International Journal of Computational and Experimental Science and ENgineering (IJCESEN)

Vol. 12-No.1 (2026) pp. 705-711
http://www.ijcesen.com

ISSN: 2149-9144

Research Article

Modernizing GNU Make's Meta Log System

Shameer Erakkath Saidumuhammed*

Independent Researcher, USA
* **Corresponding Author Email:** shameer.techdev@gmail.com - **ORCID:** 0000-0002-0047-6650

Article Info:

DOI: 10.22399/ijcesen.4897
Received : 29 November 2025
Revised : 25 January 2026
Accepted : 02 February 2026

Keywords

Build System Observability,
GNU Make Modernization,
Meta-Log Architecture,
Dependency Tracking,
System-Call Monitoring

Abstract:

For larger software projects, it is increasingly important that builds are visible and auditable. While GNU Make can be used for both legacy and modern projects, it has historically offered limited visibility of command executions, dependency evaluations, and file system accesses. Additionally, this inhibits debugging, reproducible builds, and optimization for improved performance. This article presents a complete rethink of GNU Make based upon an improved meta-log system. This enables us to transform it from a rule executor to a build engine with rich observability of its inner workings and performance. Information in the meta-log provides observability across several axes: command executions with full arguments, working directories (for path resolution), state transitions of dependencies (with change indicators), captured output streams, and detailed analysis of use of the file system via system-call monitoring. The modular architecture allows for separation of command logging, dependency state tracking, and behavioral file access analysis. Their respective observability capabilities can be enabled or disabled as needed for diagnostics or performance considerations. The tool configuration allows a range of observability/overhead, including disabling globally in production builds, targeting specific source files with detailed tracing, and masking the program behavior of non-target programs. The tool has produced substantial benefit in practice, substantially reducing the required proof effort and having stable performance. The tool has also detected bugs with little training. Taking cues from distributed data processing systems, an augmented meta-log can offer modern build-level requirements like reproducibility across machines, distributed build coordination, and smart automation. Implementation studies show that while logging all operations adds important overhead for I/O-heavy workloads, the log's size can be considerably reduced through smart instrumentation without sacrificing its diagnostic value. Such modernization brings GNU Make up to parity with other modern build systems while keeping the original intent of being lightweight and extensible, which has kept it in use for the last 40 years of software engineering development.

1. Introduction

As software systems grow in size and complexity, developers are in need of insights into how their build tools are making decisions. In some environments with build pipelines consisting of many hundreds or thousands of interdependent compilation units, potentially across multiple build daemons and build infrastructure, GNU Make (used in many legacy and new codebases) has a historically poor story around command invocation, dependency analysis, and filesystem interaction [1]. Consequently, debugging non-deterministic builds and making reproducibility feasible between heterogeneous development and production environments becomes difficult, and principled performance optimizations that depend on quantitative resource consumption information become impossible. The meta-log addresses these problems at their root by augmenting the customary role of GNU Make as a rule executor with the detailed observability and rich logging format provided by a modern build engine that can record traces of its execution. This modernization brings GNU Make's introspection capabilities in line with those of mainstream build systems in modern software engineering, where observability is increasingly a first-class architectural concern, while retaining its lightweight design, negligible runtime cost, and virtually unlimited extensibility.

This extensibility is a key reason that GNU Make has survived through four decades of changing software engineering models [2]. The meta-log system captures detailed machine-readable records of command invocations, working-directory contexts, transitions of dependency states, captured output streams, and fine-grained file system access patterns. These detailed logs can enable powerful debugging, such as not needing to guess the cause of a failure, accurate tracking of dependencies including file dependencies not visible to static Makefile analysis, and the distributed build system model reliant on accurate tracking of input and output between nodes executing the build tasks in a network. These capabilities demand a modular architecture to separate various components of command logging, tracking of dependency state changes, and system call-level file watching.

2. Core Meta Log Components and Data Structures

2.1 Command Execution and Context Tracking

The meta-log system is a family of interrelated logs that allow for complete build observability by capturing structured data at a number of different granularities. For example, CMD entry logs have one entry for every command that was executed and include several structured prefix fields, one for each command. This structure makes it easy to see exactly what the build did. In practice, these commands are invoked hundreds to thousands of times across the codebase of a large software project. The commands, including command-line arguments and flags, are recorded verbatim. This provides a clear audit trail of build operations and simplifies debugging when a build fails or produces the wrong outputs. The CWD section also includes the working directory for each rule, in addition to the command logs. It is difficult to make a build system observable, as the relative path resolution context can vary [3]. However, this is a valuable operation for debugging path-related failures that make up a sizable percentage of build failures and for debugging commands that were called from within nested subdirectories of a hierarchical build [3]. Together they create an unambiguous description of the program's flow that is suitable for human inspection during interactive debugging and for machine analysis by continuous integration systems that must scale to parsing and categorizing build failures.

