Universal Library of Innovative Research and Studies Research Article

ISSN: 3065-0003 | Volume 3, Issue 3

Open Access | PP: 44-52

DOI: https://doi.org/10.70315/uloap.ulirs.2026.0303008

A Cost Taxonomy for Template-Based Code Generators Across Multiple Target Technologies

Anna Naumova

Independent Researcher, Tampa, Florida, United States.
ORCID: 0009-0001-9641-1757

Abstract

Template-based code generators promise a single declarative model rendered into many implementation artefacts, and the promise strengthens as the number of target technologies grows: one model, several languages, frameworks and persistence stacks. Reported outcomes do not match that promise evenly. A survey of 450 practitioners found productivity effects ranging from a 27% loss to an 800% gain, with most organisations reporting increases of 20–30%. A spread that wide suggests that the quantity being estimated is not a single quantity. This review argues that the maintenance cost of a multi-target generator decomposes into five classes that behave differently, and that the classification axis which makes the difference visible is scaling behaviour: what each cost scales with, and whether it is a one-off stock or a recurring rate. The five classes are replication cost, paid once per target and growing with the number of targets; consistency cost, which recurs with every model change and multiplies by the number of targets; exception cost, which recurs and grows faster than linearly as special cases accumulate inside templates; boundary cost, a one-off investment in the seam between generated and hand-written code that does not add to the total so much as bound the exception class; and drift cost, which recurs at the pace at which each target ecosystem evolves. Thirty-six verified sources support the classification, fifteen of them examined in full text, and a public MIT-licensed generator repository serves as a checkable illustration. The practical consequence is that only the replication class is visible when a team decides to add a target technology, while three of the remaining four are rates that become apparent later, so adoption decisions taken on the visible class alone understate the commitment. The review closes with five named open problems and a first step for each.

Keywords: *Code Generation, Extension Points, Maintenance Cost, Model-Driven Engineering, Software Templates.*

INTRODUCTION

A template-based code generator turns a declarative model into source code by binding model elements to templates written against a target technology. The economic case is amplification: a small quantity of template and generator source produces a much larger quantity of working code, and the same model can be rendered again whenever the model changes. In the public repository examined below as an illustration, 30 templates totalling 3,372 lines, together with a generator of 10,919 lines, correspond to 159,218 lines of generated code, an amplification of roughly eleven to one.

The case strengthens when one generator serves several target technologies. A model expressed once should, in principle, yield a service in one language, a persistence schema for several database engines, and an interface layer in a framework chosen per project. This is the configuration in which generator platforms are sold and in which they are adopted.

Measured outcomes are inconsistent with a stable cost model. Whittle et al. (2014), surveying 450 model-driven engineering practitioners and interviewing 22 more across 17 companies in 9 industrial sectors, recorded productivity effects spanning a 27% loss to an 800% gain, while most companies reported increases between 20% and 30%. A dispersion of that magnitude across a single technology family is difficult to attribute to measurement noise. It is more consistent with organisations aggregating costs that obey different laws into one figure, and doing so in different proportions.

This review proposes one decomposition consistent with that dispersion. Its contribution is a taxonomy of five maintenance-cost classes for multi-target template-based

Citation: Anna Naumova, "A Cost Taxonomy for Template-Based Code Generators Across Multiple Target Technologies", Universal Library of Innovative Research and Studies, 2026; 3(3): 44-52. DOI: https://doi.org/10.70315/uloap.ulirs.2026.0303008.

generators, distinguished not by the artefact they touch but by **how each scales**: with the number of target technologies, with the frequency of model change, with the number of accumulated exceptions, or with the release cadence of the target ecosystems. Two synthesis matrices apply the taxonomy across cost classes and across generator families, and a closing section names five unresolved problems with a first step for each.

The classification rests on the observation, developed in Section 3.9, that two classes are stocks, only one of which enters an onboarding estimate, while three are rates that enter nothing. No classification of generator maintenance cost by scaling behaviour was found in the cost and adoption literature on model-driven engineering; where individual classes appear, they appear separately, without the stock-rate distinction that gives them their practical force.

MATERIALS AND METHODS

Candidate literature was retrieved from OpenAlex, which aggregates Crossref, PubMed, DOAJ, arXiv and other indexes, in three rounds: the review topic and each draft taxonomy class over 2015–2026; a second round reopened to 2005–2026, because the industrial cost literature on model-driven engineering is concentrated in 2008–2014; and a third round targeting two thin areas, the canonical industrial cost studies and the integration of generated with hand-written code.

