Manutenção e Evolução do Sistema BQBus usando Spec-Driven Development

Marcelo Augusto Mrad Marteleto
Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte, MG, Brasil
marcelomrad@gmail.com

Marco Túlio Valente (Orientador)
Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte, MG, Brasil

***Resumo*—A proliferação de ferramentas de IA generativa no desenvolvimento de software coloca uma questão central ainda não respondida empiricamente: *como manter e evoluir software gerado majoritariamente por IA?* Este artigo descreve a POC 2 do projeto BQBus, um sistema de recomendação de rotas de ônibus para Barbacena-MG, em que um sistema real de 22.148 linhas de código (LOC), construído 100% por IA na POC 1, foi abandonado por três meses e depois ressuscitado, corrigido e evoluído por um único desenvolvedor utilizando Spec-Driven Development (SDD) e Claude Code como agente de IA. Os resultados mostram: (RQ1) 54% das especificações SDD permaneceram precisas após três meses; (RQ2) 30 *code smells* catalogados, com duplicação e código morto correspondendo a 37% do total; (RQ3) IA corrigiu 30/30 smells em ≈380 min (≈21× mais rápido que estimativa manual); (RQ4) sete features novas em ≈1,75 h reais (fator 21×); (RQ5) o algoritmo RAPTOR mostrou-se 4,7× mais rápido no processamento server-side e semanticamente superior ao Dijkstra (baseline POC 1), considerando horários reais e retornando rotas Pareto-ótimas. No total, as ≈438 h estimadas de trabalho manual foram realizadas em ≈23,4 h reais com SDD + IA (≈95% de economia).**

***Index Terms*—Spec-Driven Development, IA Generativa, Software Legado, Manutenção de Software, RAPTOR, Claude Code, Qualidade de Software, Code Smells**

I. Introdução

A IA generativa transformou a velocidade de construção de software [4]. Estudos recentes mostram acelerações de 20–55% em tarefas isoladas de programação [5], [6]; na POC 1 do BQBus, 22.148 LOC foram produzidas em ≈30 h reais, um fator de 20,2× sobre a estimativa manual de 605 h [7].

Porém, um sistema construído rapidamente por IA precisa, eventualmente, de manutenção: correção de bugs, adição de features, evolução da arquitetura. A pergunta que ainda carece de evidência empírica em projetos reais é:

> *Spec-Driven Development facilita a manutenção e evolução de software gerado por IA?*

Para investigar essa questão, conduzimos a **POC 2** do BQBus, que simula o cenário clássico de *software legado*: o sistema ficou parado três meses após a POC 1, o desenvolvedor original estava ausente, a infraestrutura estava derrubada e o código havia acumulado dívida técnica. Neste cenário, aplicamos SDD + Claude Code para ressuscitar, diagnosticar, corrigir e evoluir o sistema.

As contribuições deste trabalho são:

1) Primeiro catálogo quantitativo de *code smells* em sistema real de 22 K LOC gerado por IA;
2) Evidência empírica da acurácia de specs SDD após 90 dias de *drift*;
3) Implementação e benchmark do algoritmo RAPTOR [1] como substituto do Dijkstra em rede real de transporte;
4) Protocolo replicável de ressurreição de legado com SDD.

II. Fundamentação Teórica

A. IA Generativa no Desenvolvimento de Software

A adoção de ferramentas de IA generativa na engenharia de software cresceu exponencialmente desde 2022. GitHub Copilot, lançado em disponibilidade geral em junho de 2022, atingiu mais de um milhão de usuários em menos de um ano [4]. Ferramentas subsequentes — ChatGPT, Claude, Gemini, Cursor — amplificaram o alcance ao público não especializado. A consequência prática foi uma aceleração significativa na escrita de código: estudos controlados reportam ganhos de 35–55% em tarefas isoladas [5], [6]; em projetos *greenfield* com escopo bem definido, como a POC 1 deste trabalho, o fator pode ultrapassar 20× [7].

Contudo, esse ganho na **construção** levanta uma questão simétrica ainda não respondida: quanto do esforço economizado na construção é redepositado na manutenção? A literatura clássica de engenharia de software estabelece que 60–80% do custo total de um sistema ocorre após a entrega inicial [16]. Se o código gerado por IA acumula dívida técnica mais rapidamente do que código escrito por humanos — ou se a ausência de documentação torna o onboarding do próximo desenvolvedor (humano ou agente) mais custoso — parte do ganho inicial é ilusório. Esta é a hipótese que o presente trabalho testa empiricamente.

B. Code Smells e Dívida Técnica em Código Gerado por IA

Code smells são sintomas superficiais de problemas mais profundos no design do software, sistematizados por Fowler [14] em 22 categorias canônicas. Não são bugs em si (o código funciona), mas indicadores de que o código será difícil de entender, testar ou modificar. As categorias mais impactantes para manutenibilidade são: God Class (uma classe que faz demais), Duplicated Code (lógica copiada em vez de extraída), Dead Code (artefatos não mais usados), Feature

Envy (método que usa mais dados de outras classes do que da própria) e Shotgun Surgery (uma mudança conceitual dispara modificações em muitas classes).

A hipótese levantada neste trabalho — e parcialmente validada — é que a IA generativa produz um subconjunto previsível de smells, derivado de um padrão cognitivo específico do LLM: **contexto de janela limitado combinado com ausência de visão sistêmica**. Em cada sessão de chat, o agente enxerga apenas o arquivo em foco; não tem ciência de que a mesma função já existe em outra classe, de que um módulo que criou para teste nunca foi removido, ou de que um campo hardcoded foi tornado configurável três commits depois em outro arquivo. Esse padrão gera os três smells mais frequentes que identificamos: duplicação, dead code e mocks em produção — todos consequências diretas de *sessões de IA sem memória compartilhada*.

O SDD mitiga esse problema ao fornecer ao agente um mapa explícito do sistema (`design.md`) antes de cada sessão, reduzindo a probabilidade de reimplementação acidental.

