DOI: 10.15514/ISPRAS-2026-38(2)-6

Continuous Observability for Client–Server IDEs: A Reproducible, Low-Overhead Approach

[1] *V.I. Miroshnikov, ORCID: 0000-0002-6218-9406 <vladislaw.miroshnikov@gmail.com>*
[2] *O.I. Bachishche, ORCID: 0009-0007-5247-4362 <bachisheo@yandex.ru>*
[1] *I.A. Kuznetsov, ORCID: 0009-0009-2215-7299 <kuznetsov.ilya.alexandrovich@gmail.com>*
[2] *A.A. Platonov, ORCID: 0009-0004-6626-2297 <planol@me.com>*
[1] *N.V. Tropin, ORCID: 0009-0006-2910-3961 <niktrop@yandex.ru>*
[2] *D.V. Vasina, ORCID: 0009-0001-0728-956X <dashavasina625@gmail.com>*
[1] *D.V. Koznov, ORCID: 0000-0003-2632-3193 <d.koznov@spbu.ru>*

[1] *St. Petersburg State University,*
7-9, Universitetskaya Embankment, St Petersburg, 199034, Russia.

[2] *ITMO University,*
Kronverksky Pr. 49, bldg. A, St. Petersburg, 197101, Russia.

Abstract. Client–server Integrated Development Environments (IDEs) are complex systems where small code changes often cause performance regressions that standard metrics miss. This paper addresses the lack of continuous, reproducible observability platforms focused on developer-perceived latency and stability in dynamic language environments like Python. We implemented a production-grade observability pipeline integrated into CI/CD that instruments backend services to capture traces and metrics. To ensure reproducibility, workloads execute in version-pinned containers against a corpus of open-source projects. Instead of static limits, the system detects regressions using a sliding-window algorithm that calculates robust z-scores and relative shift thresholds. Over one year of operation, the platform surfaced more than 40 performance issues, including a 5–6 times regression in index saving and a 25% memory drift detected via nightly test execution. It further validated architectural optimizations that yielded a 30% speedup in project reopening. The findings demonstrate that relative windowed alerting is significantly more robust than fixed thresholds for detecting anomalies in composite systems. This approach proves that comprehensive observability is achievable with negligible runtime overhead, enabling developers to identify and resolve regressions prior to merging.

Keywords: client–server IDE; observability; monitoring; Python; language server; OpenTelemetry; VictoriaMetrics; Grafana; CI/CD; regression detection; reproducibility; alerting; performance; memory usage; developer experience.

For citation: Miroshnikov V.I., Bachishche O.I., Kuznetsov I.A., Platonov A.A., Tropin N.V., Vasina D.V., Koznov D.V. Continuous Observability for Client–Server IDEs: A Reproducible, Low-Overhead Approach. Trudy ISP RAN/Proc. ISP RAS, vol. 38, issue 2, 2026, pp. 83-94. DOI: 10.15514/ISPRAS-2026-38(2)-6.

Непрерывная наблюдаемость для клиент-серверной IDE: воспроизводимый подход с низкими накладными расходами

[1] *В.И. Мирошников, 0000-0002-6218-9406 <vladislaw.miroshnikov@gmail.com>*
[2] *О.И. Бачище, ORCID: 0009-0007-5247-4362 <bachisheo@yandex.ru>*
[1] *И.А. Кузнецов, ORCID: 0009-0009-2215-7299 <kuznetsov.ilya.alexandrovich@gmail.com>*
[2] *А.А. Платонов, ORCID: 0009-0004-6626-2297 <planol@me.com>*
[1] *Н.В. Тропин, ORCID: 0009-0006-2910-3961 <niktrop@yandex.ru>*
[2] *Д.В. Васина, ORCID: 0009-0001-0728-956X <dashavasina625@gmail.com>*
[1] *Д.В. Кознов, ORCID: 0000-0003-2632-3193 <d.koznov@spbu.ru>*

[1] *Санкт-Петербургский государственный университет,*
Россия, 199034, Санкт-Петербург, Университетская наб., 7/9.

[2] *Национальный исследовательский университет ИТМО,*
Россия, 197101, Санкт-Петербург, Кронверкский пр., д. 49, лит. А.

