INSTITUTO TECNOLÓGICO DE AERONÁUTICA

Bruna Belarmino Silva

MODELAGEM DE ARQUITETURA DE GERENCIAMENTO DO TRÁFEGO NÃO TRIPULADO EM ESPAÇO AÉREO ABAIXO DE 400 PÉS

Trabalho de Graduação
2025

Curso de Engenheria Aeronáutica

CDU 629.73

Bruna Belarmino Silva

MODELAGEM DE ARQUITETURA DE GERENCIAMENTO DO TRÁFEGO NÃO TRIPULADO EM ESPAÇO AÉREO ABAIXO DE 400 PÉS

Orientador

Maj. Av. Lucas Oliveira Barbacovi (ITA)

Coorientador

Prof. Dr. Christopher Schneider Cerqueira (ITA)

ENGENHERIA AERONÁUTICA

São José dos Campos
Instituto Tecnológico de Aeronáutica

2025

Dados Internacionais de Catalogação-na-Publicação (CIP)
Divisão de Informação e Documentação

Silva, Bruna Belarmino
Modelagem de arquitetura de gerenciamento do tráfego não tripulado em espaço aéreo abaixo de 400 pés / Bruna Belarmino Silva.
São José dos Campos, 2025.
135f.

Trabalho de Graduação – Curso de Engenheria Aeronáutica– Instituto Tecnológico de Aeronáutica, 2025. Orientador: Maj. Av. Lucas Oliveira Barbacovi. Coorientador: Prof. Dr. Christopher Schneider Cerqueira.

1. UTM. 2. Drones. I. Instituto Tecnológico de Aeronáutica. II. Modelagem de arquitetura de gerenciamento do tráfego não tripulado em espaço aéreo abaixo de 400 pés.

REFERÊNCIA BIBLIOGRÁFICA

SILVA, Bruna Belarmino. **Modelagem de arquitetura de gerenciamento do tráfego não tripulado em espaço aéreo abaixo de 400 pés**. 2025. 135f. Trabalho de Conclusão de Curso (Graduação) – Instituto Tecnológico de Aeronáutica, São José dos Campos.

CESSÃO DE DIREITOS

NOME DA AUTORA: Bruna Belarmino Silva
TITULO DO TRABALHO: Modelagem de arquitetura de gerenciamento do tráfego não tripulado em espaço aéreo abaixo de 400 pés.
TIPO DO TRABALHO/ANO: Trabalho de Conclusão de Curso (Graduação) / 2025

Bruna Belarmino Silva
Rua H8A, Ap. 135
12228-460 – São José dos Campos–SP

MODELAGEM DE ARQUITETURA DE GERENCIAMENTO DO TRÁFEGO NÃO TRIPULADO EM ESPAÇO AÉREO ABAIXO DE 400 PÉS

Essa publicação foi aceita como Relatório Final de Trabalho de Graduação

---

Bruna Belarmino Silva

Autora

Lucas Oliveira Barbacovi (ITA)

Orientador

Christopher Schneider Cerqueira (ITA)

Coorientador

São José dos Campos, 07 de novembro de 2025.

Dedico este trabalho à minha mãe, que caminhou sobre o chão quente do sertão para que eu pudesse sonhar à sombra.

Agradecimentos

Agradeço, antes de tudo, a Deus, por me capacitar, sustentar e guiar em cada passo da minha vida. De forma generosa, Ele entrelaçou pessoas, caminhos e momentos para que tudo acontecesse com o propósito de me conduzir até aqui.

À minha mãe, Santana, minha maior inspiração desde criança, por me mostrar com o próprio exemplo que é possível sonhar grande e transformar a vida por meio dos estudos. Obrigada por cuidar de mim com tanto amor e por se dedicar incansavelmente — em tempo, esforço e recursos — para oferecer o melhor a mim e aos meus irmãos. Tudo o que sou e o que ainda serei existe por causa de você.

Estendo meus agradecimentos à minha família — ao meu pai, Zaqueu, e, em especial, aos meus irmãos, Pedro e Jéssica. Obrigada por torcerem por mim e me apoiarem nesta jornada, mesmo que, nos últimos anos, a presença de vocês tenha sido mais à distância.

À minha amiga Gabriela, obrigada por estar comigo em cada fase da vida ao longo desses doze anos de amizade. Hoje, ver que tantos dos nossos sonhos de antes se tornaram realidade é motivo de muita felicidade.

Ao Cursinho Exetec, em especial ao professor Marco, obrigada por me ensinar a estudar e por me mostrar que eu podia voar mais alto.

Às instituições de ensino Etapa, Farias Brito e Poliedro, agradeço por acreditarem no meu potencial e por tornarem possível o acesso a uma educação de alta qualidade.

Ao 135 — Emily, Alice e Isabel —, minha segunda família, agradeço por fazerem do H8 um lar e não apenas um lugar. Obrigada por terem cuidado de mim no momento em que mais precisei e em tantos outros. A vocês devo grande parte da minha felicidade no ITA, com a certeza de que Deus não poderia ter sido mais bondoso ao escolher com quem eu moraria nesses cinco anos. À Noa, obrigada por ser um presente nessa reta final.

À Turma 25, a maior turma que já passou pelo H8, agradeço por toda união, muitas vezes materializada em aulões em véspera de prova, e momentos compartilhados. Entramos no ITA em um cenário pandêmico e saímos com uma marca histórica, fruto do nosso esforço coletivo: somos invictos. Dizem que é impossível se formar no ITA sozinho(a) e, felizmente, graças a vocês, eu nunca saberei se é verdade.

Aos colegas da Aeronáutica, agradeço por toda ajuda oferecida e por tornarem o período de aulas mais divertido e suportável. Um agradecimento especial ao meu grupo de laboratório — Henrique, Myla, Rocha, Luigi e Gustavo — pela parceria e dedicação. Esses três anos seriam bem mais difíceis sem vocês.

Ao futsal feminino do ITA, obrigada por tornar minha rotina mais leve e alegre a cada treino e jogo. Ver que conseguimos formar um time competitivo e sólido, mesmo diante de tantos obstáculos, me enche de orgulho.

À minha psicóloga Renata, obrigada por todo o acolhimento e cuidado em um ano de tantas mudanças e recomeços.

Ao professor Christopher, agradeço por ter aceitado embarcar na jornada de me orientar neste trabalho e durante o período de estágio. Estendo meus agradecimentos ao Barbacovi, por assumir essa missão ao longo do processo.

Ao ITA, minha eterna gratidão por ter sido, para mim, mais do que uma instituição de ensino superior. Obrigada pelo ensino, pelas amizades, por abrir tantas portas, por me dar um novo lar e pela transmissão de valores que levarei pelo resto da vida.

À Bruna do passado, obrigada por ter acreditado no propósito e por não ter desistido quando a vida ofereceu outros caminhos. Sua fé e sua luta nos trouxeram até aqui e, hoje, posso dizer: valeu a pena, seu sonho está realizado!

E os sonhos, submersos
e disformes
avolumaram-se engrandecidos,
anelando-se uns aos outros
pulsaram como sangue-raiz
nas veias ressecadas
de um novo mundo.

— Conceição Evaristo

Resumo

O presente trabalho propõe uma arquitetura distribuída para o gerenciamento de tráfego aéreo de drones, com foco em operações abaixo de 400 pés (aproximadamente 120 metros) em espaço aéreo de baixa altitude. A motivação central é atender à crescente demanda por voos não tripulados, garantindo segurança, eficiência e integração com o ecossistema aeronáutico nacional.

A metodologia baseia-se na modelagem de um sistema modular composto por operadores, provedores de serviço e autoridade aeronáutica simulada, que interagem por meio do protocolo MQTT e banco de dados PostgreSQL/PostGIS. O sistema realiza verificações automáticas e hierárquicas, compreendendo análises estáticas, resolução de conflito espacial e temporal, com aplicação de *buffers*, segmentação de rotas 4D e simulação de PVR.

Foram conduzidos dez cenários de simulação para avaliar o desempenho da arquitetura, abrangendo casos de rotas fora do limite de controle, áreas restritas, reservas ativas e múltiplas missões simultâneas. Os resultados confirmaram que o sistema é capaz de detectar conflitos, ajustar rotas e reagendar janelas temporais automaticamente, reproduzindo a lógica de um ambiente UTM federado e escalável.

Conclui-se que a arquitetura proposta representa um passo significativo rumo à implementação de um UTM nacional compatível com as diretrizes do DECEA e da FAA, promovendo um modelo de cooperação, interoperabilidade e segurança operacional para integração de drones no espaço aéreo brasileiro.

Abstract

This study proposes a distributed architecture for Unmanned Traffic Management (UTM), focusing on drone operations below 400 feet in low-altitude airspace. The main goal is to ensure safe, efficient, and integrated airspace management in response to the exponential growth of unmanned aircraft operations.

The proposed system models a modular and cooperative environment comprising operators, service providers, and a simulated air authority, which communicate via the MQTT protocol and a PostgreSQL/PostGIS spatial database. It performs hierarchical verifications through static constraint evaluation, spatial and temporal deconfliction, employing 4D route segmentation, geometric buffers, and Priority Volume Reservations (PVRs) to maintain separation and operational integrity.

A total of eleven simulation scenarios were developed to validate the model, testing cases of geographical limit violations, restricted areas, active PVRs, and simultaneous missions. Results demonstrate that the architecture effectively detects conflicts, reallocates altitudes, and reschedules operations autonomously, ensuring safety and scalability in multi-drone environments.

The system therefore provides a solid foundation for a national UTM model, aligned with FAA and DECEA standards, supporting automation, interoperability, and secure integration of drones into Brazilian airspace.

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

ATM	*Air Traffic Management* (Gerenciamento de Tráfego Aéreo)
BR-UTM	Sistema Brasileiro de Gerenciamento de Tráfego Não Tripulado
CTA	Centro Técnico Aeroespacial
DCTA	Departamento de Ciência e Tecnologia Aeroespacial
DECEA	Departamento de Controle do Espaço Aéreo
DSS	*Discovery and Synchronization Service*
FAA	*Federal Aviation Administration*
MQTT	*Message Queuing Telemetry Transport*
PVR	*Priority Volume Reservation* (Reserva de Volume Prioritário)
PostgreSQL	*Postgre Structured Query Language*
PostGIS	*PostgreSQL Geographic Information System Extension*
SDSP	*Supplemental Data Service Provider* (Provedor de Dados Suplementares)
UAS	*Unmanned Aircraft System* (Sistema de Aeronave Não Tripulada)
UPP	*UTM Pilot Program*
UTM	*Unmanned Traffic Management* (Gerenciamento de Tráfego Não Tripulado)
UVR	*UAS Volume Reservation* (Reserva de Volume para Aeronave Não Tripulada)
WGS84	*World Geodetic System 1984*

Sumário

1 Introdução

Nas últimas décadas, a crescente popularização de drones – formalmente conhecidos como aeronaves não tripuladas (UAS) – tem introduzido um novo paradigma no uso do espaço aéreo. Empregados em aplicações que vão desde o uso recreativo até operações comerciais de entrega, inspeções industriais, mapeamento e segurança pública, os drones vêm se consolidando como uma tecnologia acessível, versátil e estratégica.

Como reflexo desse avanço, observa-se um crescimento contínuo nas operações autorizadas em espaço aéreo de baixa altitude. No Brasil, segundo dados do DECEA, o número de áreas cadastradas no sistema SARPAS saltou de apenas 95 em 2016 para mais de 409 mil em 2024, representando um crescimento de mais de 4.000 vezes em menos de uma década, conforme Figura 1.2.

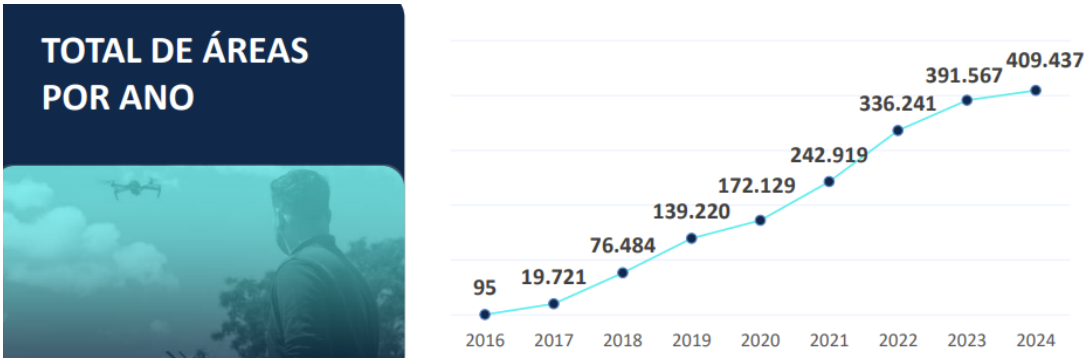


FIGURA 1.1 – Dados de requisições de acesso ao espaço aéreo brasileiro nos últimos anos. Fonte: (Controle do Espaço Aéreo (DECEA), 2025)

Esse crescimento exponencial de tráfego aéreo não tripulado traz desafios significativos para a segurança e a gestão do espaço aéreo, especialmente no que diz respeito à prevenção de colisões e interferências com a aviação tradicional.

Um aspecto crucial é que a maioria dos drones opera em altitudes muito baixas, tipicamente abaixo de 400 pés (cerca de 120 metros) do solo, em espaços aéreos não controlados. De fato, no contexto internacional e nacional, o limite de 400 ft AGL tem sido adotado como referência para delimitar a região de atuação do tráfego de drones. Por exemplo, a diretriz brasileira DCA 351-6/2022 do DECEA define o UTM como o gerenciamento de tráfego aéreo específico em baixas altitudes abaixo de 400 pés, onde

ocorrerão as operações de UAS de forma segura, eficiente e integrada com os demais interessados (Controle do Espaço Aéreo (DECEA), 2022). De modo análogo, o conceito de operações UTM da FAA concentra-se nos voos até 400 ft AGL em espaço aéreo de baixo nível, geralmente classificado como Classe G (não controlado) (FAA, 2022). Assim, os drones tendem a voar em um estrato aéreo inferior, separado das altitudes da aviação tripulada comercial, o que exige soluções dedicadas de gerenciamento.

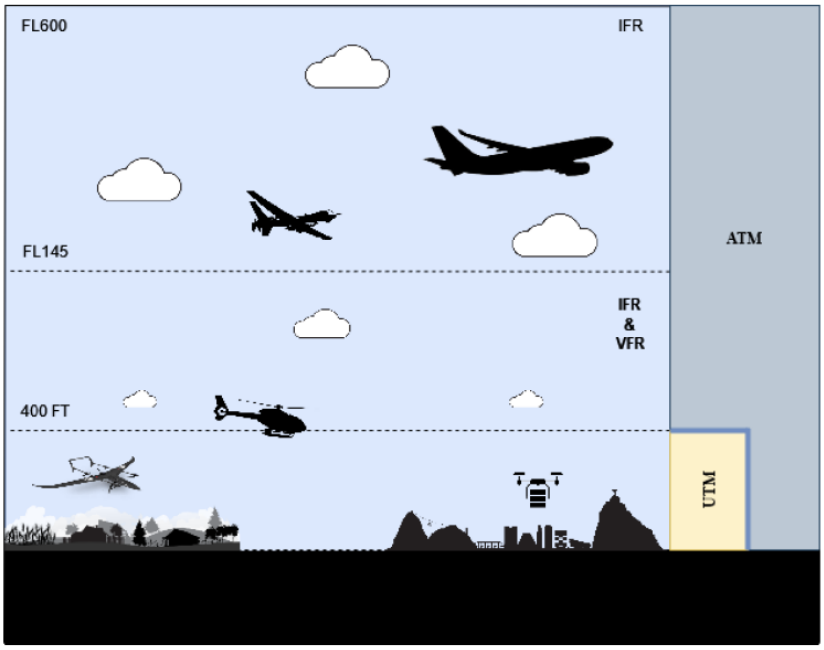


FIGURA 1.2 – Contexto Operacional dos Serviços UTM. Fonte: (Controle do Espaço Aéreo (DECEA), 2022)

Com o aumento da densidade de missões não tripuladas nesse estrato aéreo, surge a necessidade de uma nova abordagem para a gestão dessas operações. O modelo tradicional de controle de tráfego aéreo, baseado em controladores humanos e comunicações por rádio, não é adequado a um ambiente com milhares de voos simultâneos, de curta duração e com diferentes perfis operacionais. O conceito de UTM propõe, portanto, um gerenciamento distribuído, automatizado e baseado em troca digital de informações entre os diversos atores envolvidos – operadores, provedores de serviço, e autoridade aeronáutica.

Ainda assim, a implementação prática do UTM enfrenta desafios técnicos e operacionais. Um dos principais problemas está em como permitir que múltiplos drones compartilhem o mesmo espaço aéreo de forma segura e eficiente, especialmente quando não há supervisão humana direta em tempo real. Missões com rotas e horários distintos, como entregas urbanas, inspeções e monitoramentos, podem se sobrepor e gerar conflitos de trajetória. Além disso, a ausência de rotas fixas e controladas exige soluções que segmentem o espaço e o tempo de forma dinâmica, com alocação de volumes de voo que respeitem critérios de separação entre aeronaves.

Os serviços oferecidos por um sistema UTM são geralmente organizados em três camadas funcionais: (i) a verificação estática, que avalia se a missão desejada respeita restrições fixas do espaço aéreo, como zonas proibidas ou obstáculos; (ii)a resolução de conflito estratégica, que detecta e previne conflitos entre missões ainda na fase de planejamento; e (iii) o gerenciamento tático, que lida com contingências e manobras em tempo real, durante o voo (FAA, 2022).

Este trabalho concentra-se nas duas primeiras camadas – verificação estática e resolução de conflito estratégica. A proposta é desenvolver uma arquitetura que permita que operadores submetam requisições de voo, e que essas requisições sejam processadas de forma automatizada, verificando se a missão respeita as regras de espaço aéreo, não colide com obstáculos e não entra em conflito com outras missões ativas. O sistema também será responsável por responder às requisições com aprovações, ajustes ou negativas, além de manter uma base de dados com os registros das decisões tomadas.

A solução contempla tanto missões com trajetória definida quanto solicitações de reserva de volume prioritário de operação, conhecidas como PVR, previstas na literatura internacional e nas diretrizes do BR-UTM (Controle do Espaço Aéreo (DECEA), 2022). Essas reservas visam garantir exclusividade temporária em determinada região do espaço aéreo, geralmente associadas a missões sensíveis ou de emergência.

Ao adotar esse escopo, o projeto foca em garantir a segurança operacional pré-voo — etapa crítica para a escalabilidade de operações em cenários urbanos e densos — e propõe uma base sólida para futuras extensões.

A motivação principal está em viabilizar, com realismo e aderência às normas atuais, um modelo de coordenação eficiente e automatizada para missões simultâneas, em um cenário que se aproxima das demandas do BR-UTM. O projeto contribui com uma base experimental compatível com os requisitos técnicos estabelecidos por autoridades como o DECEA e FAA, além de fomentar soluções práticas para a integração segura dos drones ao espaço aéreo nacional em um cenário de múltiplas operações simultâneas.

1.1 Objetivo

Este trabalho tem como objetivo geral desenvolver uma arquitetura distribuída para gerenciamento de tráfego aéreo de drones, operando em espaço aéreo de baixa altitude (até 400 pés), com foco em coordenação autônoma de múltiplas missões simultâneas, comunicação entre agentes e verificação de segurança operacional em tempo real.

Para alcançar esse objetivo, serão implementadas as seguintes funcionalidades específicas:

- Desenvolver uma arquitetura modular e distribuída composta por operadores, provedores de serviço e autoridade reguladora;
- Criar um sistema de envio e resposta de requisições de voo, com análise automatizada de viabilidade considerando restrições geográficas, conflitos de rota e disponibilidade de altitude;
- Implementar comunicação entre agentes por meio do protocolo MQTT, permitindo troca de dados em tempo real de forma leve e escalável;
- Segmentar rotas em volumes espaço-temporais e aplicar algoritmos de resolução de conflito para garantir a segurança operacional das missões;
- Simular múltiplos cenários operacionais e registrar logs para validar o funcionamento do sistema e apoiar sua avaliação futura.

2 Fundamentação teórica

2.1 Arquitetura e Atores do Ecossistema UTM

O sistema de Gerenciamento de Tráfego de Aeronaves Não Tripuladas é uma proposta de arquitetura funcional desenvolvida para permitir que múltiplos drones operem simultaneamente, de forma segura e coordenada, em altitudes inferiores a 400 pés, especialmente em espaço aéreo Classe G. Diferente do modelo tradicional de gerenciamento de tráfego aéreo, baseado em controladores humanos e comunicações por rádio, o UTM adota uma abordagem distribuída, automatizada e digital, envolvendo diferentes atores com papéis definidos.

Os principais participantes da arquitetura funcional do UTM são:

- **UAS Operator (Operador):** responsável por planejar e gerenciar a missão de voo, incluindo o envio da intenção de operação e o monitoramento de conformidade durante a missão.
- **USS:** entidade que oferece serviços como verificação de conflitos, coordenação com outros operadores, gerenciamento de restrições e notificações. É o núcleo operacional do sistema.
- **Autoridade aeronáutica (ex.: FAA, DECEA):** supervisiona as operações e define as restrições permanentes ou temporárias do espaço aéreo. Também monitora o cumprimento das normas por meio de acesso contínuo aos dados via interfaces específicas.
- **DSS:** serviço auxiliar de sincronização e compartilhamento de dados entre múltiplos USSs, garantindo que todos os agentes tenham acesso às mesmas informações operacionais.
- **SDSP:** provedor de dados suplementares, como informações meteorológicas, obstáculos, mapas atualizados, NOTAMs e demais dados essenciais ao planejamento e à segurança das operações.

- **Outros stakeholders:** como autoridades de segurança pública e o público em geral, que podem acessar informações UTM específicas quando necessário.

Esse ecossistema é representado de forma clara no modelo da FAA apresentado na Figura 2.1, que mostra como os atores se comunicam e como os dados fluem entre eles para garantir a integridade e a coordenação das operações.

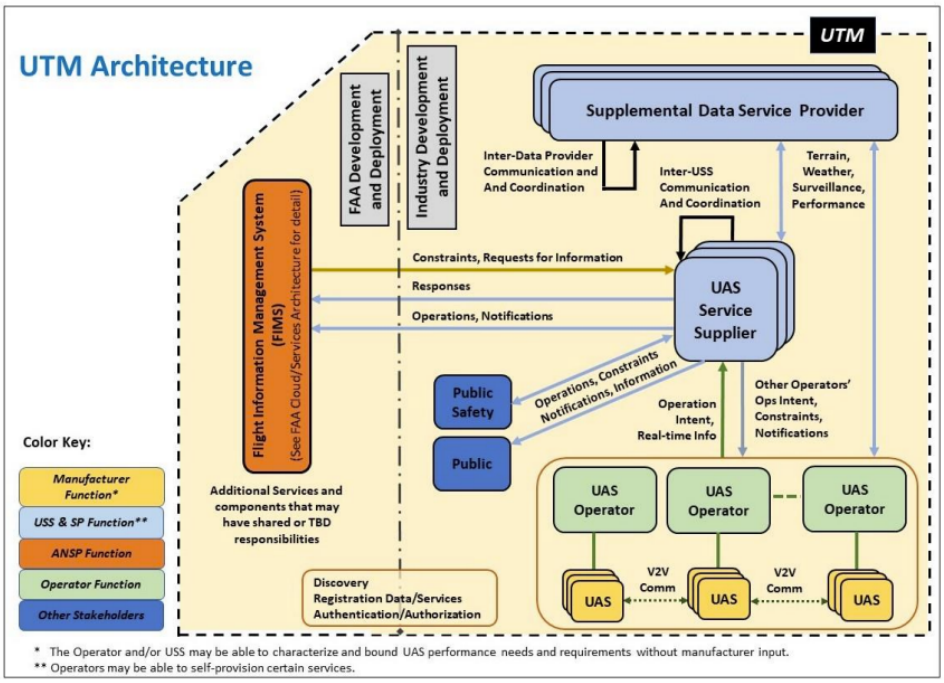


FIGURA 2.1 – Arquitetura UTM. Fonte: (FAA, 2022)

Nesse modelo, os serviços UTM são modulares e operam com base em uma arquitetura federada, onde diferentes USSs gerenciam suas próprias operações, mas compartilham dados com a autoridade e entre si, por meio do DSS e de protocolos definidos. A FAA atua como reguladora, mas não interage diretamente com cada voo; em vez disso, estabelece requisitos, supervisiona provedores, fornece informações de restrição de espaço aéreo e acessa os dados operacionais conforme necessário (FAA, 2022).

Além da FAA, outras iniciativas importantes influenciaram a definição da arquitetura UTM. Nos Estados Unidos, o programa UPP da FAA, conduzido entre 2017 e 2020, foi essencial para validar a viabilidade do modelo federado com múltiplos USSs. Os testes mostraram que a sincronização de intenções de voo, a coordenação descentralizada e a disseminação automática de informações de espaço aéreo são fatores viáveis e essenciais para a escalabilidade do UTM (FAA; NASA, 2021).

No Brasil, a arquitetura de referência proposta pelo DECEA segue princípios semelhantes aos do modelo norte-americano (Controle do Espaço Aéreo (DECEA), 2022). Nela, um módulo central simula o papel do DSS, integrando provedores (PUSS e SDSS) e facilitando a troca de dados operacionais. Esse modelo, alinhado à DCA 351-6/2022,

representa a visão inicial do BR-UTM, o ecossistema nacional em desenvolvimento para a gestão de drones no país.

A partir dessa estrutura, os serviços UTM podem ser organizados em camadas funcionais específicas, com responsabilidades distribuídas entre os atores e módulos do sistema. Esses serviços são apresentados na próxima seção.

