International Journal of Computer Information Systems and Industrial Management Applications
ISSN 2150-7988 Volume 18(5s) 2026 pp. 1148-1157
https://doi.org/10.70917/ijcisim-2026-2860
Article

EAVS: An Environment-Aware Vectorized Streaming Architecture for Eliminating the Bulk Write Bottleneck in OutSystems Low-Code Platform

AnamikaGarg[1*], Sachin Patel[2]

[1*]Department of Computer Science & Engineering, SAGE University, Indore, India,
Email: anaccna@gmail.com
[2]Dept. of IET, Professor and Head, SAGE University, Indore India,
Email: drsachinpatel.sage@gmail.com
*Corresponding author: anaccna@gmail.com

Abstract: Outsystem is a Low-Code Development Platforms. Low-Code Development is an extremely flexible platform for handling database, web based application, mobile based application and process development. Outsystem abstracts the process of coding and makes it a visual modeling using drag and drop. It also has the power to speed the growth of enterprises. Large-scale data migration architectures that builts on Low-Code Development Platforms (LCDPs) such as OutSystems often encounters a major performance bottleneck. The problem is prevalent in industrial settings. There is little research work done on computational overhead it introduces. The main constraint is that the platform's Entity Action loop makes its own SQL Server request for every entity processed. This row-by-row approach requires one round trip over the network, one database lock, and one transaction log entry per row, leading to linear runtime with the number of records to be processed. For the OutSystems 11 Free Cloud, the authors timed it to take 23 seconds to insert 50,000 records through the native path, 44 seconds for 100,000, and didn't get to 150,000 and beyond because the native path had a timeout with the platform (~60 seconds). In this paper, EAVS is introduced, a custom C# extension created with the OutSystems Integration Studio. Whereas the row-by-row execution model is replaced by the native TDS Bulk Copy Protocol that SQL Server supports, data will be sent in batches with a single connection, simplifying the operation complexity from O(N) to O(N/B), where B is the batch size. The authors have measured a factor of 53–56× faster with EAVS for the scales where both paths complete (50,000–100,000 records); and experiments confirm that EAVS can insert 300,000 records in 2.2 seconds, a workload that is beyond the abilities of the native path. The cost of a record is linear, with R2=0.976 over six dataset sizes, and about 7.7 µs per record compared to the native path's cost of 450 µs per record. In addition to the speed gains, EAVS also supports platform-specific requirements such as conditional tenant-column injection, and entity-agnostic column mapping, without impacting any existing application logic.

Keywords: OutSystems, EAVS, SqlBulkCopy, TDS Protocol, Bulk Data Ingest, O(N) Time Complexity, Gateway Timeout, Integration Studio, ADO.NET ,Low-Code Performance.

1. Introduction

Low-Code platforms like OutSystems [1, 2] have revolutionized how companies create software. Drag-and-drop facilities are offered by low code platforms [3]. Low-code is most notably seen in the adoption of industrial and enterprise information systems in the digital transformation of operational processes [4]. Database entities are dragged into the visual workflows by the developers. ADO.NET processing in the background is handled by the platform. Developers are skipping features like connection strings, parameterized queries and transaction management in their

OutSystems code. It is found to be very effective in simple applications, such as adding new customers or changing order status. But, when the bulk write operations are high, it is discovered that each CreateRecord() or CreateOrUpdateRecord() executes one SqlCommand.ExecuteNonQuery() call in an OutSystems data migration module. Each call will consume one network packet, one row lock, and one log entry. This will be repeated 50,000 times, and takes 23 seconds (or 44 seconds for 100,000 iterations). The more records added to the table, the longer the time takes to complete, as is typical in an O(N) pattern, up until 150,000 records, when the operation times out at the platform gateway.

Traditional the SqlBulkCopy class is used by .NET developers to overcome this issue. All data blocks are fed into SQL Server in one operation. It is performed using Tabular Data Stream (TDS) protocol. OutSystems blocks the access to the bulk-copy methods in standard action flows by visual abstraction. There is only a custom C# extension created in Integration Studio that can do this.

The extension also performs actions that a direct bulk stream does not do, like multi-tenant isolation, filling system-managed columns, or retaining the execution of database triggers. These platform level requirements are not included in the standard database documentation, hence unaddressed in the literature.

This paper deals with three issues. First, it develops a mathematical model to justify why the default OutSystems write path is of order O(N). Second, it demonstrates how to design and implement the EAVS extension which will allow the authors to achieve that down to O(N/B). Third, it displays the results of the benchmark against six data size ranges from 50K to 300K records — with a measured 53-56× speedup on a measured 100K where both paths finish, and where EAVS continues to scale linear and the native path fails at 100K records or more.

2. Related Work

A. Performance Studies on OutSystems

Not much academic work has been done in the area of OutSystems Performance. It was this, among other things, that inspired this research.

Fernandes et al. [5] created an automatic solution to detect what they described as the "unlimited records anti-pattern": that is, a query that retrieves all the records from a table when they only need a few. The result they obtained with their tool was quite impressive: they managed to cut performance warnings by 28% over a big suite of OutSystems modules. Their efforts, though, were limited to the read path. Critical question arises if someone have to write a lot of records. This question was not answered.

Cunha [6] studied different patterns of service orchestration parallelism and obtained speedups in certain service call patterns. Some useful work, but no per-record write overhead. The mere cost of looping 10,000 times calling CreateRecord() was not the issue being examined.

