**PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL**
**ESCOLA POLITÉCNICA**
**BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO**

# Pipeline conteinerizada para CFD reprodutível de leitos empacotados

**Bernardo Klein Heitz**

Proposta de Trabalho de Conclusão apresentada como requisito parcial à obtenção do grau de Bacharel em Ciência da Computação na Pontifícia Universidade Católica do Rio Grande do Sul.

Orientador(a): Marco Aurelio Souza Mangan

**Porto Alegre**
**2025**

## 1. INTRODUÇÃO:

Leitos empacotados são amplamente empregados na engenharia química em operações de separação e mistura, reatores e processos com transferência de calor e massa, e a Dinâmica dos Fluidos Computacional (CFD, Computational Fluid Dynamics) tem sido central para simular, investigar e analisar campos de velocidade, pressão e outras grandezas com maior detalhamento do que abordagens puramente empíricas [9], [10], [11]. Ferramentas abertas como Blender (modelagem geométrica com suporte a scripts) e OpenFOAM (malha, solução e pós-processamento) viabilizam fluxos mais automatizados e reprodutíveis [14], [16]. A literatura já explora geração sintética de leitos e empacotamento por corpo rígido no Blender (por exemplo, Flaischlen et al., 2019; Kutscherauer et al., 2022) [32], [34], cadeias integradas para leitos com partículas resolvidas (Partopour & Dixon, 2017) [33] e comparações CFD–correlações para queda de pressão, como Ergun (Pavlišič et al., 2018) [7]. Além disso, há alternativas baseadas em DEM, como o software YADE, que permitem empacotamento e exportação de geometrias [16], [42]. Ainda assim, persistem desafios práticos na rotina de pesquisa: geração de geometrias fisicamente coerentes (parede cilíndrica, tampas e partículas); configuração de casos e solvers em pacotes CFD que exigem dezenas de arquivos e parâmetros [16]; rastreabilidade de versões, geometrias e resultados [27], [26]; e integração de dados para permitir comparações e comunicar resultados entre diferentes execuções [18], [23], [25].

No cenário que nos encontramos, observamos que muitos fluxos de trabalho ainda se mantêm manuais e divididos em scripts isolados, passos pouco documentados, muitas dependências específicas de máquina e pouco controle de versão dos parâmetros [44], [52]. O resultado encontrado atualmente é de tempo excessivo de preparo, alto risco de inconsistência entre as pesquisas e estudos e a dificuldade em realizar validações [8]. No caso dos leitos empacotados, a simulação do “problema do domínio” (dimensões do leito, propriedades das partículas e condição de escoamento) em um software CFD envolve a manipulação e escolhas que são facilmente perdidas ao longo do processo [16], [9], [11]. Dessa forma, observamos que falta uma metodologia unificada, declarativa e conteinerizada que permita ao pesquisador descrever o problema em alto nível e de forma padronizada, obter geometria -> simulação -> variáveis calculadas -> visualização, preservando o histórico e favorecendo a comparação entre os resultados salvos das simulações [49], [28], [45], [14], [27], [26].

Este trabalho tem como objetivo propor a integração de ponta a ponta de um conjunto de tecnologias que compõem um pipeline voltado à geração de geometrias de leitos empacotados e à execução de simulações em software de Dinâmica dos Fluidos Computacional (CFD) [9], [10], [11]. A solução proposta inicia-se com uma linguagem de domínio específico (DSL), na qual o pesquisador descreve os parâmetros do leito empacotado e as condições de simulação [49], [28], [45]. Um parser/compilador valida

essa descrição e gera um arquivo canônico (“params.json”). Em seguida, o Blender é executado em modo headless (CLI) para produzir a geometria do leito, incluindo paredes, tampas e partículas, organizadas por empacotamento com física rígida e exportá-la em formatos como STL, OBJ e FBX [14], [3], [5]. Na sequência, um template parametrizado de software CFD de código aberto (como o OpenFOAM) é utilizado para a geração da malha (por exemplo, blockMesh e snappyHexMesh) e para a solução numérica (como o simpleFoam), resultando em variáveis calculadas de interesse, tais como Δp, Δp/L e velocidade média [16], [42], [11], [12]. Os metadados e artefatos produzidos (CSV, VTK, STL, logs) são armazenados em banco de dados relacional (PostgreSQL), enquanto os objetos 3D permanecem em repositório dedicado (MinIO) [27], [26]. Esses dados são disponibilizados por meio de uma API web (FastAPI) e visualizados em um dashboard (React com Three.js/Plotly), que permite acompanhar histórico, comparações e visualizações tridimensionais [18], [23], [25]. Todo o ecossistema é conteinerizado com Docker Compose, garantindo maior reprodutibilidade e reduzindo barreiras relacionadas à instalação e configuração [44], [52].

Também, é importante reforçar a relevância do tema deste TCC, em vista que ele está diretamente alinhado ao meu trabalho de Iniciação Científica, no qual o problema de geração de modelos Tridimensionais Computadorizados de leitos empacotados já foi identificado com gargalo na pesquisa em que atuo. No laboratório onde estou realizando meu trabalho de IC, atuo apoiando um aluno de doutorado e seu professor com a geração da malha de leitos empacotados e suas partículas utilizando o software Blender, porém, visto que essa tarefa é lenta, repetitiva e muitas vezes difícil foi encontrado uma dor nesse processo, por isso o pipeline aqui proposto ataca exatamente essa questão: Ser um sistema autocontido no próprio escopo (DSL -> Blender -> OpenFOAM -> API/Dashboard), reduz a dependência de múltiplas interfaces, cliques e expertise em vários softwares, permitindo rodar simulações com rapidez, reaproveitando resultados em diferentes formas, como por exemplo varreduras paramétricas, estudos de independência de malha, calibração de modelos e montagem de figuras/tabelas para artigos e relatórios. Com isso, o trabalho contribui com um fluxo de pesquisa, que padroniza entradas, automatiza etapas críticas e preserva o histórico das runs (execuções do pipeline), dessa forma facilitando a comparação entre as execuções, comunicação de resultados dentro de diversos tipos de projetos, além de conseguimos reduzir os gastos em pesquisa, visto que estamos utilizando programas de código aberto, evitando assim qualquer tipo de assinatura de licenças.

Os objetivos são: reduzir o esforço técnico e as dificuldades encontradas na preparação de estudos, por meio do uso de uma especificação declarativa (DSL) e de automações [49], [28], [45]; aumentar a reprodutibilidade (mesmas entradas resultando em mesmas saídas dentro de padrões estabelecidos) [44], [52]; garantir rastreabilidade de decisões e versões [27], [26], [47]; e facilitar a análise, a comparação, os estudos e o

compartilhamento de resultados por meio de um dashboard com variáveis calculadas e visualização 3D das geometrias dos leitos empacotados [18], [23], [25]. O escopo inicial concentra-se em leitos cilíndricos com partículas esféricas, escoamento incompressível e regime estacionário (com possibilidade de modelagem da turbulência por meio de modelos RANS), por serem configurações didáticas e recorrentes na literatura [11], [12]. Extensões como escoamentos multifásicos, reativos e execução em ambientes de HPC são discutidas como perspectivas para trabalhos futuros. Como critério de coerência física, adotou-se a correlação de Ergun para Δp/L como referência externa de validação em faixas apropriadas [7], além da realização de estudos de independência de malha por meio do índice de convergência de malha (GCI) [8].

Em resumo, a contribuição desta proposta de trabalho está em transformar um processo historicamente manual, “do domínio ao CFD”, em uma pipeline padronizada e reprodutível, alinhada ao foco computacional de um TCC: linguagem e compilação de entradas [49], [28], [45]; automação de modelagem/simulação [14], [16], [42]; engenharia de dados [27], [26], [47]; e aplicação web para consumo e comparação de resultados [18], [23], [25], garantindo reprodutibilidade por meio de conteinerização com Docker Compose [44], [52].

