Indicadores de Manutenibilidade no Acesso a Dados Analíticos: Um Estudo Exploratório Comparando SDK Nativo e Query Builders Tipados

João Vitor Coimbra
Centro Federal de Educação Tecnológica Celso Suckow da Fonseca (CEFET/RJ)
Rio de Janeiro, Brasil
joao.coimbra@aluno.cefet-rj.br

Diego Castro
Centro Federal de Educação Tecnológica Celso Suckow da Fonseca (CEFET/RJ)
Rio de Janeiro, Brasil
diego.castro@cefet-rj.br

Eduardo Ferreira
Centro Federal de Educação Tecnológica Celso Suckow da Fonseca (CEFET/RJ)
Rio de Janeiro, Brasil
eduardo.ferreira@cefet-rj.br

Resumo

Os acessos de aplicações a data *warehouses* analíticos, como o Google BigQuery, costumam ser feitos por SDKs (Software Development Kits) nativos que exigem compor SQL manualmente dentro do código, misturando consulta e lógica de programa, o que reduz legibilidade e manutenibilidade. Este artigo apresenta o TeraORM, um ORM (*Object-Relational Mapping*) de consultas tipado e encadeável para *warehouses* analíticos, e avalia diferenças em indicadores estáticos associados à manutenibilidade e na percepção de modificabilidade por desenvolvedores. A investigação combina uma análise estática comparativa de oito serviços reais reescritos do SDK nativo para o ORM com um estudo intrassujeitos com medidas repetidas, no qual oito participantes implementaram tarefas equivalentes nas duas abordagens e responderam a um instrumento de percepção baseado no Modelo de Aceitação de Tecnologia. A análise estática mostra redução de linhas de código em todos os oito serviços (33,8% no agregado) e melhora do Índice de Manutenibilidade em sete deles, a favor do ORM.

Palavras-chave

Manutenibilidade de Software, ORM, Acesso a Dados, Data Warehouse, Estudo Exploratório, Índice de Manutenibilidade, Qualidade de Código

1 Introdução

Sistemas de informação modernos dependem cada vez mais de *data warehouses* analíticos para sustentar relatórios, indicadores e processos de tomada de decisão orientados a dados [7, 19]. Plataformas como o Google BigQuery oferecem grande capacidade de processamento, porém o acesso programático a esses ambientes é comumente realizado por meio de SDKs nativos, nos quais o desenvolvedor compõe manualmente instruções SQL como cadeias de caracteres embutidas diretamente no código da aplicação [2]. Essa estratégia, embora flexível, entrelaça a lógica de consulta com a lógica de programa e reintroduz, no domínio analítico, sintomas historicamente associados ao descasamento entre objetos e o modelo relacional [7, 19].

A composição manual de SQL literal apresenta custos de manutenção reconhecidos. Consultas escritas como texto não são verificadas pelo compilador, de modo que erros de nome de coluna, de operador ou de parametrização só se manifestam em tempo de execução [21]. A intenção da consulta também fica diluída em fragmentos de *string* concatenados condicionalmente, o que dificulta a leitura, a revisão e a modificação segura do código ao longo do tempo. À medida que o número de relatórios e filtros cresce, esses custos se acumulam e afetam diretamente a evolução do software [2, 21].

Os ORMs (*Object-Relational Mappers*) e *query builders* tipados surgem como uma alternativa consolidada no contexto de bancos transacionais, com APIs (*Application Programming Interface*) que expressam a intenção da consulta de forma declarativa e verificável [7, 37]. Entretanto, o ecossistema de warehouses analíticos ainda carece de ferramentas equivalentes que combinem tipagem, portabilidade entre adaptadores e governança de custo [9, 21]. Diante disso, apresenta-se o TeraORM, um ORM de consultas tipado e encadeável em TypeScript, com um adaptador para *data warehouses*, com foco específico no BigQuery.

Embora ORMs e *query builders* tipados sejam amplamente utilizados em bancos transacionais, ainda é limitada a evidência sobre como essas abstrações afetam indicadores de manutenibilidade e a modificabilidade percebida por desenvolvedores em aplicações que consultam *data warehouses* analíticos. Para investigar essa lacuna, o estudo adota uma estratégia em duas frentes complementares. A primeira é uma análise estática comparativa de antes e depois, na qual oito serviços reais são reescritos do SDK nativo para o TeraORM com cobertura funcional idêntica, medindo-se indicadores de tamanho e de manutenibilidade. A segunda é um estudo intrassujeitos com medidas repetidas, no qual cada participante implementa tarefas equivalentes nas duas abordagens e reporta sua percepção por meio de um instrumento fundamentado no Modelo de Aceitação de Tecnologia [11].

As principais contribuições deste trabalho são: (i) o TeraORM, um ORM tipado para acesso analítico, com abstrações específicas para o BigQuery; (ii) uma comparação pareada de oito serviços reais antes e depois da migração, avaliando indicadores estáticos associados à manutenibilidade; (iii) um estudo exploratório intrassujeitos, com oito participantes, sobre clareza, modificabilidade e preferência percebidas; e (iv) uma discussão que separa a contribuição do artefato da evidência empírica e delimita o escopo das afirmações de manutenibilidade.