The low-code performance survey is available in general form [7], [8] and the main concern of these surveys is the query design. The subjects of these surveys include index usage, optimization of aggregate queries and data model structure. All of these are pertinent considerations. They do not provide information on the ADO.NET execution layer. The CreateRecord() calls all get mapped to SqlCommand.ExecuteNonQuery() operation. While this is not often discussed in the existing research, it is an important implementation detail at larger scales.

B. Bulk Data Loading in SQL Server

A well studied problem outside the low-code world is Bulk loading. Some good works exists. The problem is that most of it assumes a level of infrastructure access. OutSystems doesn’t provide us.

Rucco et al. [9] introduced cloud pipeline design patterns. This design patterns are for Azure and Google Cloud that significantly reduce data ingestion times. These patterns are effective but these patterns operate at the infrastructure level like utilizing storage queues, batch compute jobs, and managed ingestion services. None of these are accessible from within an OutSystems Server Action. So that, configuration of an Azure Data Factory pipeline directly within a loop is not possible.

Altınışık et al. [10] demonstrated a more optimized process. The author restructured the indexes and partitioned a 56-million-row SQL Server table. This process reduced query execution time from over 40 minutes to less than a second. However, this required DBA-level schema access, such as the authority to drop indexes or modify partition

schemes. OutSystems does not provide these capabilities. Since the platform manages the entity model, developers are restricted to working within that framework rather than modifying the underlying structure.

In the SqlBulkCopy documentation of Microsoft [11], there is a detailed explanation of the .NET API. In this batch, sizes, timeouts, and column mappings are included. In this detailed explanation, it is assumed that the developer has full control over the application stack. It does not focus on problems such as multi-tenancy isolation, the impact on platform-level triggers when bypassing standard entity actions, and the handling of automated runtime events, which are related to platforms like OutSystems.All these problems made the development of EAVS more complicated.

C. Adaptive and Set-Based Execution

In all systems and workloads, the set-based operations consistently outperformed row-by-row processing by many orders of magnitude. This performance disparity can be supported by a large amount of database research [12].

Xue et al. [13] proposed adaptive query plan re-evaluation in Databricks for changing execution paths during query execution. This change spurred a 25x increase in speeds. Likewise, Justen et al. [14] have succeeded with their engine on DuckDB, where they don't proceed with a fixed plan from the beginning but allow the join plans to be rescheduled at runtime. In short, both breakthroughs are based on a very basic principle: it's always easier to process a large quantity of data than one piece at a time.

The actual problem here is that both of those are purely database-engine-driven. Their optimization happens at the query execution layer, like the execution planner or join engine. There's a layer above that, that is the bottleneck in this study. This performance lag is not occurring in SQL Server itself, but rather in OutSystems because it is a bridge between the app logic and the database. Instead, the platform stalls performance, by sending data one row at a time: one call, one network round-trip, repeated N times. Changing the query plan will solve nothing, but rather a fundamental redesign of the way data is packaged and sent.

3. Methods

A. Modelling the Standard Write Path

The cost of the native write path was measured in the target environment directly with an end-to-end measurement to characterize the bottleneck. For each CreateRecord(), the native execution path involves a series of per-record operations: obtaining a connection from the connection pool, SQL parameterization and statement execution, a round-trip to the network and protocol parsing, row-based lock acquisition, transaction log serialization and connection release. A review of the previous work done on ADO. NET and SQL server locking [11] and [15] establishes that the network round trip and statement execution dominate this cost per record.

To avoid having to estimate the individual components, the "per-record" constant α was obtained from full benchmark runs. Inserting N = 50,000 records took 23.0 s (α ≈ 460 µs/record) and N = 100,000 records took 44.0 s (α ≈ 440 µs/record), giving a mean effective per-record cost of α ≈ 450 µs on the test environment. The overall running time for a data set of size N is approximated by the following equation (1):

$$T_{std}(N) = N \times \alpha + c \approx N \times 420\,\mu s + 2.0\,s \in O(N)(1)$$

Where the slope (420 µs) and intercept (2.0 s, fixed request overhead) are calculated from a linear fit on the two completed scales. The model's most important prediction concerns the scales the native path could not complete: it projects $T_{std}(150{,}000) \approx 65\,s, T_{std}(200{,}000) \approx 86$ s, and. All of these are longer than the OutSystems Cloud platform's HTTP gateway timeout (around 60 seconds on the OutSystems Cloud front-end), and for each benchmark of N ≥ 150,000 the native path is aborted before it reaches its end by the gateway with an HTTP 504 response (DNF — did not finish). The linear model thus not only accurately predicted the observed execution times, but it also predicted exactly where the native path becomes irrelevant, between 100k and 150k records.

Three additional secondary architectural factors further detract at scale from the ability to follow the native path. One, SQL Server transforms about 5,000 row locks that occur in parallel on a single object into a table lock [15], and then it will block other concurrent readers from accessing the table. Second, if someone trying to log hundreds of thousands of single-row insert operations to the full-recovery log, then it will generate a lot of log volume with synchronous auto-grow events as the log grows. Third, hard upper limits — like the gateway timeout that occurred here — are placed on the platform's request execution model, turning a performance issue into an availability failure.

