Автоматизовані системи управління та прилади автоматики. 2025. № 4 (187)

UDC 004.416 **DOI:** https://doi.org/10.30837/0135-1710.2025.187.254

A. Chupryna, V. Repikhov

REFERENCE MODEL FOR PREVENTIVE SOFTWARE MAINTENANCE

Subject of the study. The study focuses on preventive software maintenance processes, particularly the formalization of mechanisms for monitoring, degradation-risk identification, and proactive decision-making within the software life cycle. **The purpose** of the research is to develop a reference model for preventive maintenance that enables systematic detection of deviations from reference operating modes, assessment of their temporal dynamics, and generation of well-grounded preventive actions prior to the emergence of failures. **Research tasks.** The tasks include: establishing a classification of metrics and features for characterizing software states; constructing a formal apparatus for preventive identification using an extended comparator identification method; defining the structural components of the model; and developing models for monitoring, risk analysis, maintenance-task derivation, and adaptive parameter adjustment. **Research methods.** The methodological basis comprises the extended comparator identification method, which provides comparative evaluation of current software states relative to reference modes and quantifies deviation trajectories. Feature aggregation, normalization, formal description of the functional state space, trend analysis in temporal windows, and adaptive feedback mechanisms are applied. **Results achieved.** A five-component reference model is developed, incorporating modules for state monitoring, preventive identification, task generation, action-effect evaluation, and adaptive tuning. The model supports early detection of latent degradation through analysis of deviation trajectories, integrates with CI/CD, DevOps and MLOps pipelines, and establishes a formalized decision-support mechanism for proactive maintenance. **Conclusions.** The proposed model constitutes a comprehensive formal framework for preventive software maintenance, aimed at early detection of potential failures and long-term system stability. Its adaptive nature ensures relevance under dynamic operational conditions, while comparator identification enhances interpretability and validity of maintenance decisions. The results provide a foundation for intelligent maintenance systems and future research on automated formation of reference states and ML-enhanced risk assessment.

Keywords: preventive maintenance; comparator identification; monitoring; technical debt; degradation prediction; software quality; DevOps; MLOps.

Introduction

Software (SW) requires continuous maintenance throughout its entire life cycle. The support and maintenance phase is the longest and most resource-intensive: according to estimates, it can account for up to 80–90 % of the total cost of a software product's life cycle [1]. Maintenance covers various activities, from defect correction (corrective maintenance) and adaptation to environmental changes (adaptive maintenance) to functionality expansion (perfective maintenance). Preventive maintenance is a separate category – proactive actions aimed at anticipating potential problems before they arise. According to the international standard ISO/IEC/IEEE 14764:2022 [2], preventive maintenance is defined as "modification of a software product after its delivery, aimed at correcting hidden defects before they manifest themselves in the operating system" [2]. This proactive approach is especially important for critical software systems that have high requirements for security and availability (e.g., medical systems) [3].

The relevance of developing preventive maintenance is driven by the need to improve the reliability and quality of software products amid their growing complexity and the multitude of factors affecting the effectiveness of maintenance [4]. In traditional practice, support is often provided reactively, i.e., only after failures or user complaints have occurred. Neglecting proactive measures leads to the accumulation of technical debt and increased long-term support costs [5, 6]. Preventive actions, on the other hand, make it possible to avoid costly emergency interventions in the future. As noted in [1], the cost of emergency fixes usually exceeds the resources required to prevent them. Therefore, the transition from responding to problems to proactively preventing them is a promising direction for the development of software engineering, designed to ensure high reliability and cost-effectiveness of software maintenance.

At the global level, it is emphasized that preventive SW maintenance should be carried out in a planned and systematic manner. To implement this approach within the life cycle, it is advisable to apply the reference model for preventive support (RMPS) – a specialized framework that integrates into the process of software development and operation. The reference model provides for continuous monitoring of the state of the software system and its components, evaluation of the metrics and indicators obtained, and adaptation of the support system based on this data. Based on established criteria, the model generates recommendations for preventive actions: it determines what needs to be improved or corrected and when it should be forwarded to the development team for implementation. This ensures that the necessary maintenance work is performed in a timely manner before potential problems turn into actual failures or incidents for users. The proposed preventive maintenance framework serves as a decision-making support tool for project management, allowing you to proactively manage the evolution of the software product and maintain its high quality.

Analysis of current publications on the issue of preventive maintenance

Currently, preventive maintenance is not yet widespread in the typical software development cycle. Many organizations still focus primarily on responding to already identified defects and user requests, while systematic prevention of potential problems is implemented in a fragmented manner. At the same time, scientific literature and standards note the significant potential of the preventive approach. Methods for predicting defects and "fatigue" in software code are being developed using data analysis and machine learning, and code "smells" and ways to reduce technical debt are being investigated. The prospects for implementing software maintenance models are linked to increasing the reliability and security of software products, reducing downtime and the number of critical failures, and optimizing support costs through the timely implementation of necessary updates.

International standards for the life cycle and quality of software systematically describe maintenance [2, 7]. The ISO/IEC/IEEE 14764 standard [2] defines four types of maintenance: corrective, adaptive, perfective, and preventive. Preventive maintenance is aimed at improving reliability, security, and facilitating further development and support of the system. The ISO/IEC/IEEE 14764 [2] standard defines maintenance as an integral part of the software life cycle and describes processes that can be applied to different classes of software products. ISO/IEC 25010 [7] sets out a model of SW quality and identifies eight characteristics, including reliability, security, and maintainability. Preventive maintenance directly enhances these characteristics: it increases reliability by eliminating latent defects, improves maintainability (refactoring, updating documentation, test base), and strengthens security by promptly fixing vulnerabilities and keeping dependencies up to date. Thus, according to international standards, preventive maintenance is a formalized, planned activity that must be performed throughout the entire life cycle of a software product.

Despite the official recognition of its importance in the relevant standards, preventive maintenance receives significantly less attention in scientific literature compared to other forms of SW maintenance. An analysis of the literature over the past five years shows that preventive maintenance remains an under-researched area. For example, in a systematic review [6], only 7 of the 111 papers considered mentioned the benefits associated with preventive maintenance. Synthesis reviews (in particular [1]) emphasize that although support can account for up to 70–90 % of the SW life cycle, preventive actions remain secondary compared to defect correction or new feature implementation. A reactive support approach prevails over proactive intervention. This creates an obvious gap between the theoretical significance of preventive maintenance and its practical implementation.

Despite the general lack of research on the topic, some publications offer tools, models, or metrics directly related to preventive maintenance. Works [8–10] explored the possibility of predicting code changes (change-proneness) using software product metrics, commit history, and other factors. A vector of features is formed for each module from the history of changes: product metrics (complexity, size, "smells"), process metrics (code churn, edit frequency, defect commit density, age of last refactoring), social metrics (expertise and "ownership" of the module, number of authors, review duration). These features are used to train models that return the probability that a component will need to be changed or will produce a defect in the near future [8–10]. Empirical studies show the effectiveness of this approach for ranking risky components and planning preventive actions. Such predictive models allow identifying "vulnerable" areas of code for targeted refactoring or additional testing, which directly corresponds to the tasks of preventive support. Study [11] examines the use of machine learning algorithms to identify classes in which the presence or absence of "code smells" correlates with the probability of defects occurring. The results confirmed that removing such symptoms before real errors occur is an effective preventive approach. A number of studies (in particular [5, 12, 13]) substantiate the idea of systematic detection, registration, and repayment of technical debt through planned refactoring, library updates, etc.

Mobile applications create additional challenges for preventive support: frequent operating system updates, device diversity, high sensitivity to productivity and resources, the need for regular updates of dependencies and compliance with app store requirements. Works [14–16] emphasize the need for systematic planning of preventive releases, testing on OS beta versions, and monitoring user feedback as a source of signals for proactive improvements. Some authors suggest creating a maintenance calendar that takes into account the platform update cycle and user expectations [14].

Research on modern CI/CD and DevOps processes demonstrates the growing role of automated quality control tools that can signal the need for preventive intervention.

