

# **Laboratório de Desenvolvimento de Software**

Plano de Projeto v2.0

Grupo 2

## Conteúdo

1.	Elementos de Referência .....	3
1.1.	Objetivo do Projeto .....	3
1.2.	Dados Gerais da Equipa de Desenvolvimento: .....	4
1.3.	Regulamento interno .....	5
1.3.1	Funcionamento Interno da Equipa .....	5
1.3.2	Normas de Desenvolvimento e de Controlo de Versões .....	7
1.4.	Regras de avaliação interna.....	16
1.5.	Modelos .....	18
2.	Registos de Execução .....	18
3.	Auditorias e Reviews.....	19

## 1. Elementos de Referência

### 1.1. Objetivo do Projeto

O principal objetivo deste projeto é o de alicerçar a plataforma **SeePaw** através de um projeto de **backend**, uma **aplicação web** e uma **aplicação android**.

O backend trata da lógica de negócio e da interação com a base de dados. Expõe um conjunto de endpoints, que permitem o acesso e manipulação de informação para as aplicações.

A aplicação web é a interface principal dos utilizadores finais, que podem ser pessoas interessadas em apadrinhar ou adotar um animal, ou gestores de centros de acolhimento animal. É através desta aplicação que são inseridos os animais com toda a sua informação relevante, são feitos os pedidos de *fostering* e de *ownership*, e agendados os pedidos de visita ou de recolha de animais.

Por fim a aplicação android, permite que utilizadores que procuram apadrinhar ou adotar um animal, que possam fazer os pedidos de *fostering* e *ownership* e agendar visitas ou recolha de animais.

## 1.2. Dados Gerais da Equipa de Desenvolvimento:

Ana Sofia Vaz (8230095@estg.ipp.pt) – *Software Engineer e Team Leader*

Cristiana Pinheiro (8230091@estg.ipp.pt) – *Software Engineer*

Joel Pinto (8230098@estg.ipp.pt) – *Software Engineer*

Rui Araújo (8230112@estg.ipp.pt) – *Software Engineer*

### **Papel do *Team Leader***

- Coordenar o trabalho da equipa e garantir o cumprimento do plano do projeto.
- Atribuir tarefas e apoiar a resolução de constrangimentos.
- Enviar convocações para as reuniões e preparar ordem de trabalho das mesmas
- Assegurar a qualidade do código e o cumprimento das boas práticas.

### **Papel do *Software Engineer***

- Implementar as funcionalidades atribuídas de acordo com os requisitos definidos.
- Escrever código limpo, testável e alinhado com as boas práticas da equipa.
- Colaborar na revisão de código e na resolução de problemas técnicos.
- Participar nas reuniões de equipa e contribuir para a melhoria contínua do processo.

### 1.3. Regulamento interno

#### 1.3.1 Funcionamento Interno da Equipa

- a. **Na ausência do *Team Leader***, por questões pessoais ou profissionais, este cargo será assegurado temporariamente por Cristiana Pinheiro.
- b. **Reuniões semanais (sprint meeting)**: será agendada semanalmente pelo menos uma reunião semanal, em que as convocatórias (ver **modelo A**) para as mesmas serão enviadas com 48h de antecedência. A quantidade de reuniões semanais será de acordo com as necessidades de formalização de decisões da equipa, mas o previsto é o de uma reunião semanal.
- c. As reuniões devem ser em formato presencial ou online.
- d. As reuniões online deverão ser feitas usando a plataforma Teams.
- e. **A convocatória para as reuniões** será elaborada pelo(a) *Team Leader* e enviada por *email* institucional dos restantes membros da equipa. Posteriormente será arquivada no github no repositório “**project-artifacts**”.
- f. As atas de reunião (**modelo B**) são preenchidas pelo(a) *Team Leader* e assinadas por todos os membros da equipa. As atas são homologadas até 72h após a reunião.
- g. As atas por assinar serão colocadas no Teams e quando todos os membros da equipa tiverem assinado a mesma, esta será arquivada no github no repositório “**project-artifacts**”.
- h. Os documentos docx/excel/etc, editáveis serão colocados numa pasta partilhada com a equipa (OneDrive) denominada “LDS”. Quando estes forem validados por toda a equipa serão arquivados no github no repositório “**project-artifacts**”.
- i. A avaliação de cada membro da equipa deve ser feita em todas as milestones.
- j. A falta a uma reunião ou não cumprimento das datas de entrega das tarefas, sem uma justificação válida, serão penalizadas com uma infração de 5% na nota final dessa milestone.
- k. O repositório dos projetos (separador **Repositories**) pertencem à organização SeePaw do Github (<https://github.com/see-paw>), bem como a gestão do *Product Backlog* (separador **Projects/SeePaw Development 2025**) e armazenamento dos diferentes artefactos. O projeto de backend encontra-se no repositório “**backend**”, o projeto da web app encontra-se no repositório “**web-app**”, o projeto de android encontrar-se no repositório de “**android-app**” e por fim todos os artefactos do projeto homologados encontram-se em “**project-artifacts**”.
- l. A plataforma de comunicação do quotidiano entre os membros da equipa será feita no Whatsapp num grupo criado para o efeito.
- m. Toda a discussão pertinente a uma funcionalidade, dependência, tecnologia ou decisão arquitetural que surgir depois do início de cada sprint, deverá ser feita na issue que a originou através de comentários breves e detalhados (com recurso a screenshots ou imagens se necessário). Desta forma, toda a informação, fluxo de

