# UNIVERSIDADE DO OESTE DE SANTA CATARINA UNOESC – CAMPUS SÃO MIGUEL DO OESTE CURSO DE CIÊNCIA DA COMPUTAÇÃO

## RELATÓRIO REFERENTE AO COMPONENTE ENGENHARIA DE SOFTWARE II

**Fernando Camilo Schneider**
**Jean Carlo Toral**
**Jõao Ulisses Porto Alegre Ciriaco Teixeira**

**São Miguel do Oeste**
**2025**

# INTRODUÇÃO

Este relatório tem como objetivo apresentar e analisar os aspectos de modelagem de software, validação e testes, e o uso de padrões de projeto no sistema **Task Manager**. Desenvolvido no contexto da disciplina de Programação III e avaliado sob a ótica da Engenharia de Software II, este trabalho visa aplicar e demonstrar a integração dos conceitos aprendidos para o aprimoramento de um sistema real. Serão detalhados o Modelo Entidade-Relacionamento, os padrões de projeto empregados e as diferentes estratégias de teste que garantem a qualidade e robustez da aplicação.

# 1. MODELAGEM DE SOFTWARE

## 1.1. Diagrama de Caso de Uso

O Diagrama de Caso de Uso descreve as funcionalidades do sistema do ponto de vista do usuário. Ele foca em "o que" o sistema faz, sem detalhar "como" ele faz.

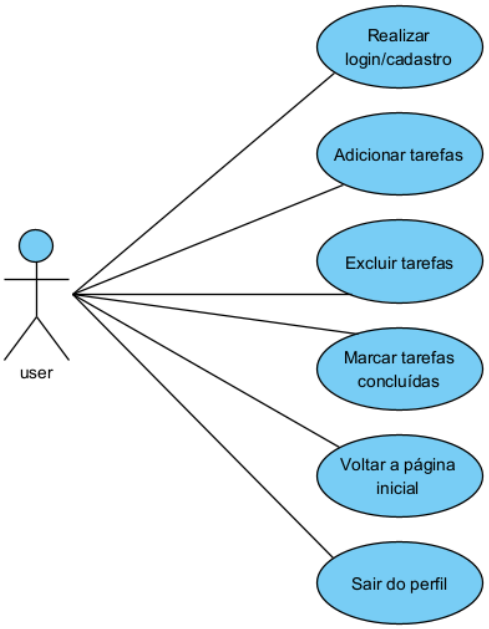


Figura 1: Diagrama de Caso de Uso

Como podemos ver na imagem usuário (ator) pode executar as seguintes

ações:

- Realizar login/cadastro: Acessar o sistema ou criar uma nova conta.
- Adicionar tarefas: Criar novos itens na sua lista de tarefas.
- Excluir tarefas: Remover tarefas existentes.
- Marcar tarefas concluídas: Alterar o status de uma tarefa para finalizada.
- Voltar à página inicial: Navegar para a tela principal do sistema.
- Sair do perfil: Encerrar a sessão de login (logout).

### 1.2 Diagrama de Sequência

O Diagrama de Sequência detalha o comportamento do sistema, mostrando a ordem cronológica das interações entre os diferentes componentes para realizar uma tarefa específica.