## 2. PROPOSTA:

A proposta do trabalho irá descrever o que é pretendido ser construído ao longo do TCC: um pipeline completo, com características declarativas, automatizadas e conteinerizadas, para a geração de modelagem geométrica e simulação CFD de leitos empacotados, incluindo ingestão de resultados e visualização em um dashboard web [14], [16], [18], [23], [25], [44], [52]. Por se tratar de uma proposta, as escolhas apresentadas funcionam como guias de implementação, por isso detalhes específicos (parâmetros, listas de campos, ajustes de malha e solver) poderão ser refinados ao longo do desenvolvimento, sem perder de vista o objetivo central do pipeline: reduzir a dificuldade operacional e aumentar a reprodutibilidade científica [8], [7], [27], [26].

### 2.1 Problema e escopo:

Problema detalhado:

Pesquisadores de engenharia química que estudam leitos empacotados costumam enfrentar quatro obstáculos práticos:

Geração de geometria fisicamente coerente (parede/estrutura cilíndrica, tampas e partículas) feita manualmente, consumindo tempo em leitos pequenos ou grandes [14], [3], [5].

Preparação da simulação CFD com muitos arquivos e parâmetros (malha, contornos, esquemas numéricos), suscetível a erros e difícil de reproduzir sem histórico claro [16], [9], [11].

Organização de resultados (variáveis calculadas, logs, malhas e campos) pouco

padronizada, o que dificulta comparações e recuperação de histórico [27], [26].

Comunicação dos achados dispersa (pastas locais, prints, planilhas), sem integração de dados e visualização consolidada [18], [23], [25].

Escopo que será focado no TCC:

Foco em leitos cilíndricos preenchidos com partículas (esferas, cubos, cilindros e planos) definidas via DSL; o modelo 3D inclui tampas inferior e superior [14], [49], [28], [45]. A versão inicial (v1) deve:

transformar uma descrição declarativa do leito (DSL) em geometria 3D (ex.: STL) [49], [28], [45], [14];

gerar malha e executar solver em CFD (ex.: OpenFOAM: blockMesh, snappyHexMesh, simpleFoam) [16], [42];

extrair variáveis calculadas, registrar metadados/artefatos e disponibilizar via API e dashboard [18], [23], [25], [27], [26];

rodar conteinerizado (Docker Compose), priorizando reprodutibilidade [44], [52].

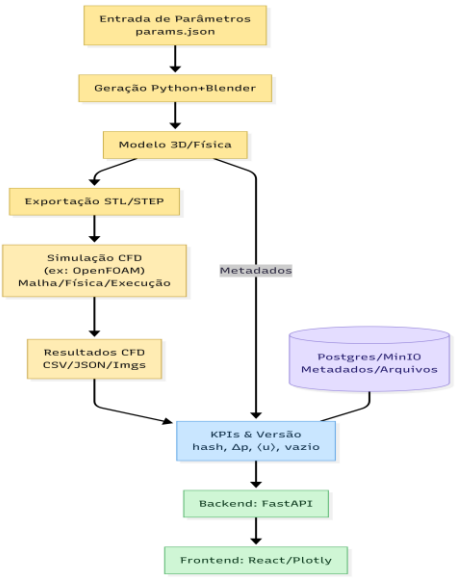
### 2.2 Visão geral da solução:

A proposta converte o problema do domínio (dimensões do leito, partículas e condições de escoamento) em um pipeline automatizado e reprodutível que entrega resultados comparáveis, versionados e fáceis de inspecionar [49], [16], [27], [26], [44], [52]. O objetivo é reduzir a necessidade de abrir o Blender manualmente, editar múltiplos arquivos no OpenFOAM e organizar resultados na mão, oferecendo rastreabilidade ponta a ponta por meio de DSL, API e dashboard [14], [16], [18], [23], [25].

Figura 1 (fluxo ponta a ponta). O usuário escreve um arquivo .bed (DSL) descrevendo o leito; o compilador gera um params.json canônico (SI e chaves padronizadas) [49], [28], [45]. O Blender (CLI) cria o modelo 3D e exporta STL; o OpenFOAM gera a malha e roda o solver; o sistema extrai variáveis calculadas (ex.: Δp, Δp/L, velocidade média, Re), armazena metadados/artefatos de forma organizada e expõe tudo via API para visualização no dashboard [14], [16], [11], [27], [26], [18], [23], [25]. O pesquisador acompanha o status e analisa os resultados no navegador, sem abrir a interface do Blender nem editar arquivos do CFD [14], [16], [18], [23].

*Figura 1 – Pipeline resumido*

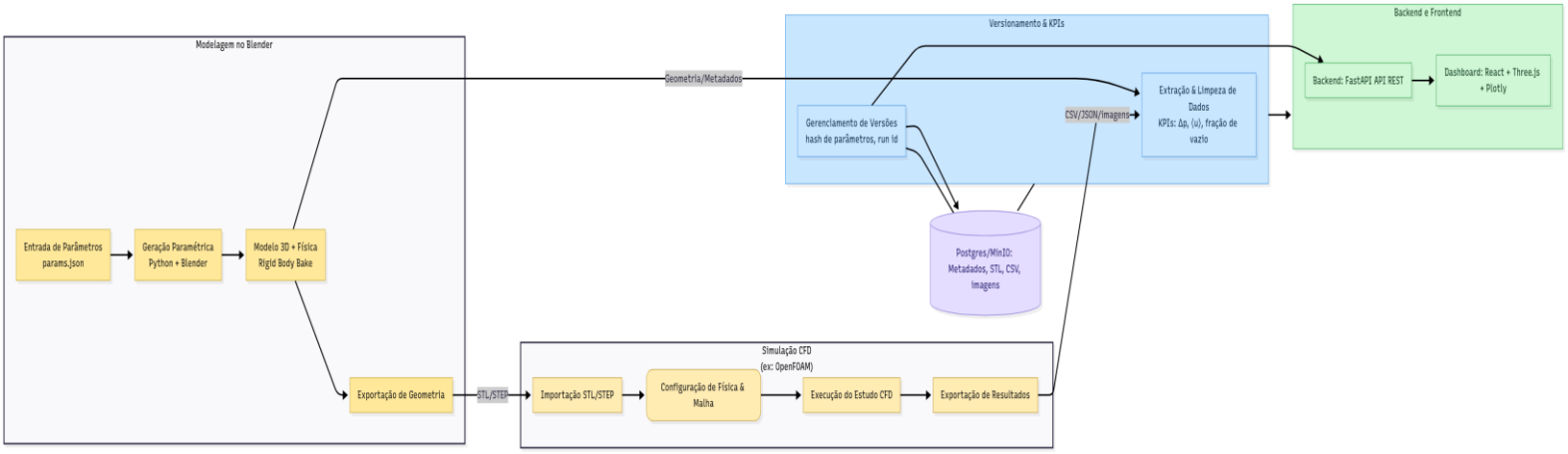
*Fonte: produção autoral (2025)*




A Figura 1 mostra o fluxo ponta a ponta: O usuário escreve o arquivo .bed, o compilador gera o params.json, o Blender produz a geometria, o OpenFOAM roda a simulação, e os resultados são armazenados e disponibilizados pela API para visualização no dashboard.

*Figura 2 – Arquitetura detalhada do sistema*

*Fonte: produção autoral (2025)*




A Figura 2 mostra os módulos principais (DSL, Blender, OpenFOAM, banco, storage, API, dashboard), o fluxo de dados entre eles e os mecanismos de versionamento e rastreabilidade.

DSL/Compilação: valida sintaxe/semântica, normaliza para SI e gera params.json canônico; um hash identifica a variante para comparação e deduplicação [49], [28], [45].

Modelagem geométrica (Blender CLI): script gera parede cilíndrica/estrutura, tampas e partículas com seed para reprodutibilidade; bake do empacotamento; exporta STL “manifold” adequado ao snappyHexMesh [14], [3], [5], [16].

CFD (OpenFOAM): pipeline executa blockMesh -> snappyHexMesh -> simpleFoam; functionObjects calculam médias (pressão, velocidade) e estatísticas durante a execução; o worker consolida um CSV padronizado com variáveis calculadas (ex.: Δp, Δp/L, velocidade média, Re, nº de células, tempo do solver, resíduos) [16], [11], [12].

