Pontifícia Universidade Católica
do Rio de Janeiro

Henrique Antonio de Leão Peres

A Multi-Agent Architecture for Automated Code Refactoring Using LLM-Driven Analysis and Verification

Projeto Final de Programação

Projeto Final de Programação, apresentado ao programa a matéria INF2102 da Computação da PUC-Rio como entrega da documentação do projeto.

Orientador: Prof. Alessandro Garcia

Rio de Janeiro
dezembro de 2025

Sumário

1
Introduction

Modern software systems evolve continuously, and maintaining code quality is one of the most time-consuming, error-prone, and intellectually demanding activities in software engineering. As codebases grow, developers must repeatedly apply refactorings, such as renaming entities, extracting functions, reorganising logic, or reducing complexity, to preserve readability, reduce technical debt, and avoid architectural degradation. Although refactoring is essential to long-term maintainability, it is rarely automated safely due to the inherent risks of altering behavior, the need for contextual understanding, and the difficulty of evaluating the correctness of proposed changes.

Recent advances in Large Language Models (LLMs) introduced a new paradigm for code generation, summarization, and transformation. LLMs are capable of producing syntactically correct and stylistically coherent code, and they can reason about structural improvements in ways that resemble human intuition. However, LLM-based refactoring remains unreliable when executed as a single, monolithic step. Without proper safety precautions, LLMs may introduce regressions, modify unrelated code, hallucinate changes, or ignore constraints that ensure behavioral preservation.

To address these limitations, this project proposes a distributed multi-agent architecture designed to orchestrate, supervise, and validate LLM-driven refactoring in a reliable and auditable manner. Instead of relying on a single LLM call, the system divides responsibilities into specialized agents, Planning, Static Analysis, Proposal Generation, Review, and Execution. Each agent focuses on a well-defined task and communicates through message-passing, enabling layered reasoning, explicit verification, and controlled application of changes.

By combining program analysis, incremental decision-making, and LLM-based reasoning, the proposed architecture aims to deliver safer, explainable, and context-aware automated refactorings. The system integrates traditional static metrics (such as cyclomatic complexity, function count and etc) with LLM generated suggestions that are subsequently validated by a rule-based Reviewer before any modification is applied to the codebase. This multi-stage pipeline reduces risk, increases transparency, and allows developers to analyze

the rationale behind each refactoring.

This work contributes (i) a modular architecture for autonomous refactoring using LLMs; (ii) an execution pipeline that provides traceability and auditability of each decision; and (iii) an empirical evaluation using multiple code examples of varying complexity to assess the system's behavior, refactoring quality, and safety mechanisms. The approach demonstrates how multi-agent reasoning can significantly improve trust and reliability in automated refactoring workflows, paving the way for future systems that integrate static analysis, program synthesis, and intelligent code transformation.

2
Problem and Motivation

Software systems evolve continuously, often accumulating technical debt as new features, bug fixes, and architectural modifications are introduced over time. To preserve long-term maintainability, developers must routinely apply refactorings such as renaming identifiers, extracting functions, reorganizing control flow, or improving modularity. Although these transformations do not change the observable behavior of a program, they require a deep understanding of both the local and global context of the codebase. As a consequence, refactoring remains a labor-intensive and error-prone activity, particularly in projects with large and diverse codebases.

2.1
Limitations of Traditional Refactoring Tools

Existing automated refactoring tools, such as those integrated into IDEs, are highly reliable for simple, well defined transformations (renaming or extract method). However, they generally rely on static patterns and rigid rules. These tools struggle with more complex scenarios that require semantic interpretation, understanding of architectural intent, or reasoning about code smells and design quality. As a result, developers cannot rely solely on traditional tools when dealing with nuanced refactorings or legacy systems with irregular coding patterns.

2.2
Challenges of LLM-Based Refactoring

The emergence of Large Language Models (LLMs) introduced new possibilities for automated code transformation. LLMs are capable of interpreting code, identifying design issues, and proposing improvements that resemble human reasoning. Despite these strengths, LLM driven refactoring presents several critical challenges:

- **Lack of behavioral guarantees:** LLMs may introduce subtle regressions, modify unrelated code segments, or hallucinate structures that do not exist in the original program.

- **Insufficient control and explainability:** A single-step LLM invocation does not provide traceability or justification for the changes it produces, making validation difficult.
- **Difficulty handling code of varying complexity:** LLMs may fail to reason consistently across files that differ widely in size, structure, or cyclomatic complexity.
- **Absence of integrated safety mechanisms:** Without external verification layers, LLM proposals cannot be automatically trusted in production environments.

These limitations make direct LLM-based refactoring unsuitable for any scenario requiring reliability, auditability, and behavioral preservation.

2.3 Motivation for a Multi-Agent Architecture

To overcome the shortcomings of both traditional refactoring tools and monolithic LLM-based approaches, this project adopts a *multi-agent architecture* in which each component specializes in a narrowly defined task. This decomposition provides several advantages:

- **Separation of responsibilities:** Planning, static analysis, proposal generation, validation, and execution are handled independently, reducing the likelihood of cascading errors.
- **Layered verification:** Refactorings are examined by multiple agents, enabling checks based on structural metrics, semantic patterns, and risk-aware policies.
- **Traceability and auditability:** Each agent produces explicit logs and structured output, allowing developers to understand and review the reasoning behind decisions.
- **Increased reliability:** The Reviewer agent acts as a safety gatekeeper, ensuring that only valid, low-risk, and appropriately sized changes reach the codebase.

This architecture supports a more trustworthy and explainable process for automated refactoring, combining the strengths of traditional static analysis with the generative capabilities of modern LLMs. The resulting system provides a structured pathway to incorporate AI-generated code transformations in real software engineering workflows.

3
Architecture and Methodology

This section presents the software architecture and methodological foundations of the proposed multi-agent system for automated refactoring. The system is designed around the principle of *specialized cooperative agents*, where each component is responsible for a narrow and clearly defined task. Rather than relying on a single Large Language Model (LLM) invocation, the architecture decomposes refactoring into a structured pipeline consisting of: Planning, Static Analysis, Proposal Generation, Review, and Execution. This division ensures reliability, transparency, and incremental validation throughout the refactoring process.

The agents communicate asynchronously through a message queue and operate as decoupled processes. Each agent produces explicit logs and structured payloads, enabling a fully auditable refactoring workflow.

3.1
Overview of the Multi-Agent Pipeline

The complete system is composed of five independent agents:

1. **Planner** – selects the target file and constructs the task context.
2. **Static Analyzer** – computes structural metrics and identifies hotspots.
3. **Proposer** – invokes an LLM to generate a refactoring proposal using contextual information.
4. **Reviewer** – evaluates the proposal using rule-based policies and structural constraints.
5. **Executor** – applies the accepted diff to a sandbox Git repository.

The agents communicate exclusively through JSON messages that encapsulate the *intent*, *payload*, and *context* of each task. This design promotes modularity, scalability, and fault isolation: a failure in one agent does not compromise the entire system.

Figura 3.1: Message-passing pipeline between agents.

3.2 Message Structure and Communication Model

All agents exchange messages through a lightweight message queue. Each message contains:

- **intent**: the role of the message (REQUEST, ANALYZE, PROPOSE, ACCEPT, REJECT),
- **task_id**: a unique identifier ensuring traceability,
- **context**: metadata describing the repository, target file, and execution constraints,
- **payload**: agent-specific data, such as metrics, diffs, or decisions.

This explicit structure provides full transparency, allowing downstream agents to understand both the origin and justification of each operation.

3.3 Planner Agent

The Planner is responsible for initiating the refactoring workflow. Its primary task is to construct the *execution context* for a given refactoring task. This includes identifying the target file, setting the repository location, and defining the parameters for subsequent agents.

Although the Planner does not analyze the code directly, it performs important organizational functions:

- selects which source file will undergo refactoring,
- constructs the refactoring request message,
- dispatches the request to the Static Analyzer,
- defines high-level goals for the LLM (“improve readability”, “reduce duplication”).

The Planner’s design ensures that each task is reproducible, explicit, and auditable.

3.4 Static Analyzer Agent

The Static Analyzer performs a structural examination of the target file. It does *not* rely on the LLM, instead, it uses deterministic program analysis tools (AST parsing, TypeScript compiler APIs) to compute well-defined metrics:

- total lines of code (LOC),
- number of functions,
- cyclomatic complexity (per function and average),
- maximum observed complexity,
- identification of hotspots,
- ESLint-derived warnings and rule violations (when available).

The Analyzer produces a **problem profile** summarizing the structural issues in the code. This profile is appended to the context forwarded to the Proposer.

This separation is critical: the LLM receives curated, structured information instead of raw code, enabling more grounded reasoning and reducing hallucination.

3.5 Proposer Agent

The Proposer invokes the LLM to generate a refactoring proposal. It uses both the source code and the structured metrics produced by the Static Analyzer as inputs to a custom prompt designed for reliability and constraint awareness.

The LLM is asked to provide:

- a concise description of the problem,
- a high-level explanation of the proposed refactoring,
- an estimation of refactoring risk (low, medium, high),
- a unified diff representing the proposed changes.

3.5.1 LLM Prompt Specification

The Proposer agent relies on a structured refactoring prompt provided to the LLM. It defines strict rules for preserving behavior, avoiding domain invention, and producing clean unified diffs.

