10.48047/jocaaa.2026.35.01.90

Ensuring Deterministic Outputs in Distributed Build Systems Through Runtime-Based Verification

Shameer Erakkath Saidumuhammed
Independent Researcher, USA

Received: 29.01.2026 Revised: 02.02.2026

Abstract

Distributed build systems are used for building large codebases, which generate many files. These can be compiled more efficiently in parallel. Builds on single machines cannot scale to larger software projects. Spreading compilation across multiple machines can improve performance. However, achieving this consistency and reproducibility for different builds poses an important engineering challenge, as customary approaches require predicting output files from tool commands and heuristics. Such prediction mechanisms do not scale well, nor do they support custom toolchains or compiling behavior. A runtime-based approach solves the issue of determinism in distributed build systems by recording real file operations during task execution using filesystem call tracing. Size-based verification protocols ensure a complete synchronization of the output before the dependent tasks are invoked. This article is independent of the tools being used to prevent problems with partial file reads and lowers the risk of race conditions between producer and consumer tasks from breaking the build. Systematic techniques to recover from dependency failures due to filesystem propagation delays at low performance overhead have been developed. To adapt these best-effort optimization techniques into correct execution models, distributed builds rely on observation and verification protocols.

Keywords: Distributed Build Systems, Output Determinism, System Call Tracing, Filesystem Synchronization, Runtime Verification, Flaky Build Prevention

I. Introduction

Modern software builds are distributed to address the compute limitations associated with the growing size of codebases, as single-node builds fail to keep up with the demands of modern developers. The multicore era has changed the shape of the software systems. With parallelism in hardware, build infrastructure also needs to adopt parallelism [1]. It becomes necessary to distribute build tasks across multiple computers.

In distributed environments, the build may fail not because of compile errors but because of partial visibility of files, leading to intermittent build failures and corrupted builds due to incorrect artifacts. Flaky test root cause analysis shows the cause of non-deterministic test failures to be environment and async wait conditions [2]. Flaky tests can be particularly difficult to diagnose since their root cause is usually distributed between multiple components.

The main requirement is that dependent tasks see the final, complete output of their dependencies. This is guaranteed implicitly for local builds by filesystem semantics. Distributed systems must process synchronization requirements for task executions and output consumptions, since they are done as separate events. Network latency causes fluctuations in when files become visible, and shared storage systems may behave differently.

10.48047/jocaaa.2026.35.01.90

Unspecified dependencies are a common source of build failures. This is particularly an issue for make-based build systems that have implicit dependencies [3], although output files may also have dependencies on files not given in the build specification. Those hidden dependencies cause builds to fail. This problem is compounded for distributed builds where ordering guarantees are lost.

This article employs a runtime observation methodology to solve the determinism problem, discovering filesystem outputs as it observes the filesystem at execution time. The contribution consists of a discovery component based on system calls and a verification protocol attaining full synchronization. These failure recovery strategies guarantee the correctness of the builds while maintaining scalability and can adapt to changing build requirements without manual adjustment.

II. Related Work and Technical Framework

Previous research into distributed build systems has largely focused on performance optimizations and task scheduling approaches; some work has focused on scalable tools, particularly output modeling and heuristics. These prediction-based mechanisms have to be constantly updated as compilers evolve. Static prediction models are also incompatible with custom-built utilities and scripts. There is a gap between determinism and having no tool knowledge.

The article also considers runtime-based output discovery as an alternative to prediction-based handling. For that, filesystem system call tracing is utilized to find out which file operations were invoked by remote tasks at runtime. Open, write, rename, unlink and close operations are logged with the ID of the created or modified file and the absolute path and size of the file when the task is done. Size-checking protocols can delay continuation until all runs are fully synchronized, since output files are in shared storage, and their sizes are verifiable from remote execution nodes, while providing upper bounds for how long they must wait.

The main use is for generality across compilers, linkers, code generators, and custom tools, without requiring tool-specific configurations or modifications to discover correct outputs. Resilience to failure is achieved by remote retry, fallback to local execution, and feedback to the scheduler to avoid unstable nodes.

III. Output Non-Determinism in Distributed Environments

A. Failure Mode Classification

Distributed build execution decouples execution from transport and reading from outputs. The build tools may also not have accurate metadata for output files or may write to storage that has not yet been synchronized. Tools may produce implicit outputs not declared in the build metadata. The output location and other properties depend on the toolchain configuration and temporary files created while compiling.

When compilers and other tools change, so must the build system. Designing for evolutionary changes in the context of distributed systems architectures is also important [4]. Static configuration cannot anticipate all outputs, but dynamic plans can adapt to changing requirements. Indeed, the output formats and output directories vary, and build systems that do not handle that amass technical debt.