Аннотация. Клиент-серверные интегрированные среды разработки (IDE) представляют собой сложные системы, в которых даже незначительные изменения исходного кода могут привести к регрессиям производительности, которые невозможно обнаружить с помощью стандартных метрик. Данная статья посвящена проблеме отсутствия платформ непрерывной и воспроизводимой наблюдаемости, ориентированных на задержки, воспринимаемые разработчиком, и стабильность работы в средах динамических языков, таких как Python. В рамках работы была реализована и интегрирована в CI/CD платформа наблюдаемости промышленного уровня, которая осуществляет инструментацию бэкенд-сервисов для сбора трассировок и метрик. Для гарантии воспроизводимости рабочие нагрузки выполняются в контейнерах с фиксированными версиями на специально отобранном корпусе проектов с открытым исходным кодом. Вместо использования статических границ измерений система обнаруживает регрессии с помощью алгоритма скользящего окна, который рассчитывает робастные z-оценки (z-scores) и пороги относительного сдвига. За год эксплуатации платформа выявила более 40 проблем производительности, включая регрессию в 5-6 раз при сохранении индексов и 25% смещение в потреблении памяти, обнаруженное благодаря ночному запуску тестов. Кроме того, система подтвердила эффективность архитектурных оптимизаций, обеспечивших 30% ускорение при повторном открытии проектов. Результаты демонстрируют, что оповещение на основе относительных показателей в скользящем окне значительно надежнее фиксированных пороговых значений для обнаружения аномалий в составных системах. Данный подход доказывает, что всесторонняя наблюдаемость достижима с ничтожно малыми накладными расходами во время выполнения, что разработчикам выявлять и устранять регрессии до слияния веток кода.

Ключевые слова: клиент-серверная среда IDE; наблюдаемость; мониторинг; язык программирования Python; языковой сервер; стандарт OpenTelemetry; система VictoriaMetrics; платформа Grafana; методология CI/CD; обнаружение регрессии; воспроизводимость; оповещения; производительность; использование памяти; опыт разработчика.

Для цитирования: Мирошников В.И., Бачище О.И., Кузнецов И.А., Платонов А.А., Тропин Н.В., Васина Д.В., Кознов Д.В. Непрерывная наблюдаемость для клиент-серверной IDE: воспроизводимый подход с низкими накладными расходами. Труды ИСП РАН, том 38, вып. 2, 2026 г., стр. 83–94 (на английском языке). DOI: 10.15514/ISPRAS–2026–38(2)–6.

1. Introduction

Integrated Development Environments (IDEs) must remain stable throughout long product development cycles. In practice, minor source code changes can increase delays in code analysis or cause memory regressions. Consequently, accurate control over an IDE's performance throughout the development cycle is essential.

The field of software performance measurement is well-established [1]. Existing literature predominantly covers fundamental resource metrics such as memory consumption, CPU utilization,

and I/O throughput. However, modern IDEs are composite systems comprising numerous distinct components. While prior research has investigated performance measurements for specific subsystems – such as debugging infrastructure [2] or code completion systems – the holistic performance measurement of an IDE as an unified product remains underexplored.

Furthermore, language-specific IDEs present unique challenges that extend beyond generic performance metrics. For instance, while type inference may be a negligible factor in statically typed languages, it is a computationally intensive process for dynamically typed languages like Python, directly impacting user perception of speed.

Another critical aspect of production-level product measurement is performance control throughout the development process. DevOps evidence demonstrates that automated, feedback-rich pipelines shorten the time required to detect and correct regressions in software-intensive products [3]. These factors collectively necessitate continuous, reproducible observability focused on what developers perceive: end-to-end latency of key actions, resource footprints under realistic workloads, and alerts that trigger on relative changes rather than fixed thresholds.

Our study focuses on developing a comprehensive approach to complex performance measurement for a production-level IDE. The research was conducted using the Pyter Python IDE, which is being developed by a team of engineers and researchers from the Chebyshev Saint Petersburg Research Center. It is a proprietary, client–server development environment for Python. The backend provides features such as parsing, indexing, type inference, code completion, and navigation services. We designed and deployed an observability platform targeting these services and their client interactions, capturing trends that affect user-visible performance and stability.

This paper contributes a production-oriented continuous observability platform for a client–server Python IDE with four design priorities: (i) reproducible runs on an open source corpus; (ii) low-overhead, versioned telemetry; (iii) long-horizon trend analysis with actionable alerts; and (iv) tight CI/CD integration, so developers can validate performance and quality before merge. We intentionally avoid complex metrics categorizations; the main text mentions only a few representative indicators (e.g., project startup; peak memory during indexing), with dashboards and examples left to figures.

Our main contributions are as follows:

- *Production observability platform*. We build a continuous, reproducible observability pipeline for a client–server IDE to track developer-visible latency, resource usage, and stability.
- *CI integration and alerting.* We integrate this pipeline into CI (nightly and pre-merge) with representative workloads on a version-pinned open-source software (OSS) corpus in pinned Docker environments, and use sliding-window, relative alerts to surface meaningful trend shifts.
- *Operational insights.* We report on one year of industrial use (>40 surfaced regressions and improvements) and distill practical guidance on metric/label design, environment pinning, alert tuning, and keeping the observability stack reliable yet lightweight.

