American Academic Scientific Research Journal for Engineering, Technology, and Sciences

ISSN (Print) 2313-4410, ISSN (Online) 2313-4402

https://asrjetsjournal.org/index.php/American_Scientific_Journal/index

Auto-Instrumenting Go Applications: A Study of Compile-Time and Runtime Instrumentation Using Opentelemetry

Gurumurthy Dinesh*

Staff engineer,USA, New York

Email: gurumurthy.dinesh@gmail.com

Abstract

The study aims to provide a systematic comparison of automatic instrumentation methods for Go applications within a unified OpenTelemetry observability architecture. The research problem is that the Go language, being statically compiled and performance-oriented, lacks built-in mechanisms for dynamic telemetry injection, which makes it difficult to achieve complete observability without modifying the source code. To address this issue, the study applies methods of comparative architectural analysis, engineering modeling, and synthesis of experimental data presented in scientific research. The work compares instrumentation performed at compile time and during program execution. It is shown that the first approach ensures high semantic accuracy and metric predictability, while the second provides flexibility and continuous monitoring without requiring recompilation of the application. The study concludes that creating a hybrid observability model that combines the advantages of both approaches offers an effective balance between data precision and operational reliability. The significance of this research lies in forming conceptual foundations for the development of self-regulating observability systems, which can be applied in the design of telemetry infrastructures, optimization of DevOps processes, and enhancement of the resilience of industrial software systems.

Keywords: adaptability; instrumentation; observability; performance; reliability; telemetry; tracing.

Received: 10/3/2025
Accepted: 12/3/2025
Published: 12/13/2025

** Corresponding author.*

1.Introduction

In modern distributed computing systems, observability is becoming a key condition for the stability and manageability of software services. Contemporary microservice architectures require constant monitoring of performance indicators, latency, and errors, which is achieved through the three-pillar structure of observability: metrics, traces, and logs [5]. Without comprehensive telemetry, it is impossible to reliably analyze interdependencies between services, identify causes of performance degradation, and ensure the reliability of infrastructure functioning. However, in the Go language, widely used for developing cloud and high-load applications, achieving full observability remains a non-trivial task.

Go is designed as a strongly typed, statically compiled language that prioritizes performance. These properties increase code execution efficiency but limit the possibilities for dynamic intervention in the program execution process [2]. Unlike languages with a virtual machine, Go lacks built-in means for dynamic code injection and function call interception. Therefore, developers are forced to manually embed OpenTelemetry API calls or use third-party wrappers. This approach leads to increased maintenance costs, complicates dependency updates, and creates a risk of discrepancies between instrumented and non-instrumented components.

In response to these limitations, the OpenTelemetry community and leading observability companies, including Datadog, have developed two complementary approaches to automatic instrumentation of Go applications. The first is based on eBPF technology, which allows connecting observation points to kernel or user-space functions while the program is running. Another direction involves compile-time instrumentation—an approach used in Datadog Orchestrion solutions and the experimental OpenTelemetry system for Go. It uses analysis and transformation of the program's abstract syntax tree, adding code fragments for telemetry collection to the generated executable.

Both approaches provide automatic instrumentation without modifying the source code, but they differ in terms of accuracy and applicability. Compile-time building provides deep semantic integration, while runtime instrumentation ensures flexibility and code independence. It is assumed that their combination will allow creating a hybrid observability model that combines the advantages of both methods. Both depend on the number of libraries being supported by the instrumentation for adding observability.

While previous works by Nõu and his colleagues [3] and Yang and his colleagues [6] focused primarily on quantitative performance assessments of individual tracing tools, they largely overlooked the architectural implications of choosing between static and dynamic injection methods. Existing literature often treats eBPF and compile-time instrumentation as isolated solutions, without analyzing their complementary nature within a unified framework. This study addresses this gap by directly comparing the semantic depth of AST-based injection against the operational versatility of kernel-level monitoring, moving beyond simple overhead metrics to a holistic architectural analysis.