2.2 Serviços UTM: Camadas Funcionais

A arquitetura de serviços do UTM é organizada segundo um modelo em três camadas funcionais complementares, cada uma com objetivos específicos, responsabilidades distintas e momentos de atuação diferentes. Essa estrutura foi formalizada nos documentos operacionais da FAA e adotada em programas de validação como o UPP2, com base na gestão cooperativa e distribuída das operações UAS em baixa altitude (FAA; NASA, 2021).

As camadas funcionais são:

- Verificação Estática: avalia, no momento do planejamento da missão, se a operação pretendida respeita as restrições fixas do espaço aéreo. Isso inclui zonas proibidas, áreas de exclusão temporária (NOTAMs), obstáculos cadastrados e limites operacionais definidos por normas. Essas informações são providas pelas autoridades e disseminadas aos operadores via USS e SDSP.

- Resolução de conflito estratégica: identifica e resolve conflitos entre planos de voo antes da decolagem. Compara volumes operacionais 4D (espaço + tempo) entre diferentes missões planejadas, prevenindo sobreposições. Os ajustes podem ser espaciais (mudança de rota ou altitude) ou temporais (reagendamento da janela de operação).

- Gerenciamento Tático: atua durante o voo em situações imprevistas, como desvios, emergências ou entrada de aeronaves não cooperativas. A resposta pode ser executada pelo operador ou por sistemas embarcados de contingência.

Durante o programa *UPP2*, a FAA demonstrou que múltiplos USSs conseguem realizar resolução de conflitos com base na troca de intenções de voo e na negociação de volumes operacionais em rede, validando o modelo descentralizado de separação estratégica (FAA; NASA, 2021).

2.3 Resolução de conflito estratégica e Segmentação 4D

Para que a resolução de conflito estratégica ocorra de forma eficiente, os planos de voo são representados como volumes operacionais 4D — ou seja, blocos de espaço aéreo com limites definidos em latitude, longitude, altitude e tempo. Um voo pode ser dividido em vários segmentos sequenciais, cada um com um tempo de entrada e saída previstos. Essa segmentação permite uma análise mais granular, tanto para otimizar o uso do espaço aéreo quanto para facilitar a identificação de sobreposições com outras operações.

As estratégias de resolução de conflito mais comumente aplicadas são:

- Separação temporal: ajustar os tempos de entrada/saída dos segmentos de voo para evitar ocupação simultânea da mesma região;
- Separação espacial: alterar a posição horizontal (rota) ou vertical (altitude) para garantir afastamento mínimo;
- Separação combinada: aplicar ajustes em mais de uma dimensão (tempo e espaço) simultaneamente.

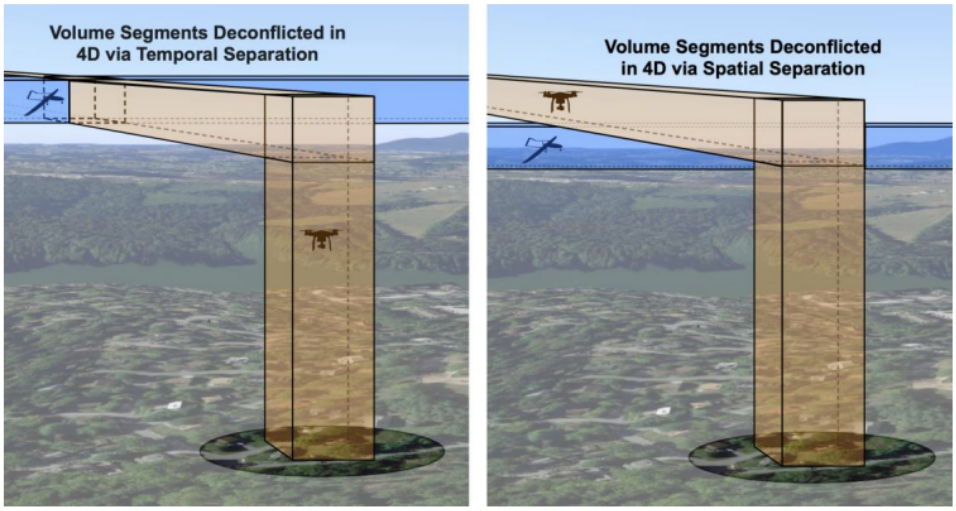


FIGURA 2.2 – Separação de volumes operacionais 4D. À esquerda, via separação temporal; à direita, via separação espacial. Fonte: (FAA, 2022).

Durante o programa UPP2, a FAA validou essa abordagem por meio de simulações e testes de campo. Um dos principais aprendizados foi que a simples segmentação de trajetórias já reduz significativamente a necessidade de coordenação ativa entre operadores (FAA; NASA, 2021). Em ambientes de alta densidade, os provedores USS aplicaram algoritmos que ajustavam automaticamente os volumes de voo conforme recebiam novas intenções, garantindo que não houvesse sobreposição crítica.

2.4 PVR

O PVR é um mecanismo previsto em arquiteturas UTM que permite reservar porções do espaço aéreo, de forma temporária e delimitada, para suportar operações consideradas prioritárias — como missões de emergência, resgate, combate a incêndios ou segurança pública. O conceito é derivado da UAS Volume Reservation, desenvolvido pela FAA e validado durante o programa UPP2 (FAA; NASA, 2021).

Segundo o UTM CONOPS da FAA, as UVRs (ou PVRs) não excluem automaticamente outros operadores do espaço aéreo, mas atuam como alertas formais aos participantes do UTM. Quando um PVR é emitido, os USSs notificam todos os operadores cujas intenções de voo se sobrepõem ao volume reservado. A responsabilidade de evitar a zona recai sobre os operadores, que devem ajustar seus planos conforme necessário para garantir a segurança das operações (FAA, 2022).

A Figura 2.3 apresenta um cenário real do UPP2 no qual uma UVR (indicada pelo círculo laranja) foi estabelecida para permitir o pouso de um helicóptero MedEvac em missão de emergência. A ativação da UVR impacta diretamente o plano da operação ferroviária, que precisa ajustar seu caminho para evitar o volume reservado. Essa mudança, por sua vez, gera conflitos com as operações de entrega 1, 2 e 3 (marcadas com círculos vermelhos), mostrando como uma única reserva pode desencadear efeitos indiretos em cadeia.

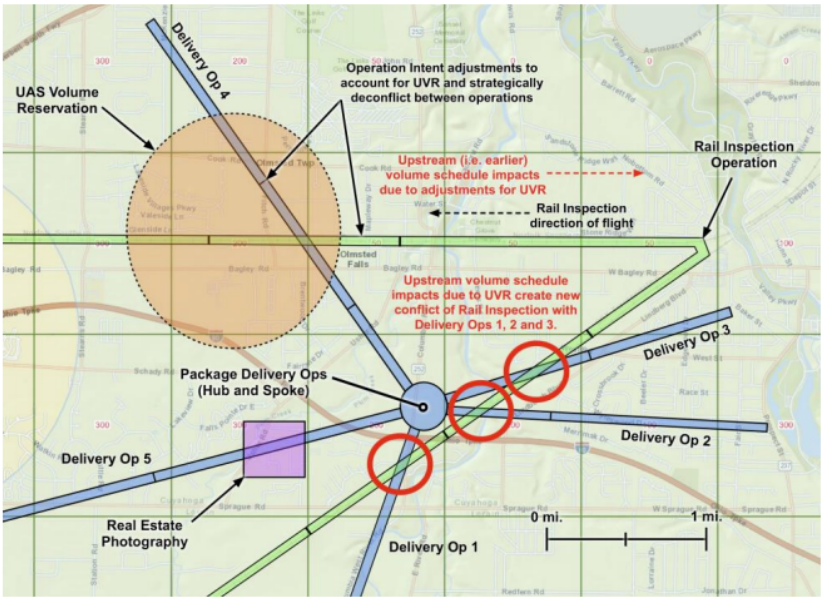


FIGURA 2.3 – Impacto direto e indireto de UVRs sobre operações UAS. Fonte: (FAA, 2022).

A Figura 2.4 ilustra o fluxo de disseminação da UVR no ecossistema UTM. O operador envia o pedido ao USS-PS, que autentica e publica a UVR para a rede de USSs via DSS. Os USSs impactados notificam seus operadores, permitindo que cada um ajuste seu

plano conforme regras pré-configuradas. Essa disseminação automatizada é essencial para manter a consciência situacional em tempo real no sistema.

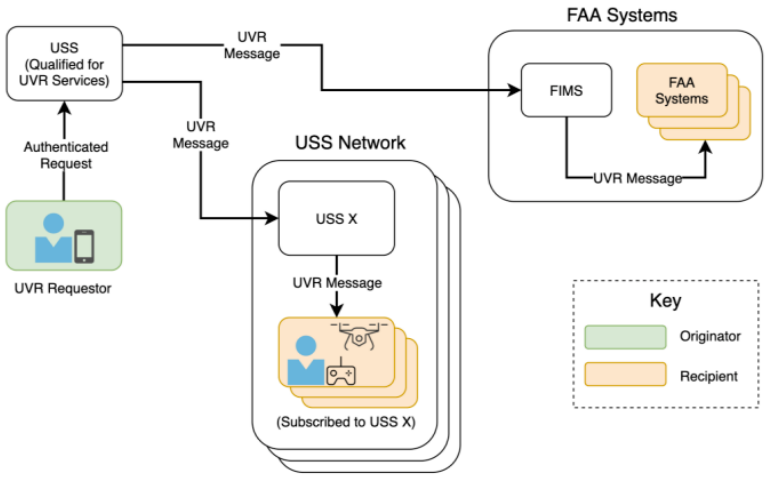


FIGURA 2.4 – Fluxo de disseminação de uma UVR no ecossistema UTM. Fonte: (FAA, 2022).

No contexto brasileiro, a DCA 351-6/2022 também reconhece a necessidade de volumes prioritários, principalmente em operações coordenadas com órgãos públicos. A arquitetura do BR-UTM prevê que esse tipo de informação será compartilhado via provedores de serviço integrados a um sistema central, garantindo ampla disseminação da restrição e rastreabilidade dos acessos (Controle do Espaço Aéreo (DECEA), 2022).

3 Metodologia

3.1 Entidades da Arquitetura UTM modelada

A arquitetura proposta neste trabalho foi modelada de forma distribuída e modular, inspirada nas diretrizes operacionais estabelecidas por órgãos como a FAA, ICAO e DECEA. O sistema simula o funcionamento de um ecossistema UTM (Unmanned Aircraft System Traffic Management), no qual diferentes agentes interagem de maneira cooperativa por meio de uma infraestrutura digital leve e descentralizada. Cada agente desempenha um papel específico no gerenciamento do tráfego aéreo não tripulado, com responsabilidades bem definidas e independência funcional.

Os principais componentes da arquitetura são:

- Drones (ou operadores): responsáveis por submeter as requisições de voo, informando dados como origem, destino, rota preferencial, faixa de horário e faixa de altitude desejada. Cada drone atua como uma instância autônoma capaz de se comunicar com o provedor de serviço.

- USS: processa as requisições recebidas, verifica sua viabilidade (com base em dados geográficos, temporais e operacionais), executa algoritmos de resolução de conflito e responde com a aprovação, negação ou sugestão de ajuste da missão. Também recebe e coordena pedidos de reserva de volume prioritário (PVR) e interage com outras entidades quando necessário.

- Autoridade aeronáutica (simulada): armazena e gerencia as áreas restritas e zonas de exclusão, além de monitorar as decisões e estados das missões registradas. Sua função no protótipo é passiva, representando o papel de supervisão definido pelas normas operacionais reais.

- SDSP: simula a entrega de informações ambientais, como condições meteorológicas locais (vento, chuva, etc.). Esses dados, acessíveis aos provedores USS, poderão futuramente ser incorporados à lógica de decisão, condicionando autorizações de voo a fatores externos. Essa funcionalidade segue o conceito de SDSP definido

pela FAA, que prevê provedores auxiliares para ampliar a consciência situacional do sistema.

3.2 Fluxo Operacional e Lógica de Decisão

O fluxo operacional do sistema foi projetado para executar de forma modular e determinística, coordenando todas as etapas envolvidas na análise de uma requisição de voo — desde a submissão inicial até o envio da resposta final ao operador. Esse encadeamento garante que cada verificação ocorra de maneira lógica e rastreável, conforme ilustrado na Figura 3.1. As decisões são tomadas com base em critérios espaciais, temporais e operacionais, aplicados de forma hierárquica.

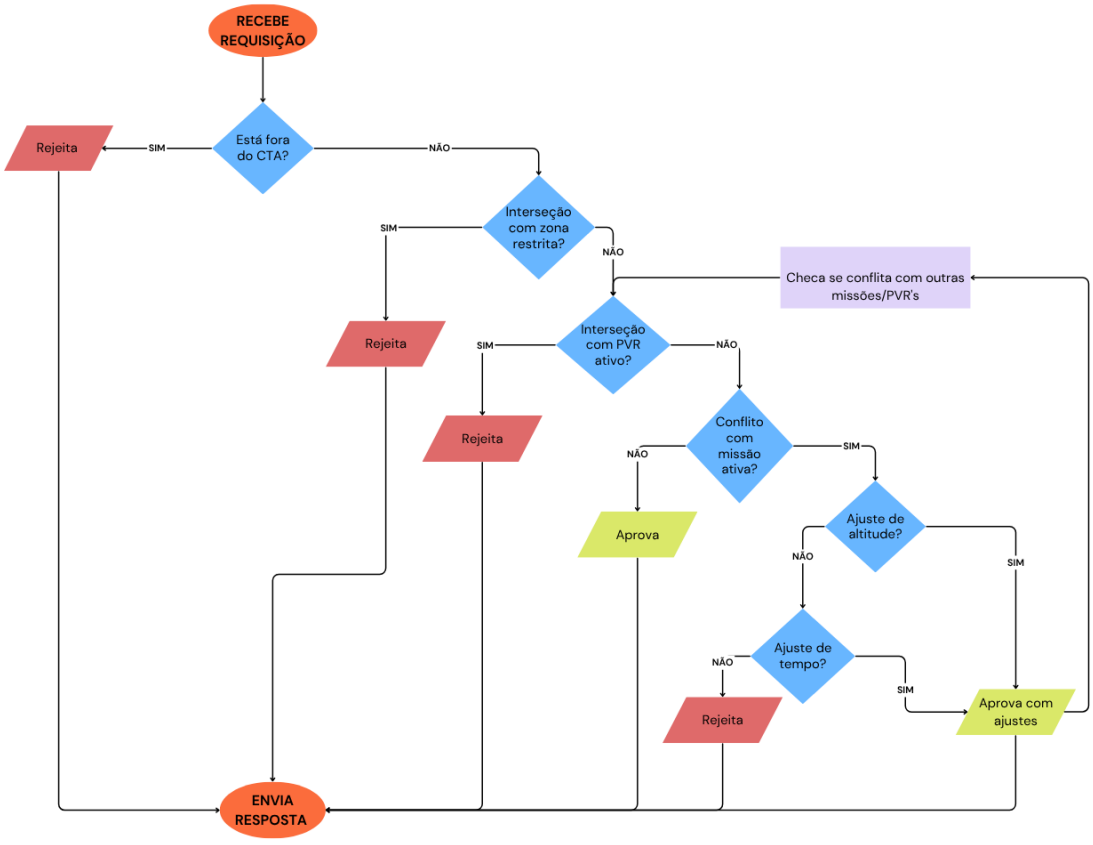


FIGURA 3.1 – Fluxo de lógica de decisão de um provedor.

1. **Submissão e registro da requisição:** O processo se inicia quando o operador envia uma requisição de voo contendo origem, destino, rota intermediária (em formato GeoJSON) e janela temporal desejada. Essa mensagem é publicada no tópico MQTT correspondente ao provedor responsável pela região de operação. O provedor então

registra a solicitação no banco de dados geoespacial, governado pela Autoridade Aeronáutica, associando à trajetória a um volume de espaço e tempo. Nenhuma altitude é atribuída nesta etapa — a definição da faixa vertical ocorre posteriormente, conforme o resultado das verificações e da resolução de conflito.

2. **Verificações hierárquicas de validação:** Após o registro, o provedor inicia uma sequência estruturada de verificações, cada uma representando um ponto de decisão no fluxo da Figura 3.1. O processo ocorre conforme descrito abaixo:

   (a) **Validação geográfica:** Verifica-se se a trajetória submetida está dentro do palco de simulações escolhido para o desenvolvimento do projeto - o DCTA. Caso esteja fora dos limites definidos, a requisição é rejeitada imediatamente e uma resposta negativa é enviada ao operador.

   (b) **Interseção com zonas restritas:** Se a rota estiver dentro do DCTA, o sistema verifica se ela cruza áreas de restrição permanente, como aeroportos, presídios ou zonas militares. Havendo interseção, a requisição é recusada. A Figura 3.2 exemplifica uma rota considerada inválida por atravessar uma área restrita.

FIGURA 3.2 – Exemplo de rota inválida.

   (c) **Interseção com PVR ativo:** O sistema verifica se o volume de voo intersecta algum PVR ativo — áreas de exclusão temporária permitidas pela autoridade simulada. Como tais volumes são intransponíveis, qualquer sobreposição leva à rejeição imediata da requisição, sem tentativa de ajuste.

   (d) **Verificação de conflito com missões ativas:** Se a rota não conflitar com restrições estáticas, o sistema prossegue à análise dinâmica. Nessa fase, compara-se o volume espaço–tempo da nova requisição com os volumes de operações já aprovadas e em vigor. A Figura 3.3 mostra um exemplo de rotas com potencial de sobreposição. Se não houver interseção espacial ou temporal relevante, a operação é aprovada diretamente. Caso contrário, ela é encaminhada ao módulo de decisão para tentativa de resolução.

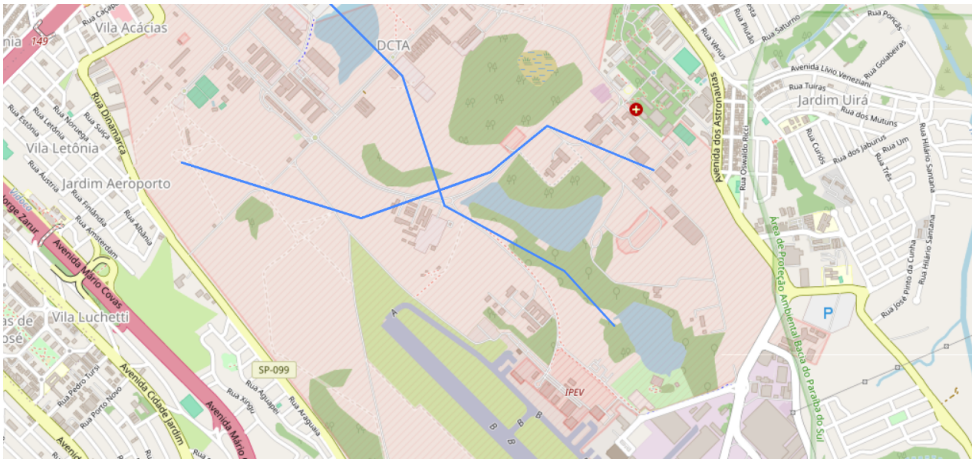


FIGURA 3.3 – Exemplo de rotas possivelmente conflitantes.

3. **Módulo de decisão e resolução de conflito:** Quando detectado um conflito, o provedor aplica a lógica de resolução de conflito hierárquico, tentando primeiro resolver o problema por ajustes espaciais (altitude) e, se necessário, por ajustes temporais (reagendamento). Essas decisões seguem o fluxo exibido no diagrama e utilizam lógicas apresentadas na Seção 3.4.

   (a) **Ajuste de altitude:** O sistema identifica as faixas verticais ocupadas pelas operações existentes e tenta alocar uma nova faixa livre entre o piso e o teto disponíveis para o setor. São respeitadas margens de separação mínima, espessura mínima de voo e o piso absoluto definido pela elevação do terreno. Se for encontrada uma faixa vertical válida, a operação é aprovada com ajustes e a altitude atribuída é registrada no banco.

   (b) **Ajuste de tempo:** Se não houver espaço vertical disponível, o sistema tenta deslocar a janela temporal da operação, mantendo sua duração original. O deslocamento é limitado por um fator máximo. Quando é encontrada uma janela livre e válida, a operação também é aprovada com ajustes. A requisição passa por um processo iterativo para checar se ocorre um novo conflito com PVR ou missão devido ao deslocamento da janela de voo. Caso não seja possível ajustar nem altitude nem tempo, a requisição é rejeitada.

4. **Envio da resposta e monitoramento operacional:** Após a decisão, o provedor envia a resposta via MQTT diretamente ao tópico do drone solicitante. Em caso de aprovação, a missão passa a ser monitorada continuamente. O provedor também dissemina informações operacionais em tempo real, incluindo:

   - novas reservas PVR criadas pela autoridade simulada;
   - alertas meteorológicos emitidos pelo SDSP;
   - alterações no status das missões ativas.

Essas mensagens são redistribuídas aos agentes impactados, promovendo consciência situacional compartilhada e coordenação entre participantes. O sistema, portanto, garante que múltiplas requisições possam ser processadas em paralelo, com alocação eficiente do espaço aéreo e rastreabilidade completa das decisões tomadas.

3.3 Estrutura do Sistema

A implementação da arquitetura foi realizada em linguagem Python, seguindo o paradigma de programação orientada a agentes. Cada entidade funcional do ecossistema UTM foi encapsulada em uma classe independente: o *Provider*, responsável pelo gerenciamento das requisições de voo e pela execução das verificações de conflito (Código A.1); o *Drone*, que representa os operadores e realiza o envio das requisições e mensagens de telemetria (Código A.2); e o SDSP, encarregado de disponibilizar informações suplementares, como dados meteorológicos e zonas geográficas (Código A.3).

Esses componentes são executados como processos autônomos e comunicam-se exclusivamente por meio do protocolo MQTT, conforme descrito nos respectivos códigos listados no Apêndice A. Essa estrutura modular permite a replicação de múltiplos drones e provedores de serviço, bem como a adição de instâncias SDSP independentes, viabilizando experimentos com diferentes densidades de tráfego e condições operacionais. Além disso, a separação em classes facilita a extensão futura do sistema, possibilitando a inserção de novos comportamentos (por exemplo, provedores especializados em resolução de conflito temporal ou drones com modos de operação diferenciados) sem alterar a estrutura principal da arquitetura.

O sistema desenvolvido foi projetado para reproduzir o comportamento de um ambiente UTM distribuído, exigindo mecanismos de comunicação eficientes entre múltiplos agentes e armazenamento consistente de informações espaciais e temporais. Para atender a esses requisitos, optou-se pela combinação do protocolo MQTT e do banco de dados PostgreSQL com extensão PostGIS. O primeiro viabiliza a troca assíncrona de mensagens entre drones e provedores, garantindo comunicação leve e escalável, enquanto o segundo oferece suporte ao armazenamento persistente e às operações geoespaciais necessárias aos processos de verificação e resolução de conflito. Juntas, essas tecnologias permitem integrar, de forma coesa, a camada de comunicação em tempo quase real com a camada de persistência e processamento de dados, assegurando desempenho, rastreabilidade e consistência nas simulações realizadas.

3.3.1 Comunicação via Protocolo MQTT

A comunicação entre os agentes do sistema foi implementada com o protocolo MQTT, adotando o paradigma *publish–subscribe*. Nesse modelo, drones, provedores e o módulo SDSP alternam-se como publicadores (*publishers*) e assinantes (*subscribers*), enviando e recebendo mensagens de forma assíncrona, conforme o contexto operacional. Essa estrutura permite a troca contínua de informações entre múltiplos agentes, mantendo baixo acoplamento entre módulos e garantindo escalabilidade para um número crescente de operações simultâneas.

Os *drones* publicam requisições de voo e PVRs em tópicos destinados aos *providers*, que, por sua vez, analisam as solicitações, aplicam as lógicas de verificação e resolução de conflito, e respondem com autorizações, comandos ou mensagens de status. Além disso, os drones também podem trocar mensagens diretas entre si e assinar atualizações publicadas pelo módulo SDSP, relacionadas a dados meteorológicos ou zonas de restrição temporária.

Os tópicos MQTT foram estruturados de forma hierárquica, permitindo identificar claramente o agente emissor, o destinatário e o tipo de informação publicada. Os principais canais de comunicação implementados no sistema são:

- `drones/{drone_id}/status`: envia periodicamente o estado do drone (posição, identificação e horário da transmissão);
- `drones/{drone_id}/messages`: canal de comunicação direta entre drones (*drone–to–drone*);
- `drones/{drone_id}/commands`: canal de comandos recebidos do provedor responsável;
- `providers/{provider_id}/requests/{drone_id}`: utilizado para o envio de requisições de voo pelos drones;
- `providers/{provider_id}/pvr/{drone_id}`: utilizado para a submissão de PVRs;
- `drones/{drone_id}/request`: canal de resposta do provedor às requisições processadas (aprovação, negação ou ajuste);
- `providers/{provider_id}/sdsp/request`: canal de solicitação de dados suplementares (meteorologia, zonas ativas, etc.) enviado pelo provedor ao módulo SDSP;
- `providers/{provider_id}/sdsp/response`: canal de resposta do SDSP ao provedor, contendo as informações requisitadas;
- `sdsp/status/health`: canal de publicação do estado geral do SDSP (serviços disponíveis e horário de atualização), mantido como mensagem retida;

- `sdsp/weather/now/{region_id}`: canal de disseminação de boletins meteorológicos por região, assinado por provedores e drones;
- `sdsp/geo/zones/{region_id}`: canal de divulgação de zonas temporárias de restrição (*no-fly zones*) emitidas pelo SDSP;
- `drones/{drone_id}/sdsp/subscribe`: utilizado pelo drone para inscrever-se em atualizações meteorológicas ou geográficas de uma região específica;
- `sdsp/drones/{drone_id}/update`: canal de envio direto de notificações do SDSP ao drone inscrito (como alertas de vento forte ou ativação de zonas restritas).