The maintenance of software systems with machine learning components is of particular scientific interest. In the context of ML systems, there is a similar increase in attention to monitoring data and model drift as triggers for proactive updates [3, 14, 15]. The first systematic review of the challenges of maintaining ML systems was published only in 2022 [16], which indicates the novelty and insufficient study of the topic. The main challenges are related to model degradation, the complexity of reproducing results, the lack of modularity, and the accumulation of technical debt in the form of uncleaned pipelines and unstructured data [17]. The authors [15] recommend the introduction of MLOps practices: versioning of models and data, continuous monitoring of model productivity, automated retraining, etc. Preventive maintenance in this context involves not only code maintenance, but also the quality and relevance of models, which quickly lose their effectiveness without proper support.

The analysis showed that preventive maintenance is recognized as an important but still under-researched topic in the scientific community. Existing approaches demonstrate the practical value of preventive actions but require further development, standardization, and validation. Thus, further study, formalization, and implementation of preventive maintenance models is a relevant and strategically expedient direction for software engineering.

Research objectives and tasks

The aim of this study is to improve the effectiveness of software maintenance by developing a reference model for preventive maintenance that provides early detection of degradation trends, formalized identification of risk states, and timely generation of preventive actions within the software system lifecycle.

The study has formed a comprehensive structure of a reference model of preventive maintenance, which includes components of monitoring, preventive identification, task definition, result evaluation, and adaptation. Mathematical models of the interaction of these components and mechanisms for linking them into a single cycle of proactive maintenance have also been determined. The practical applicability of the proposed model is separately substantiated, which creates the prerequisites for improving the reliability, security, and maintainability of software systems in dynamic operating conditions.

Materials and methods

The methodological basis for preventive software maintenance requires a clear understanding of the indicators that best reflect the potential risks of system degradation, as well as the mechanisms for interpreting them in order to make timely engineering decisions. This task is also complicated by the high dynamics of environmental changes (OS updates, device diversity, use of third-party service APIs), as well as pressure from user expectations for stability, performance, and data security. Preventive maintenance in such an environment should be based not only on fixing current defects, but also on analyzing changes in the behavioral, technical, and qualitative characteristics of the software system over time, as well as pressure from user expectations for stability, performance, and data security [4].

Preventive maintenance covers key groups of metrics that signal the need for intervention even before noticeable failures in SW operation occur. This study proposes the use of the comparator identification method [18–20], which allows early signs of degradation to be detected through comparative analysis of metrics over controlled periods. This method is at the core of the analytical mechanism of the preventive maintenance framework, ensuring the systematization of risk signals, the formulation of decisions for the development team, and their integration into the existing SW life cycle.

Within the scope of this work, metrics are grouped into four complementary classes: 1) code and architectural metrics; 2) maintainability and technical debt metrics; 3) process evolution metrics (changes and defects); 4) field (operational) and security metrics. This classification is consistent with quality standards approaches [7], where maintainability, reliability, and security are considered key characteristics that must be maintained throughout the life cycle.

Code and architectural metrics assess structural complexity and design features that directly affect the risk of defects and the cost of changes. A classic measure is McCabe's structural complexity, which reflects the number of linearly independent paths in a flow control graph [10]:

$$CC = E - N + 2P,$$

where E is the number of edges, N is the number of nodes, P is the number of connectivity components (mostly $P = 1$) [16]. High values CC correlate with complicated verification and increased defectiveness. Typical thresholds that are appropriate to use can be defined as follows: 1–10 (low risk), 11–20 (medium), 21–50 (high), >50 (very high).

Additional indicators, such as Coupling Between Objects (CBO), Response for a Class (RFC), and Lack of Cohesion of Methods (LCOM), provide insight into the modularity, encapsulation, and "fragility" of the design [10, 11]. Low cohesion and excessive coupling are early signals for preventive refactoring. Practical collection of such metrics can be performed by static analyzers (e.g., SonarQube), which also identify "code smells", i.e., informal design symptoms (long methods, "divine classes", duplication) that are empirically associated with defectiveness and reduced maintainability. Eliminating such defects is a typical preventive measure [11].

Maintainability and technical debt metrics aggregate assessments from different complexity measurements into a single indicator that is convenient to observe over time [6, 11, 12]. A common example of this type of metric is the Maintainability Index (MI), a convenient integral indicator that combines Halstead Volume (information complexity), Cyclomatic Complexity (structural complexity), and LOC (code size) and provides a quick assessment of maintainability [11]. The following formula for determining MI is considered the most well-known:

$$MI = max\left(0{,}171 - 5{,}2\ln HV - 0{,}23 \times CC - 16{,}2\ln LOC\right) \times \frac{100}{171},$$

where MI is normalized in the range [0,100]. Gradations may differ between instruments, but often $MI < 49$ indicates the presence of problems and the need for code refactoring. The metric works well as a trend indicator: if it falls from release to release, this is a clear signal for preventive action (refactoring, simplification, decomposition).

Process evolution metrics reflect how the system changes and is released (activity in the repository, commit patterns, defect and support history), which is a source of early signals.

Such indicators include Code Churn, as well as the frequency of changes to individual modules, the density of "hot" commits, the ratio of defect and feature changes, etc. Churn is a key process metric that shows how often and how much a module changes over time. It naturally fits into the preventive loop as an indicator of fragility: increased churn correlates with a higher risk of defects, regressions, and subsequent maintenance costs, so it serves as a signal for proactive interventions (refactoring, additional tests, stabilization sprints) [9].

The relative intensity of code changes is calculated as follows:

$$ChurnRate = \frac{LOC_{added} + LOC_{modified} + LOC_{deleted}}{LOC_{total}}.$$

At the operational level, classic reliability metrics such as MTTF/MTBF (mean time between failures) and MTTR (mean time to repair) record the consequences of identified problems, and trends in these metrics show that preventive practices are working [19].

Field (or operational) and security metrics measure the "health" of the system in real-world use and its resistance to vulnerabilities. The former include the proportion of error-free sessions relative to the number of users, latency and throughput of critical transactions, stability of resource usage (CPU/memory), productivity degradation indicators under load, etc. They are collected by APM (Application Performance Monitoring) and observability tools (Sentry, Grafana) and form the basis for early detection of deterioration trends [3, 19].

The security dimension includes the number and severity of known vulnerabilities in dependencies on the CVSS (Common Vulnerability Scoring System) scale, the "age" of unpatched CVEs (Common Vulnerabilities and Exposures), and the proportion of components outside the support window, which makes it possible to prevent degradation and exploitation [15].

From a preventive perspective, it is important not only to track individual indicators, but also to interpret their dynamics and combinations. An increase in cyclomatic complexity, along with an increase in churn and a decrease in MI, supported by "code smells", makes a compelling case for targeted refactoring before field incidents occur.

Similarly, spikes in latency and deterioration in reliability metrics, combined with an increase in defective commits, are grounds for preventive fixes and expanded test coverage. In practice, metric collection is integrated into CI/CD as "quality gates", and their trend analysis directly feeds into the backlog of preventive tasks.

It is assumed that the results of observations can be characterized by an assessment of the system's state. Based on the software system's operating data, a system of features for each scenario and a set of output characteristics are formed, which can be used to compare different behavioral scenarios. In general, the input data can be of any type, and the model output is accepted as binary (0 or 1) – in fact, the model implements a certain predicate that checks the condition for the fulfillment of relations between input signals [19].

Such formalization through predicate logic ensures strict fulfillment of the specified conditions for all cases, making the method objective and rigorous. If necessary, the method is extended to fuzzy data: instead of a rigid true/false, degrees of membership are allowed, which makes it possible to take into account the uncertainty of the output information [20].

Thanks to this, comparator identification has been applied not only to technical systems, but also to complex, weakly formalized objects – from the assessment of professional skills to the modeling of cognitive processes [18–20].

From a mathematical point of view, the comparator identification method has a rigorous theoretical basis. It provides an objective, highly accurate, and reliable description of the object model, which is not inferior in completeness to classical direct identification [19]. If the input and output relations are specified correctly, the solution to the comparator identification problem provides a model that is fully consistent with all observed facts (up to isomorphism in the model state space).

This means that within the limits of its formulation, the method is correct and capable of providing a complete mathematical description of the desired dependence. Moreover, its use formalizes subjective expert data: instead of intuitive "vague" features, a clearly defined function or rule is used for decision making.

Thus, based on the collection and analysis of features of software system operation scenarios, a system of input signals for a preventive maintenance model is formed. The use of the comparator identification method allows us to build strict and reasonable interdependencies between the received signals and the risks of system degradation through indirect identification. Based on the conclusions, SW maintenance tasks are formed.