Rounds one and two produced 627 records after deduplication by digital object identifier and then by title, reduced to 547 by filters on year, presence of an abstract and publication type, and to 43 by screening against the inclusion criteria, of which 34 were retained. The third round contributed 280 records, of which 8 were admitted by hand. Four records were later removed as off-topic and two dropped for lacking citation-grade metadata, giving **36 cited sources**. Inclusion required a source to concern generation of code from models, templates or domain-specific languages, or the engineering, testing, evolution and industrial adoption of such generators; to address cost, effort, productivity, maintenance, correctness assurance, or the generated-to-hand-written boundary; to be empirical, industrial, survey or systematic rather than a tool announcement; and to be peer-reviewed. Metadata was taken from OpenAlex and re-checked against Crossref, which was treated as authoritative where the two disagreed.

Fifteen of the 36 sources were examined in full text, and every numeric value attributed to them was confirmed against the sentence containing it in the source. The remaining 21 were analysed at the level of their reported results, so their placement within the taxonomy rests on abstracts and carries corresponding uncertainty. The repository used throughout as a concrete instance is Nox (github.com/NoxOrg/Nox), MIT-licensed; its measurements were obtained by cloning and counting, with line counts taken as newline counts over file contents and generated code identified by path, so that any reader can reproduce them.

DISCUSSION

What A Multi-Target Generator Actually Contains

The illustrative repository renders one declarative model into domain entities, a query interface, a database schema, validation and infrastructure code. Its scale is modest and typical: 3,295 commits, 3,534 source files, 283 declarative model files, a type system of 58 types, and support for 3 database providers. Correctness is held by 978 single-case and 216 parameterised test declarations, counted as occurrences of the corresponding xUnit attributes in the test sources, together with 49 golden reference files against which generated output is compared.

Of the code base, 1,925 files and 159,218 lines are generated, against 1,609 files and 102,566 lines written by hand. Generated code is therefore **60.8% of the code base by lines**, produced from 3,372 lines of templates and 10,919 lines of generator source.

The amplification is real, which is why the approach is adopted. A hand-written remainder of 102,566 lines persists alongside it, which is why the boundary between the two, treated in Section 3.7, is a cost centre rather than a detail.

How this Review Differs from Prior Surveys

Model-driven engineering has been surveyed repeatedly, and the surveys divide into three groups, none of which produces the decomposition proposed here.

The first group maps the field conceptually. Rodrigues da Silva (2015) organises model-driven engineering around a unified conceptual model, and Bucchiarone et al. (2020) collect the community's grand challenges. Both describe what the discipline contains rather than what operating it costs.

The second group asks the cost question empirically and answers it in aggregate. Mohagheghi and Dehlen (2008) reviewed industrial experience reports and found the evidence base thin. Hutchinson et al. (2011, 2014) and Whittle et al. (2013, 2014) built the largest practitioner data set in the area, and Mohagheghi et al. (2013) studied acceptance among developers. This work established that outcomes vary enormously, and Whittle et al. (2017) traced much of the variation to tooling. What it reports is a single effort or productivity figure per organisation, which is the aggregation this review argues obscures the mechanism.

The third group examines one cost mechanism in isolation without relating it to the others: transformation modularity (Hemel et al., 2010), generator and language composition (Ringert et al., 2015), metamodel evolution (Iovino et al., 2012), concurrent versioning of models and metamodels (Cicchetti et al., 2012), and domain-specific modelling practice in a particular sector (Nordmann et al., 2016).

The gap is therefore between aggregate cost figures and

isolated mechanisms. This review occupies it by classifying the mechanisms according to how each scales, which is what allows the aggregate figures to be read as different mixtures rather than as contradictory results.

The Taxonomy

The five classes are defined in Table 1. Each is named for what drives it, and characterised by whether it is a stock, paid once, or a rate, paid continuously.

Table 1. Five cost classes for a multi-target template-based generator

Class	Stock or rate	Scales with	What it buys	Observable symptom	Principal control
C1 Replication	Stock, once per target	Number of target technologies, N	A working back end for one more technology	Effort concentrated in a project onboarding a target	Reuse of a shared type system and model schema
C2 Consistency	Rate	N × frequency of model change	Confidence that all targets mean the same thing	Test and reference suites growing faster than features	A single oracle strategy applied uniformly
C3 Exception	Rate, faster than linear	Accumulated exceptions × templates touched	Coverage of cases the model does not express	Templates dense with conditionals	Diversion of exceptions across the boundary, C4
C4 Boundary	Stock, and a bound on C3	Paid once per generator or per target	Regeneration that does not destroy hand-written work	Disputes over which artefacts may be edited	An explicit, documented integration mechanism
C5 Drift	Rate	N × release cadence of each target ecosystem	Continued validity of generated output	Templates failing against a new framework version	Version pinning and staged migration

Figure 1 renders the same structure as a hierarchy, with C4 shown against C3 as a bound rather than as an addend.

