. In these Petri nets, M_0 assigns a token to the source place. Any complete firing sequence moves the Petri net state from M_0 to M_f, which consists of at least a token in the sink place.



(a) Petri net with gray-highlighted control-flow patterns.

(b) Trace-equivalent UML activity diagram.

Fig. 4. Petri net and its trace-equivalent UML activity diagram.

Fig. 4(a) depicts an example Petri net made of places `source`, `sink` and $\mathtt{p}_{1,\dots,5}$, and transitions $\mathtt{t}_{1\dots6}$ and τ. The figure highlights in gray the control-flow patterns that the Petri net captures, such as the concurrent pattern (i.e., the AND pattern) linked to $\mathtt{t}_1$, $\mathtt{p}_2$ and $\mathtt{p}_3$. This pattern allows the concurrent execution of transition $\mathtt{t}_4$ and either transition $\mathtt{t}_2$ or $\mathtt{t}_3$. A desirable property of Petri nets is ensuring that the final marking is always reachable and that no transitions or places become dead. This property, met by the example Petri net, guarantees a complete and token-free traversal of the net, ensuring every transition has a viable path from start to finish.

Notably, some transformation rules can also be applied to map semi-formal models to formal models. Fig. 4(b) shows a trace-equivalent UML activity diagram of the Petri net in Fig. 4(a). The XOR and AND patterns are explicitly encoded by graphic elements of the UML notation. The UML activity diagram is trace-equivalent since any trace of the Petri net is also allowed on the UML diagram. For example, the trace $\langle t_1, t_2, t_4, t_5, t_3, t_4, t_5, t_6\rangle$ can be executed on the UML activity diagram as follows: t_1 initiates two concurrent flows, with the upper path implementing an XOR choice that allows t_2 to execute, followed by t_4, and then t_5 after synchronization at the AND join. After this, the XOR split cycles back to the concurrent flow, where t_3 and t_4 execute, synchronize, and allow t_5 again, before the process concludes with t_6. This illustrates how the UML diagram faithfully represents both concurrency and exclusive choices encoded in the Petri net.

4.3. Conformance checking and anomaly detection

As the event log and Petri net are well-defined, the conformance checker can now implement a conformance checking technique to verify whether the traces of an event log follow the control-flow patterns prescribed by a reference Petri net. Such verification leads to a series of conformance-checking **diagnoses** that record two types of global and local information: the fitness and misalignments. These can be computed through alignment-based conformance checking, which first evaluates the alignments between traces and a reference Petri net.

Definition 4.3 (*Alignment and Moves*)**.** Let us denote $\gg$ as the "skipped" activity. Let $N \in \mathcal{N}$ be a reference Petri net and $\sigma \in \mathcal{A}^*$ a trace. $\sigma_L = \langle a_1, \dots, a_o\rangle \in (\mathcal{A}\cup\{\gg\})^*$ is a (finite) sequence of log moves related to σ if and only if $\sigma_L \setminus \{\gg\} = \sigma$. Then, $\sigma_N = \langle b_1, \dots, b_o\rangle \in (\mathcal{A}\cup\{\tau\}\cup\{\gg\})^*$ is a (finite) sequence of model moves if and only if $\sigma_N \setminus \{\gg\}$ is a firing sequence of N, i.e. an allowed control flow. An *alignment* $\gamma_{\sigma,N}$ is the two-row matrix

$$\gamma_{\sigma,N} = \begin{array}{c|c|c|c} a_1 & a_2 & \cdots & a_o \\ \hline b_1 & b_2 & \cdots & b_o \end{array},$$

if for all $1 \le i \le o$, $(a_i, b_i) \neq (\gg, \gg)$, where (a_i, b_i) is a *move*. The upper row is the sequence of log moves (log sequence) and the bottom row is the sequence of model moves (model sequence). We define $|\gamma_{\sigma,N}|$ the *alignment length*, i.e., the number of moves of the alignment.

Alignment-based conformance checking attempts to find the *closest* (best) alignment between a reference Petri net and a trace. Fig. 5 shows the four best alignments against the Petri net in Fig. 4(a) linked to four faulty traces $\sigma_{1,\dots,4}$. The red-highlighted alignment parts indicate mismatches between the trace and the Petri net due to violating some control-flow constraints. For example, σ_1 (leftmost trace) has wrongly-ordered activities, as $\mathtt{t}_5$ precedes $\mathtt{t}_4$ and $\mathtt{t}_2$ instead of being executed after their appearance. This violation leads to the red-highlighted model-log sequence pairs, which can be recorded as additional diagnoses to use in the methodology's further steps. Once the best alignments are found, the fitness can be computed.

Definition 4.4 (*Fitness*)**.** Let us denote $\Gamma_{\sigma,N}$ as the set of all alignments between a trace $\sigma \in \mathcal{A}^*$ and a Petri net $N \in \mathcal{N}$. Let δ be the unitary cost function for a pair of moves or an alignment. Finally, let $\gamma^w_{\sigma,N} \in \Gamma_{\sigma,N}$ be the worst-case alignment and $\gamma^*_{\sigma,N} \in \Gamma_{\sigma,N}$ the best-case alignment, i.e., the alignments with the least and most costs according to δ, respectively. The *alignment-based fitness* for σ is defined as

$$F_{\sigma,N} = 1 - \frac{\delta(\gamma^*_{\sigma,N})}{\delta(\gamma^w_{\sigma,N})}.$$

Let $L \in \mathcal{B}(\mathcal{A}^*)$ be an event log. The *alignment-based fitness* for L is defined as

$$F_{L,N} = \frac{\sum_{\sigma\in L} F_{\sigma,N}}{|L|}.$$

The misalignments are counters associated with each transition $tr \in Tr$ of the reference Petri net that record the number of mismatches between the log sequences and model sequences involving tr. For example, Fig. 5 outlines that there are 2 $\mathtt{t}_5$ mismatches in the best alignment of σ_1. A detailed description of how to calculate misalignments can be found in [10]. The misalignments and fitness constitute the diagnoses.

Definition 4.5 (*Diagnoses*)**.** Let $\mathcal{A}_\alpha \subseteq \mathcal{A}$ be the number of activities of the target software and $L \in \mathcal{B}(\mathcal{A}^*_\alpha)$ be an event log of k traces and $N \in \mathcal{N}$ a Petri net. The conformance checker collects the diagnoses $D \in \mathbb{R}^{k\times(\alpha+1)}$ as in Table 2 by alignment-based conformance checking of each trace $\sigma \in L$ against N.

These can be used by an anomaly detector to verify whether a trace is anomalous based on both its number of misalignments and the fitness value. Control-flow anomaly detection is performed by classifying each trace of the diagnoses.

Definition 4.6 (*Control-Flow Anomaly*)**.** Let $D \in \mathbb{R}^{k\times(\alpha+1)}$ be the diagnoses collected by the conformance checker. Let $d_\sigma \in D$ be the diagnoses associated with a trace σ. σ exhibits an *control-flow anomaly* if the anomaly detector classifies d_σ as anomalous.

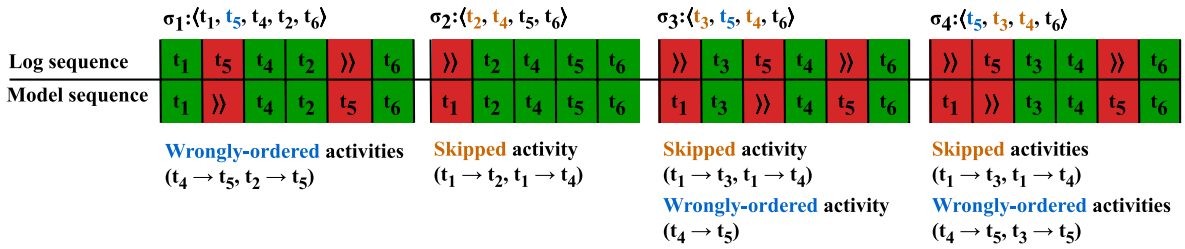


Fig. 5. The best alignments associated with four faulty traces against the Petri net in Fig. 4(a). The red-highlighted alignment parts indicate mismatches between the trace and the Petri net..

Table 2
Alignment-based conformance checking diagnoses obtained from replaying the traces $\sigma_1 \dots \sigma_{|L_o|}$ of event log L_o against a reference Petri net N.

Trace	a_1	a_2	⋯	$a_{\alpha-1}$	a_α	F_σ
σ_1	u_{a_1,σ_1}	u_{a_2,σ_1}	⋯	$u_{a_{\alpha-1},\sigma_1}$	u_{a_α,σ_1}	F_{σ_1}
σ_2	u_{a_1,σ_2}	u_{a_2,σ_2}	⋯	$u_{a_{\alpha-1},\sigma_2}$	u_{a_α,σ_2}	F_{σ_2}
⋯	⋯	⋯	⋯	⋯	⋯	⋯
σ_k	u_{a_1,σ_k}	u_{a_2,σ_k}	⋯	$u_{a_{\alpha-1},\sigma_k}$	u_{a_α,σ_k}	F_{σ_k}

Given the structure of diagnoses, any unsupervised anomaly detection algorithm can be used. This class of algorithms identifies patterns or deviations without requiring labeled data, making them suitable for scenarios where normal and abnormal behaviors are not explicitly defined [47]. In this context, the algorithm interprets the diagnosis metrics, such as misalignment counts, fitness scores, and other derived features, as indicators of behavioral conformity. By modeling the distribution of these indicators across multiple traces, it can flag outliers that significantly deviate from the expected process behavior, thereby signaling potential anomalies in the system execution. The training of the anomaly detector will be detailed in the following section.