Full LLM Prompt

```
'''Voce e um engenheiro de software especialista em refatoracao.
Seu objetivo    melhorar a qualidade interna do c digo mantendo o mesmo comportamento observ vel.

REGRAS GERAIS:

1. Preserve o DOM NIO:
   - Nao invente conceitos que nao existam no codigo original.
   - Se o codigo lida apenas com numeros, n o crie entidades como "User", "Order", etc.
   - Use apenas abstracoes coerentes com o que j  est  presente.

2. Comportamento:
   - A refatora  o deve produzir exatamente o mesmo resultado para as mesmas entradas.
   - N o adicione validacoes, logs, exce  es ou efeitos colaterais novos.

3. Refatora  o ideal:
   - Voce PODE e DEVE:
     * Renomear variaveis, parametros e fun  es internas para deixar mais claro.
     * Extrair funcoes ou metodos privados para reduzir duplica  o ou complexidade.
     * Introduzir classes/objetos internos se isso melhorar o design.
     * Reorganizar o codigo para melhorar legibilidade e manutenibilidade.
   - Mantenha a refatora  o focada no que traz ganho real de clareza/qualidade.
   - Evite mudan as cosm ticas excessivas sem ganho de clareza.
```

```
23
24 4. Formato da resposta:
25    - Responda APENAS com um diff unificado que possa ser
        aplicado com `git apply`.
26    - Use cabe alhos no formato:
27      --- a/${filePath}
28      +++ b/${filePath}
29    - N o escreva nenhuma explica  o em texto fora do
        diff.'''
```

The Proposer enforces several guardrails:

- diffs must be syntactically valid and follow unified diff conventions,
- only the target file may be modified,
- proposals must preserve original behavior.

This produces a structured refactoring proposal that is forwarded to the Reviewer.

3.6 Reviewer Agent

The Reviewer is a rule-based gatekeeper designed to prevent unsafe or excessive changes. It evaluates the LLM-generated proposal using:

- the number of modified lines inferred from the diff,
- the risk level provided by the Proposer,
- the original file metrics recomputed by the Analyzer,
- adaptive thresholds based on file complexity.

Files are classified as low, medium, or high-severity based on their structure. The Reviewer accepts or rejects proposals using the following logic:

- low-severity files: small, low-risk diffs are accepted,
- medium-severity files: moderate changes are tolerated,
- high-severity files: larger diffs may be accepted but high-risk proposals are rejected,
- any proposal above the maximum allowed threshold is rejected.

This multi-factor evaluation ensures that only valid, explainable, and safe refactorings proceed.

3.7 Executor Agent

The Executor applies accepted refactorings to a sandbox Git repository. It performs the following steps:

1. checks out a clean branch for the task,
2. applies the diff using a programmatic patching tool,
3. verifies that the patch applies cleanly,
4. commits the changes with a detailed message,
5. optionally pushes the result to a remote repository.

By isolating refactoring execution in a controlled environment, the system ensures reproducibility and avoids corrupting the original codebase.

3.8 Methodological Principles

The system's methodology integrates both deterministic analysis and probabilistic reasoning. Its key principles are:

- **Separation of concerns**: each agent handles exactly one type of reasoning.
- **Incremental verification**: refactorings must pass multiple checkpoints.
- **Auditability**: every decision is logged and reproducible.
- **Behavior preservation**: refactorings must not alter program semantics.
- **LLM constraint framing**: the LLM receives structured signals to reduce hallucination.

Together, these principles ensure that automated refactoring operates safely, transparently, and effectively.

4
Experimental Procedure

This section describes the methodology used to evaluate the proposed multi-agent refactoring system. The objective of the experimental procedure is to determine whether the architecture behaves reliably across different levels of code complexity and whether each agent performs its role consistently within the refactoring pipeline.

4.1
Experimental Goals

The experiments were designed to assess the following aspects of the system:

- the correctness and stability of the message-passing pipeline,
- the ability of the Static Analyzer to compute consistent structural metrics,
- the quality and relevance of LLM-generated refactoring proposals,
- the effectiveness of the Reviewer in applying safety constraints,
- the reliability of the Executor when applying accepted diffs,
- the system's behavior under varying code sizes and complexity levels.

4.2
Selection of Test Files

To evaluate the pipeline across heterogeneous conditions, five TypeScript files were selected. These files were constructed to represent a diverse set of software characteristics commonly found in real applications:

1. **Simple arithmetic utilities:** very small functions used to test minimal refactoring behavior.

```ts
export function sum(a: number, b: number) {
  let result = a + b;
  return result;
}

export function average(a: number, b: number) {
  const s = sum(a, b);
  const qtd = 2;
  return s / qtd;
}

```

Figure 4.2.1: First code used (simple arithmetic utilities)

2. **User scoring service:** medium-sized file with duplication, opportunities for extraction, and multiple conditional branches.

```ts
export interface User {
  id: string;
  name: string;
  age: number;
  purchases: number[];
  lastLoginAt?: Date;
}

export function calcScore(u: User): number {
  let score = 0;

  // idade
  if (u.age < 18) {
    score = score + 5;
  } else if (u.age >= 18 && u.age < 30) {
    score = score + 10;
  } else if (u.age >= 30 && u.age < 50) {
    score = score + 7;
  } else {
    score = score + 3;
  }

  // compras
  let pTotal = 0;
  for (let i = 0; i < u.purchases.length; i++) {
    pTotal = pTotal + u.purchases[i];
  }

  if (pTotal > 1000) {
    score = score + 20;
  } else if (pTotal > 500) {
    score = score + 10;
  } else if (pTotal > 100) {
    score = score + 5;
  }

  // login
  if (u.lastLoginAt) {
    const diffMs = Date.now() - u.lastLoginAt.getTime();
    const diffDays = diffMs / 1000 / 60 / 60 / 24;
    if (diffDays < 7) {
      score = score + 10;
    } else if (diffDays < 30) {
      score = score + 5;
    } else {
      score = score - 5;
    }
  } else {
    score = score - 10;
  }

  return score;
}

export function calcScoreForList(users: User[]): number[] {
  const arr: number[] = [];
  for (let i = 0; i < users.length; i++) {
    const u = users[i];
    const s = calcScore(u);
    arr.push(s);
  }
  return arr;
}

```

Figure 4.2.2: Second code used (user scoring service)

3. **Reporting service:** medium complexity with repeated logic, enabling analysis of structural patterns.

```ts
export interface Sale {
  id: string;
  value: number;
  createdAt: Date;
  status: "OPEN" | "PAID" | "CANCELLED";
}

export interface Report {
  total: number;
  average: number;
  count: number;
}

function calculateSummary(values: number[]): Report {
  if (values.length === 0) {
    return {
      total: 0,
      average: 0,
      count: 0,
    };
  }

  let total = 0;
  for (let i = 0; i < values.length; i++) {
    total = total + values[i];
  }

  return {
    total,
    average: total / values.length,
    count: values.length,
  };
}

export function buildMonthlyReport(sales: Sale[], month: number, year: number): Report {
  const filtered: number[] = [];
  for (let i = 0; i < sales.length; i++) {
    const s = sales[i];
    if (s.createdAt.getMonth() === month && s.createdAt.getFullYear() === year) {
      if (s.status === "PAID") {
        filtered.push(s.value);
      } else if (s.status === "OPEN") {
        // vendeu mas não pagou, conta metade
        filtered.push(s.value * 0.5);
      } else {
        // CANCELLED não entra
      }
    }
  }

  return calculateSummary(filtered);
}

export function buildYearlyReport(sales: Sale[], year: number): Report {
  const filtered: number[] = [];
  for (let i = 0; i < sales.length; i++) {
    const s = sales[i];
    if (s.createdAt.getFullYear() === year) {
      if (s.status === "PAID") {
        filtered.push(s.value);
      } else if (s.status === "OPEN") {
        filtered.push(s.value * 0.5);
      } else {
        // CANCELLED não entra
      }
    }
  }

  return calculateSummary(filtered);
}

```

Figure 4.2.3: Third code used (Reporting service)

4. **Feature-flag evaluation module:** larger file containing nested conditionals and deeper control flow, intended to stress-test cyclomatic complexity analysis.

```typescript
export interface FeatureContext {
  userRole: "ADMIN" | "MANAGER" | "EMPLOYEE" | "GUEST";
  country: string;
  isBetaUser: boolean;
  accountAgeDays: number;
  lastPaymentFailed: boolean;
}

export function isFeatureEnabled(
  featureName: string,
  ctx: FeatureContext
): boolean {
  if (featureName === "DASHBOARD_V2") {
    if (ctx.userRole === "ADMIN" || ctx.userRole === "MANAGER") {
      if (ctx.country === "BR" || ctx.country === "US") {
        return true;
      } else {
        if (ctx.isBetaUser) {
          return true;
        } else {
          return false;
        }
      }
    } else {
      return false;
    }
  } else if (featureName === "CHECKOUT_EXPERIMENT") {
    if (ctx.lastPaymentFailed) {
      return false;
    } else {
      if (ctx.accountAgeDays > 365) {
        return true;
      } else if (ctx.accountAgeDays > 180) {
        if (ctx.isBetaUser) {
          return true;
        } else {
          return false;
        }
      } else {
        return ctx.isBetaUser;
      }
    }
  } else if (featureName === "BETA_CHAT") {
    if (ctx.userRole === "ADMIN") {
      return true;
    }
    if (ctx.userRole === "MANAGER") {
      if (ctx.isBetaUser) {
        return true;
      } else {
        return false;
      }
    }
    if (ctx.userRole === "EMPLOYEE") {
      if (ctx.country === "BR") {
        return ctx.isBetaUser;
      } else {
        return false;
      }
    }
    return false;
  } else {
    return false;
  }
}
```

Figure 4.2.4: Fourth code used (Feature-flag evaluation module)

5. **Legacy export generator:** a long, cohesive function with multiple responsibilities, designed to evaluate behavior in high-severity contexts.

```ts
export interface ExportRow {
  id: string;
  name: string;
  email?: string;
  active: boolean;
  createdAt: Date;
  lastLoginAt?: Date;
  totalPurchases: number;
}

export function exportCsv(rows: ExportRow[]): string {
  // função intencionalmente "feia" para testar refatoração em arquivo mais complexo

  let csv = "id;name;email;status;created_at;last_login;score\n";

  for (let i = 0; i < rows.length; i++) {
    const r = rows[i];

    // status
    let status = "";
    if (r.active && r.totalPurchases > 0) {
      status = "ACTIVE";
    } else if (r.active && r.totalPurchases === 0) {
      status = "NEW";
    } else if (!r.active && r.totalPurchases > 0) {
      status = "INACTIVE_WITH_HISTORY";
    } else {
      status = "INACTIVE";
    }

    // score
    let score = 0;
    if (r.totalPurchases > 1000) {
      score = 10;
    } else if (r.totalPurchases > 500) {
      score = 7;
    } else if (r.totalPurchases > 100) {
      score = 4;
    } else if (r.totalPurchases > 0) {
      score = 2;
    }

    if (r.lastLoginAt) {
      const diffMs = Date.now() - r.lastLoginAt.getTime();
      const diffDays = diffMs / 1000 / 60 / 60 / 24;
      if (diffDays < 7) {
        score = score + 3;
      } else if (diffDays < 30) {
        score = score + 2;
      } else if (diffDays < 180) {
        score = score + 1;
      } else {
        score = score - 1;
      }
    }

    // email format
    let email = r.email ?? "";
    if (email.indexOf("@") === -1) {
      email = "";
    }

    const created = r.createdAt.toISOString();
    const lastLogin = r.lastLoginAt ? r.lastLoginAt.toISOString() : "";

    csv +=
      r.id +
      ";" +
      r.name +
      ";" +
      email +
      ";" +
      status +
      ";" +
      created +
      ";" +
      lastLogin +
      ";" +
      score +
      "\n";
  }

  return csv;
}
```

Figure 4.2.5: Fifth code used (Legacy export generator)

These files cover a broad spectrum of LOC, cyclomatic complexity, and semantic structures, allowing the evaluation to capture representative refactoring scenarios.