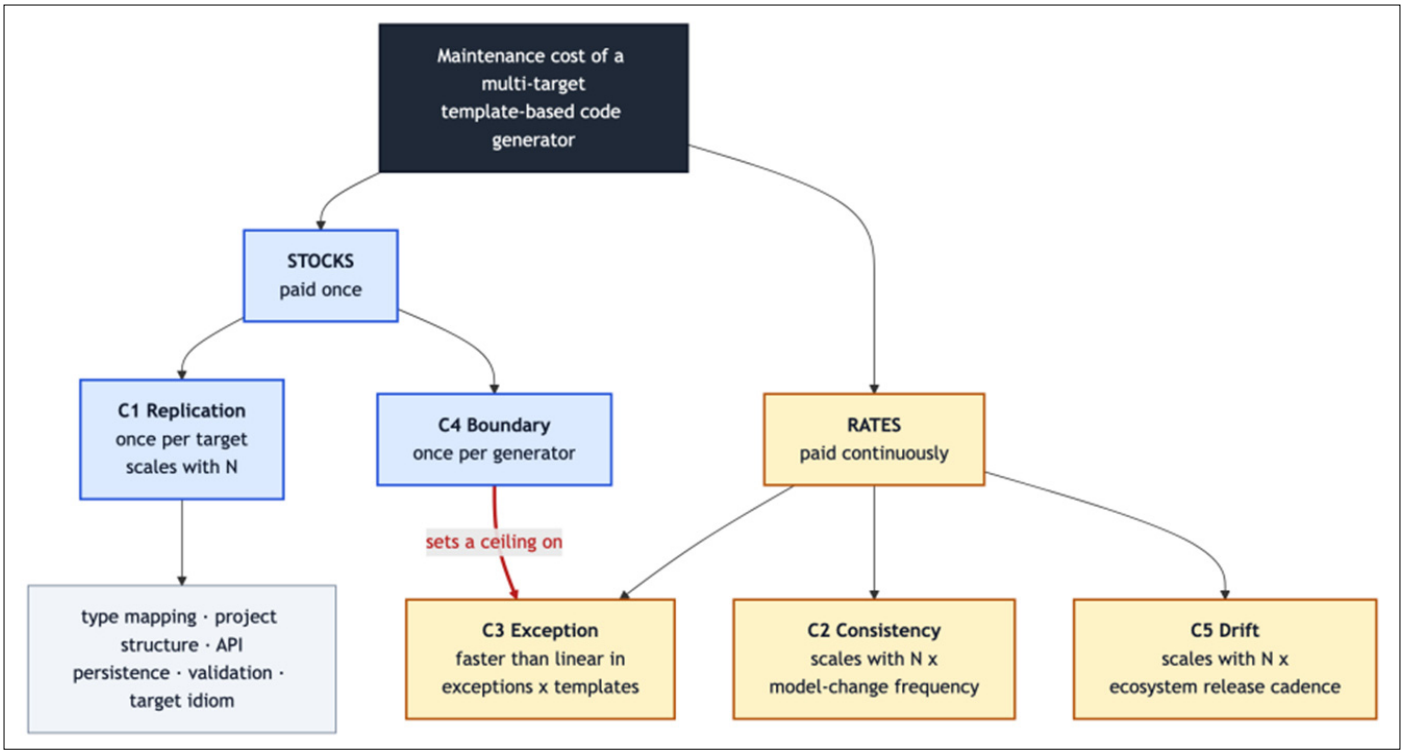

Figure 1. The five cost classes, grouped by whether each is a stock or a rate. C4 does not add to the total; it sets a ceiling on C3.

Replication Cost, and why it is the Class Teams Estimate

Adding a target technology means re-expressing the model in the idiom of that technology. The work partitions into five recurring components: mapping the model type system onto the target type system, generating project structure and build configuration, generating the interface or service layer, generating persistence, and generating validation. A sixth component, accommodating the particular conventions of the technology, resists estimation because it is discovered rather than specified.

In the illustrative repository a type system of 58 types must be expressed for each of the 3 supported persistence providers. Domain-specific language implementations show the same shape: Negm et al. (2019) survey implementation aspects across language workbenches and describe a staged development process in which target-specific concerns recur at each stage, while Nordmann et al. (2016), surveying domain-specific modelling in robotics, find the same per-platform work repeated across tools that share little.

Composition is the standard answer to replication. Ringert et al. (2015) compose languages and code generators so that a generator for a new configuration is assembled from existing parts rather than written afresh, and Hemel et al. (2010) obtain a similar effect by decomposing generation into modular model transformations. Both reduce the constant factor. Neither changes the fact that the cost is incurred once per target and therefore grows with the number of targets.

Replication cost is a stock. It is bounded, estimable in advance, and visible at the moment someone proposes a new target. This is precisely why it dominates adoption decisions, and why the classes that follow are systematically excluded from them.

Consistency Cost, and the Oracle Problem

Once a model renders to several targets, the targets must agree. Agreement is not free, and the cost recurs with every model change rather than once per target.

The illustrative repository holds agreement through 49 golden reference files plus 1,194 test methods. That apparatus scales with targets: each target needs its own reference outputs, and each model change potentially invalidates all of them.

