International Journal of Computer Information Systems and Industrial Management Applications
ISSN 2150-7988 Volume 18(18s) 2026 pp. 329-339
https://doi.org/10.70917/ijcisim-2026-4873
Article

HBA: A Heuristic Batch Advisor for Self-Tuning Bulk Data Ingestion in the OutSystems Low-Code Platform

Anamika Garg*[1], Sachin Patel[2]

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

Abstract: Low code development platforms like OutSystems allow teams to build data-centric applications using visual models that the platform takes care of the database plumbing [1] [2]. We have demonstrated in an earlier study that the write path is row-wise and fails at scale; and in a previous study we introduced EAVS [25], a C# extension to restore bulk ingestion by using SQL Server's TDS bulk-copy stream. However, EAVS [25] left one decision to the developer, the batch size B, which is not explained in the vendor's documentation, and is only suggested to be tested [10]. This paper first demonstrates that B is not a cosmetic setting. Across an eight-point sweep at N = 100,000 records on the OutSystems Free Cloud, a poorly chosen batch size (B = 500) roughly doubled execution time relative to the efficient region, and even our own informed default of B = 5,000 ran 28 to 47 percent slower than the best measured configurations. A second sweep with payloads about eleven times wider showed the efficient region itself moving by an order of magnitude, from roughly B = 10,000 down to roughly B = 1,000. Interestingly, both workloads converged near a constant batch payload of about 1 MB, which suggests that the quantity worth controlling is bytes per batch rather than rows per batch, and that no single fixed B can suit every workload. Building on that observation we present HBA, the Heuristic Batch Advisor, a self-tuning variant of the EAVS [25] bulk path that takes no batch-size input at all. HBA seeds its first batch from a byte-budget heuristic that targets about 1 MB per batch using the payload's own measured row width, then adjusts B between chunks based on observed throughput, smoothed with an exponential moving average so that a single noisy reading cannot whipsaw the controller. On thin rows at N = 100,000, HBA with zero manual tuning ran 32 percent faster than a naive configuration and matched the designers' informed default, staying within about 27 percent of the best hand-tuned batch size; on wide rows it matched the best fixed configuration outright, within about 1 percent, while beating the naive choice by 49 percent. The seed heuristic was verified directly in a cross workload test, where the controller opened B = 10,485 for 100-byte rows and B = 1,113 for 1.1-KB rows (no knowledge of the data). The instrumentation records the adaptation as it occurs, including runs in which the controller overshot and recognized the drop in throughput and then backed off. HBA is deployed outside database engines and runs in a low-code platform's hermetically sealed extension layer and has only feedback in the form of end-to-end throughput.HBA is deployed outside database engines, and is used in the hermetically sealed extension layer of a low-code platform, with end-to-end throughput being the only feedback signal.

Keywords: OutSystems, HBA, Adaptive Batch Sizing, SqlBulkCopy, Runtime Feedback, Self-Tuning Systems, Bulk Data Ingest, Low-Code Performance, EAVS [25], TDS Protocol

1. Introduction

Low code platforms have revolutionized enterprise software development. The developers integrate database entities as visual flows, and the platform automatically writes and runs the code to access the data. [1] [2] These platforms are particularly noticeable in industrial and enterprise settings, where they reduce the time and effort required to take a business process to a working system [3]. The abstraction cost however becomes apparent when it's loaded.

In our companion study, we characterized that cost for bulk writes. The time per record for the native Entity Action loop on the OutSystems 11 Free Cloud is approximately 450 microseconds, scales linearly and is not completed above 100k records when the platform gateway terminates the request before the loop finishes. The EAVS [25] extension proposed in that work removes the bottleneck by streaming records over a single connection through SQL Server's TDS Bulk Load protocol, cutting the per-record cost to under 8 microseconds. EAVS [25] made bulk ingestion feasible; it did not make it self-sufficient. The extension still expects the caller to supply a batch size, and in our own benchmarks we fixed it at B = 5,000 because that value seemed reasonable.

It turns out that "seemed reasonable" is exactly the problem this paper is about. Microsoft's own documentation for SqlBulkCopy declines to prescribe a batch size and recommends measuring in the target environment [10]. That advice is sound but impractical inside a low-code delivery model, where the whole point is that developers do not hand-tune infrastructure parameters. If the right value of B depends on the environment and the data, and the platform's audience is not expected to run tuning experiments, then the parameter should tune itself.

This paper makes three contributions. First, it quantifies the cost of getting B wrong: on the same environment and workload as the EAVS [25] study, a bad batch size doubled execution time, and our published default was itself 28 to 47 percent away from the efficient region. Second, it shows that the efficient region is workload-dependent, shifting by roughly an order of magnitude when row width changes by an order of magnitude, while the batch payload at the knee stays close to 1 MB in both cases. Third, it designs, implements, and evaluates HBA, a heuristic controller inside the extension that picks and adapts B on its own, using only the information available above the database engine: the payload it is asked to insert and the time each chunk takes.

1.1 Related Work

Research on OutSystems performance is still thin. Surveys and foundational treatments of the low-code space [21], [22], [23] discuss productivity, adoption, and definitional scope far more than runtime behaviour, and recent empirical and industry perspectives echo the same emphasis [20], [24]. Fernandes et al. addressed the read path, refactoring unbounded queries in generated applications [4], and Cunha examined service orchestration overheads inside the platform [5]. Taipalus's systematic review of database performance comparisons notes that vendor-managed execution layers are rarely studied as performance subjects in their own right [6]. Our companion EAVS [25] paper appears to be the first to model and measure the platform's bulk write path directly. Adjacent low-code engineering concerns such as testing and runtime monitoring have begun to receive attention [18], [19], but none of these works considers parameter tuning within a platform extension, which is the gap HBA occupies.