4.3 Execution Environment

All agents were executed on a Windows 11 machine using Node.js 20.18.0, TypeScript, Redis for message queueing, and a local Git repository used by the Executor. The entire multi-agent system was additionally containerized using Docker, which ensured reproducibility and isolated runtime environments for each component. The Static Analyzer, Proposer, Reviewer, and Executor could therefore be executed either directly via Node.js or through their respective Docker containers without altering system behavior.

The LLM used for proposal generation was the `gpt-5.1` model accessed via the official OpenAI API.

Each agent operated as an independent process and communicated exclusively through the Redis-backed message queue. Logging and traceability were enabled for every run to capture the full refactoring lifecycle.

4.4 Procedure

For each of the five test files, the following procedure was applied:

1. The Planner agent was initialized with the file path and repository configuration.
2. The Static Analyzer computed structural metrics and generated a problem profile.
3. The Proposer invoked the LLM and constructed a refactoring proposal containing (i) an explanation, (ii) a risk estimate, and (iii) a unified diff.
4. The Reviewer evaluated the proposal using rule-based policies that incorporate file severity, cyclomatic complexity, and the number of modified lines.
5. If accepted, the Executor applied the diff in a clean sandbox branch and committed the result.
6. All logs, metrics, messages, and proposal artifacts were collected for later analysis.

For consistency, the same experimental protocol was applied to all five test files. No manual intervention was performed while the agents executed their tasks.

4.5 Collected Data

The following data were gathered for each run:

- structural metrics: LOC, function count, cyclomatic complexity,
- hotspots identified by the Static Analyzer,
- risk level and explanation provided by the Proposer,
- number of modified lines in the diff,
- Reviewer decision (accept or reject) and justification,
- Executor logs confirming successful or failed patch application.

This dataset provides the foundation for the analysis presented in Section 5, where the behavior of the pipeline is examined in detail.

5
Results

This section presents the outcomes produced by the multi-agent refactoring system for each of the five test files. For each case, we include (i) the original code snippet, (ii) the metrics generated by the Static Analyzer, (iii) the refactoring proposal generated by the Proposer, (iv) the Reviewer decision, and (v) the Executor result. No interpretation or discussion is provided here; deeper analysis is deferred to Section 6.

5.1
Test 1: Simple Arithmetic Utilities

The first experiment evaluated the system on a small arithmetic utility file containing very simple computational logic. This scenario serves as a baseline to observe how the pipeline behaves in cases requiring minimal transformation.

5.1.1
Planner Output

```
C:\workspace\PFP\refactor-agents>pnpm run dev:planner

> refactor-agents@ dev:planner C:\workspace\PFP\refactor-agents
> ts-node --transpile-only packages/agents/planner/index.ts

[2025-12-08T12:28:48.410Z][planner] creating new task
{
  "task_id": "refac-1765196928409",
  "context": {
    "repo": "https://github.com/HenriquePeres/refactor-agents-sandbox",
    "branch": "main",
    "language": "TypeScript",
    "targets": [
      "../refactor-agents-sandbox/src/app/services/math-utils-simple.ts"
    ]
  }
}
[2025-12-08T12:28:48.422Z][queue] enqueued to static-analyzer
{
  "task_id": "refac-1765196928409",
  "intent": "REQUEST"
}
[2025-12-08T12:28:48.422Z][planner] enqueued REQUEST to static-analyzer
{
  "task_id": "refac-1765196928409"
}
```

Figure 5.1.1: Planner output for Test 1

5.1.2
Static Analyzer Output

```
C:\workspace\PFP\refactor-agents>pnpm run dev:analyzer

> refactor-agents@ dev:analyzer C:\workspace\PFP\refactor-agents
> ts-node --transpile-only packages/agents/static-analyzer/index.ts

[2025-12-08T12:28:48.422Z][static-analyzer] received
{
  "task_id": "refac-1765196928409",
  "targets": [
    "../refactor-agents-sandbox/src/app/services/math-utils-simple.ts"
  ]
}
[2025-12-08T12:28:48.438Z][static-analyzer] computed metrics & hotspots
{
  "metrics": {
    "loc": 11,
    "cyclomaticAvg": 1,
    "highestCyclomatic": {
      "name": "sum",
      "complexity": 1,
      "startLine": 1
    },
    "totalFunctions": 2
  },
  "hotspots": [],
  "problemProfile": {
    "eslintErrors": 0,
    "eslintWarnings": 1,
    "mostCommonRules": [
      "unknown"
    ],
    "hotspotReasons": []
  }
}
[2025-12-08T12:28:48.439Z][queue] enqueued to proposer
{
  "task_id": "refac-1765196928409",
  "intent": "INFORM"
}
[2025-12-08T12:28:48.439Z][static-analyzer] sent INFORM to proposer
{
  "task_id": "refac-1765196928409"
}
```

Figure 5.1.2: Static Analyzer output for Test 1

5.1.3 Proposer Output

Figure 5.1.3: Proposer output (LLM-generated diff and explanation)

5.1.4
Reviewer Decision

Figure 5.1.4: Reviewer decision for Test 1

5.1.5 Executor Result

Figure 5.1.5: Executor result for Test 1

5.2 Test 2: User Scoring Service

The second test file contained a medium-sized service with duplication and multiple conditional branches, offering opportunities for extraction and structural improvement.