2 ORM Analítico

2.1 Acesso a Dados Analíticos e a Necessidade de um ORM Específico

O acesso a *data warehouses* por SDKs nativos segue tipicamente o padrão de montar a consulta SQL como texto e submetê-la a um cliente remoto [2, 21]. O problema central, do ponto de vista de manutenção, é que o SQL literal não participa do sistema de tipos da linguagem hospedeira: nomes de colunas, operadores e formatos de parâmetro tornam-se cadeias não validadas pelo compilador, de modo que defeitos só emergem em tempo de execução e refatorações que renomeiam campos exigem varredura manual. Esse fenômeno é uma manifestação, no domínio analítico, do descasamento entre objetos e o modelo relacional [7, 19].

A mera transposição de um ORM transacional é insuficiente, porque os dois contextos têm premissas distintas. Motores analíticos como o BigQuery seguem execução massivamente paralela, varrendo grandes volumes colunares em vez de recuperar poucas linhas por chave [28]. Disso decorrem particularidades que um ORM analítico precisa tratar: o custo é proporcional ao volume escaneado, exigindo estimativa e limitação antes da execução (ausentes em ORMs transacionais) [2]; as consultas são maiores, com agregações e filtros compostos; a governança (políticas de custo, sanitização, limites de bytes) precisa ser programática e auditável; e as coordenadas de tabela (projeto, *dataset*, tabela) exigem resolução explícita de referências. Isso justifica uma ferramenta específica, e não a reutilização direta de ORMs existentes [21].

2.2 TeraORM

O ORM proposto é um *toolkit* de consultas tipado para *warehouses* analíticos, distribuído como *software* de código aberto e publicado como um conjunto de pacotes independentes: um núcleo, um pacote de conveniência que reexporta o núcleo, um adaptador para BigQuery e um módulo para integração com o framework JavaScript, com foco no NestJS [35]. Sua proposta central é substituir a composição de SQL literal por uma API fluente e encadeável, na qual o desenvolvedor descreve o modelo de dados e compõe a consulta por meio de métodos como `select`, `where`, `orderBy`, `groupBy`, `having` e agregações (`count`, `avg`, `sum`).

A arquitetura do ORM organiza-se em camadas, ilustradas na Figura 1. Na camada de aplicação, o desenvolvedor interage com um repositório tipado, obtido por injeção de dependência e associado a modelos que descrevem as colunas e a origem dos dados; no contexto NestJS, isso é mediado por *decorators* e provedores. O núcleo concentra a definição de modelos, o *query builder* encadeável, o mecanismo de políticas e um sanitizador de valores. A camada de adaptador é responsável por traduzir o plano de consulta abstrato em SQL do BigQuery por meio de um compilador, validar as colunas contra o esquema real com um validador de metadados e resolver as coordenadas de tabela por meio de um resolvedor de relações. Essa separação entre núcleo e adaptador é o que confere portabilidade à ferramenta: novos warehouses podem ser suportados pela implementação de novos adaptadores, sem alteração do código de aplicação.

Além da composição de consultas, o ORM proposto trata governança como preocupação de primeira ordem, respondendo

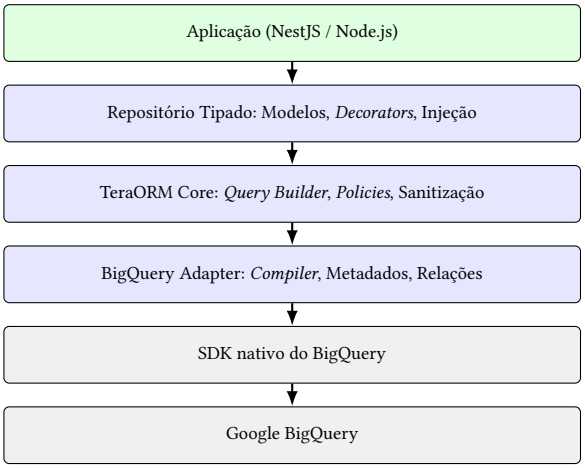

Figura 1: Arquitetura em camadas do TeraORM. O código de aplicação depende apenas do repositório tipado; a tradução para SQL, a validação de metadados e a resolução de coordenadas ficam encapsuladas no adaptador.

Código 1: Serviço de funil por estágio com SDK nativo (SQL literal).

```
async byStage(filters: LegacyFilterInput) {
  let sql = `
    SELECT ev.stage, COUNT(1) AS total
    ${baseJoin(this.projectDataset)}
    WHERE 1 = 1
  `;
  sql = appendCommonFilters(sql, filters);
  sql += " GROUP BY ev.stage ORDER BY total DESC";
  return this.client.query({ query: sql });
}
```

às particularidades discutidas na Seção 2.1. Políticas podem ser definidas por consulta ou por repositório. Elas permitem estabelecer limites de bytes processados e de custo em dólares, exigir estimativa de custo pelo adaptador antes da execução e ativar a sanitização de literais suspeitos. O adaptador de BigQuery também suporta validação prévia de metadados: compara as colunas referenciadas com o esquema real do warehouse antes de submeter a consulta. Esses mecanismos, ausentes em ORMs transacionais convencionais, distinguem a proposta no contexto analítico.

O contraste entre as duas abordagens é ilustrado pela reescrita de um serviço de funil por estágio. Na versão baseada no SDK nativo (Código 1), a consulta é composta como texto e complementada por fragmentos condicionais.

Na versão equivalente com o TeraORM (Código 2), a mesma intenção é expressa de forma declarativa e tipada, sem manipulação textual de SQL.