Implicit output generation is a particularly difficult failure mode, where not only object files are implicitly generated, but also dependency files, additional code in coverage tools, and debug info written in separate files from executable code. These auxiliary outputs are often not declared in the build specification, and tasks that require them will fail if not provided.

Toolchain flags also contribute to variability: different object files are produced by different optimization levels, a debug build produces different files than a release build, and cross-compilation leads to different

10.48047/jocaaa.2026.35.01.90

paths. To achieve determinism, build systems have to reflect these variations. As the number of configuration options increases, this becomes more difficult.

B. Prediction-Based Limitations

Most other distributed build systems provide deterministic builds by parsing command-line parameters to predict outputs. They require tool- or language-specific models to complement command parsing. Constant maintenance is required as compilers evolve. Static models cannot support utilities and scripts other than builds.

Their main weakness is that the models predict what should happen, and actual behavior can deviate for many reasons: compilers unpredictably change output, and environment variables can change behavior of tools. Differences in the two platforms cause differing results from the tools, and prediction accuracy also decreases over time.

Maintenance effort scales with the number of tools in the toolchain. Each tool requires analysis and modeling. Tools may become obsolete, and it is difficult to achieve completeness when predicting systems with combinatorial complexity, while organizations may incur rising costs for updating.

Unspecified dependencies further weaken prediction approaches [3]. For example, build specifications often do not specify dependencies between tasks. Since predictions are not aware of undeclared relationships, these failures occur unpredictably based on order. Debugging requires enumerating the actual file access patterns.

Failure Mode	Description	Impact on Build Process
Premature Output Visibility	Output files appear in shared storage before complete synchronization	Dependent tasks read incomplete artifacts
Implicit Output Generation	Tools generate auxiliary files not declared in build metadata	Missing outputs cause downstream failures
Variable Output Locations	Toolchain flags alter output paths and formats	Build specifications become inconsistent
Undeclared Dependencies	Hidden relationships between tasks remain unspecified	Failures appear when execution order changes
Prediction Model Degradation	Compiler updates invalidate expected output patterns	Accuracy decreases without constant maintenance
Configuration Complexity	Combinatorial options make complete prediction impractical	Maintenance costs increase proportionally

Table 1. Output Non-Determinism in Distributed Environments [3, 4].

IV. Shared Filesystem Synchronization Constraints

A. Write Visibility Anomalies

In shared file systems like network-attached storage, the synchronization semantics do not map to the local execution model. Distributed file systems provide varying consistency models, with tradeoffs between performance and correctness [5]. Data flushing does not necessarily follow such acknowledgments. File size metadata may lag behind the data. However, directory entries may also become visible before their file contents are stable, leading to inconsistent behavior.

10.48047/jocaaa.2026.35.01.90

Client caches only exacerbate visibility problems across nodes in a distributed filesystem. The transmission of client changes to servers is slowed down by write-back caching. Read caching may return stale data after updates have been made; cache invalidation rules thus add latency. Different clients may observe different states of the filesystem.

Network partitions introduce other failure modes for shared storage, such as leaving files in intermediate states during a temporary partition. If a resolution is not found in a timely manner, and if split-brain occurs, the filesystem may diverge. Recovery procedures may not restore files to their intended state.

Metadata operations are not always synchronized with data operations. For example, a directory operation may complete before file data is visible. Attribute changes might not always be atomic with content changes: a file creation event may occur before the file's content is readable. These can violate the assumptions of sequential execution.

B. Parallel Execution Interference

Several building nodes writing to the same data structure increases the amount of synchronization delay. Collection synchronization is important in parallel programming [6]. Ordering results when several tasks have related outputs. Dependent tasks may contain incomplete sets of artifacts, resulting in incomplete object files and link-time errors.

Binary corruption affects readers that access partially written files. Executables may contain partially written code. Shared libraries may not have the expected symbols. Archive files may have inconsistent lists. These corruptions give rise to hard-to-debug symptoms and can be far removed from the original cause on the build dependency chain.

Intermittent failures characterize the parallel execution interference, whereby builds succeed when the scheduling is early enough to avoid conflicts. Otherwise, they fail. To reproduce failures, the non-determinism must be reproduced. Non-determinism weakens faith in building systems.

Underlying these guarantees, the filesystem must provide a guarantee that a task will only start when its outputs are materialized and available, while synchronization barriers wait for filesystem propagation delays, and retries deal with transient visibility failures.

Anomaly Type	Filesystem Behavior	Resulting Build Impact
Premature Write Acknowledgment	Writes acknowledged before data fully flushed	Readers access incomplete file contents
Metadata Lag	File size updates delayed behind actual content	Size-based checks return incorrect values
Directory Entry Timing	Directory entries visible before file contents stabilize	File existence confirmed but data unavailable
Client-Side Caching	Write-back caching delays propagation to servers	Different clients observe different states
Parallel Write Interference	Concurrent operations increase synchronization latency	Ordering uncertainties cause partial artifacts
Network Partition Effects	Temporary disconnections leave files in intermediate states	Recovery may not preserve intended file states

Table 2. Shared Filesystem Synchronization Constraints [5, 6].