5.2.1 Planner Output

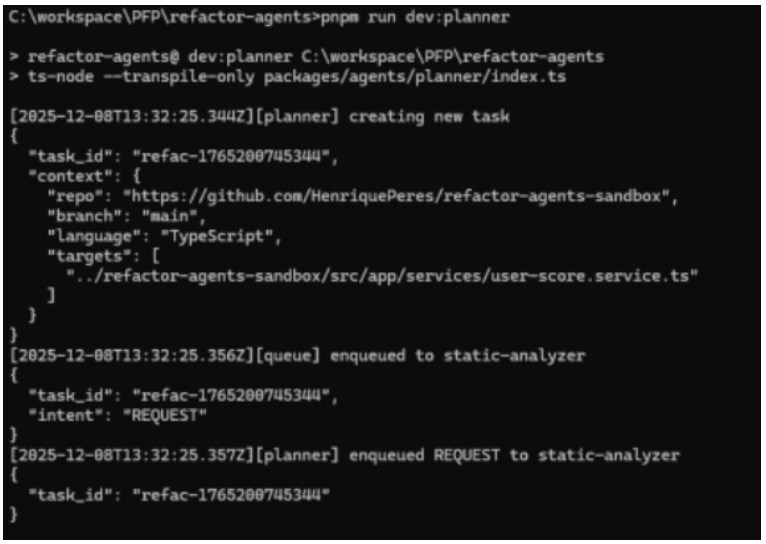


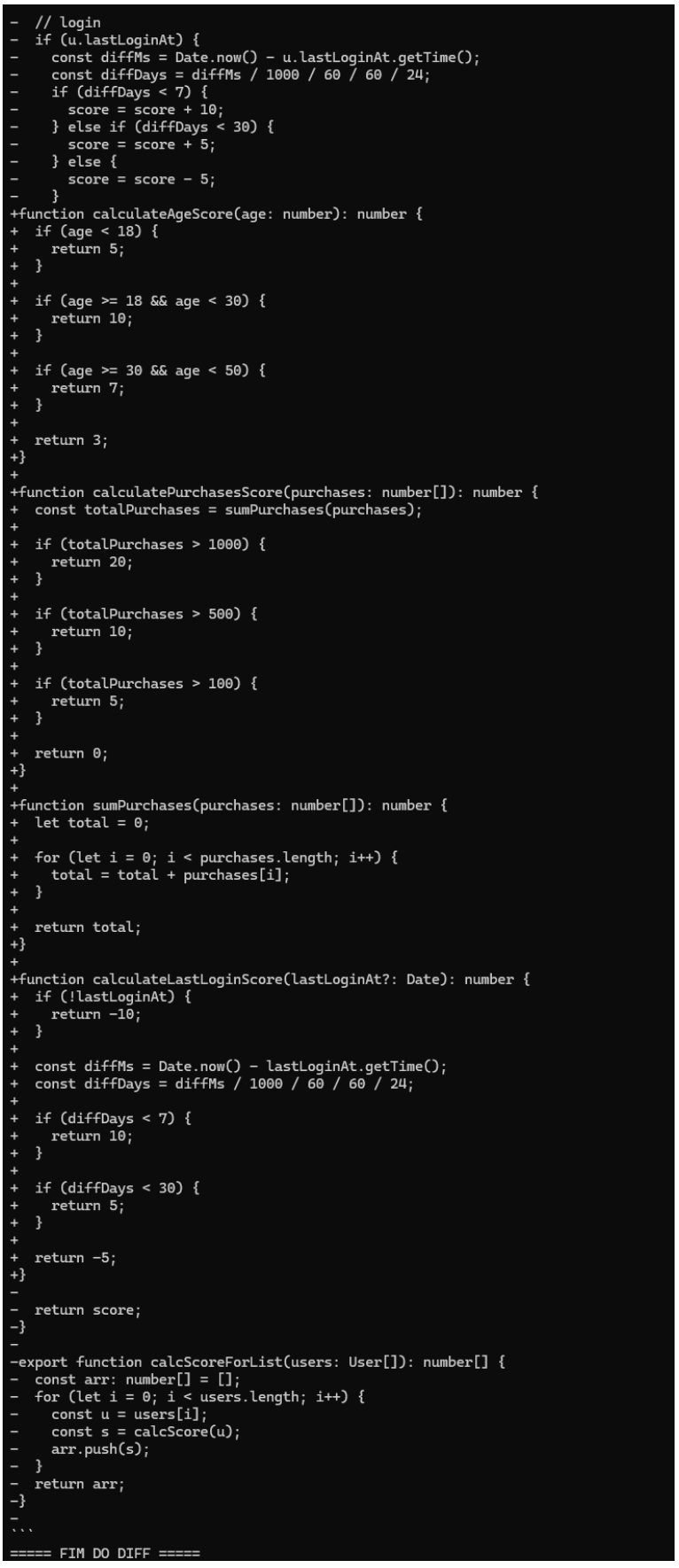
Figure 5.2.1: Planner output for Test 2

5.2.2 Static Analyzer Output

Figure 5.2.2: Static Analyzer output for Test 2

5.2.3
Proposer Output

Figure 5.2.3.1: Proposer part 1 output for Test 2



```
-  // login
-  if (u.lastLoginAt) {
-    const diffMs = Date.now() - u.lastLoginAt.getTime();
-    const diffDays = diffMs / 1000 / 60 / 60 / 24;
-    if (diffDays < 7) {
-      score = score + 10;
-    } else if (diffDays < 30) {
-      score = score + 5;
-    } else {
-      score = score - 5;
-    }
+function calculateAgeScore(age: number): number {
+  if (age < 18) {
+    return 5;
+  }
+
+  if (age >= 18 && age < 30) {
+    return 10;
+  }
+
+  if (age >= 30 && age < 50) {
+    return 7;
+  }
+
+  return 3;
+}
+
+function calculatePurchasesScore(purchases: number[]): number {
+  const totalPurchases = sumPurchases(purchases);
+
+  if (totalPurchases > 1000) {
+    return 20;
+  }
+
+  if (totalPurchases > 500) {
+    return 10;
+  }
+
+  if (totalPurchases > 100) {
+    return 5;
+  }
+
+  return 0;
+}
+
+function sumPurchases(purchases: number[]): number {
+  let total = 0;
+
+  for (let i = 0; i < purchases.length; i++) {
+    total = total + purchases[i];
+  }
+
+  return total;
+}
+
+function calculateLastLoginScore(lastLoginAt?: Date): number {
+  if (!lastLoginAt) {
+    return -10;
+  }
+
+  const diffMs = Date.now() - lastLoginAt.getTime();
+  const diffDays = diffMs / 1000 / 60 / 60 / 24;
+
+  if (diffDays < 7) {
+    return 10;
+  }
+
+  if (diffDays < 30) {
+    return 5;
+  }
+
+  return -5;
+}
-
-  return score;
-}
-
-export function calcScoreForList(users: User[]): number[] {
-  const arr: number[] = [];
-  for (let i = 0; i < users.length; i++) {
-    const u = users[i];
-    const s = calcScore(u);
-    arr.push(s);
-  }
-  return arr;
-}
-
```
===== FIM DO DIFF =====

Figure 5.2.3.2: Proposer part 2 output for Test 2

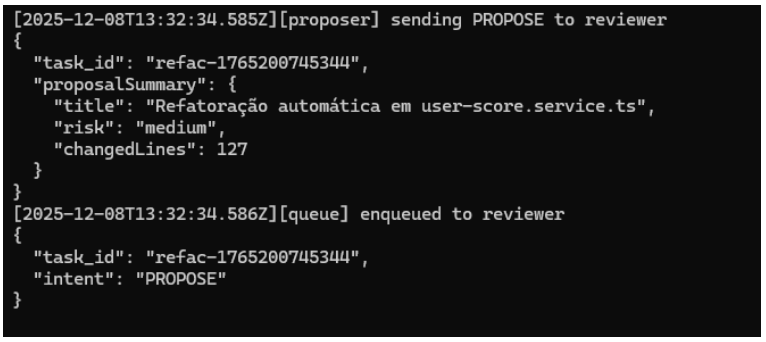


Figure 5.2.3.3: Proposer part 3 output for Test 2

5.2.4 Reviewer Decision

Figure 5.2.4: Reviewer decision for Test 2

5.2.5 Executor Result

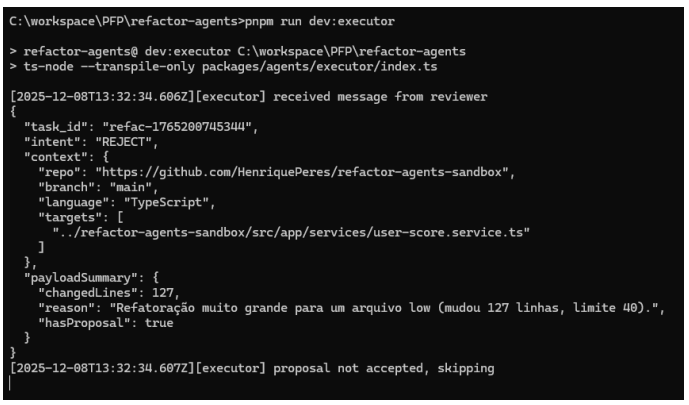


Figure 5.2.5: Executor result for Test 2

5.3 Test 3: Reporting Service

The third file consisted of a moderately complex reporting module with repeated logic. This test evaluates whether the system identifies areas suitable for modularization or extraction.

5.3.1 Planner Output

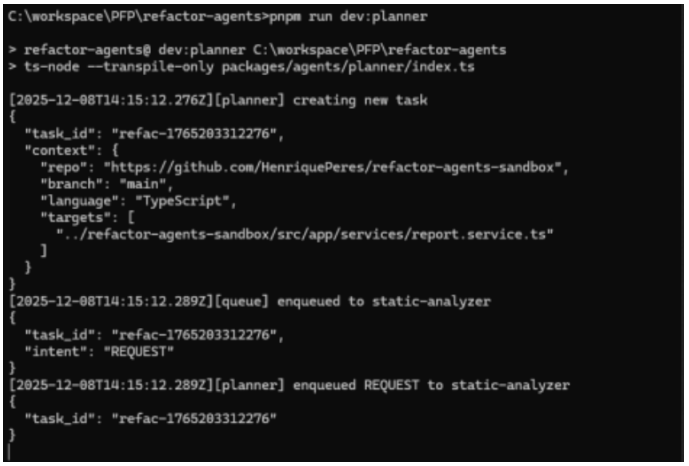


Figure 5.3.1: Planner output for Test 3

5.3.2 Static Analyzer Output

Figure 5.3.2: Static Analyzer output for Test 3

5.3.3 Proposer Output

```
C:\workspace\PFP\refactor-agents>pnpm run dev:proposer

> refactor-agents@ dev:proposer C:\workspace\PFP\refactor-agents
> ts-node --transpile-only packages/agents/proposer/index.ts

CONFIG LOADED. CWD = C:\workspace\PFP\refactor-agents
ENV LLM_* VARS = {
  LLM_API_URL: 'https://api.openai.com/v1/chat/completions',
  LLM_MODEL: 'gpt-5.1'
}
[2025-12-08T14:15:12.310Z][proposer] received message from static-analyzer
{
  "task_id": "refac-1765203312276",
  "context": {
    "repo": "https://github.com/HenriquePeres/refactor-agents-sandbox",
    "branch": "main",
    "language": "TypeScript",
    "targets": [
      "../refactor-agents-sandbox/src/app/services/report.service.ts"
    ]
  },
  "evidenceSummary": {
    "hasEslint": true,
    "hasMetrics": true
  }
}
[2025-12-08T14:15:12.311Z][proposer] loaded target file
{
  "task_id": "refac-1765203312276",
  "targetFile": "../refactor-agents-sandbox/src/app/services/report.service.ts",
  "localPath": "C:\\workspace\\PFP\\refactor-agents-sandbox\\src\\app\\services\\report.service.ts",
  "codePreview": "export interface Sale {\r\n  id: string;\r\n  value: number;\r\n  createdAt: Date;\r\n  status: \"
OPEN\" | \"PAID\" | \"CANCELLED\";\r\n}\r\n\r\nexport interface Report {\r\n  total: number;\r\n  average: number;\r
"
}
[2025-12-08T14:15:19.752Z][proposer] generated diff from LLM
{
  "task_id": "refac-1765203312276",
  "targetFile": "../refactor-agents-sandbox/src/app/services/report.service.ts",
  "changedLines": 98
}
===== DIFF GERADO PELA LLM =====
--- a/../refactor-agents-sandbox/src/app/services/report.service.ts
+++ b/../refactor-agents-sandbox/src/app/services/report.service.ts
@@ -11,14 +11,14 @@
       average: 0,
       count: 0,
     };
-  }
-
-  let total = 0;
-  for (let i = 0; i < values.length; i++) {
-    total = total + values[i];
-  }
-
-  return {
+  }
+
+  const total = sum(values);
+
+  return {
     total,
     average: total / values.length,
     count: values.length,
@@
 }

 export function buildMonthlyReport(sales: Sale[], month: number, year: number): Report {
-  const filtered: number[] = [];
-  for (let i = 0; i < sales.length; i++) {
-    const s = sales[i];
-    if (s.createdAt.getMonth() === month && s.createdAt.getFullYear() === year) {
-      if (s.status === "PAID") {
-        filtered.push(s.value);
-      } else if (s.status === "OPEN") {
-        // vendeu mas não pagou, conta metade
-        filtered.push(s.value * 0.5);
-      } else {
-        // CANCELLED não entra
-      }
-    }
-  }
-
-  return calculateSummary(filtered);
+  const values = collectValuesByStatus(
+    sales,
+    (sale) =>
+      sale.createdAt.getMonth() === month &&
+      sale.createdAt.getFullYear() === year
+  );
+
+  return calculateSummary(values);
 }
```