Section 2 synthesizes prior work on IDE quality, LSP ecosystems, and DevOps evidence [1–4]. Section 3 presents the client–server Python IDE and the reproducibility setup, describing the observability stack and operations. Section 4 evaluates the platform with real regressions/improvements and overhead. Section 5 discusses research results, future work and limitations, including engineering effort and maintenance.

2. Related Work

Historical Context and Architectural Evolution. The evolution of IDEs has traditionally focused on structure-oriented editing and monolithic architecture. Early foundational work, such as the Cornell Program Synthesizer by Teitelbaum and Reps [5] and the PECAN system by Reiss [6], emphasized

tight coupling between language semantics and user interfaces. However, recent developments have shifted towards decoupled, client-server models. The standardization of the Language Server Protocol (LSP), as detailed by Rask [7] and Microsoft [8], has revolutionized this space by separating language intelligence from editor frontends. While Rodríguez-Echeverría [9] advocates for extending LSP to graphical modeling, the literature accepts this distributed architecture as the modern standard for text-based programming.

User Experience and the AI Challenge. Research into developer behavior highlights the criticality of tool responsiveness. Murphy et al. [10] and Ko et al. [11] have long established that usability barriers and latency significantly degrade developer productivity. This challenge is exacerbated by the recent integration of AI-driven features. Studies by Hellendoorn et al. [12] and Svyatkovskiy et al. [13] demonstrate that while models enhance code completion, they introduce substantial computational overhead and non-deterministic latency. Vaithilingam et al. [14] further argue that the usability of these tools depends heavily on their performance reliability, yet standard IDEs often lack transparency regarding these internal resource costs.

The Lack of Continuous Telemetry. Despite the maturity of DevOps practices described by Erich et al. [4] and Rodríguez et al. [15] for general software delivery, there is a notable scarcity of research applying these principles to the internal observability of IDEs themselves. While functional testing tools like RCPTT [16] or Eclipse Jubula [17] exist, they focus primarily on UI correctness rather than performance trends. Current literature lacks a comprehensive framework for the continuous, metric-based observability of client-server IDEs. Specifically, there is no established methodology for using telemetry (such as OpenTelemetry) to detect performance regressions and memory drifts in language servers across release cycles, a gap this study aims to address.

3. Methodology and Experimental Setup

3.1 System Overview and Architecture

The observability platform targets the backend services of the Python IDE, specifically the Language Server [8], code model, and indexing subsystems. The platform architecture (see Fig. 1) follows a continuous delivery approach to ensure stability throughout long product development cycles [3-4]. Core IDE services were instrumented by *OpenTelemetry* [18] to capture metrics and traces at key execution boundaries.

These metrics are buffered in-process and exported via HTTP in a Prometheus compatible format to *VictoriaMetrics* [19], which serves as the persistent time-series database. Historically we used *Prometheus* + *PushGateway* for short-lived CI jobs; the move to VictoriaMetrics eliminated the extra gateway hop and reduced the on-disk metrics footprint from 83.5 GiB to 33 GiB (≈2.5×) for the same metric set (Table 1). Public benchmark reports also indicate that VictoriaMetrics can achieve lower memory and disk usage and faster query latency than Prometheus on production-like workloads [20]. Thanos [21] remains a viable alternative for Prometheus long-retention; we keep it as a design option rather than part of the core stack.

Table 1. Resource footprint and query latency of the metrics backend in our deployment (Prometheus + Pushgateway vs VictoriaMetrics) for the same metric set.

Metric	Prometheus	VictoriaMetrics	Improvement
CPU avg used	0.79 / 3 cores	0.76 / 3 cores	~ same (−3.8%)
Peak RAM used	8.12 GiB	4.5 GiB	1.8× lower (−44.6%)
Disk used (TSDB data)	83.5 GiB	33 GiB	2.5× lower (−60.5%)
Read latency (p50)	70.5 ms	4.3 ms	16.4× faster (−93.9%)
Read latency (p99)	7.0 s	3.6 s	1.94× faster (−48.6%)

Grafana [22] queries this store to visualize long-term trends and manage alerting rules. This stack was chosen to maintain low runtime overhead while enabling long-horizon trend analysis.