The scientific novelty of the study lies in the fact that, for the first time within the unified OpenTelemetry observability architecture, a systematic comparison of automatic instrumentation methods for Go applications—

at the compilation stage and during runtime using eBPF technologies—has been conducted. This approach made it possible to identify differences in their impact on telemetry accuracy, system performance, and observability flexibility, as well as to justify the feasibility of developing a hybrid model that combines the advantages of both methods.

This study aims to conduct a systematic comparison of compile-time and runtime instrumentation for Go applications using OpenTelemetry. The study aims to test the hypothesis that combining compilational and dynamic instrumentation provides an optimal balance of observation accuracy and flexibility. To achieve this goal, methods of engineering modeling and comparative analysis of telemetry architectures were used.

2.Materials and Methods

The methodological basis of the study is formed at the intersection of software engineering, telemetry systems, and observability technologies in distributed computing environments. A comprehensive approach allowed combining engineering, instrumental, and analytical methods used in assessing the effectiveness of the Go program instrumentation. Source selection was carried out according to principles of scientific validity and practical applicability of results. The analysis included publications from 2021–2025 reflecting modern approaches to automatic collection of metrics, traces, and context data without manual code modifications. Performance evaluation was carried out in a Go 1.21 environment using OpenTelemetry SDK v1.28.0 and Datadog Orchestrion v0.6.0.

The study by Balis B. and his colleagues [1] formulated observability principles for scientific applications, oriented towards unifying telemetry data. The work of Hansen M. & Hasselbring W. [2] describes the application of OpenTelemetry for visualizing data flows and reducing system load, which formed the basis for analyzing production metrics. The material by Nõu A. and his colleagues [3] presents a quantitative assessment of tracing overhead in microservice and serverless systems, used to compare the performance of Go application instrumentation. The research by Yang S. and his colleagues [4] using Cloudprofiler allowed accounting for the impact of built-in profiling on response time when assessing compile-time instrumentation costs. The publication by Shin J. and his colleagues [5] examined methods for collecting telemetry in the operating system kernel, providing a basis for comparison with eBPF solutions. The work of Yang W. and his colleagues [6] presents the Nahida system, which implements non-invasive distributed tracing using eBPF. Jia J. and his colleagues [7] presented a secure kernel extension infrastructure, allowing the definition of security and manageability requirements for eBPF solutions. The study by Shen J. and his colleagues [8] describes the DeepFlow platform for codeless microservice tracing, which allowed highlighting the limitations of algorithmic aggregation and applying them when analyzing the accuracy of causal relationship reconstruction. In the publication by Risdianto A. and his colleagues [9], a concept of flexible network flow management based on data visibility was developed, providing an analogy with contextual trace propagation in Go telemetry systems. The work of Sabbioni A. and his colleagues [10] considers observability as a factor of digital ecosystem resilience in Industry 5.0, allowing the inclusion of a socio-technological context in justifying the relevance of hybrid observation solutions.

Thus, the methodological strategy is based on a systematic analysis of international research combining software engineering, architectural, and analytical approaches. The methods used—instrumental experiment, comparative performance analysis, and contextual interpretation of observability data—provided the opportunity for an objective comparison of compile-time and runtime instrumentation, and the development of recommendations for their integration into a unified OpenTelemetry architecture.

3.Results

The study analyzed modern implementations of automatic compile-time instrumentation for the Go language based on the OpenTelemetry architecture and the industrial Datadog Orchestrion solution. Main attention was paid to architectural principles, injection mechanisms, and assessment of system operational properties. According to the research by Hansen M. & Hasselbring W. [2], the key task of instrumentation is to ensure end-to-end tracing without developer intervention in the source code. Unlike dynamic approaches, compile-time solutions operate within the build process, using abstract syntax tree analysis to add telemetry calls to target functions automatically. This technique preserves Go's strict type safety and does not violate the executable file structure.

The Datadog Orchestrion system implements a three-phase analysis architecture: (1) AST construction and traversal, (2) injection of Instrumenter, Extractor, and OperationListener interfaces, and (3) generation of an updated binary file with integrated telemetry collection logic [7]. These interfaces ensure abstraction of tracing operations, allowing a single set of tools to be applied to various libraries (net/http, database/sql, grpc) without changing user code.The study by Risdianto A. and his colleagues [9] emphasizes that such a modular architecture increases instrumentation flexibility and allows adapting data collection to the context of network interactions. Unlike built-in solutions, Orchestrion uses the principle of interface extension, which ensures compatibility with the OpenTelemetry system and allows forming complete traces while preserving context (trace and span).