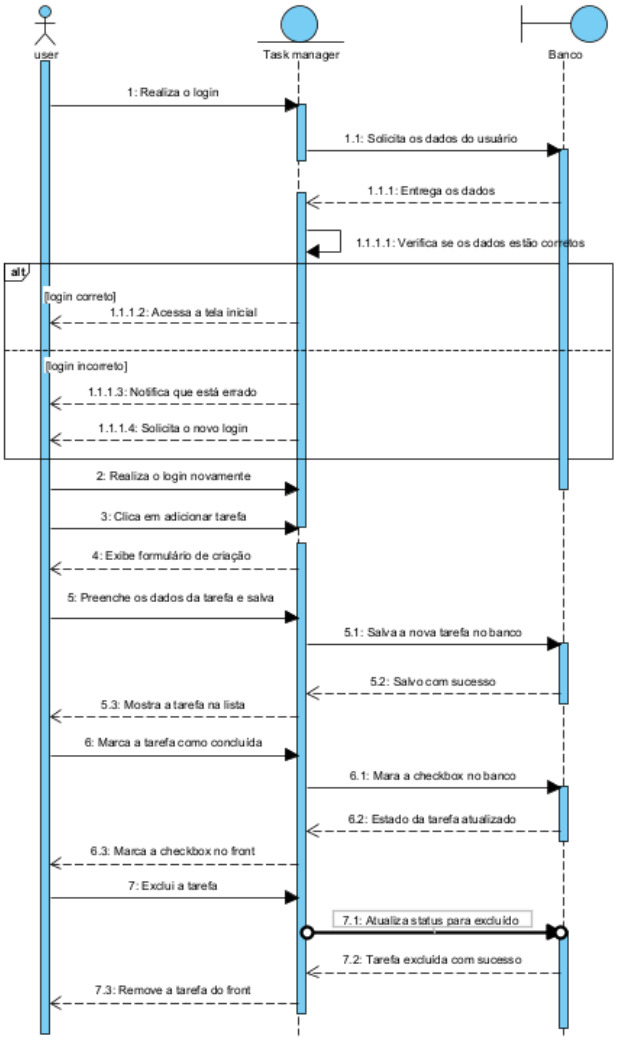


Figura 2: Diagrama de Sequência

Este diagrama ilustra os fluxos de interação entre três participantes: o "User" (usuário final), o "Task Manager" (a aplicação) e o "Banco" (o banco de dados). Os principais fluxos demonstrados são:

- Login: O usuário realiza o login, o sistema solicita os dados ao banco, verifica se estão corretos e, em caso de sucesso (login correto), acessa a tela inicial. Em caso de falha (login incorreto), notifica o erro ao usuário.
- Adicionar Tarefa: O usuário clica para adicionar, preenche o formulário e salva. O sistema envia os dados para serem salvos no banco e, após a confirmação, exibe a nova tarefa na lista para o usuário.
- Marcar Tarefa como Concluída: O usuário marca uma tarefa. O sistema atualiza o status no banco e reflete essa mudança na interface (front-end).
- Excluir Tarefa: O usuário exclui uma tarefa. O sistema executa uma atualização de status no banco (exclusão lógica) e, em seguida, remove o item da interface do usuário.

### 1.3 Diagrama de Classes

O Diagrama de Classes representa a estrutura estática do sistema em uma perspectiva orientada a objetos. Ele descreve as classes, seus atributos (dados) e métodos (comportamentos), bem como os relacionamentos entre elas.

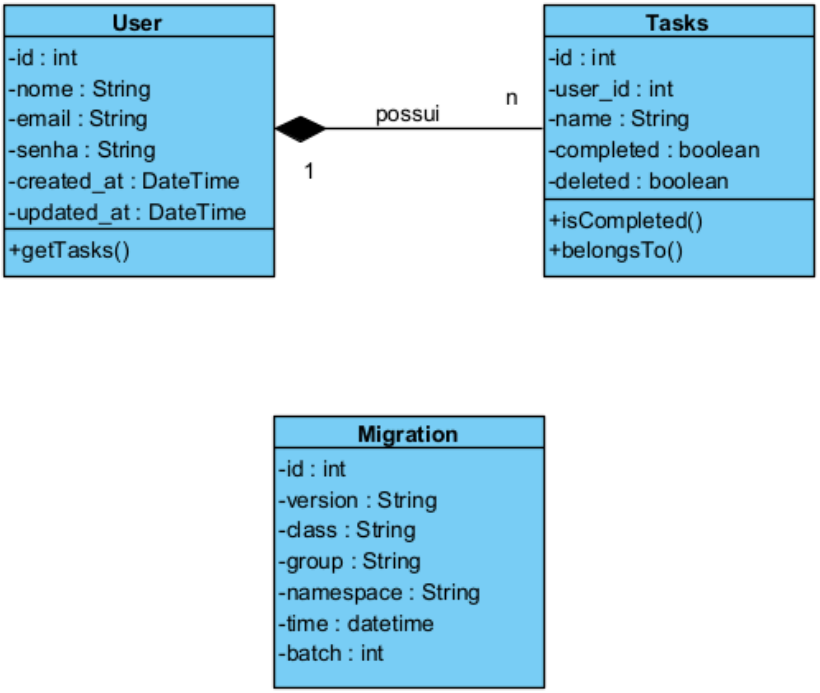


Figura 3: Diagrama de Classes

O diagrama apresenta três classes principais:

- User: Representa um usuário. Possui atributos como id, nome, email, senha e timestamps (created_at, updated_at). Contém o método getTasks() para obter as tarefas associadas a ele.
- Tasks: Representa uma tarefa. Possui atributos como id, user_id (para associar ao usuário), name, completed (booleano) e deleted (booleano). Inclui métodos como isCompleted() e belongsTo().
- Migration: Classe de controle do framework, usada para gerenciar as versões do banco de dados.