5. The methodology for LLM-enabled software control-flow anomaly detection

This section delves into the methodology that we devised to enable the use of the conformance checking-based monitor defined in Section 4. The methodology is depicted in Fig. 6 and split into three phases: **software development**, **software monitoring design** and **software anomaly detection**. Each phase involves different steps, which can either be entirely *manual*, *LLM-based*, *scripted* with *human-in-loop*, or entirely scripted. Manual steps are performed by humans without relying on scripted code or LLMs; these are mostly linked to software development, where a high level of human supervision is required. The only LLM-based step is software instrumentation, which is triggered by a prompt. Software executions are both scripted and human-in-loop, as these usually involve some degree of human interaction. The rest of the steps are entirely automated through scripted code.

5.1. Software development

5.1.1. Requirements engineering and software design

In the **requirements engineering step**, a requirements engineer analyzes the software system requirements and generates a System Requirements Specification (SRS) document reporting the software system's functional and non-functional aspects. The functional aspects regard the use cases that the software must implement, whereas non-functional aspects regard performance and dependability metrics. For example, the developers of the ERTMS/ETCS standard produced the ERTMS/ETCS SRS document (SUBSET-026). The dynamic aspects of the standard procedures are described in the document's chapter 5 (SUBSET-026-5), including the SoM scenario.

As will be shown in Section 6, the SRS is mostly written in natural language and can be subject to interpretation ambiguity. This ambiguity can initially be solved with the development of formal models in the subsequent **software design** step. The SRS is used to drive this step and design of the software architecture, which is captured by several diagrams describing static and dynamic aspects. Static aspects involve the description of components' responsibilities and their interconnection. Dynamic aspects involve the procedures that components execute and how the procedures relate to each other.

Regarding the dynamic aspects, software design typically relies on high-level modeling languages such as UML activity diagrams and the Business Process Modeling Notation (BPMN). As illustrated in Fig. 4, the labeled accepting Petri net (Definition 4.2) provides a formal bridge from these semi-formal languages to a rigorous execution framework. In this mapping, activities correspond to labeled transitions, while control-flow constructs such as loops and choices are represented through networks of arcs, places, and silent (τ) transitions.

Crucially, unlike plain Petri nets, the accepting property (defined by an initial marking M_0 and a final marking M_f) allows capturing the execution semantics of software design: high-level notations such as UML and BPMN explicitly specify start and end points, which can be directly mapped to the source place (containing the initial marking M_0) and the sink place (which should exclusively contain a token upon trace completion, corresponding to the final marking M_f).

Moreover, the use of labeled accepting Petri nets also enables the alignment-based conformance checking logic of the proposed monitor (Section 4). In fact, during this process, τ-transitions are handled by assigning them a move cost of zero, whereas the alignment with the expected sequencing of labeled transitions enables quantifying the degree of conformance to prescriptive behavior. The entire process is also strongly supported by the existing tools and libraries, such as `pm4py`[2] and `ProM`.[3] It is worth noting that partial or interrupted runs can penalize the resulting diagnoses, marking more misalignments. This is due to the traces resulting from partial or interrupted runs reaching an invalid final marking. In this case, prefix-alignments should be adopted [48].

During software design, the development team can start the verification and validation (V&V) process through *design-time testing* to ensure that the design artifacts meet the requirements. Comprehensive V&V provides the evidence needed to support safety claims and regulatory approval, especially in the context of software certification for compliance with relevant standards such as EN 50128, IEC 61508, or ISO 26262. In this case, the use of Petri nets is even more advantageous due to their formal syntax and semantics, which allow verifying whether the dynamic description leads to deadlocks, unwanted control-flow relationships, or simply invalidates the requirements. Despite its importance, design-time testing is not sufficient to prove the correctness of the software; more testing is needed in the subsequent phases.

[2] https://processintelligence.solutions/pm4py/
[3] https://promtools.org/

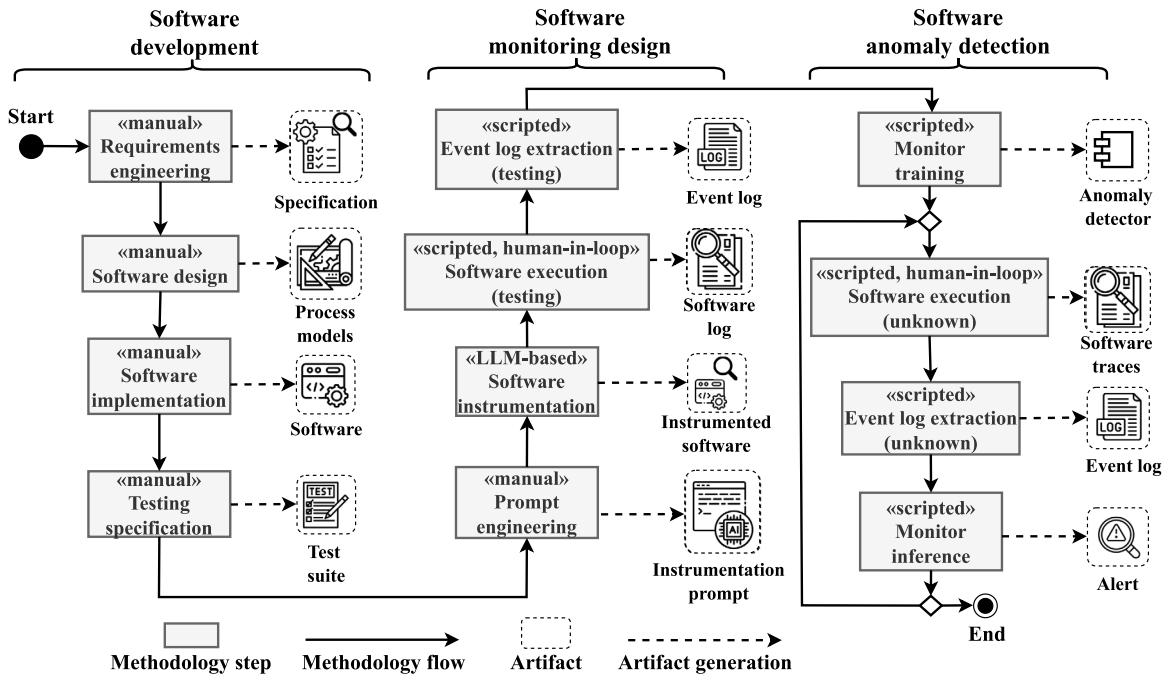


Fig. 6. The methodology for LLM-enabled software control-flow anomaly detection. The methodology is split into the software development, software monitoring design, and software anomaly detection phases. The software development phase follows the traditional development steps to specify, design, implement and test the software. The software monitoring design phase uses LLMs to link process models with the software implementation to enable the extraction of event logs during runtime testing. The software anomaly detection phase builds the anomaly detector using the logs collected during runtime testing for subsequent control-flow anomaly detection through conformance checking.

5.1.2. Software implementation and testing specification

Once the software has been thoroughly documented and its behavior, structure, and interactions have been described through process models, the next step is **software implementation**. At this stage, developers translate conceptual representations into executable code, and V&V progresses by employing white-box testing and black-box testing to ensure that the software implementation conforms to its specifications and fulfills intended requirements [49]. On the one hand, white-box testing evaluates the software correctness by inspecting its structure either statically or dynamically. It typically involves developing test suites to cover the majority of statements, paths, and/or conditions. On the other hand, black-box testing evaluates the software correctness by assessing the output conditioned by a given input. It also involves developing test suites, although they are aimed at defining what the correct output is. Through white-box and black-box testing, the **testing specification** step generates a test suite.

However, despite the completeness of the test suite, runtime conditions may give rise to so-called "unknown unknowns", which are situations that were not anticipated before the software system is deployed [19]. These situations can occur due to *residual faults*, which are defects that escape test suites [20]. Detecting the activation of residual faults is important for maintaining system safety and reliability, as these faults lead to latent errors that may eventually cause system failures [50]. One effective approach is to employ a runtime monitor that tracks software behavior as it unfolds, enabling the system to respond to unexpected or unpredictable conditions [8].

5.2. Software monitoring design

5.2.1. Prompt engineering and software instrumentation

In our methodology, the runtime monitor requires checking deviations from the prescriptive design-time models by analyzing software logs obtained from software executions. Among the different types of instrumentation techniques, source-code instrumentation is particularly promising, as it enables collecting semantically meaningful events tied to the software's structural representation. However, since software implementations often deviate significantly from high-level descriptions, bridging low-level software constructs with design-time models remains challenging. To address this, we propose using LLMs to assist in source-code instrumentation, leveraging their ability to (1) encode domain knowledge and (2) infer semantic structure from code.

While LLMs are powerful tools, it is widely acknowledged that their output is strongly dependent on the prompts used. This gave rise to **prompt engineering**, an activity concerned with designing prompt structures that effectively guide the model toward producing accurate, relevant, and contextually appropriate responses. The following is the logging prompt that we will use to instrument the code. Thus, in the **software instrumentation** step, the LLM is provided with the logging prompt, the reference process model, and the software. By acknowledging the dynamics described in the process model and mapping them to the software constructs, the LLM is capable of bridging the high-level model with the low-level implementation, providing an instrumented software that generates quality software logs.

5.2.2. Software execution and event log extraction

The test suite generated during software development can be used to stimulate the **software execution**. Such *testing* yields a software log, which contains the different events logged according to the instrumentation rules. However, while the LLM has used high-level descriptions of design-time models, the software log may still have spurious information that is not relevant to the high-level reference process model. Accordingly, **event log extraction** filters out spurious information and retains only the sequence of activities mapped to the process model activities, resulting in an event log compliant with Definition 4.1.

5.3. Software anomaly detection

5.3.1. Monitor training

To build the monitor, the event log extracted during software monitoring design is used in the **monitor training** step to train the anomaly detector to classify control-flow anomalies.