Além da redução evidente de código, a versão com ORM elimina a concatenação condicional de *strings* e expõe a estrutura da consulta a verificações estáticas, o que constitui a hipótese central de manutenibilidade avaliada neste estudo.

Código 2: Serviço equivalente com o TeraORM (API fluente e tipada).

```
async byStage() {
  return this.repo
    .select("stage")
    .count("stage", "total")
    .groupBy("stage")
    .orderByAlias("total", "desc")
    .execute();
}
```

2.3 Governança e Segurança: Demonstração Funcional

Os mecanismos de governança e de segurança descritos acima são funcionalidades efetivas do artefato, e não apenas motivação. Ainda que não constituam o objeto da avaliação empírica de manutenibilidade (Seção 4), sua operação é determinística e pode ser demonstrada diretamente a partir da implementação. A avaliação empírica isola deliberadamente a API fluente e tipada; assim, os ganhos de tamanho e de modificabilidade relatados na Seção 5 não são atribuídos a estas políticas.

Composição segura de consultas. Os valores de filtro fornecidos em tempo de execução nunca são concatenados ao texto SQL: o compilador os emite como parâmetros nomeados do BigQuery. Uma condição `where("region", "=", entrada)` é traduzida para a forma parametrizada do Código 3, na qual a entrada textual permanece um valor ligado, e não código executável.

Código 3: Defesa contra injeção de SQL por parametrização: a entrada do usuário é ligada como parâmetro nomeado, nunca concatenada ao comando.

```
// Entrada do usuario
region = "' OR 1=1 --"

// SQL compilado (valor nunca concatenado)
WHERE `region` = @p1

// Parametros ligados enviados ao BigQuery
{ p1: "' OR 1=1 --" }
```

A essa parametrização somam-se camadas complementares, verificáveis no código-fonte do artefato. Um sanitizador rejeita, por padrão, literais que contenham *tokens* sensíveis a SQL (`;`, `--`, `/* */`) ou caracteres de controle, de modo que a própria entrada do Código 3 é bloqueada antes da compilação. Identificadores de coluna e de tabela são validados contra um padrão estrito (apenas letras, dígitos e sublinhado, iniciando por letra ou sublinhado) e, no adaptador de BigQuery, contra o esquema real do *warehouse* por um validador de metadados. Os operadores de comparação pertencem a um conjunto fechado, e listas (`IN UNNEST(@p1)`) e intervalos (`BETWEEN @p1Start AND @p1End`) são igualmente parametrizados. A entrada textual livre, portanto, não alcança o comando compilado.

Governança de custo. Políticas definidas por consulta ou por repositório podem limitar os bytes processados (`maxBytesProcessed`) e o custo estimado. Antes de executar, o adaptador solicita ao BigQuery uma estimativa por *dry run*; se o volume estimado excede o limite, a consulta é bloqueada com um erro explícito (*"Query blocked by policy: estimated bytes… exceed maxBytesProcessed…"*), evitando varreduras dispendiosas em produção.

3 Trabalhos Relacionados

A avaliação de ferramentas de ORM tem sido tratada sob perspectivas distintas. Alguns autores comparam o desempenho de bancos de objetos e ferramentas de ORM [37]. Outros estudos concentram-se na validação da abstração e da eficiência, investigando os efeitos do descasamento entre objetos e o modelo relacional do banco [19]. O papel dos ORMs na segurança e na governança de dados pessoais, dimensão próxima à de manutenibilidade, também não pode ser ignorado [15]. Esses trabalhos se concentram, na maior parte, em bancos transacionais e em métricas de desempenho. O presente estudo segue outro caminho: a atenção vai para *warehouses* analíticos e para a manutenibilidade. Ele combina métricas estáticas com percepção de desenvolvedores, e avalia a influência de diferentes abordagens de acesso a dados sobre a manutenibilidade de sistemas.

No que diz respeito à mensuração de manutenibilidade, a análise apoia-se em um corpo consolidado de métricas. A complexidade ciclomática [26], as métricas de Halstead [16] e o Índice de Manutenibilidade [6, 31] fornecem uma base objetiva amplamente adotada. O modelo de manutenibilidade do SIG [17] e a norma ISO/IEC 25010 [18] enquadram conceitualmente essas medidas. O presente estudo aplica esse instrumental a um par de implementações funcionalmente equivalentes, o que permite isolar o efeito da abordagem de acesso a dados.

O ORM proposto insere-se em uma tradição consolidada de *query builders* e ORMs tipados. No ecossistema JavaScript/TypeScript, ferramentas como Knex [23], Kysely [24], Prisma [33] e Drizzle [12] oferecem composição de consultas com diferentes graus de tipagem e abstração. Em outras plataformas, jOOQ [10] e QueryDSL [34] na JVM e o LINQ na plataforma .NET [27] seguem princípios semelhantes de tornar a consulta um cidadão de primeira classe da linguagem hospedeira. Essas ferramentas, contudo, foram projetadas primordialmente para bancos transacionais e não tratam motores analíticos como o BigQuery como destino primário, nem consideram governança de custo uma preocupação de primeira ordem, lacuna discutida na Seção 2.1. A Tabela 1 posiciona o TeraORM frente a esse conjunto. As ferramentas foram selecionadas por representatividade no ecossistema JavaScript/TypeScript (Knex, Kysely, Prisma e Drizzle) e entre *query builders* tipados de outras plataformas (jOOQ), a partir de busca pela documentação oficial e pelos repositórios públicos dos projetos; a classificação em cada eixo apoia-se na documentação vigente de cada ferramenta. A combinação de foco analítico, tipagem forte, padrão de repositório e políticas de custo ainda é incomum entre essas ferramentas.