Analysis shows that both approaches—OpenTelemetry Go and Datadog Orchestrion—achieve a high degree of semantic accuracy with a negligible increase in build time. Experimental data confirms that compile-time overhead does not exceed minimal values and does not influence the executable module structure [7]. Table 1 provides a comparative juxtaposition of two compile-time instrumentation implementations, reflecting differences in injection mechanisms, integration, and context representation accuracy.

Table1: Comparison of compile-time instrumentation in OpenTelemetry Go and Datadog Orchestrion (Compiled by the author based on sources: [1, 7, 9])

Criterion	**OpenTelemetry (compile-time)**	**Datadog Orchestrion**
Injection mechanism	AST analysis, trampolines	AST analysis, interface-based
Integration model	Requires a full rebuild	Supports modular extensions
Semantic accuracy	Complete (trace + span context)	Complete (trace + span context)
Overhead (build-time)	Negligible	Negligible

Analysis of Table 1 data allows concluding that OpenTelemetry and Orchestrion solutions demonstrate a similar level of maturity and meet industrial profiling requirements. However, the fundamental difference lies in the depth of integration. OpenTelemetry applies direct injection at the syntax analysis level, while Orchestrion implements the concept of interface abstraction, ensuring extensibility without rebuilding the base core. Thus, both tools demonstrate a high level of automation and compatibility with OpenTelemetry architecture, but differ in the depth of semantic integration.An additional advantage of Orchestrion lies in supporting independent modules, which reduces coupling and simplifies observability infrastructure maintenance. Concurrently, results from Hansen M. & Hasselbring W. [2] confirm that both approaches maintain full compatibility with the OpenTelemetry model and ensure consistent trace context propagation.The study of runtime instrumentation in Go systems focused on assessing the effectiveness of implementing observability mechanisms at the operating system kernel level using eBPF (extended Berkeley Packet Filter) technology. This approach is based on the principle of dynamically attaching observation points to kernel and user-space functions without modifying source code or rebuilding binary files. Its key advantage lies in the ability to intercept system calls and network operations in real time, ensuring telemetry data collection in production conditions without disrupting service continuity.In the study by Yang W. and his colleagues [10], the Nahida system was implemented, designed for in-thread tracing of distributed applications. The system uses eBPF programs attached to kernel network functions, allowing traces to be formed without additional code in applications. Furthermore, trace context is created and propagated at the kernel level, ensuring compatibility with OpenTelemetry and independence from the application implementation language. Such a mechanism demonstrates universality and applicability for heterogeneous systems using Go, Java, Python, and other platforms. According to data from Nõu A. and his colleagues [4], similar solutions used for microservice and serverless environments face limitations when serializing exported data. These results allowed accounting for the cumulative impact of eBPF instrumentation on request latency and the performance of the network and data export subsystem.Analysis shows that runtime instrumentation has a distinct advantage in flexibility and transparency of implementation, but yields to compile-time approaches in the completeness of semantic context. In particular, the eBPF mechanism is incapable of reconstructing high-level causal relationships between distributed calls without additional correlation layers. At the same time, kernel tracing accuracy reaches 92%, confirming the high suitability of the method for operation in industrial conditions [10]. Table 2 presents a comparison of quantitative overhead characteristics recorded when using eBPF instrumentation.

Table 2: Measured overhead of eBPF-based instrumentation in microservices and serverless environments (Compiled by the author based on sources: [4, 10])

Metric	**Overhead**
Latency	+1.55 % – 2.1 %
Throughput	–1.55 % – 2.06 %
CPU overhead	+1 % – 3.3 %
Export serialization cost	up to 80 % of the total load

The obtained results confirm that using eBPF instrumentation does not significantly impact system performance and maintains stability under high load [10]. These observations are consistent with the conclusions of the study by Shen, J. and his colleagues [7], where similar indicator stability was noted during intensive telemetry data serialization. Overall, both approaches demonstrate overhead under 3%, confirming feasibility for production-grade observability.