C. Algoritmos de Roteamento em Transporte Público

O problema de encontrar a melhor rota em uma rede de transporte público difere fundamentalmente do problema em redes rodoviárias. Em rodovias, o grafo é estático e pesos são distância ou tempo de percurso. Em transporte público, o grafo é *time-dependent*: a aresta (parada A, parada B) só existe quando um ônibus parte de A em direção a B, e o tempo de espera depende do horário de consulta. Além disso, o critério de otimização raramente é único: o usuário quer minimizar tempo total *e* número de transferências simultaneamente — um problema de otimização multiobjetivo cujas soluções formam o conjunto Pareto-ótimo.

Os principais algoritmos da literatura, em ordem cronológica de publicação:

- **Dijkstra adaptado (1959/2000s)**: modifica pesos para incorporar tempo de espera. Simples de implementar, mas ignora horários reais (assume frequência uniforme) e retorna apenas uma rota.
- **Time-Dependent Dijkstra**: expande o grafo para incluir nós temporais. Correto, mas com explosão combinatória em redes médias.
- **RAPTOR (2012)** [1]: opera em rodadas sobre dados de viagens reais (*stop_times*), retorna earliest arrival por número de transferências (conjunto Pareto-ótimo nativo), e usa early stopping quando nenhuma parada melhora numa rodada.
- **Connection Scan Algorithm — CSA (2013)**: varre conexões ordenadas por horário de partida; latência muito baixa mas sem suporte nativo a Pareto.
- **McRAPTOR**: extensão do RAPTOR com critérios de conforto (lotação, acessibilidade).

Para o BQBus, o RAPTOR foi escolhido pela combinação de corretude semântica (horários reais), saída Pareto nativa e simplicidade de implementação na escala de Barbacena (31 linhas, 224 paradas). O CSA teria latência menor, mas sem vantagem prática para uma rede desta dimensão.

D. Metodologias de Governança para Desenvolvimento com IA

A literatura emergente de engenharia de software com IA identifica três abordagens de governança:

(1) *Vibe Coding* (Andrej Karpathy, 2025) [17]: desenvolvimento totalmente conversacional, sem documentação formal. Adequado para protótipos de vida curta, mas produz sistemas *orphaned by design*: o contexto necessário para mantê-los existe apenas no histórico de chat da ferramenta, inacessível a novos desenvolvedores ou agentes.

(2) *Prompt Engineering* sistemático: estrutura os prompts mas não persiste as decisões de design no repositório. Melhora consistência dentro de uma sessão, mas não resolve o problema de continuidade entre sessões.

(3) Spec-Driven Development (SDD): persiste as especificações como artefatos versionados no repositório. Resolve o problema de continuidade ao custo de disciplina explícita na manutenção dos docs. É o objeto central deste trabalho.

III. Spec-Driven Development

A. Definição e Motivação

Spec-Driven Development (SDD) é uma metodologia em que **especificações estruturadas são escritas antes do código** e mantidas como documentos vivos no repositório, servindo de fonte de verdade para agentes de IA [3]. A alternativa mais comum — o *vibe coding* — usa descrições conversacionais cujo contexto morre na sessão de chat. O SDD resolve isso ao versionar as decisões junto ao código.

A diferença central entre as abordagens está resumida na Tabela I.

Tabela I
Vibe Coding vs. Spec-Driven Development

Dimensão	Vibe Coding	SDD
Planejamento	Conversacional, implícito	Estruturado, explícito
Fonte de verdade	Histórico de chat (efêmero)	Arquivos no repo (persistente)
Paralelização	Alto risco de conflito	Coordenação determinística
Sobrevivência	Nenhuma	Mantida indefinidamente
Retrabalho	Focado em debug	Focado em planejamento

B. Níveis de Maturidade

O SDD organiza-se em três níveis crescentes de governança [3]:

1) Nível 1 — Spec-First: A especificação é escrita antes da implementação, mas pode ser descartada após a conclusão da feature. Útil para tarefas isoladas; a limitação é que a documentação não sobrevive para consulta futura.

2) Nível 2 — Spec-Anchored: As especificações são mantidas como *documentos vivos* no repositório, atualizadas ao longo do ciclo de vida da feature e durante manutenção. Registram o histórico de decisões e *trade-offs*. Exige disciplina de equipe para evitar *documentation decay*. Este é o nível praticado na POC 2.

3) Nível 3 — Spec-as-Source: A especificação é a única fonte autoritativa; o código é gerado automaticamente a partir dela. Atualmente experimental (não aplicado neste trabalho).

C. Arquitetura de Implementação

A implementação do SDD na POC 2 utiliza quatro blocos construtivos do Claude Code (Tabela II):

Tabela II
COMPONENTES SDD NO CLAUDE CODE

Componente	Papel	Ativação
`CLAUDE.md`	Regras de projeto, persistentes	Sempre ativa
Skills	Templates de fluxo SDD	Por relevância
Subagentes	Instâncias Claude isoladas	Delegação manual
Hooks	Scripts em eventos	Automático

D. Fluxo de Phase Gates

O fluxo SDD da POC 2 é orquestrado pela *skill* `sdd-workflow` e impõe quatro **phase gates** sequenciais, com pausa obrigatória para aprovação humana entre cada fase (Figura 1):

Figura 1. Phase gates do SDD na POC 2. Cada losango representa uma pausa para aprovação humana antes do próximo artefato.

E. Disciplina de Métricas

Uma regra inviolável da POC 2 é que *cada task implementada gera dados para o TCC*. Ao concluir uma task, o desenvolvedor (ou o agente) registra em `specs/<eixo>/metricas.md`:

- tempo real gasto (não estimado);
- LOC adicionadas / modificadas / removidas;
- testes adicionados / passando;
- smells resolvidos / novos smells introduzidos;
- percentual do código gerado por IA.

Sem registro de métrica, a task não é considerada completa.

F. Subagentes e Coordenação Paralela

A POC 2 utilizou três subagentes especializados em paralelo:

- `spec-writer`: redige fases de spec a partir de descrições;
- `task-implementer`: implementa uma task atômica com teste e commit;
- `qualidade-reviewer`: revisa código quanto a smells e adere à spec.

O SDD é o que torna isso seguro: com specs aprovadas, agentes paralelos recebem o mesmo plano e não geram código conflitante.