4 Avaliação do ORM

O objetivo do estudo, formulado segundo a abordagem GQM [1], é analisar implementações de acesso a dados analíticos com o propósito de comparar no que diz respeito à manutenibilidade do ponto de vista de desenvolvedores no contexto de consultas ao BigQuery realizadas com o SDK nativo e com o TeraORM. O desenho e o relato do estudo seguem recomendações metodológicas para pesquisa em Ciência da Computação [38] e diretrizes de experimentação em Engenharia de Software [22, 39]. Desse objetivo derivam três questões de pesquisa:

Tabela 1: Posicionamento do ORM frente a query builders e ORMs existentes.

Ferramenta	Foco analítico (BigQuery)	Tipagem forte	Repositório	Política de custo
Knex	Não	Parcial	Não	Não
Kysely	Não	Sim	Não	Não
Prisma	Não	Sim	Parcial	Não
Drizzle	Não	Sim	Não	Não
jOOQ	Não	Sim	Não	Não
TeraORM	Sim	Sim	Sim	Sim

- **QP1.** A adoção do TeraORM reduz o tamanho e melhora indicadores objetivos de manutenibilidade do código de acesso a dados, em relação ao SDK nativo?
- **QP2.** Como desenvolvedores percebem clareza, modificabilidade e segurança das duas abordagens ao implementar tarefas equivalentes?
- **QP3.** Qual abordagem os desenvolvedores preferem para uso futuro, e quais fatores explicam essa preferência?

O estudo combina dois métodos complementares para triangular respostas: uma análise estática comparativa (QP1) e um experimento controlado com percepção de usuários (QP2 e QP3).

A primeira frente consiste em uma comparação de antes e depois de um conjunto de serviços reais de acesso a dados. Partiu-se de uma base legada (T0), implementada com o SDK nativo do BigQuery e composição de SQL literal, e produziu-se uma versão refatorada (T1) empregando o TeraORM. Ambas as versões preservam cobertura funcional idêntica: oito serviços de domínio (grupo econômico, resumo de empreendimento, indicadores, relatórios de *onboarding*, auditoria de participante, produtividade, acompanhamento de SLA e funil por estágio) e dois módulos auxiliares. A equivalência funcional reduz diferenças de escopo entre as versões; contudo, a refatoração de T0 para T1 foi conduzida uma única vez por um dos autores, com domínio da API do TeraORM, de modo que a autoria da refatoração e a familiaridade com o ORM permanecem fatores de confusão, retomados na Seção 6.

Sobre esse par de *snapshots* foram calculadas duas métricas por meio de scripts automatizados e reproduzíveis, tanto de forma agregada quanto serviço a serviço. A primeira é o total de Linhas de Código (LOC), contabilizando apenas linhas não vazias e não comentadas. A segunda é uma estimativa do Índice de Manutenibilidade (IM) no estilo adotado por ferramentas contemporâneas, normalizada no intervalo de 0 a 100 e derivada do volume de Halstead, da complexidade ciclomática e do LOC agregados por *snapshot*, conforme a formulação:

$$IM = \max\left(0,\ \min\left(100,\ \frac{171 - 5{,}2\ln V - 0{,}23\,G - 16{,}2\ln L + C}{171} \times 100\right)\right) \tag{1}$$

em que V é o volume de Halstead, G a complexidade ciclomática, L o número de linhas de código e C o termo de ajuste associado à densidade de comentários. Complexidade ciclomática e volume de Halstead foram estimados a partir da contagem de estruturas de decisão e de *tokens*, após remoção de comentários e literais.

Tabela 2: Pares de desafios equivalentes do estudo intrassujeitos.

Par	Tarefa equivalente (variantes A e B)
AB01	Relatório de *onboarding* com filtros opcionais
AB02	Busca de empreendimentos com filtros compostos
AB03	Consulta pontual de horas por empreendimento
AB04	Ranking por horas totais com filtros

A segunda frente é um estudo intrassujeitos com medidas repetidas, no qual cada participante é exposto às duas abordagens. Foram elaborados quatro pares de desafios equivalentes, cada par contendo uma variante "A" (SDK nativo) e uma variante "B" (TeraORM) da mesma tarefa, conforme a Tabela 2. As tarefas cobrem construções comuns de acesso a dados: relatórios com filtros opcionais, buscas com filtros compostos, consulta pontual e ranking com ordenação. Além dos pares, o protocolo inclui um formulário inicial de perfil (FORM00) e um formulário final comparativo (FORM99). A ordem das abordagens não foi contrabalanceada entre os participantes; assim, efeitos de aprendizagem, fadiga e transferência entre tarefas equivalentes podem ter favorecido a abordagem realizada posteriormente, ameaça retomada na Seção 6.

Cada tarefa é validada automaticamente por dois critérios. O primeiro é a correspondência exata entre a saída produzida e o resultado esperado, verificada por testes automatizados executados com o Jest [20]. O segundo é um teste de marcador de abordagem: nas variantes "A", a submissão não pode importar o TeraORM; nas variantes "B", a submissão deve importar o TeraORM. Esse mecanismo garante que cada participante efetivamente exercite a abordagem pretendida em cada variante. As tarefas operam sobre tabelas de um projeto BigQuery dedicado ao estudo, com dados representativos de um processo de *onboarding* de empreendimentos.

