Physical Execution Plans for Model-Driven Engineering

Decoupling DSL Design from Runtime-Specific Representations

Francisco Martínez-Lasaca
Computer Science
Universitat Oberta de Catalunya
Barcelona, Spain
fmartinezlasa@uoc.edu

Juan De Lara
Universidad Autónoma de Madrid
Madrid, Spain
juan.delara@uam.es

Abstract

Model-Driven Engineering (MDE) raises the level of abstraction by representing systems using domain-specific languages (DSLs). However, the environment in which a DSL is developed frequently dictates how its artifacts are physically represented, accessed, and executed. Choices such as memory layouts, indexes, and runtime targets are typically inherited from the modeling framework rather than explicitly modeled. For instance, in the Eclipse Modeling Framework (EMF), models become object graphs in the Java Virtual Machine, limiting the control of DSL designers.

We argue that physical execution should be a first-class modeling concern and propose physical execution plans for Model-Driven Engineering. Our approach complements meta-models with execution plans that declare layouts, relations, indexes, access paths, and target runtimes without changing model meaning or modeling tasks. Inspired by user-schedulable languages and progressive lowering, our approach separates model meaning from execution.

We evaluate our approach over a set of modeling tasks, generating plan-specific Java and Rust backends that we compare against a conventional EMF baseline. These executors achieve task-dependent speedups of up to five orders of magnitude, while producing identical results. The measurements expose trade-offs among plans, opening a research agenda on cost-driven plan selection.

CCS Concepts

• **Software and its engineering → Model-driven software engineering**.

Keywords

Model-Driven Engineering, physical execution plans, data-oriented design, user-schedulable languages

ACM Reference Format:
Francisco Martínez-Lasaca and Juan De Lara. 2026. Physical Execution Plans for Model-Driven Engineering: Decoupling DSL Design from Runtime-Specific Representations. In *ACM/IEEE 29th International Conference on Model Driven Engineering Languages and Systems (MODELS 2026), October 04–09, 2026, Málaga, Spain.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3822455.3838772

MODELS 2026, Málaga, Spain

ACM ISBN 979-8-4007-2809-9/26/10
https://doi.org/10.1145/3822455.3838772

A Person meta-class defines attributes age, birthday, and code. How should a list $[p_0, p_1, \dots]$ of Persons be laid out in memory?

(a) Array of Structs (AoS) **(b) Struct of Arrays (SoA)**

Figure 1: Data layouts favor different access patterns: AoS favors element access, while SoA favors attribute scans, including aggregate computations (e.g., computing the average age of all persons).

1 Introduction

Model-Driven Engineering (MDE) [5] raises the level of abstraction through meta-models and domain-specific languages (DSLs) [38]. However, this abstraction rarely extends to physical execution: once a DSL is implemented in a concrete MDE framework, those choices are inherited from that framework rather than specified by the DSL designer. Thus, model meaning and physical realization become coupled. For example, the Eclipse Modeling Framework (EMF) [37] by default represents DSL artifacts as object graphs in the Java Virtual Machine (JVM), scattered in the heap and subject to garbage-collection pauses. Although the same logical elements could be arranged in memory differently, each favoring certain access patterns and penalizing others (cf. Figure 1), current frameworks make that choice for the DSL designer.

This lack of control becomes especially costly in scalability-sensitive scenarios [27], where the inherited representation may not fit the access patterns exercised by the modeling tasks. Notable examples of such scenarios include cyber-physical systems and digital twins [31], embedded systems [18], and low-code platforms [29], whose models are large and repeatedly queried, analyzed, executed [14], and managed at runtime [4].

Beyond MDE, other fields address similar problems by separating intent from physical execution. User-schedulable languages (USLs) [19], for example, separate *what* is computed from *how* it is executed, enabling performance improvements without changing program meaning. We argue that MDE needs an analogous separation: meta-models and modeling tasks should define domain meaning, while execution plans should define physical realization.

To this end, we propose physical execution plans for Model-Driven Engineering: an approach in which an execution planning DSL declares layouts, relations, indexes, access paths, semantic obligations, and target runtimes for a domain meta-model, so that

physical realization can change without altering model meaning or modeling tasks. These plans are lowered to intermediate representations (IRs), from which we generate plan-specific Java executors and an initial Rust backend behind Java's Foreign Function and Memory (FFM) API. The generated executors are validated against an EMF baseline. We evaluate the prototype on read-only tasks over synthetic models as a preliminary step toward this vision.

This *New Ideas and Emerging Results* (NIER) paper contributes:

(1) A vision and an architecture for execution-plan-driven MDE, where the physical execution strategy is reified as an explicit plan over a domain meta-model and its modeling tasks.
(2) An execution planning DSL, and a lowering pipeline that produces intermediate representations and plan-specific Java and Rust target code.
(3) A preliminary evaluation comparing conventional EMF execution with three plan-specific executors and providing initial evidence of plan sensitivity, task-dependent performance gains, and target-runtime decoupling through Rust/FFM.