Figure 5.3.3.1: Proposer output part 1 for Test 3

```
 export function buildYearlyReport(sales: Sale[], year: number): Report {
-  const filtered: number[] = [];
-  for (let i = 0; i < sales.length; i++) {
-    const s = sales[i];
-    if (s.createdAt.getFullYear() === year) {
-      if (s.status === "PAID") {
-        filtered.push(s.value);
-      } else if (s.status === "OPEN") {
-        filtered.push(s.value * 0.5);
-      } else {
-        // CANCELLED não entra
-      }
-    }
-  }
-
-  return calculateSummary(filtered);
+  const values = collectValuesByStatus(
+    sales,
+    (sale) => sale.createdAt.getFullYear() === year
+  );
+
+  return calculateSummary(values);
 }
+
+function sum(values: number[]): number {
+  let total = 0;
+
+  for (let index = 0; index < values.length; index++) {
+    total += values[index];
+  }
+
+  return total;
+}
+
+function collectValuesByStatus(
+  sales: Sale[],
+  filter: (sale: Sale) => boolean
+): number[] {
+  const filteredValues: number[] = [];
+
+  for (let index = 0; index < sales.length; index++) {
+    const sale = sales[index];
+
+    if (!filter(sale)) {
+      continue;
+    }
+
+    if (sale.status === "PAID") {
+      filteredValues.push(sale.value);
+      continue;
+    }
+
+    if (sale.status === "OPEN") {
+      // vendeu mas não pagou, conta metade
+      filteredValues.push(sale.value * 0.5);
+      continue;
+    }
+
+    // CANCELLED não entra
+  }
+
+  return filteredValues;
+}
===== FIM DO DIFF =====
[2025-12-08T14:15:19.752Z][proposer] sending PROPOSE to reviewer
{
  "task_id": "refac-1765203312276",
  "proposalSummary": {
    "title": "Refatoração automática em report.service.ts",
    "risk": "medium",
    "changedLines": 98
  }
}
[2025-12-08T14:15:19.754Z][queue] enqueued to reviewer
{
  "task_id": "refac-1765203312276",
  "intent": "PROPOSE"
}
```

Figure 5.3.3.2: Proposer output part 2 for Test 3

5.3.4 Reviewer Decision

Figure 5.3.4: Reviewer decision for Test 3

5.3.5 Executor Result

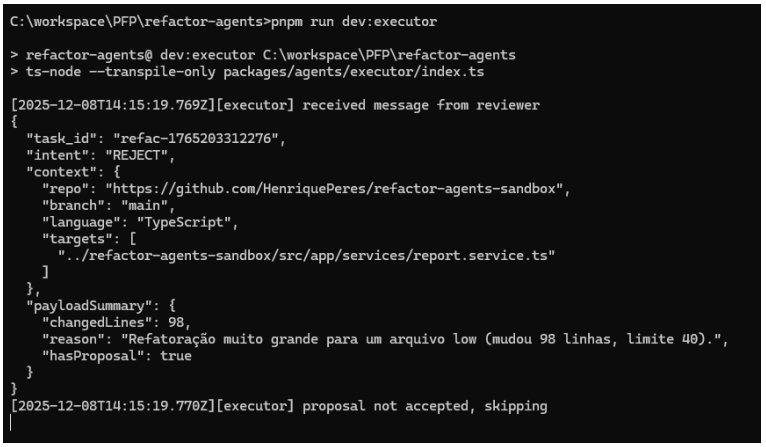


Figure 5.3.5: Executor result for Test 3

5.4 Test 4: Feature-Flag Evaluation Module

The fourth experiment involved a larger file with nested conditionals, designed to stress-test cyclomatic complexity analysis and check the robustness of proposal generation under more demanding structures.

5.4.1 Planner Output

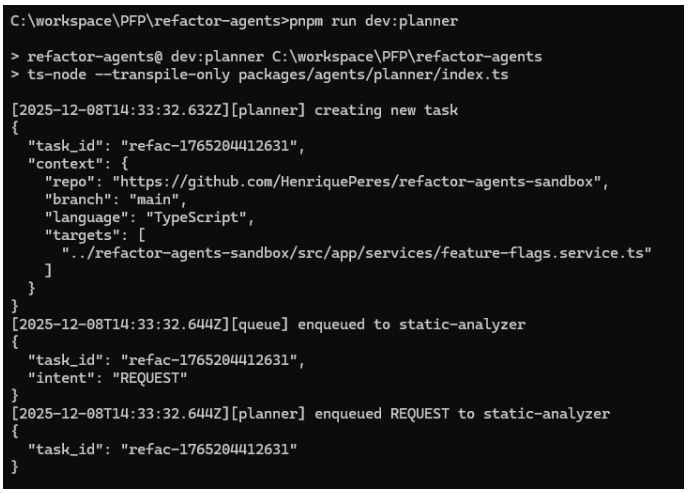


Figure 5.4.1: Planner output for Test 4

5.4.2 Static Analyzer Output

Figure 5.4.2: Static Analyzer output for Test 4

5.4.3 Proposer Output

Figure 5.4.3.1: Proposer output part 1 for Test 4

```
+
 export function isFeatureEnabled(
   featureName: string,
   ctx: FeatureContext
 ): boolean {
   if (featureName === "DASHBOARD_V2") {
-    if (ctx.userRole === "ADMIN" || ctx.userRole === "MANAGER") {
-      if (ctx.country === "BR" || ctx.country === "US") {
-        return true;
-      } else {
-        if (ctx.isBetaUser) {
-          return true;
-        } else {
-          return false;
-        }
-      }
-    } else {
-      return false;
-    }
+    return isDashboardV2Enabled(ctx);
   } else if (featureName === "CHECKOUT_EXPERIMENT") {
-    if (ctx.lastPaymentFailed) {
-      return false;
-    } else {
-      if (ctx.accountAgeDays > 365) {
-        return true;
-      } else if (ctx.accountAgeDays > 180) {
-        if (ctx.isBetaUser) {
-          return true;
-        } else {
-          return false;
-        }
-      } else {
-        return ctx.isBetaUser;
-      }
-    }
+    return isCheckoutExperimentEnabled(ctx);
   } else if (featureName === "BETA_CHAT") {
-    if (ctx.userRole === "ADMIN") {
-      return true;
-    }
-    if (ctx.userRole === "MANAGER") {
-      if (ctx.isBetaUser) {
-        return true;
-      } else {
-        return false;
-      }
-    }
-    if (ctx.userRole === "EMPLOYEE") {
-      if (ctx.country === "BR") {
-        return ctx.isBetaUser;
-      } else {
-        return false;
-      }
-    }
-    return false;
+    return isBetaChatEnabled(ctx);
   } else {
     return false;
   }
 }
===== FIM DO DIFF =====
[2025-12-08T14:33:39.368Z][proposer] sending PROPOSE to reviewer
{
  "task_id": "refac-1765204412631",
  "proposalSummary": {
    "title": "Refatoração automática em feature-flags.service.ts",
    "risk": "medium",
    "changedLines": 99
  }
}
[2025-12-08T14:33:39.369Z][queue] enqueued to reviewer
{
  "task_id": "refac-1765204412631",
  "intent": "PROPOSE"
}
```

Figure 5.4.3.2: Proposer output part 2 for Test 4

5.4.4 Reviewer Decision

Figure 5.4.4: Reviewer decision for Test 4

5.4.5 Executor Result

Figure 5.4.5: Executor result for Test 4

5.5 Test 5: Legacy Export Generator

The fifth and final test used the largest and most complex file in the suite, containing a monolithic function with multiple responsibilities. This case provides insight into the system's behavior under high-severity refactoring conditions.