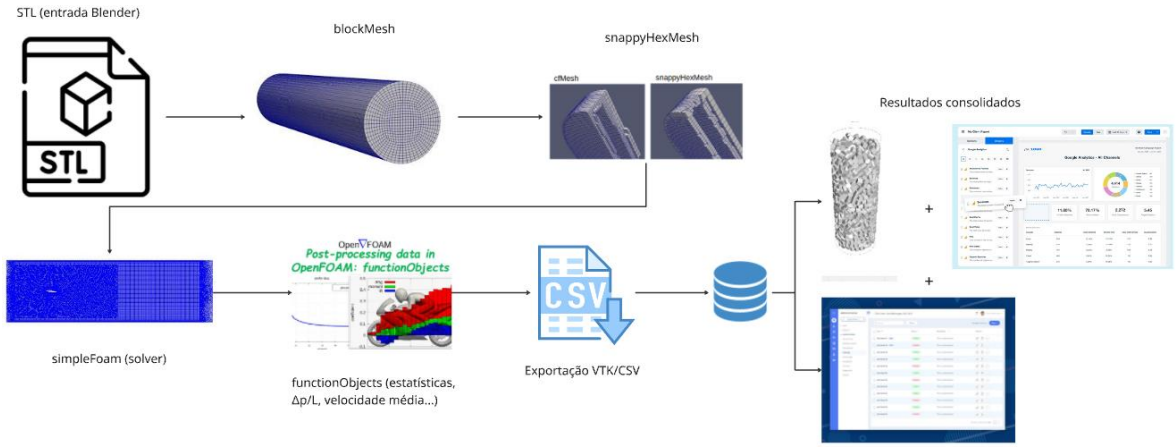
Ingestão e persistência: metadados/variáveis no PostgreSQL; artefatos pesados (STL, VTK/VTU, CSV, logs) no MinIO (compatível S3), organizados por usuário/execução, com rastro completo (hash do params.json canônico, seed, versões dos containers, tempos) [27], [26], [44], [52].

API e Dashboard: FastAPI oferece rotas autenticadas para criar jobs, consultar execuções/variantes e emitir URLs assinadas (temporárias) para arquivos; o dashboard (React + Three.js + Plotly) mostra o 3D do leito, gráficos interativos das variáveis, logs e comparação lado a lado, facilitando análise e comunicação dos resultados [18], [23], [25].

Com esse desenho, o usuário apenas descreve o leito em .bed, acompanha o status da execução e consulta resultados/artefatos no dashboard, mantendo rastreabilidade entre parâmetros, versões e arquivos em todas as etapas [49], [16], [27], [26], [18], [23], [25], [44], [52]

*Figura 3 - Pipeline de simulação CFD com OpenFOAM*

*Fonte: produção autoral (2025)*




A Figura 3 mostra o fluxo de processamento a partir da geometria exportada em STL (via Blender). O arquivo STL alimenta o `blockMesh`, que gera a malha inicial, posteriormente refinada pelo `snappyHexMesh`. Em seguida, o solver `simpleFoam` é executado, enquanto os `functionObjects` calculam estatísticas como queda de pressão (Δp, Δp/L) e velocidade média. Os resultados são exportados em formatos VTK/CSV para visualização científica e armazenados em banco de dados, sendo finalmente consolidados em relatórios, dashboards e visualização 3D.

## 2.3 Entradas -> Processamento -> Saídas:

### *2.3.1 Entradas:*

As entradas são definidas em um arquivo .bed (linguagem de domínio) que descreve, de forma declarativa, a geometria do leito, as tampas, as partículas e a política de exportação/uso posterior. O compilador lê esse arquivo, valida unidades e coerência, e normaliza tudo para SI, gerando um params.json canônico, o que é uma etapa fundamentada em DSLs e parsing/validação tipada (Lark + Pydantic) [49], [28], [45], [17]. A seguir, o resumo de cada um dos blocos.

bed, geometria do leito (cilindro):
Define o corpo do leito com diâmetro interno, altura útil, espessura da parede e folga superior (clearance). Itens opcionais incluem rugosidade e material (metadado). Regras básicas: positividade de dimensões, espessura menor que metade do diâmetro e respeito ao volume interno efetivo. Esses parâmetros servem de base para a modelagem geométrica no Blender (headless) e para a etapa de malha do CFD [14], [16].

lids, que são as tampas (superior e inferior).
Especifica o tipo de tampa (flat, hemispherical ou none) e as espessuras associadas. Pode incluir folga de vedação (seal). Se houver tampa, a espessura deve ser > 0 e a soma espessuras + clearance não pode inviabilizar o volume interno. O objetivo é manter consistência geométrica para geração do sólido e empacotamento [14].

particles, escolha e parâmetros das partículas:
No escopo inicial, foca em esferas monodispersas: kind = sphere, diameter e count ou target_porosity, exclusivos entre si. Alternativamente, descreve polidispersidade por classes (bins que somam 1.0) ou por distribuição (p. ex., lognormal com limites). Inclui propriedades físicas relevantes ao empacotamento por corpo rígido: densidade ou massa (com derivação automática para esferas), restituição, atritos (estático/rolamento), amortecimentos linear/angular, folgas mínimas partícula–parede e partícula–partícula, margem de colisão e parâmetros de posicionamento (método de inserção, altura de queda), além da seed para reprodutibilidade. O compilador aplica regras como: normalização SI, positividade de grandezas, exclusividade count XOR target_porosity, frações que somam 1.0 (no caso poli), limites físicos 0–1 para coeficientes e checagens de consistência de espaço interno (clearance/tampas) [14], [49], [28], [45]. Esses parâmetros alimentam o empacotamento rígido no Blender em modo CLI, mantendo o processo reproduzível e scriptável [14], [3], [5].

packing, controle do empacotamento (Blender, corpo rígido):

Agrupa a configuração do solver físico: método (rigid_body), gravidade, subpassos e iterações por frame, amortecimento, critério de repouso (limite de velocidade), tempo máximo de simulação e margem de colisão. Esses controles permitem obter um arranjo estável das partículas, com custo numérico previsível e resultados repetíveis (via seed) [14].

export, saídas geométricas:
Define formatos (ex.: stl_binary, obj, fbx), unidades/escala (o compilador normaliza SI),

triangulação e opções de merge by distance. Inclui a exigência de manifold (sem buracos, normais consistentes) para evitar falhas na etapa de malha do CFD [14], [16].

Modelagem da superfície interna do cilindro: dois métodos e “switches”.
Para representar o domínio do fluido no interior do leito, a proposta contempla dois caminhos complementares, selecionados por chaves (switches) no bloco export:

Método A, Superfície interna (recomendado para CFD/snappyHexMesh):
Exporta apenas a superfície interna do cilindro (lateral + tampas internas) como STL fechado/manifold, bem como as superfícies das partículas. O snappyHexMesh recorta/refina a malha a partir do background mesh. A espessura da parede fica como metadado (não afeta o STL de CFD).
Switches: wall_mode = "surface" (padrão), fluid_mode = "none".
Vantagens: malhas mais estáveis, menos risco de não-manifold, fluxo mais simples [16].

Método B, Cavidade por booleana (opcional):
Constrói explicitamente o “vazio” do fluido por subtração booleana (cilindro externo, cilindro interno e partículas), gerando um STL único do domínio de fluido.
Switches: wall_mode = "solid", fluid_mode = "cavity".
Observação: útil quando é necessário um volume único do fluido, porém operações booleanas em conjuntos grandes de partículas exigem checagens rigorosas de manifold para evitar degraus na malha [14], [16].

Padrão seguro para CFD: wall_mode="surface" e fluid_mode="none". Quando fluid_mode="cavity" for usado, recomenda-se validação automática (manifold) antes do snappy [16].