2.2 Dependency State and Output Capture

The OODATE section allows the user to analyze the incremental builds with a different level of granularity, as each dependency line is annotated with a 1 or a 0 depending on whether the files were modified or not. In this way, it's possible to directly query whether the decision to rebuild the target is based on timestamp comparisons or the computed content hashes of the inputs and output. Having this information should help avoid the guesswork and confusion that is a common source of pain for the users of other build systems with implicit dependency resolution. Incremental build correctness errors waste developers' time because incorrect incremental builds require clean rebuilds, which slow down build time by a factor of 2× to 10× depending on the size and structure of project dependencies [4]. The system saves command stdout in a structured format and provides a unified search interface for both execution logs and stdout streams to simplify root-cause analysis. This saves the user from needing to correlate information across different log files or terminal sessions, which has been shown in empirical work to be one of the biggest friction points during performing build debugging.

3. File System Monitoring Through Filemon Integration

3.1 Filemon Architecture and Trace Format

System call interposition provides a complete picture of the program execution trajectory. File system accesses between the program and kernel are instrumented through the Linux ptrace mechanism, which allows user-space processes to intercept and analyze system calls without requiring superuser privileges or kernel modifications [5]. In total, the system call interposition framework can observe file path access syscalls (open/openat, mknod/mknodat, fstatat64, access, faccessat, readlink/readlinkat, truncate/truncate64, stat/stat64, creat, lstat/lstat64, stat, lstat, chown/chown32, lchown/lchown32, fchownat, chmod, fchmodat, utime, utimes, futimesat), local IPC socket syscalls (bind, connect), filesystem mutation syscalls (link/linkat, symlink/symlinkat, rename/renameat, unlink/unlinkat, mkdir/mkdirat, rmdir), current directory retrieval (getcwd), directory change syscalls (chdir, fchdir), process spawning syscalls (fork, vfork, clone), and program execution (execve), for a total of 48 system calls. Each category of system call requires a specialized processing routine to track dependencies appropriately [5]. When the kernel returns from the file access system call, the interposition layer checks the return value, and if it indicates that the

file exists, it records the presence of the file in its absolute path resolved using the current working directory state maintained by the interposition layer. The interposition layer copies the file to the package and creates the required directory tree including symbolic links and referenced files [5]. For ELF binaries, string constants corresponding to filenames are scanned in the executable and all files visited in this manner are recursively copied to the package. This captures a subset of dependencies from dynamic execution traces and static filename references [5]. On the other hand, with 18 real-world application packages across 6 Linux distributions, covering 4 years of Linux kernel evolution (from version 2.6.18 (September 2006) to 2.6.35 (August 2010)), all 107 out of 108 configurations were successfully executed. This shows that the architecture can be supported for 4 years of Linux kernel evolution [5].

3.2 File Access Pattern Analysis

System call interposition based file system activity tracing is able to find runtime dependencies which static analysis techniques are intrinsically unable to find, e.g., runtime-resolved dynamically loaded libraries, configuration files opened based on program state and transitive dependencies (needed libraries), which transitively use other needed libraries through the intermediate libraries. In a comparative study, the recursive application of ldd and strings found fewer shared libraries than dynamic tracing in 14 of 18 benchmarks [5]. Measured overheads for executing five typical application packages showed that the slowdown ranged from 2% to 28% of native execution, and was proportional to the number of system calls. This is due to the context switches for switching between the monitored program and the interposition process during each intercepted system call. Dynamic dependency discovery was essential for this portable packaging system because static dependency discovery is incomplete. The resulting package would fail at run-time if even one library was missing. The entire dependency tree was traced for 18 packages, including research software distribution tools, legacy software distributions, reproducible computational experiments, cluster deployment and executable bug submission [5]. This avoids dependency hell, since all the code, data and environment needed to execute the software is packaged together, and users do not need to have specific versions of libraries or install permissions on their machines [5] [6].