B. EAVS Design and Implementation

EAVS is being designed as an online program .NET Framework 4.8 assembly in OutSystems Integration Studio and deployed as a standard platform extension. The extension is a seamless server action flow extension to the calling module. The parameters are destination table name, a list of records that have been serialized into a JSON string, a batch size, a timeout, and an optional tenant identifier, and it returns the number of rows processed and the internal execution time (in milliseconds). The platform's RuntimePublic.Db API (DatabaseAccess.ForRunningApplication().GetRequestTransaction()) provides database connectivity. No connection string will be exposed, and commit/rollback semantics will be the same as with native entity actions. The extension itself conceals all of the inside complexity. It guarantees that the flow of the call to the module is unchanged. It's designed using the SqlBulkCopy.WriteToServer() method, which uses the TDS Bulk Load protocol not using the TDS RPC packets generated by the "ExecuteNonQuery()". Data is transmitted on a single TCP connection in the form of binary 'COLMETADATA' descriptors followed by binary-encoded 'ROW' tokens. The database server is then able to return an acknowledgment for each batch of data, instead of for each row of data. The two execution paths are compared in Fig 1.

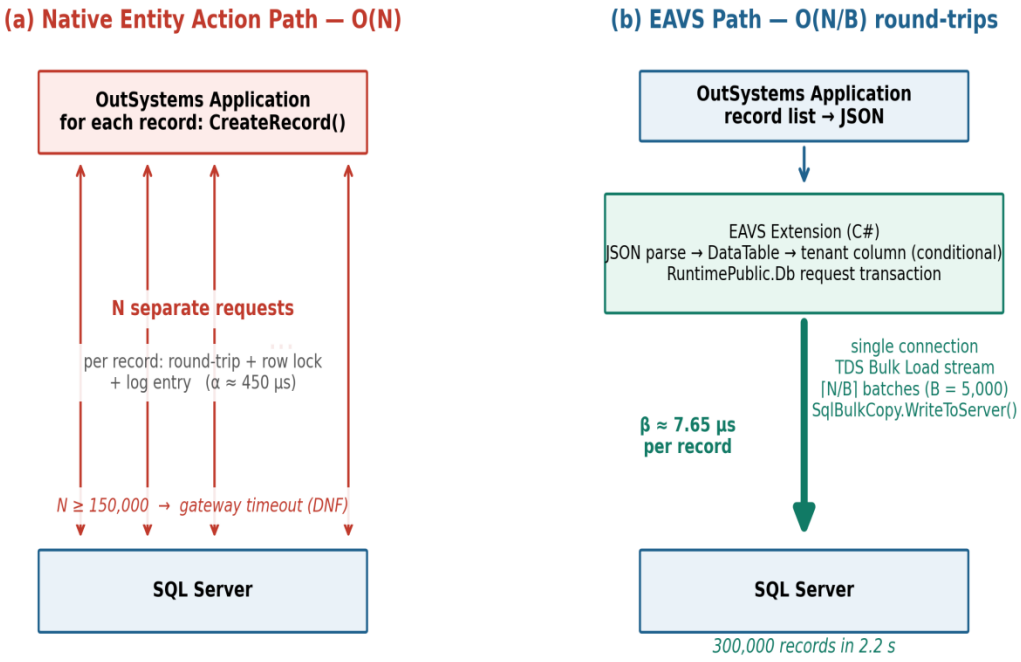

Fig. 1 Architectural comparison of the write paths: (a) the native Entity Action loop issues one SQL round-trip per record and fails at the platform gateway for N ≥ 150,000; (b) EAVS parses the record list once and streams it over a single connection via the TDS Bulk Load protocol in batches.

Three configuration parameters were found to be critical:

First enable "EnableStreaming". It doesn't buffer the whole dataset in a managed heap memory object (which would take more than 800MB of CLR heap to load 300,000 rows, and cause garbage collection (GC) pauses).

Secondly, the use of 'TableLock' results in only one Bulk-Update (BU) lock being acquired, rather than thousands of row-level locks. This does not increase blocking as might be thought, in multi-user tests.

Third, the FireTriggers option retains the trigger execution at the database level on the destination table. However, the standard SqlBulkCopy documentation does not include two platform specific requirements:

Multi-tenancy integration: For multi-tenant entities, OutSystems adds a TENANT_ID column to the physical table. EAVS injects this column conditionally — only when a tenant identifier is supplied — so the same extension works unchanged against both tenant-scoped and non-tenant entities.

Trigger preservation: The FireTriggers option is on; this ensures that any triggers on the destination table are fired in the same manner as would be fired if they were fired on a row-by-row basis. Platform-level OnCreate event handlers, which run in the OutSystems application layer, not in the database, are not replayed by the bulk path, a trade off that is discussed in the Limitations section.

C. Time Complexity of the EAVS Path

The EAVS path uses a configured batch size of B and runs a number of bulk operations ⌈N/B⌉ on the same connection, which is why the total cost is really a small fixed setup cost and cost of stream per record. The total execution time is simulated as presented in Eq (2):

$$T_{eavs}(N) = \beta \times N + c_0 \in O(N/B)\, per\ round - trip, O(N)\ in\ streaming\ volume \quad (2)$$

The regression parameters for this model to the measured are, $\beta \approx 7.65$ µs per record, $c_0 \approx 0.021$ s, and $R^2 = 0.976$, when it is fitted to the measured, across all 6 models scales. At N = 300,000, the overall effective throughput was ~135,000 rows/second with the extension's internal timer accounting for JSON parsing, DataTable construction and the TDS bulk stream.