Therefore, to implement continuous preventive SW maintenance, it is necessary to develop a framework (reference model) that will formalize the main components of the maintenance model and serve as a foundation for developing models for monitoring, identifying degradation risks, and defining maintenance tasks.

Results

The basis of the proposed preventive support model is the basic structure of software (SW) support, which includes four basic elements that correspond to the generally accepted cycle of proactive reliability assurance: monitoring, status assessment, action determination, and evaluation of the results of the actions taken.

This study proposes to extend the classical approach by adding a model tuning component that will ensure the adaptation of rules in a continuous cycle of observation and improvement of the software system for the purpose of preventive maintenance. Together, these components form a cyclical process of preventive support, similar to a monitoring and adaptation loop. It includes observation, analysis, action plan, and verification – all the elements necessary for continuous improvement of SW quality and reliability.

This approach fits in with the current trend of transition from passive support to active, intelligent support of software systems.

Components of the reference model of preventive support

The proposed reference model of preventive support thus consists of five main components: monitoring of status and usage scenarios, identification of degradation risks, determination of actions and formation of support tasks, evaluation of the results of actions, and adaptation (Fig. 1).

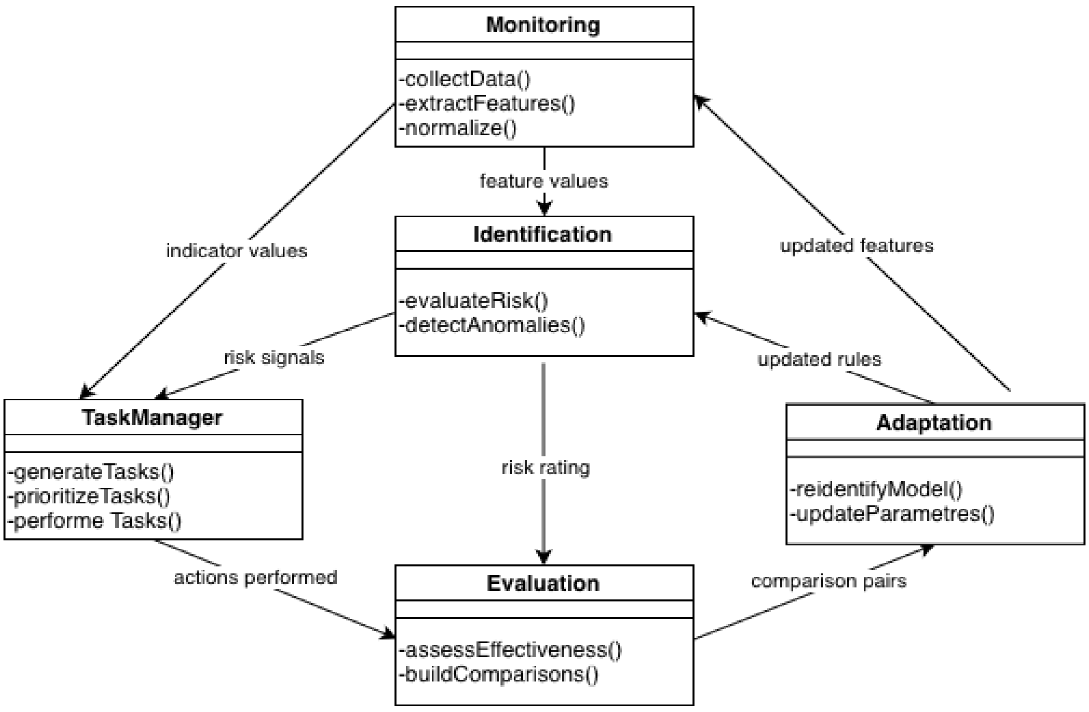


Fig. 1. Structure of the reference model for preventive SW maintenance

The monitoring process model is designed for continuous collection and tracking of metrics on the status of the application itself and the external conditions of its operation. For a mobile application with external sensors, such metrics may include, for example, application response delays, error or failure rates, abnormal sensor readings, the degree of deviation of input data from the training sample, device load indicators (CPU, memory, battery), etc. Regular monitoring allows you to accumulate historical data and record deviations in system behavior in real time.

The problem condition identification process model (preventive assessment) analyzes the collected data (using rules, statistics, or machine learning algorithms) to identify patterns and conditions that precede the occurrence of problem situations. In other words, the system attempts to predict potential failures or degradation based on detected anomalies, such as increased response time, memory leaks, sensor data distribution shifts, or decreased ML model prediction reliability. Such predictive analytics in SW is analogous to predictive maintenance in industry, where equipment failures are predicted based on data. Recent studies confirm the effectiveness of moving from purely reactive error correction to proactive prediction and prevention of failures using data analysis and AI methods.

The task identification process model is responsible for determining the actions that the development team needs to take. After identifying risky conditions, the system must determine what preventive measures or tasks should be performed to reduce the risk of a problem. In the context of agile development, this means forming specific tasks for the team (for example, including a fix for a potential memory leak in the next sprint, updating a machine learning model, improving sensor data processing, or adding monitoring of a specific parameter). In essence, this component carries out impact planning: based on the identified problem, the optimal actions are selected. In modern DevOps/AIOps practice, such decisions can be made automatically in the form of alerts to developers or even automatic remediation (e.g., service restart, model rollback). However, the automated conversion of analytical conclusions into development tasks is still in its infancy and often requires human involvement. Nevertheless, a formalized task model is important so that preventive measures do not remain at the level of signals but are implemented.

The process model for evaluating the results of actions is a feedback mechanism that assesses the effectiveness of the preventive measures taken. After completing the tasks (for example, releasing an update or reconfiguring the model), it is necessary to monitor the system again and compare the metrics before and after the intervention. This allows you to understand whether the probability of the predicted problem has decreased or whether the bottleneck has been corrected. In addition, such an analysis of results replenishes the knowledge base: it confirms or refutes the correctness of the identified conditions and the effectiveness of the measures. As a result, the results assessment model closes the classic cycle: successful cases strengthen confidence in the support system, and in case of failure, the monitoring metrics or identification rules are adjusted.

The adaptation process model is the final element of the preventive software maintenance cycle, which ensures its ability to self-improve based on feedback. Its purpose is to periodically review, refine, and update the rules, thresholds, machine learning models, and procedures used by the monitoring, preventive assessment, and task formation blocks. In particular, if a discrepancy is found between the predicted risks and the actual results of interventions, the adaptation module compares the expected and actual dynamics of metrics, assesses the degree of deviation, and determines which model parameters need to be updated. This allows for automatic adjustment of both the sensitivity of anomaly detection algorithms and the relevance of rules that initiate preventive actions.

The presence of an adaptive component is critical because the operating conditions of the mobile application, the nature of data from external sensors, and user behavior change over time, leading to the gradual degradation of static models. Thus, the adaptation block is a mechanism for "self-calibration" of the system: it closes the cycle, ensuring continuous improvement of preventive procedures and increasing the stability and reliability of the software in dynamic operating conditions.

Thus, the proposed five-component model forms a complete closed cycle of preventive support from monitoring and risk assessment to the formation of actions, evaluation of their effectiveness, and adaptive refinement of rules. This architecture ensures the software's ability to proactively self-improve, increasing its reliability in dynamic operating conditions.

Formalization of the set of software system states

Let $x \in \mathrm{R}^n$ is a is a vector of state characteristics of a software system running on a mobile device. Each component of the vector x describes a specific aspect of the system's operation, in particular, related to sensor data processing, machine learning model operation, user activity, network connection quality, device parameters, and inter-user interaction, etc. Let us denote the state of the software system at a given moment in time t as x_t. Thus, each vector $x_t \in \mathrm{R}^n$ fully characterizes the functional state of the software system over time. The vectors collected at different points in time form a set of observable states.

Let $X \subseteq \mathrm{R}^n$ – the space of all possible vectors x_t describing the software system over time. The set X is the basis for further analysis, including clustering of states, anomaly detection, formation of classes of equivalent operating modes, and construction of a system of preventive signals. Let us define the set of possible (attainable) functional states of the system as:

$$X^{\Omega} := \left\{ x_t \in X \middle| x_t \in \Omega \right\},$$

where $\Omega \subseteq \mathrm{R}$ – permissible interval or scale of values of the corresponding feature.