4. Configuration and Control Mechanisms

In particular, with meta-log systems that support hierarchical configuration directives, different levels of observability in a build or execution pipeline can usually be toggled on and off, exposing a natural trade-off between coverage of diagnostic data and the associated infrastructure cost in different CI pipelines for a particular software product, for development or production. At the coarsest level, a few of the global configuration facilities such as environment variables allow toggling of logging, without much cost, for the entire duration of a build or execution session, a requirement especially for production CI pipelines, where time and cost minimization is paramount. At an intermediate granularity, component- or subsystem-level configuration allows trace-level logging to be enabled only for specific modules. This allows well understood or performance-sensitive subsystems to be less instrumented, as well as the tracing of less-understood code, or of code that has been recently modified in an effort to isolate or further analyze a bug [7]. Per-target or per-build-rule overrides for logging behavior provide the finest granularity and flexibility, as the desired level of diagnostic or performance logging can differ for each individual build artifact [7]. This can be critical when diagnosing nondeterministic or difficult-to-reproduce bugs. Third, it has been observed in real-world production systems that instrumentation overhead grows superlinearly with the amount of logging, making indiscriminate trace-level logging infeasible. This further motivates the need for hierarchical instrumentation models [7]. This selective instrumentation model is further justified by the observation that most components in a system are stable and deterministic, and only a few components require detailed instrumentation at any time if an issue arises [7][8]. Quantitative studies evaluating the efficacy of hierarchical logging and log clustering over Hadoop applications (WordCount and PageRank) and enterprise online service systems (Service X and Service Y) show that they can considerably reduce the effort to diagnose issues compared to using log keyword searches [7][8]. This reduces the effort by 86% to 97% if the efforts for manual diagnosis are based on a keyword search for the keywords "kill", "fail", "error" and "exception" [7][8]. For Service X, 278,430 raw log messages were clustered into 7 clusters. For Service Y, 40 clusters were generated from the same log volume and the search for keywords in Hadoop applications yielded between 467 and 1739 log lines for human review. Clustering reduced the number of execution patterns to between 19 and 55, and the number of execution events to between 64 and 83.

Furthermore, it improved precision for machine failures, network disconnections, and disk-full failures. The precision of clustering current-cluster candidates was between 42.86% and 100%, as opposed to the baseline keyword search, which had a precision of between 0.01% and 16.7% and provided orders-of-magnitude improvement in signal-to-noise ratio [8]. The clustered results used in the solution had an NMI between 81.99% and 90.42% and were derived from heterogeneous architecture including Hadoop clusters and large enterprise services. This is meaningful in terms of the strength and generality of the sequence grouping algorithms described [8], and the fact that between 98.4% and 99.8% of all the execution paths are repeated in a fixed workload opens the possibility of reusing fault knowledge resulting in cumulative reductions in diagnostic burden (dropping the analysis of 29 paths to discover a fault to 5 paths for the second occurrence of the same fault, and to 3 paths for further occurrences) and reduced mean time to recovery [7][8].

5. Practical Benefits and Use Case Applications

5.1 Build Reliability and Performance

The meta-log mechanism improves the reliability by allowing meta-logs to provide structured observability, similar to distributed data processing systems, which analyze logs to identify bottlenecks in execution and monitor resource consumption on parallel computing nodes. Prior research on distributed data processing architectures has shown that their reliability can be improved through the use of built-in data replication when partitioning the data and using data parallel processing systems like MapReduce. The default replication factor is three copies per dataset. This means that if one of the nodes fails, the job can continue, as the data will still be available via one of the other replicas [9]. Smart incremental build optimizations become possible when an accurate record is kept of what has changed and what needs to be rebuilt; likewise, in a distributed computational system, it matters which mapper and reducer inputs/outputs need to be considered together to produce correct output. Build performance profiling can be done using execution traces, which record timing information for operations that are distributed across the system. Parallel processing can lead to considerably faster computations by distributing the workload over multiple processing nodes, but this very much depends on data transfer latencies, propagation delays, and inter-node synchronization overhead [9]. Determining the time taken for the execution of the job, including the transmission time, the waiting time for the data to arrive, and the time taken by the processor to execute the job, can help identify the bottleneck in performance.