informação e ideias, e toda a discussão relevante passa a ser centralizada e pode mais tarde ser compreendida por todos os membros da equipa, sendo também importante na tomada de decisão pelos membros da equipa.

- n. Em caso de conflitos de merge (num pull request) ou de problemas técnicos que impeçam o progresso de uma issue, o membro da equipa deve comunicar imediatamente ao Team Leader e à equipa através do canal apropriado (comentário na issue ou Whatsapp). A resolução deve ser priorizada, para não bloquear outras tarefas dependentes.
- o. Todos os membros da equipa devem estar disponíveis para responder a mensagens relacionadas com o projeto num prazo máximo de 24h em dias úteis. Ausências superiores a 1 dia devem ser comunicadas antecipadamente ao Team Leader.

### 1.3.2 Normas de Desenvolvimento e de Controlo de Versões

#### p. Branches:

- Cada repositório contará com duas branches principais, a branch *main/master* que guarda o histórico oficial das releases, enquanto que a branch *develop* serve como branch de integração para novas funcionalidades; a branch *develop* é inicialmente criada a partir da *main*.
- Cada nova funcionalidade deve ser criada na sua própria branch, ramificada a partir da branch *develop*.
- A convenção adotada para a criação de branches é a de *conventional branching*. Deve ser estruturada por: **<tipo>/<descrição>** (ex.: **feature**/jwt-login, **feature**/animal-crud, **bugfix**/animal-not-found, **chore**/update-dependencies, **refactor**/user-utils, etc). Uso de descrições em minúsculas, números 0-9, separadas por hífen. Para informações detalhadas: <https://conventional-branch.github.io/>
- Quando a funcionalidade estiver concluída, deve ser integrada na branch *develop* através de um Pull Request.
- As correções de erros também são tratadas em branches dedicadas para o efeito e podem ser criadas a partir de *develop* ou de *main* (no caso de ser um problema crítico).

#### q. Commits:

- Os membros da equipa devem fazer *commits* regularmente com mensagens significativas. Cada *commit* deve tratar de um único objetivo, para melhorar a rastreabilidade e permitir um *rollback* fácil, se necessário.
- É proibido fazer *push* do *commits* diretamente para as *branches main* e *dev*.
- A convenção adotada para *commits* é a de *conventional commits* (<https://www.conventionalcommits.org/en/v1.0.0/>). Podem ser do tipo **feat** (feature), **refactor**, **fix** (bug fix), **chore** (atualização de dependências ou outras tarefas), **revert** (reversão de código), **docs** (inclusão ou ajustes de documentação), **ci** (mudanças de CI/CD). Pode ser usado o ponto de exclamação “!” para frisar a alteração de mudanças que **quebram código existente**. Devem ser estruturados da seguinte forma:  
**<tipo>(âmbito opcional): <descrição>**  
[corpo opcional]  
[footer opcional]

#### Exemplos:

revert: let us never again speak of the noodle incident,

feat(auth): add new user authentication system,  
fix!: change API response format,  
docs: correct spelling of CHANGELOG,

**Com body e footer:**

feat: add user authentication system  
Implement JWT OAuth2 auth flow with Google and GitHub providers.  
Closes Issue#255