IV. O PROJETO BQBUS

A. Contexto

Barbacena-MG (aproximadamente 140.000 habitantes) não possui um aplicativo oficial de transporte público. O site da prefeitura oferece apenas informações de itinerário em formato texto, sem integração com localização ou roteiro personalizado. O BQBus preenche essa lacuna com um sistema de recomendação de rotas ponto-a-ponto.

B. POC 1 — Construção do Zero

A POC 1 foi executada entre outubro de 2025 e janeiro de 2026 utilizando **GitHub Copilot** (backend) e **Lovable.dev** (frontend), com metodologia SDD baseada em cinco personas de agentes (*Steering, Planner, Executor, Reviewer, Document AI*) [8].

Resultados da POC 1:

- 22.148 LOC (13.971 back + 8.177 front), 35 features;
- 30 testes unitários passando;
- Algoritmo Dijkstra, cache Redis, API REST completa;
- 605 h estimadas sem IA; ≈30 h reais — fator **20,2**×;
- Stack: Java 17 + Spring Boot 3.2 + PostgreSQL/PostGIS + React 18 + TypeScript.

Após a entrega, o sistema ficou **parado por três meses** sem manutenção.

C. POC 2 — Cenário de Legado

A POC 2 (maio–junho de 2026) simula o retorno ao sistema após esse período: infraestrutura derrubada, desenvolvedor original ausente, smells acumulados, specs potencialmente desatualizadas. O agente de IA utilizado foi **Claude Code (Opus 4.7)**, sem acesso a nenhum contexto da sessão anterior.

O trabalho foi organizado em **cinco Eixos**, cada um com sua própria spec SDD aprovada (Tabela III).

Tabela III
EIXOS DA POC 2 E PERGUNTAS DE PESQUISA

Eixo	Foco	RQ	Status
1	Ressurreição	RQ1	✓ 100%
2	Qualidade (smells)	RQ2, RQ3	✓ 100%
3	Evolução (10 features)	RQ4	✓ 70%
4	RAPTOR (algoritmo)	RQ5	✓ 100%
5	Deploy / CI-CD	M6	≈35%

V. METODOLOGIA

A. Perguntas de Pesquisa

RQ1 Qual a acurácia das especificações SDD após 90 dias de inatividade?

RQ2 Quais padrões de *code smells* são recorrentes em código gerado por IA?

RQ3 Com qual eficiência a IA corrige smells que ela própria gerou?

RQ4 Qual o fator de produtividade SDD+IA na manutenção e evolução de features?

RQ5 O RAPTOR melhora viabilidade e latência em relação ao Dijkstra sem custo adicional?

B. Protocolo de Execução

Cada Eixo seguiu o fluxo SDD (Seção III): (1) requirements aprovados por humano; (2) design aprovado por humano; (3) tasks aprovadas por humano; (4) implementação atômica (uma task = um commit = uma entrada em metricas.md).

Todas as métricas foram coletadas pelo próprio agente (tempos de sessão, contagem de LOC via `git show -stat`, testes via `mvn test` / `npm run lint`).

C. Stack Tecnológico

Backend: Java 17, Spring Boot 3.2, Spring Data JPA, PostgreSQL 15 + PostGIS 3.3, Redis (cache), Flyway (migrations), Resilience4j (circuit breaker), JaCoCo (cobertura), Testcontainers.
Frontend: React 18, Vite 5, TypeScript 5 (*strict*), TanStack Query 5, shadcn/ui (Radix), Tailwind CSS, react-leaflet.
Agente: Claude Code (Opus 4.7) com subagentes paralelos.

VI. Eixo 1 — Ressurreição

O Eixo 1 responde RQ1: em ≈50 min (sessão única), Claude Code:

1) Verificou containers Docker (Postgres/PostGIS + Redis já UP);
2) Subiu o backend (`mvn spring-boot:run`) — health 200 em 2,1 s;
3) Executou `mvn test`: **142/149 passando (95,3%)**;
4) Verificou E2E: API retornou HTTP 200 com cache Redis ativo (849 ms → 6 ms);
5) Auditou os docs SDD da POC 1 contra o código real.

Achado crítico: o banco local tinha 225 paradas com 0% de coordenadas geográficas (ETL parcial antigo); o backup de produção tinha 224 paradas com 100% de coordenadas. O sistema precisava do backup para funcionar de ponta a ponta. A restauração foi executada como Task 2.5.

A. RQ1 — Acurácia dos Docs SDD

Foram auditadas 25 afirmações verificáveis nos quatro documentos principais da POC 1 (`requirements.md` e `design.md` de cada spec). O resultado está na Tabela IV.

Tabela IV
Acurácia dos Docs SDD da POC 1 após 90 dias (RQ1)

Classificação	N	%
Acurado	9	36%
Parcial	9	36%
Desatualizado	7	28%
Acurácia ponderada (acurado + 0,5×parcial)	**54%**	

Desvios mais comuns: endpoints removidos (ex.: `/holidays/{year}` jamais criado), enums renomeados (`dayType` hardcoded), mocks em produção não removidos. Os docs de *design* foram mais estáveis que os de *requirements*, pois descrevem estruturas (pacotes, schema) que mudam menos.

Interpretação: 54% é suficiente para orientar onboarding de IA (o agente navegou rapidamente para os arquivos corretos usando os docs), mas insuficiente como oráculo. A spec precisa de re-validação antes de servir de fonte de verdade — esse diagnóstico (Eixo 1) é a primeira task obrigatória em qualquer manutenção.

B. Protocolo de Ressurreição SDD

Um achado secundário mas altamente reutilizável do Eixo 1 é o **Protocolo de Ressurreição SDD**: uma sequência determinística de seis passos que qualquer agente de IA pode executar para retomar um sistema legado com documentação SDD, sem acesso ao desenvolvedor original (Tabela V).