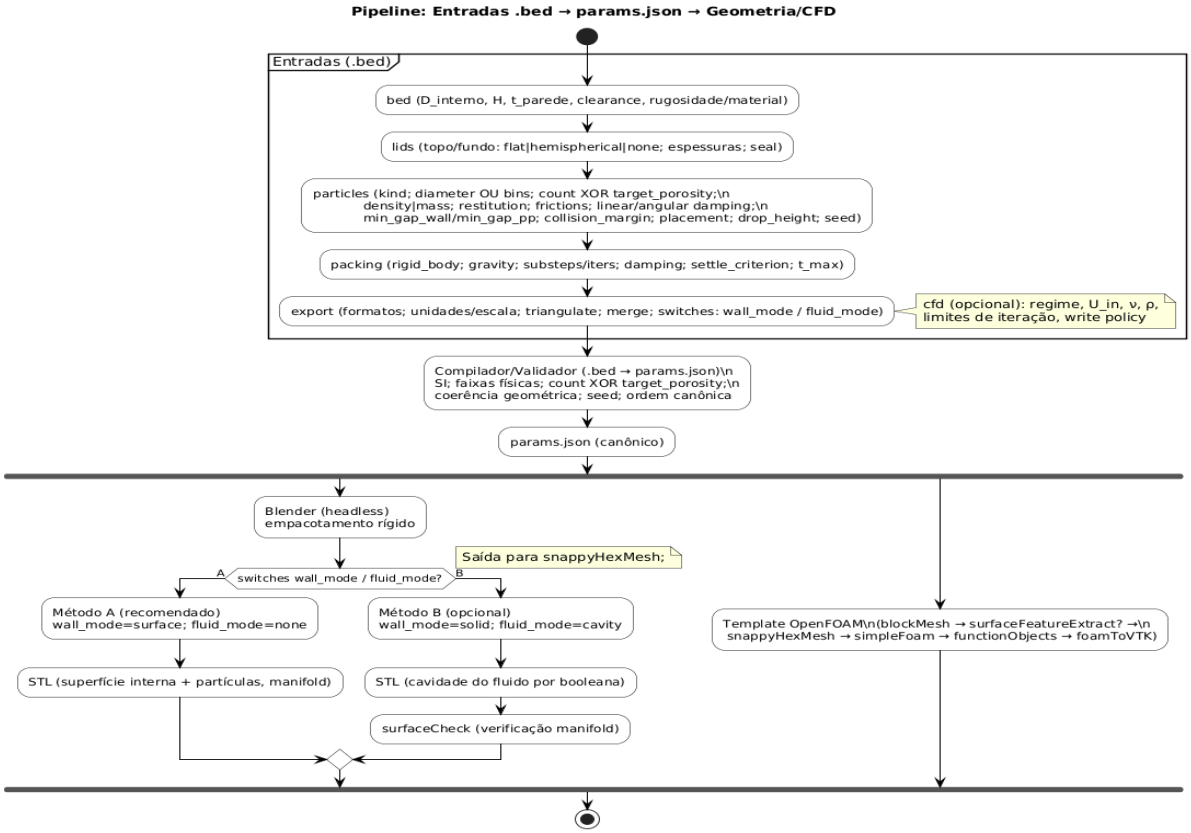
cfd (opcional) — parâmetros para o template.
Embora a simulação seja gerada por templates do OpenFOAM, o .bed pode fornecer pistas: regime (laminar/RANS), velocidade de entrada, propriedades do fluido (ν, ρ), limites de iteração e política de escrita de campos. Esses valores são consumidos ao montar os dicionários (blockMeshDict, snappyHexMeshDict, fvSchemes, fvSolution, controlDict) e a execução do simpleFoam [16], [11], [12].

Saída dessa etapa:

O compilador consolida todas as entradas em um params.json canônico (ordem estável de chaves, nomes padronizados, SI), pronto para: geração geométrica no Blender (com empacotamento rígido reprodutível) e criação do caso OpenFOAM (malha com blockMesh/snappyHexMesh e solução com simpleFoam) [14], [16]. Essa padronização viabiliza rastreabilidade e repetição controlada de experimentos nas etapas seguintes.

*Figura 4 – Pipeline: Entradas `.bed` -> `params.json` -> Geometria/CFD*

*Fonte: produção autoral (2025)*




A Figura 4, o diagrama mostra, da esquerda para a direita, os blocos de entrada do .bed (bed, lids, particles, packing, export e cfd (opcional)) convergindo no Compilador/Validador (normalização SI e geração do params.json canônico). A partir dele, o fluxo se divide: Blender (headless) com decisão pelos switches wall_mode/fluid_mode, sendo o Método A (surface: superfície interna + partículas, manifold) ou o Método B (cavity: cavidade do fluido por booleana, verificada via surfaceCheck) — produzindo STL para o snappyHexMesh; e template OpenFOAM (blockMesh -> surfaceFeatureExtract? -> snappyHexMesh -> simpleFoam -> functionObjects/foamToVTK).

### *2.3.2 Processamento:*

Compilação (Python): valida sintaxe/coerência, normaliza para SI e gera params.json canônico (Lark + Pydantic em Python) [28], [45], [17].

Geração geométrica (Blender CLI): cria o leito (paredes/tampas) e adiciona partículas com empacotamento por física rígida; exporta STL/OBJ/FBX, com checagens básicas (escala e manifold). STL é compatível com a etapa de malha do OpenFOAM [14], [15], [3], [5], [16].

Simulação CFD (OpenFOAM):

Malha: blockMesh (base) e snappyHexMesh (recorte/refino sobre STL) [16], [42].

Solver: simpleFoam para escoamento incompressível, laminar ou com turbulência (RANS) [16], [11], [12].

Registro de variáveis calculadas: Δp, Δp/L, velocidade média, número de Reynolds, número de células, tempo total de execução e resíduos de convergência [9], [11].

Exportação de campos (pressão, velocidade, turbulência) em VTK/VTU para visualização no ParaView [29], [30].

Armazenamento e versionamento: resultados numéricos e metadados no PostgreSQL; artefatos pesados (STL, VTK, CSV, logs) no MinIO; cada execução possui identificador único e empacotamento reprodutível por seeds fixas [27], [26].

Disponibilização: API em FastAPI (autenticação por JWT) expõe resultados e gera links temporários; dashboard em React permite visualização 3D do leito e análise gráfica interativa (Three.js/Plotly), além de comparação entre execuções [18], [48], [23], [25].

### *2.3.3 Saídas:*

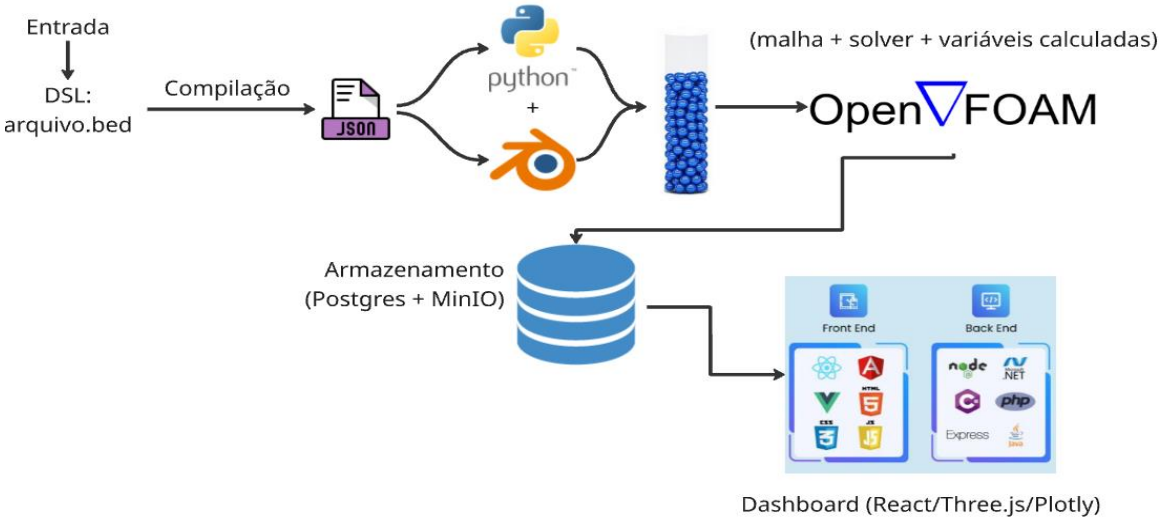
Variáveis calculadas: Δp (Pa), Δp/L (Pa·m$^{-1}$), velocidade média (m·s$^{-1}$), número de Reynolds (–), número de células, tempo total de execução (s) e resíduos de convergência [9], [11].

Artefatos organizados: STL (geometria), VTK/VTU (campos simulados), CSV (dados numéricos) e logs (execução) [16], [29], [30].