Adaptive execution itself is a mature idea inside database engines. Xue et al. describe adaptive and robust query execution for lakehouses at Databricks, where the engine re-plans mid-query using statistics it collects about itself [7]. Justen et al.'s POLAR achieves non-invasive adaptive join ordering in DuckDB [8]. Rucco et al. study ingestion efficiency on modern cloud systems [9], and a broad body of work examines high-volume real-time and stream processing, including modular high-throughput designs [13], adaptive micro-batching [12], pipeline quality and failure analysis [14], systematic performance mapping and fault-recovery benchmarking of stream frameworks [15], [16], and cost-aware resource recommendation for DAG-based big-data workflows [17]. The difference between HBA and this kind of work is not the wanting but the point of view. Cardinalities, operator costs and memory pressure are observed on the engine. No such thing as an OutSystems extension. It is conFig. d at the top of a sealed engine, tracks one signal – how long it took for its last batch – and has a database connection it didn't set up. In that context, shared cloud infrastructure, it is exactly this question that this paper addresses: Can there be useful adaptation from that position?

The advice for the sizing of batches is folklore in the practitioners' literature. Dr. Jeffrey Pollack of the SqlBulkCopy documentation says that there is no standard number, but to test [10] and there are discussions of round numbers without any measurements in the community. There does not seem to be an existing published, data-based batch-size sensitivity study on a bulk path within a low-code platform, and there is no self-tuning batch controller that has been deployed at that layer.

2. Materials and Methods

2.1 Batch-Size Sensitivity Study

All experiments extend the benchmark harness of our earlier EAVS [25] study: the same OutSystems 11 Free Cloud personal environment, the same StressTest_Record entity (an identity key plus four payload attributes), and the same millisecond timer inside the extension. The only change was exposing BatchSize as a screen input so that the same EAVS [25] action could be driven across a range of B values.

The first sweep held the workload fixed at N = 100,000 thin rows (about 100 bytes per record once serialized) and varied B across eight points from 500 to 100,000, with three to five repetitions per point spaced at least ten seconds apart. Table 1 lists the results and Fig. 1 plots them. Execution time falls steeply as B grows from 500 to about 10,000, which is the round-trip overhead region: at B = 500 the transfer needs 200 acknowledged TDS batches, and the fixed per-batch cost dominates. The curve levels off around B = 10,000 and becomes efficient at B = 0.6-0.73 seconds. There are two salient points. The naive value B = 500 costs approximately twice as long as the efficient time, and the default value that we used in the EAVS [25] study of B = 5,000 is 28 to 47 percent above the efficient region. It's hard to imagine that the owners of the system, who know the system best and based on their design, chose a suboptimal constant, and that application developers would do any better.

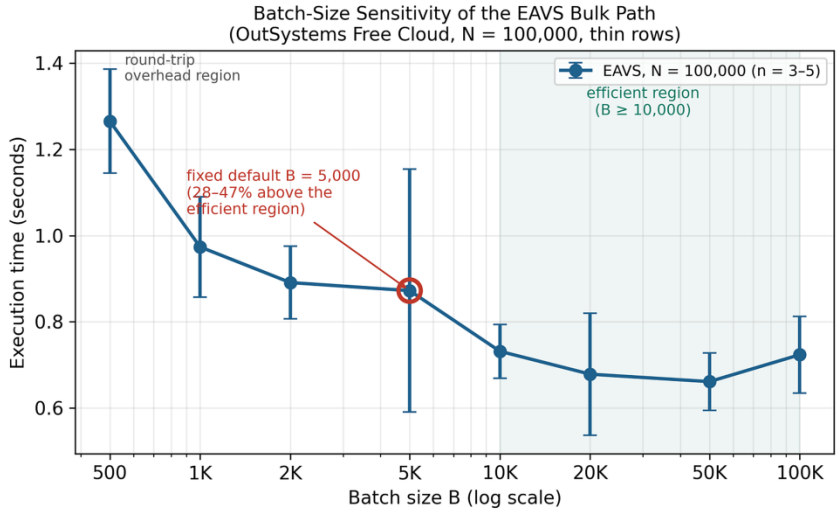

Fig. 1: Batch-size sensitivity of the EAVS bulk path (thin rows, N = 100,000, n = 3–5 per point). Execution time falls steeply until B ≈ 10,000 and then flattens; the fixed default of B = 5,000 used in that earlier study sits 28–47% above the efficient region.

Table 1: Thin-row batch-size sweep (N = 100,000; EAVS internal timer).