Tabela V
Protocolo de Ressurreição SDD — 6 Passos

#	Passo	Ação	Tempo
1	Diagnóstico de infra	Verificar containers, portas, variáveis de ambiente. Subir serviços na ordem correta.	5 min
2	Health check	`mvn spring-boot:run` + `GET /actuator/health`. Confirmar status 200.	3 min
3	Execução de testes	`mvn test` / `npm run lint`. Registrar linha base: N/M passando.	5 min
4	Auditoria de specs	Para cada `requirements.md` e `design.md`: verificar cada afirmação contra o código real. Classificar: acurado / parcial / desatualizado.	25 min
5	Auditoria de dados	Verificar integridade do banco: coordenadas geográficas, FK órfãs, migração completa.	7 min
6	Relatório de dívida	Consolidar: testes falhando, specs desatualizadas, smells identificados. Servir de insumo para Eixos seguintes.	5 min
Total			**≈50 min**

A eficácia do protocolo depende diretamente da qualidade das specs existentes. No BQBus, a acurácia de 54% foi suficiente para os Passos 4 e 5: mesmo com 28% das afirmações desatualizadas, o agente identificou rapidamente os arquivos relevantes usando os docs de *design* (mais estáveis) como mapa. Estimamos que, sem nenhuma documentação SDD, o Passo 4 seria substituído por exploração manual do repositório, multiplicando o tempo por 4–6×.

Replicabilidade: o protocolo foi executado em uma única sessão de Claude Code de ≈50 min. Todo o log de comandos foi registrado em `specs/eixo-1-ressurreicao/metricas.md`, tornando a execução auditável e reproduzível por outros pesquisadores.

VII. Eixo 2 — Qualidade: Catálogo de Smells

A. Catalogação (RQ2)

Foram identificados e revalidados **32 candidatos a smell**, dos quais **30 confirmados** (94%), 1 desclassificado e 1 reclassificado. A Tabela VI apresenta a distribuição por categoria.

Tabela VI
DISTRIBUIÇÃO DOS 30 CODE SMELLS POR CATEGORIA (RQ2)

Categoria	N	%
Duplicação de código	4	13,3
Dead code residual	4	13,3
Mock em produção	3	10,0
Type safety relaxada	3	10,0
Magic numbers / hardcoded	3	10,0
Error handling inadequado	3	10,0
God Class / Long Method	2	6,7
Acoplamento	2	6,7
Bloat / dependências mortas	2	6,7
Lógica frágil	2	6,7
Performance / estado	2	6,7
Total	**30**	**100%**

Achado central de RQ2: os smells são predominantemente **arquitetural-cognitivos** — resultam da IA não enxergar o sistema como um todo e não fechar ciclos quando muda de rumo. Os três maiores grupos (Duplicação + Dead code + Mock em prod) representam 11/30 = **37%** e são todos consequências do mesmo padrão: *a IA reimplementa em vez de extrair* e *não remove o que tornou obsoleto*.

Exemplos representativos:

- **S1 (Haversine 4×)**: a mesma função de distância geoespacial foi implementada independentemente em 4 classes, pois cada sessão de IA não sabia da existência das outras;
- **S14 (mock 800 ms)**: `useScheduleExpansion` retornava dados fictícios com `setTimeout(800)` — a IA criou o mock para testar a UI e nunca voltou para substituí-lo;
- **S18 (35 componentes shadcn mortos)**: 35 dos 49 componentes shadcn/ui instalados jamais foram referenciados, custando 3.356 LOC.

B. Correção (RQ3)

Os 30 smells foram corrigidos em **quatro lotes** organizados por severidade (Crítica → Alta → Média → Baixa), cada lote com sua própria entrada em `tasks.md`. Cada correção gerou um commit atômico.

Resultados:

- 30/30 smells resolvidos, 0 novos smells introduzidos;
- 21 commits atômicos, zero regressão em testes ou build;
- ≈380 min de IA (≈6h20) totais;
- ESLint: 16 → 7 problemas (9, sendo 5 errors);
- TS strict errors: 13 → 0;
- LOC líquido: 3.559 (front 3.088, back 471);
- Bundle CSS: 46%.

Fator RQ3: ≈13 min/smell com IA vs ≈3 h/smell estimado manualmente = ≈**14**× mais eficiente.

C. Severidade e Impacto dos Smells

A Tabela VII classifica os 30 smells por severidade, definida pelo impacto na manutenibilidade e no risco de regressão. Smells *críticos* afetam corretude em produção; *altos* prejudicam diretamente a testabilidade ou a segurança; *médios* e *baixos* afetam legibilidade e custo de evolução.

Tabela VII
SMELLS POR SEVERIDADE E ESFORÇO DE CORREÇÃO (RQ3)

Severidade	N	Tempo IA	Est. manual
Crítica	4	≈52 min	≈16 h
Alta	10	≈140 min	≈40 h
Média	12	≈120 min	≈36 h
Baixa	4	≈68 min	≈12 h
Total	**30**	**≈380 min**	**≈104 h**

Os quatro smells **críticos** merecem destaque por seu impacto direto no comportamento em produção:

- **S14 — Mock em produção com delay de 800 ms**: `useScheduleExpansion` simulava horários com `setTimeout(800)`, entregando dados falsos para usuários reais. Corrigido em 15 min: a função foi substituída pela chamada real ao backend.
- **S19 — Geocodificador sem circuit breaker**: chamadas ao Google Maps sem fallback travavam o servidor por timeout (30 s) quando a cota era excedida. Corrigido com Resilience4j em 20 min.
- **S3 — God Class `RouteController` (684 LOC)**: acumulava lógica de validação, montagem de resposta, cálculo de score e logging — impossível de testar unitariamente. Extraído para `RouteResponseAssembler`: controller caiu para 528 LOC (23%).
- **S25 — Autocomplete retornando dados mock**: campo de autocomplete do frontend usava lista estática de endereços de Barbacena em vez de chamar o Nominatim real. Corrigido e virou a feature F2 (Eixo 3).

D. Impacto em Métricas de Qualidade Objetivas

A Tabela VIII apresenta as métricas CK (*Chidamber and Kemerer*) [15] coletadas antes e depois do Eixo 2 para as cinco classes de maior complexidade do backend.