The deeper difficulty is that comparing generated output to a reference is harder than it appears, because a generator may legitimately emit different but equivalent results. In a network design context, comparing structural properties and aggregate quantities within a tolerance works where comparing exact serialised output does not, because generation can produce distinct yet equivalent configurations.

Chen et al. (2022) report that a code generation model achieves a pass@100 of 77.4% on a standard benchmark against a pass@1 of 33.5%: a correct solution is usually present among the candidates, while identifying which one is correct is the unsolved part. Their method raises pass@1 to 65.8%, an improvement of 18.8 percentage points over the strongest baseline, and gains 9.5 percentage points using only 10 test cases. The setting differs, since a deterministic generator does not sample candidates, but the asymmetry it exposes is the same one a multi-target generator meets: producing plausible output is the cheaper half, and establishing that the output is right, in every target, after every model change, is the recurring half.

Tooling for detecting divergence between modelling artefacts exists but is not yet a cost control. Babur et al. (2019) detect clones among metamodels, and Varró et al. (2016) describe incremental and reactive transformation execution that recomputes only what a change affects. Incrementality attacks exactly the right quantity, the cost per model change, and its absence from most generator stacks is one reason C2 behaves as a full rate rather than a discounted one.

That correctness is worth paying for is not in doubt. Erwig et al. (2006), motivating a generator for spreadsheets, cite reports that 90% or more of real-world spreadsheets contain errors, and build a generator precisely so that the artefact is correct by construction.

Exception Cost, and the Ceiling of the Template Approach

Templates express the regular case. Business logic that does not fit the model, relationships the model cannot state, and requirements peculiar to one customer arrive as exceptions, and each is absorbed into a template as a conditional. The practical ceiling arrives when templates become overloaded with conditions and difficult to maintain.

This class grows faster than linearly for a structural reason. An exception is not confined to one template. It is applied wherever the affected element is rendered, which in a multi-target generator means once per target that renders it, and the interaction between exceptions inside a single template compounds as their number rises.

Wimmer et al. (2012) assemble a catalogue of refactorings for model-to-model transformations, motivated by transformation code that is correct but structurally poor, a pattern they encounter in labs where about 150 students solve transformation problems. That a catalogue of refactorings is needed at all indicates that transformation and template artefacts degrade in the way ordinary code does, while receiving less of the tooling that ordinary code receives.

Whittle et al. (2017), re-analysing 19 in-depth interviews and adding 20 more, with interviewees representing over 360 years of development experience and an inter-coder agreement of 0.86, found tool issues raised on average 11 times per transcript, ranging from 3 to 18. They also record that 30% of respondents in a related survey regard model-driven engineering tools as a barrier to adoption.

Boundary Cost, the Class that Bounds Another

Boundary cost does not add to the total in the way the others do. It is a one-off investment that determines whether exception cost is bounded or open-ended.

If there is no sanctioned place for hand-written code, every exception must enter a template, and C3 accumulates without limit. If the generator defines extension points, that is, a seam at which hand-written code attaches and which regeneration does not overwrite, exceptions leave the templates and land in ordinary code, where ordinary tooling

applies. The standard artefact is generated, and specific logic is added separately so that regeneration does not destroy it.

Greifenberg et al. (2015) treat this seam as a design decision with alternatives. They compare eight mechanisms for integrating hand-written and generated code in object-oriented languages, including the generation gap and its extended form, delegation, include mechanisms, partial classes and aspect-oriented approaches, and evaluate them against explicit criteria: separation of generated from hand-written code, support for overriding generated parts, extendability of generated interfaces, independence of hand-written code at generation time, and independence from additional language constructs. Their conclusion is that no integration mechanism is always the right one, and that the choice follows from context and requirements.

That conclusion is what makes this a cost class rather than a best practice. The boundary must be chosen, implemented, documented and enforced per generator, and in a multi-target generator the chosen mechanism may not exist in every target language, since partial classes and aspects are not universally available. The investment is real. What it buys is a ceiling on the class that otherwise has none.

Drift Cost, and Comparison Across Generator Families

Target technologies evolve independently of the model. A framework major version, a language release or a deprecated library invalidates template assumptions, and the work recurs at the pace of each ecosystem rather than at the pace of the project.

Zhang et al. (2023), evaluating automated extraction of grammar optimisation rules across 6 real language cases, extract 369 rules automatically: the regenerated grammar is identical to the target for three of the six languages, while the authors report agreement between 88% and 67% for the remaining three. The residue is the point: co-evolution is partly automatable and not fully, and the unautomated remainder recurs at every evolution step. Iovino et al. (2012) argue that the impact of metamodel evolution extends beyond models to every dependent artefact, and Cicchetti et al. (2012) address the versioning problem that follows when models and metamodels change concurrently.

Table 2 applies the five classes across generator families.