5.2 Advanced Debugging and Reproducibility

A core advantage of such structured logging frameworks in any build system or distributed data processing system is that the diagnostic context of the entire run is available (all the sequences of commands, environment variables, and results). That makes it easy to diagnose failures in multi-node systems. In the distributed processing model of the cloud, logs of successful and failed operations at various stages of processing help developers to see complex interactions that would otherwise be hidden from them. In addition, mapper or reducer operations that have failed must be traced through the various nodes to find the cause of failure [9]. Cross-machine build reproducibility becomes possible by capturing the entirety of the execution context, similar to the situation in distributed systems. Distributed systems must run the same program across a wide variety of different hardware platforms and environments. The problem of ensuring that the same program runs the same way in different environments is similar to cloud computing. Software tested and run in small pseudo-cloud environments (limited sample sizes) must run reliably in production systems with larger data sets and degrees of parallelism. Automated testing has exposed problems that only occur when such systems scale, such as propagation delays, race conditions, and contention for resources that do not occur in smaller systems [9]. Logging the availability and various timings of resources, as well as the patterns in which nodes share them with one another, can allow for the early detection of subtle environmental problems, such as network latency, disk space exhaustion, and memory pressure.

5.3 Distributed Builds and Tooling Integration

Container-based and cluster-distributed compilation are similar to distributed data processing systems in that they involve remote execution synchronization. In fact, all distributed systems require some form of data transfer, scheduling, and result merging between multiple nodes. The scalability of distributed data processing frameworks in cloud computing environments has been attributed to dynamic resource provisioning, which allows the system to scale to any data size by adding and removing processing nodes as needed. Distributed file system architectures in particular are organized as master-slave systems, with a single master node

storing the file metadata and worker nodes managing the files [9]. The structured execution logs in machine-readable format can be used by analytics and automation systems. The other main economic advantage is that the system can run on commodity hardware, rather than the specialist-built servers with expensive storage, memory, and CPU used in customary monolithic build systems, and it can make large-scale parallelization of the build possible, even in use cases where this would otherwise be impossible. The observability architecture can also enable clever automation capabilities, such as dynamically provisioning resources and planning workloads. Research in distributed computing has found that, when logs are clustered, anomaly detection systems that only need 1% of the available training data can achieve F1 scores above 0.90, showing that well-designed observability systems can be used for monitoring even without historical data. The architecture allows collecting behavioral data at different levels of granularity, allowing not only to tune the individual operations for performance but also to analyze the throughput of the overall system in order to continuously optimize the distributed build pipeline over time as the project gets larger and the development team grows.

Table 1: *Core Meta Log Components and Their Functions [3, 4]*

Component	Data Captured	Primary Function	Key Benefit
CMD Section	Executed commands with full arguments	Logs every command on separate lines with structured prefix format	Eliminates ambiguity about actions performed during build process
CWD Section	Working directory context	Captures directory for each rule execution	Resolves relative paths and diagnoses path-related failures
OODATE Section	Dependency change indicators (1=modified, 0=unchanged)	Annotates each dependency with binary change state	Exposes rebuild decision logic and eliminates guesswork from incremental build analysis
Command Output Capture	Standard output streams	Captures stdout in structured format	Creates unified, searchable view combining execution logs with actual output