Após implementar as tarefas, cada participante respondeu a um instrumento de percepção fundamentado no Modelo de Aceitação de Tecnologia [11], com itens medidos em escala Likert de cinco pontos [14, 25]. O instrumento contempla três dimensões. Para a abordagem com SDK, mede clareza, modificabilidade e propensão a erros do SQL literal; para o TeraORM, clareza, modificabilidade, explicitação da intenção, esforço mental e segurança. De forma comparativa, pergunta ainda qual abordagem foi mais fácil de entender, mais fácil de modificar e qual seria preferida em trabalhos futuros. Campos de texto livre coletaram a principal vantagem e a principal dificuldade percebidas em cada abordagem. Um formulário inicial registrou a familiaridade prévia dos participantes com SQL, com JavaScript/TypeScript e com ORMs.

O experimento contou com oito participantes, estudantes de cursos de computação do CEFET/RJ recrutados por conveniência, com perfis variados de experiência. A familiaridade prévia média, em escala de 1 a 5, foi de 3,13 para SQL, 3,38 para JavaScript/TypeScript e 2,50 para ORMs. Esses números apontam um público moderadamente familiarizado com programação e com bancos de dados, mas pouco habituado a bibliotecas de mapeamento entre objetos e relações. Esse perfil é relevante para a interpretação dos resultados de percepção, discutida na Seção 6.

Tabela 3: Métricas estáticas: SDK nativo (T0) versus TeraORM (T1). O símbolo ↓ indica redução e ↑ aumento em relação ao T0.

Métrica	T0 (SDK)	T1 (TeraORM)	Variação
Linhas de código (LOC)	334	221	↓ 33,8%
Complexidade ciclomática	18	13	↓ 5
Volume de Halstead	6305,30	5258,18	↓ 16,6%
Índice de Manutenibilidade	15,92	21,06	↑ 5,14

A coleta dos dados para a avaliação foi operacionalizada por meio de uma plataforma de avaliação automatizada de exercícios de programação, publicamente acessível e implantada em infraestrutura da RNP [13], tema relacionado à literatura de *automated assessment* [5, 8, 32] e aqui empregada como instrumento de experimentação, e não como objeto de estudo. Na plataforma, o código submetido por cada participante é executado dentro de um contêiner isolado e efêmero, criado dinamicamente por um orquestrador baseado em Kubernetes [3, 4], e os testes automatizados são executados nesse ambiente controlado. Para este estudo, foi utilizado um perfil de execução específico, previamente provisionado com o ORM proposto, seu adaptador para BigQuery, a integração NestJS [30] e o SDK nativo do BigQuery, todos sobre TypeScript [29]. A plataforma registrou automaticamente cada submissão, o resultado dos testes e as respostas aos formulários, o que garantiu rastreabilidade e reprodutibilidade da coleta e permitiu observar o processo iterativo de resolução, em que cada participante realiza múltiplas tentativas até que todos os testes sejam aprovados.

Pacote de replicação. Para favorecer a verificação independente, as versões T0 (SDK) e T1 (TeraORM) dos serviços e os *scripts* de cálculo de LOC e do Índice de Manutenibilidade estão disponíveis em repositório público [36].

5 Resultados

A Tabela 3 apresenta as métricas estáticas para as duas versões funcionalmente equivalentes. A migração do SDK nativo para o TeraORM reduziu o total de linhas de código de 334 para 221, uma diminuição de 113 linhas, equivalente a 33,8%; a complexidade ciclomática agregada caiu de 18 para 13, e o volume de Halstead reduziu de 6305,30 para 5258,18. Já o Índice de Manutenibilidade estimado, calculado pela Equação 1, aumentou de 15,92 para 21,06, um ganho de 5,14 pontos a favor do TeraORM.

A interpretação desses números requer cautela quanto à magnitude absoluta do IM. Os valores baixos observados em ambas as versões decorrem, em parte, do estilo da estimativa adotada e do fato de os serviços conterem lógica de composição de consulta densa. O elemento mais informativo, portanto, não é o valor absoluto, mas a *direção* e a *consistência* da variação: em todas as métricas consideradas, a versão com TeraORM apresentou-se mais enxuta e menos complexa, o que responde afirmativamente à QP1. A redução de um terço no tamanho do código, mantida a cobertura funcional, é o achado mais confiável, por ser diretamente observável e independente de calibração de fórmula.

Para evitar que um único serviço grande domine o resultado agregado, a Tabela 4 decompõe as métricas serviço a serviço, e a Figura 2 visualiza a variação pareada de LOC. A redução de

Tabela 4: Métricas por serviço: LOC antes (T0) e depois (T1), redução percentual e variação do Índice de Manutenibilidade (ΔIM). O IM é calculado por serviço; por ser não linear no tamanho, seus valores absolutos por serviço diferem do agregado da Tabela 3, mas sete dos oito serviços preservam a direção de melhora observada no agregado.