V. Runtime-Based Output Discovery Mechanisms

10.48047/jocaaa.2026.35.01.90

A. System Call Tracing Architecture

Instead of output prediction, the approach tracks the real execution behavior of each build task on a remote machine and monitors system calls on the filesystem. System call analysis extracts behavioral patterns from an execution trace [7], containing operations such as open, write, rename, unlink and close. Tracing these activities also shows files that were created or modified. Since the absolute pathnames are clear, final sizes can verify the completeness of the work.

Although tracing covers filesystem-related system calls, it would be much more overhead to perform instruction-level tracing. Using selective tracing, only relevant output activity is emitted with little disruption. Kernel-level interception provides an exact view of what is happening without modifying the tracing tool. User-space tools are unaware.

Path resolution during tracing takes symbolic links and relative paths into account and saves the state of the working directory. Canonicalization produces identical absolute paths for all outputs. Duplicate detection removes duplicate entries from output lists. Temporary files created and deleted inside tasks are not reported in the results.

Trace analysis exports output metadata for testing. File sizes are provided when closing a file. The timestamps of the modified files are recorded for the additional source verification. Security-sensitive builds can log the permission bits. Extended attributes are preserved, if present, in outputs.

B. Tool-Agnostic Applicability

This method works for any compiler, linker, code generator, packaging script, or any other such tool, so the analysis tools need not know the implementation details used by the developer's tools to discover their output [8]. The system simply watches the files each of the tools touches. So the prediction-based failure modes are avoided.

Compiler diversity does not prevent the runtime discovery attack: different compilers produce different executables with different sets of system calls. Tracing detects actual behavior, so it does not depend on implementation. New compilers are automatically integrated. Updates to compilers do not affect output discovery. Bespoke programs are created with this kind of runtime observation in mind. Shell scripts producing output are traced like any other compiled program. Python-based build utilities have discoverable outputs. Custom code generators need no special handling. Commonly used system call interfaces improve application compatibility. Older tools, while lacking modern integration, remain usable, and decades-old build tools continue to run unchanged under tracing. Proprietary and undocumented tool behaviors become exposed. Tools using different technologies can interact, and it becomes easier to migrate from one build system to another.

System Call	Function	Output Discovery Role
open	Initiates file access for reading or writing	Identifies files accessed during task execution
write	Transfers data to file descriptors	Captures content modification operations
rename	Changes file path or name	Tracks output relocation and final naming
unlink	Removes file from filesystem	Filters temporary files deleted within tasks
close	Terminates file descriptor access	Records final file sizes at completion

10.48047/jocaaa.2026.35.01.90

Applicability Domain	Integration Requirement	Discovery Mechanism
Compilers	None	Automatic through system call interception
Linkers		Universal observation of file operations
Code Generators		Path resolution with canonicalization
Packaging Scripts		Shell command tracing at kernel level
Custom Build Utilities		Behavior captured regardless of implementation
Legacy Tools		Decades-old utilities function without modification

Table 3. Tool-Agnostic Output Identification Through Runtime Observation [7, 8].

VI. Verification Protocols and Failure Recovery

A. Size-Based Verification

After a remote task completes, some metadata containing the traced output returns to the node that launched the build. Several verification steps ensure correctness properties are met before dependent tasks are executed [9]. It checks for output files on shared storage and their sizes with the remote execution. Bounded waiting allows filesystem synchronization to activate dependent tasks only after successfully checking for output files.

Size matching strongly signals that full synchronization has occurred. Size mismatches from incomplete writes produce verification failures, truncation before dependent task accesses are detected, and partial synchronization states are detected when the size of the state is mismatched. False positives are rare if filesystem metadata is accurate.

The timing of the verification accounts for propagation delays. The initial checks may fail due to caching or delays in network communication. The number of retries prevents constant visibility failures, and the exponential backoff limits the polling of the shared storage. Timeouts prevent waiting indefinitely for failed synchronizations.

Checksums are more reliable than file sizes for detecting corruption and catching bit-level errors in files synchronized between sites. In contrast, cryptographic hashes are secure for integrity. Their high cost limits checkpointing to critical outputs.

B. Automated Recovery Strategies

Determinism increases the resilience of recovery. Specific recovery hooks are necessary for high availability and fault tolerance [10]. Remote retry solves container-specific issues through re-execution, local execution handles local synchronization issues, and scheduler feedback avoids unstable nodes in the cluster. These mechanisms reduce the need for manual intervention.