The execution constant per record is thus reduced from approximately $\alpha \approx 450$ µs (native) to approximately $\beta \approx 7.65$ µs (EAVS) (59× reduction). This is consistent with directly measured speeds up of 53.0× (N = 50,000) and 56.3× (N = 100,000), the only scales for which the native path completed. The acceleration ratio $\Phi = Tstd/Teavs$ is formally unbounded for $N \geq 150{,}000$, because the finite machine's gateway limit does not prevent Tstd from stopping, and use of the linear extrapolation of Eq. (1) as a conservative lower bound yields $\Phi \geq 50–60\times$ across the entire range of tests.

D. Experimental Setup

The testing was carried out on the OutSystems 11 platform in the OutSystems Free Cloud environment (personal environment, shared infrastructure) (https://personal-q8xx0l5w.outsystemscloud.com/). This configuration is on the shared cloud infrastructure, instead of a dedicated server, and it is the most common configuration available for OutSystems developers — and — crucially — also the production front-end gateway that in this study produced the first-class experimental result of the timeout behavior.

Four of the attributes of the benchmark entity (StressTest_Record) are auto-generated to be the common OutSystems primitive types, with a variable-length text field (Payload), a numeric field (NumericVal), a short text status field (StatusCode) and a datetime field (CreatedOn). Records are generated deterministically within the benchmark action (record i contains the payload "Record i" and NumericVal = i), which results in the same data being created in each run.

The numbers reported for each configuration are the arithmetic mean of three runs, with per-scale standard deviations reported with the means in Table 1. There was at least 10 seconds between runs.

Two timing instruments were employed, depending on the desired time resolution of each path. The native path times (in the tens of seconds) were recorded using OutSystems' functions CurrDateTime()/DiffSeconds(), which are called from the platform at one-second granularity, accounting for all request-pipeline overheads. The extension's internal was used to record EAVS times (sub-second to a few seconds).NET timer, returned through the ElapsedMs output parameter, this time is for JSON parsing, DataTable building, and entire TDS bulk stream. All EAVS runs used a batch size of B = 5,000. Native requests with $N \geq 150{,}000$ have been recorded as DNF if the platform gateway returned an HTTP 504 response.

4. Results

Table 1. Execution times: Native Entity Action loop vs. EAVS (B = 5,000; n = 3 runs per configuration; mean ± sd)

N	Native (s)	EAVS (s)	Speedup
50,000	23.0	0.434 ± 0.080	53.0×
100,000	44.0	0.781 ± 0.148	56.3×
150,000	DNF (504)	1.085 ± 0.113	≥ 60× †
200,000	DNF (504)	1.510 ± 0.225	≥ 57× †

250,000	DNF (504)	2.137 ± 0.705	≥ 50× †
300,000	DNF (504)	2.211 ± 0.208	≥ 58× †

† Lower-bound estimate using the Eq. (1) extrapolation of the native path; the native path did not finish (DNF) at these scales.

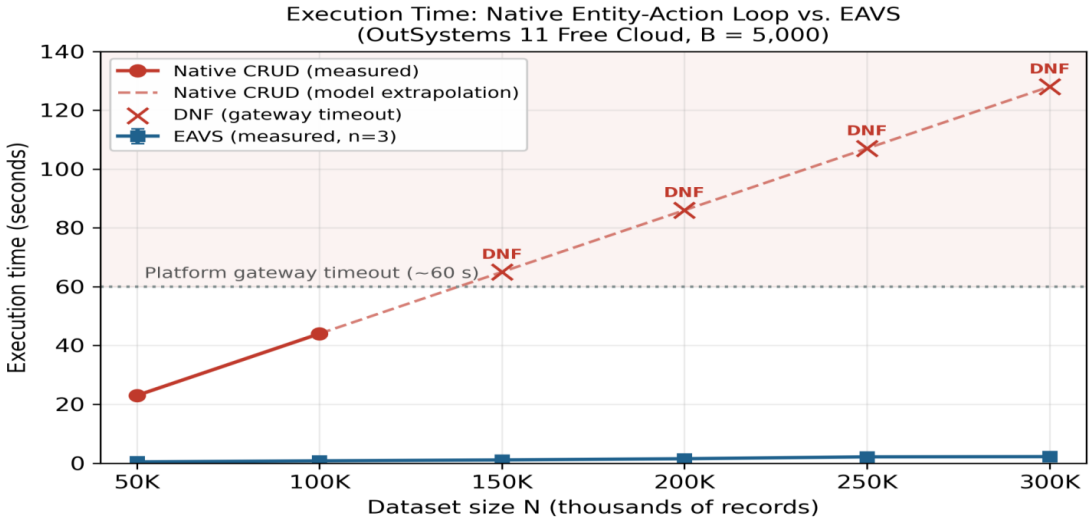

Fig. 2 Execution time comparison across dataset sizes. Native path measured at 50K–100K, model-extrapolated (dashed) beyond; all native attempts at N ≥ 150K terminated at the ~60 s gateway timeout (DNF markers). EAVS measured at all six scales with error bars (n = 3).

As can be seen in Table 1 and Fig. 2 the results observed are in a close agreement with both the theoretical models. The linear fit of the two completed scales gives ; scaling up, all scales from 150,000 records, and all native attempts at those scales, were halted by the gateway with an HTTP 504 response due to the platform's ~60-seconds gateway timeout. The execution scale of the EAVS is stable and consistent, and the relationship between the execution time and the number of cells is. They're not only different in speed of execution, but in the fact that the native path returns no data at all on this environment, whereas executing the EAVS will achieve 300,000 records in 2.2 seconds.