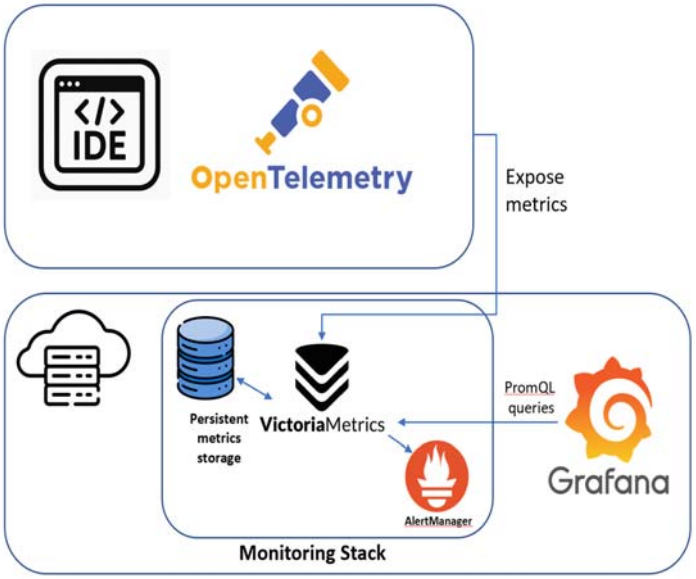

Fig. 1. Observability architecture – OpenTelemetry-instrumented IDE exporting Prometheus-compatible metrics to VictoriaMetrics; Grafana dashboards and alerting query VictoriaMetrics via query language.

3.2 Data Description

To ensure the system handles realistic workloads, we have a diverse corpus of OSS projects. The dataset spans scientific, web, and tooling ecosystems, varying significantly in project size and dependency complexity. Some of the projects from the OSS corpus are listed in Table 2.

Table 2. A subset of OSS projects from our evaluation corpus, spanning small to large codebases.

Project	Source files	Sources size (disk)	Dependency files	Dependencies size (disk)
Transformers	4,992	103 MiB	44,778	445 MiB
Ray	4,726	39.9 MiB	36,663	565 MiB
PyTorch	3,893	52.6 MiB	6,038	76.7 MiB
TensorFlow	3,205	44.8 MiB	6,914	82.4 MiB
Django	2,771	16.5 MiB	3,841	44.6 MiB
NumPy	765	9.89 MiB	11,602	124 MiB
Flask	65	536 KiB	5,143	55.4 MiB
Jedi	330	1.26 MiB	2,903	32.6 MiB

Key characteristics of the dataset include:

- *Volume and Variety:* The corpus includes massive projects like Transformers (4,992 source files) and PyTorch (3,893 source files), alongside smaller web frameworks like Flask (65 source files).
- *Dependency Management*: All measured Python projects are packaged with manifests that pin dependency versions to ensure consistent testing conditions across runs.

- *Environment*: We encapsulated the execution environment in a pinned Docker image that fixes all runtime dependencies, including Python interpreters, Java versions, and build tooling.

3.3 Implementation Details

The observability capabilities focus on detecting relative shifts rather than relying on fragile absolute thresholds, addressing the challenge that modern IDEs are composite systems where generic metrics are often insufficient [1].

- *Instrumentation*: We placed timers and counters around critical request pipelines and the indexing lifecycle in the source IDE code. Since each additional labelset effectively creates a new time series with non-trivial RAM/CPU/storage cost, uncontrolled labeling quickly becomes a scalability bottleneck in metric systems [23]. To control cardinality, we enforced a minimal label set consisting of the scenario, project ID, branch, and build identifier.
- *Alerting Algorithm*: Sliding windows were chosen because performance baselines in IDE workloads are non-stationary: they evolve with ongoing development, corpus changes, and infrastructure updates, making fixed "global" thresholds brittle. A windowed baseline adapts to the most recent stable regime while naturally discounting outdated history, which is important for long-lived products with continuous optimization. Thresholds and window sizes were selected empirically during an initial calibration phase on historical runs to balance sensitivity against false positives; in particular, the *short* window represents the most recent stable period (e.g., on the order of 1–2 weeks of nightly runs), while the *long* window captures a broader baseline (e.g., several weeks) to detect gradual drifts. We implemented a sliding-window approach to detect regressions. For a current metric value x_t, we calculate a robust baseline from the window W (last W points, excluding x_t):

$$\tilde{\mu}_t^{(W)} = \mathrm{median}\{x_{t-W}, \ldots, x_{t-1}\}, \text{ and}$$
$$\tilde{\sigma}_t^{(W)} = 1.4826 \cdot \mathrm{MAD}\{x_{t-W}, \ldots, x_{t-1}\},$$

  where MAD – median absolute deviation. Although the equations below use the median (p50) for readability, the same baseline and alerting logic is applied to other reported percentiles (including p95) as separate time series (percentile-based latency reporting is standard practice for capturing both typical and tail behavior.) [24]
- *Thresholds*: An alert trigger only when both the robust z-score (r_t) and the relative shift (δ_t) exceed defined parameters. This dual-threshold strategy filters noise and highlights statistically significant regressions.
- *Baselines*: To catch gradual drifts, we compare short- and long-window baselines

$$(W_s \ll W_\ell)\text{: } \Delta_t = \tilde{\mu}_t^{(W_s)} - \tilde{\mu}_t^{(W_\ell)}, \text{ and warn when}$$
$$|\Delta_t| \geq \gamma \cdot \max\left(\varepsilon, |\tilde{\mu}_t^{(W_\ell)}|\right)$$