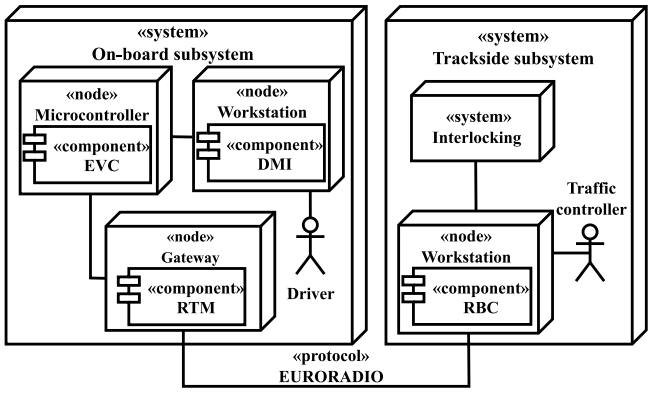


Fig. 7. Deployment diagram of ERTMS/ETCS railway systems, split into the on-board subsystem and trackside subsystem.

Definition 5.1 (*Monitor Training*)**.** Let Δ denote the universe of binary classifiers. Let $L \in \mathcal{B}(\mathcal{A}_\alpha^*)$ be an event log of k traces obtained from executing k tests against the instrumented software. Let $N \in \mathcal{N}$ be a Petri net described during the software development phase. Let $D \in \mathbb{R}^{k\times(\alpha+1)}$ be the diagnoses obtained by the conformance checker by aligning the traces of L against N. *Monitor training* is the function $\eta : \mathbb{R}^{k\times o} \to \Delta$, such that $\eta(D) = \delta$, where δ is the anomaly detector.

Through the anomaly detector, the conformance checking-based monitor is capable of classifying a new piece of data as either anomalous or normal. Many unsupervised machine learning algorithms can implement η and build a monitor. Two common approaches are clustering and dimensionality reduction [47]. Clustering involves grouping similar data points using a distance metric, such as the Euclidean distance. A threshold on the distance from the closest cluster can be set so that any new piece of data exceeding such distance is deemed anomalous. Dimensionality reduction involves building a new lower-dimensional reference system on which data are projected. Reconstructing data from the lower-dimensional reference system to the original one leads to reconstruction error. A threshold on the reconstruction error can be set so that any new piece of data exceeding such error is deemed anomalous.

5.3.2. Software execution, event log extraction, and monitor inference

When unknown software executions occur as the users interact with the software, new traces are collected using the event log extraction prompt above. To verify whether the software executed as intended, **monitor inference** handles the traces through conformance checking and evaluates whether it exhibits a control-flow anomaly according to Definition 4.6.

6. Case study

In this section, we demonstrate the application of the proposed methodology in Fig. 6 to the SoM procedure of the ERTMS/ETCS standard.

6.1. Software development

The ERTMS/ETCS SRS is mostly written in natural language and can be subject to interpretation ambiguity between different manufacturers of system components. In fact, extensive field integration testing involving different companies is usually required, which, however, cannot exclude interaction anomalies in unspecified situations and edge cases, and that can introduce additional uncertainties, which can be possibly managed using the approach described in this paper. The following are a few excerpts from SUBSET-025-5:

Excerpts from SUBSET-026-5, §5.4.2
Status of data stored in the ERTMS/ETCS on-board equipment

§5.4.2.1: At the beginning of the Start of Mission procedure, the data required may be in one of three states:

(a) **Valid** – the stored value is known to be correct.
(b) **Invalid** – the stored value may be wrong.
(c) **Unknown** – no stored value is available.

§5.4.2.2: This refers to the following data: Driver ID, ERTMS/ETCS level, RBC contact information, Train Data, Train Running Number, Train Position (see §3.6.1.3).

§5.4.2.3: Note 1: The status of data in relation to the previous and the actual mode is described in chapter 4, section "What happens to stored information when entering a mode".

§5.4.2.4: Note 2: The change of status of data in course of the procedure is shown in the table in §5.4.3.3.

Although these informal descriptions of the requirements are quite structured and are attached with detailed tables and, at times, state diagrams, they still need to be translated into artifacts that document the progressive refinement of natural language into more rigorous and possibly formal models.

We have designed two UML diagrams representing simplified views of the essential static and dynamic aspects of ERTMS/ETCS railway systems. Fig. 7 illustrates the core components of the on-board and trackside subsystems. This diagram captures the deployment of ERTMS/ETCS components—EVC, DMI, RTM, RBC, and interlocking—as specified in SUBSET-026-2, and depicts the interconnection of the on-board and trackside subsystems through the EURORADIO protocol.

Building on this structure and interpreting the requirements of SUBSET-026-5, §5.4, we have developed a UML activity diagram representing the high-level Start of Mission (SoM) control flow, shown in Fig. 8. The activity diagram abstracts implementation details such as specific message exchanges, data structures, and state-machine transitions, focusing instead on the logical sequencing of activities and decision points.

To provide clarity, the UML activities have been organized into four categories, corresponding to the essential phases of SoM: (1) initial data entry and validation; (2) RBC connection and position report; (3) driver intervention and train data; and (4) final mode assignment. Each activity is labeled using a repeatable and interpretable naming scheme: `som_<action_name>_<component>`, where `<component>` corresponds to one of the components depicted in Fig. 7, and `<action_name>` describes the specific action performed. This systematic labeling ensures repeatability and facilitates understanding across the SoM workflow.

As explained in the previous sections, transformation rules can be applied to obtain a trace-equivalent Petri net from the UML activity diagram. This is noteworthy since conformance checking algorithms are typically suited for the Petri net formalism.

The specification and the diagrams have guided the implementation of an SoM prototype in Python, which adds more details than the essential behavior of the high-level diagrams shown earlier. Listing 1 shows a code snippet of the prototype software implementing phase 1 of the SoM procedure, including the subroutine that handles the acquisition of the driver ID from the DMI.

Next, we developed a comprehensive test suite that instantiated 50 control-flow paths by stimulating the prototype with different inputs. The test suite must cover a wide range of possibilities, as the SoM procedure may instantiate many different control flows. Listing 2 shows one of the tests in the test suite aimed at forcing the software to provide full supervision to the train.

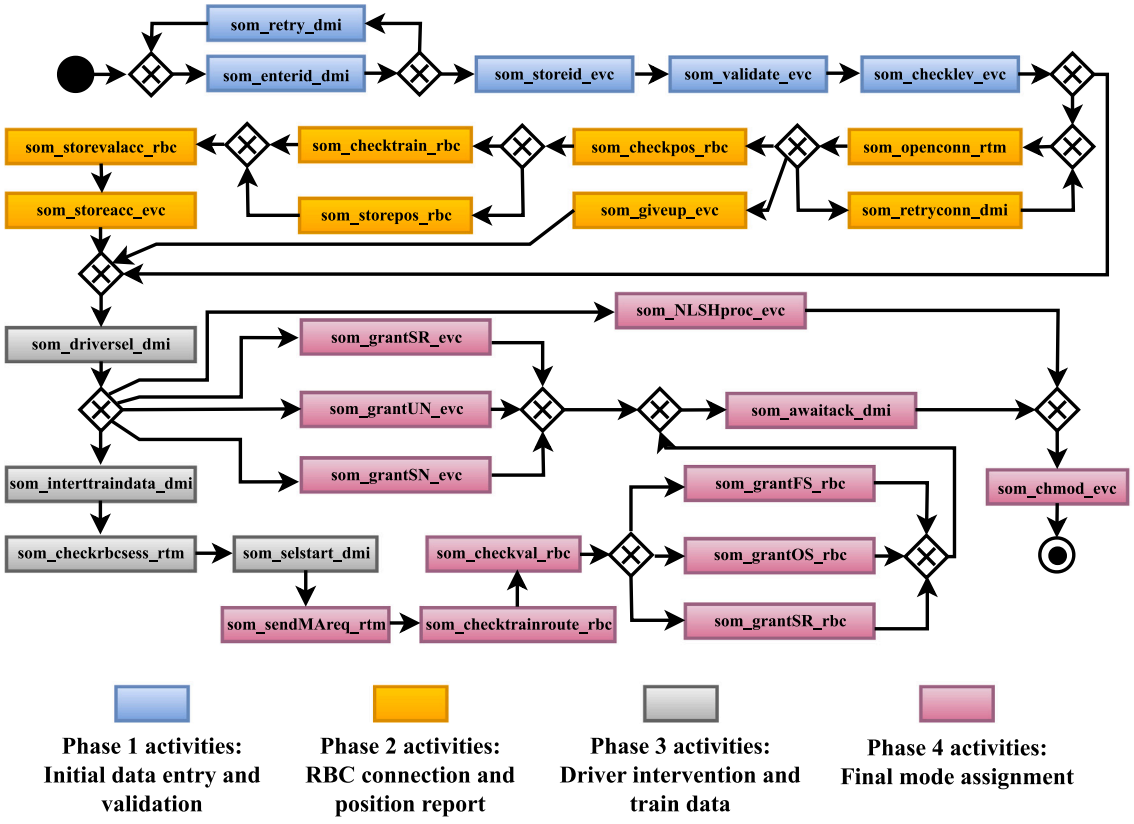


Fig. 8. High-level UML activity diagram describing the SoM procedure, split into four phases: initial data entry and validation; RBC connection and position report; driver intervention and train data; and final mode assignment.

```
def _phase_1_get_initial_data(self):
    if not (self.current_mode == Mode.STAND_BY and
    ↪ self.desk_open):
        return None
    state = self._procedure_s1_driver_id_entry()
    if state == 'D2':
        state = self._procedure_d2_check_pos_level()
    if state == 'S2':
        state = self._procedure_s2_level_entry()
    if state == 'D3':
        state = self._procedure_d3_check_level_valid
        ↪ ()
        if state == 'D7':
            return 'S3'
    return state

def _procedure_s1_driver_id_entry(self):
    if self.driver_id_status == DataStatus.UNKNOWN:
        self.driver_id = self.
        ↪ _simulate_driver_action("Please enter
        ↪ Driver ID:")
    elif self.driver_id_status == DataStatus.INVALID
    ↪ :
        self.driver_id = self.
        ↪ _simulate_driver_action("Please revalidate
        ↪ /re-enter Driver ID:")
    self.driver_id_status = DataStatus.VALID
    return 'D2'
```

Listing 1: Code snippet of the prototype software implementing phase 1 of the SoM procedure.

```
def test_path_l2_success_grant_fs(setup_test_logger
↪ ):
    script = [
        "DRIVER_007", '2', True, "session_open", "
        ↪ validate_position",
        'TD', "DATA", "TRN", "ack", "Start", "ma_fs"
    ]
    sim = ERTMSOnBoardSystem(logger=
    ↪ setup_test_logger, controller=MockController(
    ↪ script))
    sim.run_start_of_mission()
    assert sim.current_mode == Mode.FULL_SUPERVISION
```

Listing 2: Code snippet of the test to obtain full supervision.