Table 2. Cost profile by generator family. The verdicts are judgements made in this review; figures are drawn from the cited sources

Family	C1 Replication	C2 Consistency	C3 Exception	C4 Boundary	C5 Drift	Evidence anchor
Template-based, multi-target	High, linear in N; five components per target	High; oracle suites multiply with N	High unless bounded	Decisive; eight mechanisms available, none universal	High; N ecosystems evolve independently	Nox: 60.8% generated, 49 golden files, 58 types × 3 providers; Greifenberg et al. (2015)
Model-to-model transformation chains	Moderate; intermediate models absorb some target specificity	High; each stage needs its own oracle, partly offset by incrementality	High; transformation code degrades like ordinary code	Weak by default; seams often implicit	Moderate	Wimmer et al. (2012); Varró et al. (2016)
Low-code platforms	Low for the buyer; borne by the vendor	Opaque; assurance is the vendor's	Sharp ceiling at the platform's expressiveness limit	Vendor-defined, rarely negotiable	Externalised to the vendor's release cycle	Elshan et al. (2023): 63 practitioners approached, 44% participating
Assisted generation with language models	Low; no per-target template set required	Severe; output is not reproducible, so reference comparison does not apply	Low; irregular cases handled without template edits	Undefined; no regeneration contract	Tracks model and ecosystem churn	Chen et al. (2022): pass@100 77.4% against pass@1 33.5%
Hybrid: assisted authoring, deterministic generation	Unchanged from template-based	Preserved, because final generation stays reproducible	Reduced at the authoring stage	As defined by the deterministic generator	Unchanged	Section 3.11 of this article

The low-code row deserves a qualification, because the family is often presented as the successor to model-driven engineering rather than as a repackaging of it. Di Ruscio et al. (2022) argue that low-code development and model-driven engineering are two sides of one coin, and Bucaioni et al. (2022) and Rokis and Kirikova (2023) find in systematic reviews that the modelling concerns persist under new names. Sanchis et al. (2019) report the industrial appeal

in manufacturing. What changes under low-code is not the existence of the five classes but who pays them: the platform vendor absorbs C1 and C5, and the customer meets C3 as a hard expressiveness ceiling instead of as gradually thickening templates. Käss et al. (2023) and Elshan et al. (2023) document practitioner perceptions consistent with that shift, and Henderson et al. (2024), reporting adoption experience for model-based systems engineering, describe the same organisational pattern in a neighbouring discipline.

Why the Classes are Confused in Practice

The classes divide cleanly along one line. C1 and C4 are stocks. C2, C3 and C5 are rates.

Only stocks are visible when a team decides whether to add a target technology. The replication work can be scoped, and it is the figure that enters the estimate. The rates are invisible at that moment by construction: they are incurred later, at a pace set by model churn, by the arrival of exceptions and by ecosystem releases, none of which is known at decision time.

This offers one candidate account of the dispersion reported by Whittle et al. (2014); differences in domain, team maturity and measurement practice plainly contribute as well. An organisation whose models stabilise early, whose exceptions are diverted across a boundary, and whose targets are few and slow-moving pays mostly C1 and reports a large gain. An organisation with churning models, no boundary and several fast-moving targets pays three rates indefinitely and can report a net loss. Both are using the same technology and reporting the same metric. The 20–30% band containing most respondents is then the ordinary case in which the rates are present but contained.

The taxonomy also reframes the tool complaints in Whittle et al. (2017). Tools raised 11 times per transcript, and named a barrier by 30% of respondents in a related survey, are the visible surface of rate-shaped costs that the adopting organisation did not price.

Ameller et al. (2021), interviewing practitioners from 18 companies using model-driven development, 5 medium-sized and 13 large, through an instrument of 10 questions and 43 sub-questions, document a related blind spot: requirements that the modelling approach does not express are handled outside it. Concerns a model cannot state do not disappear; they become exceptions, which is C3.

Applying the Taxonomy to an Adoption Decision

The classification earns its place if it changes what a team asks before committing to an additional target technology. Each class implies a question whose answer can be gathered in advance rather than discovered afterwards, as Table 3 sets out.

Table 3. The five questions an adoption decision should answer before a target technology is added. Each answer can be established before the commitment is made

Class	Question to answer before committing	Evidence available in advance
C1 Replication	How much of the existing type system and model schema survives the move?	Count of model types needing a new mapping × persistence and interface variants required
C2 Consistency	What is the oracle for the new target, and who maintains it?	Current ratio of reference-suite maintenance to generator maintenance
C3 Exception	How many exceptions already in the templates are target-specific rather than model-level?	Audit of existing template conditionals, classified by scope
C4 Boundary	Does the chosen integration mechanism exist in the new target language?	Language feature check against the mechanism in use, e.g. partial classes or aspects
C5 Drift	What is the release cadence of the target ecosystem, and will the project track it?	Upstream release history and the project's own version-pinning policy

The consistency question is often answered by assuming the existing reference suite extends to the new target, when golden files are per-target artefacts and do not. The boundary question can fail outright: since no integration mechanism is universal (Greifenberg et al., 2015), a target whose language lacks partial classes or aspects may force a second seam, and a generator maintaining two seams pays the documentation and enforcement cost twice.

