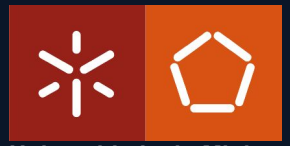


Universidade do Minho
Escola de Engenharia
Depto. de Informática

Gen AI na
Refatorização de Código

Workshop ES3A · Sessão Técnica IV · 03 Jun 2026

Manuel Rodrigues · Tulio Silva · Diogo Ferreira

ALGORITMI/LASI · Universidade do Minho

Lições aprendidas no Projeto AFONSINHO / Ecossistema AMALIA

Refatorização de Código

Introdução & Contexto

Manuel Rodrigues

ALGORITMI/LASI · Universidade do Minho

IA generativa no desenvolvimento de software: onde estamos?

85%

dos developers usam ferramentas de IA regularmente

JetBrains, 2025

https://blog.jetbrains.com/research/2025/10/state-of-developer-ecosystem-2025/

41%

do código global é gerado ou assistido por IA

Stack Overflow, 2025

https://survey.stackoverflow.co/

3.6h

poupadas por semana por developer (em média)

Panto AI, 2026

https://www.getpanto.ai/blog/ai-coding-assistant-statistics

20M+

utilizadores do GitHub Copilot (mid-2025)

GitHub, 2025

https://www.secondtalent.com/resources/ai-coding-assistant-statistics/

O ecosistema de assistentes de codificação cresceu de nicho para infraestrutura padrão em menos de 3 anos.

O que dizem os estudos controlados?

Estudo GitHub × MIT × Princeton × Wharton

55% mais rápido

tarefa idêntica com vs. sem Copilot
(1h11m vs. 2h41m, n=35, p=0.0017)

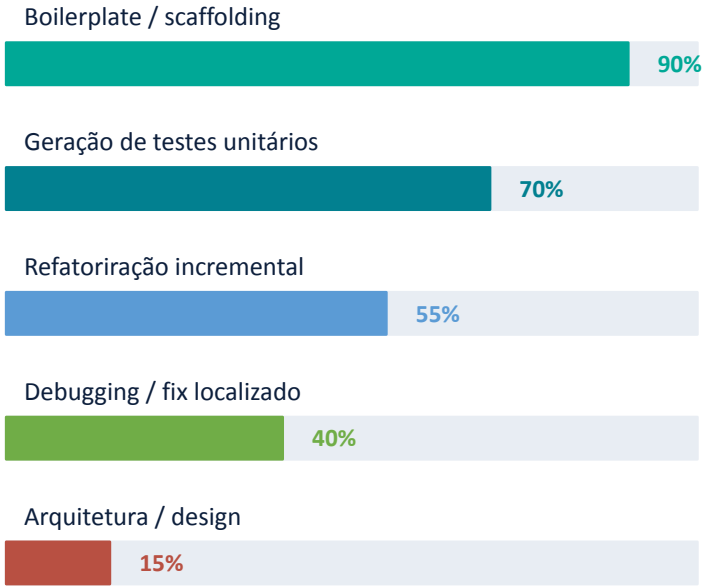
+26% PRs/semana em equipas reais
(MIT, Princeton, UPenn, 2024)

Peng, S., Kalliamvakou, E., Cihon, P., & Demirer, M. (2023). The Impact of AI on Developer Productivity: Evidence from GitHub Copilot. arXiv preprint. → https://arxiv.org/abs/2302.06590 Experiência controlada: developers com acesso ao Copilot completaram uma tarefa de implementação HTTP em JavaScript 55.8% mais rápido do que o grupo de controlo (p=0.0017). arXivarXiv

Cui, Z., Demirer, M., Jaffe, S., Musolff, L., Peng, S., & Salz, T. (2025). The Effects of Generative AI on High-Skilled Work: Evidence from Three Field Experiments with Software Developers. Management Science. DOI: 10.1287/mnsc.2025.00535 → https://ssrn.com/abstract=4945566 Três RCTs com 4.867 developers na Microsoft, Accenture e uma Fortune 100: aumento de 26.08% (SE: 10.3%) nas tarefas completadas. Developers menos experientes mostraram maiores ganhos. BibSonomyMicrosoft

Noy, S., & Zhang, W. (2023). Experimental evidence on the productivity effects of generative artificial intelligence. Science, 381(6654), 187–192. → https://doi.org/10.1126/science.adh2586 Experiência pré-registada com 453 profissionais com formação

Ganho por tipo de tarefa (auto-relato developers)