Paper structure. Section 2 discusses related work; Section 3 presents the proposed MDE architecture; Section 4 evaluates the proof of concept; Section 5 outlines future plans; and Section 6 concludes.

2 Related Work

Our work relates to the following four lines of research:

▶ *Scalable modeling.* Scalability is an enduring issue in MDE [26, 27]. The community has developed numerous model persistence frameworks (e.g., CDO[1], NeoEMF [10], Morsa [13], Teneo[2]), incremental and indexed query engines (e.g., VIATRA [9], Hawk [3], Mogwaï [11]), query optimizers [1], and model fragmentation techniques [17, 23]. These approaches optimize where models are stored and how they are queried. However, although CDO and NeoEMF already decouple the physical store from the meta-model, they remain tied to EMF and, therefore, inherit the limitations of its in-memory representation.

▶ *Execution-oriented and executable modeling.* Approaches such as xMOF [30], Kermeta [24], and GEMOC [8] separate abstract syntax from behavioral semantics and execution tooling. Domain-specific tools such as Eclipse eTrice[3], rooted in Real-time Object-Oriented Modeling (ROOM) [36], make non-functional execution decisions explicit for real-time systems. Although these approaches treat execution as an explicit, tool-supported concern at the modeling level, they do not address how model data is laid out in memory.

▶ *Logical and physical separation in databases.* Data management has long separated logical intent from physical realization. In particular, query optimizers map logical schemas and declarative queries to a physical plan of storage layouts, indexes, and access paths [7]. Extensible optimizers such as Apache Calcite[4] make this mapping reusable across engines. Analytical and lakehouse systems [2] expose row- versus column-oriented storage as a schema-independent physical choice (cf. Figure 1). Polystores extend this separation across heterogeneous engines, routing each query to the best-suited backend [12]. These approaches achieve data independence, choosing layouts, indexes, and engines separately from the schema.

▶ *User-schedulable languages (USLs) and progressive lowering.* USLs separate *what* is computed from *how* it is executed: Halide [35], Exo [21, 22], GraphIt [39], Taichi [20], and TACO [25] expose schedules, layouts, or physical representations as explicit, user-controlled concerns. MLIR (Multi-Level Intermediate Representation) [28] generalizes this via progressive lowering across IRs, now exploited by Python-like systems languages such as Mojo [32] and DSL toolkits such as xDSL [16]. Apache TVM [6] adds cost models and autotuning to search for execution strategies. These systems share the core principle of data-oriented design: *the structure of the problem domain should not dictate the structure of its data in memory* [15].

None of these lines reifies physical execution per class, relation, and task as designer-declared plans with explicit semantic obligations.

3 Vision: Physical Execution Plans for MDE

In current MDE tooling, the DSL-hosting framework dictates how artifacts are physically executed, limiting the DSL designer's control. For this reason, we propose a *user-schedulable language for MDE* that empowers DSL designers by turning physical execution into a first-class modeling concern. As in USLs such as Halide [35] and Exo [22], which separate a program's specification from its execution schedule, the DSL meta-model and its modeling tasks form the specification, while our *execution planning DSL* declares their physical realization.

3.1 The physical execution modeling pipeline

Figure 2 presents the proposed physical execution modeling pipeline, which comprises the following three stages:

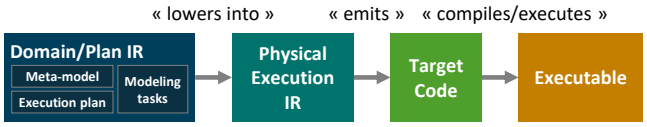


Figure 2: Proposed physical execution modeling pipeline.

▶ *1. Domain/Plan IR.* This first IR combines the domain meta-model, the modeling tasks to be executed, and the plan written in the execution planning DSL. It captures both what the model means and the physical execution intent declared by the DSL designer.

▶ *2. Physical Execution IR.* This IR, although still high-level, materializes the physical representation and access operators required by the plan. These include memory layouts such as AoS and SoA (cf. Figure 1), efficient data structures, indexes, and access paths.

▶ *3. Target Code.* This stage generates backend-specific code from the Physical Execution IR, mapping the IR to a concrete implementation. This code is then compiled and run on the target runtime selected by the plan.

Following MLIR [28], this pipeline lowers progressively across IRs, keeping each physical decision explicit and separable.