Cada provedor se inscreve automaticamente (*subscribe*) nos tópicos dos drones sob sua responsabilidade, enquanto as funções `on_connect()` e `on_message()` gerenciam, respectivamente, a conexão e o tratamento das mensagens recebidas. O mesmo ocorre para os drones, que permanecem assinantes de seus canais de comando e resposta. A comunicação entre os agentes pode ser visualizada por meio da Figura 3.4.

FIGURA 3.4 – Comunicação entre entidades.

No sistema desenvolvido, o *broker* MQTT atua como núcleo de comunicação e coordenação do ambiente UTM, funcionando como ponto central de distribuição das mensagens entre todos os agentes. Os provedores operam como módulos conectados a esse *broker*, aplicando localmente a lógica de decisão e de resposta conforme as mensagens recebidas. Já o SDSP atua como um provedor suplementar de dados, publicando boletins e respondendo sob demanda, de modo a enriquecer o contexto situacional das operações. Essa arquitetura reflete o conceito de um ecossistema federado, no qual múltiplos provedores e serviços interoperam de maneira coordenada através de um canal de comunicação comum,

garantindo coerência operacional, escalabilidade e integridade das informações trocadas entre os agentes do sistema.

3.3.2 Estrutura do Banco de Dados

O sistema foi implementado sobre PostgreSQL, acrescido da extensão PostGIS, devido à necessidade de manipular simultaneamente informações espaciais e temporais. Essa escolha se justifica pela robustez do PostgreSQL no tratamento de transações e pela capacidade do PostGIS de realizar operações geométricas nativas, como *buffers*, interseções e cálculos de distância. Essa combinação é amplamente empregada em aplicações de gerenciamento de tráfego aéreo, GIS e UTM, pois permite a integração direta entre geometria (trajetórias, volumes) e tempo (janelas de voo, reservas temporárias).

O banco de dados foi modelado para representar o ciclo completo de uma operação UTM, desde a submissão da requisição de voo até a geração da missão aprovada, incluindo o registro de segmentos intermediários, áreas restritivas (PVRs e mapas) e as informações dos principais atores do sistema — provedores, operadores e drones. Essa estrutura relacional facilita a execução de consultas espaciais e temporais durante os processos de verificação de restrições e resolução de conflito, garantindo integridade dos dados e rastreabilidade entre as diferentes etapas do processamento. As tabelas utilizadas no banco de dados, bem como as principais finalidades e campos, estão listadas na Tabela 3.1.

TABELA 3.1 – Entidades principais do banco de dados e campos relevantes

Tabela	Finalidade	Campos (nome, tipo)
requisicoes	Entrada do plano de voo (rota e janela temporal)	`request_id (integer), drone_id (integer), start_point (geometry), end_point (geometry), waypoints (geometry), start_time (timestamp), end_time (timestamp), provider_id (integer)`
segmentos	Decomposição 4D da requisição em trechos menores	`segment_id (integer), request_id (integer), segmento (geometry), start_time_min (timestamp), start_time_max (timestamp), end_time_min (timestamp), end_time_max (timestamp), alt (double precision)`
missoes	Missões aprovadas (trajeto final e faixa de altitude)	`mission_id (integer), request_id (integer), drone_id (integer), track (geometry), alt_min (double precision), alt_max (double precision), start_time (timestamp), end_time (timestamp)`
pvr	Priority Volume Reservation (áreas de restrição temporária)	`pvr_id (integer), drone_id (integer), provider_id (integer), start_time (timestamp), end_time (timestamp), status (varchar), created_at (timestamp), updated_at (timestamp), area (geometry)`
drones	Cadastro e metadados das aeronaves	`drone_id (integer), model (text), operator_id (text), status (text), last_updated (timestamp), uas_type (text), weight (double precision), provider_id (integer), visual_type (text)`
maps	Camadas espaciais de referência (DCTA e zonas restritas)	`id (integer), name (text), geom (geometry), zone_type (varchar), height (double precision), category (varchar), last_update (timestamp), alt (double precision)`
operadores	Cadastro de operadores do sistema	`operator_id (varchar), name (varchar), contact_info (varchar)`
provedores	Registro dos provedores UTM simulados	`provider_id (integer), nome (varchar), contato (varchar)`

O modelo de dados apresentado foi desenvolvido de forma a representar integralmente o ciclo de processamento de uma operação UTM, desde a submissão de uma requisição de voo até a autorização final emitida pelo provedor. Cada tabela cumpre um papel

específico dentro dessa cadeia de decisão. A tabela `requisicoes` constitui o ponto de entrada das informações do plano de voo, armazenando a rota proposta (em formato geométrico) e a janela de tempo desejada. A partir desses dados, o sistema gera registros na tabela `segmentos`, que decompõem cada trajetória em trechos espaciais e temporais discretizados. Essa segmentação permite que o processo de verificação atue de forma mais granular, identificando interseções e sobreposições em intervalos de tempo reduzidos.

Durante o processo de validação, o provedor consulta as camadas de referência armazenadas em `maps`, que representam o espaço aéreo controlado e suas zonas restritivas, e verifica possíveis conflitos com áreas de reserva registradas em `pvr`. As interseções espaciais são calculadas diretamente no banco de dados utilizando funções da extensão PostGIS, como `ST_Buffer` e `ST_Intersects`, enquanto as verificações temporais empregam o operador `OVERLAPS`, responsável por avaliar a sobreposição entre janelas de tempo. Os resultados aprovados são registrados na tabela `missoes`, com a geometria final do trajeto (`track`) e as faixas de altitude definidas (`alt_min`, `alt_max`), permitindo rastreabilidade completa entre a requisição original, seus segmentos analisados e a missão efetivamente autorizada.

Além dessas tabelas operacionais, o banco de dados também armazena informações sobre os principais atores do sistema. As tabelas `drones`, `operadores` e `provedores` registram os metadados das aeronaves, dos operadores responsáveis e dos provedores UTM simulados, respectivamente. Esses registros garantem a vinculação entre cada requisição de voo e o agente responsável por sua execução e gestão, refletindo a arquitetura federada prevista nos conceitos de gerenciamento de tráfego aéreo não tripulado. Assim, a estrutura de dados adotada fornece suporte completo tanto ao processamento técnico das missões quanto ao gerenciamento das entidades envolvidas no ambiente UTM.

3.4 Estratégias de verificação e resolução de conflito

3.4.1 *Buffers* Espaciais e Verticais

O sistema emprega *buffers* espaciais e verticais como parâmetros essenciais para uniformizar a sensibilidade das verificações de conflito. Eles são utilizados em todas as etapas do processo — desde as verificações estáticas (DCTA, áreas restritas e PVRs) até a resolução de conflito espacial e temporal — assegurando coerência entre as análises geométricas e temporais. O *buffer* de um trecho de trajetória pode ser visualizado na Figura 3.5.

No domínio horizontal, as trajetórias são expandidas lateralmente por um *buffer* aplicado em ambos os lados da rota (b_{esp}), por meio da função `ST_Buffer(waypoints::geography,` b_{esp}`)::geometry` do PostGIS, criando uma margem de segurança para detectar interse-

ções (`ST_Intersects`) e proximidades (`ST_DWithin`). Em comparações entre trajetórias, essa margem é aplicada simetricamente, resultando em um afastamento efetivo de aproximadamente ($2b_{esp}$) entre voos. No domínio vertical, adota-se uma faixa de operação de largura fixa (h_{voo}) entre janelas simultâneas de voo. Todos os valores de *buffer* são parametrizáveis no sistema e definidos na Tabela 3.3.

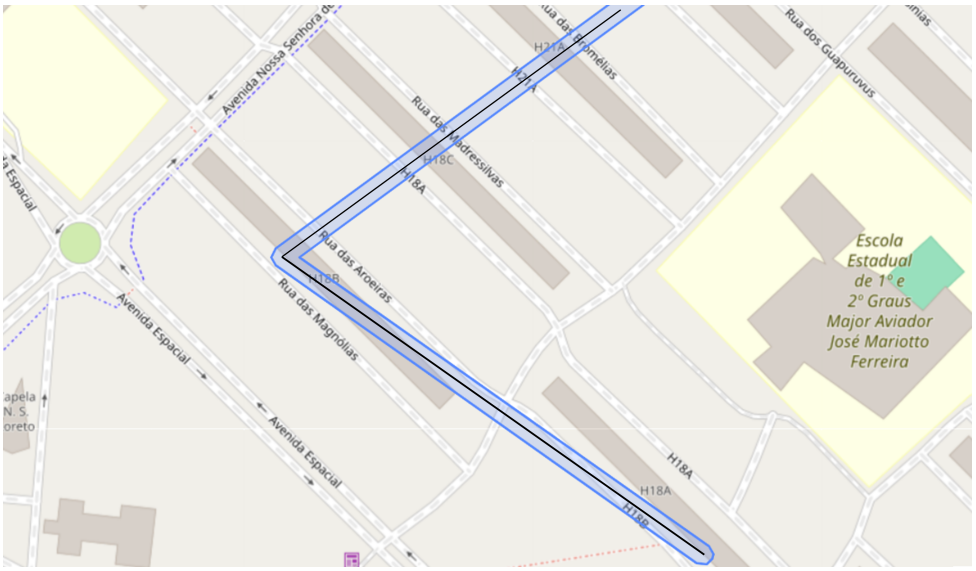


FIGURA 3.5 – Trecho de trajetória de missão com *buffer* no plano de rota.

Durante a resolução de conflito temporal, aplica-se também um *buffer* no tempo (Δ_t), expandindo as janelas de voo antes e depois do intervalo real de operação. Essa folga temporal reproduz a separação estratégica entre missões consecutivas. Assim, o sistema mantém margens consistentes nos três domínios — espacial (b_{esp}), vertical (h_{voo}) e temporal (Δ_t) — garantindo segurança e estabilidade nas decisões de verificação e resolução de conflito.

3.4.2 Segmentação da rota

A segmentação da rota é utilizada para representar a trajetória de forma discretizada, permitindo que o sistema analise o voo em partes menores e com resolução temporal adequada. Cada requisição submetida é decomposta em diversos segmentos de comprimento máximo (L_{seg}), armazenados na tabela `segmentos`, contendo a geometria individual de cada trecho (`segmento`) e as janelas temporais associadas (`start_time_min`, `end_time_max`). Essa estrutura possibilita ao sistema avaliar interseções espaciais e temporais em nível mais detalhado do que o fornecido pela rota completa. Uma trajetória segmentada espacialmente pode ser visualizada na Figura 3.6, em que cada cor representa um segmento diferente.

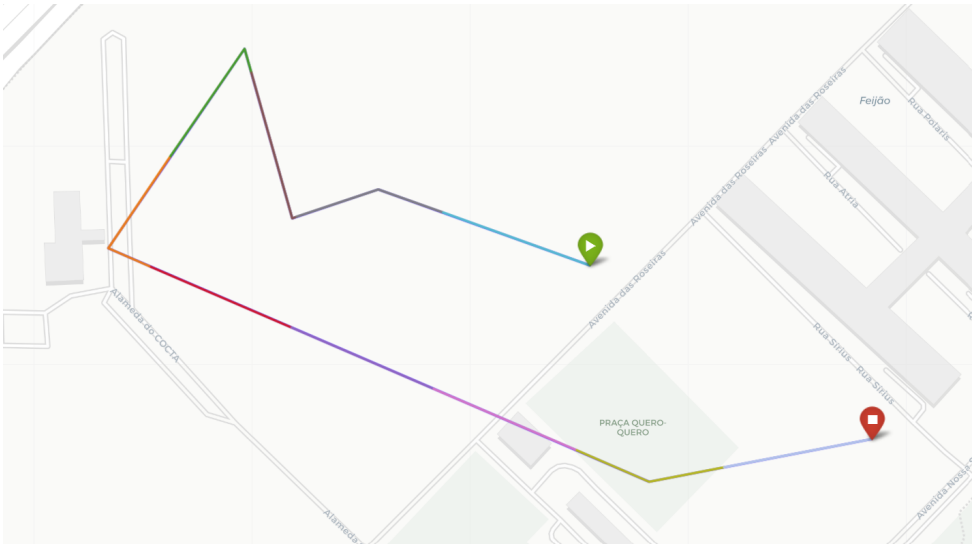


FIGURA 3.6 – Trajetória segmentada.

O uso de segmentos é essencial para os processos de verificação e resolução de conflito, pois reduz o número de negações desnecessárias. Sem essa discretização, duas trajetórias que se cruzam geometricamente seriam consideradas conflitantes, mesmo que nunca ocupassem o mesmo ponto no tempo. Com a segmentação, o sistema distingue cruzamentos espaciais sem sobreposição temporal, permitindo aprovar voos que seriam indevidamente negados em uma análise puramente geométrica.

A discretização temporal dos segmentos é proporcional à distância percorrida e à duração da missão. Assim, cada segmento recebe uma janela de tempo mínima e máxima que define o intervalo de possível ocupação do espaço aéreo. Essa modelagem 4D (espaço e tempo) fornece a base para as etapas seguintes de verificação de altitude e resolução de conflito temporal, garantindo coerência entre o comportamento simulado e a dinâmica real de um ambiente UTM. Para as simulações realizadas, o último segmento possui comprimento menor ou igual a (L_{seg}).

3.4.3 Determinação de Piso e Teto

Os limites verticais de cada operação, denominados *piso* (alt_{piso}) e *teto* (alt_{teto}), são determinados automaticamente a partir da relação entre a trajetória do voo e o relevo subjacente. A altitude do terreno em cada ponto i da rota é obtida por meio da API do Google Elevation, convertida de metros para pés conforme a Equação 3.1:

$$h_i = 3.281 \times H_i^{(m)} \tag{3.1}$$

em que h_i é a altitude do terreno em pés e $H_i^{(m)}$ representa a elevação em metros retornada pela API.

O piso operacional é definido como o ponto mais alto interceptado pela rota, considerando tanto a altitude de voo prevista a_i quanto a elevação combinada do relevo e das estruturas verticais representadas em `maps`. Essa relação é expressa pela Equação 3.2:

$$\text{alt}_{piso} = \max\left(\max_i(a_i),\ \max_j(h_j + \text{height}_j)\right) \tag{3.2}$$

onde a_i é a altitude de voo no ponto i, h_j é a altitude do terreno no polígono j, e height_j é a altura da estrutura ou obstáculo associada a esse polígono. Dessa forma, o piso corresponde ao nível mais elevado entre a rota e o relevo interceptado, assegurando que a operação não ocorra abaixo de nenhum obstáculo.

O teto operacional é determinado a partir da menor altitude observada na trajetória, acrescida de uma margem vertical de 400 pés, conforme a Equação 3.3:

$$\text{alt}_{teto} = \min_i(a_i) + 400 \tag{3.3}$$

onde $\min_i(a_i)$ é a menor altitude de voo ao longo da rota. Essa margem fixa garante separação vertical adequada entre camadas de voo e múltiplas operações simultâneas.

Por fim, a faixa vertical efetiva da operação é obtida pela diferença entre o teto e o piso, como na Equação 3.4:

$$\Delta h = \text{alt}_{teto} - \text{alt}_{piso} \tag{3.4}$$

sendo Δh a espessura da faixa vertical disponível para a operação, a qual deve satisfazer $\Delta h \geq \Delta h_{safe}$ para garantir uma espessura mínima operacional de segurança. Todos os parâmetros são definidos na Tabela 3.3.

O emprego de um referencial altimétrico absoluto e comum (nível médio do mar) assegura que todas as missões utilizem a mesma origem vertical, evitando discrepâncias entre bases de dados e garantindo consistência global nas verificações de altitude e nos processos de resolução de conflito.

Essas definições permitem ajustar automaticamente os limites verticais da operação conforme o relevo local, garantindo que a trajetória mantenha separação segura em relação ao terreno e às demais missões em execução.

3.4.4 Resolução de conflito espacial por altitude

A partir dos limites verticais definidos anteriormente, o próprio sistema — e não o usuário — determina automaticamente a faixa de altitude ideal para a operação. Essa

decisão autônoma tem como objetivo aumentar a densidade de missões simultâneas e reduzir o número de negações desnecessárias.

A janela operacional de voo é definida como uma faixa vertical de largura fixa h_{voo}.

Essa faixa é posicionada entre 10 e 30 metros acima do piso operacional (alt_{piso}), de acordo com a Equação 3.5:

$$\text{alt}_{nominal} = \text{alt}_{piso} + [10, 30] \text{ m} \approx [33, 98] \text{ ft} \tag{3.5}$$

O intervalo definido pela Equação 3.5 representa a altitude nominal da missão, utilizada inicialmente pelo sistema para alocação vertical. Caso existam outras missões ativas ocupando o mesmo espaço aéreo, o algoritmo de resolução de conflito por altitude ajusta automaticamente essa faixa dentro dos limites entre alt_{piso} e alt_{teto}.

Durante esse processo, o sistema coleta todas as faixas verticais das missões em conflito e aplica margens de segurança de $\pm\delta$ metros em torno de cada janela, conforme a Equação 3.6:

$$[h_{min}, h_{max}] \;\rightarrow\; [h_{min} - \delta,\; h_{max} + \delta] \tag{3.6}$$

A partir das janelas expandidas, o sistema identifica automaticamente intervalos verticais livres entre o piso e o teto disponíveis para alocação. Se uma faixa compatível com a largura h_{voo} for encontrada, a missão é autorizada na nova altitude; caso contrário, o sistema indica que a resolução de conflito por altitude não pôde ser resolvido.

Essa abordagem, que inicia a alocação nas camadas inferiores e progride verticalmente à medida que surgem novos conflitos, permite melhor aproveitamento do espaço aéreo e reduz a necessidade de resolução de conflitos temporais. Assim, o sistema prioriza soluções espaciais e automáticas de separação, garantindo maior eficiência e consistência entre operações simultâneas.

3.4.5 Resolução de conflito temporal

A resolução de conflito temporal é acionado quando não há espaço vertical suficiente para acomodar uma missão. Nessa situação, a geometria da rota é mantida fixa, e apenas o intervalo de tempo de voo é ajustado. O objetivo é deslocar a janela temporal de operação, sem alterar sua duração, de modo a evitar sobreposições com outras missões ativas que compartilham o mesmo espaço aéreo. O processo é conduzido no nível dos segmentos da rota, de forma que apenas os trechos que se cruzam espacialmente são avaliados quanto à compatibilidade temporal.

1. Identificação de conflitos. O sistema compara os segmentos da missão analisada com os segmentos de missões já aprovadas. Sempre que dois segmentos se intersectam espacialmente, o sistema coleta os respectivos intervalos de tempo $[t_i^{(s)}, t_i^{(e)}]$, correspondentes ao início e ao fim da ocupação do espaço aéreo, e verifica se há sobreposição segundo a condição:

$$ST\_Intersects(s_i, s_j) = \text{True} \quad \wedge \quad [t_i^{(s)}, t_i^{(e)}] \cap [t_j^{(s)}, t_j^{(e)}] \neq \emptyset$$

Se ambas as condições forem verdadeiras, considera-se que há conflito espaço-temporal entre os segmentos s_i e s_j.

2. Acolchoamento temporal. Para criar uma margem de segurança entre voos consecutivos, cada intervalo de tempo é expandido em Δ_t segundos antes e depois da sua duração real:

$$[t_i^{(s)}, t_i^{(e)}] \rightarrow [t_i^{(s)} - \Delta_t,\ t_i^{(e)} + \Delta_t] \tag{3.7}$$

TABELA 3.2 – Etapas da resolução de conflito temporal (síntese).

Etapa	**Ação realizada**	**Resultado**
1	Identifica os **segmentos que apresentam conflito espaço-temporal** com outras missões ativas	Seleção dos trechos sobrepostos
2	Aplica **acolchoamento temporal** ($\pm\Delta t$) para ampliar a margem de segurança entre operações	Criação de intervalo de segurança expandido
3	Define o **deslocamento máximo permitido** como fração do tempo total da missão ($\lvert\Delta T\rvert \leq \rho T$)	Limite de ajuste temporal configurado
4	Verifica se **adiantar ou atrasar a janela dentro desse limite** elimina o conflito	Aprovação automática ou negação da missão

Esse processo é aplicado individualmente a cada segmento, simulando uma zona de segurança temporal ao redor da operação.

3. União de intervalos ocupados. Após o acolchoamento, os intervalos são ordenados e mesclados para formar um conjunto único de janelas ocupadas:

$$U = \bigcup_i [t_i^{(s)}, t_i^{(e)}]$$

Esse conjunto U representa todos os períodos em que o espaço aéreo está temporariamente indisponível para novas missões.

4. Busca por uma nova janela. Mantendo a duração total da missão constante, definida por

$$T = t_f - t_0, \tag{3.8}$$

o sistema procura um novo intervalo $[t'_0, t'_f]$ com a mesma duração T, mas sem sobreposição com as janelas ocupadas:

$$[t'_0, t'_f] \cap U = \emptyset \quad \text{e} \quad t'_f - t'_0 = T \tag{3.9}$$

O deslocamento máximo permitido é definido em função da duração total do voo e de um fator ρ que controla a flexibilidade da realocação:

$$|\Delta T| \leq \rho T \tag{3.10}$$

em que $\Delta T = t'_0 - t_0$ representa o deslocamento aplicado à janela original. A busca é feita de forma incremental, em passos discretos de δt, até encontrar a primeira posição livre que satisfaça a Equação 3.9.

5. Rechecagem e validação. Uma vez identificada uma nova janela temporal, o sistema realiza uma verificação completa, testando novamente as interseções espaciais e temporais com missões e PVRs ativos. Se o novo intervalo não introduzir novos conflitos, o ajuste é aceito; caso contrário, considera-se que a resolução de conflito temporal falhou.

Os parâmetros Δ_t, ρ e δt tornam o algoritmo ajustável a diferentes contextos operacionais. Essa abordagem busca equilibrar flexibilidade e segurança, utilizando ajustes temporais apenas quando não há alternativas verticais disponíveis. Com isso, o sistema mantém coerência com o modelo hierárquico de separação adotado no UTM, priorizando primeiro soluções espaciais e depois temporais, o que garante melhor aproveitamento do espaço aéreo e consistência entre as operações simultâneas.

3.4.6 Parâmetros Adotados nas Simulações

A Tabela 3.3 apresenta os valores de parâmetros utilizados nas simulações, abrangendo *buffers* espaciais, verticais e temporais, além dos fatores de segmentação e deslocamento temporal.

TABELA 3.3 – Parâmetros utilizados nas simulações.

Parâmetro	**Símbolo**	**Valor adotado**
Comprimento do segmento	L_{seg}	100 m ($\approx$ 328.1 ft)
Buffer espacial lateral	b_{esp}	5 m ($\approx$ 16.4 ft)
Separação efetiva entre trajetórias	$2b_{esp}$	10 m ($\approx$ 32.8 ft)
Largura da janela vertical	h_{voo}	20 m ($\approx$ 65.6 ft)
Posicionamento nominal acima do piso	h_{nom}	$[10, 30]$ m ($\approx$ [32.8, 98.4] ft)
Buffer vertical de segurança	δ	$\pm$5 m ($\approx \pm$16.4 ft)
Espessura mínima livre	Δh_{min}	30 m ($\approx$ 98.4 ft)
Acolchoamento temporal	Δ_t	60 s
Passo de busca temporal	δt	15 s
Fator máximo de deslocamento	ρ	0.5 e 0.05

3.5 Cenários de Simulação

Para validar o comportamento da arquitetura proposta, foram definidos cenários de simulação que testam individualmente as funcionalidades desenvolvidas. A ideia central não é apenas verificar o funcionamento isolado de cada módulo, mas também observar a interação entre eles em situações operacionais variadas.

As simulações serão executadas em uma região geográfica baseada no entorno do Departamento de Ciência e Tecnologia Aeroespacial (DCTA), localizado em São José dos Campos (SP). Essa área foi escolhida por apresentar:

- presença de zonas sensíveis (escolas, aeroporto, áreas residenciais, áreas militares sensíveis);
- presença de variação de altitude (região do aeroporto), relevante para a obtenção de elevação;
- áreas livres que permitem missões não conflitantes;

- potencial realista de testes com PVRs.

As trajetórias dos drones serão submetidas ao sistema com parâmetros controlados, e a resposta dos módulos será analisada conforme os objetivos de cada caso. A Tabela 3.4 resume os onze cenários considerados.

TABELA 3.4 – Casos de simulação realizados para validação do sistema

Nº	**Objetivo**	**Descrição do Caso**	**Resultado Esperado**
1	Verificação de limite geográfico (DCTA)	Submeter rota parcialmente fora do polígono do DCTA	Requisição negada na verificação estática
2	Interseção com zona restrita	Trajetória cruza polígono classificado como restrito em `maps`	Requisição negada por restrição estática
3	Interseção com PVR ativo	Trajetória intercepta área reservada de PVR vigente no tempo	Requisição negada por conflito com PVR ativo
4	Missão de referência aprovada	Rota válida dentro do DCTA, sem conflitos espaciais/temporais	Requisição aprovada sem ajustes (altitude e tempo mantidos)
5	resolução de conflito por altitude entre missões	Novas requisições na mesma rota e janela; alocação sequencial em faixas superiores	Missões subsequentes aprovadas com ajuste de altitude; negativa ao atingir o teto operacional
6	Conflito a nível de segmento — aprovado	Rotas se cruzam espacialmente, mas sem sobreposição temporal nos segmentos	Requisição aprovada (sem ajuste); sem conflito espaço–temporal
7	Conflito a nível de segmento — negado	Rotas com interseção espacial e temporal no mesmo segmento; sem faixa vertical disponível	Requisição negada (impossível resolver por altitude)
8	Resolução de conflito temporal — aprovado	Falta de espaço vertical; busca de janela livre no tempo respeitando limite de deslocamento de intervalo de tempo	Requisição aprovada com janela reprogramada
9	Resolução de conflito temporal — negado	Mesmo cenário, porém com deslocamento de tempo insuficiente para eliminar a sobreposição	Requisição negada (sem janela livre dentro do limite de ajuste)
10	Rechecagem pós-ajuste temporal	Após realocação no tempo, checar conflito com PVR/ missões ativas	Requisição negada por novo conflito detectado na rechecagem

Esses casos cobrem as principais camadas funcionais da arquitetura — verificação estática, resolução de conflito estratégica e resposta operacional. Juntos, fornecem uma base sólida para avaliar a robustez, escalabilidade e realismo do sistema desenvolvido.