  (e.g., 5–15%). Since baseline metrics are recalculated using a sliding window, older metrics are naturally forgotten. So, for example, a memory spike to 2 GB is evaluated relative to today's 1 GB for a given project, not last year's 3 GB.

3.4 Experimental Setup and Evaluation Metrics

We integrated the observability platform into the CI/CD pipeline [3] to execute two types of workloads: nightly runs for trend establishment on development branches, and manual runs for pre-merge validation.

The evaluation measures the following key indicators while following the established software performance measurement principles [1]:

- *Latency*: We recorded p50 and p95 latency for user-visible actions, including code completion, "go-to-definition," and find usages.
- *Resource Utilization*: The system tracks peak memory usage during indexing and the frequency of Garbage Collection (GC) events per window.
- *Lifecycle Performance:* Specific timers measure project initialization time, initial indexing duration, and "warm reopen" speed.
- *Stability*: We monitored session stability by tracking exception rates and indices saving time.

All metrics are aggregated per run to filter single-execution flakes, ensuring that observed deltas represent reproducible shifts in the IDE's performance profile.

4. Results

This section presents the experimental findings derived from the deployment of the continuous observability platform within a production CI/CD environment. The evaluation assesses the platform's efficacy in three key dimensions: (i) the sensitivity of regression detection using relative sliding windows, (ii) the validation of performance improvements, and (iii) production benefit in terms of engineering costs.

4.1 Detection of Performance Regressions

To evaluate the system's ability to identify stability anomalies, we analyzed the performance of the alerting algorithm based on robust z-scores (r_t) and relative shifts (Δ_t). The results demonstrate that the sliding-window approach successfully filters noise while highlighting statistically significant regressions.

Case 1: Immediate Regression Detection (Latency)

In one prominent case study involving the indexing subsystem, the nightly run detected a severe regression in the time required to save indices to the disk. As illustrated in Fig. 2, the alert rule triggered immediately when the relative shift approached more than 400% and the robust z-score exceeded the threshold ($r_t > 5$).

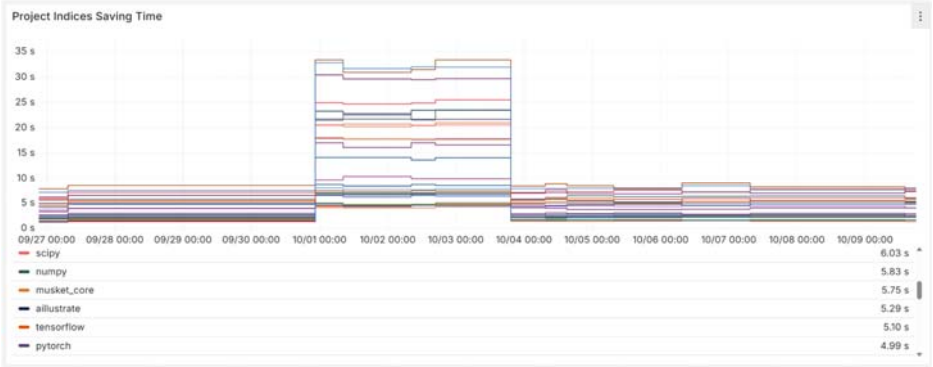

Fig. 2: Peak indices saving time-series with 5-6 times spike and post-fix recovery.

The visualization provided by the platform allowed the engineering team to correlate the spike with a specific commit range. Investigation revealed that a failure in the index cache caused redundant

operations during the save workflow. Following a hot-fix, the subsequent nightly run confirmed that the metric had returned to its baseline, demonstrating the platform's precision in catching step-changes.

Case 2: Gradual Drift Detection (Memory)

The platform also proved robust in detecting gradual resource degradation, which is often difficult to identify through standard threshold-based monitoring. For the PyTorch project, the system tracked a subtle upward drift in memory usage after initialization over several days.