Three of the five questions concern rates, and those are the ones an onboarding estimate leaves out. The evidence needed splits too. The replication, consistency and exception questions are answerable from the generator's own history. The boundary and drift questions need two facts about the incoming target instead: which integration mechanisms its language supports, and how fast its ecosystem releases. Both are cheap to establish before committing, which is what makes the taxonomy usable prospectively rather than only as a post-mortem.

The Deterministic and Probabilistic Division of Labour

Generation by language model is now a large field. Jiang et al. (2024) survey it from 294 arXiv, 73 ACM, 36 IEEE and 261 bibliographic records, and report annual growth from a single paper across 2018–2020 to 6 in 2021, 11 in 2022, 75 in 2023 and 140 in 2024, over models that have grown to 175 billion and 540 billion parameters trained on corpora such as an 825 GiB text collection and a snapshot of more than 2.8 million open-source repositories.

This technology has a sharply asymmetric cost profile. It relieves C3, since an irregular requirement is handled without editing a template, and it lowers C1, since no per-target template set is required. It damages C2 severely, because the output is not reproducible, and reference comparison presumes that the same input yields the same output.

Liu et al. (2024) characterise recurring quality defects in generated code and propose refinement to remove them, Du et al. (2024) find performance degrading as the unit of generation grows from a function to a class, Xu et al. (2022) report that developers accept such assistance selectively when it is embedded in the editor, and Joel et al. (2025) show the difficulty rising for languages and domains with little training data, which is the situation of most proprietary target technologies. Russo (2024), surveying 100 software engineers, frames the question as one of adoption conditions rather than raw capability.

That asymmetry, not a forecast about the field, is what recommends a division of labour. Language models are well suited to preparing the model, drafting templates and writing the custom code that sits beyond the boundary, all activities where a human reviews the result before it is committed. Final generation of infrastructure that a business depends on is better left to a deterministic generator whose output can be compiled, tested against references and analysed statically. The hybrid arrangement in the last row of Table 2 keeps the C3 relief where it is cheap and keeps C2 checkable where it matters.

Open Challenges

1. No cost model separates stocks from rates. Published estimates of model-driven engineering effort report a single quantity, so an organisation cannot distinguish a one-off from a commitment. *First step:* report generator effort as two figures, per-target onboarding effort and effort per model change, in any future industrial study.

2. Boundary mechanisms are described as patterns, not evaluated as cost controls. Greifenberg et al. (2015) compare eight mechanisms against design criteria, but the literature does not measure how much exception cost each mechanism actually diverts. *First step:* instrument a multi-target generator to count exceptions absorbed into templates against those handled beyond the boundary, before and after a mechanism is introduced.

3. Oracle cost is unmeasured. Reference suites and golden files scale with targets, yet no study reports the ratio of effort spent maintaining the oracle to effort spent maintaining the generator. *First step:* publish that ratio for a single generator over one release cycle.

4. Target drift is treated as a port, not as a rate. Framework and language evolution appears in the literature as a migration event rather than as a recurring charge proportional to the number of targets. *First step:* record, per target, the number of template edits attributable to upstream releases over a fixed period.

5. The reproducibility boundary for assisted generation is undefined. If a language model participates in generation, the conditions under which its output may enter a regenerated artefact are not specified anywhere. *First step:* state a regeneration contract, namely which artefacts are reproducible from the model alone and which are not, as an explicit property of the generator.

Limitations

The classification is grounded in one practitioner's experience of two generator platforms, supported by 36 sources of which 15 were examined in full. The cost classes are argued rather than measured: no study cited here reports effort decomposed along these five axes, which is challenge 1 above. The illustrative repository is a single project in one language ecosystem, and its measurements characterise that project rather than the class of generators. It is also not independent of the argument: the author contributed to that repository, and the same professional experience informs the taxonomy, so the illustration corroborates the classes rather than testing them. Placement of abstract-level studies within the taxonomy carries the uncertainty noted in Section 2.

CONCLUSION

The maintenance cost of a template-based generator serving several target technologies is not one quantity. It separates into replication, consistency, exception, boundary and drift costs, which differ in what they scale with and in whether they are paid once or continuously. Replication is a stock and is visible when a target is proposed; consistency, exception and drift are rates and are not. Boundary cost is a stock whose function is to bound the exception rate rather than to add to the total, which is why the decision to define a seam between generated and hand-written code determines whether total cost grows in proportion to the number of targets or faster. Reported productivity effects spanning a 27% loss to an 800% gain are consistent with organisations paying different mixtures of these five classes while reporting one number.