[1] https://eclipse.dev/cdo
[2] https://wiki.eclipse.org/Teneo
[3] https://eclipse.dev/etrice
[4] https://calcite.apache.org

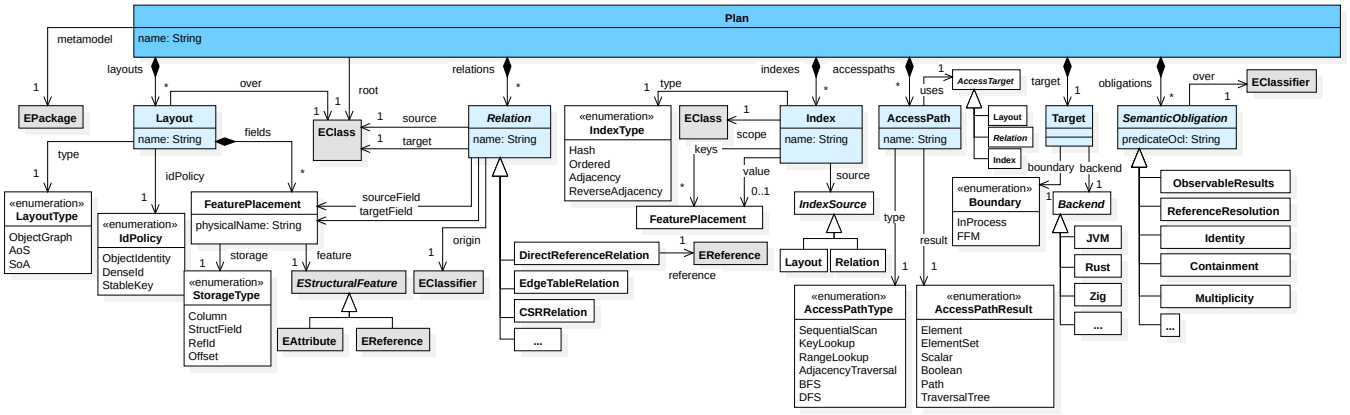


Figure 3: Meta-model of the execution planning DSL. Meta-classes from the Ecore package are highlighted in gray.

3.2 The execution planning DSL

Figure 3 presents the DSL's abstract syntax as a preliminary meta-model, while Figure 5 shows example instances. A Plan is defined over an input meta-model, a root meta-class, and a set of modeling tasks. It contains the following physical declarations.

Layouts assign each meta-class a physical representation via FeaturePlacements that bind attributes and references to physical names and storage (e.g., columns, struct fields, reference identifiers, or offsets). Relations materialize references or edge-like classifiers: a relation binds either a single EReference (a DirectReferenceRelation) or a reified edge class (e.g., a meta-class with source/target roles) projected through FeaturePlacements. It is then realized as a direct reference, edge table, or *compressed sparse row* (CSR) [34], optionally reversed to support reverse adjacency.

Index declarations add auxiliary structures over layouts or relations. Each specifies a scope, key, value, and uniqueness flag, and is lowered to a hash map for lookups. AccessPath declarations state *how* a task reaches data (e.g., sequential scan, key lookup, breadth-first search (BFS), or depth-first search (DFS)) as the physical counterpart of a task's *what*; different plans may thus resolve the same task to different access paths.

A Target fixes the execution environment through two orthogonal choices: a Backend (i.e., runtime and code generator, e.g., JVM or Rust, with Zig[5] support envisioned) and a Boundary (i.e., InProcess or FFM, the latter for native libraries reached through Java's FFM API [33]), so the same plan can be emitted for another runtime without changing its other declarations.

Finally, SemanticObligations are conditions that a plan must satisfy. For instance, lowering a relation to CSR is allowed only if traversing its offset and target arrays is observationally equivalent to traversing the original references (ObservableResults) and if each underlying EReference still resolves to its original target (ReferenceResolution). The prototype checks obligations empirically rather than enforcing them during lowering.

Taken together, execution plans let designers vary five physical dimensions: layouts, relations, indexes, access paths, and target runtimes. All such variations leave the meta-model and modeling tasks unchanged and remain subject to explicit semantic obligations. The current prototype targets read-only tasks. Supporting model writes is left as future work.

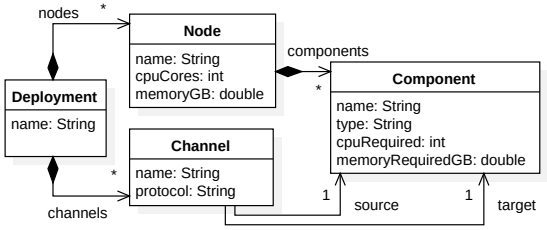


Figure 4: Component deployment meta-model.

4 Evaluation

To provide preliminary evidence for the vision, we evaluate a prototype. The implementation and all the artifacts used in the evaluation are publicly available on GitHub.[6]

We pose the following research questions:

> **RQ1.** *Does changing the physical execution plan produce measurably different execution outcomes, while leaving the meta-model and the modeling tasks fixed?*
>
> **RQ2.** *What preliminary performance trade-offs does explicit physical execution planning expose compared with conventional EMF execution?*

▶ *Experimental domain.* Figure 4 presents the component deployment meta-model used in the evaluation. It defines Deployments containing Nodes that host Components, with Channels connecting Components.