Serviço	LOC T0	LOC T1	ΔIM
Grupo econômico	33	23	+3,66
Resumo de empreendimento	36	24	+3,76
Indicadores (KPIs)	32	21	+4,40
Relatórios de *onboarding*	37	36	−1,08
Auditoria de participante	32	22	+3,94
Produtividade	33	21	+4,69
Acompanhamento de SLA	34	21	+5,02
Funil por estágio	32	22	+3,97

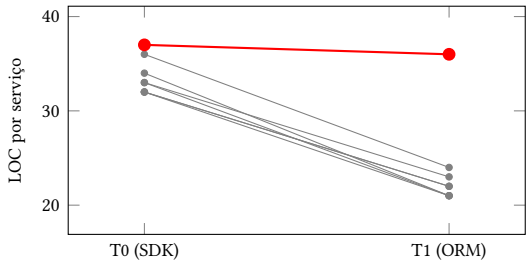

Figura 2: LOC pareado por serviço (T0 → T1). Todos os oito serviços apresentam redução de LOC; o serviço de relatórios de *onboarding*, destacado, reduz apenas 2,7% e permanece praticamente estável.

LOC ocorre em *todos* os oito serviços, com mediana de 32,3% (intervalo interquartil 31,0–34,9), e o Índice de Manutenibilidade melhora em sete dos oito. O único serviço fora desse padrão, o de relatórios de *onboarding*, concentra a menor redução (2,7%) e leve piora de IM; trata-se justamente do serviço com maior número de filtros opcionais, cuja lógica condicional persiste mesmo após a migração. Esse contraponto reforça, por contraste, que os ganhos se concentram onde a composição de consultas é mais densa e reduz o risco de o resultado agregado ser artefato de um único serviço.

A Tabela 5 resume as médias, as medianas e os desvios-padrão das respostas em escala Likert de cinco pontos para os itens relativos ao TeraORM, agregando os oito participantes. A abordagem com ORM foi bem avaliada quanto à explicitação da intenção da consulta (média 4,38), à clareza (média 4,13) e à segurança na composição de filtros e parâmetros (média 4,00), além da modificabilidade percebida, com média 3,75. O esforço mental necessário para estruturar a consulta recebeu a menor média entre os itens (3,38), resultado possivelmente relacionado à familiarização inicial com a API. Os desvios-padrão, todos abaixo de 0,75, indicam baixa dispersão e, portanto, boa concordância entre os participantes. A mediana se mantém em 4 para quatro dos cinco itens e cai para 3 apenas no esforço mental.

Quando confrontadas diretamente (Figura 3), a maioria (seis de oito) considerou o TeraORM mais fácil de modificar, enquanto

Tabela 5: Percepção do TeraORM (Likert de 1 a 5, $n = 8$).

Dimensão percebida	Média	Mediana	DP
Explicitação da intenção da consulta	4,38	4	0,52
Clareza da solução	4,13	4	0,64
Segurança na composição de filtros	4,00	4	0,53
Facilidade de modificação	3,75	4	0,71
Redução de esforço mental	3,38	3	0,74

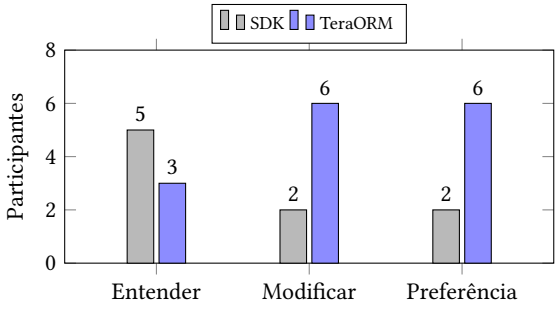

Figura 3: Comparação direta entre as abordagens ($n = 8$): facilidade de entendimento, facilidade de modificação e preferência para trabalhos futuros.

cinco dos oito acharam o SDK mais fácil de compreender. Esse contraste é central para a interpretação do estudo: a clareza imediata favoreceu o SQL literal, familiar à maioria, ao passo que a facilidade de manutenção favoreceu a abstração declarativa do ORM.

A avaliação subjetiva por par de desafio reforça essa leitura: o TeraORM foi avaliado de forma igual ou melhor que o SDK em três dos quatro pares, com exceção do AB03 (consulta pontual de uma linha), em que a solução direta com SDK foi percebida como ligeiramente mais simples. Os benefícios do ORM se acentuam, portanto, em consultas com filtros compostos e agregações, e se diluem em consultas triviais.

Quanto à preferência para trabalhos futuros, seis dos oito participantes escolheram o TeraORM e dois preferiram o SDK, conforme a Figura 3. Nos depoimentos em texto livre, quem preferiu o TeraORM destacou a explicitação da intenção, a maior segurança na composição de filtros e a redução de retrabalho ao modificar consultas; as principais dificuldades foram a falta de familiaridade com a API e a verbosidade em casos triviais. Já quem preferiu o SDK ancorou-se na expressividade direta e na preferência estética — um participante resumiu que "é mais bonitinho escrever *query*" —, justificativa coerente com a baixa familiaridade prévia com ORMs (2,50 em média) e com o maior esforço mental inicial da Tabela 5.

O uso da plataforma permitiu observar o processo de resolução: os registros mostram uma trajetória iterativa típica, com a pontuação evoluindo de nula a parcial até a aprovação integral. Nas variantes com ORM, uma falha recorrente nas tentativas iniciais foi o teste de marcador, que exige a importação explícita do TeraORM, indicando que parte do esforço inicial se concentrou na familiarização com a API. O desenho do estudo, contudo, não permite separar esse efeito de uma possível complexidade intrínseca da abordagem.