**r. Pull Request:**

- Os programadores submetem *merge/pull requests* para as branches *develop* ou *main*. É obrigatório fazer um *pull request* antes de qualquer *merge* para a *main* ou *dev*. Cada pedido deve incluir uma descrição detalhada de todas as alterações realizadas.
- **O pull request deve ser aberto tão cedo quanto possível**, isto é, logo que a *issue* correspondente esteja ***in progress*** na ***project board***. Deve ser criado em modo **draft pull request**, para que seja possível seguir as mudanças feitas ao código com maior facilidade, pelos commits enviados para a branch em que está a ser desenvolvida a nova funcionalidade.
- O **título do pull request (PR)** deve seguir o formato dos conventional commits, **e referenciar a issue (user story) que deu origem a esse PR**. O PR mantém todos os commits individuais da branch em que está a ser desenvolvida a nova feature, ao qual é adicionado um commit de merge adicional, cuja mensagem desse commit é o **título** desse PR.  
**Exemplo:** feat: implement user authentication system (#45).  
É importante que a *issue* correspondente seja mencionada no título do PR, para que seja mais fácil fazer a associação/tracking no github.
- Os *merge requests* para as branches *develop* ou *main* exigem revisão e aceitação por, pelo menos, **dois outros membros da equipa**, excluindo o autor do pedido. Os *reviewers* em questão têm que ser adicionados ao pedido de PR. O *assignee* é o autor do pull request.
- **A label** do PR deve ser de acordo com o tipo de pull request em questão (feature, bug, etc).
- Só depois de passarem com sucesso **todas as pipelines** na branch de funcionalidade ou na *develop* é que o *merge* pode ser aprovado pelo reviewer para ser incorporada na *develop* ou *main*, respetivamente.  
**Fica a cargo do autor do PR** resolver quaisquer conflitos/problemas que falhem na pipeline de CI/CD.
- O código deve passar nos **testes unitários** dentro da branch de funcionalidade antes de ser integrado na branch *develop*.
- O código deve passar nos **testes de integração** na branch *develop* antes de ser integrado na *main*.



- As branches *main* e *develop* estão protegidas e não permitem *commits* diretos. Todas as alterações devem passar por um *Pull Request* aprovado. **É da responsabilidade de cada membro fazer backup regular do seu trabalho através de commits frequentes na sua branch de desenvolvimento.**

**s. Continuous Integration:**

- A *pipeline* de **CI/CD do Github** compila e testa o código em cada *pull request* para as branches *develop* ou *main*, de forma a garantir que as novas alterações não quebram funcionalidades existentes. Quaisquer erros ou conflitos devem ser resolvidos por quem inicia o *pull request*.
- Durante cada integração são executados testes automatizados incluindo testes unitários, testes de integração (se for o caso) e análise de qualidade do código.

**t. Issues (User Stories):**

- A US tem um número único no repositório/projeto sobre o qual é criada e um nome descritivo. Quando é criada, em primeiro lugar deve ser escolhido o repositório/projeto ao qual a *issue* se refere. Posteriormente, a *User Story* é detalhada. É estimada e priorizada em cada Sprint Review. Nesta fase a *issue* fica na board **“Backlog”**.
- A US deve ter critérios de aceitação que definem quando a mesma está concluída
- Apenas quando a *User Story* avança para a fase **“Ready”** é que está realmente pronta para ser atribuída à **iteração atual (sprint)** e ao **Developer**.
- De seguida, a *User Story* passa para **“In Progress”**, o que significa que está a ser trabalhada.
- Quando o desenvolvimento está completo e o *Pull Request* está aberto e pronto para *merge* (sem problemas na pipeline), a *Issue* transita para **“In Review”**.
- Quando todas as tarefas forem concluídas, a *Issue* transita para a board de **“Done”**, o que corresponde a uma *Issue* que está associada a um *Pull Request* bem sucedido do sprint decorrente.
- No início de cada sprint as *Issues* do sprint anterior passam para a board de **“Archive”**, onde são agrupadas todas as **Issues** anteriores de sprints passados.
- **É da responsabilidade do Developer** a quem foi atribuída a tarefa de a fazer transitar para as *boards* seguintes. Este processo facilita o planeamento semanal nas reuniões de equipa.
- **Criação de Sub-issue a partir de um Epic existente:**