Therefore, vectors x_t are elements of observation streams:

$$D = \left\{ x_t \in X^{\Omega} \middle| t \in T \right\},$$

where T is a set of discrete observation points in time.

However, vectors x_t can be unstable, noisy, or interdependent. Therefore, for effective analysis of the SW state, we introduce a reflection:

$$\phi : X^{\Omega} \to Z,$$

where $Z \subseteq \mathrm{R}^m$ is a set of established (aggregated, normalized, discretized) features suitable for comparison, class construction, and use in the comparator identification method.

Let W be the size of the observation window (the number of steps back in time). Then, from the set of observations $X_t^W = \left\{ x_\tau \in X^{\Omega}, \tau \in [t - W + 1, t] \right\}$, i.e., from the time windows of observations, the aggregated features z_t are formed as:

$$z_t = \varphi\left(\left\{ x_\tau \right\}_{\tau=1}^{W} \right),$$

where φ – aggregation or transformation function: average, maximum, trend, autocorrelation, derivative, etc.

In general:

$$z_t = \Phi\left(X_t^W \right), z_t \in \mathrm{R}^m,$$

where $\Phi : \mathrm{R}^{W \times n} \to \mathrm{R}^m$ – feature construction function.

These aggregated features z_t, $t \in T$ are the basis for normalizing and comparing SW operating modes, constructing equivalence relations (in the comparator identification method), and determining functional classes. The set of all such vectors forms a space:

$$Z = \left\{ z_t \middle| t \in T \right\}.$$

Let us call those feature vectors $z^* \in Z$ that correspond to the normal (stable or functionally acceptable) mode of operation of the software system reference vectors. Let us denote the set of reference states as:

$$Z_{ref} = \left\{ z_i^* \in Z \middle| i = 1,\ldots,r \right\},$$

where r – number of known or recorded reference implementations.

Let us define the functional equivalence relation $\sim$ on the set Z if for two states $z_1, z_2 \in Z$:

$$z_1 \sim z_2 \Leftrightarrow \rho\left(z_1, z_2\right) \le \delta,$$

where $\rho\left(z_1, z_2\right)$ – selected similarity metric (e.g., Euclidean, Mahalanobis), $\delta > 0$ – acceptable deviation threshold.

Thus, each element $z \in Z$ either belongs to the class of one of the standards z_i^*, or initiates a new class.

The key idea of the proposed approach to preventive SW maintenance is not to attempt to identify numerical parameter values, but to determine classes of equivalent functional states based on comparisons with benchmarks. State identification is then carried out according to the following scheme:

Step 1. Calculate the distances $d_i = \rho\left(z_t, z_i^*\right), i = 1,\ldots,r$.

Step 2. If there is such i that $\mathrm{d}_i \le \delta$, then $z_t \sim z_i^*$ and z_t belongs to class i. Otherwise, the state of the software system at time t must be marked as abnormal or unknown.

This approach has certain limitations, i.e., the method is applicable when the system allows for the selection of representative reference implementations, features are aggregated, filtered from noise, and normalized, and a metric is available that preserves functional similarity rather than just geometric proximity.

The comparator identification method [18, 20] allows not only to record "failures", but also to classify states according to their functional proximity to known normal modes based on known feature values or to identify states with potential functional shifts (when the definedfeature vector does not match any reference).

Thus, this allows for preventive diagnostics without the need for explicit failures or "normal/abnormal" labels.

The complexity of preventive diagnostics lies in the fact that the state of the software system can be classified as normal based on formal characteristics, but the risk of degradation may increase. To eliminate uncertainty, it is proposed to extend the comparator identification method by identifying trends of deviation from reference states.

To do this, at each moment in time t, the SW state z_t is not only compared with the reference points, but also analyzed in terms of the dynamics of its approach or departure from the reference space Z_{ref} during the time interval $[t - W + 1, t]$.

For each moment in time t, the distance to the nearest reference point:

$$\mathrm{d}_t = \min_i \rho\left(z_t, z_i^*\right),\ z_i^* \in Z_{ref}.$$

Then the trajectory of distances over the time window W:

$$D_t^W = \left\{ d_{t-W+1}, d_{t-W+2}, \ldots, \mathrm{d}_t \right\}.$$

This sequence reflects whether the system is approaching or moving away from the reference value.

To assess the trend or deviation trend, you can use the gradient assessment.

$$\Delta_t = \frac{\mathrm{d}_t - \mathrm{d}_{t-W}}{W},$$

or sliding average change

$$\bar{\Delta}_t = \frac{1}{W-1} \sum_{i=1}^{W-1} \left(d_{t-i+1} - d_{t-i} \right).$$

Thus, the identification of the SW state occurs according to the specified values of the aggregated features z_t, the distance to the nearest reference state d_t, and the assessment of the deviation trend from the reference states Δ_t. If the permissible threshold for deviation growth is exceeded or a trend toward deviation growth is identified, the preventive monitoring system generates a signal indicating an increased risk of software system degradation.

Thus, comparative identification is used to determine the trajectory of deviation from a set of benchmarks. The key object is not the SW state z_t, but the sequence of deviations from a set of reference states D_t^z. This model allows predictive signals to be issued regarding the growth of degradation risks and forms the basis for developing an action plan.

Formalization of a reference model for preventive software maintenance

In accordance with the above structure of the reference model for preventive SW maintenance, let us consider the following basic models that form the basis of the framework.

1. SW state monitoring provides data collection on SW functioning, reflecting the state of the application and the external environment, forming a stream of observations $D_t = \left\{ x_{t1}, x_{t2}, \ldots, x_{tn} \right\}$. To do this, you need to specify a set of data sources K that form the raw data stream $s_t = \left\{ s_t^{(k)}, kmK \right\}$, the interval (frequency) of data collection τ, the operator for extracting SW state characteristics $\Psi\left(s_t\right)$, and the function $\Phi\left(X_t^W\right)$ for constructing features $z_t mZ$. That is,

$$\mathsf{M}_{mon} : K, \tau, \Psi, \Phi, Z.$$

The basic monitoring model reflects defined procedures for regular collection of data on SW functioning and determination of aggregate indicators for identifying the state of the software system and risks of its degradation.

2. Preventive identification determines deviations and potentially dangerous SW states based on an extended method of comparator identification. It is proposed to consider the following classes of SW degradation risks $Q = \left\{ q_1, q_2, q_3, q_4 \right\}$, where q_1 determines stable operating mode, q_2 is the SW functioning state with a possible future risk (if a trend of deviation from

reference states is detected), q_3 combines states with a tendency to return to the standard and q_4 is a class that defines a critical situation (the deviation from the standard exceeds the threshold and there is a tendency to move away). Therefore

$$\mathsf{M}_{id} : z_t, D_t^W \to Q .$$

Thus, the basic preventive identification model generates an alarm signal for the preventive monitoring system based on the analysis of the SW status and trends approaching the reference modes.

3. The task identification component generates structured tasks for the development team and is an integral part of the decision support system. Task definition is based on a defined risk class q_i, anomaly identification, i.e., the identification of features that influenced the results of preventive identification, and the existing knowledge base of relevant precedents. Let there be a set of task templates $A = \{a_{1,} a_2, \ldots, a_h\}$, a given function $\mathsf{G}(z_t)$ for defining essential features Z_t, and a defined class q_i. Then the task identification model is

$$\mathsf{M}_{task} : Z_t, q_i \to A .$$

So, based on the model M_{task} tasks and priorities for their implementation are defined.

4. The basic model for evaluating the results of actions is designed to measure the effect of preventive actions taken in terms of reducing the risks of software system degradation. Let $M = \{M^{(u)}, u = \overline{1,U}\}$ – a set of metrics (evaluation criteria) for SW quality. Then it is possible to determine the integral effect of the actions a performed, for example, as

$$\varepsilon(a) = \sum_{u=1}^{U} w_u \Delta M^{(u)}(a) .$$

In other words, evaluating the actions taken helps determine how the quality of the SW has changed and provides data for forming pairwise comparisons of states in terms of the obtained quality metric values, which collectively replenishes the knowledge base of precedents. That is,

$$\mathsf{M}_{eval} : A \to M, \varepsilon(a) .$$

Therefore, M_{eval} evaluation of actions allows to form feedback between the planning of the development team's work on SW maintenance and the actual impact of the changes made on the stability and state of the SW.