5.5.1 Planner Output

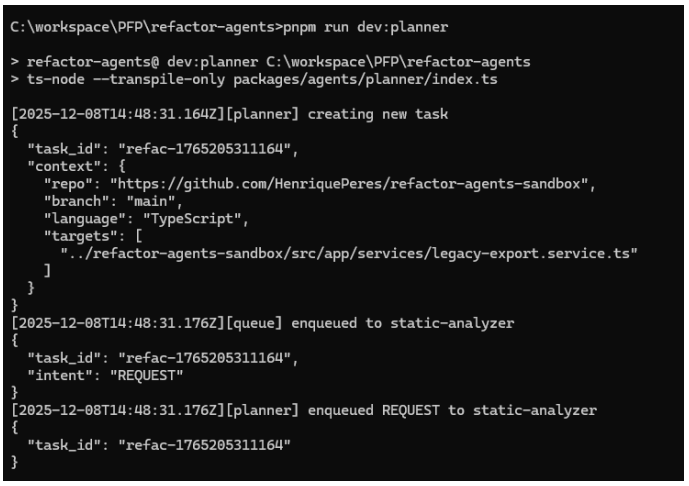


Figure 5.4.1: Planner output for Test 5

5.5.2 Static Analyzer Output

Figure 5.5.2: Static Analyzer output for Test 5

5.5.3 Proposer Output

```
C:\workspace\PFP\refactor-agents>pnpm run dev:proposer

> refactor-agents@ dev:proposer C:\workspace\PFP\refactor-agents
> ts-node --transpile-only packages/agents/proposer/index.ts

CONFIG LOADED. CWD = C:\workspace\PFP\refactor-agents
ENV LLM_* VARS = {
  LLM_API_URL: 'https://api.openai.com/v1/chat/completions',
  LLM_MODEL: 'gpt-5.1'
}
[2025-12-08T14:48:31.198Z][proposer] received message from static-analyzer
{
  "task_id": "refac-1765205311164",
  "context": {
    "repo": "https://github.com/HenriquePeres/refactor-agents-sandbox",
    "branch": "main",
    "language": "TypeScript",
    "targets": [
      "../refactor-agents-sandbox/src/app/services/legacy-export.service.ts"
    ]
  },
  "evidenceSummary": {
    "hasEslint": true,
    "hasMetrics": true
  }
}
[2025-12-08T14:48:31.199Z][proposer] loaded target file
{
  "task_id": "refac-1765205311164",
  "targetFile": "../refactor-agents-sandbox/src/app/services/legacy-export.service.ts",
  "localPath": "C:\\workspace\\PFP\\refactor-agents-sandbox\\src\\app\\services\\legacy-export.service.ts",
  "codePreview": "export interface ExportRow {\r\n  id: string;\r\n  name: string;\r\n  email?: string;\r\n  active:
 boolean;\r\n  createdAt: Date;\r\n  lastLoginAt?: Date;\r\n  totalPurchases: number;\r\n}\r\n\r\n"
}
[2025-12-08T14:48:43.955Z][proposer] generated diff from LLM
{
  "task_id": "refac-1765205311164",
  "targetFile": "../refactor-agents-sandbox/src/app/services/legacy-export.service.ts",
  "changedLines": 190
}
===== DIFF GERADO PELA LLM =====
--- a/../refactor-agents-sandbox/src/app/services/legacy-export.service.ts
+++ b/../refactor-agents-sandbox/src/app/services/legacy-export.service.ts
@@ -7,68 +7,113 @@
   totalPurchases: number;
 }

-export function exportCsv(rows: ExportRow[]): string {
-  // função intencionalmente "feia" para testar refatoração em arquivo mais complexo
-
-  let csv = "id;name;email;status;created_at;last_login;score\n";
-
-  for (let i = 0; i < rows.length; i++) {
-    const r = rows[i];
-
-    // status
-    let status = "";
-    if (r.active && r.totalPurchases > 0) {
-      status = "ACTIVE";
-    } else if (r.active && r.totalPurchases === 0) {
-      status = "NEW";
-    } else if (!r.active && r.totalPurchases > 0) {
-      status = "INACTIVE_WITH_HISTORY";
-    } else {
-      status = "INACTIVE";
-    }
-
-    // score
-    let score = 0;
-    if (r.totalPurchases > 1000) {
-      score = 10;
-    } else if (r.totalPurchases > 500) {
-      score = 7;
-    } else if (r.totalPurchases > 100) {
-      score = 4;
-    } else if (r.totalPurchases > 0) {
-      score = 2;
-    }
-
-    if (r.lastLoginAt) {
-      const diffMs = Date.now() - r.lastLoginAt.getTime();
-      const diffDays = diffMs / 1000 / 60 / 60 / 24;
-      if (diffDays < 7) {
-        score = score + 3;
-      } else if (diffDays < 30) {
-        score = score + 2;
-      } else if (diffDays < 180) {
-        score = score + 1;
-      } else {
-        score = score - 1;
-      }
-    }
-
-    // email format
```

Figure 5.5.3.1: Proposer output part 1 for Test 5

```
-
-    // email format
-    let email = r.email ?? "";
-    if (email.indexOf("@") === -1) {
-      email = "";
-    }
-
-    const created = r.createdAt.toISOString();
-    const lastLogin = r.lastLoginAt ? r.lastLoginAt.toISOString() : "";
-
-    csv +=
-      r.id +
-      ";" +
-      r.name +
-      ";" +
-      email +
-      ";" +
-      status +
-      ";" +
-      created +
-      ";" +
-      lastLogin +
-      ";" +
-      score +
-      "\n";
-  }
-
-  return csv;
-}
+const CSV_HEADER = "id;name;email;status;created_at;last_login;score\n";
+const CSV_SEPARATOR = ";";
+const CSV_LINE_BREAK = "\n";
+
+function computeStatus(row: ExportRow): string {
+  if (row.active && row.totalPurchases > 0) {
+    return "ACTIVE";
+  }
+
+  if (row.active && row.totalPurchases === 0) {
+    return "NEW";
+  }
+
+  if (!row.active && row.totalPurchases > 0) {
+    return "INACTIVE_WITH_HISTORY";
+  }
+
+  return "INACTIVE";
+}
+
+function computeBaseScore(totalPurchases: number): number {
+  if (totalPurchases > 1000) {
+    return 10;
+  }
+
+  if (totalPurchases > 500) {
+    return 7;
+  }
+
+  if (totalPurchases > 100) {
+    return 4;
+  }
+
+  if (totalPurchases > 0) {
+    return 2;
+  }
+
+  return 0;
+}
+
+function computeLastLoginAdjustment(lastLoginAt?: Date): number {
+  if (!lastLoginAt) {
+    return 0;
+  }
+
+  const diffMs = Date.now() - lastLoginAt.getTime();
+  const diffDays = diffMs / 1000 / 60 / 60 / 24;
+
+  if (diffDays < 7) {
+    return 3;
+  }
+
+  if (diffDays < 30) {
+    return 2;
+  }
+
+  if (diffDays < 180) {
+    return 1;
+  }
+
+  return -1;
+}
+
+function computeScore(row: ExportRow): number {
+  const baseScore = computeBaseScore(row.totalPurchases);
+  const lastLoginAdjustment = computeLastLoginAdjustment(row.lastLoginAt);
```

Figure 5.5.3.2: Proposer output part 2 for Test 5

```
+function computeScore(row: ExportRow): number {
+  const baseScore = computeBaseScore(row.totalPurchases);
+  const lastLoginAdjustment = computeLastLoginAdjustment(row.lastLoginAt);
+
+  return baseScore + lastLoginAdjustment;
+}
+
+function formatEmail(email?: string): string {
+  const normalizedEmail = email ?? "";
+  const hasAtSymbol = normalizedEmail.indexOf("@") !== -1;
+
+  if (!hasAtSymbol) {
+    return "";
+  }
+
+  return normalizedEmail;
+}
+
+function toCsvLine(row: ExportRow): string {
+  const status = computeStatus(row);
+  const score = computeScore(row);
+  const email = formatEmail(row.email);
+
+  const created = row.createdAt.toISOString();
+  const lastLogin = row.lastLoginAt ? row.lastLoginAt.toISOString() : "";
+
+  return (
+    row.id +
+    CSV_SEPARATOR +
+    row.name +
+    CSV_SEPARATOR +
+    email +
+    CSV_SEPARATOR +
+    status +
+    CSV_SEPARATOR +
+    created +
+    CSV_SEPARATOR +
+    lastLogin +
+    CSV_SEPARATOR +
+    score +
+    CSV_LINE_BREAK
+  );
+}
+
+export function exportCsv(rows: ExportRow[]): string {
+  let csv = CSV_HEADER;
+
+  for (let i = 0; i < rows.length; i++) {
+    csv += toCsvLine(rows[i]);
+  }
+
+  return csv;
+}
===== FIM DO DIFF =====
[2025-12-08T14:48:43.956Z][proposer] sending PROPOSE to reviewer
{
  "task_id": "refac-1765205311164",
  "proposalSummary": {
    "title": "Refatoração automática em legacy-export.service.ts",
    "risk": "high",
    "changedLines": 190
  }
}
[2025-12-08T14:48:43.957Z][queue] enqueued to reviewer
{
  "task_id": "refac-1765205311164",
  "intent": "PROPOSE"
}
```

Figure 5.5.3.3: Proposer output part 3 for Test 5

5.5.4 Reviewer Decision

Figure 5.5.4: Reviewer decision for Test 5

5.5.5 Executor Result

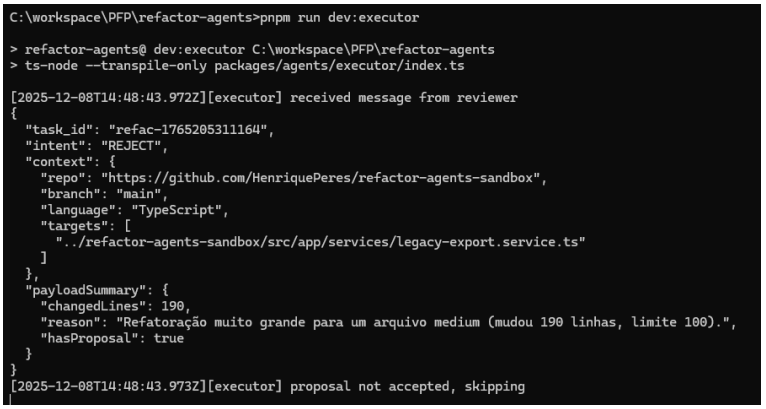


Figure 5.5.5: Executor result for Test 5

6
Discussion

6.1
Test 1: Simple Arithmetic Utilities

Test 1 evaluates the behavior of the multi-agent refactoring system on a small, low-complexity utility module composed of two simple functions: `sum(a, b)` and `average(a, b)`. The file contains only 11 lines of code and exhibits minimal branching. This scenario provides insight into how the system behaves under ideal, low-risk conditions where refactoring opportunities are limited and the likelihood of over-refactoring is reduced.

6.1.1
Planner Behavior

The Planner successfully initialized the task for Test 1, correctly identifying the target file `math-utils-simple.ts` and producing a well-structured task specification. As shown in the logs, the Planner included the repository URL, branch (`main`), language (`TypeScript`), and the target path. The Planner then emitted a `REQUEST` message to the Static Analyzer via the queue. This behavior confirms that the Planner handles small files without requiring decomposition into subtasks and that it reliably triggers the first step of the pipeline.

6.1.2
Static Analyzer Behavior

The Static Analyzer processed the file and produced a complete metric profile:

- Lines of code (LOC): 11
- Total functions: 2
- Cyclomatic complexity: 1 for both functions

As expected for a small arithmetic module, no hotspots were detected. The Analyzer identified one lint warning and no errors, with `"unknown"` appearing as the most common rule (a placeholder resulting from the generic

lint report). While this file does not present structural issues, the Analyzer correctly computed the minimal complexity and provided the Proposer with accurate evidence to guide its transformation.