1. Abrir a Development Board (see-paw/Projects/SeePaw Development 2025)
2. Remover o "-status:Epics" da barra de pesquisa
3. Selecionar o Epic relevante do qual a issue será associada
4. Pressionar "Create sub-issue", criar "Blank issue"
5. Preencher com título e descrição apropriados: curto e descritivo
6. Atribuir as labels corretas (bug, feature, chore, etc.) e prioridade
7. Adicionar assignees e reviewers
8. Abrir um draft pull request com o número da issue na sua descrição
9. Preencher a issue usando o template/diretrizes o

**Criação de Issue que não pertence a nenhum Epic existente:**

1. Abrir a Development Board (see-paw/Projects/SeePaw Development 2025)
2. Clicar em "Add Item" na board Backlog
3. Escolher o repositório relacionado com a funcionalidade a desenvolver 4. Clicar no texto e clicar em "Create new issue"
5. Preencher com título e descrição apropriados: curto e descritivo
6. Atribuir as labels corretas (bug, feature, chore, etc.) e prioridade
7. Adicionar assignees e reviewers
8. Preencher a issue usando o template/diretrizes

○ **Template/Diretrizes para Issues/User Stories:**

○ **Descrição do Problema:**

- - Qual é o problema principal que esta issue procura resolver
- - Adicionar uma descrição sintetizada mas compreensiva (que pode incluir screenshots ou outros tipos de imagens)
- - Referenciar discussões importantes relacionadas com esta issue/user story, com datas de reuniões se aplicável
- - Referenciar requisitos relacionados na Wiki do project-artifacts. Como referenciar Requisitos Relacionados:

**Referenciar os requisitos funcionais e não funcionais da Wiki** que estão dentro do âmbito desta User Story:

- [Requisito Funcional RFX – Nome do Requisito]([https://github.com/see-paw/project-artifacts/wiki/2-Requirements#REQUIREMENT\\_ID](https://github.com/see-paw/project-artifacts/wiki/2-Requirements#REQUIREMENT_ID))
- Se for relevante: [Requisito Não-Funcional RNFX – Nome do Requisito]([https://github.com/see-paw/project-artifacts/wiki/2-Requirements#REQUIREMENT\\_ID](https://github.com/see-paw/project-artifacts/wiki/2-Requirements#REQUIREMENT_ID))
- Assegurar que todos os requisitos ligados estão claramente identificados e alinhados com o propósito da story

**CrITÉrios de Aceitação:** Adicionar critérios de aceitação relevantes

**Notas Adicionais:**

- Adicionar contexto, diagramas, mockups, ou qualquer informação relevante, se aplicável
- Mencionar dependências com outras stories ou tarefas técnicas se aplicável
- Incluir referências a documentos de design ou discussões se necessário

**Responsabilidades do Developer** quando encontra um problema ou requisito em falta:

- (No caso de ser 1 requisito novo) Se o problema está relacionado com um novo requisito: Criar o requisito na respetiva secção de aprovação pendente e notificar o Team Leader
- (No caso de ser 1 requisito novo) Após aprovação, movê-lo para a secção aprovada
- Criar ou atualizar uma issue no repositório correto
- Atribuir as labels corretas (bug, feature, chore, etc.) e prioridade
- Referenciar a issue em commits e PRs usando #<issue\_number>
- Seguir a convenção de nomes de branches (feature/..., bugfix/...)
- Atualizar o estado da issue conforme o progresso o Todas as alterações devem ser rastreáveis através de: **issue → commit → PR → merge**

**u. Issues (Epics):**

- Os *Epics* representam funcionalidades ou objetivos de alto nível que englobam múltiplas User Stories relacionadas. Proporcionam uma visão estratégica do progresso do projeto e ajudam a organizar o desenvolvimento em torno de funcionalidades principais.
- Uma *Epic* tem um número único e um título descritivo que representa uma funcionalidade ou objetivo de alto nível. Cada issue que representa uma *Epic* deve ter obrigatoriamente a tag “**Epic**”.