5. The basic configuration model performs the function of adaptation and re-identification, i.e., updating the rules and parameters of the comparison model. Let there be a set of paired comparisons formed at a given moment in time t

$$C_t = \{(Z_i, Z_j, y_{ij})\},$$

where y_{ij} determines how the state of the software system has changed (approached or deviated from the reference). Let Θ determines the parameters of the comparator model used for preventive identification, then let us denote $f_\theta(Z_t)$ as an identification model that implements transformation M_{id} with parameters Θ.

Then the adaptation model implements an iterative cycle

$$\mathrm{M}_{adapt}:\Theta_{t+1}=\arg\min_{\Theta}\mathrm{L}\left(C_t,\Theta\right)+\alpha\Theta-\Theta_t{}^2,$$

where L is the functional of the disturbance of the system of comparator equations, α is the stability parameter.

Thus, at the configuration stage, the preventive identification model is updated over time by accumulating historical data on the functioning of the software system, which ensures the ability of the preventive maintenance system to self-adapt.

Example of using a reference model for preventive maintenance

Let's consider a mobile application for patients with diabetes mellitus, which receives glucose readings from a Bluetooth glucometer, stores measurement history, builds short-term glucose level forecasts, and issues warnings. Preventive support is critically important for such an application: degradation of forecast accuracy or failure of sensor data synchronization without obvious application crashes directly affects medical risks.

During the operation of a software system that interacts with an external sensor device (glucometer), a stream of raw telemetry data is generated. Raw data is considered to be the direct results of the application's interaction with the hardware device and internal software components, which are recorded during the execution of individual read operations.

Each attempt to obtain or process sensor data is considered a discrete point in time, regardless of whether it was successful or ended in error. Thus, discrete observation moments correspond to events such as: initiating data reading from the glucometer; receiving a response from the device or recording a timeout; validating and parsing the received data; generating a glucose level forecast based on the input measurements. A set of raw data is recorded for each event:

$G(t)$ – actual glucose level;

$\hat{G}(t)$ – predicted value;

$s(t)$ – operation result (successful/error);

$\tau(t)$ – operation execution time.

Raw data is linked to events rather than to the state of the system, has different frequencies of occurrence, and may contain noise, so it cannot be used directly to assess the functional state of the system. To transition from raw data to formalized indicators, a set of current indicators is introduced, $r_i(t)$, which are calculated at the level of individual events and have a clear engineering meaning. Let's introduce three key operational metrics that the monitoring system collects daily:

$r_1(t)=G(t)-\hat{G}(t)$ – instantaneous glucose prediction error (absolute error in mmol/L between the actual glucose meter reading and the model prediction);

$r_2(t)=\begin{cases}0, if\ s(t)=\ fail\\ 1, if\ s(t)=ok\end{cases}$ – incorrect measurement indicator (0 or 1);

$r_3(t)=\tau(t)$ – Synchronization time with the glucometer (from screen opening to current value display, ms).

Monitoring operates in daily windows $W_k=\{t\,|\,t=1,2,\ldots\}$, where k is the day number and t is the event number in the monitoring window. That is, metric values are collected at discrete points in time throughout the day, forming observation streams. The number of successful reads is defined as $C_k=\sum\limits_{t\in W_k} r_2(t)$, and the total number of attempts is $N_k=|W_k|$.

For each day, we determine the aggregated features:

$x_1(k)=\dfrac{1}{C_k}\sum\limits_{t\in W_k}|r_1(t)|$ – mean error per day;

$x_2(k)=\dfrac{N_k-C_k}{N_k}$ – percentage of incorrect read attempts per day;

$x_3(k)=\dfrac{1}{N_k}\sum\limits_{t\in W_k} r_3(t)$ – mean synchronization time.

Then the SW state vector can be denoted as $x(k)=\left(x_1(k),x_2(k),x_3(k)\right)$. Thus, raw telemetry data is converted into current indicators $r_i(t)$ at the level of individual events. Next, at fixed time intervals W_k, these indicators are aggregated into a state feature vector $x(k)$ that describes the functional state of the software system per day (W_k interval).

Normalization is introduced to ensure correct combination of features. Normalization allows: bringing heterogeneous features to a dimensionless scale; assessing the degree of deviation of the current state from the normal mode; forming integral state indicators and applying formal identification rules. For a reference stable period (for example, the first 21 days after release), we form a set of reference states:

$$K_{ref}=\{1,2,\ldots,21\},\ Z_{ref}=\{x(k)\,|\,k\in K_{ref}\}.$$

For the reference period, we calculate the mean values and standard deviations:

$$\mu_i=\frac{1}{|K_{ref}|}\sum_{k\in K_{ref}} x_i(k),\ \sigma_i^2=\frac{1}{|K_{ref}|}\sum_{k\in K_{ref}}\left(x_i(k)-\mu_i\right)^2,\ i=1,2,3.$$

Then the normalized features are defined as:

$$z_i(k)=\frac{x_i(k)-\mu_i}{\sigma_i^2},\ i=1,2,3.$$

Normalized state vector:

$$z(k)=\left(z_1(k),z_2(k),z_3(k)\right).$$

The obtained normalized vector $z(k)$ is used as input for further comparator identification and classification of the functional states of the software system. Next, the reference (baseline) mode of operation of the software system is determined. The set of indices K_{ref} can be expanded using all k that correspond to the period of stable operation. $Z_{ref}=\{x(k)\,|\,k\in K_{ref}\}$ is a reference set of normalized states. In the simplest case, the reference is represented by a single representative vector (the center of the reference mode).

$$z_{ref} = \frac{1}{|K_{ref}|} \sum_{k \in K_{ref}} z(k),$$

where $\sum_{k \in K_{ref}} z(k)$ means the component-wise sum of vectors, and z_{ref} is the centroid of the set of normalized states of the reference mode.

We define the deviation from the reference as the weighted Euclidean distance:

$$d(k) = \sqrt{w_1 \left(z_1(k) - z_{1,ref}\right)^2 + w_2 \left(z_2(k) - z_{2,ref}\right)^2 + w_3 \left(z_3(k) - z_{3,ref}\right)^2}, \; w_1 + w_2 + w_3 = 1,$$

where, for example, we give greater weight to medically critical errors in prognosis: $w_1 = 0,4$; $w_2 = 0,3$; $w_3 = 0,3$.

The local (component) deviation is determined separately for each feature:

$$\delta_i(k) = \left| z_i(k) - z_{i,ref} \right|, i = 1,2,3.$$

The vector $\delta(k)$ is used to explain the results (which metrics caused the growth $d(k)$ and to further form preventive tasks.

For preventive support, it is important not only to know the current value of the deviation, but also how it changes over time. To assess the deviation trend, we use a sliding window with a length of, for example, $L = 7$ days:

$$g(k) = \frac{d(k) - d(k - L + 1)}{L - 1}, k \ge L.$$

A pair of indicators $\left(d(k), g(k)\right)$ is sufficient statistics for further classification of states. To interpret the monitoring results and support decision-making, a set of discrete classes of the functional state of the software system is introduced:

R_0 – normal mode;

R_1 – deviation from the standard, but there is a tendency for the deviation to decrease;

R_2 – high-risk mode (preventive intervention is necessary);

R_3 – critical condition.

Let us introduce predicates for recognizing conditions related to degradation and risks of the software system:

$P_1(k) \Leftrightarrow d(k) \le d_0$ determines proximity to the reference mode, d_0 – the threshold of proximity to the reference;

$P_2(k) \Leftrightarrow |g(k)| \le g_0$ determines proximity to the reference during a sliding window, g_0 – the threshold of acceptable trend deviation;

$P_3(k) \Leftrightarrow g(k) < 0$ corresponds to a situation of decreasing deviation from the reference, i.e., there is a tendency to return to the reference;

$P_4(k) \Leftrightarrow g(k) > 0$ determines the tendency of deviation from the reference mode.

Then, the correspondence of the state to one of the classes is determined by the following conditions:

$$R_0(k) \Leftrightarrow \left(P_1(k) \wedge P_2(k)\right) = 1;$$

$$R_1(k) \Leftrightarrow \left(\neg P_1(k) \wedge P_3(k)\right) = 1;$$

$$R_2(k) \Leftrightarrow \left(P_1(k) \wedge P_4(k) \wedge \neg P_2(k)\right) = 1;$$

$$R_3(k) \Leftrightarrow \left(\neg P_1(k) \wedge P_4(k) \wedge \neg P_2(k)\right) = 1.$$

Analysis of experimental results for a 60-day period of operation allows us to clearly illustrate the operation of the proposed classification system. Figure 2 shows a graph of changes in forecast error values during the experiment.