REFERENCES

1. Ameller, D., Franch, X., Gomez, C., Martinez-Fernandez, S., Araujo, J., Biffl, S., Cabot, J., Cortellessa, V., Fernandez, D. M., Moreira, A., Muccini, H., Vallecillo, A., Wimmer, M., Amaral, V., Bohm, W., Bruneliere, H., Burgueno, L., Goulao, M., Teufl, S., & Berardinelli, L. (2021). Dealing with Non-Functional Requirements in Model-Driven Development: A Survey. IEEE Transactions on Software Engineering, 47(4), 818-835. https://doi.org/10.1109/tse.2019.2904476
2. Babur, Ö., Cleophas, L., & van den Brand, M. (2019). Metamodel clone detection with SAMOS. Journal of Computer Languages, 51, 57-74. https://doi.org/10.1016/j.cola.2018.12.002

3. Bucaioni, A., Cicchetti, A., & Ciccozzi, F. (2022). Modelling in low-code development: a multi-vocal systematic review. Software and Systems Modeling, 21(5), 1959-1981. https://doi.org/10.1007/s10270-021-00964-0

4. Bucchiarone, A., Cabot, J., Paige, R. F., & Pierantonio, A. (2020). Grand challenges in model-driven engineering: an analysis of the state of the research. Software and Systems Modeling, 19(1), 5-13. https://doi.org/10.1007/s10270-019-00773-6

5. Chen, B., Zhang, F., Nguyen, A., Zan, D., Lin, Z., Lou, J., & Chen, W. (2022). CodeT: Code Generation with Generated Tests. arXiv preprint. https://doi.org/10.48550/arxiv.2207.10397

6. Cicchetti, A., Ciccozzi, F., & Leveque, T. (2012). A Solution for Concurrent Versioning of Metamodels and Models. The Journal of Object Technology, 11(3), 1:1. https://doi.org/10.5381/jot.2012.11.3.a1

7. Di Ruscio, D., Kolovos, D., de Lara, J., Pierantonio, A., Tisi, M., & Wimmer, M. (2022). Low-code development and model-driven engineering: Two sides of the same coin?. Software and Systems Modeling, 21(2), 437-446. https://doi.org/10.1007/s10270-021-00970-2

8. Du, X., Liu, M., Wang, K., Wang, H., Liu, J., Chen, Y., Feng, J., Sha, C., Peng, X., & Lou, Y. (2024). Evaluating Large Language Models in Class-Level Code Generation. Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE), 1-13. https://doi.org/10.1145/3597503.3639219

9. Elshan, E., Dickhaut, E., & Ebel, P. A. (2023). An Investigation of Why Low Code Platforms Provide Answers and New Challenges. Proceedings of the 56th Hawaii International Conference on System Sciences (HICSS). https://doi.org/10.24251/hicss.2023.746

10. Erwig, M., Abraham, R., Kollmansberger, S., & Cooperstein, I. (2006). Gencel: a program generator for correct spreadsheets. Journal of Functional Programming, 16(3), 293-325. https://doi.org/10.1017/s0956796805005794

11. Greifenberg, T., Hölldobler, K., Kolassa, C., Look, M., Nazari, P. M. S., Müller, K., Pérez, A., Plotnikov, D., Reiß, D., Roth, A., Rumpe⋆, B., Schindler, M., & Wortmann, A. (2015). A Comparison of Mechanisms for Integrating Handwritten and Generated Code for Object-Oriented Programming Languages. Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD), 74–85. https://doi.org/10.48550/arxiv.1509.04498

12. Hemel, Z., Kats, L. C. L., Groenewegen, D. M., & Visser, E. (2010). Code generation by model transformation: a case study in transformation modularity. Software & Systems Modeling, 9(3), 375-402. https://doi.org/10.1007/s10270-009-0136-1

13. Henderson, K., McDermott, T., & Salado, A. (2024). MBSE adoption experiences in organizations: Lessons learned. Systems Engineering, 27(1), 214-239. https://doi.org/10.1002/sys.21717

14. Hutchinson, J., Whittle, J., Rouncefield, M., & Kristoffersen, S. (2011). Empirical assessment of MDE in industry. Proceedings of the 33rd International Conference on Software Engineering (ICSE), 471-480. https://doi.org/10.1145/1985793.1985858

15. Hutchinson, J., Whittle, J., & Rouncefield, M. (2014). Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure. Science of Computer Programming, 89, 144-161. https://doi.org/10.1016/j.scico.2013.03.017

16. Iovino, L., Pierantonio, A., & Malavolta, I. (2012). On the Impact Significance of Metamodel Evolution in MDE. The Journal of Object Technology, 11(3), 3:1. https://doi.org/10.5381/jot.2012.11.3.a3

17. Jiang, J., Wang, F., Shen, J., Kim, S., & Kim, S. (2024). A Survey on Large Language Models for Code Generation. arXiv preprint arXiv:2406.00515. https://doi.org/10.48550/arxiv.2406.00515