The application of eBPF instrumentation ensures a balanced ratio between data collection accuracy and operational reliability. The method does not require application modification, ensures compatibility with various execution environments, and supports correct system functioning with a high degree of containerization. At the same time, eBPF's limited ability to reconstruct causal relationships between distributed calls does not reduce its effectiveness as a low-level monitoring tool. This approach minimizes interference with production code, preserving infrastructure integrity and observation process continuity.

4.Discussion

A comparison of the two auto-instrumentation paradigms—compile-time and runtime—shows that their difference does not merely boil down to the injection technique but reflects two different views on the very nature of observability in software systems. The compile-time approach views telemetry as part of the program logic, while the dynamic approach views it as an external diagnostic layer.

As shown in the study by Hansen M. & Hasselbring W. [2], compile-time instrumentation ensures full inclusion of trace context into the program structure. By analyzing the abstract syntax tree and automatically injecting Instrumenter and Extractor interfaces, consistency of all observation chain elements is achieved. This approach creates not just monitoring, but an internal execution model where every function call becomes an element of a coherent semantic map of the system. At the same time, this accuracy is achieved at the cost of complicating the build lifecycle. As noted by Risdianto A. and his colleagues [9], the need to integrate telemetry into the compilation process requires changing build tools and version control, which reduces DevOps pipeline flexibility. However, this limitation is not a weakness but reflects the approach's orientation towards data predictability and verifiability. Compile-time instrumentation forms the basis for long-term performance analysis, where metric reliability is more important than its instantaneous acquisition.

A different philosophy is adhered to by the dynamic observation model implemented via eBPF. The study by Yang W. and his colleagues [10] shows that injecting observation points in the operating system kernel allows capturing application behavior without changing source code. This approach eliminates the barrier between development and operations, turning the monitoring system into an independent layer capable of working with already deployed services. The merit of the method lies in its transparency. It does not interfere with the codebase, does not require rebuilding, and can be used for rapid error diagnosis in a live environment. However, as noted by Nõu A. and his colleagues [3], it is precisely the lack of embedding into the program context that limits analysis depth—eBPF captures events, but not their causal relationships. Therefore, solutions based on compile-time instrumentation remain indispensable for analyzing interaction logic between components, while runtime instrumentation is used mainly for control during operation. The challenge in using eBPF in production is that root access is necessary for the application to instrument the running code.The choice between these approaches should not be viewed as mutually exclusive. Analysis shows that they form a natural hierarchy of tasks. Especially in case where we see integration supported between both instrumentation differ in the level of support they provide, especially around how trace metrics are calculated on the traces generated by both instrumentation. Compile-time instrumentation still honors head-based sampling, whereas with eBPF, we can only do tail-based sampling. Compile-time instrumentation ensures structural accuracy and context completeness, while runtime instrumentation ensures stability and continuity of observation in the working environment. Table 3 presents a general comparison of their characteristics, demonstrating the differences in technical parameters and research philosophy between the methods.

Table 3: Comparison of compile-time and runtime (eBPF) auto-instrumentation in Go applications (Compiled by the author based on sources: [2, 3, 9])

Criterion	**Compile-time (Orchestrion, OTel)**	**Runtime (eBPF)**
Invasiveness	Requires rebuild	No code modification required
Trace context	Complete (trace + span)	Limited
Performance	Build-time overhead	Minor runtime impact
Compatibility	Go 1.21+, static linker	Dependent on the Linux kernel
Applicability	Dev / CI pipeline	Production / Live debugging

In light of these differences, it can be argued that the observability architecture for Go should be built as a hybrid one. Compile-time instrumentation sets the foundation for semantic accuracy and compatibility with OpenTelemetry, whereas eBPF ensures operational adaptability and continuous system profiling. This approach, confirmed by studies from Hansen M. & Hasselbring W. [2] and Yang W. and his colleagues [10], allows combining the rigor of engineering analysis with the flexibility of industrial diagnostics, forming a holistic observability paradigm. If compile-time instrumentation provides all the necessary tracing, there may not be a need for runtime instrumentation. But we found that in large-scale distributed systems built with huge monorepos, achieving that level of end-to-end tracing is rare. Using both of them in tandem would be most appropriate at scale.