As shown in Fig. 3, the trend rule triggered a warning when the deviation between the short-window and long-window baselines exceeded the 10% boundary, even though the absolute value had not yet reached a critical failure point. This early detection allowed developers to identify a change in the code model that unnecessarily increased character resolution before preheating. Upon rolling back the change, the short-window median realigned with the long-window baseline within two nightly runs.

Fig. 3: Memory usage drift and rollback on PyTorch project.

4.2 Validation of Sustained Improvements

Beyond regression detection, the platform provided valuable data for validating architectural optimizations. We measured the impact of two specific optimizations: lazy type inference on demand and the implementation of an intermediate representation for type structure.

The results indicate a striking improvement in resource efficiency. As apparent from the data, these optimizations resulted in a 2-3 times reduction in peak memory usage and a 30% increase in speed for "warm reopen" scenarios across the corpus. The improvement rules flagged these shifts as positive "green" annotations, and the observability history confirmed that these gains persisted over subsequent releases, validating the stability of the optimization.

4.3 Production Benefit and Engineering Cost

A critical goal of this study was to demonstrate that comprehensive IDE observability is achievable with modest engineering investment. The operational data collected over one year of industrial use corroborates this.

- *Issue Resolution:* The platform successfully surfaced over 40 distinct issues, ranging from memory leaks to tail-latency inflations in code completion and navigation. The integration

of manual runs allowed several regressions to be detected and resolved pre-merge, preventing them from reaching the main development branch.

- *Engineering Effort:* The construction of the initial platform required approximately 1.5 months (240 person-hours). The ongoing maintenance cost is low, averaging 5–8 person-hours per month for tasks such as upgrades and panel adjustments.
- *Runtime Overhead:* We observed that the instrumentation of the Language Server and core services with OpenTelemetry introduced negligible runtime overhead. Following a disciplined label set (scenario, project ID, branch), we avoided high-cardinality overheads typical for time-series metric systems by not overusing labels and keeping metric cardinality low [23].

These results indicate that the proposed sliding-window methodology and architecture provide a highly effective, low-overhead solution for maintaining the performance stability of complex client-server IDEs.

5. Discussion & Limitations

This study demonstrates that a continuous, reproducible observability platform improves the stability and performance control of client–server IDEs. By instrumenting the Language Server and core services with OpenTelemetry and analyzing trends via sliding-window alerts, the platform successfully surfaced over 40 distinct performance issues in a production environment over one year. These results suggest that metrics such as latency and memory usage under realistic workloads provide more reliable protection against regressions than ad hoc logging or general resource metrics.

Our experience suggests that relative, sliding-window alerting can be more resilient than fixed absolute thresholds in composite client–server IDE workloads. In our deployment, comparing a short-term window against a longer baseline helped detect a 5–6 times spike in index saving time and provided early warning of a gradual memory drift that eventually reached ~25%. Separately, the continuous telemetry pipeline under pinned environments enabled data-driven quantification of architectural optimizations, including an observed ~30% speedup in “warm reopen” scenarios across subsequent runs.

Furthermore, the implementation proved highly cost-effective; the platform required only 240 person-hours to construct and incurred a low maintenance overhead of 5–8 hours per month. This contradicts the assumption that bespoke, high-fidelity monitoring requires prohibitive engineering investment, provided that label cardinality is disciplined and instrumentation is conservative.

The choice of the metrics storage backend is also critical to hardware costs. In our case, switching from Prometheus + PushGateway to VictoriaMetrics reduced memory consumption by approximately 10 times without significant changes to the system.

Inevitably, the reliance on relative sliding windows introduces a trade-off regarding sensitivity; false positives may occur in low-variance series or following legitimate environmental shifts. However, the system mitigates this by enforcing dual thresholds – requiring both a high robust z-score and a relative shift.

Additionally, while our OSS corpus spans diverse project sizes, from Flask to PyTorch, it acts as a representative sample and may not capture every specific enterprise edge case. Nevertheless, the use of pinned Docker environments ensures that observed regressions represent code changes rather than environmental noise.

The implications of this research extend to the broader DevOps lifecycle of enterprise-grade software products with a long lifecycle. Integrating these performance checks into the CI/CD

pipeline enables a "shift-left" approach, where regressions are identified via manual runs before merging. This workflow is portable and can be adapted to other IDEs and languages.

Future work will focus on extending this telemetry model to opt-in, anonymized end-user data, necessitating the development of rigorous privacy safeguards and threat models to handle production telemetry at scale. A/B change validation and plugin-compatibility signalization are natural next steps, as is packaging the pipeline for other IDEs and languages. The broader message is infrastructural: sustained IDE quality demands continuous observation that developers trust and can act upon.