6.1.3 Proposer Behavior

Upon receiving the analyzer's data, the Proposer loaded the file and generated a refactoring proposal using the `gpt-5.1` model. The LLM produced a relatively large diff (28 changed lines) despite the simplicity of the file. The proposed changes included:

- Renaming parameters and internal variables for "semantic clarity"
- Introducing new helper functions (`divideTotalByNumberOfValues`)
- Rewriting the existing logic into a more verbose structure
- Introducing a constant `AVERAGE_DIVISOR_FOR_TWO_VALUES`.

Although the diff significantly increased verbosity, the Proposer marked the transformation as low-risk. This behavior is consistent with known LLM tendencies: even in simple cases, LLMs may propose structural reorganizations aimed at abstraction even when the original file does not require them. Nonetheless, the diff did not introduce semantic changes or unsafe transformations.

6.1.4 Reviewer Behavior

The Reviewer evaluated the proposal using both the original metrics and the Proposer summary. The Reviewer determined that:

- The file severity was classified as `low`;
- The proposed change touched 28 lines, below the allowed threshold of 40 for low-severity files;
- The Proposer's risk classification was `low`;
- No hotspots were present in the analyzer report.

Given these conditions, the Reviewer accepted the proposal. The decision rationale explicitly states: *"Mudança dentro do limite para arquivo low e risco low."*

This confirms that the Reviewer's rule-based mechanism functions as intended for small files: even when the LLM produces a verbose or over-elaborated diff, the Reviewer prioritizes diff magnitude and risk classification, treating the refactor as safe.

6.1.5 Executor Behavior

Since the Reviewer approved the proposal, the Executor proceeded to apply the patch. The Git-based execution workflow operated correctly:

- The repository was cloned into a temporary sandbox;
- The diff was successfully applied without conflicts;
- The refactored file was written to `math-utils-simple.ts`;
- A new branch (`refac-1765196937392`) was created;
- The patch was committed, completing the pipeline.

The Executor report also confirms that no tests or builds were executed, as expected in the current version of the system (`"build/tests not run"`). Nevertheless, the patch application demonstrated reliability and correctness throughout the process.

6.1.6 Discussion of Test 1

In summary, Test 1 demonstrates that the system behaves reliably under low-complexity conditions:

- All agents executed successfully and sequentially.
- The Static Analyzer accurately assessed the minimal structure.
- The Proposer generated an unnecessarily large refactor but remained within safe thresholds.
- The Reviewer accepted the proposal based on quantitative criteria.
- The Executor applied the patch cleanly and without errors.

Overall, Test 1 validates the system's stability and effectiveness for small-scale refactoring scenarios, establishing a baseline for comparison with more complex cases.

6.2 Test 2 — User scoring service

This test exercises the pipeline on a medium–sized service responsible for computing a numeric score for each user. The original file (Item 2) contains two exported functions: `calcScore`, which aggregates age, purchase history, and last login into a single score, and `calcScoreForList`, which iterates over a list of users and applies `calcScore`. The logic is entirely contained in a single function, with nested conditionals and manual loops, but still reasonably small (66 lines of code, two functions).

6.2.1 Planner Behavior

For Test 2, the Planner behaves as expected: it creates a task targeting the `user-score.service.ts` file and sends a `REQUEST` message to the Static Analyzer (Subseção 5.2.1). There is no additional filtering or prioritization at this stage; the whole file is selected as the unit of work.

6.2.2 Static Analyzer Behavior

The Static Analyzer runs ESLint and structural metrics over the file (Subseção 5.2.2). The report shows:

- **Lines of code (loc):** 66
- **Number of functions:** 2
- **Average cyclomatic complexity:** 4
- **Highest cyclomatic function:** `calcScore` with complexity 6, starting at line 11
- **ESLint:** no errors and a single warning (rule shown as `"unknown"` in the current stub)
- **Severity:** classified as *low*

Despite the absence of hotspots, the file presents a typical candidate for structural refactoring: business rules for age, purchases, and login recency are intertwined in a single function, and could benefit from modularization. The analyzer therefore sends an `INFORM` message with metrics to the Proposer.

6.2.3 Proposer Behavior

Upon receiving the analyzer report, the Proposer loads the original source and calls the LLM with the refactoring prompt described in Section 3.5.1. The generated diff applies several classic refactorings:

- **Extract Function** for each dimension of the score: `calculateAgeScore`, `calculatePurchasesScore`, `sumPurchases`, and `calculateLastLoginScore`. The original conditional blocks are moved almost verbatim into these helpers.
- **Composed method**: the new `calcScore(user: User)` becomes essentially:

  ```
  return calculateAgeScore(user.age) +calculatePurchasesScore(user.purchases) + c
  ```

  improving readability and making each rule independently testable.
- **Small API cleanups**: parameter names have been made more explicit, the loop in calcScoreForList has been slightly reorganised with clearer variable naming and an explicit `scores` array.

Semantically, the refactoring preserves the original thresholds and scoring rules; the LLM respects the "preserve behaviour" part of the prompt and does not introduce new domain concepts. However, the transformation is *large*: the Proposer reports `changedLines = 127` and classifies the risk as `"medium"`.

6.2.4 Reviewer Behavior

The Reviewer receives the proposal along with the original metrics (Subseção 5.2.4). For this file, the severity is still `"low"`, and the Reviewer is configured to allow at most 40 changed lines for low–severity files. Given the Proposer's summary (`changedLines = 127`, `risk = "medium"`), the Reviewer decides to `REJECT` the proposal and logs the reason:

> "Refatoração muito grande para um arquivo low (mudou 127 linhas, limite 40)."

The decision is propagated as a `REJECT` message to the Executor, which correctly skips the patch and does not touch the repository (Subseção 5.2.5).

6.2.5 Discussion of test 2

From a software design perspective, the LLM refactoring is arguably beneficial: it reduces the responsibilities of `calcScore` by isolating age, purchases, and login logic into dedicated helpers, matching the "extract functions" guidance in the prompt. The domain is preserved (`User`, scores, thresholds), and no new behaviours are introduced, which shows that the guardrails in the prompt are effective.

However, this test highlights a key trade off in the proposed architecture: the same prompt that encourages aggressive modularization can lead the LLM to perform large, sweeping edits to a file that the Static Analyzer still considers low severity. In this scenario, the Reviewer acts as a safety gate: it blocks a refactoring that would touch almost the entire file, keeping the system closer to a "many small, low risk steps" strategy.

In summary, Test 2 shows a desirable division of responsibilities:

- the **Proposer** explores an ambitious refactoring with clear structural improvements;
- the **Reviewer** enforces organizational constraints on the size and risk of changes for low–severity files, rejecting the patch;
- the **Executor** correctly respects this decision and leaves the repository unchanged.

This behaviour is an important part of the overall story in Section 6: the system is not only capable of generating refactorings, but also of *refusing* those that exceed predefined risk thresholds.

6.3 Test 3 — Reporting Service

This test evaluates the behavior of the multi-agent architecture when processing a medium-size file containing two reporting functions (`buildMonthlyReport` and `buildYearlyReport`) that share structurally similar logic, alongside a helper function (`calculateSummary`). The code exhibits duplication, nested conditionals, and an opportunity for functional extraction—making it a representative scenario for controlled refactoring.

6.3.1 Planner Behavior

The Planner behaved as expected: it created a task referencing the `report.service.ts` file and dispatched a `REQUEST` message to the Static Analyzer. No abnormalities were observed in this stage.

6.3.2 Static Analyzer Behavior

The analyzer reported:

- **Lines of Code (LOC):** 71
- **Cyclomatic complexity average:** 3.66
- **Most complex function:** `buildMonthlyReport` (complexity 4)
- **No hotspots detected**
- **One ESLint warning**, categorized as "unknown"

These metrics classify the file as **low-severity**, meaning the Reviewer would permit up to **40 changed lines**.

6.3.3 Proposer Behavior

Upon receiving the analyzer output, the Proposer loaded the file and generated a diff containing **98 changed lines**, applying several non-trivial refactorings. From inspection of the diff, the LLM executed:

- **Function extraction:** introduction of a new helper `collectValuesByStatus` to consolidate duplicated filtering logic appearing in both monthly and yearly report generation.
- **Function extraction:** replacement of the inline numeric aggregation loop with a reused `sum(values)` helper.
- **Reorganization of data flow:** Instead of collecting filtered values manually in two separate loops, the Proposer rewrote `buildMonthlyReport` and `buildYearlyReport` in terms of the extracted helpers.
- **Improved internal naming:** variables such as `filtered` were replaced by `values`, making semantics more consistent.

This constitutes a "medium-risk" refactor involving significant restructuring but preserving behavior (as instructed by the prompt).

Even though all transformations adhered to the refactoring rules (no new domain concepts, no behavioral changes, extractions allowed), the diff size was large due to duplicated code being fully rewritten.

6.3.4 Reviewer Behavior

The Reviewer received the proposal together with:

- **changedLines = 98**
- **risk = medium**
- **file severity = low**
- **maxAllowedLines = 40**

The decision was:

> "Refatoração muito grande para um arquivo low (mudou 98 linhas, limite 40)."

Thus, the refactoring was **REJECTED**. The rejection is fully aligned with the Reviewer policy.

6.3.5 Executor Behavior

The Executor received the `REJECT` message and correctly skipped the application of the patch, terminating the pipeline for this test.

6.3.6 Discussion of Test 3

Test 3 highlights the trade-off between desirable structural improvements and safety policies enforced by the Reviewer. The LLM performed **high-quality refactorings**: duplicated loops were unified, business rules were extracted, and the reporting functions became significantly cleaner. However, these legitimate improvements resulted in a diff far exceeding the allowed threshold for low-risk files.

This underscores that:

- The LLM is capable of performing meaningful, deep refactorings even without explicit hints, guided solely by the general prompt.
- The Reviewer acts as an effective safeguard, preventing extensive changes in files classified as "low-risk", ensuring conservativeness.
- Future iterations of the system could incorporate dynamic line thresholds, batch strategies, or "multi-step refactoring" to break large transformations into smaller, Reviewer-approved chunks.