▶ *Modeling tasks.* From a given model, we define the following four modeling tasks that stress distinct access patterns:[7]

[5] https://ziglang.org

[6] https://github.com/PEP-MDE/pep-mde

[7] Note that, with $|V|$ and $|E|$ denoting the number of components and channels, respectively, T1 is $O(|V|)$; T2 is $O(1)$ if a hash index is available, $O(|V|)$ otherwise; and T3 and T4 are $O(|V|+|E|)$ in the worst case, but can be substantially accelerated if the graph structure is favorable or if adjacency structures are available.

```
plan LookupPlan for DeploymentMM rooted at Deployment {
  layout Component as SoA {
    fields @id as componentId, name, type,
           cpuRequired, node.@id as nodeId
  }
  layout Channel as SoA {
    fields @id as channelId, name, source.@id as sourceId,
    target.@id as targetId, protocol
  }
  relation ChannelGraph over Channel (sourceId → targetId) as EdgeTable
  index ComponentByName on Component {
    scope Deployment; key name; value componentId; unique
  }
  access AvgCpu by scan Component.cpuRequired
  access ByName by lookup ComponentByName
  access Reachability by scan ChannelGraph
  access NoCycles by scan ChannelGraph
  backend JVM
  validate ObservableResults
  validate ReferenceResolution
}
```

(a) P1. Lookup plan

```
plan TraversalPlan for DeploymentMM rooted at Deployment {
  layout Component as SoA {
    fields @id as componentId, name, type,
           cpuRequired, node.@id as nodeId
  }
  layout Channel as SoA {
    fields @id as channelId, name, source.@id as sourceId,
    target.@id as targetId, protocol
  }
  relation ChannelGraph over Channel (sourceId → targetId) as CSR
  relation ChannelInverse over Channel (sourceId → targetId) as CSR
           direction Reverse
  access AvgCpu by scan Component.cpuRequired
  access ByName by scan Component.name
  access Reachability by bfs ChannelGraph
  access NoCycles by dfs ChannelGraph
  backend JVM
  validate ObservableResults
  validate ReferenceResolution
}
```

(b) P2. Traversal plan

```
// Same contents as P2
// but targeting Rust through FFM
plan TraversalRustPlan
  for DeploymentMM
  rooted at Deployment {
  // layout...
  // relation...
  // access...
  backend Rust {
    boundary FFM
  }
  validate ObservableResults
  validate ReferenceResolution
}
```

(c) P3. Traversal plan + Rust

Figure 5: Three execution planning DSL plans for the same meta-model.

T1	Compute the average number of CPUs required across all components · *sequential column scan over Component.cpuRequired*
T2	Find a component by name · *random key lookup*
T3	Compute transitive reachability via BFS over channels · *graph traversal*
T4	Detect cycles in the channel graph via DFS · *graph traversal*

▶ *Physical execution plans.* The execution planning DSL complements the meta-model with a physical execution profile for each workload. By varying layout, relation, index, access-path, and target declarations, the DSL designer selects a profile without altering the meta-model or the modeling tasks. This evaluation uses three example plans (cf. Figure 5) — one lookup-oriented, one traversal-oriented, and one Rust-targeted:

▶ The lookup plan (5a) materializes a hash index from `name` to `componentId`, making T2 constant-time but leaving graph tasks to scan channels.

▶ The traversal plan (5b) instead materializes channels as a CSR representation (plus an inverted CSR for reverse adjacency), making T3/T4 scale with the traversed subgraph but leaving T2 as a linear scan.

▶ Finally, the Rust traversal plan (5c) reuses the traversal declarations but sets the backend to `Rust` with the `FFM` boundary, so calls for the column-scan task (T1) and graph-traversal tasks (T3 and T4) cross the boundary.

These plans yield the executors P1–P3, which we compare against the EMF baseline (P0). The executors cover a subset of the DSL's design space — SoA layouts; edge-table and CSR relations; a hash index; scan, lookup, BFS, and DFS access paths; and the JVM and Rust backends:

#	Runtime & physical representation	Access paths
P0	Java · EObject graph	object references
P1	Java · SoA + hash index + edge table	scan · hash lookup · edge-table scan
P2	Java · SoA + CSR + inverted CSR	scan · linear lookup · BFS/DFS
P3	Rust via FFM · SoA + CSR + inverted CSR	scan · linear lookup · BFS/DFS

▶ *Experimental setup.* Benchmarks ran on a Windows 11 laptop (Intel i7-7700HQ @ 2.80 GHz, 16 GB RAM) using OpenJDK 25 and Rust 1.96. Plans were lowered to JVM or Rust code at build time

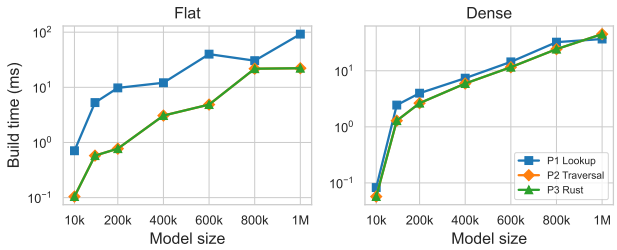


Figure 6: Build time of auxiliary structures (indexes and CSR) vs. model size.

using JavaPoet.[8] P3 was invoked through Java's FFM API (Project Panama) [33], while all other executors ran on the JVM. Models were generated deterministically for two topologies — *flat* (one node, one channel per component) and *dense* (one node per 100 elements, 2× channels per component) — at seven sizes from 10k to 1M elements. Both topologies keep the channel graph connected, so T3 and T4 traverse a size-proportional subgraph.