Tabela VIII
MÉTRICAS CK DAS 5 CLASSES MAIS COMPLEXAS — ANTES/DEPOIS (RQ3)

Classe	WMC		CBO		LOC	
	Antes	Depois	Antes	Depois	Antes	Depois
RouteController	38	24	12	8	684	528
DijkstraCalc.	22	22	9	7	312	298
DataLoadingOrch.	19	15	11	9	287	261
GeocodingService	17	13	8	6	241	198
RouteSearchSvc.	14	14	7	7	189	189
Total	110	88	47	37	1713	1474
Redução	20%		21%		14%	

onde **WMC** (*Weighted Methods per Class*) mede complexidade ciclomática total da classe, e **CBO** (*Coupling Between*

Objects) mede o número de outras classes acopladas. Reduções de 20% em WMC e 21% em CBO são significativas: cada ponto a menos em CBO reduz o raio de impacto de uma mudança, diminuindo o risco de regressão.

Papel do SDD na correção: os critérios de aceitação dos smells em `tasks.md` especificavam métricas-alvo (ex.: "`RouteController` deve ter WMC $\leq$ 25 após extração do assembler"), o que guiou o agente com precisão cirúrgica — sem over-engineering e sem risco de esquecer o escopo.

VIII. Eixo 3 — Evolução de Features

A. Features Implementadas

Sete das dez features planejadas foram entregues em ≈1,75 h reais (Tabela IX). As três restantes (PWA, acessibilidade, telemetria) foram declaradas *buffer* cortável na proposta.

Tabela IX
Features Entregues no Eixo 3 (RQ4)

ID	Feature	Tempo	Fator	LOC+
F1	Mapa Leaflet (react-leaflet)	15 min	32×	93
F2	Autocomplete real (Nominatim)	33 min	18×	57
F3	Horários reais por trecho	20 min	24×	15
F4	Seletor tipo de dia	10 min	24×	12
F5	RAPTOR na UI (chega às HH:mm)	10 min	—	6
F6	Badge de confiança da rota	10 min	18×	18
F9	Multi-rota Pareto	20 min	24×	76
Total		**≈118 min**	**≈21×**	**277**

Achado de RQ4: SDD facilitou a implementação das features F3, F5 e F9 porque o design do Eixo 4 (RAPTOR) já havia mapeado exatamente quais arquivos e campos usar — o agente navegou diretamente para os pontos corretos sem exploração.

IX. Eixo 4 — Algoritmo RAPTOR

A. Motivação

O Dijkstra da POC 1 opera sobre um grafo estático de custos — ignora horários reais e retorna apenas uma rota. Para uma cidade real, a pergunta relevante é *"se eu sair agora, quando chego?"*, que requer um algoritmo sensível a horários. O RAPTOR (Round-Based Public Transit Routing, Delling et al. [1]) foi escolhido por:

- retornar earliest arrival por número de transferências;
- produzir naturalmente o conjunto Pareto-ótimo (tempo × transferências);
- operar em rodadas, o que permite early stopping trivial;
- ser academicamente consolidado para redes de transporte público.

B. Modelo de Dados do RAPTOR

O RAPTOR opera sobre três estruturas de dados centrais que diferem do modelo relacional usado pelo Dijkstra. Em vez de um grafo de arestas ponderadas, o RAPTOR precisa de *viagens* com horários discretos (Tabela X).

Tabela X
Estruturas de Dados do RAPTOR vs. Entidades do BQBus

Estrutura RAPTOR	Campos	Origem no BQBus
`stop_times`	stopId, routeId, tripId, departureSecs, arrivalSecs	`Horario` + `Sequencia-RuaItinerario`
`routes`	routeId, stops[ ] (sequência ordenada)	`Linha` + `ParadaOnibus`
`transfers`	fromStop, toStop, walkTimeSecs	`TransportGraph` (TRANSFERENCIA)

A escolha de derivar as três estruturas das entidades existentes — sem criar novas tabelas no banco — foi uma decisão de design especificada em `eixo-4-raptor/design.md`: *"RAPTOR usa leitura diferente das mesmas tabelas; nenhum schema novo; compatibilidade total com o Dijkstra existente."*. Esse nível de detalhe no design foi possível **apenas** porque as entidades do BQBus estavam documentadas nos `design.md` dos Eixos 1 e 2 — um exemplo concreto de como specs anteriores habilitam decisões futuras com zero exploração redundante.

A classe `RaptorDataLoader` lê as três estruturas na inicialização do contexto Spring e armazena em memória (2,4 MB para 224 paradas e 1.652 horários), tornando cada chamada de roteamento um processo puramente em memória sem acesso ao banco durante o cálculo.

C. Implementação

A implementação introduziu sete classes novas em `service/routing/raptor/`:

Listing 1. Núcleo do RAPTOR: rodada + embarque mais cedo

```
// Rodada k: para cada rota com parada marcada
for (RaptorRoute route : raptorData.getRoutes()) {
  Trip trip = null; // viagem mais cedo ativa
  for (StopInRoute s : route.getStops()) {
    if (markedStops.contains(s.stopId()) && trip ==
        null)
      trip = pickEarliestTrip(route, s.stopId(), tau
          [k-1], tipoDia);
    if (trip != null) {
      int arrival = trip.departureFromOrigin() + s.
          offsetSecs();
      if (arrival < tau[k][s.stopId()]) {
        tau[k][s.stopId()] = arrival;
        parent[k][s.stopId()] = new ArrivalLabel(
            trip, s);
        newlyMarked.add(s.stopId());
      }
    }
  }
}
```

O RAPTOR é exposto via `RouteCalculationStrategy`, compartilhando a mesma interface do Dijkstra (criada no Eixo 2/S3), e selecionável via `?engine=raptor|dijkstra` (default: raptor).

D. Benchmark (RQ5)

O benchmark foi executado com 50 pares O-D reais das 224 paradas de Barbacena, `dayType=WEEKDAY`,

`dateTime=2026-06-23T08:00:00`, Redis limpo antes de cada rodada. Os 50 pares foram distribuídos em seis categorias para cobrir os casos canonicamente difíceis em roteamento de trânsito público (Tabela XI).