Interface web: visualização 3D interativa, gráficos comparativos e histórico filtrável com downloads (API + dashboard) [18], [23], [25].

*Figura 5 – Fluxo Entradas -> Processamento -> Saídas*

*Fonte: produção autoral (2025)*




A Figura 5 representa os blocos de entrada (arquivo .bed), compilação, geração de geometria (Blender), simulação CFD (OpenFOAM), armazenamento em bancoe

disponibilização via API/dashboard, finalizando nas saídas (variáveis calculadas, artefatos e interface web).

### 2.4 Arquitetura e Tecnologias:

DSL e compilação: Python com Lark (parsing) e Pydantic (validação/normalização SI) gera um params.json estável para versionamento [28], [45], [17], [49].

Modelagem geométrica: Blender 4.x em modo headless (CLI) via Python API, criando o leito com partículas e exportando STL/OBJ/FBX (superfície triangulada adequada a malha e visualizadores) [14], [15], [3], [5].

Simulação CFD: OpenFOAM (LTS) com blockMesh (malha base), snappyHexMesh (recorte/refino no STL) e simpleFoam (incompressível, laminar/RANS). Variáveis de interesse: Δp, Δp/L, velocidade média, Re, nº de células, tempo de execução e resíduos; campos (p, U, k, ε) podem ser visualizados no ParaView/VTK [16], [42], [11], [12], [29], [30].

Orquestração: Redis como fila de tarefas e worker em Python com timeouts e retries [46].

Dados: PostgreSQL 16 para metadados/variáveis e MinIO para artefatos grandes (STL/VTK/CSV/logs), organizados por usuário/execução; acesso via SQLAlchemy [27], [26], [47].

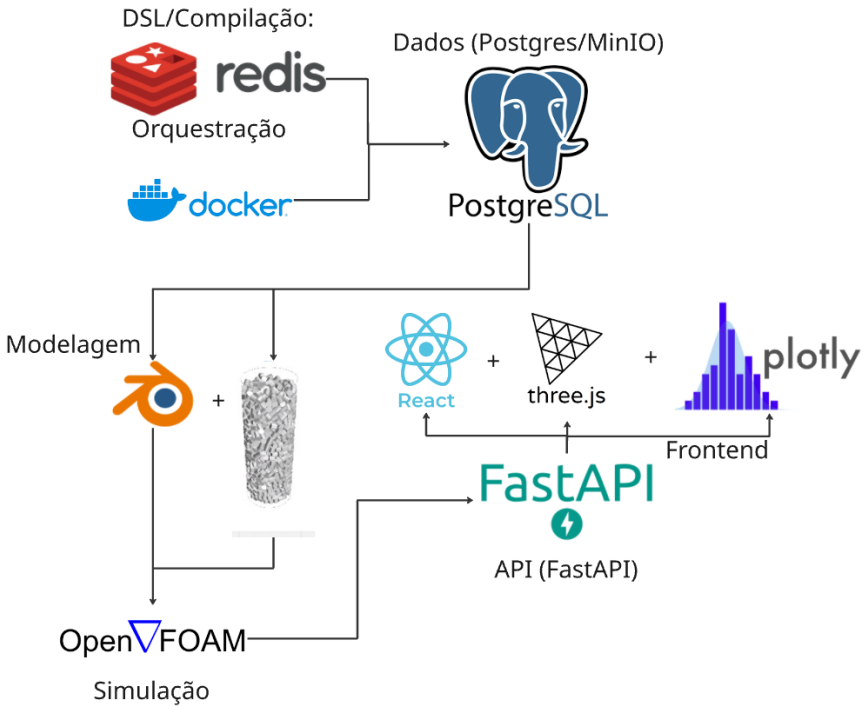
API: FastAPI (Python) com JWT para autenticação; rotas para login, criação/consulta de execuções e links temporários (OpenAPI para documentação) [18], [48], [55].

Frontend: React para UI, Three.js para visualização 3D do leito e Plotly para gráficos interativos [23], [25], [21], [22].

Empacotamento/execução: Docker Compose reúne banco, storage, fila, API, worker e frontend; execução integrada com docker compose up -d [44], [52] Essas tecnologias são abertas e consolidadas, com forte suporte à automação; a separação em serviços facilita evolução e substituição de componentes (ex.: trocar o solver CFD) [44], [52].

*Figura 6 - Arquitetura e Tecnologias utilizadas*

*Fonte: produção autoral (2025)*




A Figura 6 mostra de forma esquemática os componentes técnicos (DSL/compilação, Blender, OpenFOAM, Redis, Postgres, MinIO, FastAPI, React/Three.js/Plotly, Docker Compose) e como eles se integram.

### *2.4.1 Fluxo de visualização web:*

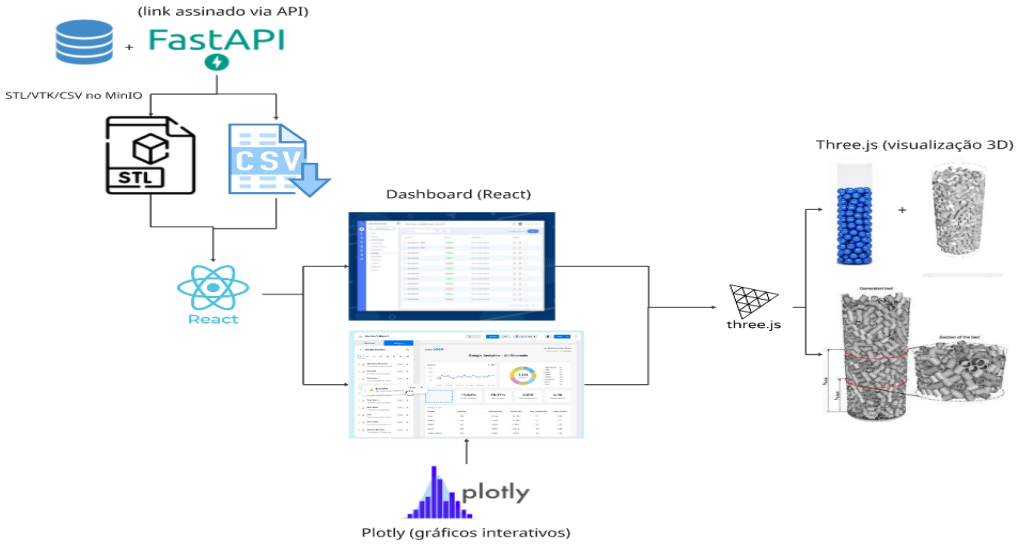
Nesta etapa, os artefatos gerados (STL/VTK/CSV/logs) são mantidos no MinIO; a API FastAPI expõe rotas autenticadas (JWT) que emitem URLs assinadas e temporárias para acesso seguro aos arquivos [26], [18], [48].

O dashboard em React consome essas URLs: carrega a geometria 3D (STL/glTF) no Three.js e plota séries/valores em Plotly, permitindo inspeção, comparação e download sem expor diretamente o storage [23], [25].

Esse desenho separa as responsabilidades (armazenamento <-> API <-> Interface com o usuário), reforça segurança e facilita a rastreabilidade entre execução, parâmetros e resultados [26], [18].

*Figura 7 - Fluxo de visualização web*

*Fonte: produção autoral (2025)*




A Figura 7 mostra como os artefatos (STL/VTK/CSV/logs) são mantidos no MinIO, acessados via URLs assinadas fornecidas pela API FastAPI, consumidos pelo dashboard em React e exibidos em Three.js (3D) e Plotly (gráficos).

### 2.5 O que será entregue do MVP ao v1:

MVP (até o final do semestre atual):

DSL e compilador validando parâmetros e gerando params.json padronizado para casos didáticos [28], [45], [49].

Blender headless gerando model.stl reprodutível para um leito base [14], [3], [5].

Pipeline mínima de CFD (OpenFOAM) para caso leve, produzindo globals.csv com variáveis calculadas essenciais (Δp, Δp/L, velocidade média, nº de células, tempo, resíduos) [16], [11], [12].

API e Dashboard para upload do .bed, visualização 3D e consulta às variáveis calculadas (FastAPI/React/Three.js/Plotly) [18], [23], [25].