Each measurement was preceded by seven warm-up iterations and repeated 11 times; we report the median. T2 was measured as a batch of 64 lookups for reliability, whereas the other tasks were measured as single passes. Reported task times exclude the one-time construction of auxiliary structures, whose cost stays below 100 ms even for the largest model and the least favorable plan (Figure 6) and is amortized across repeated task executions over the same representation.

4.1 RQ1: Physical-plan sensitivity

To evaluate RQ1, we built P1 and P2 from the same deployment models, executed all four modeling tasks on each, and compared their wall-clock times. Figure 8 contrasts P1 and P2 at 10k elements on both topologies.

Under the lookup plan, T2 (name resolution) is answered through the `ComponentByName` hash index in 0.004 ms, versus 0.56 ms for the

[8] https://github.com/palantir/javapoet

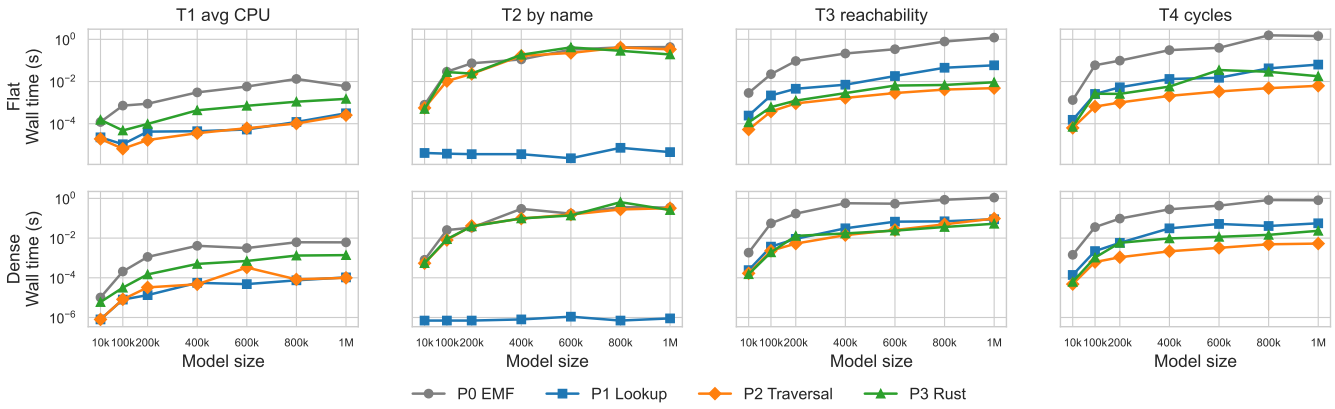
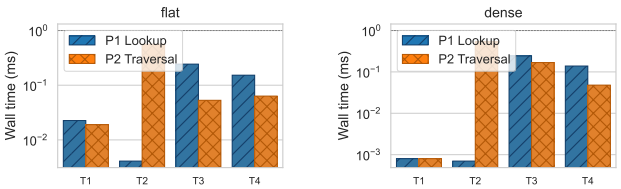


Figure 7: RQ2: Wall-clock time vs. model size for all generated executors and modeling tasks.



(a) Flat topology **(b) Dense topology**

Figure 8: RQ1: Wall-clock time for the lookup and traversal plans across all modeling tasks at 10k elements.

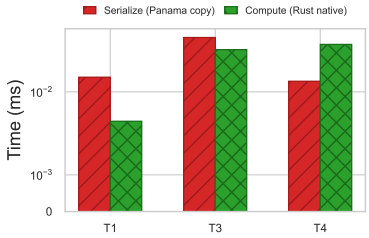


Figure 9: RQ2: Serialization and compute time for the P3 Rust backend at 10k elements.

traversal plan, which lacks the index and must scan the component column. The lookup plan is thus over 100× faster even at this small model size. Conversely, T3 (BFS reachability) and T4 (DFS cycle detection) run faster under the traversal plan's CSR representation than under the lookup plan's edge-table scans: at 10k on the flat topology, the traversal plan completes T3 in 0.05 ms versus 0.24 ms (4.6×) and T4 in 0.06 ms versus 0.15 ms (2.4×); on the dense topology the gaps are 1.4× (T3) and 2.9× (T4). T1 (column scan on SoA) behaves identically under both plans, confirming that performance differs only where the physical declarations differ.