6. Conclusion

This study presented a production-grade, continuous observability platform designed to secure the stability and responsiveness of client–server IDEs throughout long development cycles. By integrating OpenTelemetry with a high-performance time-series database and implementing a robust sliding-window alerting algorithm, we addressed the critical challenge of detecting performance regressions in composite software systems where generic resource metrics often fail. The proposed architecture emphasizes reproducibility through pinned environments and open-source corpus, ensuring that observed metric shifts reflect genuine code changes rather than environmental noise.

Experimental validation over a one-year industrial deployment demonstrated the platform's superior sensitivity and cost-effectiveness. The system successfully surfaced over 40 distinct performance anomalies, including a severe 5-6 times latency regression in index saving and a 25% memory drift, which were identified via relative trend analysis rather than brittle absolute thresholds. Furthermore, the engineering investment proved highly efficient; the platform was constructed with 240 person-hours of effort and maintained with minimal monthly overhead, while the transition to VictoriaMetrics yielded an order-of-magnitude reduction in storage requirements. These results confirm that comprehensive, developer-centric observability is achievable with modest resources when label cardinality and instrumentation are disciplined.

References

[1]. Jain R. K., The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling. New York, NY, USA: Wiley, 1991.

[2]. Beller M., Spruit N., Spinellis D., Zaidman A. On the dichotomy of debugging behavior among programmers, in Proceedings of the 40th International Conference on Software Engineering (ICSE), 2018, pp. 572-583.

[3]. Forsgren N., Humble J., Kim G. Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations. IT Revolution, 2018.

[4]. Erich F. M. A., Amrit C., Daneva M. A qualitative study of DevOps usage in practice, J. Syst. Softw., vol. 126, pp. 41-51, Apr. 2017.

[5]. Teitelbaum T., Reps T. The Cornell Program Synthesizer: A syntax-directed programming environment, Commun. ACM, vol. 24, no. 9, pp. 563-573, Sep. 1981. DOI: 10.1145/358746.358755.

[6]. Reiss S. P. PECAN: Program development systems that support multiple views, IEEE Trans. Softw. Eng., vol. SE-11, no. 3, pp. 276-285, Mar. 1985. DOI: 10.1109/TSE.1985.232211.

[7]. Rask J. K. The Specification Language Server Protocol: A proposal for standardised LSP extensions, Master's thesis, Dept. Comput. Sci., Aalborg Univ., Aalborg, Denmark, 2018. DOI: 10.4204/EPTCS.338.3.

[8]. Language Server Protocol Specification – 3.17, Microsoft, Redmond, WA, USA, 2022. Available at: https://microsoft.github.io/language-server-protocol/, accessed 07.10.2025.

[9]. Rodríguez-Echeverría R. et al. Towards a Language Server Protocol infrastructure for graphical modeling tools, in Proc. 33rd Annu. ACM Symp. Appl. Comput. (SAC), Pau, France, 2018, pp. 370-377.

[10]. Murphy G. C., Kersten M., Findlater L. How are Java software developers using the Eclipse IDE? IEEE Softw., vol. 23, no. 4, pp. 76-83, Jul./Aug. 2006.

[11]. Ko A. J., Myers B. A., Aung H. H. Six learning barriers in end-user programming systems, in Proc. IEEE Symp. Vis. Lang. Hum.-Cent. Comput. (VL/HCC), Rome, Italy, 2004, pp. 199-206.

[12]. Hellendoorn V. J. et al. Global and local learning for code completion, in Proc. 41st Int. Conf. Softw. Eng. (ICSE), Montreal, QC, Canada, 2019, pp. 510-521.
[13]. Svyatkovskiy A., Deng S. K., Fu S., Sundaresan N. IntelliCode Compose: Code generation using Transformer, in Proc. 28th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE), Sacramento, CA, USA, 2020, pp. 1433-1443.
[14]. Vaithilingam P., Zhang T., Glassman E. L. Expectation vs. experience: Evaluating the usability of code generation tools powered by LLMs, in CHI Conf. Hum. Factors Comput. Syst., New Orleans, LA, USA, 2022, Art. no. 332.
[15]. Rodríguez P. et al. Continuous deployment of software-intensive products and services: A systematic mapping study, J. Syst. Softw., vol. 123, pp. 264-291, Jan. 2017.
[16]. Eclipse Foundation, Eclipse RCP Testing Tool (RCPTT), Available at: https://www.eclipse.org/rcptt/, accessed 07.10.2025.
[17]. Eclipse Foundation, Eclipse Jubula, Available at: https://projects.eclipse.org/projects/technology.jubula, accessed 07.10.2025.
[18]. OpenTelemetry is a collection of APIs, SDKs, and tools: main page. Available at: https://opentelemetry.io/, accessed 07.10.2025.
[19]. VictoriaMetrics Observability Stack. Available at: https://victoriametrics.com/, accessed 07.10.2025.
[20]. VictoriaMetrics, “Sustainability – Benchmark Results,” VictoriaMetrics. Available at: https://victoriametrics.com/sustainability/, accessed 07.10.2025.
[21]. Thanos - open source, highly available Prometheus setup with long term storage capabilities. Available at: https://thanos.io/, accessed 07.10.2025.
[22]. Grafana: the open and composable observability platform. Available at: https://grafana.com/, accessed 07.10.2025.
[23]. Prometheus, “Instrumentation,” Prometheus Documentation. Available at: https://prometheus.io/docs/practices/instrumentation/, accessed 07.10.2025.
[24]. Google, “Monitoring Systems with Advanced Analytics,” Site Reliability Engineering Workbook. Available at: https://sre.google/workbook/monitoring/, accessed 07.10.2025.