Mais velocidade ≠ maior entrega de valor

O que os developers relatam

- 55% mais rápido em tarefas isoladas (GitHub/MIT)
- 90% sentiu-se mais produtivo (Accenture, 2024)
- 85% mais confiante no próprio código
- 9 em 10 poupam ≥ 1h/semana (JetBrains, 2025)

O que as organizações medem

- Sem melhoria mensurável em delivery velocity (Faros AI, 10k devs)
- 41% mais churn em código AI-assisted (GitClear, 211M linhas)
- 1.7× mais issues em PRs co-escritos por IA (CodeRabbit, 2025)
- Apenas 16% relata ganho significativo de produtividade (Stack Overflow, 2025)

A velocidade individual só se converte em valor quando CI/CD, code review e testes acompanham o mesmo ritmo. É aqui que entra a engenharia.

METR (2025). *Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity.* arXiv:2507.09089. → https://arxiv.org/abs/2507.09089 RCT com 16 developers experientes (mediana: 5 anos de experiência no projeto), 246 tarefas reais: os developers com acesso a ferramentas de IA (Cursor Pro + Claude 3.5/3.7) demoraram **19% mais tempo** do que sem IA. Os próprios developers estimavam uma redução de 24%. arXiv
Ziegler, A., Kalliamvakou, E., Li, X. A., Rice, A., Rifkin, D., Simister, S., Sittampalam, G., & Aftandilian, E. (2022). *Productivity assessment of neural code completion.* ACM SIGPLAN International Symposium on Machine Programming, pp. 21–29. → Telemetria do Copilot: taxa de aceitação de sugestões como preditor da produtividade percebida.
Google DORA Report (2024). *State of DevOps 2024.* (n=39.000+ profissionais) Cada aumento de 25% na adoção de IA estava associado a uma queda de 1.5% na velocidade de entrega e de 7.2% na estabilidade dos sistemas. InfoWorld

Porquê refatorização? O problema que a IA resolve melhor.

60%

do tempo de dev gasto
a compreender código existente
(não a escrever novo)

55%

das equipas evitam
refatorização por falta
de tempo e confiança

O que a IA faz bem na refatorização

- **Explicar código legado**
  Contexto completo → menos alucinações
- **Sugerir extrações SOLID**
  Isolar responsabilidades em blocos pequenos
- **Gerar testes unitários**
  Âncora antes de qualquer alteração
- **Traduzir entre stacks**
  Python → TypeScript, bloco a bloco
- **Renomear com coerência**
  Nomenclatura consistente em todo o ficheiro

O novo desafio: sistemas neuro-simbólicos

Componente Simbólico

Developer controla

Arquitetura · regras de negócio · contratos de dados · validações determinísticas

Componente Neuronal (LLM)

Probabilístico

Geração de código · refatorização assistida · explicação de lógica legada · sugestões

O Desafio de Engenharia

Gestão da combinação

Previsibilidade · testabilidade · manutibilidade num sistema parcialmente não-determinístico

Gerir esta combinação é o novo desafio de engenharia. A sessão de hoje aborda exactamente isto.

A IDEIA CENTRAL

"Gen AI como copiloto, não como piloto."

O developer define a estratégia e valida cada passo. A IA executa e sugere.

① Estratégia humana

Decidir o que, quando
e como refatorizar

② IA como ferramenta

Sugerir, gerar, explicar
sempre com revisão

③ Validação contínua

Testes + CI/CD + avaliação
como fecho do ciclo

ESTRUTURA DA SESSÃO

Três blocos, três perspetivas

1

Manuel Rodrigues

Introdução & Contexto

Adoção global · Paradoxo da produtividade · Sistemas neuro-simbólicos · Fio condutor da sessão

2

Diogo Ferreira

Princípios & Práticas

SOLID | BAML | CI/CD

SOLID assistido por IA · Ferramentas (Antigravity, BAML) · Ciclo de vida de sistemas com LLMs · Caso prático

3

Tulio Silva

Avaliação & Fine-tuning

ICC | SFT+LoRA | DPO

LLM-as-Judge · Diagnóstico de falhas · Quando (não) fazer fine-tuning · Lições do projeto AFONSINHO

Refatorização de Código

Princípios & Práticas

Diogo Ferreira

ALGORITMI/LASI · Universidade do Minho

Práticas modernas para sistemas que integram LLMs

Agile, DevOps e MLOps não são independentes — num sistema neuro-simbólico, os três se sobrepõem