To answer RQ1, changing the physical execution plan while leaving the meta-model and modeling tasks fixed produces measurably different execution outcomes. The lookup plan resolves names over 100× faster, while the traversal plan runs graph traversals 1.4–4.6× faster at 10k elements. Performance diverges only where the physical declarations differ, giving preliminary evidence of plan sensitivity in the evaluated tasks.

4.2 RQ2: Performance trade-offs

To evaluate RQ2, we examined the performance trade-offs exposed by these plans. In particular, P2 isolates the effect of the CSR-oriented execution plan on the JVM, while P3 reuses that plan to expose the cost of crossing a Rust/FFM boundary.

Figure 7 reports wall-clock time versus model size for all four executors. Relative to the EMF baseline (P0) at 1M elements, the execution plans yield large and consistent gains on their target tasks. For the column scan (T1), the SoA layout delivers 23× (flat) to 62× (dense) speedups over EMF on the JVM. For by-name resolution (T2), the lookup plan's hash index cuts the time by about five orders of magnitude: 0.004 ms versus 431 ms on flat at 1M. The traversal plan declares no index and stays at EMF's linear-scan speed (within 1.3×), confirming that the benefit is plan-specific. For graph traversal, the CSR representation accelerates T4 (DFS cycle detection) by 153× (dense) to 228× (flat) and T3 (BFS reachability) by 11× (dense) to 250× (flat). The gaps are smaller on the dense topology because its randomly wired adjacency also degrades cache locality for the array-based traversal, not only for EMF. The Rust backend (P3) remains 4–131× faster than EMF but is generally slower than the JVM CSR plan (P2) because each invocation crosses the FFM boundary. In all measurements and in both topologies, every generated executor produced checksums identical to the EMF baseline, providing evidence that the plans preserve the observable results of the tasks while changing physical execution.

Figure 9 decomposes the cost of the Rust–Java bridge for P3 at 10k elements into serialization (copying data across the Panama boundary) and native computation. For T1, serialization dominates (about 0.017 ms versus 0.004 ms of compute on the flat topology), making P3 slower than the JVM executors on scan-like tasks whose computation is trivial. For T3 and especially T4, native computation

dominates the boundary copy (T4: about 0.052 ms compute versus 0.015 ms serialization). As a result, the Rust/FFM backend tracks the CSR-oriented traversal behavior of P2 while remaining far ahead of EMF. P3 therefore shows that the target runtime can be made explicit in the execution plan, and that native execution is not automatically faster than JVM execution: crossing the boundary carries a serialization cost that is amortized only when the modeling task involves substantial native computation.

To answer RQ2, explicit physical execution planning exposes clear preliminary trade-offs compared with conventional EMF execution. At 1M elements, the SoA layout accelerates the column scan (T1) by up to 62×, a hash index accelerates key lookup (T2) by up to $\sim 10^5\times$, and CSR accelerates graph traversals (T3 and T4) by up to 250×. These gains are plan-specific, and an index-less plan stays at EMF's lookup speed. Their main price is a one-time build below 100 ms. Targeting Rust keeps the traversal gains but adds a serialization cost at the FFM boundary.

4.3 Threats to validity

We identify the following threats to validity in our evaluation:

▶ *Construct validity.* We measured execution efficiency using wall-clock time and correctness via checksums. Because wall-clock time can combine algorithmic efficiency with JVM overheads (e.g., garbage-collection pauses), we used warm-up iterations to stabilize execution. Furthermore, while checksums validate state equivalence only, not side effects, they establish that the execution plans preserve the observable results of the original modeling tasks. Memory footprint profiling remains future work.

▶ *Internal validity.* Comparing specialized data structures against a baseline as generic as EMF introduces the risk that speedups stem simply from bypassing its abstraction layers. However, as we evaluated multiple distinct execution plans, the performance variation among our generated executors indicates the direct impact of the physical layout choices.

▶ *External validity.* The evaluation relies on a single meta-model, four read-only tasks, and synthetic topologies. While synthetic models cannot capture the complexity found in real-world artifacts, testing on both flat and dense topologies up to 1M elements covers two structural extremes. Furthermore, we deliberately excluded write-intensive tasks, as our focus remains on query-heavy and batch-processing MDE scenarios, which are common in practice.

▶ *Conclusion validity.* To ensure measurement reliability despite JVM non-determinism and timer resolution limits, we batched fast operations, such as the 64 lookups in T2, and used medians to filter out outliers. Given the size of the observed differences (up to five orders of magnitude), these safeguards provide preliminary confidence in the reported trends.

5 Future Plans

Building on these results, we envision three directions toward a full-length paper.