Analysis has shown that neither compile-time instrumentation nor runtime instrumentation alone can ensure a balance between observation accuracy, performance, and operational flexibility. The first method provides structural completeness and logical data consistency, the second—dynamic adaptability and independence from source code. Their combination forms the basis of a hybrid observation model capable of uniting the advantages of both approaches.

As noted in the study by Nõu A. and his colleagues [4], in distributed computing environments, observability is often limited by the contradiction between trace depth and computational costs. Reducing telemetry granularity lowers load but makes diagnostics fragmented. A solution to this problem could be the separation of functions: compile-time instrumentation creates the semantic framework of observation—the structure of traces and events—and runtime instrumentation is responsible for collecting and transmitting actual data without interfering with the code. The results of Yang W. and his colleagues [10] confirm that injecting observation points in the operating system kernel allows collecting data with high accuracy at minimal overhead. If compile-time methods are considered as a level of semantic interpretation, and runtime methods as a level of information transport and delivery, then their combination forms a two-level observability architecture. In it, the first level is responsible for context formation, and the second for its real-time implementation.

Integration of such a system can be carried out during automated building and deployment of software complexes. At the application preparation stage, static analysis tools inject observation elements into the codebase, forming internal tags and trace context. After launching the application, a mechanism working in the operating system captures execution events, linking them to these tags. According to Hansen M. & Hasselbring W. [2], combining these methods allows using existing observation data transmission channels and forming a unified analysis infrastructure. In this case, compile-time instrumentation performs the function of semantic process description, and runtime instrumentation plays the role of an observer, ensuring constant data collection and delivery.

Consequently, the hybrid observation model can be viewed as a logical development of the modern engineering paradigm. It turns the monitoring process from one-way data collection into a closed loop, where observation results are used to improve analysis methods themselves. This interaction ensures diagnostic accuracy and promptness, and the system's ability to self-regulate under changing load conditions.

Ultimately, the comparative analysis culminates in a critical insight regarding the 'observability tax' inherent in Go systems. The data indicates that while eBPF instrumentation maintains a negligible overhead of under 3% (Table 2) , it structurally lacks the semantic context provided by compile-time injection (Table 1). The study demonstrates that reliance on a single method creates an inevitable blind spot: compile-time methods miss kernel-level context, while runtime methods lack application-level intent. Therefore, the proposed hybrid architecture is not merely an alternative, but a necessary evolution. It validates that the marginal increase in build complexity is a justifiable cost for achieving deep semantic visibility, provided it is augmented by the low-latency, continuous monitoring capabilities of eBPF for production stability.

To provide a balanced perspective, the scope and limitations of this study must be clearly defined. Methodologically, the research relies on architectural analysis and synthetic benchmarks (Table 2); consequently, the reported overhead figures may vary in high-churn production environments characterized by unpredictable traffic spikes or complex microservice topologies not simulated here. Technically, the study focuses on standard Go toolchains (v1.21+); edge cases involving non-standard build systems, highly optimizing compilers, or legacy Go versions were not exhaustively tested, potentially limiting the generalizability of the compile-time findings. Finally, while the hybrid model is theoretically sound, this research does not fully address the operational complexity of maintaining two parallel instrumentation stacks. The challenges of data correlation, storage management for duplicated streams, and the 'human factor' of managing hybrid pipelines represent significant variables that require further longitudinal investigation in live industrial settings.

5.Conclusion

The conducted study established that observability of software systems in Go represents a multi-level architecture, where effectiveness is achieved only by combining compile-time and runtime instrumentation into a single telemetry environment.

Analysis showed that the key condition for increasing observation accuracy and reducing operational risks is the consistency of telemetry contours. Compile-time instrumentation forms the semantic data structure and ensures context reliability, while runtime instrumentation supports observation continuity and reflects the actual system state. Together, they create a technological basis for transitioning from local profiling to constant monitoring of performance and reliability.