B	Measured runs (s)	Mean (s)	Std. dev.
500	1.252, 1.392, 1.152	1.265	0.120
1,000	0.915, 0.898, 1.108	0.974	0.116
2,000	0.941, 0.794, 0.938	0.891	0.084
5,000	0.605, 1.166, 0.846	0.872	0.281
10,000	0.722, 0.675, 0.798	0.732	0.062
20,000	0.553, 0.540, 0.688, 0.726, 0.886	0.679	0.140

50,000	0.738, 0.621, 0.625	0.661	0.066
100,000	0.752, 0.795, 0.624	0.724	0.089

The second sweep was to determine if the efficient region is an environmental property or workload property. Each record's Payload attribute was expanded to an approximate 1.1KB of synthetic text, N was changed to 20,000 to maintain the total volume, and the sweep was repeated from B = 500 to 20,000. There were occasionally noticeable spikes in individual runs with values like 1.73 s and 1.78 s next to some of the sub-half-second neighbours, which is why Table 2 also includes the median value; we make this choice plain. The pattern is clearly evident in both directions. The curve hits the ground at B = 1000 when row width is 1000, rather than 10,000: an order-of-magnitude difference in the efficient region for an order-of-magnitude difference in the width of rows. The two normalized curves are plotted on top of each other in Fig. 2.

Table 2: Fat-row batch-size sweep (~1.1 KB/row, N = 20,000). Medians are reported because isolated shared-cloud spikes distort means.

B	Measured runs (s)	Median (s)
500	0.550, 1.557, 0.790	0.790
1,000	0.450, 0.451, 0.497	0.451
2,000	0.509, 1.778, 0.358, 0.411	0.460
5,000	0.728, 0.396, 0.381	0.396
10,000	0.339, 0.491, 1.730, 0.953	0.722
20,000	0.423, 0.490, 0.409	0.423

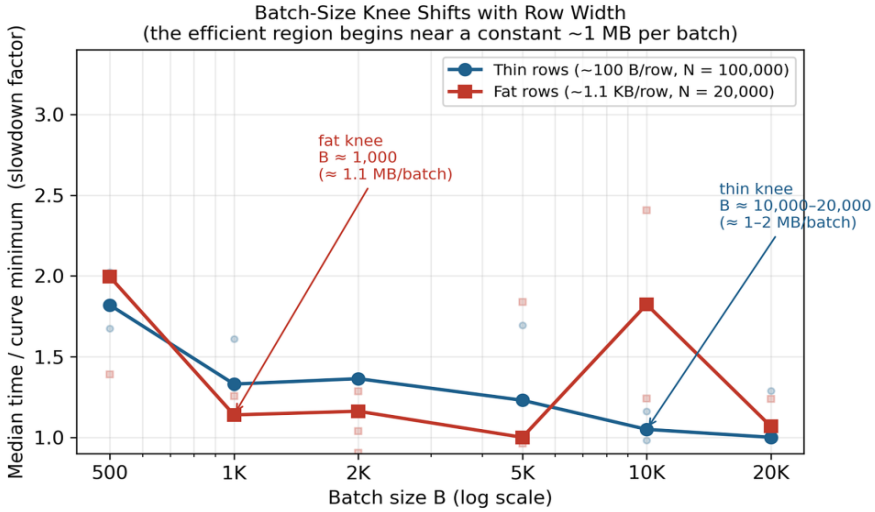

Fig. 2: The efficient region shifts with row width: thin rows flatten near B ≈ 10,000–20,000, fat rows near B ≈ 1,000. Both knees correspond to a batch payload of roughly 1 MB. Light markers are individual runs.

Multiplying the knee position by the row width exposes the regularity that motivates HBA's design. Thin rows: about 10,000 rows times about 100 bytes is about 1 MB per batch. Fat rows: about 1,000 rows times about 1.1 KB is about 1.1 MB per batch. Within the resolution of these measurements, the efficient region begins at a near-constant byte volume per batch, not a constant row count. We refer to this as the constant-byte-batch observation. It converts an intractable tuning problem, where the right row count depends on data the developer may not think about, into a simple one: aim for about a megabyte per batch, whatever the rows look like.

2.2 HBA Design

HBA is a controller wrapped around the same TDS bulk stream that EAVS [25] uses. It removes the BatchSize input entirely and replaces it with two heuristics, both derived from the study above.

Heuristic 1, the byte-budget seed. The extension already receives the full serialized payload, so the mean row width w is one division away: the JSON length over the record count. The initial batch size is then described in Equation 1.

$$B_0 = clamp\left(\frac{\beta}{w}, B_{min}, B_{max}\right) \qquad (1)$$

where β = 1,048,576 bytes is the byte budget suggested by the constant-byte-batch observation, and the clamp bounds are Bmin = 500 and Bmax = 50,000. On thin rows Equation (1) yields about 10,500; on fat rows about 950. In other words, the controller starts each workload roughly where that workload's own efficient region begins, without being told anything.

Heuristic 2, smoothed throughput feedback. The insert proceeds in chunks of the current B, each chunk a single TDS batch on the same connection and transaction. After every chunk the controller computes the chunk's throughput τk in rows per second and maintains an exponential moving average described in Equation 2.

$$\hat{\tau}_k = 0.5 \bullet \hat{\tau}_{\{k-1\}} + 0.5 \bullet \tau_k \qquad (2)$$

The adaptation rule compares the fresh reading with the smoothed history, and not with the previous single reading: if $\tau_k \geq 1.10 \bullet \hat{\tau}_{\{k-1\}}$then the batch is half the size (B ← min(1.5B,B_{max} if $\tau_k \leq 0.75 \bullet \hat{\tau}_{\{k-1\}}$then it is reduced by a third (B ← max(2B/3,B_{min}) otherwise it is not altered. These thresholds are not evenly distributed and are intentional. If there is clear evidence of growth, it requires earlier shrinkage; there is one anomalous chunk which can do whipsawing on shared infrastructure; the EMA in Equation (2) prevents whipsawing by one anomalous chunk. The multiplicative steps provide geometric exploration – the controller can make its way from 1,000 to 20,000 in roughly seven good chunks but not jump blindly.