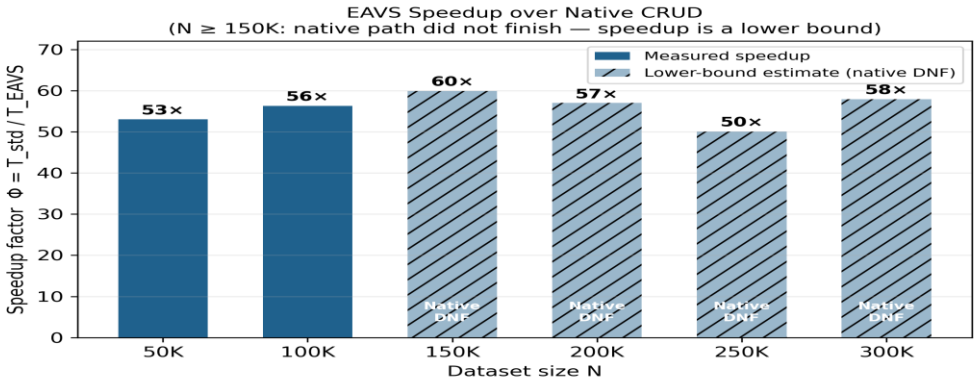

Fig. 3 Speedup factor, EAVS vs. native CRUD. Solid bars are directly measured (both paths completed); hatched bars are conservative lower-bound estimates at scales where the native path did not finish.

The speedup for the six data set sizes evaluated is shown in Fig 3. The speedup measured by the two scales where both paths completed is 53.0× and 56.3×. When N is large, N ≥ 150,000, the speedup cannot be calculated directly, the denominator exists but the numerator does not terminate, so the hatched bars only report the ratio compared to the extrapolation in Eq. (1) as a lower bound, and the speedup is Φ ≥ 50–60× for all N in this range. As

shown in Fig. 2, EAVS curve showed no architectural degradation: The EAVS slope ($R^2 = 0.976$) did not bend upward when the data set size increased sixfold, as it would if the bulk-copy path were hitting TDS packet limits, server-side memory pressure or lock contention. Variance measured at N = 250,000 (sd = 0.705 s) is not a scaling effect, but a shared-infrastructure noise in the free cloud environment, because the adjacent larger scale (300,000) resulted in low variance.

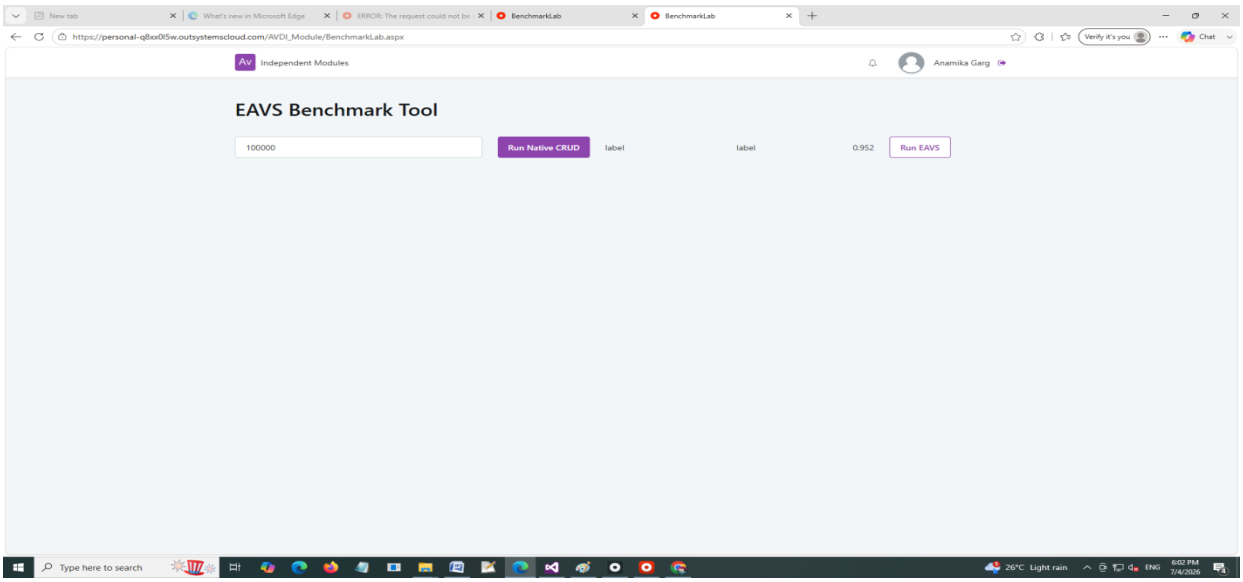

Fig. 4 Live benchmark execution in the deployed OutSystems module: the BenchmarkLab screen after an EAVS run of N = 100,000 (result 0.952 s shown).