At the initial interval of the experiment, the values $d(k)$ (see Fig. 3) do not exceed the threshold $d_0 = 1$, and the trend $g(k)$ fluctuates near zero, which corresponds to the fulfillment of conditions $\left(P_1(k) \wedge P_2(k)\right)$. This is consistent with the classification of the system into class R_0 and confirms a stable mode of operation, which is clearly visible in the graph.

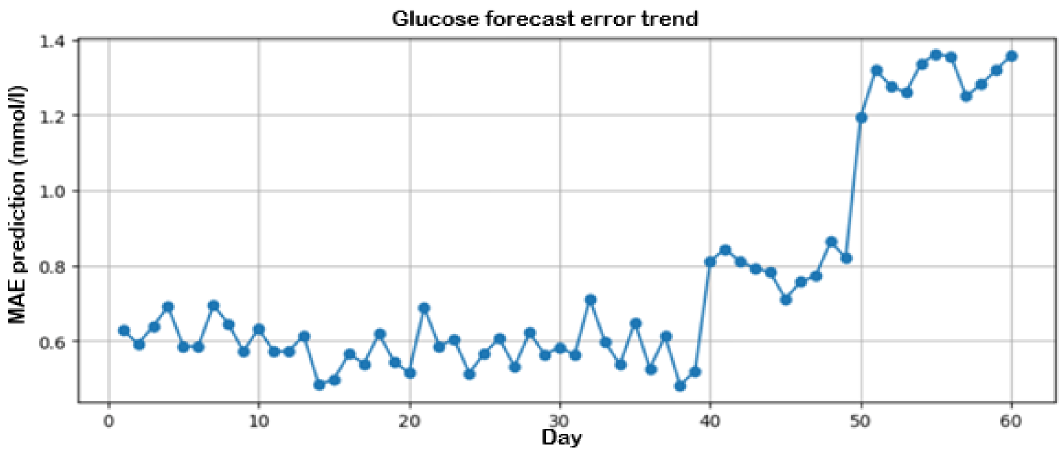


Fig. 2. $x_1(k)$ values during 60 days of observation

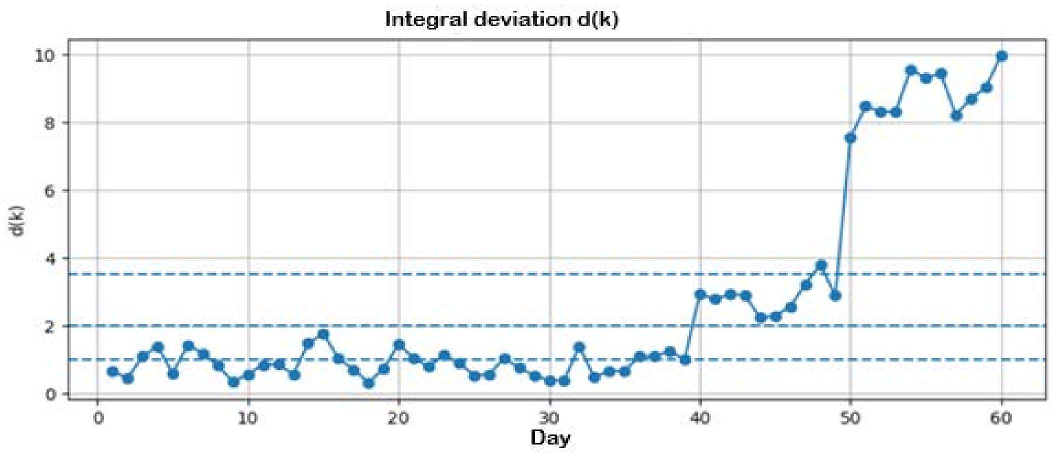
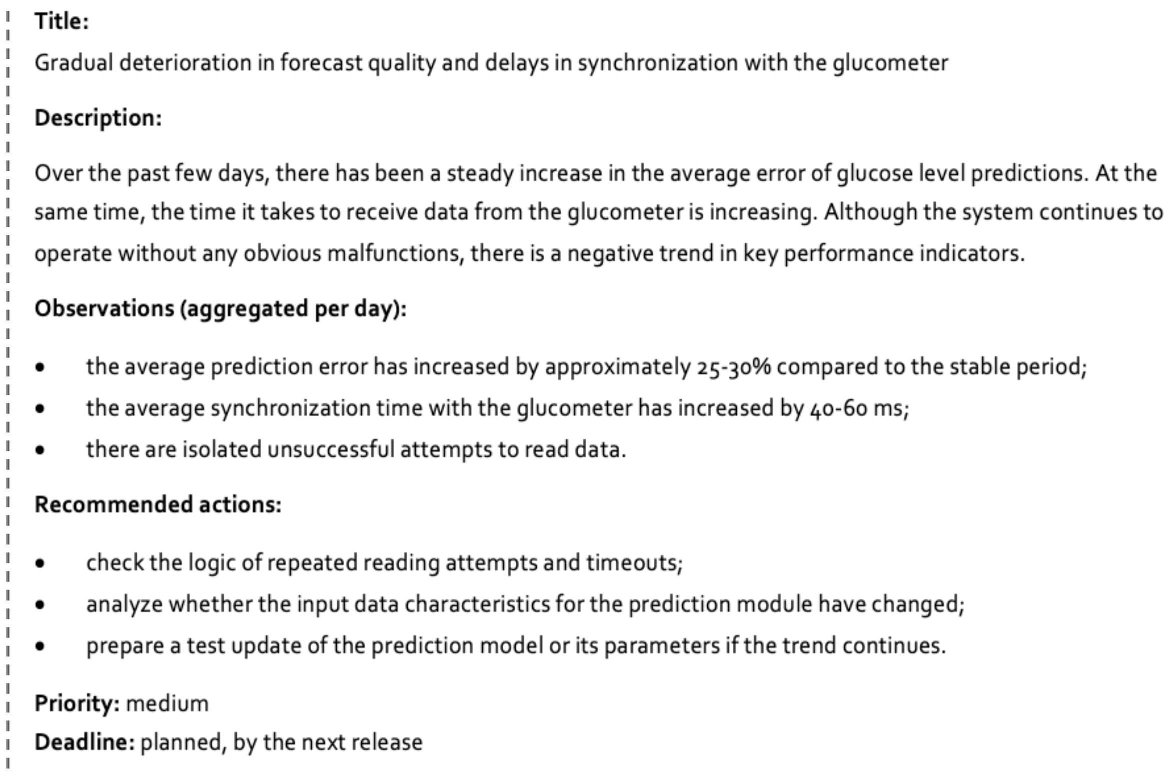


Fig. 3. Trend in deviation from the reference mode

With the onset of monotonous growth of the integral deviation (Fig. 3), it is possible to signal the formation of a pre-crisis state and the need for preventive action, which is determined by the transition to class R_2. Further simultaneous violation of the conditions of proximity to the standard and the permissible trend leads to the state being classified as class R_3, which corresponds to the critical state of the software system.

The proposed system of predicates and classes provides a formal link between numerical monitoring characteristics and engineering decisions. The transition to classes R_2 and R_3 is used as a trigger for the automatic generation of support tickets, in which the values $d(k)$, $g(k)$ and component deviations serve as justification for the necessary preventive or urgent actions.

The results of the classification of the functional state of the software system are not communicated to the development team in the form of formal classes or threshold values. Instead, they are transformed into support tickets (tasks for the development team) containing a clear description of the problem, quantitative observations, and recommended technical actions. An example of a ticket in the event of a transition to a class R_2 (high-risk mode) is shown in Figure 4.



Title:

Gradual deterioration in forecast quality and delays in synchronization with the glucometer

Description:

Over the past few days, there has been a steady increase in the average error of glucose level predictions. At the same time, the time it takes to receive data from the glucometer is increasing. Although the system continues to operate without any obvious malfunctions, there is a negative trend in key performance indicators.