O relacionamento entre User e Tasks é de 1 para n (um para muitos), indicado pela linha que os conecta. Isso significa que um User pode possuir (possui) várias Tasks, mas cada Task pertence a apenas um User.

### 1.4 Diagrama Entidade-Relacionamento (ER)

O Diagrama Entidade-Relacionamento (ER) descreve a estrutura lógica do banco de dados. Ele mostra as tabelas, seus campos (colunas), os tipos de dados de cada campo e os relacionamentos entre as tabelas.

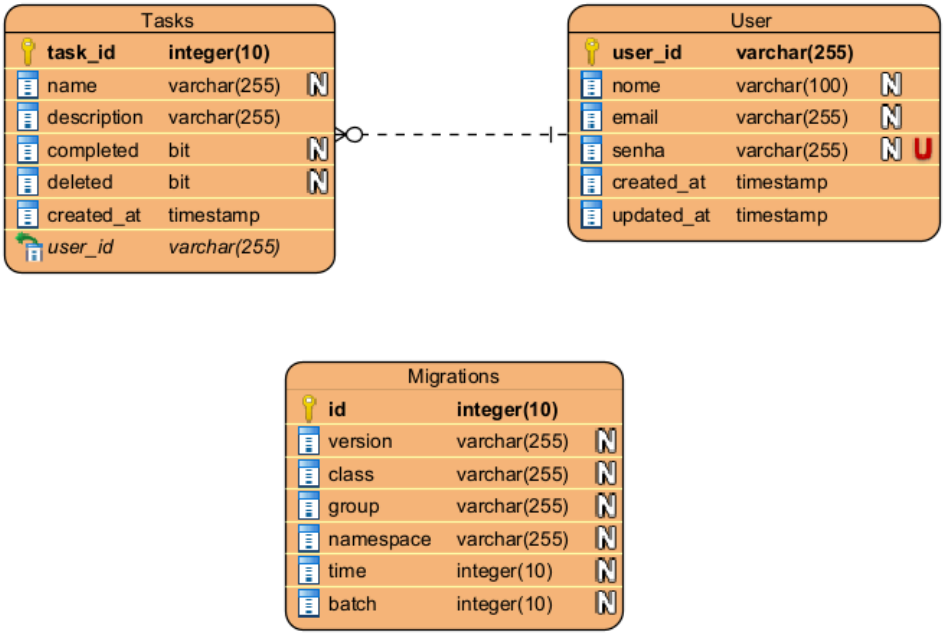


Figura 4: Diagrama Entidade Relacionameto

Este diagrama é a representação física do modelo de classes no banco de dados:

- Tabela User: Contém as colunas user_id (chave primária), nome, email, senha, created_at e updated_at. A coluna email possui uma restrição de unicidade (U), garantindo que não existam dois usuários com o mesmo email.
- 
- Tabela Tasks: Possui as colunas task_id (chave primária), name, description, completed, deleted, created_at e user_id.
- Relacionamento: A coluna user_id na tabela Tasks é uma chave estrangeira que se conecta à chave primária da tabela User. Isso implementa o relacionamento de "um para muitos", garantindo que cada tarefa no banco de

dados esteja obrigatoriamente ligada a um usuário existente.

- Tabela Migrations: Tabela auxiliar para controle de versionamento do banco de dados, padrão em frameworks como o CodeIgniter.

## 2. PADRÕES DE PROJETO

A arquitetura do **Task Manager** é solidamente fundamentada em padrões de projeto, o que confere ao sistema características essenciais como modularidade, manutenibilidade e escalabilidade. Os principais padrões identificados e suas aplicações são:

### 2.1. Model-View-Controller (MVC)

O padrão arquitetural MVC é a base do sistema, promovendo uma clara separação de responsabilidades:

- **Models**: Classes como TaskModel e UserModel encapsulam a lógica de negócios e gerenciam a interação com o banco de dados.
- **Views**: Arquivos PHP localizados em app/Views/ (e.g., tasks/Index.php, auth/login.php) são responsáveis pela apresentação dos dados ao usuário, contendo a estrutura HTML e a lógica de exibição.
- **Controllers**: Classes como Tasks e Auth atuam como intermediários, processando as requisições HTTP, interagindo com os Models para manipular os dados e selecionando as Views apropriadas para renderizar a resposta.