Stack conteinerizada iniciada com um comando para executar um caso fim a fim [44], [52].

v1 (monografia final):

Execução OpenFOAM em caso base com estudo de independência de malha (GCI) [16], [8].

Validação física: comparação Δp/L simulado × correlação de Ergun, com meta de erro definida [7].

Dashboard expandido: comparação lado a lado (2–3 execuções) e download de artefatos (STL, VTK/VTU, CSV, logs) [29], [30], [16].

Documentação completa: guia do usuário, manual técnico e pacote de replicação (versões, parâmetros, comandos) [44], [52].

### 2.6 Modelo de dados e rastreabilidade:

Cada execução armazena informações suficientes para reprodutibilidade e auditoria:

Variants: configuração utilizada (hash do params.json canônico, parâmetros normalizados, seed do empacotamento, versões de containers e commit do repositório) [44], [52].

Runs: registro de cada execução (vínculo à variant, início/fim, estado, solver/malha e tempo total de execução) [16], [44].

Metrics: variáveis calculadas por execução (nome, unidade, valor, vínculo ao run), como Δp, Δp/L, velocidade média, nº de células, tempo do solver e resíduos [9], [11].

Files: artefatos produzidos (STL, VTK/VTU, CSV, logs) com localização no MinIO (compatível S3) e checksum para integridade [26], [29], [30].

### 2.7 Como o trabalho será validado:

Produtividade e usabilidade (parcial): comparar o tempo de preparo manual de um caso didático com o tempo via pipeline, visando redução ≥ 50%; verificar se um usuário não especialista executa o fluxo completo a partir de um guia curto. A avaliação considera execução conteinerizada e repetível do ambiente [44], [52].

Foco: maior agilidade, menor dificuldade e experiência simples.

Validação operacional: iniciar a pilha conteinerizada em ambiente limpo e concluir um job exemplo sem intervenção, com variáveis calculadas e artefatos disponíveis no dashboard (API FastAPI + frontend React/Three.js/Plotly) [18], [23], [25], [44], [52].

Em resumo: subir o stack, rodar o job e visualizar resultados no painel.

Planejamento experimental (DOE enxuto): variar velocidade superficial e diâmetro de partícula, com 3 repetições (seeds) por ponto para estimar a variabilidade do empacotamento (rigid-body/empacotamento no Blender) [14], [32].

Resultados esperados (tabelas e gráficos):

curvas de Δp/L comparadas à correlação de Ergun [7];

gráficos de convergência dos resíduos e estabilidade do solver simpleFoam [16];

boxplots de produtividade (tempos manual vs. pipeline) apresentados no dashboard (Plotly) [25].

### 2.8 Riscos e como pretendo lidar:

Variabilidade do empacotamento de partículas: o uso de física rígida e processos estocásticos pode produzir arranjos distintos e pequenas variações nos resultados [14], [32].

Mitigação: fixar a seed ao comparar variantes e executar 3 réplicas (seeds diferentes) para estimar a variabilidade.

Adoção por usuários não especialistas: dificuldades em detalhes de CFD e ambiente de execução.

Mitigação: guia passo a passo, mensagens de erro claras na DSL (parser/validação com Lark/Pydantic), upload simples do .bed via dashboard (API FastAPI e UI

React/Three.js/Plotly) [28], [45], [18], [23], [25]. A execução conteinerizada reduz atritos de instalação e diferenças de ambiente [44], [52].

### 2.9 Como esta proposta orienta o restante do trabalho:

A proposta define o que será desenvolvido (pipeline declarativa do .bed ao dashboard), como será entregue (MVP -> v1), quais tecnologias serão usadas (Blender, OpenFOAM, FastAPI, React, Docker) e como os resultados serão avaliados (independência de malha por GCI, comparação Δp**/L × Ergun**, reprodutibilidade e ganhos de produtividade) [14], [16], [18], [23], [44], [52], [8], [7]. Funciona como guia para as próximas etapas, alinhando decisões técnicas e expectativas entre orientador e banca.

# 3. CRONOGRAMA:

## 3.1 Cronograma inicial de Atividades:

*Tabela 1 – Cronograma inicial de atividades*

*Fonte: produção autoral (2025)*

| Atividade | Mês 1 | Mês 2 | Mês 3 | Mês 4 |
|---|---|---|---|---|
| A1. Levantamento bibliográfico | Sem. 1–2 | | | |
| A2. Especificação da DSL | Sem. 3–4 | | | |
| A3. Parser/compilador da DSL | | Sem. 1–2 | | |
| A4. Geração geométrica (Blender) | | Sem. 2–3 | | |
| A5. Template OpenFOAM (caso base) | | Sem. 3–4 | | |
| A6. Pipeline malha/solver | | | Sem. 1–2 | |
| A7. Pós-processamento (variáveis) | | | Sem. 2 | |
| A8. Ingestão e persistência (DB/MinIO) | | | Sem. 3 | |
| A9. API (FastAPI) | | | Sem. 3–4 | |
| A10. Frontend (React/Plotly/Three.js) | | | | Sem. 1–2 |
| A11. Integração ponta a ponta (E2E) | | | | Sem. 2 |
| A12. Validação inicial (numérica/física) | | | | Sem. 3 |
| A13. Documentação TCC1 | | | | Sem. 3–4 |
| A14. Apresentação (slides/ensaio) | | | | Sem. 4 |
| A15. Gestão e checkpoints | Contínuo (semanal, meses 1–4) | | | |

A Tabela 1 apresenta a distribuição preliminar das atividades do TCC ao longo do semestre, incluindo levantamento bibliográfico, desenvolvimento da DSL, integração de ferramentas, execução de simulações e documentação.

### 3.2 Cronograma detalhado:

*Tabela 2 – Cronograma detalhado*

*Fonte: produção autoral (2025)*

| Atividade | Entregas principais | Critério de aceite |
|---|---|---|
| A1. Levantamento bibliográfico | Referências (DSL, Blender API, OpenFOAM, Ergun, visualização), arquivo references.bib organizado | 10–15 referências-chave revisadas e salvas por tema |
| A2. Especificação da DSL | Definição de blocos (bed, particles, physics, export), unidades e exemplos | Rascunho da gramática + exemplos .bed válidos e inválidos com mensagens adequadas |
| A3. Parser/compilador da DSL | Implementação em Python (Lark + Pydantic), geração de params.json canônico | Mesmos .bed ⇒ mesmo params.json; erros com linha/coluna reportados |
| A4. Geração geométrica (Blender) | Script headless: paredes, tampas, partículas com empacotamento; exportação STL | STL em escala correta e geometria manifold |
| A5. Template OpenFOAM (caso base) | Estrutura de caso base (0/, constant/, system/) com dicionários parametrizados | blockMesh e snappyHexMesh -overwrite executam sem erro |
| A6. Pipeline malha/solver | Automação blockMesh → snappyHexMesh → simpleFoam, com logs e monitoramento | Solver conclui com resíduos dentro da meta no caso didático |
| A7. Pós-processamento (variáveis) | Scripts para Δp/L, velocidade média, Re, nº de células, tempo; globals.csv padronizado | Arquivo globals.csv com colunas fixas e valores plausíveis |
| A8. Ingestão e persistência (DB/MinIO) | Metadados e variáveis no Postgres; artefatos (STL/VTK/CSV/logs) no MinIO | Execução registrada no DB com variáveis e links para arquivos |
| A9. API (FastAPI) | Rotas: upload .bed, listar execuções, obter link de arquivos | Documentação /docs acessível e rotas testáveis via curl |
| A10. Frontend (React/Plotly/Three.js) | Páginas: login, novo job, lista de execuções, detalhes 3D/variáveis, comparação | Usuário envia .bed, acompanha status e visualiza resultados |
| A11. Integração ponta a ponta (E2E) | Pipeline completo: .bed → params.json → STL → malha/solver → variáveis → DB/API → dashboard | Caso didático roda de ponta a ponta sem intervenção manual |
| A12. Validação inicial (numérica/física) | Estudo de malha (3 níveis) + curva Δp/L vs. Ergun; medições de tempo de preparo | Tabela preliminar de GCI e erro Δp/L dentro do esperado |
| A13. Documentação TCC1 | Proposta consolidada, guia rápido do usuário (2 págs), README atualizado | Revisão do orientador e entrega no prazo |
| A14. Apresentação (slides/ensaio) | Slides de 10–15 min com problema, método, demo curta e resultados | Ensaio cronometrado; narrativa clara |
| A15. Gestão e checkpoints | Reuniões semanais, atas e ajustes de rota | Atas curtas e decisões registradas |

A Tabela 2 mostra a organização temporal do trabalho, com as atividades estruturadas em semanas/meses, permitindo visualizar marcos de entrega (MVP → v1) e dependências entre tarefas.