The design is intentionally limited to a constraint that adaptive systems on the engine side do not deal with. HBA does not have access to memory counters, lock statistics or query plans. Everything about its model of the world is the sequence of chunk timings. This sensitivity study is sufficient to determine the efficient region; the evaluation in Section 4 examines whether the region is indeed efficient.

2.3 Implementation

HBA is part of the same extension in Integration Studio as EAVS [25] and is implemented as a second action. It has no batch-size parameter, and the destination table name, the record list to be inserted, the timeout period and an optional tenant identifier are its arguments. The action that is returned by Integration Studio is illustrated in Fig. 3 and returns the row count, the elapsed time in the controller, the final number of rows that the controller ended up with, and a string called BatchLog that writes each chunk as a B:throughput pair. The last output is important for the research more than for the debug; all the adaptation trajectories reported in this paper are read directly from the log of the system deployed, and not reconstructed ex post facto.

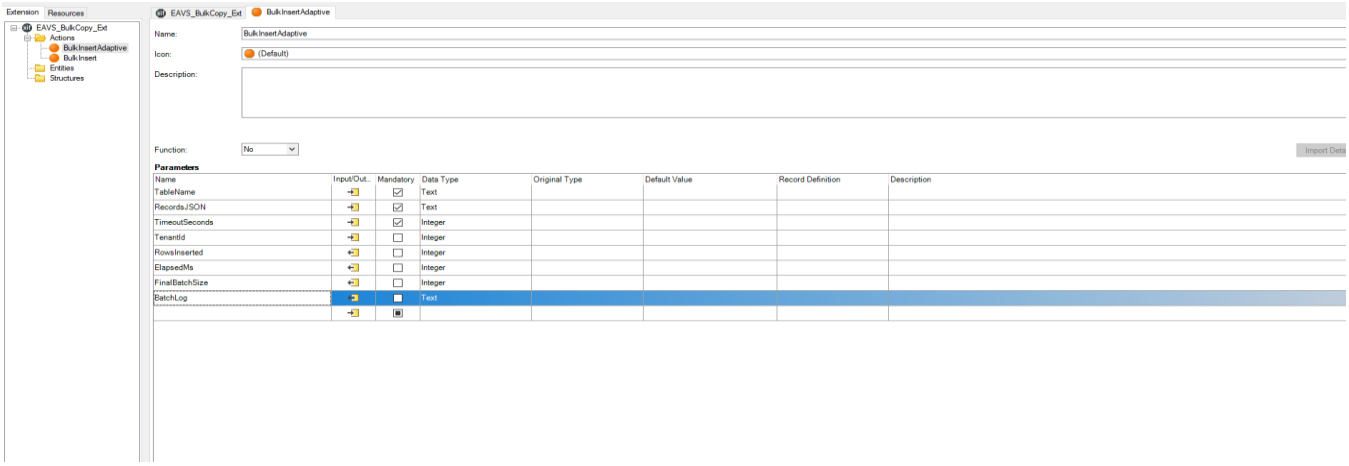

Fig. 3: The BulkInsertAdaptive action as defined in Integration Studio. The interface deliberately exposes no batch-size input; FinalBatchSize and BatchLog are first-class outputs.

The two paths differ only in batching policy because internally the action reuses the documented RuntimePublic.Db connection path, and the JSON-to-DataTable parser, and Fixed-B. The individual chunks are each copied into their own DataTable and written using SqlBulkCopy, under the transaction of the calling request, with TableLock and FireTriggers turned on, and streaming on, following the semantics of SQL Server [11]. Each TDS batch is a chunk, so the per-chunk stopwatch that is put into the controller - Equation (2) - actually measures just the unit that the controller decides on. The full controller (logging etc.) is just about 60 lines of C#.

2.4 Experimental Setup

The environment is the same as the previous companion study: OutSystems 11 on the Free Cloud, shared infrastructure, the five-column StressTest_Record entity, deterministic record generation, and the internal millisecond timer of the extension. Each configuration was run for three (five as noted) runs separated at least 10 seconds. The workload for the HBA evaluation is N = 100,000 thin rows, consistent with the sweep in Section 3A, such that the fixed-B numbers shown in Table 1 can be used as baselines. If isolated cloud spikes are observed in individual runs, we report the unaltered data and indicate the summary statistic that we used, no data is discarded.

3. Results and Discussion

3.1 HBA Against Fixed Batch Sizes

Table 3 and Fig. 4 compare the performance of HBA with three hand-tuned, fixed configurations from Table 1, the worst per sweep (B = 500), the default we shipped with EAVS (B = 5,000), and the best hand-tuned (B = 20,000). No batch-size information was provided to HBA.

Table 3: HBA against fixed batch sizes (N = 100,000 thin rows).