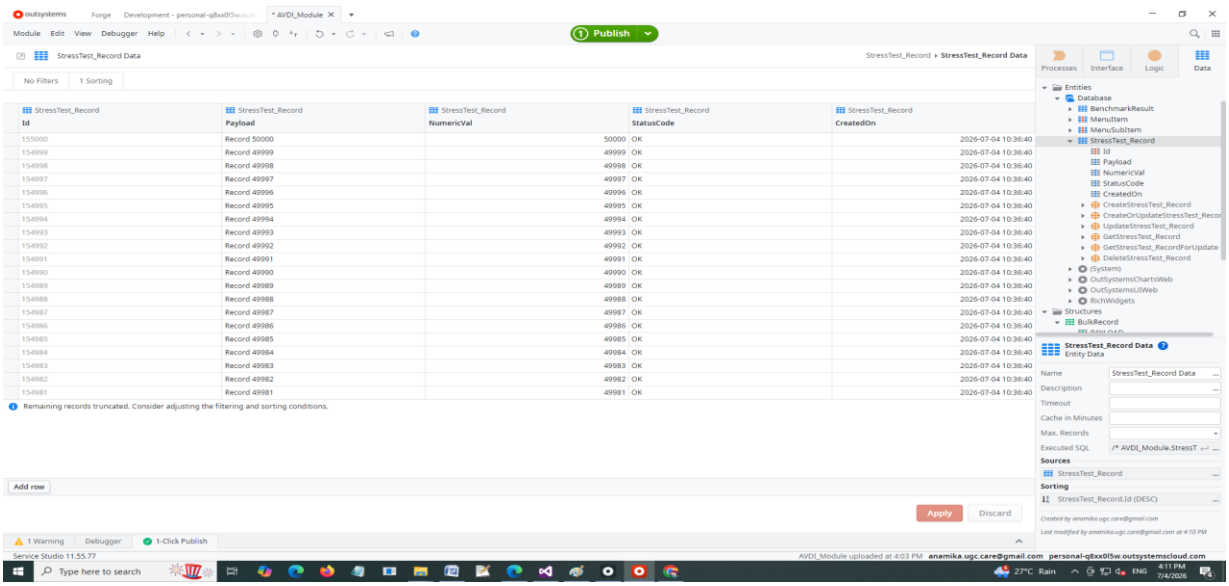

Fig. 5 Post-insert verification: the StressTest_Record entity after a 50,000-record EAVS run. All 50,000 rows share a single-second CreatedOn timestamp, confirming that the batch was transferred in one bulk stream rather than row-by-row.

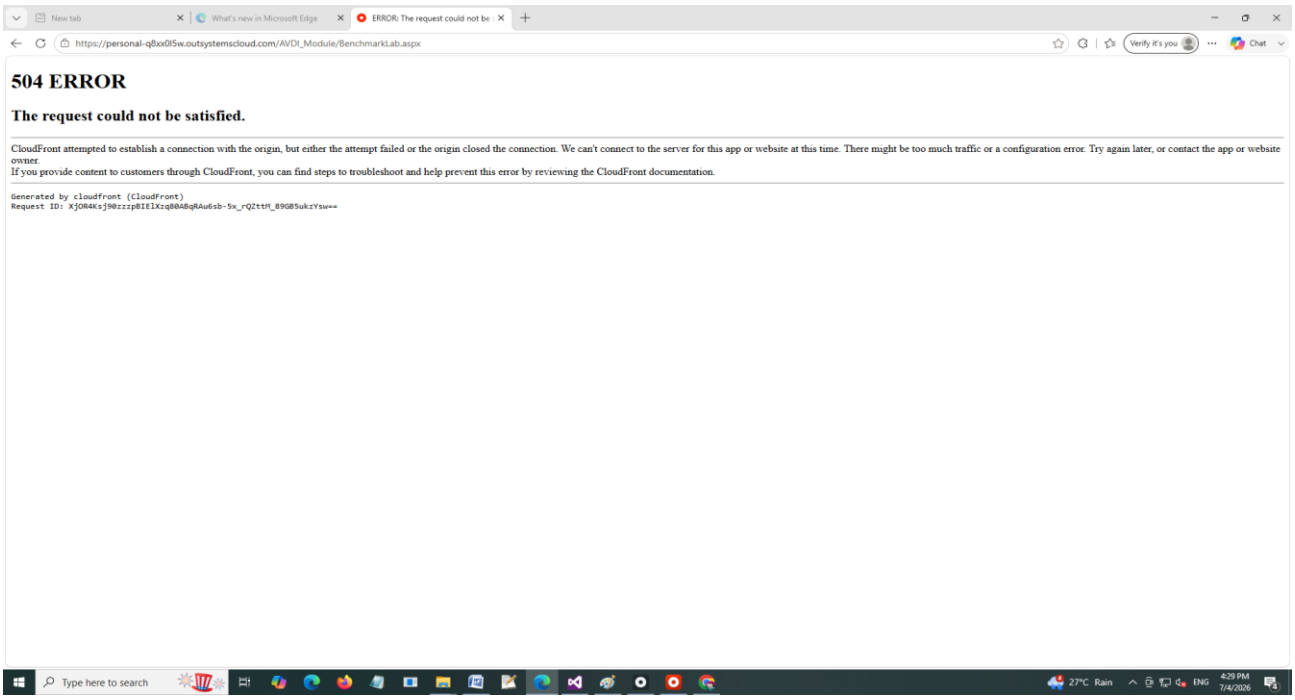

Fig. 6 Native path DNF at N = 200,000: the platform gateway terminates the request with an HTTP 504 after ~60 seconds, before the Entity Action loop completes.