4 Resultados

4.1 Requisição fora dos limites do DCTA

4.1.1 Descrição do cenário

Este cenário tem como objetivo validar a verificação estática dos limites do espaço aéreo controlado (DCTA). A simulação representa uma requisição cujo plano de voo ultrapassa parcialmente os limites definidos no mapa base.

As informações operacionais da requisição, incluindo o drone, o provedor e o operador associados, encontram-se resumidas na Tabela 4.1, enquanto a trajetória da requisição pode ser visualizada na Figura 4.1. Esperava-se que o sistema identificasse automaticamente a violação dos limites do DCTA e negasse o voo durante a etapa de verificação estática, sem avançar para as fases de resolução de conflito.

TABELA 4.1 – Parâmetros operacionais da requisição analisada no cenário

Parâmetro	**Valor**
Drone associado	DJI Phantom 4 (`drone_id = 4`)
Provedor responsável	`provider_id = 1`
Operador vinculado	`operator_id = 68`
Início da operação solicitada	28/11/2025 – 08:00:00
Final da operação solicitada	28/11/2025 – 08:30:00

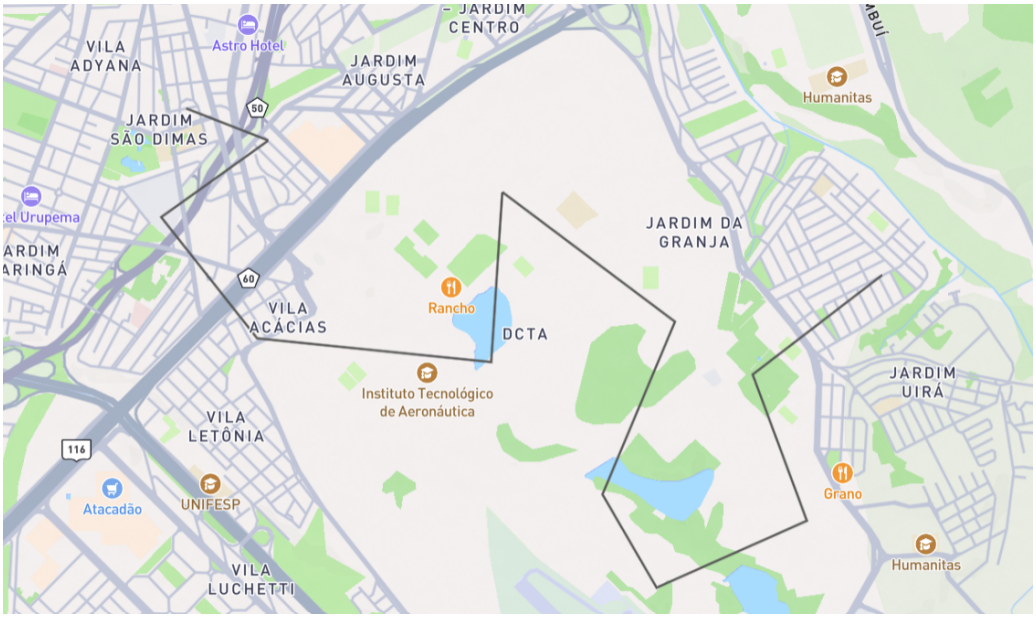


FIGURA 4.1 – Trajetória submetida, evidenciando que parte da rota encontra-se fora do polígono do CTA.

4.1.2 Evidências do sistema

```
[2025-11-02 19:39:29] INFO - Provider_1 - Inscrito no tópico drones/8/status
[2025-11-02 19:39:29] INFO - Provider_1 - Inscrito nos tópicos providers/1/requests/# e pvr/#
[2025-11-02 19:39:29] INFO - Provider_3 - Conectado com sucesso ao broker!
[2025-11-02 19:39:29] INFO - Provider_3 - Inscrito no tópico drones/10/status
[2025-11-02 19:39:29] INFO - Provider_3 - Inscrito no tópico drones/9/status
[2025-11-02 19:39:29] INFO - Provider_3 - Inscrito no tópico drones/11/status
[2025-11-02 19:39:29] INFO - Provider_3 - Inscrito no tópico drones/13/status
[2025-11-02 19:39:29] INFO - Provider_3 - Inscrito nos tópicos providers/3/requests/# e pvr/#
[2025-11-02 19:39:38] INFO - Provider_1 - [2025-11-02 19:39:38] Mensagem recebida no tópico providers/1/requests/4: 2
[2025-11-02 19:39:38] INFO - Provider_1 - Requisição identificada com ID: 2
90 segments were successfully inserted for request_id 2.
[2025-11-02 19:41:19] INFO - Provider_1 - Resposta enviada para 4: Voo negado: fora dos limites do CTA.
```

FIGURA 4.2 – Log do *Provider* mostrando a identificação da requisição e a resposta de negativa.

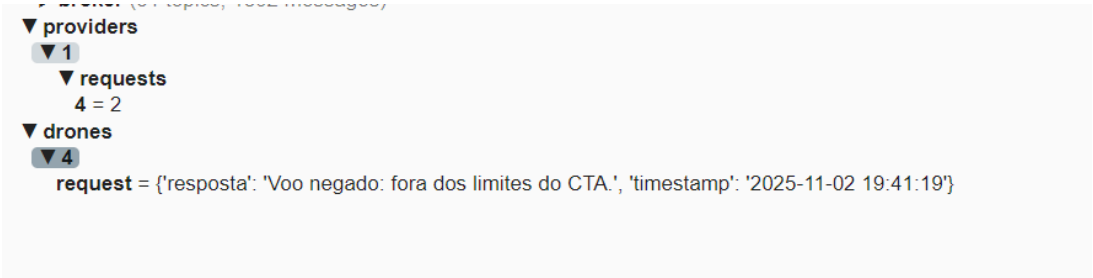


FIGURA 4.3 – Mensagem MQTT publicada pelo *Provider* ao drone, indicando a negativa do voo.

4.1.3 Mensagem JSON recebida pelo drone

```
{
  "resposta": "Voo negado: fora dos limites do CTA.",
```

```
  "timestamp": "2025-11-02 19:41:19"
}
```


4.1.4 Mensagem JSON recebida pelo drone


```
{
  "resposta": "Voo negado: fora dos limites do CTA.",
  "timestamp": "2025-11-02 19:41:19"
}
```


4.1.5 Análise e discussão

A resposta "Voo negado: fora dos limites do CTA" confirma o funcionamento correto da verificação estática. A análise espacial utiliza a função `ST_Within`, que verifica se a geometria da trajetória (`waypoints`) está inteiramente contida na geometria de referência do CTA (`maps.id = 1`). Como parte da rota ultrapassava os limites do polígono do DCTA (Figura 4.1), o sistema identificou a violação e negou automaticamente a operação. Esse teste valida o primeiro nível de checagem do sistema, anterior aos processos de resolução de conflito, garantindo que apenas voos dentro do espaço aéreo autorizado possam prosseguir para análise de altitude ou temporal. O comportamento obtido está totalmente alinhado com o resultado esperado, demonstrando o correto funcionamento da camada de verificação estática.

4.2 Requisição em área restrita

4.2.1 Descrição do cenário

Este cenário tem como objetivo verificar o comportamento do sistema frente a uma requisição cuja trajetória intercepta uma área classificada como restrita no mapa de referência.

As informações operacionais da requisição, incluindo o drone, o provedor e o operador associados, encontram-se resumidas na Tabela 4.2, enquanto a trajetória correspondente pode ser visualizada na Figura 4.4. As informações da área restrita, por sua vez, encontra-se na Tabela 4.3. A simulação busca validar se o mecanismo de verificação estática é capaz de identificar corretamente essa interseção e negar o voo antes do início do processo de resolução de conflito.

TABELA 4.2 – Parâmetros operacionais da requisição analisada no cenário de área restrita

Parâmetro	**Valor**
Drone associado	DJI Phantom 4 (`drone_id = 4`)
Provedor responsável	`provider_id = 1`
Operador vinculado	`operator_id = 68`
Início da operação solicitada	28/11/2025 – 08:00:00
Final da operação solicitada	28/11/2025 – 08:30:00

TABELA 4.3 – Informações da área restrita interceptada pela trajetória

maps_id	**Nome**	**Tipo de zona**	**Categoria**
2	Aeroporto de São José	Restrita	Aeroporto

FIGURA 4.4 – Trajetória, cruzando o polígono correspondente a uma área restrita dentro do CTA.

4.2.2 Evidências do sistema

```
e - Bain/Desktop/TG-UTM/Códigos/Códigos_v1/provider.py"
12 segments were successfully inserted for request_id 7.
[2025-11-02 21:43:51] INFO - Deconflit - [7] Altitude original: 2049.7515003795393-2115.371500379539 | Piso/Teto: 2016.941500379539-2364.649643
157959
[2025-11-02 21:43:51] INFO - Deconflit - [7] Sem conflitos de rota. Mantendo altitude original.
[2025-11-02 21:43:51] INFO - Provider_1 - Resposta enviada para 4: {"status": "aprovado", "message": "Voo autorizado.", "altitude": {"final_min
": 2049.8, "final_max": 2115.4, "requested_min": 2049.8, "requested_max": 2115.4, "changed": false, "note": "mantida"}, "Window": {"start_time"
: "2025-11-28 08:00:00", "end_time": "2025-11-28 08:30:00"}, "request_id": 7, "timestamp": "2025-11-02 21:43:51"}
[2025-11-02 21:54:14] INFO - Provider_1 - [2025-11-02 21:54:14] Mensagem recebida no tópico providers/1/requests/4: 8
[2025-11-02 21:54:14] INFO - Provider_1 - Requisição identificada com ID: 8
19 segments were successfully inserted for request_id 8.
[2025-11-02 21:54:43] INFO - Provider_1 - Resposta enviada para 4: {"status": "negado", "reason": "Trajet\u00f3ria passa por \u00e1rea restrita
.", "timestamp": "2025-11-02 21:54:43"}
```

FIGURA 4.5 – Log do *Provider* mostrando a recepção da requisição 8 e a resposta de negativa por área restrita.

FIGURA 4.6 – Mensagem MQTT publicada pelo *Provider 1* para o drone 4.

4.2.3 Mensagem JSON recebida pelo drone

```
{
  "status": "negado",
  "reason": "Trajetória passa por área restrita.",
  "timestamp": "2025-11-02 21:54:43"
}
```

4.2.4 Análise e discussão

A mensagem "*Trajetória passa por área restrita*" confirma que a verificação espacial foi executada corretamente. O sistema utiliza a função `ST_Intersects` para verificar a interseção entre o `waypoints` da requisição e os polígonos classificados como "restrita" na camada `maps`.

O comportamento observado é consistente com o esperado: o voo foi negado durante a verificação estática, garantindo que nenhuma missão seja aprovada em regiões definidas como restritas.

4.3 Requisição em área com PVR ativo

4.3.1 Descrição do cenário

Este cenário tem como objetivo avaliar o comportamento do sistema diante de uma requisição cuja trajetória intercepta uma *Priority Volume Reservation* (PVR) ativa.

As informações operacionais da requisição, incluindo o drone, o provedor e o operador associados, encontram-se resumidas na Tabela 4.4, enquanto a trajetória correspondente pode ser visualizada na Figura 4.7. Esperava-se que o sistema detectasse a interseção entre a rota e a área reservada do PVR ativo e negasse automaticamente a solicitação de voo ainda na fase de verificação estática.

TABELA 4.4 – Parâmetros operacionais da requisição analisada no cenário com PVR ativo

Parâmetro	**Valor**
Drone associado	DJI Phantom 4 (`drone_id = 4`)
Provedor responsável	`provider_id = 1`
Operador vinculado	`operator_id = 68`
Início da operação solicitada	28/11/2025 – 08:00:00
Final da operação solicitada	28/11/2025 – 08:30:00

TABELA 4.5 – Informações do PVR ativo interceptado pela trajetória

Parâmetro	**Valor**
Drone associado	Parrot Anafi (`drone_id = 7`)
Provedor responsável	`provider_id = 2`
Operador vinculado	`operator_id = 24`
Início da operação	28/11/2025 – 07:50:00
Final da operação	28/11/2025 – 09:30:00
Status do PVR	Aprovado

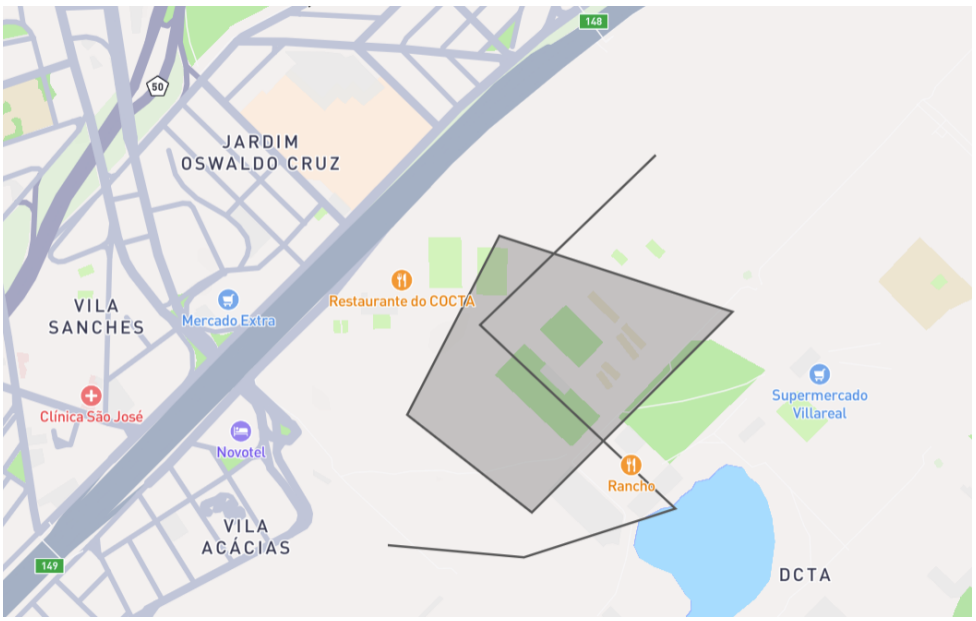


FIGURA 4.7 – Interseção espacial entre a trajetória e a área do PVR.

4.3.2 Evidências do sistema

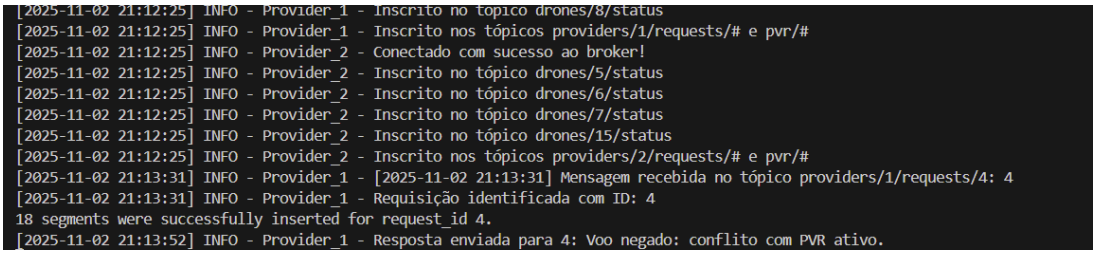


FIGURA 4.8 – Log do *Provider 1* mostrando a identificação da requisição e a negativa por conflito com PVR ativo.

4.3.3 Mensagem JSON recebida pelo drone

```
{
  "resposta": "Voo negado: conflito com PVR ativo.",
  "timestamp": "2025-11-02 21:13:52"
}
```

4.3.4 Análise e discussão

A negativa "*Voo negado: conflito com PVR ativo*" confirma que o sistema realiza corretamente a interseção espacial e temporal entre a trajetória e as áreas de reserva. A função `ST_Intersects` é aplicada entre o *buffer* da trajetória (`ST_Buffer(waypoints::geography, 5)::geometry`) e a geometria do PVR (`p.area`), enquanto a restrição temporal é avaliada pelo operador `OVERLAPS`.

Conforme mostrado na Figura 4.7, a trajetória cruzava a região reservada, e a janela temporal de voo (08:00–08:30) estava inteiramente contida no período ativo do PVR (07:50–09:30). Dessa forma, a requisição foi negada ainda na etapa de verificação estática.

O comportamento obtido está alinhado ao esperado: o sistema impede sobreposição de voos em regiões reservadas, garantindo o cumprimento das restrições temporárias de uso do espaço aéreo. Esse teste valida a integração entre as verificações espaciais e temporais aplicadas a PVRs, demonstrando a coerência da hierarquia de checagens implementada no sistema.

4.4 Resolução de conflito por altitude entre missões

4.4.1 Primeira missão autorizada

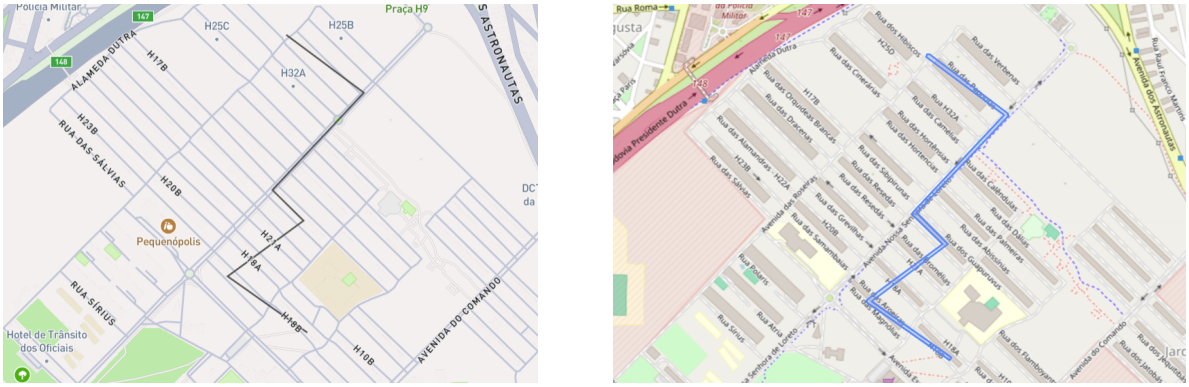
O primeiro teste de resolução de conflito espacial foi realizado com a submissão de uma trajetória completamente contida dentro dos limites do CTA, sem sobreposição com outras missões ativas ou áreas restritas. Essa operação foi utilizada como referência para os experimentos subsequentes, servindo de base para a análise de voos que compartilham a mesma rota.

As informações operacionais da requisição, incluindo o drone, o provedor e o operador associados, estão apresentadas na Tabela 4.6, enquanto a trajetória correspondente pode ser visualizada na Figura 4.9a.

O objetivo inicial deste cenário foi validar o fluxo completo de aprovação de uma missão em condições ideais, ou seja, sem necessidade de ajustes de altitude ou tempo. Esperava-se que o voo fosse aprovado diretamente, confirmando o funcionamento da etapa de verificação estática e o registro da missão no banco de dados.

TABELA 4.6 – Parâmetros operacionais da requisição analisada no cenário de referência paraa resolução de conflito por altitude

Parâmetro	**Valor**
Drone associado	DJI Phantom 4 (`drone_id = 4`)
Provedor responsável	`provider_id = 1`
Operador vinculado	`operator_id = 68`
Início da operação solicitada	28/11/2025 – 08:00:00
Final da operação solicitada	28/11/2025 – 08:30:00



(a) Trajetória submetida.

(b) Missão registrada no banco de dados.

FIGURA 4.9 – Requisição de trajetória e registro de missão no banco de dados

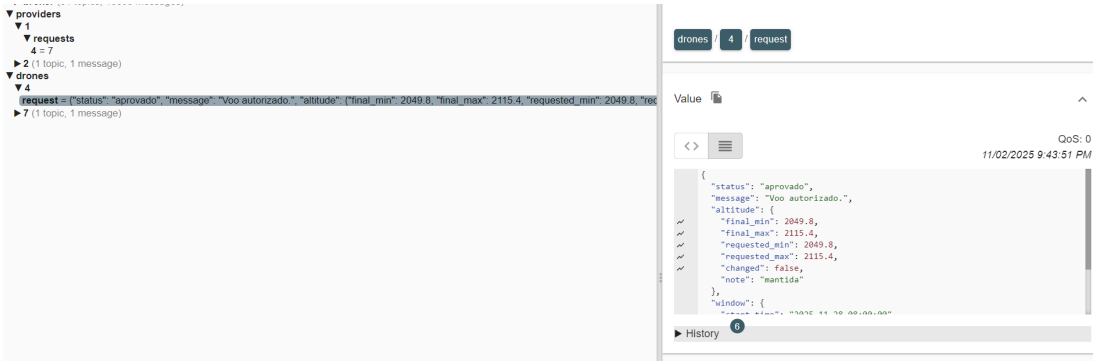


FIGURA 4.10 – Mensagem MQTT publicada pelo *Provider 1* com o status de voo autorizado.

4.4.2 Mensagem JSON recebida pelo drone


```
{
  "status": "aprovado",
  "message": "Voo autorizado.",
  "altitude": {
    "final_min": 2049.8,
    "final_max": 2115.4,
    "requested_min": 2049.8,
    "requested_max": 2115.4,
    "changed": false,
    "note": "mantida"
  },
  "window": {
    "start_time": "2025-11-28 08:00:00",
    "end_time": "2025-11-28 08:30:00"
  },
  "request_id": 7,
  "timestamp": "2025-11-02 21:43:51"
}
```


A missão foi aprovada conforme o esperado, sem necessidade de ajuste de altitude (`altitude.changed = false`). O JSON mostra que não houve conflitos de rota e que a altitude nominal foi mantida entre 2049,8 e 2115,4 pés. Essa operação valida o comportamento básico do sistema e estabelece uma referência para os testes seguintes de resolução de conflito por altitude. A missão foi registrada na tabela que armazena missões aprovadas e seu registro, com *buffer* espacial, pode ser visualizado na Figura 4.9b.

A partir dessa missão aprovada, novas requisições foram submetidas utilizando a mesma trajetória e janela temporal, de modo a avaliar o comportamento do algoritmo

de alocação vertical quando duas operações compartilham o mesmo espaço aéreo.

4.4.3 Missões subsequentes na mesma trajetória

Três novas missões foram submetidas logo após a aprovação da primeira, todas utilizando a mesma trajetória e janela temporal (08:00–08:30). Nesse cenário, esperava-se que o sistema aplicasse o mecanismo de resolução de conflito por altitude, alocando novas faixas verticais acima das missões já aprovadas, de modo a garantir a separação espacial entre elas. O comportamento esperado seria a aprovação das missões subsequentes até o preenchimento total do espaço aéreo disponível, com a negação automática daquelas que ultrapassassem o limite de teto definido para a região.

TABELA 4.7 – Relação das requisições submetidas no teste de resolução de conflito por altitude

Drone ID	Provider ID	Operador ID	Request ID
7	2	35	9
11	3	23	10
13	3	14	11

4.4.4 Evidências do sistema

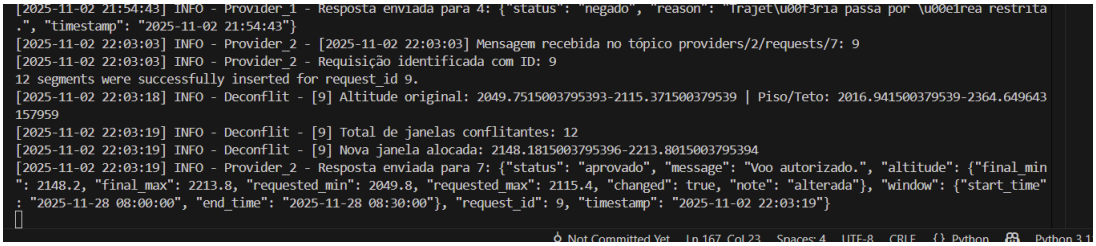


FIGURA 4.11 – Log do *Provider 2* mostrando a aprovação da requisição 9 com ajuste de altitude.

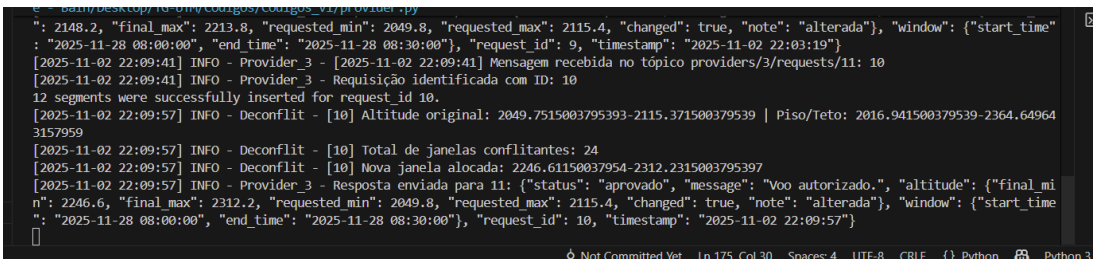


FIGURA 4.12 – Log do *Provider 3* confirmandoa resolução de conflito da segunda missão (requisição 10) em nova faixa vertical.