AGILE

Iterações curtas

Refatorização incremental em sprints — nunca tudo de uma vez

Feedback contínuo

Cada alteração assistida por IA deve ser testada antes da próxima

Adaptação

O modelo de LLM muda; o código que o integra tem de ser desacoplado para acompanhar sem reescritas

DEVOPS

CI/CD

Testes automáticos garantem que Refatorizações não quebram o sistema em produção

Infraestrutura como código

Configurações de LLM (prompts, parâmetros) versionadas e auditáveis

Observabilidade

Logs das chamadas ao LLM — detectar degradação de qualidade em produção

MLOPS

Versionamento de modelos

Qual modelo está em produção? Quando foi actualizado?

Avaliação contínua

LLM-as-Judge com rubrica fixa — mede qualidade antes e após mudanças

Pipeline de afinação

Fine-tuning só com dados da tarefa real; mede impacto com mesma rubrica

Refatorizar ou reescrever? A decisão mais importante

O tamanho, o estado do código e a existência de testes determinam a estratégia

Reescrever do zero

< ~2 000 linhas

- Protótipo fortemente acoplado
- Ausência de testes — nada ancora a refatorização
- Mudança de linguagem ou paradigma

→ Gen AI como tradutor supervisionado

Avaliar caso a caso

~2 000 – 5 000 linhas

- Há módulos estáveis que se podem preservar?
- Existem testes de integração como contrato?
- O custo de reescrita supera o de limpeza?

→ Gen AI mapeia dependências e sugere ordem

refatorização incremental

> ~5 000 linhas

- Múltiplos módulos interdependentes
- Equipa a crescer — legibilidade importa
- Testes existentes como rede de segurança

→ Gen AI como parceiro: explicar, sugerir, testar

Princípios SOLID na refatorização assistida por IA

Gen AI sugere — mas os princípios SOLID são o critério de aceitação das sugestões

S	**Single Responsibility** *Cada módulo faz uma coisa.*	Pede à IA: "Esta função tem mais do que uma responsabilidade? Propõe como dividir."
O	**Open/Closed** *Aberto a extensão, fechado a modificação.*	IA pode gerar novos módulos sem tocar no código existente — desde que a arquitectura o permita.
L	**Liskov Substitution** *Subtipos substituíveis pelos tipos base.*	Ao migrar entre stacks, verifica se os novos componentes respeitam os contratos anteriores.
I	**Interface Segregation** *Interfaces pequenas e específicas.*	BAML e chamadas a LLM devem ter interfaces narrow — facilita testes e substituição de modelo.
D	**Dependency Inversion** *Depende de abstrações, não de implementações.*	O código de negócio não deve depender directamente do modelo. Injector, não instância directa.

Operação de sistemas que integram LLMs: o ciclo de vida

Refatorizar é o meio — operar com qualidade é o objetivo

1 Testes antes de refatorizar

Gera testes unitários com Gen AI a partir do código existente. São o contrato do comportamento actual — sem eles, não sabes o que estás a preservar.

2 Refatorização incremental verificada

Pequenas alterações, teste a cada passo. Gen AI sugere uma de cada vez; CI/CD confirma que o sistema continua de pé antes do passo seguinte.

3 Versionamento e observabilidade

Prompts, configurações de LLM e schemas BAML versionados como código. Logs das chamadas ao modelo em produção para detectar degradação.

4 Avaliação contínua pós-deploy

LLM-as-Judge com rubrica fixa mede qualidade das saídas em produção. Alertas quando métricas descem abaixo do threshold definido.

Ferramentas: Antigravity e BAML

A escolha de ferramentas condiciona como a IA participa — e como o sistema é testável e mantível

Antigravity — IDE com IA integrada

Contexto completo
Lê o ficheiro inteiro — não só o que copias no prompt. Menos alucinações; sugestões mais alinhadas com o código real.

Sugestões em linha
A IA sugere no próprio editor; o developer aceita ou rejeita linha a linha — revisão integrada no fluxo.

Agentes especializados
SYSTEM PROMPTS com as regras de arquitectura do projecto — a IA segue padrões SOLID sem que precises de os repetir.

BAML — Boundary AI Markup Language

Interface simbólica para o LLM
Define tipos, prompts e parsers numa linguagem declarativa — o LLM devolve saídas estruturadas, não texto livre.

Dependency Inversion na prática
O código de negócio depende do schema BAML, não do modelo. Trocar de LLM não requer reescrever lógica.