- As **Epics** estão numa *board* específica (*Epics*), que por defeito não aparecem na **Development Board** da organização, através de um filtro na barra de pesquisa. Do lado esquerdo da *Development Board* é possível ter uma planificação visual da percentagem de conclusão de uma determinada *Epic*.
- O estado de uma *Epic* é automaticamente calculado com base no progresso das User Stories associadas (ex: 3/5 User Stories concluídas = 60%)
- **Cada Epic é uma parent issue**, e todas as *issues* (*user stories*) associadas a esse Epic são sub-issues dessa *parent issue*.
- As *Epics* devem ter critérios de aceitação ao nível macro que definem quando a funcionalidade completa está pronta para produção. Neste projeto os critérios macro são **a conclusão de todas as sub-issues associadas ao Epic**.
- O ciclo de vida das *Epics* na Development Board é de: *Epics* -> *Done* -> *Archive*
- Quando todas as *issues* desse *Epic* estão concluídas, essa *Epic* é movida para a board “*Done*” e posteriormente arquivada em “*Archive*”.
- No planeamento de *sprint*, as User Stories são selecionadas tendo em consideração o progresso e prioridade das *Epics* parentais.
- Criação de um *Epic*:
  1. Abrir a *Development Board* (see-paw/Projects/SeePaw Development 2025)
  2. Clicar em "Add Item" na board *Epics*
  3. Escolher o repositório relevante. Neste projeto o repositório padrão para a criação de *Epics* é o “backend”. Não tem um significado particular neste projeto, já que as *Epics* do projeto têm um âmbito que abrange vários repositórios.
  4. Criar uma nova *issue* com um título descritivo
  5. Adicionar a label *Epic*
  6. Preencher o template do *Epic* seguindo as diretrizes:
    - Breve descrição da funcionalidade ou objetivo de alto nível que este *Epic* representa
    - Requisitos Relacionados: Listar os requisitos funcionais e não-funcionais relevantes da Wiki do projeto através de um link específico para isso [ID do Requisito–Nome](https://github.com/see-paw/project-artifacts/wiki/link-para-requisito))
    - Critérios de Aceitação Macro: Lista de condições que definem quando a funcionalidade completa está pronta para produção

- User Stories: Esta secção será automaticamente preenchida à medida que as sub-issues são criadas
- Considerações Técnicas: Informações sobre arquitetura, dependências ou limitações técnicas

**v. Priorização das US/Issues:**

- A priorização é definida durante as reuniões de planeamento de sprint. Mediante a pertinência da funcionalidade, ou da importância para o avanço do projeto, é atribuída um grau de importância à issue que simboliza a prioridade com que esta tem que ser tratada.
- As de prioridade **critical** são as que têm que ser resolvidas com o máximo de urgência, pois são problemas que afetam ou que bloqueiam o desenvolvimento e/ou progresso de outras Issues.
- Prioridade 1 – Low (baixa prioridade)
- Prioridade 2 – Medium (prioridade média)
- Prioridade 3 – High (alta prioridade)
- Prioridade 4 – Critical (crítica/bloqueante)

**w. Estimativa das US:**

- São acordadas em cada reunião de sprint. Cada membro da equipa vota (XS, S, M, L) e revelam as estimativas ao mesmo tempo. Discutem discrepâncias na votação e fazem uma nova votação até haver um consenso. Os critérios de estimativa podem ser de complexidade técnica, quantidade de testes a produzir, documentação requerida, tipo de dependências externas ou tecnologias a adotar (especialmente no caso de ser uma tecnologia que ainda não foi devidamente explorada).
- A correspondência entre o tamanho estimado e os **Story Points** é a seguinte:
  - XS: (cerca de 1 hora) → **1 Story Point**
  - S: (até 1 dia) → **2 Story Points**
  - M: (até 3 dias) → **3 Story Points**
  - L: (até 1 semana) → **4 Story Points**
- No final de cada sprint, é feita a soma total dos Story Points concluídos, sendo esta informação representada graficamente no **Burndown Chart**, permitindo avaliar o progresso da equipa e a capacidade de entrega por sprint.

**x. Definição de tarefas de cada U.S./Issue:**

- Cada tarefa deve ser pequena (idealmente < 1 dia de trabalho). Salvo exceções que o justifiquem.
- Deve ter um propósito claro e critérios de conclusão bem definidos (ver secção o).

**y. Atribuição de US/Issues:**

Após a atribuição, deve ser registado na respetiva Issue:

- *Assignee* (membro responsável)
- Prazo de conclusão esperado
- Comentário com contexto adicional, se necessário. **Mudanças de atribuição**

só podem ser feitas mediante discussão em reunião ou aprovação do *Team Leader*, sendo sempre documentadas na Issue correspondente.