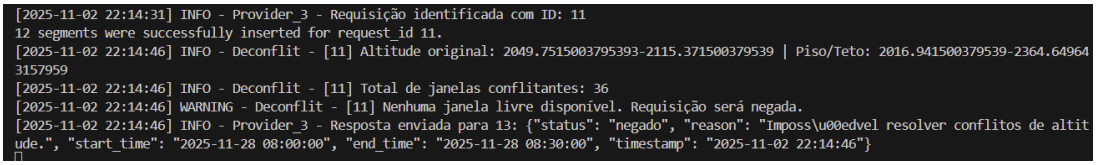


FIGURA 4.13 – Log do *Provider 3* indicando a negação da requisição 11 por ultrapassar o teto operacional.

FIGURA 4.14 – Mensagens MQTT de todas as 4 missões com mesma trajetória.

4.4.5 Mensagens JSON recebidas pelos drones

```
# Requisição 9
{
  "status": "aprovado",
  "message": "Voo autorizado.",
  "altitude": {
    "final_min": 2148.2,
    "final_max": 2213.8,
    "requested_min": 2049.8,
    "requested_max": 2115.4,
    "changed": true,
    "note": "alterada"
  },
  "window": {
    "start_time": "2025-11-28 08:00:00",
    "end_time": "2025-11-28 08:30:00"
  },
  "request_id": 9,
  "timestamp": "2025-11-02 22:03:19"
}

# Requisição 10
{
  "status": "aprovado",
```

```
  "message": "Voo autorizado.",
  "altitude": {
    "final_min": 2246.6,
    "final_max": 2312.2,
    "requested_min": 2049.8,
    "requested_max": 2115.4,
    "changed": true,
    "note": "alterada"
  },
  "window": {
    "start_time": "2025-11-28 08:00:00",
    "end_time": "2025-11-28 08:30:00"
  },
  "request_id": 10,
  "timestamp": "2025-11-02 22:09:57"
}


# Requisição 11
{
  "status": "negado",
  "reason": "Impossível resolver conflitos de altitude.",
  "start_time": "2025-11-28 08:00:00",
  "end_time": "2025-11-28 08:30:00",
  "timestamp": "2025-11-02 22:14:46"
}
```


4.4.6 Análise e discussão

O comportamento observado esteve em conformidade com o esperado. Após a aprovação da primeira missão, o sistema aplicou o mecanismo de resolução de conflito por altitude, realocando as missões subsequentes em faixas verticais progressivamente superiores. Esse processo garantiu a separação mínima entre operações e evitou sobreposições espaciais e temporais.

A Tabela 4.8 apresenta as missões executadas neste teste, evidenciando o aumento gradual das altitudes e a correta aplicação do algoritmo de redistribuição vertical. As duas primeiras missões adicionais foram aprovadas com ajuste de altitude, enquanto a terceira foi negada por inexistência de faixas verticais disponíveis, após o atingimento do

limite superior do espaço aéreo definido para o cenário.

TABELA 4.8 – Resumo das missões executadas no teste de resolução de conflito por altitude entre missões

Missão ID	Drone ID	Request ID	Altitude mínima (ft)	Altitude máxima (ft)
2	4	7	2049.8	2115.4
3	7	9	2148.2	2213.8
4	11	10	2246.6	2312.2

Esses resultados confirmam quea resolução de conflito espacial por altitude atua de maneira coerente com os parâmetros estabelecidos, garantindo a separação vertical entre missões até o esgotamento do espaço aéreo disponível. O comportamento observado demonstra a efetividade do algoritmo de alocação vertical, que prioriza a maximização do uso do espaço aéreo sem comprometer os critérios de segurança operacional.

4.5 Detecção de conflitos a nível de segmento - requisição aprovada

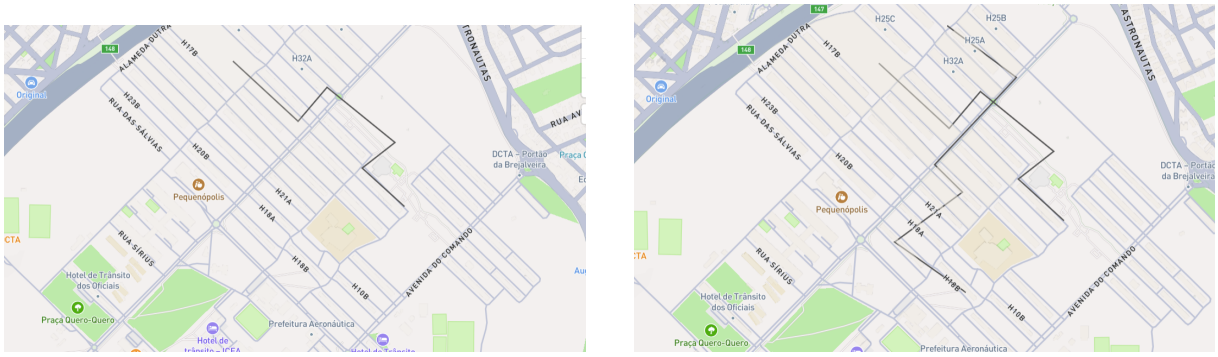
4.5.1 Descrição do cenário

Este cenário tem como objetivo validar o comportamento do sistema na detecção de conflitos em nível de segmento, verificando se o mecanismo de segmentação é capaz de distinguir situações em que as trajetórias se cruzam espacialmente, mas não apresentam sobreposição temporal. A simulação envolve uma requisição cuja rota intercepta parcialmente a trajetória da missão com `mission_id = 2` na janela de voo de menor altitude, aprovada conforme discutido na seção 4.4, permitindo avaliar se o sistema reconhece corretamente a ausência de conflito espaço–temporal.

As informações operacionais da requisição estão apresentadas na Tabela 4.9. A trajetória enviada e a intersecção de trajetórias podem ser visualizadas nas Figuras 4.15a e 4.15b, respectivamente.

TABELA 4.9 – Parâmetros operacionais da requisição analisada no cenário de detecção de conflitos a nível de segmento

Parâmetro	**Valor**
Drone associado	DJI Mavic Air 2 (`drone_id = 13`)
Provedor ável	`provider_id = 3`
Operador vinculado	`operator_id = 8`
Início da operação solicitada	28/11/2025 – 08:00:00
Final da operação solicitada	28/11/2025 – 08:30:00



(a) Trajetória da requisição submetida para análise de conflito a nível segmento.

(b) Trajetórias da requisição e da missão previamente aprovada.

FIGURA 4.15 – Trajetória submetida e cruzamento espacial entre trajetórias.

4.5.2 Evidências do sistema

```
[2025-11-02 22:14:46] INFO - Provider_3 - Resposta enviada para 13: {"status": "negado", "reason": "Imposs\u00edvel resolver conflitos de altit
ude.", "start_time": "2025-11-28 08:00:00", "end_time": "2025-11-28 08:30:00", "timestamp": "2025-11-02 22:14:46"}
[2025-11-02 22:28:23] INFO - Provider_3 - [2025-11-02 22:28:23] Mensagem recebida no tópico providers/3/requests/13: 12
[2025-11-02 22:28:23] INFO - Provider_3 - Requisição identificada com ID: 12
10 segments were successfully inserted for request_id 12.
[2025-11-02 22:28:38] INFO - Deconflit - [12] Altitude original: 2049.7515003795393-2115.371500379539 | Piso/Teto: 2016.941500379539-2369.02854
85839845
[2025-11-02 22:28:38] INFO - Deconflit - [12] Sem conflitos de segmento. Mantendo altitude original.
[2025-11-02 22:28:38] INFO - Provider_3 - Resposta enviada para 13: {"status": "aprovado", "message": "Voo autorizado.", "altitude": {"final_mi
n": 2049.8, "final_max": 2115.4, "requested_min": 2049.8, "requested_max": 2115.4, "changed": false, "note": "mantida"}, "window": {"start_time
": "2025-11-28 08:00:00", "end_time": "2025-11-28 08:30:00"}, "request_id": 12, "timestamp": "2025-11-02 22:28:38"}
```

FIGURA 4.16 – Log do *Provider 3* mostrando a aprovação da requisição a partir de checagem de conflito a nível segmento.

FIGURA 4.17 – Mensagem MQTT enviada ao drone: voo aprovado, altitude mantida (sem necessidade de ajuste), janelas temporais inalteradas.

4.5.3 Consulta no banco de dados

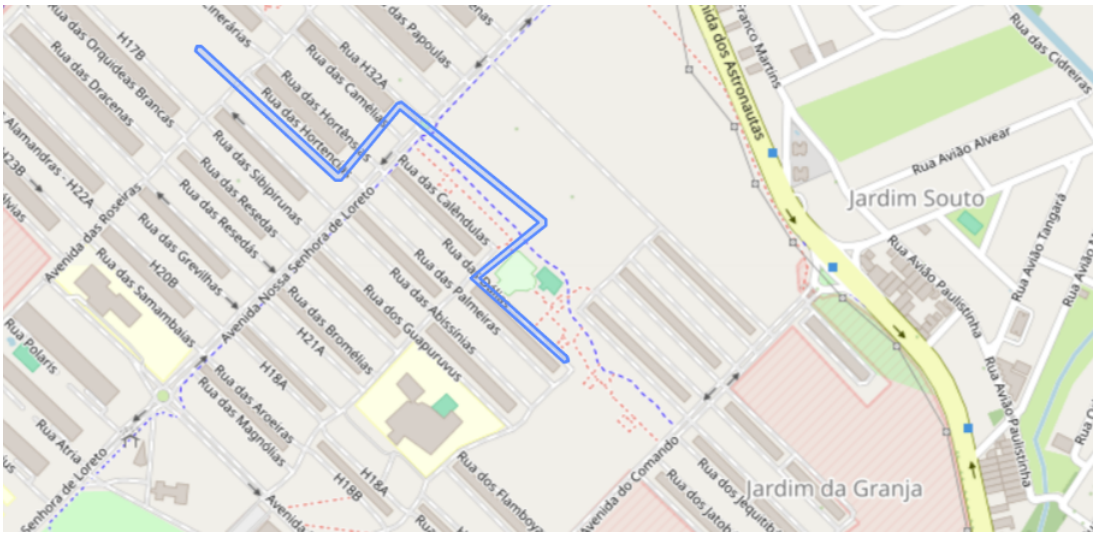


FIGURA 4.18 – Consulta SQL mostrando a missão aprovada.

4.5.4 Mensagem JSON recebida pelo drone


```
{
  "status": "aprovado",
  "message": "Voo autorizado.",
  "altitude": {
    "final_min": 2049.8,
    "final_max": 2115.4,
    "requested_min": 2049.8,
    "requested_max": 2115.4,
    "changed": false,
    "note": "mantida"
  },
  "window": {
    "start_time": "2025-11-28 08:00:00",
    "end_time": "2025-11-28 08:30:00"
  },
  "request_id": 12,
  "timestamp": "2025-11-02 22:28:38"
}
```

4.5.5 Análise e discussão

Apesar da interseção espacial entre as rotas, a decisão foi de aprovação, pois os segmentos envolvidos não apresentaram *overlap* temporal. O algoritmo atua de forma granular: (i) decompõe a rota, (ii) detecta interseções espaciais entre *buffers* de segmentos, e (iii) confirma o conflito apenas se as janelas temporais dos segmentos também se sobrepõem. Esses passos evitam negações desnecessárias, preservando a altitude nominal e as janelas originais quando a separação temporal já é suficiente.

4.6 Detecção de conflitos a nível de segmento - requisição negada

4.6.1 Descrição do cenário

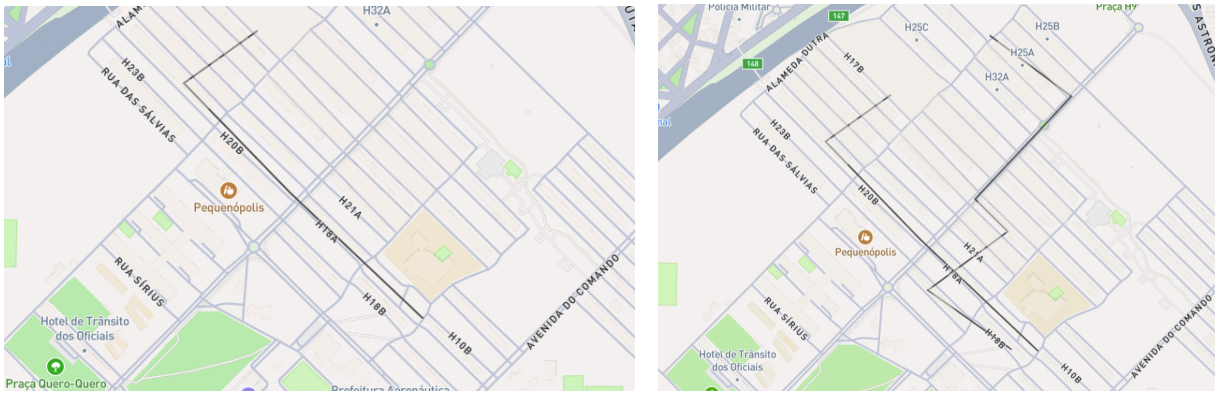
Este cenário tem como objetivo avaliar o comportamento do sistema diante de uma situação de conflito simultâneo no espaço e no tempo, em que o mecanismo de segmentação por si só não é suficiente para evitar a colisão operacional. A simulação consiste em uma nova requisição cuja trajetória cruza parcialmente a rota da missão previamente aprovada de `mission_id = 2`, apresentada na Seção 4.4, resultando em interseção espacial e temporal entre os segmentos correspondentes.

As informações operacionais da requisição estão apresentadas na Tabela 4.10, enquanto as trajetórias podem ser visualizadas na Figura 4.19. Como ilustrado na Figura 4.20, o segmento à esquerda representa o trecho da missão ativa, enquanto o da direita corresponde à nova requisição submetida. Nesse caso, o sistema deveria identificar a coincidência espaço–temporal e negar automaticamente a operação, uma vez que não havia possibilidade de resolução de conflito vertical disponível.

Durante o processo de verificação, o algoritmo tentou aplicara resolução de conflito por altitude, elevando a nova trajetória para faixas superiores do espaço aéreo. No entanto, todas as camadas verticais encontravam-se ocupadas por missões previamente aprovadas — especificamente as de de `mission_id = 3` e `mission_id = 4` —, ambas associadas à mesma trajetória da missão de `mission_id = 2`. Assim, o sistema não pôde realocar a operação sem infringir as separações mínimas estabelecidas, resultando na negação automática da requisição.

TABELA 4.10 – Parâmetros operacionais da requisição analisada no cenário de conflito simultâneo espaço–temporal

Parâmetro	**Valor**
Drone associado	Parrot Anafi (`drone_id = 1`)
Provedor responsável	`provider_id = 1`
Operador vinculado	`operator_id = 15`
Início da operação solicitada	28/11/2025 – 08:00:00
Final da operação solicitada	28/11/2025 – 08:30:00



(a) Trajetória da requisição submetida para análise de conflito a nível de segmento.

(b) Região de interseção espacial entre as rotas, evidenciando a coincidência espaço–temporal.

FIGURA 4.19 – Trajetória da requisição e região de interseção espacial no cenário de conflito a nível de segmento.

(a) Segmento 252 da missão previamente aprovada (`request_id = 7`).

(b) Segmento 333 da nova requisição (`request_id = 14`) utilizada para teste de resolução de conflito espacial a nível segmento.

FIGURA 4.20 – Interseção espacial e temporal entre a missão ativa e a nova requisição, observada no cenário de reprovação a nível de segmento.

4.6.2 Validação espaço–temporal do conflito

Para confirmar a interseção entre os segmentos das missões envolvidas, foi executada uma consulta SQL, combinando as verificações espaciais e temporais. A Tabela 4.11 apresenta o resultado para o segmento `252` da requisição de `request_id = 7` e o segmento `333` da requisição de `request_id = 14`.

TABELA 4.11 – Resultado da análise espaço–temporal entre dois segmentos de missões distintas.

Seg.`request id = 7`	seg.`request id = 14`	Início da sobreposição	Fim da sobreposição	Duração (s)	Interseção Espacial
252	333	08:22:03	08:22:50	47,4	Sim

Apesar de a verificação ter sido realizada com a missão de `mission_id = 2` (requisição `request_id = 7`), o mesmo intervalo de sobreposição seria obtido caso a análise fosse repetida para as missões com `mission_id = 3` e `mission_id= 4`, correspondentes às requisições de `request_id = 9` e `request_id = 10`, respectivamente. Isso ocorre porque todas compartilham a mesma trajetória espacial, variando apenas as faixas de altitude alocadas. Dessa forma, o resultado confirma a consistência da metodologia de verificação espaço–temporal, independentemente da camada vertical em que a missão se encontra.

O resultado confirma a existência de uma sobreposição espaço–temporal de aproximadamente 47 segundos entre os segmentos analisados. Isso comprova que o sistema identificou corretamente o conflito a nível de segmento, reconhecendo que ambos os trechos ocuparam o mesmo espaço aéreo no mesmo intervalo de tempo.

4.6.3 Evidências do sistema

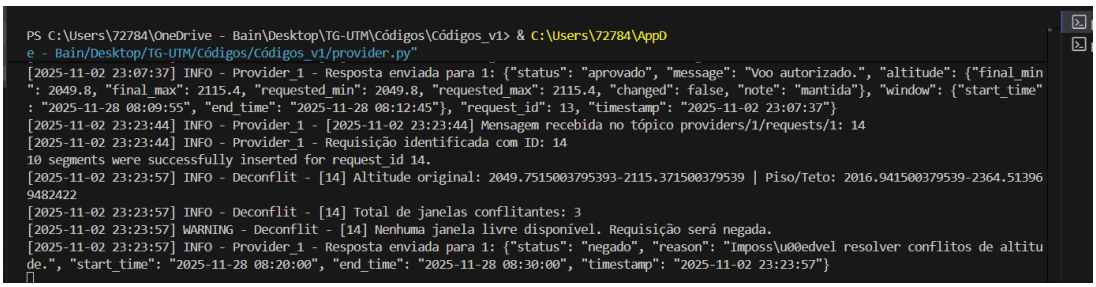


FIGURA 4.21 – Log do *Provider 1* mostrando a detecção de janelas conflitantes e a negação da requisição por impossibilidade de resolver conflitos de altitude.

FIGURA 4.22 – Mensagem MQTT publicada pelo provedor indicando a negativa da requisição por impossibilidade de resolução de conflito.

4.6.4 Mensagem JSON recebida pelo drone


```
{
  "status": "negado",
  "reason": "Impossível resolver conflitos de altitude.",
  "start_time": "2025-11-28 08:20:00",
  "end_time": "2025-11-28 08:30:00",
  "timestamp": "2025-11-02 23:23:57"
}
```


4.6.5 Análise e discussão

A segmentação da trajetória permitiu identificar que os trechos sobrepostos ocupavam o mesmo espaço aéreo exatamente no mesmo intervalo de tempo, caracterizando um conflito efetivo. Nesse caso, a divisão em segmentos não é suficiente para permitir a aprovação, já que a coincidência temporal inviabiliza o compartilhamento do espaço.

O sistema tentou resolver o conflito por altitude, aplicando o algoritmo de redistribuição vertical. No entanto, as camadas superiores já estavam ocupadas por missões previamente aprovadas na mesma rota, e o teto máximo do espaço aéreo (2364 pés) havia sido atingido. Assim, não havia mais faixa de altitude disponível para alocação, resultando na negativa automática da requisição.

Até esse ponto dos testes, o mecanismo de resolução de conflito temporal ainda não havia sido implementado. Portanto, o sistema agiu corretamente ao negar a operação, uma vez que não havia solução espacial (por altitude) possível. O comportamento observado confirma que o algoritmo de resolução de conflito espacial atua de forma consistente e segura, restringindo novas missões quando todos os recursos verticais disponíveis já foram utilizados.

4.7 Resolução de conflito temporal - requisição aprovada

4.7.1 Descrição do cenário

O presente cenário utiliza a mesma trajetória analisada na Seção 4.6, porém com o módulo de resolução de conflito temporal habilitado. O objetivo é avaliar a capacidade do sistema de realocar a operação no tempo quando a redistribuição vertical não é possível, mantendo a coerência com os parâmetros de separação configurados. Os dados operacionais da requisição estão presentes na Tabela 4.12

TABELA 4.12 – Parâmetros operacionais da requisição analisada no cenário de resolução de conflito temporal.

Parâmetro	**Valor**
Drone associado	Parrot Anafi (`drone_id = 1`)
Provedor responsável	`provider_id = 1`
Operador vinculado	`operator_id = 15`
Início da operação solicitada	28/11/2025 – 08:20:00
Final da operação solicitada	28/11/2025 – 08:30:00

A missão foi originalmente solicitada para o intervalo de 08:20–08:30, com duração total de dez minutos. O parâmetro de deslocamento temporal máximo (*max_shift_ratio*) foi definido em 0,5, permitindo que a janela fosse ajustada em até 50 % da sua duração, ou seja, até cinco minutos ($\Delta t_{\max} = 0{,}5 \times 10$ min $= 5$ min). Assim, a maior janela admissível de deslocamento possível corresponderia a 08:25–08:35.

4.7.2 Evidências do sistema

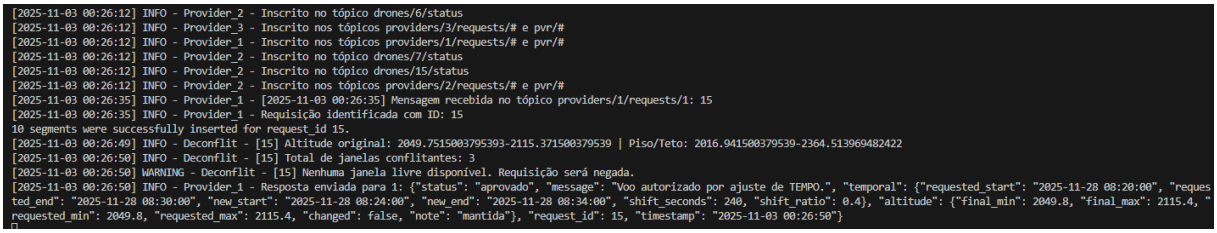


FIGURA 4.23 – Log do *Provider 1*: identificação das janelas conflitantes, aplicação do acolchoamento e realocação temporal da missão.

FIGURA 4.24 – Mensagem MQTT indicando a aprovação do voo após ajuste temporal.

4.7.3 Mensagem JSON recebida pelo drone


```
{
  "status": "aprovado",
  "message": "Voo autorizado por ajuste de TEMPO.",
  "temporal": {
    "requested_start": "2025-11-28 08:20:00",
    "requested_end":   "2025-11-28 08:30:00",
    "new_start":       "2025-11-28 08:24:00",
    "new_end":         "2025-11-28 08:34:00",
    "shift_seconds":   240,
    "shift_ratio":     0.4
  },
  "altitude": {
    "final_min": 2049.8,
    "final_max": 2115.4,
    "changed":  false,
    "note":     "mantida"
  }
}
```


4.7.4 Análise e discussão

A análise dos segmentos em conflito indicou que a sobreposição espaço–temporal entre as missões estendia-se até 08:22:50. O algoritmo aplica um acolchoamento temporal (*pad_seconds*) de 60 s antes e depois de cada janela ocupada, de modo que o intervalo efetivamente bloqueado se estendeu até 08:23:50.

A busca por janelas temporais livres é realizada de forma incremental, com passo

$\delta t = 15$ s, conforme o algoritmo descrito em 3.4.5 . Dessa forma, os instantes avaliados são

$$t = t_0 + n\delta t, \quad n \in \mathbb{N},$$

com $t_0 = 08{:}20{:}00$. Como o primeiro instante imediatamente posterior à borda acolchoada (08:23:50) que coincide com a grade de passos ocorre em $t = 08{:}24{:}00$, este foi o horário selecionado como início da nova janela:

$$t_{\text{novo\_início}} = 08{:}24{:}00.$$

O deslocamento aplicado foi de $\Delta t = 240$ s $= 4$ min, valor que respeita o limite máximo permitido pelo parâmetro configurado ($\Delta t_{\max} = 300$ s), satisfazendo

$$\Delta t < \Delta t_{\max}.$$

A nova janela de operação foi, portanto, ajustada para 08:24–08:34, eliminando completamente a sobreposição temporal com as missões ativas e mantendo a altitude nominal inalterada. Um resumo das operações pode ser visualizado na Tabela 4.13 . O resultado evidencia o funcionamento adequado do módulo de resolução de conflito temporal, que atua de acordo com as restrições paramétricas definidas, garantindo separação temporal mínima entre operações e preservando a consistência do gerenciamento do espaço aéreo.

TABELA 4.13 – Resumo do ajuste temporal aplicado no resolução de conflito

Parâmetro	**Valor**
Início da janela original	08:20:00
Fim da janela original	08:30:00
Duração da missão	10 min
Início da sobreposição detectada	08:22:03
Término da sobreposição	08:22:50
Acolchoamento aplicado (*pad_seconds*)	± 60 s
Término do intervalo bloqueado	08:23:50
Passo de busca temporal (δt)	15 s
Primeiro instante livre encontrado	08:24:00
Deslocamento aplicado (Δt)	240 s (4 min)
Limite máximo permitido ($\Delta t_{\max}$)	300 s (5 min)
Nova janela de operação	08:24–08:34
Condição	$\Delta t < \Delta t_{\max}$
Situação final	Conflito eliminado, altitude mantida

4.8 Resolução de conflito temporal - requisição negada

4.8.1 Descrição do cenário

Este cenário utiliza a mesma requisição da Seção 4.7, com o módulo de resolução de conflito temporal habilitado, porém com um valor reduzido do parâmetro ρ, definido em 0,05. Esse valor corresponde a 5% da duração total da missão, o que, para uma operação de dez minutos, representa um deslocamento máximo permitido de apenas 30 segundos ($\Delta t_{\max} = 0{,}05 \times 10 \text{ min} = 30 \text{ s}$). O objetivo deste teste foi verificar o comportamento do sistema quando o limite de deslocamento temporal é insuficiente para eliminar a sobreposição com outras missões ativas.