Versionável e testável
Schemas e prompts versionados como código — alterações auditáveis, testáveis em CI/CD como qualquer outro componente.

Antes de refatorizar código que integra LLMs:

Um checklist prático — Agile · DevOps · MLOps · SOLID

!	**O sistema tem >2 000–3 000 linhas?**	Abaixo disso, reescrita do zero (com IA como tradutor supervisionado) é provavelmente mais eficiente.
✓	**Tens testes que confirmem o comportamento actual?**	Gera-os com Gen AI antes de começar. São o contrato — sem eles não sabes o que preservaste.
✓	**Aplicaste o princípio SOLID mais violado primeiro?**	Identifica a violação mais crítica (tipicamente SRP) e refatoriza esse módulo com IA. Verifica. Avança.
✓	**Tens CI/CD a correr testes a cada iteração?**	Refatorização incremental sem feedback automático é fazer na escuridão. DevOps é parte do processo.
!	**Os prompts e schemas BAML estão versionados como código?**	Alterações ao LLM são alterações ao sistema. Têm de ser auditáveis, reversíveis e testáveis.
✓	**Tens uma rubrica de avaliação da qualidade em produção?**	MLOps: LLM-as-Judge com métricas fixas. Sem monitorização contínua, não detectas degradação.

Refatorização de Código

Avaliação & Fine-Tuning

Tulio Silva

ALGORITMI/LASI · Universidade do Minho

O problema: integrar um LLM num sistema real é mais do que fazer uma chamada à API

Quando escolhes um LLM para o teu sistema, precisas responder:

- ? Este modelo é suficientemente bom para esta tarefa específica?
- ? Como medir "suficientemente bom" de forma sistemática?
- ? O modelo vai falhar de formas que não antecipei?
- ? Vale a pena fazer fine-tuning — e quais os riscos?
- ? Como garantir que o comportamento se mantém em produção?

Nesta sessão

1. Escolher e validar um juiz automático
2. Avaliar modelos candidatos
3. Diagnosticar o que falha e porquê
4. Decidir (ou não) fazer fine-tuning
5. Medir o impacto de forma rigorosa

Passo 1: como medir qualidade de saída — o LLM-as-a-Judge

O que é?

Usas um modelo forte (ex. GPT-4o, Claude) como avaliador automático das saídas do teu modelo-alvo, com base numa rubrica definida por ti.

Por que funciona?

Zheng et al. (NeurIPS 2023) mostraram que um juiz LLM forte atinge >80% de concordância com humanos — o mesmo nível de acordo entre dois humanos.

Quando usar?

Sempre que teres saída de texto livre e não conseguires avaliar com métricas determinísticas (F1, BLEU são insuficientes para qualidade real).

Atenção: consistência ≠ correção. Um juiz pode ser consistentemente errado. É preciso validar antes de confiar.

Passo 2: validar o juiz antes de o usar em produção

Protocolo: executar o mesmo prompt N vezes (temperatura=0) e medir consistência

>0.95	>0.85	>50%	0%
ICC(3,k) consistência excelente	Similaridade semântica justificativas	Instâncias com desvio-padrão = 0	Taxa de falha de infra

O que verificar:

Estabilidade numérica	O juiz dá a mesma nota para o mesmo output em runs independentes?
Coerência das justificativas	As explicações são semanticamente consistentes entre execuções?
Alinhamento humano	A validação automática não garante alinhamento com julgamento humano — isso é um passo adicional.

⚠️ *Evitar interfaces de chat para avaliação: sem controlo de temperatura → variância elevada → resultados não reprodutíveis. Usar API com temperatura fixa.*

Passo 3: avaliar os modelos candidatos — não confiar em benchmarks genéricos

O que tende a acontecer:

Modelos não são iguais em todas as tarefas
Um modelo top em texto formal pode ser medíocre em edição controlada.

Benchmarks genéricos enganam
MMLU, HellaSwag etc. não refletem o teu caso de uso real.

Escala de parâmetros ≠ qualidade na tarefa
Modelos pequenos bem afinados superam modelos grandes genéricos.

Meta-comentários são um sinal de alerta
O modelo explica o que vai fazer em vez de fazer — indica falta de instruction tuning.