6 Conclusão e Trabalhos Futuros

Os resultados objetivos e subjetivos convergem: no agregado, houve redução de LOC (cerca de um terço), da complexidade ciclomática e do volume de Halstead e aumento do IM, que melhorou em sete dos oito serviços; do lado subjetivo, a maioria percebeu o ORM como mais fácil de modificar e o preferiu para uso futuro. O ponto de tensão é o descompasso entre clareza percebida e manutenibilidade: a mesma maioria julgou o SQL literal mais fácil de *entender* à primeira vista, mas o ORM mais fácil de *modificar*. Isso sugere que a familiaridade com SQL produz uma clareza imediata que não se traduz necessariamente em facilidade de evolução, e que decisões baseadas apenas nessa clareza inicial podem subestimar o custo de manutenção acumulado.

Um segundo achado é a dependência do benefício em relação à complexidade da consulta: os ganhos do ORM foram maiores em tarefas com filtros compostos e agregações e menores em consultas triviais, indicando que a abstração compensa mais em bases com consultas complexas e evolutivas.

A partir desses achados, a avaliação de ferramentas de acesso a dados analíticos deveria considerar o impacto de manutenibilidade ao longo do ciclo de evolução, e não só o desempenho; e a baixa familiaridade prévia com ORMs surge como explicação plausível para parte do esforço mental inicial, hipótese a investigar em estudos contrabalanceados e apoiada por documentação e exemplos.

Ameaças à validade.

- **Construção.** A manutenibilidade foi operacionalizada por LOC e por uma estimativa do IM (análise léxica, não sintática completa) [6, 16, 26]; são *proxies* estáticos que não medem diretamente o esforço ou o sucesso de atividades reais de manutenção.
- **Interna.** A refatoração de T0 para T1 foi feita uma única vez por um autor com domínio da API, de modo que os ganhos refletem um uso experiente e não podem ser atribuídos exclusivamente à abordagem; a ordem das abordagens também não foi contrabalanceada (possíveis efeitos de aprendizagem e ordem).
- **Externa.** Oito participantes e um domínio específico de *onboarding* sobre o BigQuery limitam a generalização; a percepção deve ser lida como indício exploratório, e não como evidência conclusiva.
- **Conclusão.** Dado o número reduzido de participantes, optou-se por estatística descritiva em vez de testes de hipótese, e as conclusões subjetivas são apresentadas como tendências observadas.

Como trabalhos futuros, planeja-se ampliar a amostra, diversificar domínios e *warehouses*, substituir as estimativas léxicas por análise estática completa, conduzir estudos longitudinais e contrabalanceados que meçam o custo de manutenção e avaliar empiricamente a governança de custo do TeraORM.

Disponibilidade de Artefatos

Os artefatos estão publicamente acessíveis: SDK e implementação do TeraORM; estudo de caso reproduzível; e plataforma EEVEE.

Referências

[1] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. 1994. The Goal Question Metric Approach. *Encyclopedia of Software Engineering* (1994).

[2] Praveen Borra. 2024. An overview of cloud data warehouses: Amazon redshift (aws), azure synapse (azure), and google bigquery (gcp). *international journal of advanced research in computer science* 15, 3 (2024), 23–27.

[3] Brendan Burns, Joe Beda, and Kelsey Hightower. 2019. *Kubernetes: Up and Running* (2 ed.). O'Reilly Media.

[4] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. Borg, Omega, and Kubernetes. *Commun. ACM* 59, 5 (2016), 50–57. doi:10.1145/2890784

[5] B. Cheang, A. Kurnia, A. Lim, and W.-C. Oon. 2003. On automated grading of programming assignments in an academic institution. *Computers & Education* 41, 2 (2003), 121–131. doi:10.1016/S0360-1315(03)00030-7

[6] Don Coleman, Dan Ash, Bruce Lowther, and Paul Oman. 1994. Using Metrics to Evaluate Software System Maintainability. In *Computer*, Vol. 27. 44–49. doi:10.1109/2.303623

[7] Derek Colley, Clare Stanier, and Md Asaduzzaman. 2020. Investigating the Effects of Object-Relational Impedance Mismatch on the Efficiency of Object-Relational Mapping Frameworks. *Journal of Database Management* 31, 4 (2020), 1–23. doi:10.4018/JDM.2020100101

[8] Sébastien Combéfis. 2022. Automated Code Assessment for Education: Review, Classification and Perspectives on Techniques and Tools. *Software* 1, 1 (2022), 3–30. doi:10.3390/software1010002

[9] Folasade Dahunsi, Ayobami Joseph, Oluwafemi Sarumi, and Olumide Obe. 2021. Database management system for mobile crowdsourcing applications. *Nigerian Journal of Technology* 40, 4 (2021), 713–727.

[10] Data Geekery. 2025. jOOQ: A Typesafe SQL DSL for Java. Disponível em: https://www.jooq.org/. Acesso em: 03 jul. 2026.

[11] Fred D. Davis. 1989. Perceived Usefulness, Perceived Ease of Use, and User Acceptance of Information Technology. *MIS Quarterly* 13, 3 (1989), 319–340. doi:10.2307/249008

[12] Drizzle ORM. 2025. Drizzle ORM: A TypeScript ORM for SQL Databases. Disponível em: https://orm.drizzle.team/. Acesso em: 03 jul. 2026.

[13] EEVEE Platform. 2026. EEVEE: Automated Assessment Platform (public instance). https://eevee.testbeds.rnp.br/. Instância pública implantada em infraestrutura da RNP. Acesso em: 03 jul. 2026.