4.8.2 Evidências do sistema

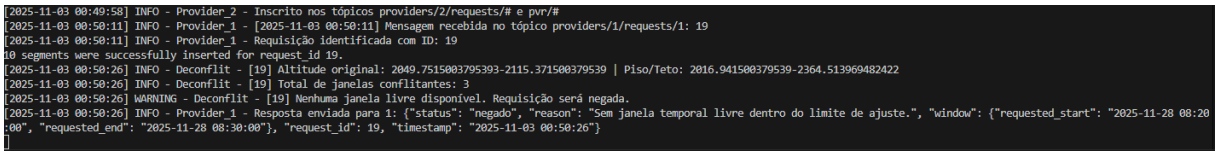


FIGURA 4.25 – Log do *Provider 1*: cálculo das janelas conflitantes e detecção da ausência de intervalos livres dentro do limite configurado.

FIGURA 4.26 – Mensagem MQTT indicando a negativa da operação por ausência de janela temporal livre dentro do limite de ajuste.

4.8.3 Mensagem JSON recebida pelo drone

```
{
  "status": "negado",
  "reason": "Sem janela temporal livre dentro do limite de ajuste.",
  "window": {
    "requested_start": "2025-11-28 08:20:00",
    "requested_end":   "2025-11-28 08:30:00"
  },
  "request_id": 19,
  "timestamp": "2025-11-03 00:50:26"
}
```

4.8.4 Análise e discussão

Durante a execução do algoritmo de busca temporal, o sistema avaliou a disponibilidade de janelas livres deslocando incrementalmente a operação no tempo, com passo de 15 segundos ($\delta t = 15$ s), respeitando o limite máximo de deslocamento configurado de 30 segundos. Como a sobreposição entre missões se estendia até 08:22:50, e o acolchoamento de 60 segundos ampliava a zona de bloqueio até 08:23:50, não havia nenhum intervalo viável dentro do limite imposto.

Matematicamente, o deslocamento máximo permitido era dado por:

$$\Delta t_{\max} = \rho \times T_{\text{missão}} = 0{,}05 \times 600 \text{ s} = 30 \text{ s},$$

enquanto a separação temporal necessária para eliminar o conflito era de:

$$\Delta t_{\text{necessário}} = 08{:}23{:}50 - 08{:}20{:}00 = 230 \text{ s}.$$

Como $\Delta t_{\text{necessário}} > \Delta t_{\max}$, o sistema corretamente concluiu pela impossibilidade de realocação temporal, resultando na negativa automática da operação. Um resumo pode ser visualizado na Tabela 4.14

TABELA 4.14 – Resumo do ajuste temporal com limite reduzido de deslocamento

Parâmetro	**Valor**
Início da janela original	08:20:00
Fim da janela original	08:30:00
Duração da missão	10 min
Início da sobreposição detectada	08:22:03
Término da sobreposição	08:22:50
Acolchoamento aplicado (*pad_seconds*)	$\pm$ 60 s
Término do intervalo bloqueado	08:23:50
Passo de busca temporal (δt)	15 s
Deslocamento necessário ($\Delta t_{\text{necessário}}$)	230 s (3 min 50 s)
Limite máximo permitido ($\Delta t_{\max}$)	30 s ($\rho = 0{,}05$)
Nova janela possível	–
Condição	$\Delta t_{\text{necessário}} > \Delta t_{\max}$
Situação final	Sem janela livre; operação negada

Esse comportamento confirma que o módulo de resolução de conflito temporal opera de forma aderente aos parâmetros configurados, não ultrapassando os limites definidos de ajuste. Dessa forma, o sistema preserva a previsibilidade e a consistência das decisões, garantindo que alterações temporais ocorram apenas dentro dos critérios de segurança e tolerância estabelecidos.

4.9 Rechecagem após resolução de conflito temporal

4.9.1 Descrição do cenário

Este cenário utiliza a mesma requisição da Seção 4.7, agora coma resolução de conflito temporal habilitado e a rechecagem automática de conflito com PVR ou missão após o ajuste. A requisição original foi submetida para o intervalo `08:20–08:30`. O sistema identificou sobreposição espaço–temporal com outra missão e aplicoua resolução de conflito temporal com as seguintes configurações: $\rho = 0{,}5$ (até metade da duração), `pad_seconds` = 60 e `step_seconds` = 15.

O objetivo deste cenário é verificar se, após a realocação temporal, o sistema é capaz de identificar corretamente a existência de uma nova interseção com uma missão ou um PVR ativo. Para isso, foi testada a detecção de conflito com o PVR apresentado na Tabela 4.15, cuja área de cobertura intercepta a trajetória ajustada apósa resolução de conflito, conforme a Figura 4.27.

TABELA 4.15 – Informações do PVR ativo interceptado pela trajetória

Parâmetro	**Valor**
Drone associado	Parrot Anafi (`drone_id = 6`)
Provedor responsável	`provider_id = 2`
Operador vinculado	`operator_id = 14`
Início da operação	28/11/2025 – 08:32:00
Final da operação	28/11/2025 – 10:32:00
Status do PVR	Aprovado

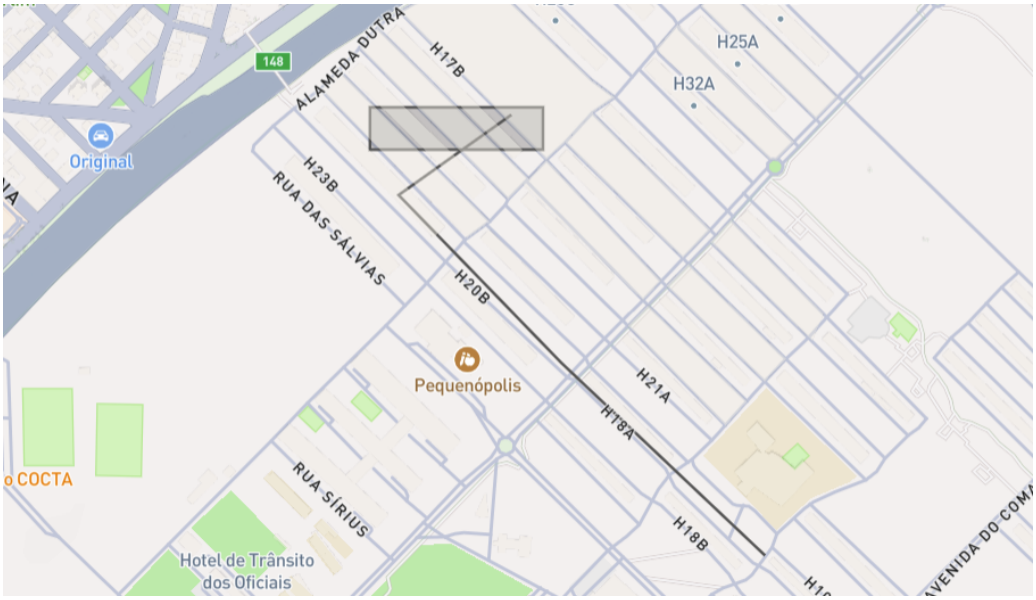


FIGURA 4.27 – Sobreposição da janela ajustada `08:24–08:34` com o PVR ativo `08:32–10:32`.

4.9.2 Ajuste temporal realizado

A Tabela 4.16 resume a sequência de eventos temporais observada durante o ajuste. Nela, é possível comparar a janela originalmente solicitada, o instante de término da sobreposição identificada e o novo intervalo ajustado pelo sistema. Nota-se que a nova janela de operação (`08:24:00–08:34:00`) foi deslocada exatamente após o término do conflito acolchoado, permanecendo dentro do limite máximo de deslocamento permitido e coincidindo parcialmente com a janela do PVR ativo.

Item	Início	Fim
Janela solicitada	08:20:00	08:30:00
Término da sobreposição	–	08:22:50
Término com acolchoamento	–	08:23:50
Janela ajustada pelo sistema	08:24:00	08:34:00
PVR ativo	08:32:00	10:32:00

TABELA 4.16 – Linha do tempo do ajuste temporal e janela do PVR ativo.

4.9.3 Evidências do sistema

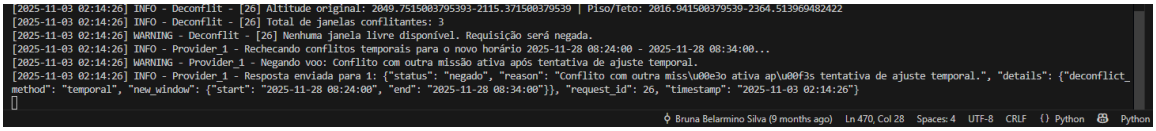


FIGURA 4.28 – Log do provedor: aplicação do resolução de conflito temporal (`08:24–08:34`) e, na rechecagem, detecção de conflito com PVR ativo.

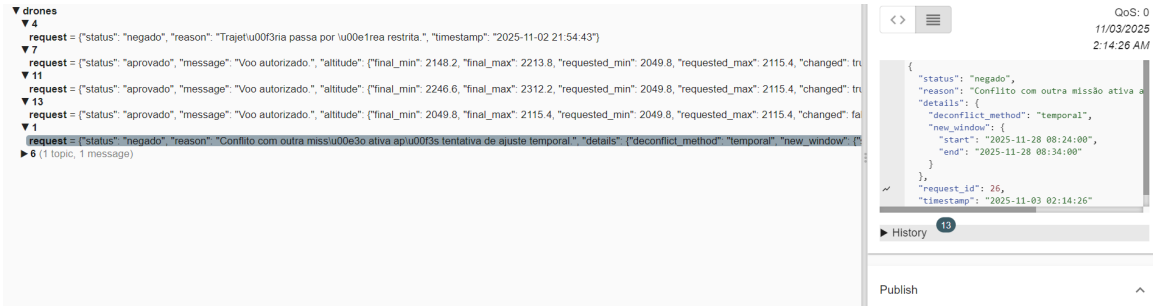


FIGURA 4.29 – Mensagem MQTT retornada: negação após tentativa de ajuste temporal, com detalhamento da nova janela e do motivo.

4.9.4 Mensagem JSON recebida pelo drone


```
{
  "status": "negado",
 "reason": "Conflito com outra missão ativa após tentativa de ajuste temporal.",
```

```
  "details": {
    "deconflict_method": "temporal",
   "new_window": { "start": "2025-11-28 08:24:00", "end": "2025-11-28 08:34:00" }
  },
  "request_id": 26,
  "timestamp": "2025-11-03 02:14:26"
}
```


4.9.5 Análise e discussão

O sistema primeiro desloca a janela solicitada para `08:24–08:34`, operando com acolchoamento de 60 s e passo de 15 s, e respeitando $\rho = 0{,}5$. Em seguida, executa a rechecagem de conflitos para a janela ajustada, abrangendo tanto missões aprovadas quanto PVRs ativos. Como a janela temporal resultante passa a intersectar o PVR vigente entre `08:32–10:32`, a requisição é corretamente negada. O resultado evidencia o comportamento iterativo da solução: o ajuste temporal não é suficiente por si só; qualquer janela proposta é novamente validada contra todas as restrições operacionais vigentes antes da decisão final.

5 Conclusão

Os resultados obtidos neste trabalho demonstram a eficácia e a consistência do modelo de gerenciamento de tráfego não tripulado proposto. A arquitetura desenvolvida comprovou ser tecnicamente capaz de conduzir, de forma autônoma, o fluxo completo de uma operação UTM — desde a comunicação entre os diferentes agentes até a resolução dos conflitos operacionais — mantendo coerência entre as dimensões espacial e temporal do voo e garantindo segurança operacional.

O sistema implementado reproduz a interação entre os principais atores do ecossistema UTM: drones, provedores de serviço e o SDSP, governados pela autoridade aérea. Esses componentes comunicam-se continuamente por meio do protocolo MQTT, trocando mensagens, autorizações, comandos e atualizações de dados suplementares, o que possibilita a coordenação distribuída das operações em um ambiente federado e escalável.

Os experimentos realizados confirmaram o funcionamento correto da hierarquia funcional do sistema. A verificação estática atuou como filtro inicial, rejeitando automaticamente solicitações que violavam restrições geográficas, como limites do DCTA, áreas restritas e PVRs. Na sequência, o módulo de resolução de conflito espacial distribuiu adequadamente as operações em faixas verticais distintas, respeitando margens de separação e recusando novas missões quando o teto operacional era atingido. Por fim,a resolução de conflito temporal mostrou-se eficiente ao reagendar operações em tempo real quando o espaço vertical se encontrava saturado, evitando sobreposições e preservando a previsibilidade do sistema.

Além da capacidade técnica, o modelo proposto favorece a colaboração e a comunicação entre as entidades participantes do ecossistema UTM — operadores, provedores de serviço e autoridades. A arquitetura foi concebida para operar em um ambiente distribuído, em que as decisões são tomadas com base em informações compartilhadas e parâmetros comuns, reforçando os princípios de cooperação, interoperabilidade e equidade de acesso ao espaço aéreo, conforme as diretrizes da FAA e da ICAO.

Como continuidade natural, recomenda-se avançar para uma fase de validação operacional em ambiente real, com métricas quantitativas de desempenho, tempo de resposta e taxa de sucesso nas resoluções de conflito. Também se sugere o aprimoramento do mó-

dulo de resolução de conflito temporal, incorporando métodos que redistribuam trajetórias e tempos de segmento sem alterar a janela global da operação, considerando tempos de percurso, velocidades e metas contratuais (SLA). Estudos de escalabilidade, interoperabilidade e integração com provedores externos devem complementar essa evolução, preparando o sistema para cenários urbanos de maior densidade e complexidade.

Em síntese, o modelo desenvolvido estabelece uma base sólida e colaborativa para a implementação de um sistema UTM nacional. Ao combinar automação, transparência e cooperação entre agentes, a solução representa um passo importante rumo à integração segura e eficiente de drones no espaço aéreo brasileiro, em conformidade com as melhores práticas internacionais.

Referências

CONTROLE DO ESPAÇO AÉREO (DECEA), D. de. **DCA 351-6: Concepção Operacional para o Gerenciamento de Tráfego de Aeronaves Não Tripuladas (UTM)**. Brasília: [*s. n.*], jul. 2022. https://publicacoes.decea.mil.br/publicacao/dca-351-6. Portaria publicada pelo Comando da Aeronáutica. Citado nas pp. 21, 22, 25, 29.

CONTROLE DO ESPAÇO AÉREO (DECEA), D. de. **Informativo Trimestral SARPAS NG - 4º Trimestre de 2024 (Outubro a Dezembro)**. Brasília: [*s. n.*], jan. 2025. https://www.decea.mil.br/static/uploads/2025/01/Informativo_trimestral_SARPAS_04_2024-_out_dez_2024-v.02_jan.pdf. Versão 2.0, publicada em janeiro de 2025. Citado na p. 20.

FAA. **Unmanned Aircraft System (UAS) Traffic Management (UTM) Concept of Operations v2.0**. Washington, D.C.: [*s. n.*], ago. 2022. https://www.faa.gov/sites/faa.gov/files/2022-08/UTM_ConOps_v2.pdf. Versão 2.0, publicada em agosto de 2022. Citado nas pp. 21, 22, 25, 27–29.

FAA; NASA. **UTM Pilot Program (UPP) Phase 2 – Final Report**. Washington, D.C.: [*s. n.*], jul. 2021. https://www.faa.gov/sites/faa.gov/files/2021-07/FY20_UPP2_Final_Report.pdf. FAA and NASA, July 29, 2021. Final Report on the demonstration of UTM services and architecture in high-density UAS environments. Citado nas pp. 25–28.