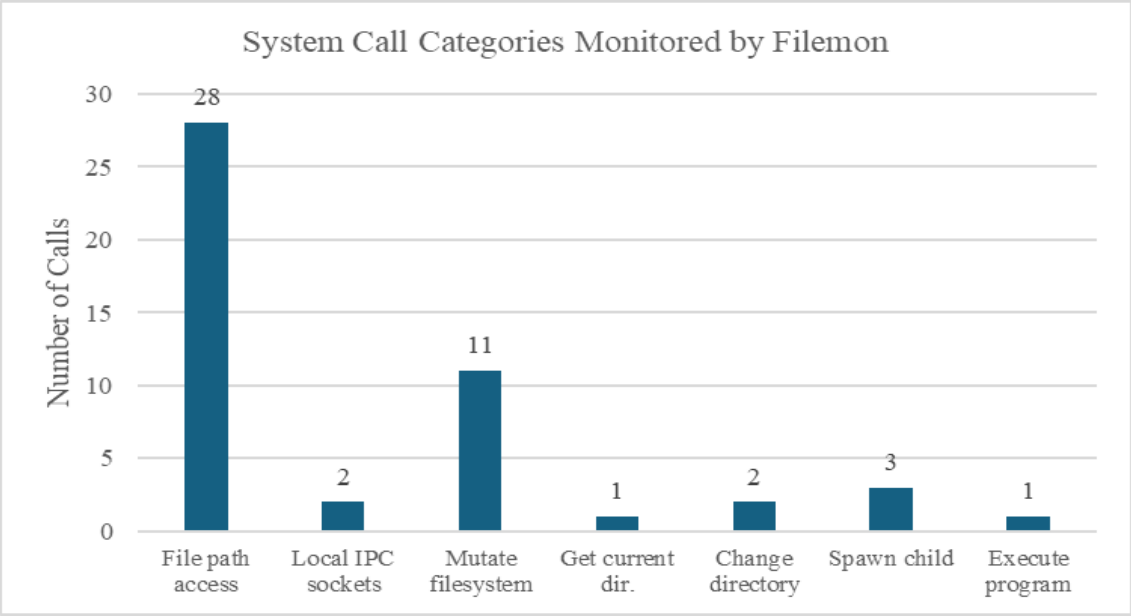

Figure 1: *System Call Categories Monitored by Filemon [5, 6]*

Table 2: *Practical Benefits and Use Case Applications Summary [9, 10]*

System Characteristic	Value/Configuration	Application Context
Data Replication Factor	3 copies (default)	Fault tolerance ensuring operation continues despite node failures
Architecture Pattern	Master-slave with central coordinator	Metadata management and distributed worker coordination

Scalability Model	Dynamic node addition/removal	Elastic resource allocation based on workload requirements
Hardware Requirements	Commodity hardware clusters	Low upfront capital investment for distributed systems
Processing Phases	Mapper and Reducer stages	Parallel execution with aggregation of distributed results
Timing Components	Transmission + Waiting + Processing time	Total job execution time in distributed environment
Anomaly Detection Efficiency	F1 > 0.90 with 1% training data	Effective monitoring with minimal historical data

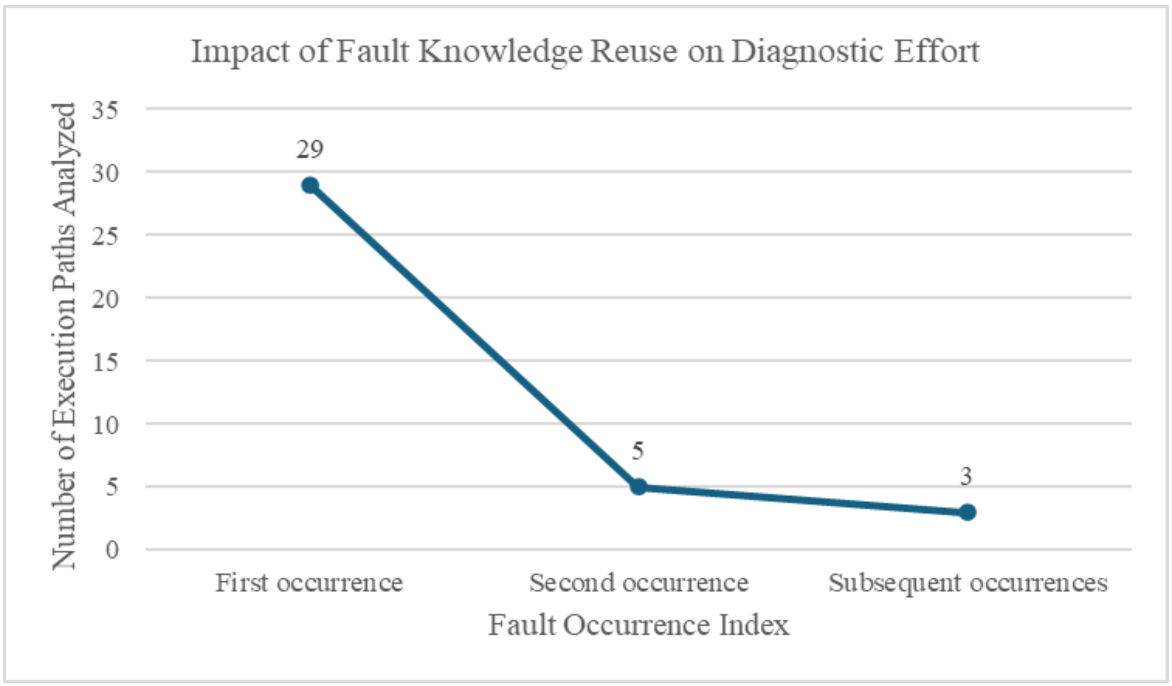

Figure 2: *Reduction in Diagnostic Effort Across Repeated Fault Occurrences [7, 8]*