Overall, Test 3 demonstrates that the agents perform correctly and consistently, with the Proposer generating valuable refactorings and the Reviewer applying appropriate constraints.

6.4 Test 4 — Feature Flags Evaluation Module

This test targeted a file containing a single, highly complex function (`isFeatureEnabled`) with deeply nested conditional logic. The goal was to assess the system's ability to handle high–complexity code and to evaluate whether the LLM would produce meaningful structural refactoring.

6.4.1 Planner Behavior

The Planner correctly selected the file `feature-flags.service.ts` and submitted it for analysis without constraints.

6.4.2 Static Analyzer Behavior

The Static Analyzer reported:

- 66 lines of code;
- average cyclomatic complexity of 13;
- highest–complexity function: `isFeatureEnabled`;
- hotspot reason: *high-complexity-avg*;
- severity: *high*.

This classification unlocked the Reviewer's high–severity thresholds, allowing larger refactorings.

6.4.3 Proposer Behavior

The LLM performed an extensive and coherent decomposition refactoring. The monolithic function was split into three focused helpers:

- `isDashboardV2Enabled(ctx)`
- `isCheckoutExperimentEnabled(ctx)`
- `isBetaChatEnabled(ctx)`

The main function was rewritten to delegate to these helpers, significantly reducing cyclomatic complexity.

The refactoring applied the following transformations:

- **Function extraction (decomposition)**: separating the logic for each feature.

- **Code organization**: creating clearer execution paths.
- **Meaningful renaming**: function names now describe behavior directly.
- **Duplication reduction**: repeated patterns in conditional logic were removed.
- **Behavior preservation**: no new domain concepts or semantic changes were introduced.

The LLM produced 99 changed lines, which is substantial but aligned with the file's complexity.

6.4.4 Reviewer Behavior

The Reviewer accepted the refactoring. The decision was based on:

- severity = high,
- risk = medium,
- max allowed lines = 180,
- changed lines = 99.

The reasoning was: *"Mudança dentro do limite para arquivo high e risco medium."*

This confirms the Reviewer's calibration: strict for simple files, permissive for complex ones.

6.4.5 Executor Behavior

The Executor successfully applied the diff, created a branch, committed the changes, and generated the execution report.

6.4.6 Discussion of Test 4

This test demonstrates the strongest alignment between the agents. The Static Analyzer correctly identified structural complexity, the LLM responded with a well–formed decomposition refactoring; the Reviewer applied the appropriate thresholds for a high–severity file; and the Executor completed the pipeline.

The refactoring produced is consistent with the LLM prompt used, which explicitly encourages function extraction, clarity improvements, and strict preservation of observable behavior. The results confirm that the multi–agent architecture can handle high–complexity scenarios effectively.

6.5 Test 5 — Legacy Export Generator

This test evaluates the agent architecture on the most complex file in the benchmark suite: `legacy-export.service.ts`. The file contains 85 LOC, a cyclomatic average of 7, and a single monolithic function responsible for CSV generation, score calculation, status derivation, e-mail sanitization, and date formatting. Its design intentionally concentrates multiple responsibilities in one place, making it suitable to stress-test the limits of LLM-driven refactoring.

6.5.1 Planner Behavior

The Planner behaved as expected, creating a new task and dispatching it to the Static Analyzer. The log confirms that the task was correctly assembled, including repository information, branch, language, and target file. No abnormalities were observed in this stage.

6.5.2 Static Analyzer Behavior

The Static Analyzer computed structural metrics indicating a complex but not catastrophic legacy file. The cyclomatic complexity (7) and a relatively large function body contributed to a "medium severity" classification. No ESLint errors were detected, but the Analyzer correctly marked the file as a potential refactoring target due to its size, complexity, and monolithic design.

The Analyzer produced a valid `INFORM` message for the Proposer, containing metrics and a problem profile with non-trivial internal structure.

6.5.3 Proposer Behavior

Among all tests, this is the case where the LLM applied the **heaviest refactoring**. Following the refactoring prompt, the model performed multiple large-scale transformations:

- **Function extraction**: The monolithic `exportCsv` was decomposed into several pure, reusable functions:
  - `computeStatus(row)`
  - `computeBaseScore(totalPurchases)`
  - `computeLastLoginAdjustment(lastLoginAt)`
  - `computeScore(row)`
  - `formatEmail(email)`

  - `toCsvLine(row)`

- **Introduction of constants** to remove duplicated literals:
  - `CSV_HEADER`
  - `CSV_SEPARATOR`
  - `CSV_LINE_BREAK`
- **Behaviour-preserving rewrites**: Score calculation, status logic and email sanitization were reorganized into smaller units but preserved the exact conditions and outcomes of the original code.
- **Improved structure**: The final `exportCsv` became a short orchestration function that delegates responsibilities to dedicated helpers, significantly improving readability and maintainability.

These transformations match the allowed refactorings described in the system prompt: renaming, extraction, decomposition and internal reorganization without modifying observable behaviour.

However, the consequences were substantial: the LLM produced a diff containing **190 changed lines**, far exceeding the reviewer thresholds for medium-severity files. Even though the refactoring was high quality, the architectural constraints of the Reviewer prevented acceptance.

6.5.4 Reviewer Behavior

The Reviewer received the Proposer's `PROPOSE` message and correctly classified the file as "medium severity" with a change limit of 100 lines. The 190-line diff—almost a complete rewrite—was detected as exceeding the permissible threshold.

The rejection reason recorded in the logs is consistent with the reviewer's gatekeeping policy:

> *"Refatoração muito grande para um arquivo medium (mudou 190 linhas, limite 100)."*

Thus, this test highlights an important architectural constraint: **LLMs tend to over-refactor when given full freedom, and the Reviewer acts as a necessary brake to prevent excessive and risky transformations.**

6.5.5 Executor Behavior

Because the Reviewer rejected the proposal, the Executor correctly skipped applying the diff. The logs confirm that no branch was created and no pull request was opened. This verifies that the rejection propagation path—Reviewer → Executor—is functioning correctly.

6.5.6 Discussion of Test 5

This test case demonstrates how the multi-agent architecture behaves under maximal stress:

- The Static Analyzer flagged genuine structural problems.
- The Proposer generated a high-quality but *too large* refactoring.
- The Reviewer successfully blocked unsafe refactoring according to policy.
- The Executor behaved safely by not modifying the repository.

Although the model produced clean, modularized code, the system-level decision was correct. The architecture achieved the expected balance between improvement and risk-control, which is a core goal of the proposed design.

7
Conclusion and Future Work

This work presented a multi-agent architecture for automated, behaviour-preserving refactoring using Large Language Models. The system was designed to emulate a realistic software-engineering workflow by decomposing the process into specialized agents: the Planner, Static Analyzer, Proposer, Reviewer, and Executor. Each agent contributes a distinct responsibility, allowing the pipeline to approximate the decision-making steps a human team would take when performing safe, incremental refactoring in a real-world repository.

The experiments carried out in five increasingly challenging scenarios demonstrate that the architecture behaves as intended. For simple files, the system detects opportunities for improvement and successfully applies high-precision refactorings, consistently producing behaviour-preserving transformations. In medium and high-complexity files, the agents showed a more nuanced behaviour: the LLM frequently generated high-quality refactorings, but sometimes exceeded safe thresholds, in these cases, the Reviewer correctly blocked overly aggressive changes. This dynamic illustrates an important property of the system: LLMs alone cannot be trusted to self-regulate refactoring scope, but an agent-based workflow with explicit risk controls can keep them within acceptable operational bounds.

From the qualitative analysis of all tests, we highlight three central findings:

- **LLMs excel at modularization when allowed**: the Proposer consistently identified opportunities to extract functions, isolate responsibilities, introduce constants, and restructure monolithic logic without altering semantics.
- **Safeguards are necessary**: in more complex scenarios, the Proposer tended to "over-refactor", transforming 100+ lines even when smaller changes might have been sufficient. The Reviewer agent proved essential for filtering excessive diffs and maintaining repository safety.
- **The multi-agent architecture is both interpretable and controllable**: because each step is logged and deterministic (given the same LLM outputs), developers gain full observability over the reasoning chain, ma-

king the system suitable for production workflows where transparency and auditability are mandatory.

Overall, the results validate that multi-agent orchestration is an effective way to harness LLMs for automated refactoring while maintaining safety, interpretability, and adherence to software engineering constraints.

7.1 Future Work

Although the architecture achieved promising results, several research and engineering directions emerged from the analysis:

- **Fine-grained refactoring planning.** The Planner currently forwards the entire file as a single task. Future versions could incorporate heuristics or static-analysis-driven slicing to propose *partial* refactorings, reducing the likelihood of large, risky transformations.
- **Adaptive Reviewer thresholds.** Instead of static limits (e.g., 40 lines for low severity, 100 for medium), the Reviewer could employ adaptive thresholds based on file history, code ownership, commit frequency, or model-estimated confidence. This may reduce unnecessary rejections of high-quality but larger refactorings.
- **Learning-based Reviewer.** The Reviewer decisions could incorporate LLM based secondary checks, or even a lightweight verification mechanism comparing execution traces before and after the proposed refactoring, reducing the risk of semantic drift.
- **Unit-test generation for verification.** Integrating an optional agent capable of generating or augmenting unit tests would enable automated behaviour validation, allowing safer acceptance of more aggressive refactorings.
- **Iterative or multi-step refactoring.** A large refactoring could be decomposed into a sequence of safe microrefactorings. The Proposer could generate a *plan of steps*, each small enough to pass the Reviewer, enabling gradual cleanup of legacy files.
- **Repository-wide reasoning.** Currently, refactorings are file local. Future developments could use hierarchical context windows, repository embeddings, or dependency graph analysis to support multi-file refactorings, such as API unification or cross-module renaming.

In summary, this work demonstrates that LLMs are already capable of producing high-precision, maintainable, and behaviour-preserving refactorings,

especially when placed inside a carefully controlled multi-agent architecture. With enhanced planning, adaptive risk control, and trace-based semantic verification, such systems have the potential to mature into reliable autonomous contributors to real world software development workflows.