## 4. RECURSOS A SEREM UTILIZADOS:

Esta seção resume o que é necessário ter disponível para que o projeto seja viável tecnicamente e logisticamente, avaliando criticidade, disponibilidade, indisponibilidade e resolver problemas caso algo falhe. A ideia é evitar surpresas, dessa forma, se ocorrer erros e falhas relacionados a tempo, máquina, software, já existe alternativas pensadas.

*Tabela 3 – Matriz rápida de recursos*

*Fonte: produção autoral (2025)*

| Categoria | Recursos principais | Riscos | Soluções |
|---|---|---|---|
| Hardware | Local: 4 vCPUs, 8 GB RAM, SSD ≥ 20 GB. Recomendado: 8–32 GB RAM, SSD ≥ 100 GB. | Espaço insuficiente; máquina lenta | Compressão/limpeza periódica, uso de casos menores, execução parcial em outra máquina |
| Software | Docker + Compose, Blender 4.x (CLI/Python API), OpenFOAM (LTS), Python 3.10+, FastAPI, SQLAlchemy, Redis, MinIO SDK, PostgreSQL 16, React, Node 20+, Three.js, Plotly, Git/GitHub, Make, VS Code | Incompatibilidade (Windows), versões quebradas | Uso de WSL2/VM Linux, fixar versões/tags, execução local ou remota como alternativa |
| Dados/armazenamento | Volumes Docker (Postgres, MinIO, shared/). Estrutura de cada run: input.bed, params.json, STL, caso OpenFOAM, logs/KPIs. Tamanhos: STL 5–50 MB; malha 100–800 MB; VTK 50–400 MB. | Perda de dados; consumo excessivo por VTK | Backups automatizados e testados, quotas de usuário, exportar VTK apenas no final, compactar |
| Pessoas/tempo | Orientador (encontros semanais), 2–3 usuários-teste, autor (8–12h/semana), coorientador opcional | Agenda apertada; falta de testers | Pautas objetivas, relatórios curtos, guia de uso simplificado, apoio de colegas/coorientador |
| Orçamento | Software open source (custo zero). Opcional: VPS (~R$ 30–70/mês) para demo. Upgrade de RAM/SSD se necessário. | Sem verba para VPS | Demo local em localhost ou túnel temporário (ngrok/Cloudflared) |
| Acessos/cadastros | GitHub (código), Docker Hub (imagens), MinIO (artefatos). | Limite de armazenamento | Artefatos pesados no MinIO; Git apenas para scripts e metadados |
| Licenças/compliance | Código: MIT ou Apache-2.0. Dependências: respeitar licenças. Dados: citar fontes, usar CC quando aplicável. Privacidade: dados mínimos (nome/e-mail), exclusão, JWT, URLs temporárias. | Risco de não conformidade legal/ética | Listar licenças (THIRD_PARTY_LICENSES.md), aplicar boas práticas de privacidade |

A Tabela 3 apresenta de forma resumida os recursos necessários, seus principais riscos e soluções previstas para garantir a viabilidade técnica e logística do projeto.

### 4.1 Critérios de prontidão (Definition of Ready):

Antes de iniciar a fase de implementação integrada, será checado:

Ambiente conteinerizado: Docker e Docker Compose instalados; docker compose up -d sobe os serviços base — PostgreSQL (db), MinIO (storage) e Redis (fila). [44], [52], [27], [26], [46]

Armazenamento: espaço livre ≥ 100 GB (ou SSD externo configurado).

Repositório: arquivos mínimos presentes — docker-compose.yml, .env.example e README inicial.

Backups: rotinas configuradas — pg_dump diário (PostgreSQL) e mc mirror semanal (MinIO) para replicação. [27], [26]

Gestão: agenda de checkpoints semanais com o orientador definida.

Com esses itens assegurados, o projeto tem viabilidade técnica e operacional para atingir o MVP no semestre e evoluir até a v1 na monografia, com execução reprodutível via containers. [44], [52]

## 5. CONSIDERAÇÕES FINAIS:

Este documento apresentou uma proposta prática e executável para o TCC: construir uma pipeline declarativa, automatizada e conteinerizada que leva da descrição de um leito empacotado (por meio de uma DSL, uma linguagem simples de domínio) até a

simulação CFD utilizando, como exemplo, o OpenFOAM, e a visualização dos resultados em um dashboard web [49], [28], [45], [16], [18], [23], [25], [44], [52]. Os pilares são rastreabilidade (saber como e com quais versões algo foi gerado) [27], [26], [47], reprodutibilidade (mesma entrada levará ao mesmo resultado) [44], [52] e usabilidade (fluxo simples para o usuário).

No lado técnico, foram integrados componentes de código aberto e consolidados, como o Blender (modo CLI), OpenFOAM, FastAPI, React, PostgreSQL e MinIO, todos orquestrados por Docker Compose [14], [16], [18], [23], [25], [27], [26], [44], [52]. A DSL faz a ponte entre a engenharia química e o software: o usuário descreve o leito e as condições (com unidades e limites verificados), o sistema normaliza tudo e executa o restante sem edições manuais de arquivos do CFD ou abertura do Blender com interface [49], [28], [45]. Os resultados saem como variáveis calculadas (como Δp/L, velocidade média, número de células, tempo do solver, resíduos de convergência) e artefatos versionados (STL, VTK, CSV, logs), prontos para análise e comparação [7], [11], [12], [29], [30]. Essa arquitetura resolve dificuldades comuns no fluxo de pesquisa que envolve CAD → CFD → Pós-processamento [9], [10].

No lado metodológico, foram definidos objetivos, perguntas e hipóteses testáveis (produtividade, reprodutibilidade, coerência física e independência de malha). O plano de validação inclui GCI (checagem de sensibilidade à malha) [8], comparação Δp/L × Ergun (coerência com a teoria) [7] e métricas de tempo/uso (benefício prático da pipeline). Cada etapa possui critérios claros para orientar decisões ao longo do desenvolvimento do trabalho.

O cronograma propõe um avanço incremental (MVP até v1), com atividades divididas (DSL, Blender, OpenFOAM, ingestão, API, dashboard, integração e validação), limites semanais e marcos que facilitam ajustes no trabalho [14], [16], [18], [23], [25]. Em paralelo, foram mapeados recursos (hardware, software e pessoas) e planos de contingência (presets didáticos, controle de versões, limites por usuário e backups) caso ocorram erros, reforçando a viabilidade no semestre [27], [26], [44].

Foram adotadas expectativas realistas para a entrega da v1 (leito cilíndrico com esferas e variáveis essenciais). Esses limites controlam o escopo sem perder relevância científica e deixam espaço para trabalhos futuros [11], [12].

Quanto à ética e segurança, foram previstas autenticação JWT, segregação de dados por usuário, registro de proveniência (hash do params.json canônico, seed do empacotamento, versões dos containers, tempos), respeito a licenças e citações adequadas [48], [27], [26], [44], [52]. Ao final, haverá um pacote de replicação (com DOI) para que terceiros possam reproduzir os resultados principais [27].

Por fim, este texto tem como propósito desenvolver um mapa de navegação: declarando o foco do projeto, oferece um roteiro claro para as entregas. Ajustes finos de parâmetros, prazos ou escolhas técnicas são esperados, mas sempre preservando o núcleo da proposta: uma solução baseada em código aberto, reprodutível, fácil de usar, reduzindo as

dificuldades de pesquisadores na simulação CFD de leitos empacotados e elevando a qualidade das análises e pesquisas na área da Engenharia e Tecnologia de Materiais [14], [16], [18], [23], [25], [44], [52].