First, we will split the collapsed pipeline of Section 3 into finer stages as tasks demand. A Logical Query/Traversal IR, kept separate from a Physical Storage IR, would serve composed tasks such as filters, joins, and indexed selections. Backend-specific IRs below Target Code would support additional targets such as Zig, WebAssembly, or LLVM. Finer stages would also provide natural checkpoints for semantic obligations, since each lowering step could enforce those it affects.

Second, we will study workload-aware plan selection. Indexes speed up lookups but add build cost, whereas CSR speeds up traversal at the expense of other access patterns. To navigate such trade-offs, we envision cost models and auto-scheduling that recommend plans from a workload description, following the auto-tuning practice of compilers such as TVM [6], with profiling data from past executions feeding these models. Ultimately, this direction could lead to a *multi-runtime* setup that persists several representations and routes each task to the best-suited one, analogous to polystores [12].

Finally, we will broaden validation. We plan to cover more DSL families, real-world models beyond our synthetic generators, and write/update workloads with their incremental-maintenance costs. We will also measure the memory footprint, compare against optimized repositories and frameworks such as NeoEMF [10] and Hawk [3], as well as hand-written executors, and characterize which DSL transformations preserve observable behavior.

6 Conclusion

This NIER paper proposes physical execution planning for MDE, an approach that reifies physical execution choices as explicit modeling artifacts. A DSL designer can declare these choices explicitly instead of inheriting them from the modeling framework, while preserving the domain meta-model and modeling tasks.

Our proof of concept implements a collapsed lowering pipeline: it produces Domain/Plan IR and Physical Execution IR artifacts, generates plan-specific Java code and an initial Rust/FFM backend, and compares observable task results against an EMF baseline. The preliminary evaluation shows that, for the read-only tasks considered, different execution plans over the same meta-model favor different task families, and that generated executors can improve performance by up to 250× on graph traversal and by about $10^5\times$ on indexed lookups compared with conventional EMF execution.

These results suggest that physical execution planning is promising for selected scalability-sensitive MDE workloads and motivate a broader research agenda on user-schedulable modeling infrastructures, richer modeling tasks, additional targets, and formal preservation conditions.

Acknowledgments

This article has been funded by the SE4GenAI (PID2023-147592OB-I00) and BOOSTER (PID2024-155231OB-I00) projects funded by MCIU / AEI /10.13039/501100011033 / FEDER, EU.

References

[1] Qurat Ul Ain Ali, Dimitris Kolovos, and Konstantinos Barmpis. 2021. Identification and Optimisation of Type-Level Model Queries. In *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE, 751–760.

[2] Michael Armbrust, Ali Ghodsi, Reynold Xin, and Matei Zaharia. 2021. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. In *11th Annual Conference on Innovative Data Systems Research (CIDR '21)*.

[3] Konstantinos Barmpis and Dimitrios S. Kolovos. 2013. Hawk: towards a scalable model indexing architecture. In *Proceedings of the Workshop on Scalability in Model Driven Engineering, Budapest, Hungary, June 17, 2013*. ACM, 6.

[4] Gordon Blair, Nelly Bencomo, and Robert B. France. 2009. Models@run.time. *Computer* 42, 10 (2009), 22–27.

[5] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. 2017. *Model-Driven Software Engineering in Practice, Second Edition*. Morgan & Claypool Publishers.

[6] Tianqi Chen et al. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*. USENIX Association, 578–594.

[7] Edgar F. Codd. 1970. A relational model of data for large shared data banks. *Commun. ACM* 13, 6 (1970), 377–387.

[8] Benoit Combemale, Olivier Barais, and Andreas Wortmann. 2017. Language engineering with the GEMOC studio. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. IEEE, 189–191.

[9] György Csertán, Gábor Huszerl, István Majzik, Zsigmond Pap, András Pataricza, and Dániel Varró. 2002. VIATRA – Visual automated transformations for formal verification and validation of UML models. In *Proceedings 17th IEEE International Conference on Automated Software Engineering*. IEEE, 267–270.

[10] Gwendal Daniel, Gerson Sunyé, Amine Benelallam, Massimo Tisi, Yoann Vernageau, Abel Gómez, and Jordi Cabot. 2017. NeoEMF: A multi-database model persistence framework for very large models. *Science of Computer Programming* 149 (2017), 9–14.

[11] Gwendal Daniel, Gerson Sunyé, and Jordi Cabot. 2016. Mogwaï: a framework to handle complex queries on large models. In *2016 IEEE Tenth International Conference on Research Challenges in Information Science (RCIS)*. IEEE, 1–12.

[12] Jennie Duggan, Aaron J. Elmore, Michael Stonebraker, Magda Balazinska, Bill Howe, Jeremy Kepner, Sam Madden, David Maier, Tim Mattson, and Stan Zdonik. 2015. The BigDAWG Polystore System. *ACM SIGMOD Record* 44, 2 (Aug. 2015), 11–16.