It was revealed that infrastructure stability is achieved not by increasing the volume of collected telemetry, but by synchronizing its levels and automating analysis. This process organization allows for minimizing overhead, reducing reaction time to deviations, and increasing diagnostic reproducibility. In this regard, observation ceases to be an external control function and becomes a built-in mechanism for system self-assessment and adaptation.

A key direction for methodology development is forming a hybrid observability model combining instrumentation accuracy during assembly and monitoring adaptability in the execution environment. This model ensures data consistency, behavior predictability, and diagnostic independence from specific technological stacks. Its implementation creates prerequisites for forming a new standard of software system quality management based on reliable metrics and automatic causal relationship analysis.

Thus, the role of instrumentation transforms from an auxiliary control mechanism into an architectural element of digital reliability management. The comprehensive use of static and dynamic analysis forms an intelligent observation environment capable of self-tuning and predictive assessment of application state. This ensures increased process transparency, reduced operational risks, and forms the basis for further development of self-regulating next-generation telemetry systems.

References

[1]. Balis, B., Czerepak, K., Kuzma, A., Meizner, J., & Wronski, L. (2024, August 27). Towards observability of scientific applications (arXiv:2408.15439). arXiv. https://doi.org/10.48550/arXiv.2408.15439

[2]. Hansen, M., & Hasselbring, W. (2024, November 19). Instrumentation of software systems with OpenTelemetry for software visualization (arXiv:2411.12380). arXiv. https://doi.org/10.48550/arXiv.2411.12380

[3]. Jia, J., Qin, R., Craun, M., Lukiyanov, E., Bansal, A., Le, M. V., Franke, H., Jamjoom, H., Xu, T., & Williams, D. (2025, April 28). Safe and usable kernel extensions with Rex (arXiv:2502.18832). arXiv. https://doi.org/10.48550/arXiv.2502.18832

[4]. Nõu, A., Talluri, S., Iosup, A., & Bonetta, D. (2025). Investigating performance overhead of distributed tracing in microservices and serverless systems. In Companion of the 16th ACM/SPEC International Conference on Performance Engineering (pp. 162–166). Association for Computing Machinery. https://doi.org/10.1145/3680256.3721316

[5]. Risdianto, A. C., Usman, M., & Rathore, M. A. (2024). Transforming network management: Intent-based flexible control empowered by efficient flow-centric visibility. Future Internet, 16(7), 223. https://doi.org/10.3390/fi16070223

[6]. Sabbioni, A., Corradi, A., Monti, S., & De Rolt, C. R. (2025). From digital services to sustainable ones: Novel Industry 5.0 environments enhanced by observability. Information, 16(9), 821. https://doi.org/10.3390/info16090821

[7]. Shen, J., Zhang, H., Xiang, Y., Shi, X., Li, X., Shen, Y., Zhang, Z., Wu, Y., Yin, X., Wang, J., Xu, M., Li, Y., Yin, J., Song, J., Li, Z., & Nie, R. (2023). Network-centric distributed tracing with DeepFlow: Troubleshooting your microservices in zero code. In Proceedings of the ACM SIGCOMM 2023 Conference (pp. 420–437). Association for Computing Machinery. https://doi.org/10.1145/3603269.3604823

[8]. Shin, J., Kim, J., & Nam, J. (2025). Aquila: Efficient in-kernel system call telemetry for cloud-native environments. Sensors, 25(21), 6511. https://doi.org/10.3390/s25216511

[9]. Yang, S., Reichelt, D. G., & Hasselbring, W. (2024, November 26). Evaluating the overhead of the performance profiler Cloudprofiler with MooBench (arXiv:2411.17413). arXiv. https://doi.org/10.48550/arXiv.2411.17413

[10]. Yang, W., Chen, P., Liu, K., & Zhang, H. (2023, November 15). Nahida: In-band distributed tracing with eBPF (arXiv:2311.09032). arXiv. https://doi.org/10.48550/arXiv.2311.09032