Remote retry on different nodes addresses infrastructure heterogeneity: hardware failures on specific machines on build farms, network disconnections of individual nodes, and resource exhaustion prohibiting completion on a highly utilized machine. Retry on healthy nodes often succeeds without diagnosing the cause of failure.

Local execution is the fallback for maximum reliability, where execution and consumption cannot become out of step. This incurs a performance overhead of local execution, but tasks on the critical path may be prioritized. Hybrid strategies combine distribution with local reliability.

10.48047/jocaaa.2026.35.01.90

Scheduler learning further improves the distribution of work by assigning tasks in lower numbers to nodes that have frequently failed. Work destined for nodes that fail to process tasks in a specific manner is then routed based on success rates. Feedback loops improve node selection algorithms continuously.

Verification Step	Process Description	Outcome
Existence Confirmation	Each output file presence confirmed in shared storage	Missing files trigger verification failure
Size Matching	File sizes verified against remote execution observations	Incomplete writes detected through size discrepancy
Bounded Waiting	Retry loops accommodate transient visibility failures	Exponential backoff prevents excessive polling
Checksum Verification	Content hashes confirm data integrity beyond size agreement	Bit-level corruption detected in synchronized files
Recovery Strategy	Trigger Condition	Resolution Action
Remote Retry	Container-specific issues suspected	Re-execution on different node
Local Execution Fallback	Synchronization failures persist	Task executed on initiating node
Node Avoidance	Frequent failures observed on specific node	Scheduler routes tasks elsewhere
Scheduler Learning	Historical success rates analyzed	Probabilistic decisions improve node selection
Workload Reduction	Nodes exhibit repeated failures	Reduced task assignments to unstable nodes

Table 4. Output Verification Protocols and Failure Handling Mechanisms [9, 10].

Conclusion

Since the synchronization of the distributed build infrastructure depends on the synchronization framework, discarding predicted output-for-input relations with runtime output verification allows distributed compilation to enter reliable execution frameworks. For large codebases, such guarantees from the build infrastructure are essential, and so reproducible builds are of great importance for developer productivity, with intermittent or unexplained pipeline failures unacceptable. System-call-based discovery is tool-agnostic, can handle arbitrary build commands without special-casing tools, and is easier to maintain than a prediction-based approach. Toolchain evolution is not a threat to building correctness or stability of a build system. New tools can be integrated via universal file system-level observability. Size-based verification protocols require full output synchronization and do not allow dependent tasks to start until all outputs are available. Partial visibility failures are avoided by bounding verification loops. To avoid race conditions between a producing task and a consuming task, a strict order is enforced so that cascading build failures don't propagate due to filesystem propagation delays. Bounded overhead preserves the performance advantages of distributed execution models. Tracing is currently limited to filesystem operations. Verification adds little to critical path delay, and the build time remains practical even as complexity grows.Reliability is considerably increased due to automatic failure recovery and self-

10.48047/jocaaa.2026.35.01.90

healing. Automatic retries in the event of transient failures and nodes chosen based on learned behavior improve node selection. Learning from execution history enhances the build system's reliability.

References

[1] Hridesh Rajan, "Building Scalable Software Systems in the Multicore Era," ACM, 2010. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/1882362.1882423

[2] Wing Lam et al., "Root Causing Flaky Tests in a Large-Scale Industrial Setting," Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2019. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/3293882.3330570

[3] Cor-Paul Bezemer et al., "An empirical study of unspecified dependencies in make-based build systems," Springer, 2017. [Online]. Available: https://www.researchgate.net/profile/Ahmed-E-Hassan-2/publication/315918566

[4] Zhemei Fang and Daniel DeLaurentis, "Dynamic Planning of System of Systems Architecture Evolution," ScienceDirect, 2014. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050914001185

[5] ELIEZER LEVY and ABRAHAM SILBERSCHATZ, "Distributed File Systems: Concepts and Examples," ACM Computing Surveys, 1990. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/98163.98169

[6] BENOIT DALOZE et al., "Parallelization of Dynamic Languages: Synchronizing Built-in Collections," ACM, 2018. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/3276478

[7] Quentin Fournier et al., "On Improving Deep Learning Trace Analysis with System Call Arguments," arXiv, 2021. [Online]. Available: https://arxiv.org/pdf/2103.06915

[8] Aditya Gondhalekar et al., "Tool-agnostic Framework for Systems Engineering Implementation," 10th Asia Oceania Systems Engineering Conference, 2016. [Online]. Available: https://www.researchgate.net/profile/Shripad-Kale/publication/312284397

[9] ILYA SERGEY et al., "Programming and Proving with Distributed Protocols," Proceedings of the ACM on Programming Languages, 2018. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/3158116

[10] Charlie Luca, "High Availability, Fault Tolerance, and Disaster Recovery Strategies," 2024. [Online]. Available: https://www.researchgate.net/profile/Charlie-Luca/publication/388527642