6.2. Software monitoring design

Having developed both the software and its test suite, the instrumentation can now be performed through an LLM. Specifically, we have used the software instrumentation prompt seen in Section 5 and provided LLMs with both the prototype software and the process model in XML format. With both domain-specific knowledge and full access to the code internals, LLMs are able to interpret code structure and place logging instructions where they identify the best fit. For example, Listing 3 shows the instrumentation of one of the phase 1 procedures shown in Listing 1 by Gemini Pro 3.0, one of the LLMs we used in our experiments. The LLM has successfully identified the correct code locations where the `som_enterid_dmi`, `som_retry_dmi`, `som_storeid_evc`, and `som_validate_evc` activities had to be logged.

```
def _procedure_s1_driver_id_entry(self):
    if self.driver_id_status == DataStatus.UNKNOWN:
        self.logger.info("som_enterid_dmi")
        self.driver_id = self.
        ↪ _simulate_driver_action("Please enter
        ↪ Driver ID:")
    elif self.driver_id_status == DataStatus.INVALID
    ↪ :
        self.logger.info("som_retry_dmi")
        self.driver_id = self.
        ↪ _simulate_driver_action("Please revalidate
        ↪ /re-enter Driver ID:")

    self.logger.info("som_storeid_evc")
    self.driver_id_status = DataStatus.VALID

    self.logger.info("som_validate_evc")
    return 'D2'
```

Listing 3: Code snippet of the _procedure_s1_driver_id_entry function instrumented by Gemini Pro 3.0.

6.3. Software anomaly detection

The event log resulting from the software execution with the test suite can now be used for monitor training. This is necessary because although LLMs allow obtaining quality event logs, they may introduce some noise, such as missing or duplicated activities. While these discrepancies may indicate anomalies, they can be treated as source-code instrumentation noise. Thus, by allowing some degree of noise, the monitor can become more effective at finding true positives while reducing the amount of false positives thanks to this tolerance.

Since the only available traces are those obtained through the test suite, the event log only contains normal traces. After performing conformance checking against the reference Petri net, unsupervised, one-class algorithms can be used to learn how to characterize normal behavior from the diagnoses of Definition 4.5. In the next section, we will show the application of several unsupervised, one-class machine learning techniques to the diagnoses generated through our methodology.

7. Evaluation

The evaluation aims to substantiate the answers to the research questions with metrics that quantify source-code instrumentation quality (RQ1) and control-flow anomaly detection effectiveness (RQ2). The software for the experiments is implemented using Python and was run on a Windows 11 machine with an Intel® Core™ i9-11900K CPU @ 3.50 GHz and 32 GB of RAM. The software uses machine learning and process mining libraries, such as `scikit-learn` and `pm4py`. In addition, it uses the APIs of several LLMs to query the models with the prompts shown in Section 5. The software is available online on GitHub.[4]

7.1. RQ1: Source-code instrumentation

7.1.1. Experimental setup

LLM model set. In order to provide a diversified analysis of the source-code instrumentation quality obtained by LLMs, we are evaluating the output of different state-of-the-art models. In particular, we focus on LLMs that exhibit implicit reasoning capabilities. Unlike explicit Chain-of-Thought (CoT) approaches that rely on external triggers, implicit reasoning allows models to decompose complex problems into intermediate steps within their internal hidden states or latent representations, without requiring specific step-by-step prompting [51,52]. We have considered two state-of-the-art LLMs that have been reported to have reasoning capabilities: Gemini 3.0 Pro (gemini-3.0-pro) [53] and Claude Sonnet 4.5 (claude-sonnet-4.5) [54]. Notably, the Gemini and Sonnet families have shown remarkable performance in software engineering tasks, as reported in the SWE-bench results [55]. In addition, we have considered Gemini 3.0 Flash (gemini-3.0-flash) and Claude Haiku 4.5 (claude-haiku-4.5), which offer shallower reasoning capabilities that are expected to influence the end results. Finally, it is worth noting that these LLMs expose several parameters that influence the determinism of their outputs, including temperature, top-p, and top-k sampling [56]. In our experiments, all models were evaluated using the default inference parameters specified by their respective APIs (e.g., a temperature of 1.0). While such defaults introduce controlled non-determinism, they are designed to balance coherence and diversity, enabling exploratory generation rather than strict greedy decoding [57]. Adhering to provider-defined default settings ensures consistency across models and reflects their intended real-world usage, thereby supporting fair and comparable evaluation.

Instrumentation prompt. Each LLM is fed with a prompt to guide the software instrumentation. The prompt includes input artifacts, assigns a role to the LLM, provides the context, specifies point-to-point actions, and establishes the output type. The instrumentation prompt used for each LLM is the following.

Instrumentation prompt

Input: The SoM source code, the test suite, and the XML version of the process model.
Role: You are a software developer responsible for instrumenting an existing software system.
Context: You are instrumenting the source code of an ERTMS/ETCS-compliant prototype of the Start-of-Mission procedure. The prototype simulates the different control flows of the procedure documented in the standard with varying degrees of randomness. The prototype needs to be instrumented. You have access to the process model that guided the development process. You have access to the test suite that will be executed with pytest.
Action: Instrument the code through these guidelines:

(1) Do not modify the code logic.
(2) Instrument the code following the control flow of the process model and its activities.
(3) Before instrumenting, extract and list all activity names from the XML file.
(4) Only log the process model activities without adding any additional text.
(5) Only use activity labels that appear in this exact list — do not invent or infer any labels.
(6) Make the instrumented code callable from the test suite. Only ensure that it is callable, do not perform any further fixes.
(7) Do not provide any further output than the one specified below.

Output: The instrumented software.

The actions of the prompt guide the LLM throughout the process. In particular: action (1) ensures that the system under test's logic is not invalidated; action (2) constrains the instrumentation process to force the alignment with the process model's description; action (3) explicitly tells the LLM to extract the list of activities within the process model to have a matching between the software execution and the process model; action (4) ensures that the log does not add noise to the monitored software; action (5) shields against possible hallucinations in recording the process model activities; action (6) ensures a matching

[4] https://github.com/francescovitale/pm_software_monitor

Table 3
The fitness, number of variants, average trace length and coverage related to the source-code instrumentation performed by each LLM. The bold values indicate the best-performing LLM according to the control-flow coverage percentage.

Reasoning	Model	Fitness (%)	Coverage (%)	Trace Length	Variants
Deep	**claude-sonnet-4.5**	**86.617 ± 2.366**	**82.849 ± 2.636**	**14 ± 1**	**22 ± 1**
	gemini-3.0-pro	81.674 ± 3.060	77.579 ± 3.292	12 ± 1	23 ± 1
Shallow	gemini-3.0-flash	71.426 ± 7.335	66.823 ± 7.042	15 ± 1	25 ± 0
	claude-haiku-4.5	43.759 ± 15.410	46.391 ± 10.679	8 ± 2	20 ± 3

Kruskal–Wallis test p-values: $p_{Fitness} < 0.001$, $p_{Coverage} < 0.001$

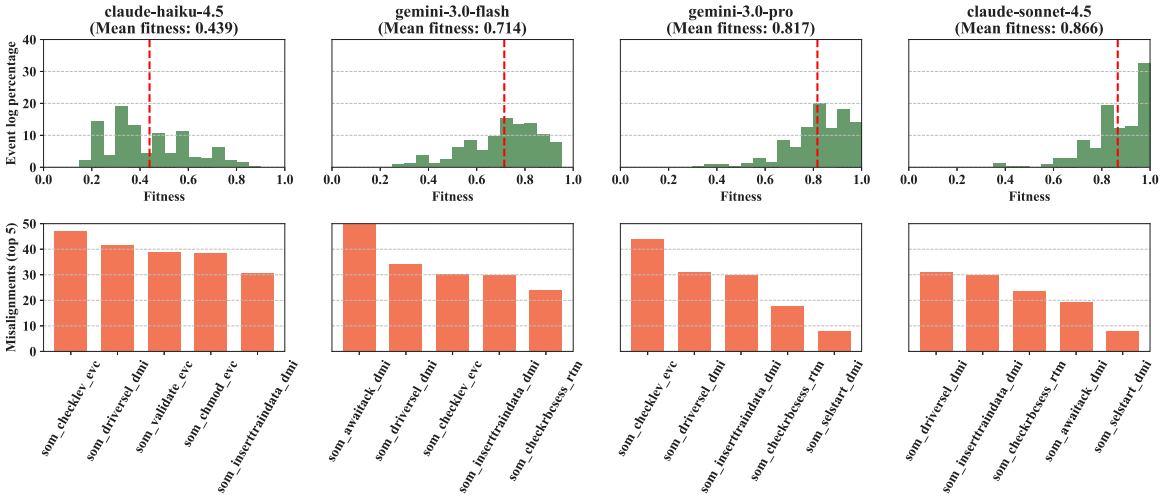


Fig. 9. The fitness value distributions of the event logs obtained for each LLM, and the corresponding top-5 per-activity misalignments.

between the test suite and the code for logging its execution without modifying neither the code nor the test suite; and action (7) avoids post-processing actions that are useless for the task at hand.