[14] Amanda Feijó, Ernesto Vicente, and Sérgio Petri. 2020. O Uso das Escalas Likert nas Pesquisas de Contabilidade. *Revista Gestão Organizacional* 13 (2020), 27–41. doi:10.22277/rgo.v13i1.5112

[15] António Gonçalves and Anacleto Correia. 2024. The OECD-Inspired ORM Blueprint for Personal Data Security. In *19th Iberian Conference on Information Systems and Technologies (CISTI'2024)*. Salamanca, Spain.

[16] Maurice H. Halstead. 1977. *Elements of Software Science*. Elsevier North-Holland, New York, NY, USA.

[17] Ilja Heitlager, Tobias Kuipers, and Joost Visser. 2007. A Practical Model for Measuring Maintainability. In *6th International Conference on the Quality of Information and Communications Technology (QUATIC)*. 30–39. doi:10.1109/QUATIC.2007.8

[18] International Organization for Standardization. 2011. ISO/IEC 25010:2011 Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models. (2011). Standard.

[19] Christopher Ireland, David Bowers, Michael Newton, and Kevin Waugh. 2009. A Classification of Object-Relational Impedance Mismatch. *2009 First International Confernce on Advances in Databases, Knowledge, and Data Applications* (2009), 36–43. doi:10.1109/DBKDA.2009.11

[20] Jest Documentation. 2025. Jest: Delightful JavaScript Testing. Disponível em: https://jestjs.io/. Acesso em: 01 jul. 2026.

[21] Vinicius Cardoso Jungers, Eduardo Aparecido de Oliveira, and Vinicius Aparecido De Souza. 2024. Comparação de Desempenho entre os ORMs TypeORM, Prisma e Sequelize em Aplicações Node. js. *Revista Eletro^nica e-Fatec* 14, 2 (2024).

[22] Barbara A. Kitchenham, Shari Lawrence Pfleeger, Lesley M. Pickard, Peter W. Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. 2002. Preliminary Guidelines for Empirical Research in Software Engineering. *IEEE Transactions on Software Engineering* 28, 8 (2002), 721–734. doi:10.1109/TSE.2002.1027796

[23] Knex.js. 2025. Knex.js: A SQL Query Builder for JavaScript. Disponível em: https://knexjs.org/. Acesso em: 03 jul. 2026.

[24] Kysely. 2025. Kysely: The Type-Safe SQL Query Builder for TypeScript. Disponível em: https://kysely.dev/. Acesso em: 03 jul. 2026.

[25] Rensis Likert. 1932. A Technique for the Measurement of Attitudes. *Archives of Psychology* 22, 140 (1932), 1–55.

[26] Thomas J. McCabe. 1976. A Complexity Measure. *IEEE Transactions on Software Engineering* SE-2, 4 (1976), 308–320. doi:10.1109/TSE.1976.233837

[27] Erik Meijer, Brian Beckman, and Gavin Bierman. 2006. LINQ: Reconciling Object, Relations and XML in the .NET Framework. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. 706. doi:10.1145/1142473.1142552

[28] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: Interactive Analysis of Web-Scale Datasets. *Proceedings of the VLDB Endowment* 3, 1–2 (2010), 330–339. doi:10.14778/1920841.1920886

[29] Microsoft. 2025. TypeScript: JavaScript With Syntax For Types. Disponível em: https://www.typescriptlang.org/. Acesso em: 01 jul. 2026.

[30] NestJS Documentation. 2025. NestJS: A progressive Node.js framework. Disponível em: https://nestjs.com. Acesso em: 01 jul. 2026.

[31] Paul Oman and Jack Hagemeister. 1992. Metrics for Assessing a Software System's Maintainability. In *Proceedings Conference on Software Maintenance (ICSM)*. 337–344. doi:10.1109/ICSM.1992.242525

[32] José Paiva, José Leal, and Álvaro Figueira. 2022. Automated Assessment in Computer Science Education: A State-of-the-Art Review. *ACM Transactions on Computing Education* 22, 3 (2022). doi:10.1145/3513140

[33] Prisma. 2025. Prisma ORM: Next-generation Node.js and TypeScript ORM. Disponível em: https://www.prisma.io/. Acesso em: 03 jul. 2026.

[34] QueryDSL. 2025. QueryDSL: Unified Queries for Java. Disponível em: https://querydsl.com/. Acesso em: 03 jul. 2026.

[35] Blind Review. Blind Review. Blind Review. *Blind Review* Blind Review, Blind Review (Blind Review), Blind Review.

[36] TeraORM Project. 2026. TeraORM Study Case: Reproducible Before/After Maintainability Artifacts. https://github.com/garymabu/TeraORM-Study-Case. Repositório com serviços T0 (SDK) e T1 (TeraORM) e scripts de LOC e Índice de Manutenibilidade. Acesso em: 03 jul. 2026.

[37] Pieter van Zyl, Derrick G. Kourie, and Andrew Boake. 2006. Comparing the performance of object databases and ORM tools. *Proceedings of the 2006 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists (SAICSIT)* (2006), 1–11. doi:10.1145/1216262.1216263

[38] Raul Wazlawick. 2017. *Metodologia de Pesquisa para Ciência da Computação*. Elsevier Brasil.

[39] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. 2012. Experimentation in Software Engineering. (2012). doi:10.1007/978-3-642-29044-2