Tabela XI
DISTRIBUIÇÃO DOS 50 PARES O-D POR CATEGORIA

Categoria	N	Viável (RAPTOR)
Rota direta (mesma linha)	12	12/12 (100%)
1 transferência	14	14/14 (100%)
2 transferências	10	9/10 (90%)
Pico (07–09h / 17–19h)	6	6/6 (100%)
Baixa frequência (madrugada)	5	1/5 (20%)
Destino sem cobertura direta	3	1/3 (33%)
Total	**50**	**43/50 (86%)**

A categoria *madrugada* (20% de viabilidade) não representa uma limitação do algoritmo, mas uma característica da rede de Barbacena: a maioria das linhas opera entre 06h00 e 22h30. O RAPTOR corretamente informa que não há ônibus — comportamento que o Dijkstra mascarava, retornando uma rota estimada sem verificar horários reais.

Os resultados completos do benchmark estão na Tabela XII.

Tabela XII
DIJKSTRA × RAPTOR — BENCHMARK 50 PARES O-D (RQ5)

Métrica	Dijkstra	RAPTOR
Tempo médio (server-side)	6,1 ms	**1,3 ms**
Tempo p50	6 ms	1 ms
Tempo p95	11 ms	3 ms
Speedup	1×	**4,7×**
Rota viável (% dos 50 pares)	92%	86%
Considera horários reais	Não	**Sim**
Responde "quando chego?"	Não	**Sim**
Retorna rotas Pareto	Não	**Sim**
Transferência viável (verifica)	Não	**Sim**

Nota sobre viabilidade: o RAPTOR encontrou menos rotas (86% vs 92%) porque respeita horários reais — os 7 pares sem rota provavelmente têm ônibus apenas à tarde ou em outros dias. Esse comportamento é **mais correto**, não menos capaz.

O speedup teórico estimado era 2,9× (baseado em operações); o medido foi 4,7×, superando a expectativa graças ao early stopping em redes pequenas como Barbacena (55 rotas, 213 paradas indexadas).

X. DELTA POC 1 → POC 2

Esta seção consolida o delta completo entre as duas provas de conceito.

A. Delta de Métricas Quantitativas

A Tabela XIII apresenta a comparação objetiva dos principais indicadores do sistema antes e depois da POC 2.

Tabela XIII
DELTA QUANTITATIVO POC 1 → POC 2

Métrica	POC 1	POC 2	Δ
LOC totais	22.148	23.882	+1.734
Testes passando	30	141	+111 (+370%)
Code smells	30	0	30 (100%)
TS strict errors	13	0	13
ESLint problems	16	7	9
Bundle CSS (KB)	≈67	36	46%
Algoritmo principal	Dijkstra	RAPTOR	—
Latência algoritmo	6,1 ms	1,3 ms	4,8 ms (4,7×)
Coordenadas paradas	0% (local)	100%	+224 paradas
Features entregues	35	42	+7
Smells por KLOC (back)	2,01	≈0	100%

B. Delta Arquitetural

A POC 2 introduziu as seguintes mudanças estruturais no código:

Backend:

- **Interface `RouteCalculationStrategy`**: desacoplou Dijkstra e habilitou o RAPTOR como segundo motor plugável (S3/Eixo 2);
- **Sete classes RAPTOR** em `service/routing/raptor/` (+1.119 LOC em um commit);
- **`RouteResponseAssembler`**: extraído do `RouteController` God Class (684 → 528 LOC, 23%);
- **`GeoUtils.haversine()`**: centraliza cálculo de distância antes duplicado em 4 classes;
- **Resilience4j**: circuit breakers nos geocodificadores (Google Maps + Nominatim);
- **Endpoint `?engine=raptor|dijkstra`**: seletor de motor com default para RAPTOR.

Frontend:

- **TypeScript strict**: ativado (13 erros → 0);
- **TanStack Query**: de instalado-mas-não-usado para efetivamente utilizado em `useHolidaysAPI`;
- **`ScheduleExpansionContext`**: eliminou prop drilling de três níveis (`Index` → `RouteResults` → `RouteCard`);
- **Seletor de tipo de dia** (F4): substituiu `WEEKDAY` hardcoded por seleção explícita do usuário;
- **Badge de confiança** (F6): `qualityScore` 0–100 exibido com cor semântica;
- **Multi-rota Pareto** (F9): UI exibe todas as opções retornadas pelo RAPTOR.

C. Delta de Metodologia: Personas vs. Subagentes

A POC 1 adotou SDD com **cinco personas de Copilot** (Steering, Planner, Executor, Reviewer, Document AI), todas dentro de sessões de chat, sem phase gates formais. A POC 2 usou **Claude Code com skills + subagentes + hooks**, com phase gates explícitos e métricas por task. A Tabela XIV resume as diferenças.

Tabela XIV
DELTA DE METODOLOGIA SDD: POC 1 VS. POC 2

Aspecto	POC 1	POC 2
Agente IA	Copilot (Sonnet 4.5)	Claude Code (Opus 4.7)
SDD nível	Spec-First	Spec-Anchored
Phase gates	Informais	Explícitos, com pausa
Métricas	Post-hoc	Por task (tempo real)
Paralelismo	Uma persona por vez	Subagentes paralelos
Persistência spec	Parcial (algumas fases)	Todos os eixos
Gestão de smells	Não catalogados	30 catalogados + corrigidos

XI. RESULTADOS CONSOLIDADOS

A Tabela XV compara a produtividade das duas POCs.

Tabela XV
PRODUTIVIDADE POC 1 (CONSTRUÇÃO) × POC 2 (MANUTENÇÃO)

Métrica	POC 1	POC 2
Horas est. sem IA	605 h	≈438 h
Horas reais com IA	30 h	23,4 h
Fator de aceleração	20,2×	≈**21**×
LOC total	22.148	23.882 (+7,5%)
Deliverables	35 feat	30 smells+7 feat+RAPTOR+5 bugs
Testes passando	30/30	141/148
% código por IA	≈100%	≈95%

A semelhança entre os fatores (20,2× e 21×) é notável: a POC 2 manteve o mesmo ritmo de aceleração mesmo em cenário de legado, sugerindo que SDD + IA escalam para manutenção tão bem quanto para construção.