Metrics. quality of source-code instrumentation is evaluated using the following metrics. By comparing the event log obtained from the software execution using the test suite with the reference Petri net, we obtain the fitness. The higher the fitness, the higher the adherence of the control flow to the design-time description. Regarding event log-specific metrics, we consider the average trace length and the number of trace variants. Finally, we consider a key metric: the coverage. Let L and N be the event log resulting from the execution of the test suite and N the reference Petri net, we define the control-flow coverage as follows:

$$\text{Coverage} = 1 - \frac{\#misalignments}{\sum_{\sigma \in L} |\gamma^*_{\sigma,N}|}.$$

The coverage indicates how aligned the source-code instrumentation is to the Petri net.

Data generation and experiment runs. The four LLMs are independently stimulated with the instrumentation prompt for five runs each, resulting in five independent experiment executions per model. This is to account for the variability introduced by the non-zero temperature. For each run, the model context is fully reset, i.e., no conversational history or prior outputs are retained across runs. Once the instrumented version of the SoM source code is obtained, the test suite is executed and the software logs collected. Next, alignment-based conformance checking is performed for each trace of the software log, capturing both the fitness and coverage. In addition to the intra-LLM validation of the anomaly detection quality, we are including an inter-LLM validation that aims at verifying whether the quality of the instrumentation performed by different LLMs impacts anomaly detection. To this aim, we evaluate whether the conformance checking-based monitor obtained with the best-performing LLM is able to correctly identify normal and anomalous traces collected by the less-performing LLMs.

7.1.2. Results

Table 3 reports the results obtained for each LLM. The LLMs are further distinguished by their reasoning capabilities, placing them at either deep or shallow reasoning. As expected, the two best-performing LLMs with respect to the coverage metric are those with deeper reasoning, namely gemini-3.0-pro and claude-sonnet-4.5, which achieve 77.579% and 82.849% coverage while maintaining high fitness values (81.674% and 86.617%). This indicates that these models provided a high-quality instrumentation that aligns with the model (see Definition 4.4) and covers most of its control-flow paths. The two shallower LLMs, gemini-3.0-flash and claude-haiku-4.5, could not achieve the same performance, dropping fitness (71.426% and 43.759%) and coverage (66.823% and 46.391%). These differences in performance are statistically significant, as highlighted by the p-values obtained through the Kruskal–Wallis statistical test.

Fig. 9 shows the fitness values distributions (top) of the traces of the event logs linked to the LLMs. The best fitness distribution is achieved by claude-sonnet-4.5, for which more than 30.0% of traces achieve a fitness of up to ≈ 1. This can also be visualized in the bar plots below, which show the top-5 misaligned activities by counting their misalignments (see Definition 4.3). The number of misalignments for each activity increases as the fitness distribution gets lower, highlighting the correlation between the global indicator and the local diagnoses. It is worth noting that the best-performing LLMs, claude-sonnet-4.5 and gemini-3.0-pro, have similar misaligned activities, which also suggests that there may be some systematic errors in the implementation of the source code.

7.2. RQ2: Control-flow anomaly detection

7.2.1. LLM selection and cross-validation

We are going to evaluate and validate the ability of identifying control-flow anomalies with the proposed conformance checking-based monitor by exploiting the instrumentation performed by the LLMs as follows. First, we select the software logs from claude-sonnet-4.5 and

Table 4
The anomaly detection metrics associated with each technique and anomaly type using the software logs generated through claude-sonnet-4.5. The bold figures indicate the best-performing technique for the selected metric per anomaly type.

Anomaly	Technique	Accuracy (%)	Recall (%)	Precision (%)	F1-score (%)	AUC (%)
MA	FT	46.757 ± 9.767	32.400 ± 19.815	74.198 ± 7.291	41.606 ± 20.120	61.583 ± 9.605
	DBSCAN	75.135 ± 3.686	**99.600 ± 0.800**	73.376 ± 3.064	84.456 ± 1.902	80.050 ± 6.352
	AE	**88.919 ± 2.620**	92.000 ± 3.578	**91.703 ± 2.346**	**91.798 ± 2.016**	**91.358 ± 4.411**
WOA	FT	72.703 ± 8.126	70.800 ± 18.787	87.966 ± 5.432	76.365 ± 10.541	84.767 ± 2.470
	DBSCAN	83.784 ± 6.837	**98.000 ± 3.098**	82.741 ± 8.613	89.347 ± 3.976	92.717 ± 4.283
	AE	**92.162 ± 1.576**	95.200 ± 2.400	**93.408 ± 2.152**	**94.255 ± 1.150**	**95.033 ± 3.015**
UA	FT	39.865 ± 12.550	24.500 ± 24.551	50.877 ± 19.848	29.820 ± 25.549	56.208 ± 7.099
	DBSCAN	81.081 ± 4.441	94.000 ± 8.000	82.407 ± 8.245	87.120 ± 2.530	90.783 ± 5.826
	AE	**92.973 ± 2.886**	**97.200 ± 2.040**	**92.777 ± 2.413**	**94.928 ± 2.067**	**94.617 ± 2.399**
ALL	FT	45.862 ± 15.072	40.933 ± 19.811	92.437 ± 2.619	53.785 ± 18.496	66.989 ± 6.036
	DBSCAN	89.425 ± 1.568	**99.867 ± 0.267**	89.195 ± 1.537	94.222 ± 0.801	89.456 ± 3.655
	AE	**93.103 ± 1.149**	94.933 ± 0.800	**97.009 ± 0.908**	**95.957 ± 0.673**	**93.669 ± 1.610**

internally validate the anomaly detection performance of the conformance checking-based monitor. Next, we perform a cross-validation of the conformance checking-based monitor of claude-sonnet-4.5 with the software logs generated from the other LLMs.

7.2.2. Fault injection

Fault types. evaluate the capability of LLM-enabled conformance checking-based monitors, we inject three types of control-flow anomalies:

- **Missing Activities (MA)**: The control flow is corrupted by removing an activity from the trace.
- **Unknown Activities (UA)**: The control flow adds activities that were not included in the model.
- **Wrongly-Ordered Activities (WOA)**: The control flow swaps the order of activities, opposed to the model prescriptions.

These three anomalies can be linked to the software fault taxonomy of Duraes and Madeira [20,58]. Specifically, the MA anomaly type corresponds to their *missing/wrong construct* category, which the authors identified as the dominant type of software bug. For example, a missed `if` construct could skip the execution of an activity, or a wrong construct could not cover a condition in which an activity is expected to be executed. Similarly, the UA anomaly type maps to their *extraneous construct* category, representing surplus code. For example, redundant code could be wrongfully inserted by a programmer, causing the duplication of an activity. Finally, the WOA anomaly type is a clear example of their *wrong construct* category, which includes defects where the code is wrongly coded or ill-formed. For example, a statement could be misplaced within the control flow, causing an activity to be executed before its prerequisites are met, or sequential activities to be executed in the reverse order.

Injection process. injection process is as follows. Given an event log $L \in \mathcal{B}(\mathcal{A}_\alpha^*)$ generated by simulating the model in Fig. 8, we inject control-flow anomalies according to a Poissonian mechanism governed by a rate parameter $\lambda > 0$. For each trace $\sigma \in L$, the number of injected anomalies K is drawn from a Poisson distribution $K \sim \text{Poisson}(\lambda)$, so that $P(K = k) = \frac{e^{-\lambda}\lambda^k}{k!}, k = 0, 1, 2, \ldots$. The value of K determines how many modifications are applied to trace σ. Let $AT \in \{\text{MA}, \text{WOA}, \text{UA}\}$ denote the anomaly type, for which MA involves randomly deleting K activities from σ, WOA involves random K swaps between pairs of activities in σ, and UA involves randomly inserting K new activities drawn uniformly from an unknown pool of activities at random positions in σ. σ is transformed into an anomalous version $\tilde{\sigma}$ by applying the corresponding number and type of modifications: $\tilde{\sigma} = f(\sigma, AT, K)$, where $f(\cdot)$ denotes the stochastic transformation induced by the selected anomaly mechanism. In this way, the Poisson parameter λ controls the expected number of anomalies per trace, providing a flexible and statistically grounded mechanism for simulating heterogeneous anomalous behavior in event logs.

7.2.3. Experimental setup

Techniques. The selected conformance checking-based techniques are trained according to the framework in [10], where the normal event log is split into three sublogs: the training, validation and test event logs. Diagnoses are extracted for each event log. The training diagnoses are used to train two unsupervised machine learning techniques: the density-based spatial clustering of applications with noise (DBSCAN), which deals with noisy, high-dimensional and arbitrarily distributed data [59], and the autoencoder (AE), which is a neural network-based approach for dimensionality reduction that involves encoding and decoding the input data with the aim of minimizing the reconstruction error [60]. The validation diagnoses are used in two slightly different ways. For DBSCAN, a threshold is computed based on the distances of the validation data from the training clusters, whereas for AE, a threshold is built on the reconstruction error of validation data. The anomalous event logs are handled similarly and classified according to the threshold used. We also consider the baseline Fitness Thresholding (FT) technique, which is solely based on setting a threshold on the fitness metric. This is the most intuitive approach for conformance checking-based anomaly detection, although it is not as effective.

Metrics. We use the widely recognized precision, recall, and F1-score metrics commonly employed by the anomaly detection community. In particular, given negative and positive traces, the traces flagged as, respectively, normal and anomalous, the aforementioned metrics are calculated through the true positives (TP), true negatives (TN), false positives (FP) and false negatives (FN) classified by the technique:

$$\text{Accuracy} = \frac{TP + TN}{TN + TP + FP + FN},$$
$$\text{Precision} = \frac{TP}{TP + FP},$$
$$\text{Recall} = \frac{TP}{TP + FN},$$
$$\text{F1-score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}.$$

In addition, we also use the Area Under the Receiving Operating Curve (AUC). The ROC curve plots the *True Positive Rate* (TPR) against the *False Positive Rate* (FPR), defined respectively as:

$$\text{TPR} = \frac{TP}{TP + FN},$$
$$\text{FPR} = \frac{FP}{FP + TN}.$$

The AUC is then computed as the integral of the ROC curve:

$$\text{AUC} = \int_0^1 \text{TPR}(\text{FPR})\, d(\text{FPR}),$$

and can be interpreted as the probability that the model assigns a higher anomaly score to a randomly chosen anomalous trace than to a randomly chosen normal one.