Configuration	Runs (s)	Mean (s)	vs. HBA
Worst manual (B = 500)	1.252, 1.392, 1.152	1.265	HBA 32% faster
EAVS default (B = 5,000)	0.605, 1.166, 0.846	0.872	on par (1.5%)
HBA (no tuning input)	0.691, 0.812, 0.847, 0.932, 1.015	0.859	—
Best manual (B = 20,000)	0.553–0.886 (n = 5)	0.679	within ~27%

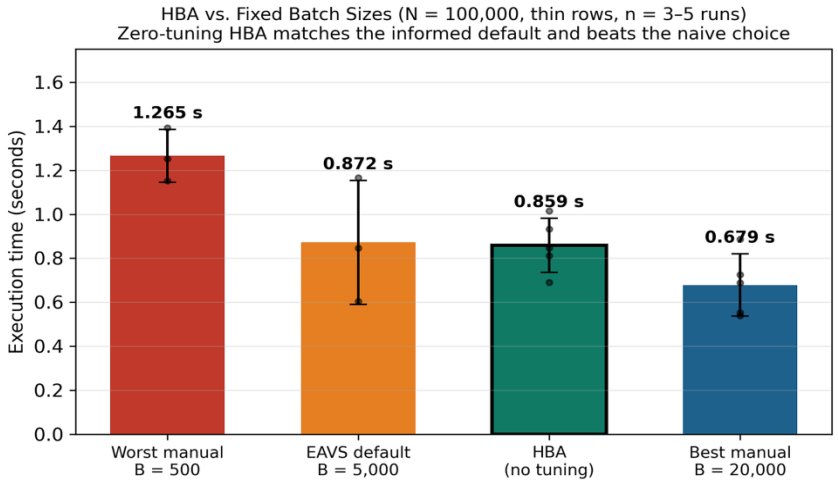

Fig. 4: HBA versus fixed batch sizes at N = 100,000. With zero tuning input, HBA beats both the naive configuration and the designers' default, and approaches the best hand-tuned value. Dots are individual runs.

The headline result is deliberately modest and, we would argue, more useful for it. HBA does not beat the best hand-tuned constant on this workload; a controller that pays exploration costs on a 100,000-row job should not be expected to. What it does is remove the tuning decision entirely while running 32 percent ahead of a naive choice, statistically on par with the informed default (0.859 versus 0.872 seconds, a difference well inside the run-to-run noise), and within about 27 percent of an optimum that took a fifteen-run sweep to locate. For a parameter for which the vendor documentation does not specify an optimum [10] without a sweep is the practically relevant target and HBA achieves it. On the workload for which tuning is more important, HBA performs even better as shown in Section 4-B. We choose B- 20,000 over B- 50,000.B = 50,000 is too high. It takes 0.661 seconds to complete the task. This represents a 2.6% difference, which falls within the range of noise levels that can occur during run-to-run operations in this environment. This is so because B= 500000 is the upper clamp bound of HBA. it is the ending point of the controller's range. It is not the interior point. When compared to this method, the capabilities of any other method shouldn’t be underestimated. However, it might be difficult to understand the results obtained using those methods.

The following is a verbatim copy of the BatchLog output for the thin-row runs (see Fig. 5(a)). All five trajectories start at B = 10,485, the value of Equation (1) evaluated on the measured row width, before any feedback was received, putting the controller in the efficient region from Table 1. These then branch off instructively, and between them they show all three of the controller's modes. Two runs rode improving throughput up to 23,590 in clean staircases. One suffered a slowdown mid-transfer, going down from approximately 290,000 to 130,000 rows per second, and then backed off to 4,660 and came back as things got even better. In the first two runs, the controller gradually returned to a target of approximately 15,700, oscillating with some cloud noise, which is the characteristic response of a self-contained feedback loop.The signature of a self-contained feedback loop is the oscillation back to the target, which occurs twice here, once with cloud noise, and once spontaneously. The fifth run was such that it saw 10 chunks with its throughput lying within the thresholds, which is the desired action when controller does not have clear instructions: "do nothing". Fig. 6 depicts the screen deployed right after one of these runs, and the raw log displayed as it was emitted by the extension.

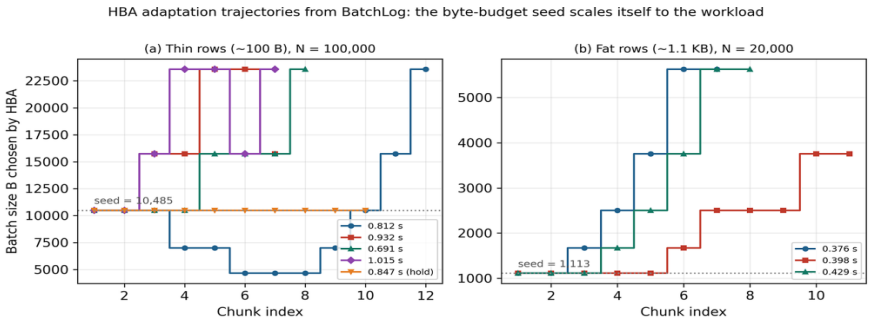

Fig. 5: Live adaptation trajectories from the extension's BatchLog output. (a) Thin rows: all five runs seed at B = 10,485 and exhibit growth, defensive dips, overshoot self-correction, and one full-run hold. (b) Fat rows: all three runs seed at B = 1,113, an order of magnitude lower, exactly as Equation (1) predicts from the wider rows.