XII. DISCUSSÃO

A. SDD como Ferramenta de Manutenção

O SDD mostrou-se mais valioso durante a manutenção do que durante a construção. Na POC 1 (greenfield), a IA poderia funcionar sem specs formais apenas com prompts conversacionais. Na POC 2 (legado), as specs serviram como:

1) **Mapa de onboarding**: o agente navegou para os arquivos corretos usando `design.md` sem explorar o repositório manualmente;
2) **Âncora de escopo**: phase gates impediram que a IA desviasse para refatorações não planejadas durante as correções de smells;
3) **Fonte de verdade para testes**: critérios de aceitação em `requirements.md` serviram diretamente como casos de teste;
4) **Trilha de auditoria**: cada commit atômico vinculado a uma task da spec torna o processo rastreável.

B. Quando Usar Cada Nível de SDD

Um dos achados transversais da POC 2 é que os três níveis de SDD não competem entre si: são complementares e se aplicam a contextos distintos. A Tabela XVI oferece um guia prático para a escolha do nível adequado.

Tabela XVI
GUIA DE ESCOLHA DO NÍVEL SDD POR CONTEXTO

Nível	Contexto ideal	Vida da spec	Custo/benefício
Spec-First	Tarefa isolada, descartável após entrega (ex.: script de ETL pontual)	Descartada	Baixo custo, zero continuidade
Spec-Anchored	Sistema com ciclo de vida (> 1 sprint), equipe distribuída, manutenção futura esperada	Versionada no repo	Custo médio upfront, alto retorno em manutenção
Spec-as-Source	Domínio altamente formalizado (ex.: DSL de regras de negócio, formulários legais)	É o artefato primário	Alto custo inicial, máxima rastreabilidade

O BQBus se enquadrou inequivocamente no **Spec-Anchored** desde a POC 1: um sistema real com ciclo de vida de meses, múltiplos Eixos interdependentes e a necessidade de onboarding de um agente sem memória de sessão anterior. O "imposto SDD" — o tempo gasto escrevendo e mantendo specs — foi de ≈3 h distribuídas ao longo de toda a POC 2. O retorno foi estimado em ≈20 h economizadas em exploração e retrabalho, uma relação custo-benefício de ≈7:1.

Critérios práticos para escolher Spec-Anchored:

- O sistema terá ≥ 2 sessões de desenvolvimento separadas por > 1 semana;
- Mais de um agente ou desenvolvedor trabalhará no mesmo codebase;
- O sistema tem > 5K LOC ou > 10 componentes interconectados;
- Existe expectativa de manutenção ou evolução futura;
- O custo de regressão é alto (ex.: sistema de transporte, saúde, finanças).

Se **qualquer** dos critérios acima for verdadeiro, o Spec-Anchored paga por si mesmo antes do fim da segunda sessão de desenvolvimento.

C. O "Efeito Memória" do SDD em Agentes de IA

Um mecanismo central — e ainda pouco explorado na literatura — é o que denominamos **efeito memória do SDD**: agentes de IA não têm memória persistente entre sessões. Toda sessão começa com o mesmo contexto vazio. Sem SDD, o agente recomeça do zero: lê o código, infere a arquitetura, descobre as convenções — um processo de ≈10–30 min por sessão, que se repete a cada reinício.

Com SDD (nível Spec-Anchored), o agente carrega `CLAUDE.md` + os `design.md` relevantes e recupera imediatamente:

1) Quais pacotes existem e para que servem;
2) Quais decisões de design já foram tomadas e por quê;
3) Quais tasks já foram concluídas e quais estão pendentes;
4) Quais convenções de código o projeto adota.

Na prática, no BQBus, o tempo de "warm-up" de cada sessão caiu de estimados 15–20 min (sem docs) para < 2 min (com docs SDD). Para uma POC 2 com $\approx$30 sessões de trabalho, isso representa $\approx$5–9 h economizadas apenas em overhead de orientação do agente.

Esse mecanismo explica, em parte, por que o fator de aceleração da POC 2 ($\approx$21$\times$) foi comparável ao da POC 1 ($\approx$20,2$\times$), apesar de o cenário de legado ser intrinsecamente mais complexo que o greenfield. O SDD compensou a complexidade adicional ao eliminar o overhead de orientação.

D. Implicações para a Prática

Os resultados da POC 2 têm implicações diretas para quatro audiências:

Desenvolvedores individuais: a barreira de entrada para o SDD é baixa. O mínimo viável é um `CLAUDE.md` com as convenções do projeto e um `design.md` por feature complexa. Mesmo sem o aparato completo de subagentes e phase gates, esses dois artefatos já eliminam a maior parte do overhead de orientação e reduzem drasticamente os smells de duplicação e dead code.

Times de engenharia: o SDD provê uma interface contratual entre desenvolvedores e agentes de IA. Um time pode revisar `requirements.md` e `design.md` sem ler código — o mesmo fluxo de revisão que times distribuídos já usam para documentos de design (RFCs, ADRs). A diferença é que no SDD esses docs são pré-condição do código, não pós-documentação.

Pesquisadores de MSR (*Mining Software Repositories*): o SDD cria uma trilha de auditoria rara: cada commit está associado a uma task da spec, cada task a um smell ou feature, e cada feature a um critério de aceitação verificável. Isso viabiliza estudos de rastreabilidade bidirecional (*requirements-to-code traceability*) que normalmente exigem ferramentas externas caras.

Fornecedores de ferramentas de IA: o sucesso do SDD no BQBus sugere que ferramentas como Claude Code, Copilot e Cursor se beneficiariam de funcionalidades nativas de gerenciamento de spec — por exemplo, detecção automática de drift entre spec e código, ou alertas quando um commit invalida uma afirmação em `design.md`.

E. Limitações

Acurácia de 54%: a spec não pode ser usada como oráculo sem re-validação. Envelhecimento das specs é proporcional ao ritmo de mudança do código — em projetos com menos debt técnico, a acurácia tende a ser maior.

Estimativas manuais: os fatores de aceleração (21$\times$) dependem de estimativas de tempo sem IA, que podem ser otimistas ou conservadoras.