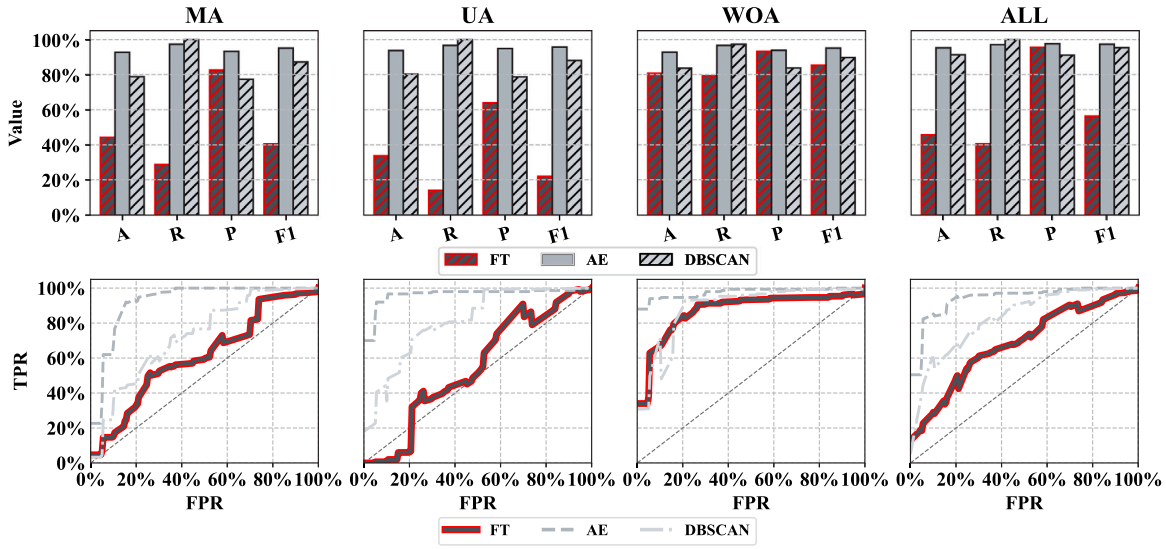


Fig. 10. Bar plots of the accuracy (A), recall (R), precision (P), and F1-score (F1), and ROC curves associated with each technique and anomaly type.

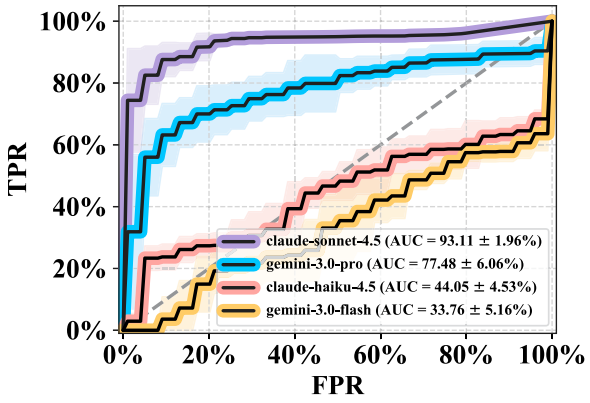


Fig. 11. ROC curves associated with the classification of normal and anomalous traces by the claude-sonnet-4.5's conformance checking–based monitor trained using the AE.

Data generation and experiment runs. The fault injection procedure is applied three times against the traces of an event log L of 50 traces generated from the model in Fig. 8. The first application of f injects MA anomalies in the traces of L, resulting in L_{MA}. Similarly, the second and third applications result in L_{WOA} and L_{UA}. In all three cases, we set $\lambda = 3$, i.e., on average, each trace has three instances of the selected anomaly. Finally, we consider $L_{ALL} = L_{MA} \cup L_{WOA} \cup L_{UA}$.

7.2.4. Results

Intra-LLM validation. Table 4 reports the results achieved for each conformance checking-based technique. The best-performing techniques are always either DBSCAN or AE, i.e., those that integrate unsupervised machine learning on top of conformance checking. They significantly outperform the baseline FT technique for the MA and UA anomalies, whereas the WOA anomaly is well discriminated by FT as well. This indicates that while fitness allows finding wrongly-ordered anomalies, it fails at identifying missing and unknown anomalies, for which an extension using other machine learning techniques is required. Considering the overall performance across all the anomalies, the AE is able to top the other two techniques, achieving up to 95.957% F1-score and 93.669% AUC. Fig. 10 visually compares the metrics through bar plots and shows the corresponding ROC curves from which AUC is computed. The bar plots especially outline the superiority of the DBSCAN and AE techniques for all the anomaly types, especially for the accuracy, recall, and F1 metrics. The ROC curves at the bottom confirm this observation, and also show that the AE is the most stable across a wide variety of reconstruction error thresholds, i.e., its AUC is higher.

Inter-LLM validation. Fig. 11 shows the ROC curves associated with the classification of normal and anomalous traces by the claude-sonnet-4.5's conformance checking-based monitor trained using the AE. The plot clearly shows that the event logs of claude-haiku-4.5 and gemini-3.0-flash lead to poor classification performance with AUC as low as 44.96% and 35.22%, respectively. On the other hand, while the classification of traces within the event log of gemini-3.0-pro is not as good as claude-sonnet-4.5, it still achieves a significantly higher AUC (79.17%). These differences can be attributed to lower-quality instrumentation that deviates from the expected control flow, inducing Petri-net misalignments and increasing the false-positive rate. Hence, the preliminary analysis performed in the context of RQ1 is necessary prior to deploying the conformance checking–based software monitor.

8. Discussion

8.1. RQ1: Source-code instrumentation

Obtaining quality software logs is pivotal to assessing the correctness of the computer-based system's execution. The main challenge in instrumenting code regards the adequate placement of log instructions and the connection of software traces to high-level models to ensure correct operation. We addressed this challenge by the systematic incorporation of LLMs in the engineering process of high-stakes systems. We have considered a set of LLMs that incorporate deep and shallow reasoning capabilities to evaluate their ability to understand both the source code and the process model of the prototype of SoM, the case-study ERTMS/ETCS scenario, and bridge the two by properly instrumenting the source code. The stimulation of these LLMs with an accurately engineered prompt aimed at preserving the code logic while instrumenting the code outlined satisfying results. In particular, the two LLMs with deeper reasoning (gemini-3.0-pro and claude-sonnet-4.5) achieved up to 82.849% and 77.579% coverage of the reference model's control flow.

8.2. RQ2: Control-flow anomaly detection

The ability to identify control-flow anomalies through the proposed conformance checking-based monitor has been evaluated with well-established anomaly detection metrics and different techniques trained

on the diagnoses linked to event logs gathered by the best-performing LLM that resulted from the experimentation of RQ1. In particular, we have used the normal traces generated by the claude-sonnet-4.5 model to carry out our intra-LLM evaluation. The approach has proven successful, as different types of control-flow anomalies that can be linked to well-known software defects could be identified by the combination of alignment-based conformance checking and the AE with 95.957% F1-score and 93.669% AUC. This finding outlines the suitability of LLM-based source-code instrumentation for subsequent conformance checking-based control-flow anomaly detection. In addition, it is worth noting that using the AE does not invalidate the explainability of the proposed approach, as feature-based explanations can be provided through post-hoc explainability tools, such as SHAP, to identify the misaligned activities that caused the highest reconstruction errors [10].

In addition, we have performed inter-LLM evaluation by comparing the ROC curves resulting from the classification of the event logs of the other LLMs, namely gemini-3.0-pro, claude-haiku-4.5, and gemini-3.0-flash. The results outlined that the source-code instrumentation quality is linked to the AUC metric, as gemini-3.0-pro achieved significantly higher AUC than the remaining two LLMs.

8.3. Threats to validity

Following common guidelines for empirical studies in software engineering [61], we discuss potential threats to construct, internal, and external validity. Construct validity concerns whether the employed evaluation metrics adequately capture the intended phenomena. Internal validity addresses the extent to which observed results can be attributed to the studied factors rather than to uncontrolled variables. External validity concerns the extent to which the findings can be generalized beyond the specific context of the study.

8.3.1. Construct validity

We used a wide variety of metrics to evaluate both the instrumentation quality of LLMs and the ability of conformance checking to leverage LLM-based instrumentation for control-flow anomaly detection. While we believe that the quality of anomaly detection was comprehensively evaluated, our coverage metric focuses exclusively on the degree to which control-flow paths of the reference process model are exercised. Hence, it is worth mentioning that there are other, more traditional ways of assessing instrumentation quality, such as measuring *code coverage* (e.g., statement, branch, or path coverage), *instrumentation overhead* (i.e., the runtime and memory impact introduced by instrumentation), and *trace completeness* (the proportion of relevant execution events successfully captured).

8.3.2. Internal validity

Our experiments revealed that different LLMs, despite being instructed with the same prompt, produced varying results in terms of fitness, control-flow coverage, average trace length, and number of variants. While these differences can be partly attributed to the intrinsic architectural and training complexities of the selected LLMs, other factors may have contributed to the observed variability. For instance, aspects such as prompt formatting to adapt to the specific strengths and weaknesses of the given LLM, context length handling, or differences in the pretraining and fine-tuning data should be further explored. Finally, LLMs exhibit non-deterministic behavior that produces different outputs for the same prompt, unless constrained decoding settings are applied (e.g., low temperature, top-1 sampling, or greedy decoding). While we addressed the variation of source-code instrumentation through repeated and independent runs, a thorough sensitivity analysis on the parameters influencing LLM determinism could shed light on variability and the resulting quality of source-code instrumentation.

Regarding the anomaly detection experiments, although we employed both clustering-based and reconstruction-based techniques on top of conformance checking diagnoses, their performance might still be affected by other experimental factors. These include the specific fine-tuning procedures applied, the choice of hyperparameter configurations, and the stochastic nature of optimization processes.