Passo 4: diagnosticar — um modelo pode ser bom em A e mau em B pelo mesmo motivo

A causa mais comum: o modelo foi treinado para gerar, não para verificar/editar

Tarefas de GERAÇÃO

Modelo tipicamente treinado para isto

- Sumarização
- Resposta a perguntas
- Geração de código
- Tradução

Tarefas de EDIÇÃO/VERIFICAÇÃO

Requer fine-tuning específico

- Correção localizada
- Code review
- Validação de formato
- Deteção de erros

SINAIS DE ALERTA

O modelo não foi treinado para editar

- Explica em vez de corrigir
- Reformula tudo em vez de editar
- Ignora tokens de controlo
- Inventa correções desnecessárias

Passo 5: a decisão de fine-tuning — quando compensa e quando não

Antes de investir em fine-tuning, pergunta-te:

1. **Prompt engineering esgotado?**

   Testa com chain-of-thought, few-shot, system prompts detalhados. FT é caro — prompts são grátis.

2. **Tens dados suficientes e corretos?**

   Sem dados da tarefa real, o modelo aprende a forma, não o conteúdo. Centenas de exemplos genéricos ≠ dados dirigidos.

3. **Tens infraestrutura de MLOps?**

   FT sem pipeline de avaliação é fazer na escuridão. Precisas medir antes e depois com a mesma rubrica.

4. **O ganho justifica o custo?**

   FT muda o comportamento global. Pode melhorar A e degradar B (catastrophic forgetting).

Se decidires avançar com fine-tuning: abordagem e riscos

Abordagem recomendada

1 **Dados sintéticos dirigidos**
Usa um modelo forte para gerar exemplos do comportamento que queres — não exemplos genéricos

2 **SFT + LoRA**
Parameter-efficient fine-tuning: só treinas adaptadores (1-5% dos pesos). Preserva comportamento base.

3 **DPO/RLHF (opcional)**
Para alinhar por preferência: resposta curta+correta > resposta longa+comentada

4 **Avaliação pós-FT com mesma rubrica**
Obrigatório comparar antes/depois com o mesmo juiz e o mesmo conjunto de testes

Riscos a monitorizar

Catastrophic forgetting
FT em tarefa A pode degradar tarefa B

Overfitting ao estilo
Com poucos dados, aprende formatação, não competência

Deriva de comportamento
Respostas mudam de formas inesperadas fora do domínio de treino

Custo sem retorno
Sem dados corretos, o comportamento não muda de forma robusta

Lições aprendidas: o que vimos na prática

Caso AFONSINHO / Ecossistema AMALIA — modelos ≤10B em português europeu

Especialização temática ≠ capacidade operacional

Um modelo com score alto em texto jurídico formal pode falhar sistematicamente em edição localizada na área da pedagogia. A tarefa importa mais do que o domínio.

Fine-tuning sem dados dirigidos não funciona

Com exemplos genéricos, o modelo aprendeu a forma superficial mas não reconstruiu a competência de verificação. Dados da tarefa real são essenciais.

O juiz LLM fornece uma medida ≠ alinhamento

ICC(3,k) > 0.95 confirma que o juiz é estável. Sem isso, não teríamos conseguido medir o impacto de nenhuma mudança.

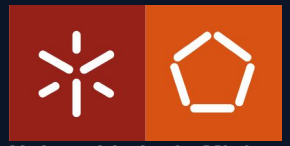
Meta-comentários são um bug, não uma feature

Quando o modelo explica o que faz em vez de fazer, é um sinal de que o instruction tuning não cobre a tua tarefa. Isso resolve-se com dados, não com prompts maiores.

Antes de integrar um LLM no teu sistema:

Um checklist prático

✓	**Avalia na tua tarefa real,**	não em benchmarks genéricos. Cria o teu próprio dataset de teste.
✓	**Valida o juiz antes de usá-lo**	— ICC e semântica das justificativas. Uma régua consistente pode ser consistentemente errada.
!	**Testa instruction-following**	— o modelo faz o que peço, ou comenta o que vai fazer?
!	**Não faças fine-tuning sem dados**	da tarefa real e sem pipeline de avaliação antes/depois.
✓	**MLOps fecha o ciclo:**	avaliar → diagnosticar → afinar → reavaliar com a mesma rubrica.
✓	**Monitoriza catastrophic forgetting**	— FT em A pode degradar B. Mede tudo.



Universidade do Minho
Escola de Engenharia
Depto. de Informática

Gen AI na
Refatorização de Código

Workshop ES3A · Sessão Técnica IV · 03 Jun 2026

Manuel Rodrigues · Tulio Silva · Diogo Ferreira

ALGORITMI/LASI · Universidade do Minho

Lições aprendidas no Projeto AFONSINHO / Ecossistema AMALIA