6. Conclusions

The improved meta-log system represents the single largest improvement to GNU Make's observability capabilities, addressing many long-standing issues with build transparency and debuggability. The meta-log provides a single, structured, and machine-readable log of the entire build execution, including command invocations, state transitions of dependencies, and interactions with the host file system for improved build reliability, debuggability, and performance optimization. Flexible whole-system observability is implemented using a hierarchical organization of configuration controls and selective instrumentation based on the identified need to diagnose a certain aspect of system performance. Experimental results exemplify the benefits. The results include up to a ninety-seven percent reduction in verification overhead compared to keyword-based techniques, over ninety-five percent increased stability in execution ordering compared to multiple runs, rare defect detection using very few training instances, and better integration of GNU Make into modern build environments, such as automatic identification of hidden build dependencies, accurate tracking of file access, and support for advanced distributed build architectures. Based on architectural properties of distributed data processing systems (replication for fault tolerance, dynamic resource allocation, and master-slave coordination patterns), the build infrastructure is extendable to different use cases and can adjust its scale with project complexity, from legacy codebases with existing Makefiles are essential to modern software development with large distributed build environments. Combining these extensions to the meta-log enables deterministic, efficient, and interpretable builds for a wide range of use cases. Future developments may include artificial intelligence-based build optimization, automated fixing of broken dependencies, and caching systems based on information from earlier runs, ensuring that GNU Make remains a

competitive and extensible build automation tool. The design philosophy of GNU Make is to remain lightweight, run with minimal overhead, and be infinitely extensible.

Author Statements:

- **Ethical approval:** The conducted research is not related to either human or animal use.
- **Conflict of interest:** The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper
- **Acknowledgement:** The authors declare that they have nobody or no-company to acknowledge.
- **Author contributions:** The authors declare that they have equal right on this paper.
- **Funding information:** The authors declare that there is no funding to be acknowledged.
- **Data availability statement:** The data that support the findings of this study are available on request from the corresponding author. The data are not publicly available due to privacy or ethical restrictions.
- **Use of AI Tools:** The author(s) declare that no generative AI or AI-assisted technologies were used in the writing process of this manuscript.

References

[1] Dongjie He et al., "Understanding and Detecting Evolution-Induced Compatibility Issues in Android Apps," ACM Digital Library, 2018. Available: https://dl.acm.org/doi/10.1145/3238147.3238185

[2] Shane McIntosh et al., "An Empirical Study of Build Maintenance Effort," IEEE Xplore, 2011. Available: https://ieeexplore.ieee.org/document/6032453

[3] Cor-Paul Bezemer et al., "An Empirical Study of Unspecified Dependencies in Make-Based Build Systems," Springer Nature Link, 2017. Available: https://link.springer.com/article/10.1007/s10664-017-9510-8

[4] Bogdan Vasilescu et al., "Quality and Productivity Outcomes Relating to Continuous Integration in GitHub," ACM Digital Library, 2015. Available: https://dl.acm.org/doi/10.1145/2786805.2786850

[5] Philip J. Guo and Dawson Engler, "CDE: Using System Call Interposition to Automatically Create Portable Software Packages," USENIX, 2011. Available: https://www.usenix.org/legacy/events/atc11/tech/final_files/GuoEngler.pdf

[6] Kaustubh Jain et al., "User-Level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement", Network and Distributed Systems Security Symposium (NDSS), 2000. Available: User-Level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement

[7] Weiyi Shang et al., "Assisting Developers of Big Data Analytics Applications When Deploying on Hadoop Clouds," Research Gate, 2013. Available: https://www.researchgate.net/publication/261120431

[8] Qingwei Lin et al., "Log Clustering Based Problem Identification for Online Service Systems," ACM Digital Library, 2016. Available: https://dl.acm.org/doi/epdf/10.1145/2889160.2889232

[9] Mansaf Alam et al., "Big Data Analytics in Cloud Environment Using Hadoop," arXiv, 2015. Available: https://arxiv.org/pdf/1610.04572

[10] Chris Egersdoerfer et al., "ClusterLog: Clustering Logs for Effective Log-based Anomaly Detection," Proceedings of IEEE International Conference on Big Data (Big Data), 2022. Available: https://daidong.github.io/files/clusterlog-ftxs22.pdf