8.3.3. External validity

The transferability of our methodology to other domains is predicated on four prerequisites: (1) the availability of a design-time behavioral specification convertible into a formal model (e.g., the Petri net in Definition 4.2); (2) access to the system's source code for instrumentation; (3) a comprehensive test suite capable of establishing a reliable baseline of normal execution; and (4) an LLM with sufficient context window and reasoning capabilities to bridge the abstraction gap between the model and the implementation.

Regarding the generalizability of the ERTMS/ETCS results, broader validation is required. Future work should target real-world software implementations, specifically focusing on the detection of residual faults that are proven to escape standard verification strategies.

9. Conclusions

In this paper, we took on the problem of runtime monitoring of complex computer-based systems to detect control-flow anomalies due to unexpected changes in the system, the environment, or other "unknown unknowns". Given the limitations of runtime verification of declarative specifications in terms of strict consistency with prescriptive specifications and flexibility, we proposed a methodology to architect software monitors by combining LLMs with conformance checking to enable quality source-code instrumentation and fuzzy runtime control-flow analysis. We tested the methodology on a case-study scenario from ERTMS/ETCS, an industrial railway interoperability standard. The evaluation highlighted that the proposed methodology provides high-quality event logs through LLM-based source-code instrumentation and enables effective conformance checking-based control-flow anomaly detection.

Possibilities for industrial exploitation of the proposed approach include implementation through appropriate architectural frameworks for digital twins (see, e.g., [62,63]). Future work will further investigate the efficacy of the methodology in real-world case studies, addressing the generalization of the proposal. In addition, a deeper investigation of the factors linked to LLMs that may influence the outcome of the instrumentation quality will be performed.

CRediT authorship contribution statement

Francesco Vitale: Writing – review & editing, Writing – original draft, Visualization, Software, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Francesco Flammini:** Writing – review & editing, Validation, Supervision, Methodology, Conceptualization. **Mauro Caporuscio:** Writing – review & editing, Validation, Supervision, Methodology. **Nicola Mazzocca:** Supervision, Funding acquisition, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

The work of Francesco Vitale and Nicola Mazzocca was partly supported by the Spoke 9 “Digital Society & Smart Cities” of ICSC - Centro Nazionale di Ricerca in High Performance-Computing, Big Data and Quantum Computing, funded by the European Union - NextGenerationEU (PNRR-HPC, CUP: E63C22000980007).

The work of Francesco Flammini was partly supported by the Swiss State Secretariat for Education, Research and Innovation (SERI) under contracts no. 23.00321 (Academics4Rail) and 24.00528 (PhDs EU-Rail); those projects have been selected within the European Union’s Horizon Europe research and innovation programme under grant agreements no. 101121842 and 101175856, respectively. Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the funding agencies, which cannot be held responsible for them.

Data availability

The data has been linked in the manuscript.

References

[1] C.-H. Huang, D.A. Peled, S. Schewe, F. Wang, A game-theoretic foundation for the maximum software resilience against dense errors, IEEE Trans. Softw. Eng. 42 (2016) 605–622, http://dx.doi.org/10.1109/TSE.2015.2510001.

[2] B. Karaduman, B.T. Tezel, M. Challenger, Rational software agents with the BDI reasoning model for Cyber–Physical Systems, Eng. Appl. Artif. Intell. 123 (2023) 106478, http://dx.doi.org/10.1016/j.engappai.2023.106478.

[3] E. Viganò, O. Cornejo, F. Pastore, L.C. Briand, Data-driven mutation analysis for cyber-physical systems, IEEE Trans. Softw. Eng. 49 (2022) 2182–2201, http://dx.doi.org/10.1109/TSE.2022.3213041.

[4] A. Bombarda, S. Bonfanti, A. Gargantini, Y. Lei, F. Duan, RATE: A model-based testing approach that combines model refinement and test execution, Softw. Test. Verif. Reliab. 33 (2023) e1835, http://dx.doi.org/10.1002/stvr.1835.

[5] R. Casaluce, A. Burattin, F. Chiaromonte, A.L. Lafuente, A. Vandin, White-box validation of quantitative product lines by statistical model checking and process mining, J. Syst. Softw. 210 (2024) 111983, http://dx.doi.org/10.1016/j.jss.2024.111983.

[6] S. Pradhan, A. Dubey, T. Levendovszky, P.S. Kumar, W.A. Emfinger, D. Balasubramanian, W. Otte, G. Karsai, Achieving resilience in distributed software systems via self-reconfiguration, J. Syst. Softw. 122 (2016) 344–363, http://dx.doi.org/10.1016/j.jss.2016.05.038.

[7] J. Pan, Z. Liu, D. Li, L. Wang, B. Li, An empirical study of software architecture resilience evaluation methods, J. Syst. Softw. 202 (2023) 111726, http://dx.doi.org/10.1016/j.jss.2023.111726.

[8] M. Cinque, D. Cotroneo, R.D. Corte, A. Pecchia, Characterizing Direct Monitoring Techniques in Software Systems, IEEE Trans. Reliab. 65 (2016) 1665–1681, http://dx.doi.org/10.1109/TSE.2022.3213041.

[9] S. Gu, G. Rong, H. Zhang, H. Shen, Logging Practices in Software Engineering: A Systematic Mapping Study, IEEE Trans. Softw. Eng. 49 (2023) 902–923, http://dx.doi.org/10.1109/TSE.2022.3166924.

[10] F. Vitale, M. Pegoraro, W.M. van der Aalst, N. Mazzocca, Control-flow anomaly detection by process mining-based feature extraction and dimensionality reduction, Knowl.-Based Syst. 310 (2025) 112970, http://dx.doi.org/10.1016/j.knosys.2025.112970.

[11] M. Cinque, D. Cotroneo, A. Pecchia, Event Logs for the Analysis of Software Failures: A Rule-Based Approach, IEEE Trans. Softw. Eng. 39 (2013) 806–821, http://dx.doi.org/10.1109/TSE.2012.67.

[12] K. Kejstová, P. Ročkai, J. Barnat, From model checking to runtime verification and back, in: Runtime Verification: 17th International Conference, RV 2017, Seattle, WA, USA, September 13-16, 2017, Proceedings 17, Springer, 2017, pp. 225–240, http://dx.doi.org/10.1007/978-3-319-67531-2_14.

[13] A. Cimatti, C. Tian, S. Tonetta, NuRV: A nuXmv Extension for Runtime Verification, in: Runtime Verification, 2019, pp. 382–392, http://dx.doi.org/10.1007/978-3-030-32079-9_23.

[14] C. Czepa, U. Zdun, On the Understandability of Temporal Properties Formalized in Linear Temporal Logic, Property Specification Patterns and Event Processing Language, IEEE Trans. Softw. Eng. 46 (2020) 100–112, http://dx.doi.org/10.1109/TSE.2018.2859926.

[15] A. Cimatti, T.M. Grosen, K.G. Larsen, S. Tonetta, M. Zimmermann, Exploiting Assumptions for Effective Monitoring of Real-Time Properties Under Partial Observability, in: Software Engineering and Formal Methods, 2025, pp. 70–88, http://dx.doi.org/10.1007/978-3-031-77382-2_5.

[16] R. Jongeling, F. Ciccozzi, J. Carlson, A. Cicchetti, Consistency management in industrial continuous model-based development settings: a reality check, Softw. Syst. Model. 21 (8) (2022) 1511–1530, http://dx.doi.org/10.1007/s10270-022-01000-5.

[17] N. Weidmann, Tolerant consistency management in model-driven engineering, in: Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, 2018, pp. 192–197, http://dx.doi.org/10.1145/3270112.3275339.

[18] M.A. Tröls, A. Mashkoor, A. Egyed, Multifaceted consistency checking of collaborative engineering artifacts, in: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), IEEE, 2019, pp. 278–287, http://dx.doi.org/10.1109/MODELS-C.2019.00044.

[19] B. Sinclair-Desgagné, Chasing Unknown Unknowns with the Available Data , Int. J. Data Sci. Anal. (2025) http://dx.doi.org/10.1007/s41060-025-00796-x.

[20] R. Natella, D. Cotroneo, J.A. Duraes, H.S. Madeira, On Fault Representativeness of Software Fault Injection, IEEE Trans. Softw. Eng. 39 (1) (2013) 80–96, http://dx.doi.org/10.1109/TSE.2011.124.

[21] G. Tetlie, G. Muller, S. Kokkula, Literature Review and Research Design for Systems Integration: Case Study in Defense Systems, in: The Proceedings of the 2023 Conference on Systems Engineering Research, Springer Nature Switzerland, Cham, 2024, pp. 691–705, http://dx.doi.org/10.1007/978-3-031-49179-5_47.

[22] L.E. Prokop, Historical Aerospace Software Errors Categorized to Influence Fault Tolerance, in: 2024 IEEE Aerospace Conference, 2024, pp. 1–12, http://dx.doi.org/10.1109/AERO58975.2024.10521061.

[23] P. Singh, N.R. Pal, S. Verma, O.P. Vyas, Fuzzy Rule-Based Approach for Software Fault Prediction, IEEE Trans. Syst. Man, Cybern.: Syst. 47 (2017) 826–837, http://dx.doi.org/10.1109/TSMC.2016.2521840.

[24] M. Du, F. Li, G. Zheng, V. Srikumar, DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning, in: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, 2017, pp. 1285–1298, http://dx.doi.org/10.1145/3133956.3134015.

[25] G. Denaro, R. Heydarov, A. Mohebbi, M. Pezzè, Prevent: An Unsupervised Approach to Predict Software Failures in Production, IEEE Trans. Softw. Eng. 49 (2023) 5139–5153, http://dx.doi.org/10.1109/TSE.2023.3327583.

[26] X. Han, S. Yuan, M. Trabelsi, LogGPT: Log Anomaly Detection via GPT, in: 2023 IEEE International Conference on Big Data (BigData), 2023, pp. 1117–1122, http://dx.doi.org/10.1109/BigData59044.2023.10386543.