18. Joel, S., Wu, J., & Fard, F. (2025). A Survey on LLM-based Code Generation for Low-Resource and Domain-Specific Programming Languages. ACM Transactions on Software Engineering and Methodology. https://doi.org/10.1145/3770084

19. Käss, S., Strahringer, S., & Westner, M. (2023). Practitioners' Perceptions on the Adoption of Low Code Development Platforms. IEEE Access, 11, 29009-29034. https://doi.org/10.1109/access.2023.3258539

20. Liu, Y., Le-Cong, T., Widyasari, R., Tantithamthavorn, C., Li, L., Le, X. B. D., & Lo, D. (2024). Refining ChatGPT-Generated Code: Characterizing and Mitigating Code Quality Issues. ACM Transactions on Software Engineering and Methodology, 33(5), 1-26. https://doi.org/10.1145/3643674

21. Mohagheghi, P., & Dehlen, V. (2008). Where Is the Proof? - A Review of Experiences from Applying MDE in Industry. Model Driven Architecture – Foundations and Applications, Lecture Notes in Computer Science, 432-443. https://doi.org/10.1007/978-3-540-69100-6_31

22. Mohagheghi, P., Gilani, W., Stefanescu, A., & Fernandez, M. A. (2013). An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases. Empirical Software Engineering, 18(1), 89-116. https://doi.org/10.1007/s10664-012-9196-x

23. Negm, E., Makady, S., & Salah, A. (2019). Survey on Domain Specific Languages Implementation Aspects. International Journal of Advanced Computer Science and Applications, 10(11). https://doi.org/10.14569/ijacsa.2019.0101183

24. Nordmann, A., Hochgeschwender, N., Wigand, D. L., & Wrede, S. (2016). A Survey on Domain-specific Modeling and Languages in Robotics. Journal of Software Engineering for Robotics. https://doi.org/10.6092/joser_2016_07_01_p75

25. Ringert, J. O., Roth, A., Rumpe⋆, B., & Wortmann, A. (2015). Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. Journal of Software Engineering for Robotics. https://doi.org/10.6092/joser_2015_06_01_p33

26. Rodrigues da Silva, A. (2015). Model-driven engineering: A survey supported by the unified conceptual model. Computer Languages, Systems & Structures, 43, 139-155. https://doi.org/10.1016/j.cl.2015.06.001

27. Rokis, K., & Kirikova, M. (2023). Exploring Low-Code Development: A Comprehensive Literature Review. Complex Systems Informatics and Modeling Quarterly, 68-86. https://doi.org/10.7250/csimq.2023-36.04

28. Russo, D. (2024). Navigating the Complexity of Generative AI Adoption in Software Engineering. ACM Transactions on Software Engineering and Methodology, 33(5), 1-50. https://doi.org/10.1145/3652154

29. Sanchis, R., García-Perales, Ó., Fraile, F., & Poler, R. (2019). Low-Code as Enabler of Digital Transformation in Manufacturing Industry. Applied Sciences, 10(1), 12. https://doi.org/10.3390/app10010012

30. Varró, D., Bergmann, G., Hegedüs, Á., Horváth, Á., Ráth, I., & Ujhelyi, Z. (2016). Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework. Software & Systems Modeling, 15(3), 609-629. https://doi.org/10.1007/s10270-016-0530-4

31. Whittle, J., Hutchinson, J., Rouncefield, M., Burden, H., & Heldal, R. (2013). Industrial Adoption of Model-Driven Engineering: Are the Tools Really the Problem?. Model-Driven Engineering Languages and Systems (MODELS), Lecture Notes in Computer Science, 1-17. https://doi.org/10.1007/978-3-642-41533-3_1

32. Whittle, J., Hutchinson, J., & Rouncefield, M. (2014). The State of Practice in Model-Driven Engineering. IEEE Software, 31(3), 79-85. https://doi.org/10.1109/ms.2013.65

33. Whittle, J., Hutchinson, J., Rouncefield, M., Burden, H., & Heldal, R. (2017). A taxonomy of tool-related issues affecting the adoption of model-driven engineering. Software & Systems Modeling, 16(2), 313-331. https://doi.org/10.1007/s10270-015-0487-8

34. Wimmer, M., Martínez, S., Jouault, F., & Cabot, J. (2012). A Catalogue of Refactorings for Model-to-Model Transformations. The Journal of Object Technology, 11(2), 2:1. https://doi.org/10.5381/jot.2012.11.2.a2

35. Xu, F. F., Vasilescu, B., & Neubig, G. (2022). In-IDE Code Generation from Natural Language: Promise and Challenges. ACM Transactions on Software Engineering and Methodology, 31(2), 1-47. https://doi.org/10.1145/3487569

36. Zhang, W., Hebig, R., Strüber, D., & Steghöfer, J. P. (2023). Automated Extraction of Grammar Optimization Rule Configurations for Metamodel-Grammar Co-evolution. Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering (SLE), 84-96. https://doi.org/10.1145/3623476.3623525