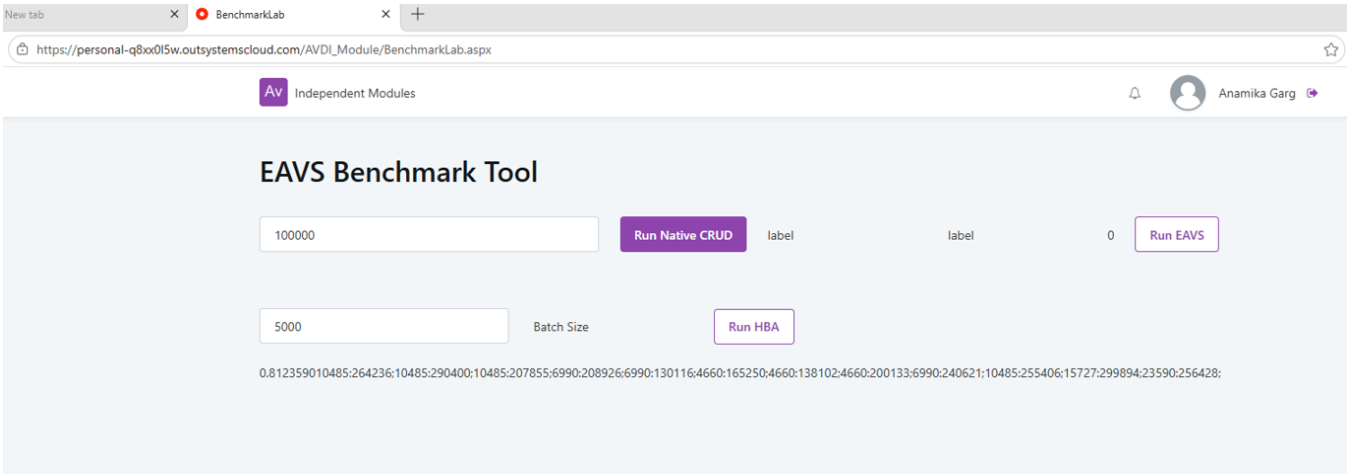

Fig. 6: The deployed benchmark screen immediately after thin-row HBA run 1 (0.812 s). The raw BatchLog rendered below the controls is the source transcript for the corresponding trajectory in Fig. 5(a).

3.2 Cross-Workload Validation of the Byte-Budget Seed

Since the fat-row workload in Table 2 is approximately 9x the size of the serialized row when taking into account JSON overhead, then Equation (1) should start at about 1/9th the size of the thin-row seed. We switched the benchmark generator back to the 1.1-KB payload, changed nothing in the extension, and ran HBA three times at N = 20,000. Every run opened at exactly B = 1,113, against 10,485 on thin rows, matching the row-width ratio; Fig. 5(b) shows the trajectories and Fig. 7 shows one of the runs as captured live in the browser. The controller then grew toward the 2,500 to 5,600 range as throughput allowed. This is the constant-byte-batch observation of Section 3A operating in the other direction: the same 1 MB budget that pointed at 10,485 thin rows points at about a thousand fat ones, and the controller found both without being told which workload it was handling.

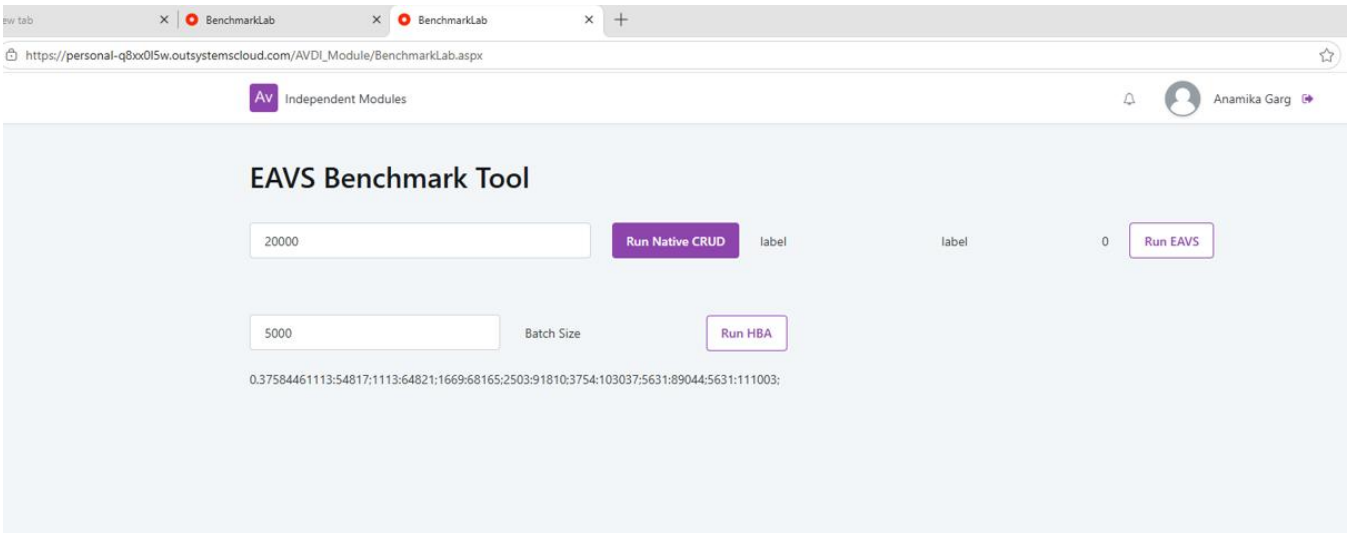

Fig. 7: Fat-row HBA run 1 as captured in the browser (0.376 s): the log opens at B = 1,113, the value Equation (1) predicts for 1.1-KB rows, with no configuration change anywhere in the system.