Observations (aggregated per day):

- the average prediction error has increased by approximately 25-30% compared to the stable period;
- the average synchronization time with the glucometer has increased by 40-60 ms;
- there are isolated unsuccessful attempts to read data.

Recommended actions:

- check the logic of repeated reading attempts and timeouts;
- analyze whether the input data characteristics for the prediction module have changed;
- prepare a test update of the prediction model or its parameters if the trend continues.

Priority: medium

Deadline: planned, by the next release

Fig. 4. Example of a preventive ticket (early degradation)

The example considered demonstrates that the proposed reference model provides a coherent internal logic for monitoring the state of the software system based on aggregated operational metrics and their dynamics. The model formalizes the transition from raw data to integrated indicators of deviation and trends, allowing for the correct identification of both stable operating modes and early signs of degradation. The use of formalized classification conditions ensures the reasonable formation of alerts without the need for rigidly fixed scenarios, and the analysis of component deviations provides structured information about the possible causes of deterioration in the quality of the system's operation and directions for its elimination. This allows the results of identification to be translated into engineering tasks that are understandable to the development team in the form of standard support tickets.

Thus, the model supports the automated formation of tasks for the development team based on quantitative observations and their trends, which significantly reduces the time between the first signs of degradation and the start of engineering intervention. This ensures the transition from reactive incident resolution to proactive software maintenance in real operating conditions.

Discussion

The comparator identification method [18, 20] is a special variant of solving identification problems when direct measurement of the parameter of interest is impossible. In classical identification, an attempt is made to determine the law of their connection based on the input and output signals of the object. However, in many cases, the output of the system (for example, the "failure susceptibility level") cannot be measured directly. For such situations, indirect identification methods are used, and the most convenient among them is the declared method of comparator identification.

The idea of using the comparator identification method in the task of preventive SW maintenance looks promising, since the very nature of the problem corresponds to the setting of indirect identification. It is not possible to directly measure the "probability of future failure" or the "level of problematicity" of the system state, but we can compare different states using indirect reliability indicators. Despite its scientific validity, the practical feasibility of this method in complex software systems should be critically evaluated.

First, the comparator identification method essentially provides a binary conclusion. Additional mechanisms are needed for a fine gradation of risks.

Second, the key step – forming a system of equations – requires the availability or accumulation of representative data on various SW operating situations.

Third, software and its operating environment change over time (updates, new features, changing loads), so the model will have to be regularly updated as new data becomes available. This leads to the need to modify the classical method.

In engineering practice, statistical methods and machine learning are more often used for SW support [9, 15], which usually requires a well-established infrastructure for collecting and processing large amounts of data. The comparator approach, on the contrary, can introduce an element of strict justification and interpretability. Its mathematical correctness and ability to work with fuzzy information provide significant advantages for solving the complex task of preventive maintenance, where uncertainty and multi-criteria are the main characteristics. If the method is adapted, it can become part of intelligent maintenance systems, increasing the objectivity of decisions on preventive measures.

In general, the idea of applying the comparator identification method is in line with the global trend towards proactive, data-driven software maintenance and appears promising, although the effective application of predictive analytics also requires a culture that is ready to accept the model's recommendations.

The results show that the proposed reference model of preventive support can become a fundamental basis for the transition from reactive software support to systematic proactive quality management. Unlike traditional approaches, which focus primarily on fixing already identified defects, the developed model uses a formalized system of aggregated features, deviation trends, and comparator assessment, which allows identifying degradation risks without the presence of obvious failures.

This approach provides early warning of potential problems and creates a basis for automated task generation, which is especially important in highly dynamic environments such as mobile applications or systems with machine learning components and smart manufacturing complexes [21].

It is important to emphasize that the proposed model forms not only a mechanism for analyzing the state of the software system, but also a closed cycle of its evolution through adaptive updating of parameters and identification rules. This allows the maintenance system to respond to changes in the external environment, SW updates, the emergence of new usage modes, and the dynamics of machine learning models. The adaptive component ensures the stability of the approach in the long term, preventing accuracy degradation in conditions of data drift, seasonal load, or changes in user activity profiles.

At the same time, the results highlight a number of important challenges. The effectiveness of the method largely depends on the quality of the set of reference states and the adequacy of the selected distance metrics. The presence of insufficiently representative benchmarks or noisy features can lead to incorrect signals, which is critical in systems where the frequency of false alarms must be minimized.

Another aspect that requires further research is the integration of the model into real CI/CD and DevOps processes: determining the frequency of analysis, limitations on computing resources, communication with the task backlog, and the balance between automated and manual developer solutions.

Overall, the results obtained indicate the promise of combining comparator identification with the principles of preventive maintenance. This approach provides a formally justified, interpreted, and at the same time practically applicable mechanism for detecting hidden degradation trends, which enhances the overall ability of software systems to self-identify and proactively improve.

Further research directions may be related to the automatic formation of reference classes, the expansion of the model by means of machine learning, the verification of effectiveness on different classes of systems, and the development of prototypes for integration into the support infrastructure.

Conclusions

The paper proposes a reference model for preventive software maintenance that covers the full cycle of proactive software system management, from monitoring and detecting degradation trends to forming maintenance tasks, evaluating results, and adaptively updating identification rules. Unlike traditional reactive approaches, the model provides early detection of potentially dangerous changes in software behavior and allows intervention before critical failures occur.

The analytical mechanism of the model is based on an extended comparator identification method, which provides a formalized comparison of current states with a set of reference modes and the identification of deviation trends. This approach makes it possible to detect hidden degradation processes even in the absence of obvious incidents, which significantly increases the effectiveness of preventive support.

The adaptive component of the model ensures that identification parameters and rules remain relevant in dynamic operating conditions, particularly when there are changes in load, user behavior, or data drift in systems with machine learning components.

The formal models formulated for each element of the reference structure (monitoring, preventive identification, task definition, result evaluation, and adaptation) create a comprehensive framework for preventive support. It can be integrated into modern CI/CD, DevOps, and MLOps processes, serving as a mechanism for improving the reliability, security, and maintainability of software systems.

The results obtained open up prospects for further research in the areas of automatic reference state formation, the use of machine learning methods to refine risk class boundaries, the development of tools for integration with the support infrastructure, and experimental validation of the model on different types of software products.

The proposed model can become the basis for the creation of intelligent preventive maintenance systems that ensure the stable and predictable functioning of software complexes in the long term.

References

1. Masrat, A., Makki, M. A., Gawde, H. (2021), "Software maintenance models and processes: An overview", *SSRN Electronic Journal*. DOI: https://doi.org/10.2139/ssrn.3838444
2. ISO/IEC/IEEE 14764:2022. Software engineering – Software life cycle processes – Maintenance. – Geneva: International Organization for Standardization.
URL: https://www.iso.org/standard/80710.html (дата звернення: 10.10.2025).
3. Young, Z., Steele, R. (2022), "Empirical evaluation of performance degradation of machine learning-based predictive models – A case study in healthcare information systems", *International Journal of Information Management Data Insights*, Vol. 2, Iss. 1., Art. 100070.
DOI: https://doi.org/10.1016/j.jjimei.2022.100070