### 2.2. Front Controller

Todas as requisições HTTP direcionadas ao sistema são centralizadas em um único ponto de entrada: public/index.php. Este arquivo, em conjunto com as regras de roteamento definidas em app/Config/Routes.php, é responsável por direcionar cada requisição para o Controller e método adequados. Essa abordagem centraliza o controle do fluxo da aplicação e simplifica o gerenciamento de requisições.

### 2.3. Active Record / Repository

As classes TaskModel e UserModel estendem CodeIgniter\Model, o que as posiciona como uma camada de abstração para o acesso a dados. Essa extensão permite que os objetos do modelo incorporem funcionalidades para interagir diretamente com o banco de dados (e.g., findAll(), findById(), insert(), update()), caracterizando o padrão Active Record. Essa abordagem simplifica as operações CRUD e mantém a lógica de persistência próxima à representação dos dados, muitas vezes combinando-se com princípios do padrão Repository.

### 2.4. Filter (ou Intercepting Filter)

O sistema emprega filtros para aplicar lógica transversal às requisições, antes ou depois do processamento principal. Exemplos notáveis incluem:

- AuthFilter.php: Garante que apenas usuários autenticados possam acessar rotas específicas (e.g., /tasks, /tasks/*), implementando um controle de acesso robusto.
- PerformanceFilter.php: Adiciona cabeçalhos de controle de cache e otimizações de compressão (gzip) para melhorar o desempenho da aplicação, além de cabeçalhos de segurança (X-Content-Type-Options, X-Frame-Options, X-XSS-Protection).

Esses filtros executam a lógica nos métodos before() e after() da requisição, separando efetivamente as preocupações não relacionadas à lógica de negócios principal.

### 2.5. Dependency Injection (DI) e Service Locator

- **Injeção de Dependência (DI)**: Os Controllers (e.g., Tasks.php, Auth.php) recebem suas dependências (como instâncias de TaskModel e UserModel) diretamente em seus construtores. Essa prática promove o desacoplamento entre as classes, tornando-as mais independentes, flexíveis e, consequentemente, mais fáceis de testar.
- **Service Locator**: O CodeIgniter utiliza um Service Locator, exemplificado pela

classe Config/Services.php. Este padrão permite que os componentes do sistema solicitem instâncias de serviços (e.g., service('request'), service('response'), service('toolbar')) de um registro central, sem a necessidade de conhecer os detalhes de sua criação. Isso contribui para um código mais modular e facilita a substituição de implementações de serviço.

## 3. VALIDAÇÃO E TESTES

A etapa de validação e testes foi fundamental para assegurar a qualidade, a robustez e o desempenho do sistema Task Manager. Foram aplicadas diferentes categorias de teste para cobrir desde os componentes individuais até o comportamento do sistema como um todo sob carga.

### 3.1. Teste Unitário

Foram implementados testes unitários utilizando o PHPUnit para verificar o comportamento isolado de componentes críticos do sistema. O foco foi garantir que as menores partes do código funcionassem corretamente.

- Teste de Models:
    - TaskModelTest.php: Validou as regras de negócio do TaskModel, como os comprimentos mínimo e máximo para o nome da tarefa e a integridade dos campos completed e deleted.
    - BasicTaskTest.php e SimpleTaskTest.php: Verificaram a estrutura de dados das tarefas, a lógica de validação e a correta alternância do status de conclusão.
- Teste de Controllers:
    - TasksControllerTest.php: Assegurou que o TasksController é instanciado corretamente e que seus métodos públicos essenciais (index, list, add, delete, toggle) estão visíveis e funcionais, garantindo a integridade da interface do controlador.

### 3.2. Teste de Funcionalidade (End-to-End)

Os testes de funcionalidade foram realizados para validar se os fluxos de trabalho completos correspondiam aos requisitos do sistema. A ferramenta Katalon Recorder foi utilizada para automatizar e simular as interações do usuário de ponta a ponta.