## z. Semantic Versioning

Cada repositório do github tem configurado um sistema automático de *semantic versioning*. É automatizado através de um github workflow (.github/workflows/release.yml), que gere este sistema automaticamente com base nos commits realizados, daí a importância do uso do sistema de **conventional commits**.

O sistema adotado é um dos padrões mais utilizados na indústria de incremento de versões de software: **MAJOR.MINOR.PATCH**. (<https://semver.org/>)

- **MAJOR** são alterações que mudam/quebram a compatibilidade do código (feat!., fix!., etc);
- **MINOR** representam novas funcionalidades compatíveis (ex.: feat:);
- **PATCH** está associado a correções e atualizações menores (ex.: fix:, docs:, chore:);

Ex.:

**MINOR** increment: 2.1.4 → 2.2.0, neste caso faz-se um reset ao número correspondente ao PATCH, que fica a 0.

**MAJOR** increment: 2.1.4 → 3.0.0, neste caso faz-se um reset aos números correspondentes ao de **PATCH** e de **MINOR**, ambos ficam a 0.

**PATCH** increment: 1.2.3 → 1.2.4, apenas é feito o incremento no último número, por exemplo um *bugfix* ou uma atualização de documentação.

## Criação de *Rules* no GitHub:

São definidas as seguintes regras:

- É obrigatório um *pull request* antes de um *merge* para a *main* ou *dev*
- É necessária a aprovação de dois elementos da equipa desse *pull request*, que não o próprio que o pediu
- É proibido fazer *push* diretamente para a *main* ou *dev*

As *guidelines* para a criação destas regras são as seguintes:

- 1) Entrar no repositório → “Settings” → “Rules” → “Rulesets” → “New rule set” → atribuir um nome à regra → “Add target” → “Include by pattern” (criar para a main e para a dev)
- 2) Em seguida escolher as seguintes regras:
  - “Require a pull request before merging” com “Required approval” = 2
  - “Require approval of the most recent reviewable push”
  - “Block force pushes”
  - “Restrict deletions”
- 3) Por fim carregar em “Create”

The screenshot shows the GitHub 'Rulesets / New branch ruleset' configuration page. On the left is a sidebar with navigation links: General, Access, Collaborators and teams, Code and automation (with sub-links for Rulesets, Insights, Actions, Models, Webhooks, Copilot, Environments, Pages, and Custom properties), Security (with sub-links for Advanced Security, Deploy keys, and Secrets and variables), and Integrations (with sub-links for GitHub Apps and Email notifications). The 'Rulesets' link is highlighted. The main content area is titled 'Rulesets / New branch ruleset' and contains several sections: a warning box about protecting important branches, a 'Ruleset Name' field with the value 'Protected branches - main & dev', an 'Enforcement status' dropdown set to 'Disabled', a 'Bypass list' section with an 'Add bypass' button, and a 'Target branches' section with a table of branch targeting criteria. The table has two rows: 'main' and 'dev', each with a green circle icon and a trash icon. An 'Add target' button is at the top right of the table.

Rulesets / New branch ruleset

**Protect your most important branches**

Rulesets define whether collaborators can delete or force push and set requirements for any pushes, such as passing status checks or a linear commit history.

Your rulesets won't be enforced on this private repository until you [upgrade this organization account to GitHub Team](#).

Ruleset Name \*

Protected branches - main & dev

Enforcement status

Disabled

Bypass list

Exempt roles, teams, and apps from this ruleset by adding them to the bypass list.

+ Add bypass

Bypass list is empty

Target branches

Which branches should be matched?

Branch targeting criteria	
main	🗑️
dev	🗑️

Add target

**Rules**  
Which rules should be applied?

**Branch rules**

☐ Restrict creations  
Only allow users with bypass permission to create matching refs.

☐ Restrict updates  
Only allow users with bypass permission to update matching refs.

☒ Restrict deletions  
Only allow users with bypass permissions to delete matching refs.

☐ Require linear history  
Prevent merge commits from being pushed to matching refs.

☐ Require deployments to succeed  
Choose which environments must be successfully deployed to before refs can be pushed into a ref that matches this rule.

☐ Require signed commits  
Commits pushed to matching refs must have verified signatures.

☒ Require a pull request before merging  
Require all commits be made to a non-target branch and submitted via a pull request before they can be merged.

Hide additional settings ^

**Required approvals**

2

The number of approving reviews that are required before a pull request can be merged.