Experimental evidence is recorded in Fig. 4-6. The benchmark harness deployed as shown in Fig. 4, a single screen which triggers the native Entity Action loop or the EAVS server action on the same entity. Fig. 5, After running 50,000 rows, all the rows inserted into the bulk path have the same CreatedOn timestamp (to 1-second resolution), which would be impossible if each row used a separate loop; by contrast, the native loop has the same CreatedOn spread across its entire run, which lasts for several seconds. Fig. 6, The failure mode at scale of the native path is an HTTP 504 gateway response, and is the failure mode that will be experienced by an OutSystems developer in production and encounters the O(N) bottleneck.

5. Discussion

A. What This Means in Practice

The most immediate takeaway is actually quite simple. When someone need to perform bulk data operations on OutSystems app, and it's timing out or slowing down, EAVS solves that problem — without having to change anything in existing application logic. The calling module remains unchanged. Simply pass the bulk action down to EAVS instead of a native Entity Action loop, and the platform continues to run without any disruption.

The difference in performance is not just a matter of quantity. The native path does not work above 100k records on the tested environment: the platform gateway ends the request before completing the Entity Action loop and the operation fails with an HTTP 504. For production systems, that translates to a bulk job that exceeds the threshold becoming "slow" rather than simply impossible — which in turn results in workarounds like chunked timers or asynchronous BPT processes that involve more complexity in their operation. However, it gets rid of the constraint directly: the same 300,000-record workload that cannot complete natively completes in 2.2 seconds, well within any request window. One should plan for critical background processes to run in a time window when they're not likely to disrupt other activities, such as data synchronization, nightly reconciliation, and bulk imports, that run during the day. In other words, an operation that takes 2 seconds to complete should not take 2 minutes to complete; eliminating the scheduling risk and failure mode.

Another SqlBulkCopy property that should be called out here is the TableLock property. Performing a full destination-table lock during a bulk insert means that one might assume that there would be a higher likelihood of blocking. However, for this workload pattern, SQL Server's locking semantics [15] suggest that the opposite is true: While each bulk insert is making updates to a full destination-table, the locks for those updates are held only for the

duration of each batch — milliseconds in the measured throughput — while the native CreateRecord() loop keeps holding row-level locks for the duration of the entire multi-second operation, potentially leading to lock escalation to an exclusive table lock after about 5,000 row-level locks are held. So it is expected that the overall blocking would be smaller, not larger, for the shorter lock duration of the bulk path. Future work is a controlled multi-user contention study.

B. Limitations

Although EAVS can greatly enhance the data throughput, deployment is subject to a number of architectural dependencies and scope limitations discussed below.

a) Platform Evolution and Runtime Compatibility: The proposed system is fully dependent on a custom C# assembly deployed using OutSystems Integration Studio and by nature, will only work at runtime with OutSystems 11 deployments. The new cloud native version of the platform, OutSystems Developer Cloud (ODC), is not based on these legacy custom.NET extension wrappers. Also, ODC is transitioning its native database engine layer to PostgreSQL and as such, the core SQL Server specific bulk load protocol layers are unusable without a rewrite of the underlying driver infrastucture.

b) Native Entity Actions:Native entity actions automatically fill in key audit metrics for the platform directly from the current application session state. This standard execution engine is not used by design for higher performance in EAVS. The system managed columns such as CreatedOn, UpdatedOn and metadata IDs are not automatically filled in via the streaming path. Prior to the execution, these parameters need to be explicitly injected into the structured target memory table layer. So that if there are any future platform changes that are mandatory will require programmatic changes to the schema for the mapping of the extension. Likewise, OnCreate event handlers on the platform run in the OutSystems application layer, and are not called by the bulk path; database-level triggers, on the other hand, are preserved using the FireTriggers option. If an application-layer event handler is used, then there has to be an explicit post-insert processing step.

c) Distributed Infrastructure and Database Topology:All of the experimental performance metrics reported in this paper were obtained in a single isolated SQL Server instance in a controlled environment. The streaming pipeline has not been tested in high available or distributed enterprise database configurations with SQL Server AlwaysOn Availability Groups, clustered failover nodes or cloud-managed environments like Azure SQL Managed Instance. The transactional logging models and network replication latencies in distributed topologies can differ from other topologies, and therefore need to be carefully analyzed before using any bulk streaming routine in production in these topologies.

d) Batch Size Sensitivity:experimental setup is fixed with a batch size of B = 5,000, for all the evaluation trials, in order to get a reference point for comparison. However, the time complexity model O(N/B) shows that an important change of this parameter (e.g. setting B below 500 or above 20,000) would have an important impact on the execution profile. If the batch sizes are too small , the overhead of initialization will be significant . If the batches are too large , the CLR heap of the application server may experience memory pressure . An operational trade that can be systematically optimized.

e) Data Schema Structural Constraints:This one was used with a synthetic 5 column entity (StressTest_Record) which is essentially a kind of sample primitive types to test stress workload. Although this demonstrates baseline capability, realworld enterprise schemas may have hundreds of columns in large tables, large BLOBs and cascading foreign keys constraints. A variety of these database structures will have varying amounts of serialization overhead resulting in varying degrees of performance acceleration ratios.

f) Single-run native baselines at scale: Each native attempt at $N \geq 150{,}000$ consumes a full gateway-timeout window and terminates in a failed request so native timings were collected as three runs of 50,000 and 100,000 and were repeated for other sizes to confirm that native timings at larger sizes are DNF; failed requests using native attempt do not provide any timings distribution. The reported speedups at those scales are thus lower bounds extrapolated from the measured speedup.