[13] Javier Espinazo-Pagán, Jesús Sánchez Cuadrado, and Jesús García Molina. 2011. Morsa: A Scalable Approach for Persisting and Accessing Large Models. In *Model Driven Engineering Languages and Systems, 14th International Conference, MODELS 2011, Wellington, New Zealand, October 16-21, 2011. Proceedings*. Springer, 77–92.

[14] Joeri Exelmans, Ciprian Teodorov, and Hans Vangheluwe. 2025. Operation-based versioning as a foundation for live executable models. *Software and Systems Modeling* 24, 3 (2025), 721–739.

[15] Richard Fabian. 2018. *Data-Oriented Design: Software Engineering for Limited Resources and Short Schedules*. Richard Fabian.

[16] Mathieu Fehr et al. 2025. xDSL: Sidekick Compilation for SSA-Based Compilers. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization, CGO 2025, Las Vegas, NV, USA, March 1-5, 2025*. ACM, 179–192.

[17] Antonio Garmendia, Esther Guerra, Juan de Lara, Antonio García-Domínguez, and Dimitris S. Kolovos. 2019. Scaling-up domain-specific modelling languages through modularity services. *Information and Software Technology* 115 (2019), 97–118.

[18] Abel Gómez et al. 2020. Scalable modeling technologies in the wild: an experience report on wind turbines control applications development. *Software and Systems Modeling* 19, 5 (2020), 1229–1261.

[19] Mary Hall, Cosmin E. Oancea, Anne C. Elster, Ari Rasch, Sameeran Joshi, Amir Mohammad Tavakkoli, and Richard Schulze. 2025. Scheduling Language Chronology: Past, Present, and Future. *ACM Transactions on Architecture and Code Optimization* 22, 3, Article 100 (Sept. 2025), 31 pages.

[20] Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. 2019. Taichi: a language for high-performance computation on spatially sparse data structures. *ACM Transactions on Graphics* 38, 6 (2019), 201:1–201:16.

[21] Yuka Ikarashi et al. 2025. Exo 2: Growing a Scheduling Language. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2025, Rotterdam, The Netherlands, 30 March 2025 – 3 April 2025*. ACM, 426–444.

[22] Yuka Ikarashi, Gilbert Louis Bernstein, Alex Reinking, Hasan Genc, and Jonathan Ragan-Kelley. 2022. Exocompilation for productive programming of hardware accelerators. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 – 17, 2022*. ACM, 703–718.

[23] Karim Jahed, Mojtaba Bagherzadeh, and Juergen Dingel. 2021. On the benefits of file-level modularity for EMF models. *Software and Systems Modeling* 20, 1 (2021), 267–286.

[24] Jean-Marc Jézéquel, Olivier Barais, and Franck Fleurey. 2009. Model driven language engineering with kermeta. In *International Summer School on Generative and Transformational Techniques in Software Engineering*. Springer, 201–221.

[25] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–29.

[26] Dimitrios S. Kolovos et al. 2013. A research roadmap towards achieving scalability in model driven engineering. In *Proceedings of the Workshop on Scalability in Model Driven Engineering, Budapest, Hungary, June 17, 2013*. ACM, 2.

[27] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. 2009. The Grand Challenge of Scalability for Model Driven Engineering. In *Models in Software Engineering*. Springer, 48–53.

[28] Chris Lattner et al. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2021, Seoul, South Korea, February 27 – March 3, 2021*. IEEE, 2–14.

[29] Francisco Martínez-Lasaca, Pablo Díez, Esther Guerra, and Juan de Lara. 2026. A model and workflow-driven approach for engineering domain-specific low-code platforms and applications. *Software and Systems Modeling* 25, 2 (2026), 579–614.

[30] Tanja Mayerhofer, Philip Langer, Manuel Wimmer, and Gerti Kappel. 2013. xMOF: Executable DSMLs based on fUML. In *International Conference on Software Language Engineering*. Springer, 56–75.

[31] Judith Michael et al. 2025. Model-Driven Engineering for Digital Twins: Opportunities and Challenges. *Systems Engineering* 28, 5 (2025), 659–670.

[32] Modular Inc. 2026. Mojo. https://mojolang.org. Accessed: 2026-07-01.

[33] OpenJDK Community. 2024. JEP 454: Foreign Function & Memory API. https://openjdk.org/jeps/454. Accessed: 2026-07-01.

[34] Sergio Pissanetzky. 1984. *Sparse Matrix Technology*. Academic Press, London, UK.

[35] Jonathan Ragan-Kelley et al. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. ACM, 519–530.

[36] Bran Selic. 1996. Real-time Object-Oriented Modeling. *IFAC Proceedings Volumes* 29, 5 (1996), 1–6.

[37] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. 2008. *EMF: Eclipse Modeling Framework*. Pearson Education.

[38] Andrzej Wąsowski and Thorsten Berger. 2023. *Domain-Specific Languages – Effective Modeling, Automation, and Reuse*. Springer.

[39] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. GraphIt: a high-performance graph DSL. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–30.