4. Ibrahim, K. S. K., Mansor, Z., Yahaya, J., Deraman, A. (2025), "Software maintenance assessment: An analysis of determination factors", *Journal of Mathematical Sciences and Informatics*, Vol. 5, No. 1. DOI: https://doi.org/10.46754/jmsi.2025.06.006
5. Kleinwaks, H., Gärtner, M., Reich, J., Woehrle, G. (2023), "Technical debt in systems engineering - A systematic literature review", *Systems Engineering*, Vol. 26, Iss. 6, P. 741–760. https://doi.org/DOI: 10.1002/sys.21681
6. Lenarduzzi, V., Besker, T., Taibi, D., Martini, A., Arcelli Fontana, F. (2021), "A systematic literature review on technical debt prioritization: Strategies, processes, factors, and tools", *Journal of Systems and Software*, Vol. 171, Art. 110827. DOI: https://doi.org/10.1016/j.jss.2020.110827
7. ISO/IEC 25010:2023. Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Product quality model. – Geneva: International Organization for Standardization. URL: https://www.iso.org/standard/78176.html (дата звернення: 12.10.2025).
8. Abdu, A., Abdo, H. A., Ullah, I., Khan, J., Gu, Y. H., Algabri, R. (2025), "Deep multi-metrics learning for mobile app defect prediction using code and process metrics", *Scientific Reports*, Vol. 15, Iss. 1, Art. 38620. DOI: https://doi.org/10.1038/s41598-025-22566-2
9. Dai, H., Xi, J., Dai, H.-L. (2024), "Improving effort-aware just-in-time defect prediction with weighted code churn and multi-objective slime mold algorithm", *Heliyon*, Vol. 10, Iss. 18, e37360. DOI: https://doi.org/10.1016/j.heliyon.2024.e37360
10. Rebro, D. A., Rossi, B., Chren, S. (2023), "Source code metrics for software defects prediction", *Proc. 38th ACM/SIGAPP Symposium on Applied Computing*, P. 1668–1677. DOI: https://doi.org/10.48550/arXiv.2301.08022
11. Heričko, T., Šumak, B. (2023), "Exploring Maintainability Index variants for software maintainability measurement in object-oriented systems", *Applied Sciences*, Vol. 13, Iss. 5, Art. 2972. DOI: https://doi.org/10.3390/app13052972
12. Moulla, D. K., Mnkandla, E., Oumarou, H., Fehlmann, T. (2023), "Technical debt measurement: An exploratory literature review", *Proc. Int. Workshop on Software Measurement and Metrics (IWSM)*, P. 27–38.
13. Saraiva, J. D., Neto, J. G., Kulesza, U., Freitas, G., Rebouças, R., Coelho, R. (2021), "Technical debt tools: A systematic mapping study", *Proc. 23rd International Conference on Enterprise Information Systems (ICEIS)*, P. 280–303. DOI: https://doi.org/10.5220/0010459100880098
14. Dreyfus, P.-A., Pélissier, A., Psarommatis, F., Kiritsis, D. (2022), "Data-based model maintenance in the era of Industry 4.0: A methodology", *Journal of Manufacturing Systems*, Vol. 63, P. 304–316. https://doi.org/DOI: 10.1016/j.jmsy.2022.03.015.
15. Eken, B., Pallewatta, S., Tran, N. K., Tosun, A., Babar, M. A. (2025), "A multivocal review of MLOps practices, challenges and open issues", *ACM Computing Surveys*, Vol. 58, Iss. 2, P. 1–35. DOI: https://doi.org/10.1145/3747346.
16. Paleyes, A., Urma, R.-G., Lawrence, N. D. (2022), "Challenges in deploying machine learning: A survey of case studies", *ACM Computing Surveys*, Vol. 55, Iss. 6, Art. 114. DOI: https://doi.org/10.1145/3533378
17. Indykov, V., Strüber, D., Lwakatare, L. E., Felderer, M. (2025), "Architectural tactics to achieve quality attributes of machine-learning-enabled systems: A systematic literature review", *Journal of Systems and Software*. DOI: https://doi.org/10.1016/j.jss.2025.112373
18. Cherednichenko, O., Vovk, M., Sharonova, N., Vorzhevitina, A. (2025), "Comparator-based identification of food edibility from natural language description", *AICS-CoLInS 2025*, *CEUR Workshop Proceedings*, Vol. 4015, P. 185–199. DOI: https://doi.org/10.31110/COLINS/2025-3/014

19. Sharonova, N., Doroshenko, A., Cherednichenko, O. (2018), "Issues of fact-based information analysis", *Proc. 2nd International Conference on Computational Linguistics and Intelligent Systems (COLINS), CEUR Workshop Proceedings*, Vol. 2136, P. 11–19.
20. Sharonova, N., Doroshenko, A., Cherednichenko, O. (2018), "Towards the ontology-based approach for factual information matching", *Proc. 7th International Scientific and Technical Conference "Information Systems and Technologies" (IST-2018),* P. 230–233.
21. Xu, J., Kovatsch, M., Mattern, D., Mazza, F., Harasic, M., Paschke, A., Lucia, S. (2022), "A review on AI for smart manufacturing: Deep learning challenges and solutions", *Applied Sciences,* Vol. 12, Iss. 16, Art. 8239. DOI: https://doi.org/10.3390/app12168239

Received (Надійшла) 24.11.2025
Accepted for publication (Прийнята до друку) 07.12.2025
Publication date (Дата публікації) 28.12.2025

About the Authors / Відомості про авторів

Chupryna Anastasiya – PhD (Engineering Sciences), Associate Professor, Kharkiv National University of Radio Electronics, Associate Professor at the Department of Software Engineering, Kharkiv, Ukraine; e-mail: anastasiya.chupryna@nure.ua; ORCID ID: https://orcid.org/0000-0003-0394-9900

Repikhov Vadym – Kharkiv National University of Radio Electronics, PhD Student, Department of Software Engineering, Kharkiv, Ukraine; e-mail: vadym.repikhov@nure.ua;
ORCID ID:https://orcid.org/0000-0002-1274-4205

Чуприна Анастасія Сергіївна – кандидат технічних наук, доцент, Харківський національний університет радіоелектроніки, доцент кафедри програмної інженерії, Харків, Україна.

Репіхов Вадим Миколайович – Харківський національний університет радіоелектроніки, аспірант кафедри програмної інженерії, Харків, Україна.

ОПОРНА МОДЕЛЬ ПРЕВЕНТИВНОГО СУПРОВОДЖЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Предметом роботи є процеси превентивного супроводження програмного забезпечення, зокрема формалізація механізмів моніторингу, ідентифікації ризиків деградації та побудови рішень щодо проактивного втручання в життєвий цикл програмних систем. **Мета дослідження** – розробити опорну модель превентивного супроводження, яка забезпечує систематичне виявлення відхилень від еталонних режимів роботи SW, оцінювання тенденцій їх зміни й генерацію обґрунтованих превентивних дій до появи критичних відмов. З огляду на окреслену мету необхідно було виконати такі **завдання:** запропонувати класифікацію метрик і ознак для опису функціонального стану SW; побудувати формальний апарат превентивної ідентифікації на основі розширеного методу компараторної ідентифікації; визначити структурні компоненти опорної моделі; розробити моделі моніторингу, аналізу ідентифікованих ризиків, керування задачами супроводження й адаптивного оновлення параметрів моделі. **Методи дослідження.** Методологічну основу становить розширений метод компараторної ідентифікації, який дає змогу порівнювати поточні стани SW з еталонними режимами та кількісно оцінювати динаміку їх відхилення. Застосовано агрегування та нормалізацію ознак, формалізацію множини функціональних станів, аналіз часових вікон і трендових властивостей, а також механізми зворотного зв'язку й адаптивної корекції параметрів. **Досягнуті результати.** Сформовано п'ятикомпонентну опорну модель, що містить моделі: моніторингу стану SW; превентивної ідентифікації ризиків деградації; формування задач супроводження; оцінювання результативності превентивних дій; адаптації параметрів. Розроблена модель забезпечує виявлення латентних процесів деградації за допомогою аналізу траєкторій відхилень, інтегрується в сучасні процеси *CI/CD*, *DevOps* і *MLOps* та створює формалізований механізм підтримки прийняття рішень щодо вчасного супроводження. **Висновки.** Запропонована модель формує цілісний формальний фреймворк превентивного супроводження, спрямований на раннє виявлення потенційних відмов і підтримку довгострокової стабільності програмних систем. Адаптивний характер моделі забезпечує її актуальність у динамічних умовах експлуатації, а застосування компараторної ідентифікації підвищує інтерпретованість і обґрунтованість прийнятих рішень. Отримані результати створюють підґрунтя для розроблення інтелектуальних систем супроводження й подальших досліджень з автоматизації формування еталонних станів і розширення моделі засобами машинного навчання.

Ключові слова: превентивне супроводження; компараторна ідентифікація; моніторинг; технічний борг; прогнозування деградації; якість ПЗ; *DevOps*; *MLOps*.

Bibliographic descriptions / Бібліографічні описи

Chupryna, A., Repikhov, V. (2025), "Reference model for preventive software maintenance", *Management Information Systems and Devises*, No. 4 (187), P. 254–277.
DOI: https://doi.org/10.30837/0135-1710.2025.187.254

Чуприна А. С., Репіхов В. М. Опорна модель превентивного супроводження програмного забезпечення. *Автоматизовані системи управління та прилади автоматики*. 2025. № 4 (187). С. 254–277. DOI: https://doi.org/10.30837/0135-1710.2025.187.254