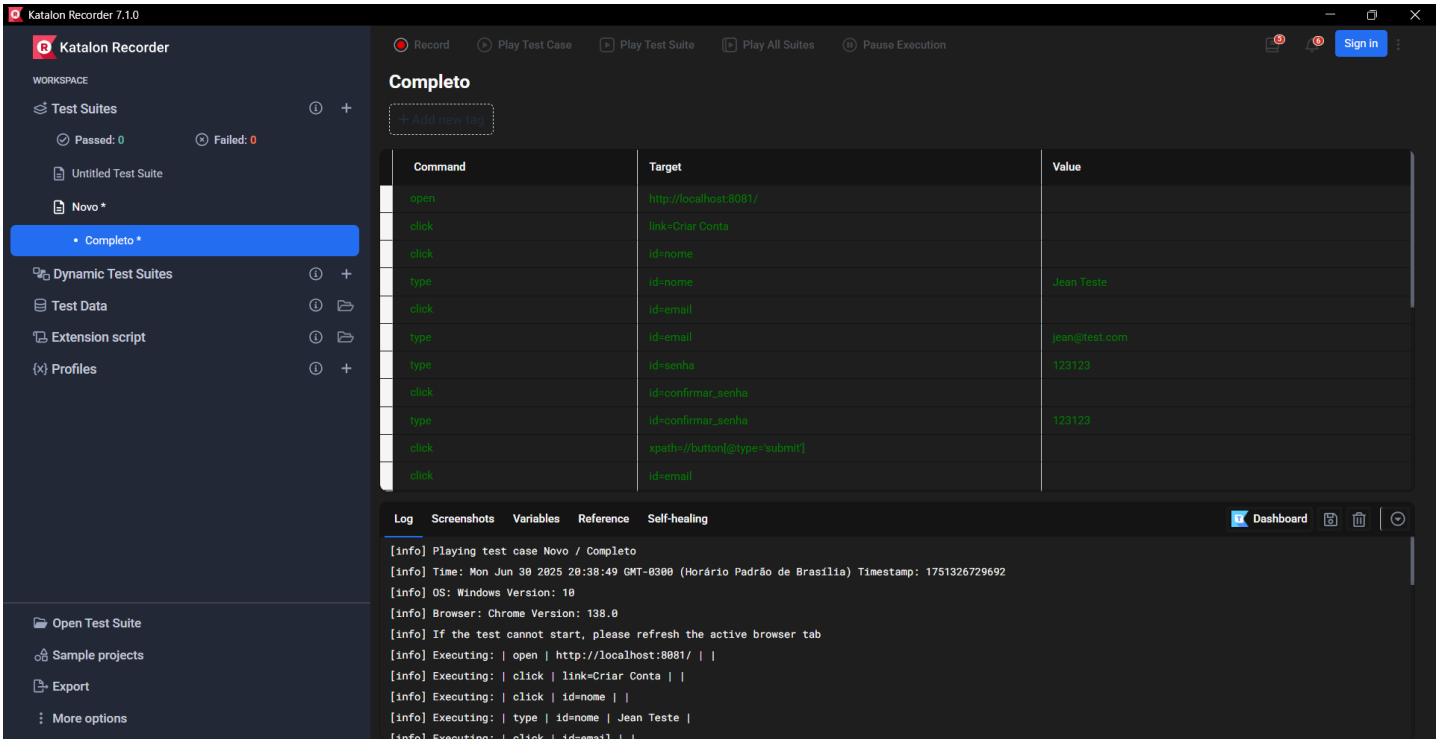


Figura 5: Script de teste automatizado no Katalon Recorder para o fluxo de cadastro.

A imagem acima exemplifica um dos cenários de teste, onde o Katalon executa uma sequência de comandos (open, click, type) para simular o preenchimento de um formulário. Cenários como este foram criados para validar as principais funcionalidades da aplicação, incluindo:

- Criação de novas tarefas.
- Marcação e desmarcação de tarefas (alternando o status completed).
- Verificação da exclusão de tarefas.

Esses testes garantiram que a jornada do usuário, do início ao fim, funcionasse conforme o esperado, validando a integração entre a interface e a lógica de negócio.

### 3.3. Teste de Carga

Para avaliar o desempenho e a estabilidade do sistema sob estresse, foram conduzidos testes de carga com o Apache JMeter. O objetivo era entender como a aplicação se comportava com múltiplos acessos simultâneos.