☐ Dismiss stale pull request approvals when new commits are pushed  
New, reviewable commits pushed will dismiss previous pull request review approvals.

☒ Require approval of the most recent reviewable push  
Whether the most recent reviewable push must be approved by someone other than the person who pushed it.

☐ Require conversation resolution before merging  
All conversations on code must be resolved before a pull request can be merged.

☐ Automatically request Copilot code review  
Request Copilot code review for new pull requests automatically if the author has access to Copilot code review.

**Allowed merge methods**

Merge, Squash, Rebase

Which merge methods will be allowed for pull requests created in this repository? Select all that apply.

## 1.4. Regras de avaliação interna

A equipa realizará uma reunião oficial no final de cada *milestone* dedicada à avaliação interna. Nesta reunião, cada elemento será avaliado pelos restantes membros do grupo, sem possibilidade de autoavaliação, sendo o processo registado em ata e incluído no Plano de Projeto. A avaliação terá em conta os critérios enunciados na Figura 1.1:

Figura 1.1 – Critério para avaliação interna, e o respetivo peso.

Critério	Descrição	Peso (%)	Escala (1–5)
<b>Assiduidade e Pontualidade</b>	Presença nas reuniões e cumprimento dos horários definidos	10 %	1 = Faltas frequentes / atrasos, 5 = Sempre presente e pontual
<b>Participação Ativa</b>	Contributos em discussões, propostas de soluções,	20 %	1 = Passivo, 5 = Participa de forma consistente e relevante



	envolvimento nas tarefas		
<b>Cumprimento de Prazos</b>	Respeito pelas datas-limite acordadas para tarefas e entregas	15 %	1 = Nunca cumpre, 5 = Cumpre sempre sem atrasos
<b>Qualidade do Trabalho</b>	Clareza, rigor técnico e alinhamento com os requisitos	25 %	1 = Trabalho incompleto/deficiente, 5 = Trabalho de elevada qualidade
<b>Quantidade de Trabalho</b>	<i>Story points</i> /tarefas concluídas e aceites pelo grupo	15 %	1 = Pouco contributo, 5 = Contributo elevado e consistente
<b>Colaboração e Trabalho em Equipa</b>	Cooperação, entreaajuda, comunicação e ambiente positivo no grupo	15 %	1 = Individualista / não colabora, 5 = Forte espírito de equipa

## 1.5. Modelos

### **Modelo de cronograma**

O cronograma é planeado através do microsoft planner, onde os membros da equipa estão incluídos, que são onde estão definidas as tarefas com as respetivas datas e durações. De seguida é exportado um ficheiro excel, que é posteriormente convertido para o formato “.csv”.

Por fim, é usada uma ferramenta para gerar um gráfico de Gantt (<https://www.onlinegantt.com/#/gantt>), com o objetivo de obter-se uma configuração visual do cronograma.

### **Modelo A (Modelo de convocatórias)**

O modelo de atas pode ser consultado no ficheiro Conconvatória\_template.pdf

### **Modelo B (Modelo de atas)**

O modelo de atas pode ser consultado no ficheiro Ata\_template.pdf

## 2. Registos de Execução

Todos os artefactos homologados e assinados encontram-se no repositório “**project-artifacts**” da organização SeePaw.

### 3. Auditorias e Reviews

**No final de cada milestone, a equipa realiza uma reunião de auditoria conduzida pelo Team Leader para verificar:**

- Conclusão de todas as *User Stories/Issues*
- Consistência do código e documentação
- Rastreabilidade entre *issues*, *commits* e entregas
- Conformidade com as normas de desenvolvimento estabelecidas
- Aprovação e assinatura de todos os artefactos homologados

Correções a issues existentes identificadas durante as auditorias devem ser registadas como novas issues com prioridade apropriada e atribuídas ao próximo sprint, se aplicável.

**Durante a auditoria, são também analisados:**

- Burndown charts da milestone
- Velocity da equipa
- Percentagem de conclusão de Epics

**Não-conformidades críticas identificadas durante a auditoria que afetem a qualidade ou integridade do produto devem ser:**

- Documentadas imediatamente
- Classificadas como prioridade "Critical"
- Resolvidas antes do fecho formal do milestone

Os resultados da auditoria são considerados na avaliação interna dos membros da equipa (ver secção 1.4), nomeadamente nos critérios de qualidade do trabalho e cumprimento de prazos.