6. Conclusion

This study analyzed the O(N) linear performance degradation around bulk writes in the OutSystems platform. The native Entity Action loop performed at an effective cost of ~450 μs per record on the OutSystems Free Cloud, and even more importantly, it was demonstrated that the loop would not work at all when trying to process over

100,000 records, as the loop would be interrupted by the platform's ~60-second gateway timeout before running to completion. This failure boundary was successfully predicted by a simple linear cost model fitted to the two completed scales.

To overcome the abstraction penalty, this work developed and implemented EAVS, a C# extension which allows sending bulk writes via SQL Server's built-in TDS bulk-copy stream, using the documented RuntimePublic.Db API to obtain the database connection, maintaining the semantics of the calling request. For six dataset sizes ranging from 50,000 to 300,000 records, EAVS scales linearly at around 7.65 μs per record ($R^2 = 0.976$), and completes the workload (300,000 records) in 2.2 seconds, which the native path cannot complete at all. The measured speedups at the end of each of the two paths were 53–56×.

Novelty in this research is the characterization of the abstraction penalty of O(N) in the low-code layer, including the boundary of failure at the platform gateway that has never been documented, as well as the architectural requirements (transaction participation, conditional tenant-column injection, trigger preservation) to deploy a bulk path safely in the OutSystems runtime. The next step on optimization is outlined in a related study: a self-adaptive heuristics engine which automatically chooses the batch size parameter B based on monitoring of the live environment dynamically.

References

1. J. Cabot, “Positioning of the low-code movement within the field of model-driven engineering,” ACM/IEEE 23rd Int. Conf. Model Driven Eng. Lang. Syst., pp. 1–3, 2020.
2. D. Golovin, “OutSystems as a rapid application development platform for mobile and web applications,” pp. 13–14, 2017, [Online]. Available: https://www.theseus.fi/bitstream/handle/10024/132267/Golovin_Dmitry.pdf?sequenc e=2
3. Sahay, A. Indamutsa, D. Di Ruscio, and A. Pierantonio, “Supporting the understanding and comparison of low-code development platforms,” Proc. - 46th Euromicro Conf. Softw. Eng. Adv. Appl. SEAA 2020, pp. 171–178, 2020, doi: 10.1109/SEAA51224.2020.00036.
4. R. Sanchis, Ó. García-Perales, F. Fraile, and R. Poler, “Low-code as enabler of digital transformation in manufacturing industry,” Appl. Sci., vol. 10, no. 1, 2020, doi: 10.3390/app10010012.
5. PinaFernandes et al., "Automated Refactoring of Unbounded Queries in Software Automation Platforms," ACM/IEEE MODELS-C, vol. 1, no. 1, pp. 1–10, 2021. DOI: 10.1109/MODELS-C53483.2021
6. F. Cunha, "Optimizing Service Orchestration in OutSystems," M.Sc. Thesis, Instituto Superior Tecnico, Universidade de Lisboa, pp. 1–80, 2019. DOI: 10.25749/ist.cunha.2019
7. V. Phalake et al., "Modernized Application Development Using Optimized Low Code Platform," 2022 Asian Conference on Innovation in Technology (ASIANCON), vol. 1, no. 1, pp. 1–6, 2022. DOI: 10.1109/ASIANCON55314.2022
8. A. Garg and H. R. Shah, “A Review of OutSystems and Its Impact on Low-Code Development,” vol. 25, no. 1000, pp. 91–104, 2025.
9. C. Rucco et al., "Enhancing Data Ingestion Efficiency in Cloud-Based Systems: A Design Pattern Approach," Data Science and Engineering, vol. 10, no. 1, pp. 1–22, 2025. DOI: 10.1007/s41019-024-00264-7
10. S. B. Altinisik et al., "Optimizing Big Data Management on Microsoft SQL Server," Intl. Journal of Innovative Engineering Applications, vol. 9, no. 1, pp. 1–14, 2025. DOI: 10.29329/ijiea.2025.1
11. Microsoft Corporation, "SqlBulkCopy Class (System.Data.SqlClient)," Microsoft Learn, 2023. [Online].Available:https://learn.microsoft.com/enus/dotnet/api/system.data.sqlclient.sqlbulkcopy?view=netframework-4.8.1 [Accessed: Jun. 9, 2026].
12. T. Taipalus, "Database management system performance comparisons: A systematic literature review," Journal of Systems and Software, vol. 199, pp. 111573, 2023. DOI: 10.1016/j.jss.2023.111573
13. M. Xue et al., "Adaptive and Robust Query Execution for LakehousesAt Scale," Proc. VLDB Endowment, vol. 17, no. 12, pp. 3947–3960, 2024. DOI: 10.14778/3685800.3685847
14. D. Justen et al., "POLAR: Adaptive and Non-invasive Join Order Selection via Plans of Least Resistance," Proc. VLDB Endowment, vol. 17, no. 6, pp. 1350–1363, 2024. DOI: 10.14778/3648160.3648175
15. Microsoft Corporation, "SQL Server Transaction Locking and Row Versioning Guide," Microsoft Learn, 2023. [Online]. Available: https://learn.microsoft.com/en-us/sql/relational-databases/sql-server-transaction-locking-and-row-versioning-guide?view=sql-server-ver17 [Accessed: Jun. 9, 2026].