The performance outcome on this workload is stronger than on thin rows, which is consistent with the sensitivity data: the fat-row efficient region is narrower (Table 2), so tuning matters more, and an automatic tuner has more to win. HBA averaged 0.401 seconds across the three runs (0.376, 0.398, 0.429), within about 1 percent of the best fixed configuration found in the entire fat sweep (B = 5,000, median 0.396 s) and 49 percent faster than the naive B = 500 (median 0.790 s). On the workload where a wrong constant hurts most, the zero-tuning controller effectively matched the best constant a fifteen-run sweep could find.

Two secondary observations. First, HBA's run-to-run variance (0.691 to 1.015 s) is comparable to that of the fixed configurations in Table 1, so adaptivity did not add instability. Second, the final batch sizes chosen by the controller (15,726 to 23,590) bracket exactly the region Table 1 identifies as efficient, even though the controller never saw Table 1. The sweep and the controller agree with each other through entirely independent routes, which is about as much internal validation as a two-experiment study can offer.

3.3 Interpretation

The practical reading of these results is that batch sizing in a low-code bulk path can be taken away from the developer without giving up much performance. The constant-byte-batch observation does most of the work: once the target is expressed in bytes, a good starting point can be computed from the payload itself, and feedback only needs to handle drift and noise rather than find the region from scratch. That division of labour is why sixty lines of C# suffice.

The contrast with engine-side adaptive execution [7], [8] is worth restating. Those systems adapt because they can see the engine's internals. HBA adapts despite seeing nothing but its own chunk timings, from the sealed extension layer of a platform whose runtime the developer does not control. The result suggests that the abstraction boundary of low-code platforms, which our companion paper identified as the source of the O(N) write penalty, is permeable to feedback-based optimization even where it is opaque to instrumentation.

For practitioners the deployment story is short. BulkInsertAdaptive is a drop-in server action with one fewer parameter than the fixed path. Teams that have already tuned a batch size for a stable workload lose nothing by keeping it; teams facing varied or unknown workloads, which in enterprise integration is most of them, get near-optimal behavior without running a sweep like Section 3A, which took the better part of an evening on this environment.

3.4 Limitations

Single environment. All measurements come from one OutSystems Free Cloud personal environment on shared infrastructure. The specific constants, the 1 MB budget above all, may differ on dedicated or on-premise deployments; the method for finding them, a two-workload sweep, transfers directly, but the numbers should not be assumed portable.

Noise and repetitions. The three to five repetitions per configuration are sufficient for observation of the structure but not for making fine statistical statements; the difference between HBA and the best manual configuration in Table 3 for example, should be interpreted as indicative, not precise. The median summary shown in Table 2 was due to isolated cloud spikes; we prefer stating it that way rather than smoothing them out.

Plateau workloads. Large fixed batches work well over a broad range on the thin row workload (Table 1) and the cost of not using HBA there is bounded; this is reflected in the evaluation: HBA performs as well as, but not better than, the informed default. The evaluation of Section 4-B in the fat-row case indicates that the controller is getting better the wider the row, the binary payload or the truly memory-constrained server, but other workloads with wider rows have not been tested and may perform differently on the large-batch side.

Controller constants. The thresholds (1.10, 0.75), the steps (1.5, 2/3), and the EMA weight (0.5) were selected by reasoning and a few iterations, rather than by systematic search. They were active in every run that we performed, but tuning the parameters of the controller would improve the design, and although it is obvious that a self tuning system that needs to be tuned is subject to regression, the constants were workload independent.

4. Conclusion

This paper started from an uncomfortable measurement: the batch size we ourselves published as a default in the EAVS [25] study was 28 to 47 percent away from the empirical optimum on our own environment, and the optimum moved by an order of magnitude when the rows got wider. The redeeming regularity was that both workloads' efficient regions began near the same batch payload of about 1 MB.

HBA turns that regularity into a controller: a byte-budget seed that starts each workload inside its own efficient region, plus smoothed throughput feedback that tracks conditions from there. Deployed as a self-tuning action in the same OutSystems extension, it required no batch-size input and delivered performance on par with the informed default on thin rows, 32 percent ahead of a naive configuration, and within one percent of the best fixed value on wide rows, where tuning matters most; a cross-workload test confirmed the byte-budget seed scaling itself from 10,485 to 1,113 as the serialized row width grew roughly ninefold, and the controller's own logs document growth, holds, defensive shrinking, and overshoot correction under real shared-cloud noise. To our knowledge this is the first self-tuning batch controller demonstrated inside the extension layer of a low-code platform, and together with the earlier characterization study it completes an arc: first make bulk ingestion possible above the abstraction, then make it look after itself.