## 6. REFERÊNCIAS:


[1] LUTZ, Mark. Learning Python. 5. ed. Sebastopol: O'Reilly Media, 2013.

[2] MATTHES, Eric. Python Crash Course: A Hands-On, Project-Based Introduction to Programming. 2. ed. San Francisco: No Starch Press, 2019.

[3] BRITO, Allan. Blender Quick Start Guide: 3D Modeling, Animation, and Rendering with Blender 2.8. Birmingham: Packt, 2018.

[4] VALENZA, Enrico. Blender 3D Cookbook. 2. ed. Birmingham: Packt, 2015.

[5] CONLAN, Chris. The Blender Python API: Add-on Development. Berkeley: Apress, 2017.

[6] SUTHERLAND, Jeff; SUTHERLAND, J. J. Scrum: a arte de fazer o dobro do trabalho na metade do tempo. Rio de Janeiro: Leya, 2014.

[7] ERGUN, Sabri. Fluid flow through packed columns. Chemical Engineering Progress, v. 48, n. 2, p. 89–94, 1952.

[8] ROACHE, Patrick J. Verification and Validation in Computational Science and Engineering. Albuquerque: Hermosa Publishers, 1998.

[9] VERSTEEG, H. K.; MALALASEKERA, W. An Introduction to Computational Fluid Dynamics: The Finite Volume Method. 2. ed. Harlow: Pearson, 2007.

[10] FERZIGER, Joel H.; PERIĆ, Milovan. Computational Methods for Fluid Dynamics. 3. ed. Berlin: Springer, 2002.

[11] POPE, Stephen B. Turbulent Flows. Cambridge: Cambridge University Press, 2000.

[12] WILCOX, David C. Turbulence Modeling for CFD. 3. ed. La Cañada: DCW Industries, 2006.

[13] SCHLICHTING, Hermann; GERSTEN, Klaus. Boundary-Layer Theory. 8. ed. Berlin: Springer, 2000.

[14] BLENDER FOUNDATION. Blender Python API. Disponível em: https://docs.blender.org/api/current/

[15] BLENDER FOUNDATION. Python Console — Blender Manual. Disponível em: https://docs.blender.org/manual/pt/3.5/editors/python_console.html

[16] OPENFOAM FOUNDATION. OpenFOAM documentation: User Guide; snappyHexMesh; simpleFoam. Disponível em: https://www.openfoam.com/documentation/

[17] PYTHON SOFTWARE FOUNDATION. Python 3.x Documentation. Disponível em: https://docs.python.org/3/

[18] FASTAPI. FastAPI Documentation. Disponível em: https://fastapi.tiangolo.com/

[19] PALLETS PROJECTS. Flask Documentation (3.1.x). Disponível em: https://flask.palletsprojects.com/

[20] DJANGO SOFTWARE FOUNDATION. Django overview. Disponível em: https://www.djangoproject.com/start/overview/

[21] OPENJS FOUNDATION. Node.js Documentation. Disponível em: https://nodejs.org/

[22] VERCEL. Next.js Learn. Disponível em: https://nextjs.org/learn

[23] THREE.JS FOUNDATION. Three.js Documentation. Disponível em: https://threejs.org/

[24] BABYLON.JS TEAM (Microsoft). Babylon.js — Web-Based 3D Engine. Disponível em: https://www.babylonjs.com/

[25] PLOTLY. Plotly Python Graphing Library. Disponível em: https://plotly.com/python/

[26] MINIO, Inc. MinIO Documentation. Disponível em: https://min.io/docs/

[27] POSTGRESQL GLOBAL DEVELOPMENT GROUP. PostgreSQL 16 Documentation. Disponível em: https://www.postgresql.org/docs/

[28] LARK PROJECT. Lark: Modern Parsing Library for Python. Disponível em: https://github.com/lark-parser/lark

[29] KITWARE. ParaView Documentation. Disponível em: https://www.paraview.org/documentation/

[30] KITWARE. vtk.js Documentation. Disponível em: https://kitware.github.io/vtk-js/

[31] ESSS. Quarta técnica — Leitura de informações de partículas do ANSYS Fluent para CFD-Post (DPM). Disponível em: https://www.esss.com/blog/quarta-tecnica-leitura-de-informacoes-de-particulas-do-fluent-para-cfd-post-dpm/

[32] PROCESSES (MDPI). Synthetic Packed-Bed Generation for CFD Simulations: Blender vs. STAR-CCM+. Disponível em: https://www.mdpi.com/2305-7084/3/2/52

[33] CUTEC — Clausthal University of Technology. Relatório técnico sobre packed beds (versão aic.17284). Disponível em: https://www.cutec.de/fileadmin/Sites/CUTEC/Documents/Chemische-Energiesysteme/Ver%C3%B6ffentlichungen/aic.17284_Finalpaper__002_.pdf

[34] BLENDERNATION. Using Blender for solving real engineering problems. 2017. Disponível em: https://www.blendernation.com/2017/10/17/using-blender-solving-real-engineering-problems/

[35] BLENDER STACKEXCHANGE. How to create randomly packed bed? Disponível em: https://blender.stackexchange.com/questions/138930/how-to-create-randomly-packed-bed

[36] COUPLER.IO. Top 19 exemplos e templates de dashboards financeiros. Disponível em: https://blog.coupler.io/pt/dashboards-financeiros/

[37] MEDIUM. Essential Tech Stacks You Should Know to Stay Ahead in 2025. Disponível em: https://medium.com/hack-the-stack/essential-tech-stacks-you-should-know-to-stay-ahead-in-2025-41eb53ffd596

[38] REDDIT — r/AskEngineers. How to make a packed bed column for simulations? Disponível em: https://www.reddit.com/r/AskEngineers/comments/1qg74ud/how_to_make_a_packed_bed_column_for_simulations/

[40] RESEARCHGATE. Blender simulation of bed filling with particle array of 250 particles (Figura). Disponível em: https://www.researchgate.net/figure/a-Blender-simulation-of-bed-filling-with-particle-array-of-250-particles-and-b_fig2_366443894

[41] GITHUB — NJANAKIEV, Nikola. blender-scripting: Introduction to Blender scripting. Disponível em: https://github.com/njanakiev/blender-scripting

[42] OPENFOAM FOUNDATION. *OpenFOAM Development Wiki*. Disponível em: https://develop.openfoam.com/Development/openfoam/-/wikis/home

[43] OPENLB DEVELOPMENT TEAM. *OpenLB – Release repository: Doxygen configuration*. Disponível em: https://gitlab.com/openlb/release/-/blob/master/doc/DoxygenConfig

[44] DOCKER, Inc. *Docker Documentation*. Disponível em: https://docs.docker.com/

[45] PYDANTIC. *Pydantic Documentation*. Disponível em: https://docs.pydantic.dev/

[46] REDIS Ltd. *Redis Documentation*. Disponível em: https://redis.io/docs/

[47] SQLALCHEMY. *SQLAlchemy 2.x Documentation*. Disponível em: https://docs.sqlalchemy.org/

[48] JONES, M.; BRADLEY, J.; SAKIMURA, N. JSON Web Token (JWT). *RFC 7519*. IETF, 2015. Disponível em: https://www.rfc-editor.org/rfc/rfc7519

[49] FOWLER, Martin. *Domain-Specific Languages*. Boston: Addison-Wesley, 2010.

[50] PARR, Terence. *Language Implementation Patterns: Create Your Own Domain-Specific and General-Purpose Languages*. Shelter Island: Pragmatic Bookshelf, 2010.

[51] PARR, Terence. *The Definitive ANTLR 4 Reference*. 2. ed. Raleigh: Pragmatic Bookshelf, 2013.

[52] DOCKER, Inc. *Docker Compose – Documentation*. Disponível em: https://docs.docker.com/compose/

[53] TRAEFIK LABS. *Traefik Proxy Documentation*. Disponível em: https://doc.traefik.io/traefik/

[54] NGINX, Inc. *NGINX Documentation*. Disponível em: https://nginx.org/en/docs/

[55] OPENAPI INITIATIVE. *OpenAPI Specification (OAS)*. Disponível em: https://spec.openapis.org/oas/latest.html