Apêndice A - Arquivos de códigos desenvolvidos

A.1 Provider

```
"""
provider.py
-----------

Provedor UTM responsável por:
- Assinar tópicos MQTT relevantes;
- Processar requisições de voo (missões e PVRs);
- Executar checagens estáticas, de altitude e (fallback) de conflito temporal;
- Responder drones com o resultado (aprovado/negado/erro).

Dependências externas:
- paho-mqtt (cliente MQTT 5)
- Banco PostgreSQL/PostGIS (via get_db_connection)
- Módulos internos: register_segment, db_connection, check_static, deconflit,
    check_alt, logger
"""

from __future__ import annotations

import json
import time
from datetime import datetime, timedelta
from typing import Any, Iterable, List, Optional, Tuple

import paho.mqtt.client as mqtt

from check_alt import operation_alt
```

```
from check_static import check_restrictions
from db_connection import get_db_connection
from deconflit import resolve_altitude_conflicts
from logger import get_logger
from register_segment import process_request

# ==============================
# Constantes de configuração
# ==============================
DEFAULT_BROKER = "localhost"
DEFAULT_PORT = 1883
DEFAULT_KEEPALIVE = 300

# Parâmetros padrão do deconflito temporal
DEFAULT_MAX_SHIFT_RATIO = 0.5      # fração da duração total (p.ex., 0.5 = até 50% de ajuste)
DEFAULT_SPATIAL_BUFFER_M = 10      # raio (m) para considerar interseção espacial
DEFAULT_PAD_SECONDS = 60           # folga (s) antes/depois ao unir intervalos ocupados
DEFAULT_STEP_SECONDS = 15          # passo (s) ao buscar janelas alternativas
DEFAULT_DIRECTION = "both"         # "forward", "backward" ou "both"


def _safe_round(x: Optional[float], nd: int = 1) -> Optional[float]:
    """Arredonda valor numérico ou retorna None quando não aplicável."""
    return None if x is None else round(float(x), nd)


# ============================================================================
# Deconflito temporal
# ============================================================================
def temporal_deconflict(
    request_id: int,
    cursor,
    desired_start: datetime,
    desired_end: datetime,
    max_shift_ratio: float = DEFAULT_MAX_SHIFT_RATIO,
    spatial_buffer_m: int = DEFAULT_SPATIAL_BUFFER_M,
    pad_seconds: int = DEFAULT_PAD_SECONDS,
    direction: str = DEFAULT_DIRECTION,
    step_seconds: int = DEFAULT_STEP_SECONDS,
```

```
) -> Tuple[Optional[datetime], Optional[datetime], Optional[int]]:
    """
    Tenta encontrar uma janela temporal sem conflito (fallback ao deconflito de altitude).

    A janela de voo é deslocada em múltiplos de `step_seconds` até o limite
    `max_shift_ratio * duração_original`, respeitando interseção espacial via buffer.

    Retorna:
        (new_start, new_end, shift_seconds)
        - (None, None, None) quando não há solução temporal dentro do limite.
    """
    duration = (desired_end - desired_start)
    duration_s = int(duration.total_seconds())
    if duration_s <= 0:
        return (None, None, None)

    # Intervalos ocupados por outros segmentos que se interceptam espacialmente
    cursor.execute(
        """
        WITH my AS (
            SELECT segmento::geometry AS g
            FROM segmentos
            WHERE request_id = %s
        ),
        other AS (
            SELECT s.segmento::geometry AS g, s.start_time_min, s.end_time_max
            FROM segmentos s
            WHERE s.request_id IN (SELECT request_id FROM missoes)
              AND s.request_id <> %s
        )
        SELECT DISTINCT o.start_time_min, o.end_time_max
        FROM my m
        JOIN other o
          ON ST_Intersects(
               ST_Buffer(m.g::geography, %s)::geometry,
               ST_Buffer(o.g::geography, %s)::geometry
             )
        ORDER BY 1;
        """,
```

```
        (request_id, request_id, spatial_buffer_m, spatial_buffer_m),
    )
    busy = cursor.fetchall()

    if not busy:
        return (desired_start, desired_end, 0)

    # Une intervalos ocupados com folga (padding) para evitar bordas de colisão
    intervals: List[Tuple[datetime, datetime]] = []
    for s_min, e_max in busy:
        s = s_min - timedelta(seconds=pad_seconds)
        e = e_max + timedelta(seconds=pad_seconds)
        intervals.append((s, e))
    intervals.sort(key=lambda x: x[0])

    merged: List[Tuple[datetime, datetime]] = []
    cur_s, cur_e = intervals[0]
    for s, e in intervals[1:]:
        if s <= cur_e:
            cur_e = max(cur_e, e)
        else:
            merged.append((cur_s, cur_e))
            cur_s, cur_e = s, e
    merged.append((cur_s, cur_e))

    def fits_at(t: datetime) -> Optional[Tuple[datetime, datetime]]:
        """Verifica se [t, t+duration] cabe sem sobrepor nenhum intervalo ocupado."
    ""
        end = t + duration
        for s, e in merged:
            if not (end <= s or t >= e):
                return None
        return (t, end)

    # Primeiro tenta exatamente no horário desejado
    fit = fits_at(desired_start)
    if fit:
        return (*fit, 0)

    # Busca a menor variação temporal possível (para frente e/ou para trás)
    max_shift_s = int(max_shift_ratio * duration_s)
```

```
    candidates: List[Tuple[int, datetime, datetime]] = []
    for d in range(step_seconds, max_shift_s + step_seconds, step_seconds):
        if direction in ("both", "forward"):
            t = desired_start + timedelta(seconds=d)
            fit = fits_at(t)
            if fit:
                candidates.append((d, *fit))
        if direction in ("both", "backward"):
            t = desired_start - timedelta(seconds=d)
            fit = fits_at(t)
            if fit:
                candidates.append((-d, *fit))

    if not candidates:
        return (None, None, None)

    candidates.sort(key=lambda x: abs(x[0]))
    best_shift, new_start, new_end = candidates[0]
    return (new_start, new_end, int(best_shift))


# ============================================================================
# Classe principal: Provider
# ============================================================================
class Provider:
    """
    Provedor UTM (camada de orquestração de missões/PVR com MQTT e banco).

    Atributos principais:
        provider_id: identificador do provedor
        drones: lista de drones gerenciados
        client: cliente MQTT (paho-mqtt v5)
    """

    def __init__(
        self,
        provider_id: int,
        broker: str = DEFAULT_BROKER,
        port: int = DEFAULT_PORT,
        keepalive: int = DEFAULT_KEEPALIVE,
    ) -> None:
```

```
186         self.provider_id = provider_id
187         self.logger = get_logger(f"Provider_{self.provider_id}")
188         self.broker = broker
189         self.port = port
190         self.keepalive = keepalive
191 
192         self.drones: List[int] = []
193         self._started = False
194         self._backoff = 1
195 
196         # Cliente MQTT (API v5)
197         self.client = mqtt.Client(
198             mqtt.CallbackAPIVersion.VERSION2,
199             client_id=f"provider_{provider_id}",
200             protocol=mqtt.MQTTv5,
201         )
202         self.client.on_connect = self.on_connect
203         self.client.on_message = self.on_message
204         self.client.on_disconnect = self.on_disconnect
205 
206         self.fetch_from_db()
207 
208     # ----------------- Acesso ao banco -----------------
209     def fetch_from_db(self) -> None:
210         """Carrega lista de drones pertencentes a este provedor."""
211         conn = None
212         try:
213             conn = get_db_connection()
214             cursor = conn.cursor()
215             cursor.execute(
216                 "SELECT drone_id FROM drones WHERE provider_id = %s",
217                 (self.provider_id,),
218             )
219             self.drones = [row[0] for row in cursor.fetchall()]
220             if self.drones:
221                 self.logger.info("Drones carregados: %s", self.drones)
222             else:
223                 self.logger.warning("Nenhum drone associado encontrado.")
224         except Exception as e:
225             self.logger.error("Erro ao buscar drones no banco: %s", e)
226         finally:
```

```
            if conn:
                cursor.close()
                conn.close()

    # ----------------- Callbacks MQTT -----------------
    def on_connect(self, client, userdata, flags, reason_code, properties) -> None:
        """Callback de conexão: assina tópicos do provedor e dos drones gerenciados."""
        if reason_code == 0:
            self.logger.info("Conectado ao broker.")
            self._backoff = 1
        else:
            self.logger.error("Falha ao conectar. reason_code=%s", reason_code)

        # Tópicos de status de cada drone
        for drone_id in self.drones:
            self.client.subscribe(f"drones/{drone_id}/status")
            self.logger.info("Inscrito em drones/%s/status", drone_id)

        # Tópicos do provedor
        self.client.subscribe("providers/communication")
        self.client.subscribe(f"providers/{self.provider_id}/communication")
        self.client.subscribe(f"providers/{self.provider_id}/requests/#")
        self.client.subscribe(f"providers/{self.provider_id}/pvr/#")
        self.client.subscribe(f"providers/{self.provider_id}/sdsp/response")
        self.logger.info(
            "Inscrito em providers/%s/{requests,#}, providers/%s/pvr/# e sdsp/response",
            self.provider_id,
            self.provider_id,
        )

    def on_disconnect(self, client, userdata, reason_code, properties) -> None:
        """Callback de desconexão: tenta reconectar com backoff simples."""
        self.logger.warning("Desconectado do broker. reason_code=%s", reason_code)
        if self._started:
            wait = min(self._backoff, 30)
            self.logger.info("Tentando reconectar em %ss...", wait)
            time.sleep(wait)
            try:
                self.client.reconnect()
```

```
266                 self._backoff = min(self._backoff * 2, 30)
267             except Exception as e:
268                 self.logger.error("Erro ao reconectar: %s", e)
269
270     def on_message(self, client, userdata, msg) -> None:
271         """Callback de mensagem: roteia por tópico (requisições, PVR e SDSP)."""
272         timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
273         payload_text = msg.payload.decode()
274         self.logger.info("[%s] Mensagem em %s: %s", timestamp, msg.topic,
    payload_text)
275
276         # Respostas SDSP
277         if msg.topic == f"providers/{self.provider_id}/sdsp/response":
278             try:
279                 data = json.loads(payload_text)
280                 self.logger.info("[SDSP] resposta: %s", data)
281             except Exception as e:
282                 self.logger.error("[SDSP] erro lendo resposta: %s", e)
283             return
284
285         # Requisições de missão (requests) e PVR
286         if "requests" in msg.topic or "pvr" in msg.topic:
287             try:
288                 _, provider_id, key, drone_id_str = msg.topic.split("/")
289                 drone_id = int(drone_id_str)
290                 request_id = int(payload_text)
291                 self.logger.info("Requisição ID: %s", request_id)
292
293                 if key == "requests":
294                     process_request(request_id)
295                     self.verificar_voo(drone_id, request_id)
296                 else:
297                     self.verificar_pvr(drone_id, request_id)
298             except ValueError as e:
299                 self.logger.error("Erro ao processar tópico/request_id: %s", e)
300
301     # ----------------- Lógica de PVR -----------------
302     def verificar_pvr(self, drone_id: int, request_id: int) -> None:
303         """Valida PVR (dentro do CTA, sem interseção com zonas restritas)."""
304         conn = None
305         try:
```

```
306             conn = get_db_connection()
307             cursor = conn.cursor()
308
309             cursor.execute(
310                 """
311                 SELECT ST_Within(r.area, ST_Force2D(m.geom)) AS dentro_cta
312                 FROM pvr r
313                 JOIN maps m ON m.id = 1
314                 WHERE r.pvr_id = %s;
315                 """,
316                 (request_id,),
317             )
318             dentro_cta = cursor.fetchone()[0]
319
320             cursor.execute(
321                 """
322                 SELECT COUNT(*)
323                 FROM pvr r
324                 JOIN maps m ON m.zone_type = 'restrita'
325                 WHERE r.pvr_id = %s
326                   AND ST_Intersects(r.area, ST_Force2D(m.geom));
327                 """,
328                 (request_id,),
329             )
330             intersecta_restrita = cursor.fetchone()[0] > 0
331
332             if not dentro_cta:
333                 resposta = "Voo negado: fora dos limites do CTA."
334             elif intersecta_restrita:
335                 resposta = "Voo negado: trajetória passa por área restrita."
336             else:
337                 resposta = "Voo autorizado."
338                 cursor.execute(
339                     "UPDATE pvr SET status = %s WHERE pvr_id = %s;",
340                     ("aprovado", request_id),
341                 )
342                 conn.commit()
343
344             self.resposta_req(drone_id, resposta)
345
346         except Exception as e:
```

```
347             self.logger.error("Erro ao verificar PVR: %s", e)
348         finally:
349             if conn:
350                 cursor.close()
351                 conn.close()
352 
353     # ----------------- Lógica de missão -----------------
354     def verificar_voo(self, drone_id: int, request_id: int) -> None:
355         """
356         Verifica missão em três etapas:
357         1) Checagens estáticas (CTA/restritas/PVR).
358         2) Deconflito por altitude.
359         3) Fallback temporal (quando 2 falha), rechecando conflitos.
360         """
361         conn = None
362         try:
363             conn = get_db_connection()
364             cursor = conn.cursor()
365 
366             # 1) Checagens estáticas
367             dentro_cta, intersecta_restrita, intersecta_pvr = check_restrictions(
368                 request_id, cursor
369             )
370             if not dentro_cta:
371                 self.resposta_req(
372                     drone_id, {"status": "negado", "reason": "Fora dos limites do
    CTA."}
373                 )
374                 return
375             if intersecta_restrita:
376                 self.resposta_req(
377                     drone_id,
378                     {"status": "negado", "reason": "Trajetória passa por área
    restrita."},
379                 )
380                 return
381             if intersecta_pvr:
382                 self.resposta_req(
383                     drone_id, {"status": "negado", "reason": "Conflito com PVR
    ativo."}
384                 )
```

```
385                 return
386
387             # Janela e trilha solicitadas
388             cursor.execute(
389                 """
390                 SELECT start_time, end_time, ST_Buffer(waypoints::geography, 5)::
    geometry
391                 FROM requisicoes
392                 WHERE request_id = %s;
393                 """,
394                 (request_id,),
395             )
396             start_time, end_time, track = cursor.fetchone()
397
398             # Altitudes de referência solicitadas
399             original_alt_min, original_alt_max, _, _ = operation_alt(request_id,
    cursor)
400
401             # 2) Tenta deconflito por ALTITUDE
402             authorized, alt_min, alt_max = resolve_altitude_conflicts(
403                 request_id, cursor
404             )
405
406             if not authorized:
407                 # 3) Fallback: deconflito por TEMPO
408                 new_start, new_end, shift_sec = temporal_deconflict(
409                     request_id,
410                     cursor,
411                     start_time,
412                     end_time,
413                     max_shift_ratio=DEFAULT_MAX_SHIFT_RATIO,
414                     spatial_buffer_m=DEFAULT_SPATIAL_BUFFER_M,
415                     pad_seconds=DEFAULT_PAD_SECONDS,
416                     direction=DEFAULT_DIRECTION,
417                     step_seconds=DEFAULT_STEP_SECONDS,
418                 )
419
420                 if new_start is None:
421                     self.resposta_req(
422                         drone_id,
423                         {
```

```
                            "status": "negado",
                            "reason": "Sem janela temporal livre dentro do limite de ajuste.",
                            "window": {
                                "requested_start": start_time.isoformat(sep=" "),
                                "requested_end": end_time.isoformat(sep=" "),
                            },
                            "request_id": request_id,
                        },
                    )
                    return

                # Rechecagem após ajuste temporal
                self.logger.info(
                    "Rechecando conflitos temporais em %s - %s...", new_start, new_end
                )

                # Missões aprovadas
                cursor.execute(
                    """
                    WITH my AS (
                        SELECT ST_Buffer(waypoints::geography, 5)::geometry AS geom
                        FROM requisicoes
                        WHERE request_id = %s
                    )
                    SELECT COALESCE(COUNT(*),0)
                    FROM missoes m, my
                    WHERE m.request_id <> %s
                      AND ST_Intersects(ST_Force2D(m.track), my.geom)
                      AND (m.start_time, m.end_time) OVERLAPS (%s, %s);
                    """,
                    (request_id, request_id, new_start, new_end),
                )
                conflict_with_mission = cursor.fetchone()[0] > 0

                # PVRs aprovados
                cursor.execute(
                    """
                    WITH my AS (
                        SELECT ST_Buffer(waypoints::geography, 5)::geometry AS geom
```

```
                        FROM requisicoes
                        WHERE request_id = %s
                    )
                    SELECT COALESCE(COUNT(*),0)
                    FROM pvr p, my
                    WHERE ST_Intersects(ST_Force2D(p.area), my.geom)
                      AND (p.start_time, p.end_time) OVERLAPS (%s, %s)
                      AND p.status = 'aprovado';
                    """,
                    (request_id, new_start, new_end),
                )
                conflict_with_pvr = cursor.fetchone()[0] > 0

                if conflict_with_mission or conflict_with_pvr:
                    reason = (
                        "Conflito com outra missão ativa"
                        if conflict_with_mission
                        else "Conflito com PVR ativo"
                    )
                    full_reason = f"{reason} após tentativa de ajuste temporal."
                    self.logger.warning("Negando voo: %s", full_reason)
                    self.resposta_req(
                        drone_id,
                        {
                            "status": "negado",
                            "reason": full_reason,
                            "details": {
                                "deconflict_method": "temporal",
                                "new_window": {
                                    "start": new_start.isoformat(sep=" "),
                                    "end": new_end.isoformat(sep=" "),
                                },
                            },
                            "request_id": request_id,
                        },
                    )
                    return

                # Sem conflitos: registra missão mantendo a altitude original (se definida)
                alt_min, alt_max = original_alt_min, original_alt_max
```

```
503                 if alt_min is None or alt_max is None:
504                     cursor.execute(
505                         """
506                         INSERT INTO missoes (request_id, track, start_time,
    end_time)
507                         VALUES (%s, %s, %s, %s)
508                         """,
509                         (request_id, track, new_start, new_end),
510                     )
511                 else:
512                     cursor.execute(
513                         """
514                         INSERT INTO missoes
515                             (request_id, track, alt_min, alt_max, start_time,
    end_time)
516                         VALUES (%s, %s, %s, %s, %s, %s)
517                         """,
518                         (request_id, track, alt_min, alt_max, new_start, new_end),
519                     )
520                 conn.commit()
521
522                 self.resposta_req(
523                     drone_id,
524                     {
525                         "status": "aprovado",
526                         "message": "Voo autorizado por ajuste de TEMPO.",
527                         "temporal": {
528                             "requested_start": start_time.isoformat(sep=" "),
529                             "requested_end": end_time.isoformat(sep=" "),
530                             "new_start": new_start.isoformat(sep=" "),
531                             "new_end": new_end.isoformat(sep=" "),
532                             "shift_seconds": int(shift_sec),
533                             "shift_ratio": round(
534                                 abs(shift_sec) / (end_time - start_time).
    total_seconds(),
535                                 3,
536                             ),
537                         },
538                         "altitude": {
539                             "final_min": _safe_round(alt_min),
540                             "final_max": _safe_round(alt_max),
```

```
                            "requested_min": _safe_round(original_alt_min),
                            "requested_max": _safe_round(original_alt_max),
                            "changed": False,
                            "note": "mantida",
                        },
                        "request_id": request_id,
                    },
                )
                return

            # ALTITUDE funcionou: registra missão com alt_min/alt_max finais
            altitude_changed = (alt_min != original_alt_min) or (alt_max != original_alt_max)
            cursor.execute(
                """
                INSERT INTO missoes (request_id, track, alt_min, alt_max, start_time, end_time)
                VALUES (%s, %s, %s, %s, %s, %s)
                """,
                (request_id, track, alt_min, alt_max, start_time, end_time),
            )
            conn.commit()

            self.resposta_req(
                drone_id,
                {
                    "status": "aprovado",
                    "message": "Voo autorizado.",
                    "altitude": {
                        "final_min": _safe_round(alt_min),
                        "final_max": _safe_round(alt_max),
                        "requested_min": _safe_round(original_alt_min),
                        "requested_max": _safe_round(original_alt_max),
                        "changed": altitude_changed,
                        "note": "mantida" if not altitude_changed else "alterada",
                    },
                    "window": {
                        "start_time": start_time.isoformat(sep=" "),
                        "end_time": end_time.isoformat(sep=" "),
                    },
                    "request_id": request_id,
```

```
                },
            )

        except Exception as e:
            self.logger.error("Erro na verificação do voo: %s", e)
            if conn:
                conn.rollback()
            self.resposta_req(
                drone_id, {"status": "erro", "reason": str(e), "request_id": request_id}
            )
        finally:
            if conn:
                cursor.close()
                conn.close()

    # ----------------- Utilidades -----------------
    def enviar_comando(self, drone_id: int, comando: str) -> None:
        """Publica um comando no tópico do drone, se ele for gerenciado por este provedor."""
        if drone_id in self.drones:
            timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
            mensagem = {"comando": comando, "timestamp": timestamp}
            self.client.publish(f"drones/{drone_id}/commands", json.dumps(mensagem))
            self.logger.info("Comando enviado para %s: %s", drone_id, comando)
        else:
            self.logger.warning(
                "Drone %s não é gerenciado por este provedor.", drone_id
            )

    def enviar_mensagem(self, conteudo: str, destinatario: Optional[int] = None) -> None:
        """Envia mensagem para um provedor específico ou em broadcast."""
        timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
        if destinatario is not None:
            mensagem = {
                "origem": self.provider_id,
                "conteudo": conteudo,
                "timestamp": timestamp,
            }
```

```
617             self.client.publish(
618                 f"providers/{destinatario}/communication", json.dumps(mensagem)
619             )
620             self.logger.info("Mensagem enviada para %s: %s", destinatario, conteudo
    )
621         else:
622             self.client.publish("providers/communication", conteudo)
623             self.logger.info("Mensagem broadcast enviada: %s", conteudo)
624 
625     def resposta_req(self, drone_id: int, ans: Any) -> None:
626         """Responde ao drone com payload JSON padronizado."""
627         if drone_id in self.drones:
628             ts = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
629             if isinstance(ans, dict):
630                 payload = {**ans, "timestamp": ts}
631             else:
632                 payload = {"resposta": ans, "timestamp": ts}
633             msg = json.dumps(payload, default=str)
634             self.client.publish(f"drones/{drone_id}/request", msg)
635             self.logger.info("Resposta enviada para %s: %s", drone_id, msg)
636         else:
637             self.logger.warning(
638                 "Drone %s não é gerenciado por este provedor.", drone_id
639             )
640 
641     # ----------------- Ciclo de vida -----------------
642     def iniciar(self) -> None:
643         """Inicia loop MQTT e conecta ao broker."""
644         if self._started:
645             self.logger.warning("Provider já iniciado; ignorando chamada duplicada.
    ")
646             return
647         self._started = True
648         self.client.loop_start()
649         self.client.connect(self.broker, self.port, keepalive=self.keepalive)
650 
651     def parar(self) -> None:
652         """Finaliza loop MQTT e encerra conexão."""
653         self._started = False
654         try:
655             self.client.loop_stop()
```

```
656             self.client.disconnect()
657         except Exception:
658             # Silencia exceções ao finalizar
659             pass
660 
661     # ----------------- Integração SDSP -----------------
662     def sdsp_weather_check(self, region: str, corr_id: str) -> None:
663         """Solicita boletim meteorológico ao SDSP."""
664         payload = {"type": "weather_check", "region": region, "corr_id": corr_id}
665         self.client.publish(
666             f"providers/{self.provider_id}/sdsp/request", json.dumps(payload), qos
    =1
667         )
668         self.logger.info("[SDSP] weather_check solicitado: %s", payload)
669 
670     def sdsp_zones_check(self, region: str, corr_id: str) -> None:
671         """Solicita zonas geoespaciais (NOTAM) ao SDSP."""
672         payload = {"type": "zones", "region": region, "corr_id": corr_id}
673         self.client.publish(
674             f"providers/{self.provider_id}/sdsp/request", json.dumps(payload), qos
    =1
675         )
676         self.logger.info("[SDSP] zones solicitado: %s", payload)
```

Código A.1 – Script principal do provedor UTM — gerenciamento de missões, verificação de restrições e comunicação MQTT

A.2 Drone

```
1  """
2  drone.py
3  --------
4 
5  Cliente do drone: conecta ao broker MQTT, assina tópicos do próprio drone,
6  envia requisições de voo/PVR e recebe comandos/mensagens/respostas.
7 
8  Principais responsabilidades:
9  - Conectar ao broker MQTT e assinar tópicos do drone;
10 - Publicar mensagens de controle e status;
11 - Registrar requisições de voo e PVR no banco e notificar o provider;
12 - Integrar com SDSP (subscribe de região).
```

```
13
14  Dependências externas:
15  - paho-mqtt (cliente MQTT 5)
16  - Banco PostgreSQL/PostGIS (via get_db_connection)
17  - logger interno (get_logger)
18  """
19
20  from __future__ import annotations
21
22  import json
23  import time
24  from datetime import datetime
25  from typing import Any, Optional
26
27  import paho.mqtt.client as mqtt
28
29  from db_connection import get_db_connection
30  from logger import get_logger
31
32  # ------------------------------------------------------
33  # Configuração padrão de conexão MQTT
34  # ------------------------------------------------------
35  DEFAULT_BROKER = "localhost"
36  DEFAULT_PORT = 1883
37  DEFAULT_KEEPALIVE = 300
38
39
40  class Drone:
41      """
42      Cliente de drone com integração a MQTT e banco de dados.
43
44      Atributos principais:
45          drone_id: identificador do drone
46          provider_id: identificador do provider responsável (carregado do BD)
47          client: cliente MQTT (paho-mqtt v5)
48      """
49
50      def __init__(
51          self,
52          drone_id: int,
53          broker: str = DEFAULT_BROKER,
```

```
        port: int = DEFAULT_PORT,
        keepalive: int = DEFAULT_KEEPALIVE,
    ) -> None:
        self.drone_id = drone_id
        self.logger = get_logger(f"Drone_{self.drone_id}")

        self.broker = broker
        self.port = port
        self.keepalive = keepalive

        # atributos carregados do BD (inicializados como None)
        self.model: Optional[str] = None
        self.operator_id: Optional[int] = None
        self.status: Optional[str] = None
        self.uas_type: Optional[str] = None
        self.weight: Optional[float] = None
        self.provider_id: Optional[int] = None
        self.visual_type: Optional[str] = None

        # cliente MQTT (API v5)
        self.client = mqtt.Client(
            mqtt.CallbackAPIVersion.VERSION2,
            client_id=f"drone_{drone_id}",
            protocol=mqtt.MQTTv5,
        )
        self.client.on_connect = self.on_connect
        self.client.on_message = self.on_message
        self.client.on_disconnect = self.on_disconnect

        # carrega metadados do drone
        self.fetch_from_db()

    # ----------------- Banco de dados -----------------
    def fetch_from_db(self) -> None:
        """Carrega atributos do drone a partir do banco."""
        conn = None
        try:
            conn = get_db_connection()
            cursor = conn.cursor()

            cursor.execute(
```

```
                "SELECT * FROM drones WHERE drone_id = %s",
                (self.drone_id,),
            )
            result = cursor.fetchone()
            if result:
                # mapeamento por posição conforme schema existente
                (
                    self.drone_id,
                    self.model,
                    self.operator_id,
                    self.status,
                    _created_at,  # coluna [4] ignorada, caso exista
                    self.uas_type,
                    self.weight,
                    self.provider_id,
                    self.visual_type,
                ) = (
                    result[0],
                    result[1],
                    result[2],
                    result[3],
                    result[4] if len(result) > 4 else None,
                    result[5] if len(result) > 5 else None,
                    result[6] if len(result) > 6 else None,
                    result[7] if len(result) > 7 else None,
                    result[8] if len(result) > 8 else None,
                )
                self.logger.info("Drone carregado do banco com sucesso.")
            else:
                self.logger.warning("Drone %s não encontrado no banco.", self.
    drone_id)

        except Exception as e:
            self.logger.error("Erro ao buscar drone no banco: %s", e)
        finally:
            if conn:
                cursor.close()
                conn.close()

    # ----------------- Callbacks MQTT -----------------
    def on_connect(self, client, userdata, flags, reason_code, properties) -> None:
```

```
135         """Assina os tópicos do drone após a conexão."""
136         if reason_code == 0:
137             self.logger.info("Conectado ao broker.")
138         else:
139             self.logger.error("Falha ao conectar. reason_code=%s", reason_code)
140 
141         # evita assinar sem provider
142         if self.provider_id is None:
143             self.logger.warning(
144                 "provider_id não definido para o drone %s. "
145                 "Verifique cadastro no BD.",
146                 self.drone_id,
147             )
148 
149         # tópicos do próprio drone
150         self.client.subscribe(f"drones/{self.drone_id}/commands")
151         self.client.subscribe(f"drones/{self.drone_id}/messages")
152         self.client.subscribe(f"drones/{self.drone_id}/request")
153         self.client.subscribe(f"drones/{self.drone_id}/pvr")
154         self.client.subscribe(f"sdsp/drones/{self.drone_id}/update")
155         self.logger.info(
156             "Inscrito em drones/%s/{commands,messages,request,pvr} e sdsp/drones/%s
    /update",
157             self.drone_id,
158             self.drone_id,
159         )
160 
161     def on_disconnect(self, client, userdata, reason_code, properties) -> None:
162         """Tenta reconectar com backoff simples."""
163         self.logger.warning("Desconectado do broker. reason_code=%s", reason_code)
164         try:
165             time.sleep(2)
166             self.client.reconnect()
167         except Exception as e:
168             self.logger.error("Erro ao reconectar: %s", e)
169 
170     def on_message(self, client, userdata, msg) -> None:
171         """Roteia mensagens recebidas para tratamento adequado."""
172         payload_text = msg.payload.decode()
173         self.logger.info("Mensagem recebida em %s: %s", msg.topic, payload_text)
174 
```

```
175         # Atualizações do SDSP
176         if msg.topic == f"sdsp/drones/{self.drone_id}/update":
177             try:
178                 data = json.loads(payload_text)
179                 self.logger.info("[SDSP] update: %s", data)
180             except Exception as e:
181                 self.logger.error("[SDSP] erro lendo update: %s", e)
182             return
183 
184         # Respostas do provider para o drone (aprovação/negação/erros)
185         if msg.topic == f"drones/{self.drone_id}/request":
186             # Apenas registra; o aplicativo cliente poderia tomar ações locais
187             self.logger.info("Resposta de requisição recebida: %s", payload_text)
188             return
189 
190         # Comandos diretos
191         if msg.topic == f"drones/{self.drone_id}/commands":
192             self.logger.info("Comando recebido: %s", payload_text)
193             return
194 
195     # ----------------- Publicações -----------------
196     def enviar_mensagem(self, destino_id: int, conteudo: str) -> None:
197         """Envia mensagem direta para outro drone."""
198         mensagem = {"origem": self.drone_id, "conteudo": conteudo}
199         self.client.publish(
200             f"drones/{destino_id}/messages", json.dumps(mensagem), qos=1
201         )
202         self.logger.info("Mensagem enviada para %s: %s", destino_id, conteudo)
203 
204     def enviar_requisicao_voo(
205         self,
206         start_point: Any,
207         end_point: Any,
208         waypoints: Any,
209         start_time: str,
210         end_time: str,
211     ) -> None:
212         """
213         Registra a requisição de voo no banco e notifica o provider.
214 
215         Observação:
```

```
            `start_point`, `end_point` e `waypoints` devem estar no formato
            esperado pelo schema (ex.: geometria WKT/GeoJSON armazenável).
        """
        conn = None
        try:
            conn = get_db_connection()
            cursor = conn.cursor()

            cursor.execute(
                """
                INSERT INTO requisicoes
                    (drone_id, start_point, end_point, waypoints, start_time,
end_time)
                VALUES (%s, %s, %s, %s, %s, %s)
                RETURNING request_id;
                """,
                (self.drone_id, start_point, end_point, waypoints, start_time,
end_time),
            )
            request_id = cursor.fetchone()[0]
            conn.commit()
            self.logger.info("Requisição registrada com ID: %s", request_id)

            # notifica o provider responsável
            if self.provider_id is None:
                self.logger.error(
                    "provider_id ausente; não é possível notificar provider."
                )
                return

            self.client.publish(
                f"providers/{self.provider_id}/requests/{self.drone_id}",
                json.dumps({"request_id": request_id}),
                qos=1,
            )
            self.logger.info(
                "Requisição enviada ao provider %s (drone %s, req %s).",
                self.provider_id,
                self.drone_id,
                request_id,
            )
```

```
255         except Exception as e:
256             if conn:
257                 conn.rollback()
258             self.logger.error("Erro ao enviar requisição de voo: %s", e)
259         finally:
260             if conn:
261                 cursor.close()
262                 conn.close()
263 
264     def enviar_requisicao_pvr(self, area_wkt: Any, start_time: str, end_time: str)
    -> None:
265         """Registra PVR no banco e notifica o provider responsável."""
266         conn = None
267         try:
268             conn = get_db_connection()
269             cursor = conn.cursor()
270 
271             if self.provider_id is None:
272                 self.logger.error(
273                     "provider_id ausente; não é possível registrar/enviar PVR."
274                 )
275                 return
276 
277             cursor.execute(
278                 """
279                 INSERT INTO pvr (drone_id, provider_id, area, start_time, end_time)
280                 VALUES (%s, %s, %s, %s, %s)
281                 RETURNING pvr_id;
282                 """,
283                 (self.drone_id, self.provider_id, area_wkt, start_time, end_time),
284             )
285             pvr_id = cursor.fetchone()[0]
286             conn.commit()
287             self.logger.info("PVR registrada com ID: %s", pvr_id)
288 
289             self.client.publish(
290                 f"providers/{self.provider_id}/pvr/{self.drone_id}",
291                 json.dumps({"pvr_id": pvr_id}),
292                 qos=1,
293             )
294             self.logger.info(
```

```
                "PVR enviada para o provider %s (drone %s, pvr %s).",
                self.provider_id,
                self.drone_id,
                pvr_id,
            )
        except Exception as e:
            if conn:
                conn.rollback()
            self.logger.error("Erro ao enviar PVR: %s", e)
        finally:
            if conn:
                cursor.close()
                conn.close()

    def sdsp_subscribe_region(self, region: str) -> None:
        """Pede ao SDSP assinatura de atualizações para uma região."""
        payload = {"region": region}
        self.client.publish(
            f"drones/{self.drone_id}/sdsp/subscribe", json.dumps(payload), qos=1
        )
        self.logger.info("[SDSP] subscribe enviado: %s", payload)

    # ----------------- Ciclo de vida -----------------
    def iniciar(self) -> None:
        """Inicia o loop MQTT e conecta ao broker."""
        self.client.loop_start()
        self.client.connect(self.broker, self.port, keepalive=self.keepalive)

    def parar(self) -> None:
        """Finaliza o loop MQTT e encerra a conexão."""
        try:
            self.client.loop_stop()
            self.client.disconnect()
        except Exception:
            pass
```

Código A.2 – Cliente do drone: envio de requisições de voo e PVR, comunicação com o provedor e integração com o SDSP