Информация об авторах / Information about authors

Владислав Игоревич МИРОШНИКОВ – магистр компьютерных наук Санкт-Петербургского государственного университета (факультет математики и компьютерных наук), старший инженер-исследователь команды IDE в Chebyshev Research Center (CRC), с 2021 года работает в области создания инструментов разработки.

Vladislav Igorevich MIROSHNIKOV – Master of Computer Science, St. Petersburg State University (Faculty of Mathematics and Computer Science); Senior Research Engineer in the IDE Team at the Chebyshev Research Center (CRC). Since 2021, focuses on Developer Tools and IDE Infrastructure.

Ольга Игоревна БАЧИЩЕ – аспирант института прикладной информатики университета ИТМО. Инженер-исследователь в команде IDE в исследовательском центре имени Чебышёва (CRC) с 2023 года. Научные интересы: статический анализ программ.

Olga Igorevna BACHISHCHE – postgraduate student at ITMO University's Institute of Applied Computer Science. Research engineer on the IDE team at the Chebyshev Research Center (CRC) since 2023. Research interests: static program analysis.

Илья Александрович КУЗНЕЦОВ – магистрант кафедры системного программирования математико-механического факультета Санкт-Петербургского государственного университета, инженер-исследователь команды IDE в Chebyshev Research Center (CRC) с 2022 года.

Ilya Alexandrovich KUZNETSOV – Master's student at the Department of System Programming, Faculty of Mathematics and Mechanics, Saint Petersburg State University; IDE Team Research Engineer at Chebyshev Research Center (CRC) since 2022.

Александр Андреевич ПЛАТОНОВ – магистр информационных технологий, ведущий инженер команды IDE Чебышёвского исследовательского центра. С 2020 года профессионально занимается направлением наблюдаемости (observability) распределённых программных систем.

Alexander Andreevich PLATONOV – Master of Information Technologies, Lead Engineer of the IDE team at the Chebyshev Research Center. Since 2020, he has been working professionally in the field of observability of distributed software systems.

Николай Владимирович ТРОПИН – ведущий инженер, технический лидер команды IDE Чебышёвского исследовательского центра, закончил Математико-механический факультет СПбГУ. Работает в области создания инструментов разработки с 2013 года.

Nikolay Vladimirovich TROPIN – leading engineer, technical leader of the IDE team at the Chebyshev Research Center, graduated from the Faculty of Mathematics and Mechanics of St. Petersburg State University. He has been working in the field of development tools since 2013.

Дарья Владимировна ВАСИНА – ведущий инженер лаборатории средств разработки облачного ПО в исследовательском центре имени Чебышёва. Окончила факультет компьютерных технологий и управления Университета ИТМО по направлению «Информатика и вычислительная техника». Специализируется на создании инструментов для разработки программного обеспечения с интеграцией технологий искусственного интеллекта.

Darya Vladimirovna VASINA is a Lead Engineer at the Cloud Software Development Tools Laboratory of Chebyshev Research Center. She graduated from the Faculty of Computer Technologies and Control at ITMO University, majoring in Computer Science and Engineering. She specializes in creating software development tools with the integration of artificial intelligence technologies

Дмитрий Владимирович КОЗНОВ – доктор технических наук, профессор кафедры системного программирования Санкт-Петербургского государственного университета, Сфера научных интересов: программная инженерия, модельно-ориентированная разработка программного обеспечения, программные данные, машинное обучение.

Dmitry Vladimirovich KOZNOV – Dr. Sci. (Tech.), Associate Professor, Professor St. Petersburg State University (SPbSU). Research interests: software engineering, model-driven software development, program data, machine learning.