REFERENCES

1. Cabot, J., 2020. Positioning of the low-code movement within the field of model-driven engineering. Proceedings of the ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings (MODELS-C), pp. 1–3. https://doi.org/10.1145/3417990.3420210
2. Golovin, D., 2017. OutSystems as a rapid application development platform for mobile and web applications.B.Eng.thesis, Helsinki Metropolia University of Applied Sciences. https://www.theseus.fi/handle/10024/132267 (accessed 3 August 2026).
3. Sanchis, R., García-Perales, Ó.,Fraile, F., Poler, R., 2020. Low-code as enabler of digital transformation in manufacturing industry.Applied Sciences, 10(1), 12. https://doi.org/10.3390/app10010012
4. Fernandes, P., et al., 2021. Automated refactoring of unbounded queries in software automation platforms. Proceedings of the ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), pp. 1–10. https://doi.org/10.1109/MODELS-C53483.2021.00012
5. Cunha, F., 2019. Optimizing service orchestration in OutSystems.M.Sc.thesis, Instituto Superior Técnico, Universidade de Lisboa. https://doi.org/10.25749/ist.cunha.2019
6. Taipalus, T., 2024. Database management system performance comparisons: a systematic literature review. Journal of Systems and Software, 208, 111872. https://doi.org/10.1016/j.jss.2023.111872

7. Xue, M., et al., 2024. Adaptive and robust query execution for lakehouses at scale. Proceedings of the VLDB Endowment, 17(12), 3947–3960. https://doi.org/10.14778/3685800.3685818
8. Justen, D., et al., 2024. POLAR: adaptive and non-invasive join order selection via plans of least resistance. Proceedings of the VLDB Endowment, 17(6), 1350–1363. https://doi.org/10.14778/3648160.3648175
9. Rucco, C., Longo, A., Saad, M., 2026. Enhancing data ingestion efficiency in cloud-based systems: a design pattern approach. Data Science and Engineering, 11, 287–302. https://doi.org/10.1007/s41019-025-00300-2
10. Microsoft Corporation, 2023. SqlBulkCopy Class (System.Data.SqlClient). Microsoft Learn. https://learn.microsoft.com/dotnet/api/system.data.sqlclient.sqlbulkcopy (accessed 3 August 2026).
11. Microsoft Corporation, 2023. SQL Server transaction locking and row versioning guide. Microsoft Learn. https://learn.microsoft.com/sql/relational-databases/sql-server-transaction-locking-and-row-versioning-guide (accessed 3 August 2026).
12. Leonarczyk, R., Mencagli, G., Griebler, D., 2025. Self-adaptive micro-batching for low-latency GPU-accelerated stream processing.International Journal of Parallel Programming, 53(2). https://doi.org/10.1007/s10766-025-00793-4
13. Dakov, S., Dakova, M., 2025. Event-driven data orchestration: a modular approach for high-volume real-time processing. Engineering Proceedings, 100(1), 48. https://doi.org/10.3390/engproc2025100048
14. Foidl, H., Golendukhina, V., Ramler, R., Felderer, M., 2024. Data pipeline quality: influencing factors, root causes of data-related issues, and processing problem areas for developers. Journal of Systems and Software, 207, 111855. https://doi.org/10.1016/j.jss.2023.111855
15. Vogel, A., Henning, S., Ertl, O., Rabiser, R., 2023. A systematic mapping of performance in distributed stream processing systems. Proceedings of the 49th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Durres, Albania, pp. 293–300. https://doi.org/10.1109/SEAA60479.2023.00052
16. Vogel, A., Henning, S., Perez-Wohlfeil, E., Ertl, O., Rabiser, R., 2024. A comprehensive benchmarking analysis of fault recovery in stream processing frameworks. Proceedings of the 18th ACM International Conference on Distributed and Event-Based Systems (DEBS), pp. 171–182. https://doi.org/10.1145/3629104.3666040
17. Aseman-Manzar, M.-M., Karimian-Aliabadi, S., Entezari-Maleki, R., Egger, B., Movaghar, A., 2023. Cost-aware resource recommendation for DAG-based big data workflows: an Apache Spark case study. IEEE Transactions on Services Computing, 16(3), 1726–1737. https://doi.org/10.1109/TSC.2022.3203010
18. Khorram, F., Mottu, J.-M., Sunyé, G., 2020. Challenges and opportunities in low-code testing. Proceedings of the ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), pp. 1–10. https://doi.org/10.1145/3417990.3420204
19. Kourouklidis, P., Kolovos, D., Matragkas, N., Noppen, J., 2020. Towards a low-code solution for monitoring machine learning model performance. Proceedings of the ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), pp. 1–8. https://doi.org/10.1145/3417990.3420197
20. Guthardt, T., Kosiol, J., Hohlfeld, O., 2024. Low-code vs. the developer: an empirical study on the developer experience and efficiency of a no-code platform. Proceedings of the ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), pp. 1–12. https://doi.org/10.1145/3652620.3688332
21. Bock, A.C., Frank, U., 2021. Low-code platform.Business& Information Systems Engineering, 63(6), 733–740. https://doi.org/10.1007/s12599-021-00726-8
22. Hirzel, M., 2023. Low-code programming models. Communications of the ACM, 66(10), 76–85. https://doi.org/10.1145/3587691
23. Sahay, A., Indamutsa, A., Di Ruscio, D., Pierantonio, A., 2020. Supporting the understanding and comparison of low-code development platforms. Proceedings of the 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 171–178. https://doi.org/10.1109/SEAA51224.2020.00036
24. Carroll, N., Holmström, J., Matook, S., 2024. Special issue editorial: transforming business with low-code and no-code. MIS Quarterly Executive, 23(3), xvii–xxxv.
25. [25] AnamikaGarg and Sachin Patel, “EAVS: An Environment-Aware Vectorized Streaming Architecture for Eliminating the Bulk Write Bottleneck in OutSystems Low-Code Platform”, IJCISIM, vol. 18, no. 5s, pp. 1148–1157, Jul. 2026.