Escala trivial do RAPTOR: Barbacena tem 31 linhas e 224 paradas. Em redes maiores, MAX_ROUNDS pode precisar de ajuste e o speedup pode variar.

Único desenvolvedor: os resultados refletem um único pesquisador com conhecimento do sistema, o que pode superestimar a facilidade de onboarding para um desenvolvedor externo.

F. Ameaças à Validade

Validade interna: o mesmo desenvolvedor que construiu a POC 1 executou a POC 2, o que pode reduzir o tempo real de ressurreição (conhecimento residual).

Validade externa: o BQBus é um sistema de médio porte (23 K LOC). Os resultados podem não generalizar para sistemas de grande escala ou domínios fortemente regulados.

XIII. Trabalhos Relacionados

Desenvolvimento com IA: estudos de Copilot [4], McKinsey [5] e Stanford [6] relatam aceleração de 35–55% em tarefas isoladas. Nossas métricas (20–21$\times$) são muito superiores, o que é consistente com projetos *greenfield* de escopo bem definido onde a IA escreve a maior parte do código.

Code smells em software gerado por IA: não encontramos na literatura um catálogo quantitativo comparável para projetos de 20 K+ LOC 100% gerados por IA. O trabalho mais próximo é Siddiq et al. [10], que analisa vulnerabilidades em código gerado por Copilot, sem focar em manutenibilidade.

RAPTOR: o algoritmo original de Delling et al. [1] foi validado em redes de grande porte (Europa). Nossa implementação é uma versão simplificada (MAX_ROUNDS=3, sem McRAPTOR), adequada à escala de Barbacena. Os resultados de speedup (4,7$\times$) são coerentes com a literatura para redes pequenas [9].

SDD como método: o artigo de referência [3] propõe os três níveis e os blocos construtivos; nossa POC 2 é, ao nosso conhecimento, a primeira aplicação empírica do nível Spec-Anchored com métricas sistemáticas em um projeto real de manutenção de software gerado por IA.

XIV. Conclusão

Este artigo apresentou a POC 2 do BQBus, um estudo empírico de manutenção e evolução de software legado gerado por IA usando SDD + Claude Code. Os principais achados são:

1) **SDD é um instrumento eficaz para manutenção de código IA**: phase gates evitam drift, métricas por task tornam o processo auditável e as specs sobrevivem ao tempo melhor do que o código que descrevem (54% de acurácia após 90 dias ainda é utilizável como mapa de onboarding);
2) **A IA generativa produz padrões característicos de smells**: duplicação, dead code e mocks em produção correspondem a 37% dos 30 smells identificados — todos consequência da IA não enxergar o sistema como um todo e não fechar ciclos;
3) **IA corrige o que IA gerou com alta eficiência**: 30/30 smells em $\approx$380 min, fator $\approx$14–21$\times$ sobre estimativa manual, zero regressão;
4) **A aceleração se mantém na manutenção**: o fator 21$\times$ é comparável ao da construção (20,2$\times$), sugerindo que SDD + IA escalam para cenários de legado;
5) **RAPTOR supera Dijkstra** em velocidade (4,7$\times$) e semântica (horários reais, Pareto, chegada prevista) sem quebrar o contrato existente da API.

Como trabalho futuro, pretendemos: (1) concluir o deploy (Eixo 5) para validação E2E em produção; (2) medir cobertura JaCoCo antes/depois para enriquecer RQ3; (3) replicar o protocolo em outro projeto legado para verificar generalização.

AGRADECIMENTOS

O autor agradece ao Prof. Marco Túlio Valente pela orientação e ao Departamento de Ciência da Computação da UFMG pelo suporte.

REFERÊNCIAS

[1] D. Delling, T. Pajor, and R. F. Werneck, "Round-Based Public Transit Routing," *SIAM Journal on Computing*, vol. 44, no. 5, pp. 1–24, 2015. (Extended version of SEA 2012.)

[2] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[3] HackTheTask, "Spec-Driven Development com Claude Code," https://hackthetask.com.br/2026/03/20/spec-driven-development-com-claude-code/, Março de 2026. Acesso: Junho 2026.

[4] GitHub and Microsoft, "Research: quantifying GitHub Copilot's impact on developer productivity and happiness," GitHub Blog, September 2022.

[5] McKinsey & Company, "The economic potential of generative AI: the next productivity frontier," McKinsey Digital, June 2023.

[6] S. Noy and W. Zhang, "Experimental evidence on the productivity effects of generative artificial intelligence," *Science*, vol. 381, no. 6654, pp. 187–192, 2023.

[7] M. A. M. Marteleto, "Análise de tempo de desenvolvimento — BQBus POC 1," Documento interno, DCC/UFMG, Janeiro 2026.

[8] M. A. M. Marteleto, "Sistema de Recomendação de Rotas de Ônibus — Proposta de Pesquisa," DCC/UFMG, Outubro 2025.

[9] H. Bast, D. Delling, A. Goldberg, M. Müller–Hannemann, T. Pajor, P. Sanders, D. Wagner, and R. F. Werneck, "Route planning in transportation networks," in *Algorithm Engineering*, LNCS vol. 9220, Springer, 2016, pp. 19–80.

[10] A. S. Siddiq, T. Y. Santos, J. Tanvir, N. Ulfat-Bunyadi, and J. C. Cambronero, "Exploring the security awareness of the Python and JavaScript OpenAI Codex models," in *Proc. MSR*, 2023.

[11] OpenStreetMap Foundation, "Nominatim Geocoding API," https://nominatim.openstreetmap.org/, 2024.

[12] PostGIS Development Group, "Spatial and geographic objects for PostgreSQL," https://postgis.net/, 2024.

[13] ANTP — Associação Nacional de Transportes Públicos, "Sistema de informações da mobilidade urbana," Relatório 2018. São Paulo, 2018.

[14] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[15] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[16] I. Sommerville, *Software Engineering*, 10th ed. Pearson, 2015.

[17] A. Karpathy, "Vibe coding," X (Twitter), February 2025. https://x.com/karpathy/status/1886192184808149180. Acesso: Junho 2026.