[27] W. Guan, J. Cao, H. Zhao, Y. Gu, S. Qian, WAKE: A Weakly Supervised Business Process Anomaly Detection Framework via a Pre-Trained Autoencoder, IEEE Trans. Knowl. Data Eng. 36 (2024) 2745–2758, http://dx.doi.org/10.1109/TKDE.2023.3322411.

[28] Z. Zhang, S. Li, L. Zhang, J. Ye, C. Hu, L. Yan, LLM-LADE: Large language model-based log anomaly detection with explanation, Knowl.-Based Syst. 326 (2025) 114064, http://dx.doi.org/10.1016/j.knosys.2025.114064.

[29] A. Rawal, J. McCoy, D.B. Rawat, B.M. Sadler, R.S. Amant, Recent Advances in Trustworthy Explainable Artificial Intelligence: Status, Challenges, and Perspectives, IEEE Trans. Artif. Intell. 3 (2022) 852–866, http://dx.doi.org/10.1109/TAI.2021.3133846.

[30] J. Havare, S. Chaudhary, G. Ramakrishnan, K. Maharajan, S. Tamilselvam, A Code Comprehension Benchmark for Large Language Models for Code, 2025, arXiv:2507.10641.

[31] N. Martin, D.A. Fischer, G.D. Kerpedzhiev, K. Goel, S.J. Leemans, M. Röglinger, W.M. van der Aalst, M. Dumas, M. La Rosa, M.T. Wynn, Opportunities and challenges for process mining in organizations: results of a Delphi study, Bus. Inf. Syst. Eng. 63 (2021) 511–527, http://dx.doi.org/10.1007/s12599-021-00720-0.

[32] L. Zimmermann, F. Zerbato, B. Weber, What makes life for process mining analysts difficult? A reflection of challenges, Softw. Syst. Model. (2023) 1–29, http://dx.doi.org/10.1007/s10270-023-01134-0.

[33] A. Mamudu, W. Bandara, M.T. Wynn, S.J. Leemans, Process Mining Success Factors and Their Interrelationships, Bus. Inf. Syst. Eng. (2024) 1–20, http://dx.doi.org/10.1007/s12599-024-00860-z.

[34] F. Laroche, L. Guihéry, European Rail Traffic Management System (ERTMS): Supporting competition on the European rail network? Res. Transp. Bus. Manag. 6 (2013) 81–87, http://dx.doi.org/10.1016/j.rtbm.2012.12.006.

[35] European Union Agency for Railways, Report on Railway Safety and Interoperability in the EU 2024, Technical Report, European Union Agency for Railways (ERA), 2024, URL https://www.era.europa.eu/system/files/2024-06/Report%20on%20Railway%20Safety%20and%20Interoperability%20in%20the%20EU%202024.pdf. (Accessed on 24 March 2026).

[36] M.H.t. Beek, A. Fantechi, S. Gnesi, Product line models of large cyber-physical systems: the case of ERTMS/ETCS, in: Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1, 2018, pp. 208–214, http://dx.doi.org/10.1145/3233027.3233046.

[37] L. Briand, Y. Labiche, J. Leduc, Toward the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software, IEEE Trans. Softw. Eng. 32 (2006) 642–663, http://dx.doi.org/10.1109/TSE.2006.96.

[38] T. Zhang, X. Zheng, Y. Zhang, J. Zhao, X. Li, A declarative approach for Java code instrumentation, Softw. Qual. J. 23 (2015) 143–170, http://dx.doi.org/10.1007/s11219-013-9220-4.

[39] Z. Jia, S. Li, X. Liu, X. Liao, Y. Liu, SMARTLOG: Place error log statement by deep understanding of log intention, in: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering, SANER, IEEE, 2018, pp. 61–71, http://dx.doi.org/10.1109/SANER.2018.8330197.

[40] M. Cinque, R. Della Corte, A. Pecchia, Discovering Hidden Errors from Application Log Traces with Process Mining, in: 2019 15th European Dependable Computing Conference, EDCC, 2019, pp. 137–140, http://dx.doi.org/10.1109/EDCC.2019.00034.

[41] A. Pecchia, I. Weber, M. Cinque, Y. Ma, Discovering process models for the analysis of application failures under uncertainty of event logs, Knowl.-Based Syst. 189 (2020) 105054, http://dx.doi.org/10.1016/j.knosys.2019.105054.

[42] A. Engelke, M. Schulz, Instrew: leveraging LLVM for high performance dynamic binary instrumentation, in: Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, 2020, pp. 172–184, http://dx.doi.org/10.1145/3381052.3381319.

[43] A. Mastropaolo, V. Ferrari, L. Pascarella, G. Bavota, Log statements generation via deep learning: Widening the support provided to developers, J. Syst. Softw. 210 (2024) 111947, http://dx.doi.org/10.1016/j.jss.2023.111947.

[44] D.-S. Wang, T.-I. Kim, C.-G. Lee, H.-T. Hong, Automated Logging With Large Language Models: Eliminating Task-Specific Pretraining, IEEE Access 13 (2025) 213260–213272, http://dx.doi.org/10.1109/ACCESS.2025.3645237.

[45] W. Mu, N. Li, Z. Ji, J. Guo, B. Huang, FlexInstru: A flexible instrumentation framework for tracing long-running native workloads, J. Syst. Softw. 234 (2026) 112739, http://dx.doi.org/10.1016/j.jss.2025.112739.

[46] W.M.P. van der Aalst, J. Carmona, Process Mining Handbook, Springer, Cham, Switzerland, 2022, http://dx.doi.org/10.1007/978-3-031-08848-3.

[47] T. Zoppi, A. Ceccarelli, T. Capecchi, A. Bondavalli, Unsupervised Anomaly Detectors to Detect Intrusions in the Current Threat Landscape, ACM/IMS Trans. Data Sci. 2 (2021) http://dx.doi.org/10.1145/3441140.

[48] D. Schuster, G.J. Kolhof, Scalable online conformance checking using incremental prefix-alignment computation, in: International Conference on Service-Oriented Computing, Springer, 2020, pp. 379–394.

[49] ISO/IEC/IEEE, Software and Systems Engineering – Software Testing, International Organization for Standardization, Geneva, Switzerland, 2022, Standard ISO/IEC/IEEE 29119-1:2022. URL https://www.iso.org/standard/81291.html. (Accessed on 24 March 2026).

[50] L. Carnevali, S. Cerboni, L. Montecchi, E. Vicario, FaultFlow: An MDE Library for Dependability Evaluation of Component-Based Systems, IEEE Trans. Dependable Secur. Comput. 22 (4) (2025) 3431–3448, http://dx.doi.org/10.1109/TDSC.2025.3532340.

[51] Y. Deng, K. Prasad, R. Fernandez, P. Smolensky, V. Chaudhary, S. Shieber, Implicit Chain of Thought Reasoning via Knowledge Distillation, 2023, arXiv:2311.01460.

[52] X. Wang, D. Zhou, Chain-of-Thought Reasoning Without Prompting, 2024, arXiv:2402.10200.

[53] Google DeepMind, Gemini 3 Pro Model Card, Google Deep Mind Technical Reports, 2025, URL https://storage.googleapis.com/deepmind-media/Model-Cards/Gemini-3-Pro-Model-Card.pdf. (Accessed on 24 March 2026).

[54] Anthropic, Claude Sonnet 4.5 System Card, 2025, System documentation for the Sonnet 4.5 model.https://www.anthropic.com/claude/sonnet. (Accessed on 24 March 2026).

[55] C.E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, K.R. Narasimhan, SWE-bench: Can Language Models Resolve Real-world Github Issues? in: International Conference on Learning Representations, 2024, URL https://www.swebench.com/.

[56] L. Li, L. Sleem, N. Gentile, G. Nichil, R. State, Exploring the Impact of Temperature on Large Language Models: Hot or Cold? Procedia Comput. Sci. 264 (2025) 242–251, http://dx.doi.org/10.1016/j.procs.2025.07.135, International Neural Network Society Workshop on Deep Learning Innovations and Applications 2025.

[57] M. Renze, The Effect of Sampling Temperature on Problem Solving in Large Language Models, in: Y. Al-Onaizan, M. Bansal, Y.-N. Chen (Eds.), Findings of the Association for Computational Linguistics: EMNLP 2024, 2024, pp. 7346–7356, http://dx.doi.org/10.18653/v1/2024.findings-emnlp.432.

[58] J.A. Durães, H.S. Madeira, Emulation of Software Faults: A Field Data Study and a Practical Approach, IEEE Trans. Softw. Eng. 32 (11) (2006) 849–867, http://dx.doi.org/10.1109/TSE.2006.113.

[59] E. Schubert, J. Sander, M. Ester, H.P. Kriegel, X. Xu, DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN, ACM Trans. Database Syst. 42 (2017) http://dx.doi.org/10.1145/3068335.

[60] M. Sakurada, T. Yairi, Anomaly Detection Using Autoencoders with Nonlinear Dimensionality Reduction, in: Proceedings of the MLSDA 2014 2nd Workshop on Machine Learning for Sensory Data Analysis, MLSDA '14, Association for Computing Machinery, New York, NY, USA, 2014, pp. 4–11, http://dx.doi.org/10.1145/2689746.2689747.

[61] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in Software Engineering, Springer, Berlin, Heidelberg, 2012, http://dx.doi.org/10.1007/978-3-642-29044-2.

[62] F. Flammini, Digital twins as run-time predictive models for the resilience of cyber-physical systems: a conceptual framework, Phil. Trans. R. Soc. A 379 (2207) (2021) 20200369, http://dx.doi.org/10.1098/rsta.2020.0369.

[63] A. De Benedictis, F. Flammini, N. Mazzocca, A. Somma, F. Vitale, Digital Twins for Anomaly Detection in the Industrial Internet of Things: Conceptual Architecture and Proof-of-Concept, IEEE Trans. Ind. Inform. 19 (12) (2023) 11553–11563, http://dx.doi.org/10.1109/TII.2023.3246983.