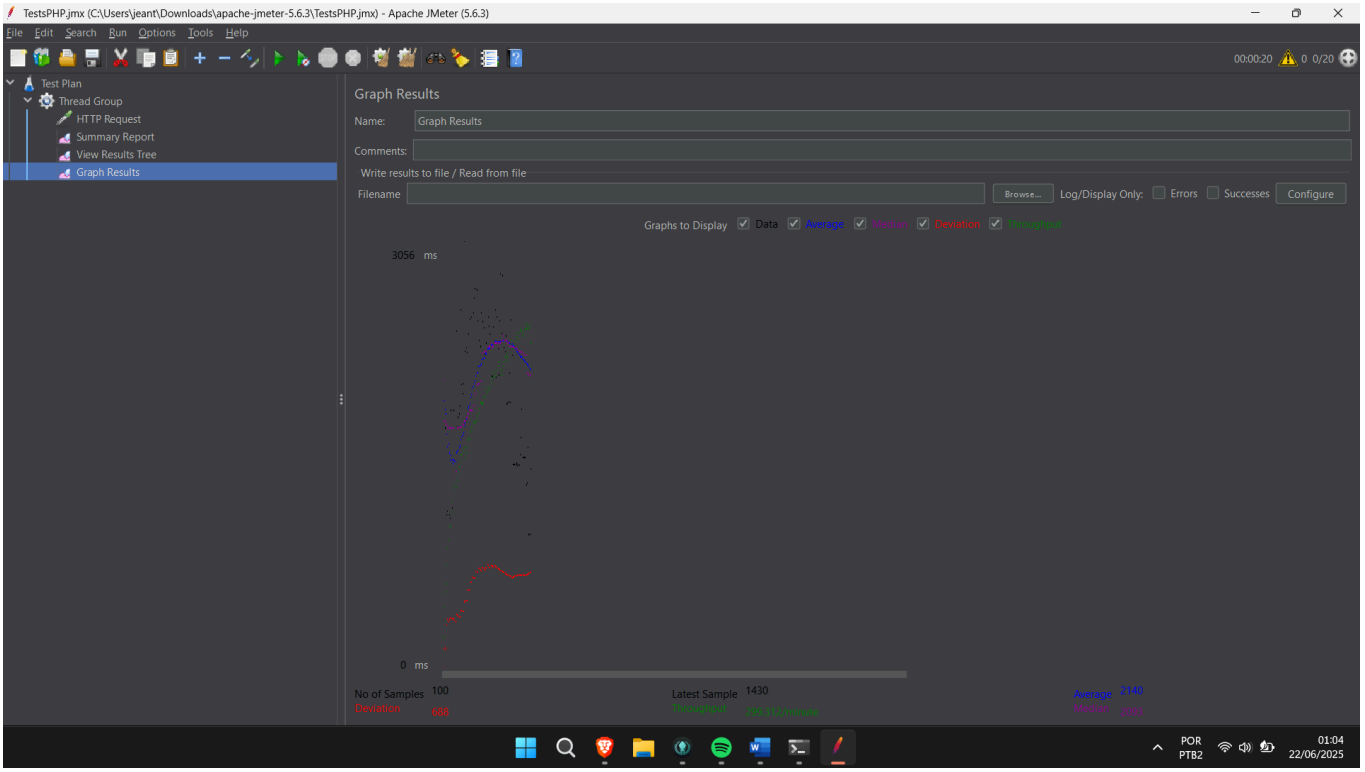


Figura 6: Gráfico de resultados do JMeter mostrando os tempos de resposta durante o teste de carga.

O teste simulou a atividade de múltiplos usuários realizando requisições ao sistema, totalizando 100 amostras (No of Samples), como pode ser visto no canto inferior esquerdo da Figura 2. Os resultados revelaram gargalos de performance, com os principais indicadores sendo:

- Mediana (Median): 2003 ms, indicando que metade das requisições levou mais de 2 segundos para ser processada.
- Desvio Padrão (Deviation): 2747 ms, um valor alto que aponta uma grande inconsistência nos tempos de resposta.

O gráfico demonstra visualmente essa lentidão. A análise foi crucial para confirmar que, sob carga, o sistema apresenta lentidão, fornecendo dados concretos

que direcionam a necessidade de otimizações no código, consultas ao banco de dados e na infraestrutura para garantir a escalabilidade da aplicação.

## CONCLUSÃO

A execução do projeto **Task Manager** demonstrou a aplicação prática e integrada de conceitos fundamentais de Engenharia de Software. A modelagem relacional do banco de dados estabeleceu uma estrutura de dados consistente e eficiente. A adoção estratégica de padrões de projeto como MVC, Front Controller, Active Record/Repository, Filter e Dependency Injection promoveu a modularidade, reusabilidade e manutenibilidade do código-base. Por fim, a estratégia de testes abrangente, que incluiu testes unitários, de funcionalidade e de carga, foi essencial para validar o comportamento do sistema, identificar falhas e apontar oportunidades claras de melhoria de desempenho. A sinergia entre essas práticas resultou em um sistema mais robusto, bem estruturado e preparado para futuras evoluções.