A.3 SDSP

```python
"""
sdsp.py
-------

Serviço SDSP (simples) usado no ambiente de simulação:
- Atende requisições do provider:
    providers/{provider_id}/sdsp/request  ->  providers/{provider_id}/sdsp/response
- Recebe inscrição de drones e envia atualizações:
    drones/{drone_id}/sdsp/subscribe      ->  sdsp/drones/{drone_id}/update
- Publica boletins retidos por região:
    sdsp/status/health
    sdsp/weather/now/{region_id}
    sdsp/geo/zones/{region_id}

Observação:
Este módulo não integra APIs externas; gera dados mockados e publica no broker.
"""

from __future__ import annotations

import json
import time
import threading
from datetime import datetime
from typing import Dict, List

import paho.mqtt.client as mqtt

# ---------------------------------------------------------------------
# Parâmetros de conexão e publicação
# ---------------------------------------------------------------------
BROKER = "localhost"
PORT = 1883
KEEPALIVE = 300
QOS = 1

# ---------------------------------------------------------------------
# Utilidades internas
# ---------------------------------------------------------------------
def _now() -> str:
    """Timestamp UTC ISO-8601 (precisão de segundos) com sufixo 'Z'."""
```

```
    return datetime.utcnow().isoformat(timespec="seconds") + "Z"


def _j(data: dict) -> str:
    """JSON compacto, preservando acentos e convertendo tipos básicos."""
    return json.dumps(data, ensure_ascii=False, separators=(",", ":"), default=str)


def _mock_weather(region: str) -> dict:
    """Gera condições meteorológicas determinísticas por região (mock)."""
    seed = sum(ord(c) for c in region) % 360
    wind_dir = seed
    wind_kt = 6 + (seed % 12)
    gust_kt = wind_kt + (seed % 6)
    visibility = 7 + (seed % 4)
    ceiling = 1500 + (seed % 600)
    precip = "none" if seed % 3 == 0 else ("light" if seed % 3 == 1 else "moderate"
    )
    return {
        "wind": {"dir_deg": wind_dir, "speed_kt": wind_kt, "gust_kt": gust_kt},
        "visibility_km": visibility,
        "cloud_ceiling_ft": ceiling,
        "precip_intensity": precip,
        "obs_time": _now(),
    }


def _mock_zones(region: str) -> List[dict]:
    """Gera zona temporária fictícia (mock) para uma região."""
    base = abs(hash(region)) % 1000
    return [
        {
            "id": f"TEMP-{region}-{base}",
            "restriction": "no-fly",
            "alt_ft": [0, 1200],
            "valid_from": _now(),
            "valid_to": _now(),
            "source": "MOCK-NOTAM",
            "geom": {
                "type": "Polygon",
                "coordinates": [
```

```
                    [
                        [-46.58, -23.65],
                        [-46.57, -23.65],
                        [-46.57, -23.64],
                        [-46.58, -23.64],
                        [-46.58, -23.65],
                    ]
                ],
            },
        }
    ]


# ---------------------------------------------------------------------
# Serviço SDSP
# ---------------------------------------------------------------------
class SDSP:
    """
    Serviço SDSP com interface MQTT.

    Responsabilidades:
        - Processar pedidos do provider (weather/zones) e responder;
        - Aceitar inscrições de drones e enviar updates;
        - Publicar boletins retidos por região em intervalo periódico.

    Atributos principais:
        client: cliente MQTT (paho-mqtt v5)
        drone_region: mapa drone_id -> região inscrita
    """

    def __init__(self, broker: str = BROKER, port: int = PORT, keepalive: int =
    KEEPALIVE) -> None:
        self.client = mqtt.Client(
            mqtt.CallbackAPIVersion.VERSION2,
            client_id="sdsp_service",
            protocol=mqtt.MQTTv5,
        )
        self.client.on_connect = self.on_connect
        self.client.on_message = self.on_message
        self.client.on_disconnect = self.on_disconnect

```

```
122         self.broker, self.port, self.keepalive = broker, port, keepalive
123         self.drone_region: Dict[int, str] = {}  # memória simples de assinaturas
124 
125     # ----------------- Ciclo de vida -----------------
126     def start(self) -> None:
127         """Inicia loop MQTT, conecta ao broker e dispara broadcasts periódicos."""
128         self.client.loop_start()
129         self.client.connect(self.broker, self.port, keepalive=self.keepalive)
130         threading.Thread(target=self._broadcast_loop, daemon=True).start()
131 
132     def stop(self) -> None:
133         """Encerra loop e desconecta do broker."""
134         try:
135             self.client.loop_stop()
136             self.client.disconnect()
137         except Exception:
138             pass
139 
140     # ----------------- Callbacks MQTT -----------------
141     def on_connect(self, client, userdata, flags, reason_code, properties) -> None:
142         """Assina tópicos necessários e publica status inicial (retido)."""
143         # provider -> sdsp (requests)
144         self.client.subscribe("providers/+/sdsp/request", qos=QOS)
145         # drone -> sdsp (subscription)
146         self.client.subscribe("drones/+/sdsp/subscribe", qos=QOS)
147         # publicar health retained
148         self.client.publish(
149             "sdsp/status/health",
150             _j({"service": "sdsp", "status": "ok", "capabilities": ["weather", "
    zones"], "ts": _now()}),
151             qos=QOS,
152             retain=True,
153         )
154 
155     def on_disconnect(self, client, userdata, reason_code, properties) -> None:
156         """Callback de desconexão (sem reconexão automática aqui)."""
157         # manter simples, reconexão pode ser orquestrada externamente
158         pass
159 
160     def on_message(self, client, userdata, msg) -> None:
161         """Roteia mensagens por tópico."""
```

```
        topic = msg.topic
        try:
            payload = json.loads(msg.payload.decode() or "{}")
        except Exception:
            payload = {}

        # providers/{provider_id}/sdsp/request
        if topic.startswith("providers/") and topic.endswith("/sdsp/request"):
            provider_id = topic.split("/")[1]
            self._handle_provider_request(provider_id, payload)

        # drones/{drone_id}/sdsp/subscribe
        elif topic.startswith("drones/") and topic.endswith("/sdsp/subscribe"):
            drone_id = int(topic.split("/")[1])
            self._handle_drone_subscribe(drone_id, payload)

    # ----------------- Handlers -----------------
    def _handle_provider_request(self, provider_id: str, data: dict) -> None:
        """Trata pedido do provider (tipo: weather_check | zones)."""
        typ = data.get("type", "weather_check")
        region = data.get("region", "SBSP")
        corr = data.get("corr_id", "na")

        if typ == "weather_check":
            result = _mock_weather(region)
        elif typ == "zones":
            result = {"zones": _mock_zones(region)}
        else:
            result = {"info": "unknown request type"}

        resp = {"corr_id": corr, "region": region, "result": result, "status": "ok"
    , "timestamp": _now()}
        self.client.publish(f"providers/{provider_id}/sdsp/response", _j(resp), qos
    =QOS)

    def _handle_drone_subscribe(self, drone_id: int, data: dict) -> None:
        """Registra inscrição do drone e envia update imediato."""
        region = data.get("region", "SBSP")
        self.drone_region[drone_id] = region

        upd = {"region": region, "weather": _mock_weather(region), "ts": _now()}
```

```
        self.client.publish(f"sdsp/drones/{drone_id}/update", _j(upd), qos=QOS)

    # ----------------- Broadcasts periódicos -----------------
    def _broadcast_loop(self) -> None:
        """Publica boletins periódicos por região e atualizações para drones inscritos."""
        regions_seed = ["SBSP", "SBRJ", "SBGR"]
        while True:
            # boletins retidos por região
            for region in regions_seed:
                self.client.publish(
                    f"sdsp/weather/now/{region}",
                    _j({"region": region, "now": _mock_weather(region), "ttl_seconds": 300, "ts": _now()}),
                    qos=QOS,
                    retain=True,
                )
                self.client.publish(
                    f"sdsp/geo/zones/{region}",
                    _j({"region": region, "zones": _mock_zones(region), "ttl_seconds": 600, "ts": _now()}),
                    qos=QOS,
                    retain=True,
                )
            # push direto para drones inscritos
            for drone_id, region in list(self.drone_region.items()):
                upd = {"region": region, "weather": _mock_weather(region), "ts": _now()}
                self.client.publish(f"sdsp/drones/{drone_id}/update", _j(upd), qos=QOS)

            time.sleep(30)
```

Código A.3 – Serviço SDSP simplificado: fornecimento de dados suplementares (meteorologia e zonas geográficas) via MQTT

A.4 Verificação de restrições estáticas

```
"""
check_static.py
---------------
```

```
4
5   Verificações estáticas de restrições para uma requisição de voo:
6   - Confirma se a trajetória está dentro dos limites do CTA.
7   - Detecta interseções com áreas restritas (maps.zone_type = 'restrita').
8   - Verifica conflitos com PVRs ativos (status = 'aprovado') no mesmo período
        temporal.
9
10  Retorna uma tupla:
11      (dentro_cta, intersecta_restrita, intersecta_pvr)
12
13  Dependências:
14  - Requer conexão ativa e cursor PostgreSQL/PostGIS.
15  - O banco deve conter as tabelas: requisicoes, maps e pvr.
16  """
17
18  def check_restrictions(request_id, cursor):
19      """
20      Executa três verificações espaciais e temporais para a requisição informada.
21
22      Args:
23          request_id (int): ID da requisição de voo.
24          cursor (psycopg2.cursor): Cursor ativo da conexão PostgreSQL/PostGIS.
25
26      Returns:
27          tuple[bool, bool, bool]:
28              (dentro_cta, intersecta_restrita, intersecta_pvr)
29      """
30      # 1) Dentro do CTA (maps.id = 1)
31      cursor.execute(
32          """
33          SELECT ST_Within(r.waypoints, ST_Force2D(m.geom)) AS dentro_cta
34          FROM requisicoes r
35          JOIN maps m ON m.id = 1
36          WHERE r.request_id = %s;
37          """,
38          (request_id,),
39      )
40      dentro_cta = cursor.fetchone()[0]
41
42      # 2) Interseção com áreas restritas (maps.zone_type = 'restrita')
43      cursor.execute(
```

```
        """
        SELECT COUNT(*)
        FROM requisicoes r
        JOIN maps m ON m.zone_type = 'restrita'
        WHERE r.request_id = %s
          AND ST_Intersects(r.waypoints, ST_Force2D(m.geom));
        """,
        (request_id,),
    )
    intersecta_restrita = cursor.fetchone()[0] > 0

    # 3) Interseção com PVR ativo (status = 'aprovado')
    #    Avalia sobreposição espacial e temporal simultânea.
    cursor.execute(
        """
        SELECT COUNT(*)
        FROM requisicoes r
        JOIN pvr p
          ON p.status = 'aprovado'
         AND r.start_time <= p.end_time
         AND r.end_time   >= p.start_time
        WHERE r.request_id = %s
          AND ST_Intersects(
                ST_Buffer(r.waypoints::geography, 5)::geometry,  -- 5m de folga
    lateral
                ST_Force2D(p.area)
              );
        """,
        (request_id,),
    )
    intersecta_pvr = cursor.fetchone()[0] > 0

    return dentro_cta, intersecta_restrita, intersecta_pvr
```

Código A.4 – Rotina de verificação de restrições espaciais e temporais (CTA, áreas restritas e PVR ativo)

A.5 Cálculo de altitudes operacionais

```
"""
check_alt.py
------------
```

```
4
5   Funções auxiliares para cálculo e recuperação de altitudes de missões.
6
7   Principais funcionalidades:
8   - Calcular altitude operacional (mínima/máxima) com base nos polígonos e segmentos
        associados.
9   - Consultar altitudes registradas de uma missão já aprovada.
10
11  Dependências:
12  - Conexão PostgreSQL/PostGIS com tabelas:
13      requisicoes, maps, segmentos e missoes.
14  """
15
16  def operation_alt(request_id, cursor):
17      """
18      Calcula altitudes operacionais da requisição informada.
19
20      Etapas:
21        1. Verifica polígonos interceptados pela trajetória e calcula a maior elevaçã
        o (height + alt).
22        2. Obtém altitudes mínima e máxima registradas na tabela 'segmentos'.
23        3. Define piso e teto da trajetória considerando margens verticais.
24
25      Args:
26          request_id (int): ID da requisição de voo.
27          cursor (psycopg2.cursor): Cursor ativo para o banco de dados.
28
29      Returns:
30          tuple[float | None, float | None, float | None, float | None]:
31              (altitude_min, altitude_max, alt_piso, alt_teto)
32              Retorna (None, None, None, None) se não houver dados suficientes
33              ou (None, None) se o voo ultrapassar o teto operacional.
34      """
35      # 1) Verifica polígonos interceptados e calcula a altura máxima combinada (
        height + alt)
36      cursor.execute(
37          """
38          WITH trajetoria AS (
39              SELECT ST_Buffer(waypoints::geography, 5)::geometry AS waypoints_buffer
40              FROM requisicoes
41              WHERE request_id = %s
```

```
        ),
        poligonos_interceptados AS (
            SELECT m.id, m.height, m.alt
            FROM maps m, trajetoria t
            WHERE ST_Intersects(m.geom, t.waypoints_buffer)
        )
        SELECT MAX(m.height + m.alt) AS altura_maxima_poligono
        FROM poligonos_interceptados m;
        """,
        (request_id,),
    )
    altura_maxima_poligono = cursor.fetchone()[0]

    # 2) Busca altitudes mínima e máxima da tabela 'segmentos' para o mesmo request_id
    cursor.execute(
        """
        SELECT MIN(alt), MAX(alt)
        FROM segmentos
        WHERE request_id = %s;
        """,
        (request_id,),
    )
    resultado_segmentos = cursor.fetchone()
    altitude_min_segmentos, altitude_max_segmentos = resultado_segmentos

    # 3) Define piso e teto operacionais
    if altura_maxima_poligono is not None:
        # piso: o ponto mais alto entre o terreno/obstáculo e o segmento máximo
        alt_piso = max(altura_maxima_poligono, altitude_max_segmentos)
    else:
        alt_piso = altitude_max_segmentos

    # teto: o ponto mais baixo + 400 pés de separação
    alt_teto = altitude_min_segmentos + 400

    # Conversão e margem operacional (10 a 30 metros acima do piso)
    alt_min_op = alt_piso + 10 * 3.281
    alt_max_op = alt_piso + 30 * 3.281

    # Se a faixa exceder o teto operacional, o voo é inválido
```

```
    if alt_max_op > alt_teto:
        return None, None, None, None

    return alt_min_op, alt_max_op, alt_piso, alt_teto


def get_mission_alt(request_id, cursor):
    """
    Obtém altitudes registradas de uma missão aprovada.

    Args:
        request_id (int): ID da requisição de voo.
        cursor (psycopg2.cursor): Cursor ativo para o banco de dados.

    Returns:
        tuple[float | None, float | None]:
            (alt_min, alt_max) da missão correspondente, ou (None, None) se ausente
    .
    """
    try:
        cursor.execute(
            "SELECT alt_min, alt_max FROM missoes WHERE request_id = %s;",
            (request_id,),
        )
        result = cursor.fetchone()
        if result:
            return result[0], result[1]
        return (None, None)
    except Exception as e:
        print(f"Erro ao obter altitudes da missão {request_id}: {str(e)}")
        return (None, None)
```

Código A.5 – Funções auxiliares para cálculo automático de piso e teto operacionais e obtenção de altitudes registradas

A.6 Deconflito espacial e temporal

```
"""
deconflit.py
------------

```

```
Rotinas de deconflito espacial/temporal e alocação de altitudes para requisições.

Principais funcionalidades:
- Detectar conflitos de rota (proximidade/interseção entre trilhas).
- Detectar conflitos entre segmentos com janela temporal sobreposta.
- Resolver conflitos por ALTITUDE (prioritário).
- (Fallback) Sugerir ajuste por TEMPO quando não há altitude viável.

Dependências:
- PostgreSQL/PostGIS (tabelas: requisicoes, segmentos, missoes).
- check_alt.operation_alt, check_alt.get_mission_alt
- logger interno (get_logger)
"""

from __future__ import annotations

from datetime import datetime, timedelta
from typing import List, Optional, Sequence, Tuple

from psycopg2 import sql

from check_alt import get_mission_alt, operation_alt
from logger import get_logger

logger = get_logger("Deconflit")

FT_PER_M = 3.281  # conversão aproximada de metro para pés


# ============================================================================
# Conflitos de rota e de segmento
# ============================================================================
def check_route_conflict(
    request_id: int, cursor, buffer_meters: int = 5
) -> List[int]:
    """
    Retorna lista de request_id de outras requisições (já aprovadas como missões)
    cuja trilha esteja próxima/intersecte a trilha desta requisição.

    Critérios:
      - distância geodésica até `buffer_meters` (ST_DWithin)
```

```
      - interseção entre buffers das trilhas (ST_Intersects com ST_Buffer)

    Args:
        request_id: requisição a ser avaliada.
        cursor: cursor PostgreSQL/PostGIS ativo.
        buffer_meters: raio lateral para considerar proximidade/interseção.

    Returns:
        Lista de request_id em conflito de rota.
    """
    try:
        cursor.execute(
            """
            SELECT r2.request_id
            FROM requisicoes r1
            LEFT JOIN requisicoes r2
              ON r2.request_id != r1.request_id
             AND r2.request_id IN (SELECT request_id FROM missoes)
            WHERE r1.request_id = %s
              AND ST_DWithin(
                    r1.waypoints::geography,
                    r2.waypoints::geography,
                    %s
                )
              AND ST_Intersects(
                    ST_Buffer(r1.waypoints::geography, %s)::geometry,
                    ST_Buffer(r2.waypoints::geography, %s)::geometry
                );
            """,
            (request_id, buffer_meters, buffer_meters, buffer_meters),
        )
        conflicts = [row[0] for row in cursor.fetchall() if row[0] is not None]
        logger.debug("[%s] Conflitos de rota: %s", request_id, conflicts)
        return conflicts
    except Exception as e:
        logger.error("[%s] Erro em check_route_conflict: %s", request_id, e)
        return []


def check_segment_conflict(
    request_id: int,
```

```
    cursor,
    route_conflicts: Sequence[int],
    buffer_meters: int = 5,
) -> List[Tuple[int, int]]:
    """
    Retorna conflitos em nível de segmento com sobreposição temporal.

    Para cada segmento da requisição, verifica interseção espacial com segmentos
    de outras missões (route_conflicts) e a condição temporal:
        (meu_start <= end_max_outro) AND (meu_end >= start_min_outro)

    Args:
        request_id: requisição a ser avaliada.
        cursor: cursor PostgreSQL/PostGIS ativo.
        route_conflicts: lista de request_id previamente detectados em conflito de rota.
        buffer_meters: raio para buffers laterais (geography).

    Returns:
        Lista de tuplas (segment_id_conflitante, request_id_conflitante).
    """
    if not route_conflicts:
        return []

    conflicts: List[Tuple[int, int]] = []
    try:
        # Segmentos da requisição avaliada
        cursor.execute(
            """
            SELECT segment_id, segmento, start_time_min, end_time_max
            FROM segmentos
            WHERE request_id = %s;
            """,
            (request_id,),
        )
        current_segments = cursor.fetchall()

        # Montagem segura da cláusula IN com psycopg2.sql
        in_list = sql.SQL(",").join(map(sql.Literal, route_conflicts))

        for seg_id, seg_wkb_hex, start_min, end_max in current_segments:
```

```
            query = sql.SQL(
                """
                SELECT s.segment_id, s.request_id
                FROM segmentos s
                WHERE s.request_id <> %s
                  AND s.request_id IN ({route_conflicts})
                  AND ST_Intersects(
                        ST_Buffer(ST_GeomFromEWKB(%s)::geography, %s),
                        ST_Buffer(s.segmento::geography, %s)
                      )
                  AND (%s <= s.end_time_max AND %s >= s.start_time_min);
                """
            ).format(route_conflicts=in_list)

            cursor.execute(
                query,
                (
                    request_id,
                    bytes.fromhex(seg_wkb_hex),
                    buffer_meters,
                    buffer_meters,
                    start_min,
                    end_max,
                ),
            )
            for s_id, s_req in cursor.fetchall():
                conflicts.append((s_id, s_req))

        # Remove duplicatas
        conflicts = list(set(conflicts))
        logger.debug("[%s] Conflitos de segmento: %s", request_id, conflicts)
        return conflicts

    except Exception as e:
        logger.error("[%s] Erro em check_segment_conflict: %s", request_id, e)
        return []


# ============================================================================
# Deconflito por ALTITUDE
# ============================================================================
```

```python
def resolve_altitude_conflicts(
    request_id: int, cursor
) -> Tuple[bool, Optional[float], Optional[float]]:
    """
    Tenta alocar janela de altitude viável, priorizando manter a altitude original.

    Passos:
      1) Calcula altitudes operacionais (operation_alt) e piso/teto.
      2) Verifica conflitos de rota/segmento com missões ativas.
      3) Se houver conflitos, tenta achar janela livre com find_available_altitude_window.

    Returns:
        (authorized, alt_min, alt_max)
        - authorized=False quando não há janela de altitude possível.
    """
    alt_min, alt_max, alt_piso, alt_teto = operation_alt(request_id, cursor)
    logger.info(
        "[%s] Altitude original: %s-%s | Piso/Teto: %s-%s",
        request_id,
        alt_min,
        alt_max,
        alt_piso,
        alt_teto,
    )

    route_conflicts = check_route_conflict(request_id, cursor)
    logger.debug("[%s] Conflitos de rota: %s", request_id, route_conflicts)

    if not route_conflicts:
        logger.info("[%s] Sem conflitos de rota. Mantendo altitude original.", request_id)
        return True, alt_min, alt_max

    segment_conflicts = check_segment_conflict(request_id, cursor, route_conflicts)
    logger.debug("[%s] Conflitos de segmento: %s", request_id, segment_conflicts)

    if not segment_conflicts:
        logger.info(
            "[%s] Sem conflitos de segmento. Mantendo altitude original.", request_id
```

```
206         )
207         return True, alt_min, alt_max
208
209     # Coleta janelas de altitude ocupadas por missões em conflito
210     conflicting_windows: List[Tuple[float, float]] = []
211     for seg_id, mission_id in segment_conflicts:
212         try:
213             conflict_alt_min, conflict_alt_max = get_mission_alt(mission_id, cursor
    )
214             if conflict_alt_min is not None and conflict_alt_max is not None:
215                 conflicting_windows.append((conflict_alt_min, conflict_alt_max))
216                 logger.debug(
217                     "[%s] Missão %s (seg %s): Alt %s-%s",
218                     request_id,
219                     mission_id,
220                     seg_id,
221                     conflict_alt_min,
222                     conflict_alt_max,
223                 )
224         except Exception as e:
225             logger.error(
226                 "[%s] Erro ao obter altitudes da missão %s: %s", request_id,
    mission_id, e
227             )
228
229     logger.info(
230         "[%s] Total de janelas conflitantes: %d",
231         request_id,
232         len(conflicting_windows),
233     )
234     logger.debug("[%s] Janelas: %s", request_id, conflicting_windows)
235
236     new_alt_min, new_alt_max = find_available_altitude_window(
237         conflicting_windows, alt_piso, alt_teto
238     )
239
240     if new_alt_min is not None:
241         logger.info(
242             "[%s] Nova janela alocada: %s-%s", request_id, new_alt_min, new_alt_max
243         )
244         return True, new_alt_min, new_alt_max
```

```
245
246     logger.warning("[%s] Sem janela de altitude livre.", request_id)
247     return False, None, None
248
249
250 def find_available_altitude_window(
251     conflicting_windows: Sequence[Tuple[float, float]],
252     alt_piso: float,
253     alt_teto: float,
254 ) -> Tuple[Optional[float], Optional[float]]:
255     """
256     Dada a lista de janelas ocupadas e [alt_piso, alt_teto], encontra janela livre.
257
258     Regras:
259       - Ordena janelas ocupadas por altitude mínima.
260       - Aplica buffer vertical de 10 m (5 m) ao redor de cada janela ocupada.
261       - Exige largura mínima de ~30 m (em pés) para a janela livre.
262       - Retorna uma faixa padronizada de 20 m, 5 m acima do começo da janela livre.
263
264     Returns:
265         (new_alt_min, new_alt_max) ou (None, None) se não houver espaço.
266     """
267     if alt_piso is None or alt_teto is None:
268         return (None, None)
269
270     # Ordena e aplica buffer vertical
271     buffered: List[Tuple[float, float]] = []
272     for win_min, win_max in sorted(conflicting_windows, key=lambda x: x[0]):
273         buffered.append((win_min - 5 * FT_PER_M, win_max + 5 * FT_PER_M))  # 10 m
274
275     # Varre as lacunas livres
276     free: List[Tuple[float, float]] = []
277     current = alt_piso
278     for win_min, win_max in buffered:
279         if current < win_min:
280             free.append((current, win_min))
281         current = max(current, win_max)
282
283     if current < alt_teto:
284         free.append((current, alt_teto))
285
```

```
    required_size = 30 * FT_PER_M  # 30 m
    for fmin, fmax in free:
        if (fmax - fmin) >= required_size:
            # janela padronizada: 20 m, iniciando 5 m acima da borda inferior
            new_min = fmin + 5 * FT_PER_M
            new_max = new_min + 20 * FT_PER_M
            return (new_min, new_max)

    return (None, None)


# ============================================================================
# (Fallback) Deconflito por TEMPO
# ============================================================================
def temporal_deconflict(
    request_id: int,
    cursor,
    desired_start: datetime,
    desired_end: datetime,
    max_shift_ratio: float = 0.5,  # limite |shift|/duração (01)
    spatial_buffer_m: int = 10,    # mesmo buffer usado nos checks
    pad_seconds: int = 60,         # acolchoar janelas ocupadas
    direction: str = "both",       # "both" | "forward" | "backward"
) -> Tuple[Optional[datetime], Optional[datetime], Optional[int]]:
    """
    Busca (new_start, new_end, shift_seconds) para a MESMA duração sem conflito
    temporal
    com missões cuja trilha intersecta a minha (espacialmente). Caso não seja possí
    vel,
    retorna (None, None, None).
    """
    duration = (desired_end - desired_start)
    duration_s = int(duration.total_seconds())
    if duration_s <= 0:
        return (None, None, None)

    # 1) Janelas ocupadas por missões ativas que INTERSECTAM espacialmente
    cursor.execute(
        """
        WITH my AS (
            SELECT segmento::geometry AS g
```

```
            FROM segmentos
            WHERE request_id = %s
        ),
        other AS (
            SELECT s.segmento::geometry AS g, s.start_time_min, s.end_time_max
            FROM segmentos s
            WHERE s.request_id IN (SELECT request_id FROM missoes)
              AND s.request_id <> %s
        )
        SELECT DISTINCT o.start_time_min, o.end_time_max
        FROM my m
        JOIN other o
          ON ST_Intersects(
               ST_Buffer(m.g::geography, %s)::geometry,
               ST_Buffer(o.g::geography, %s)::geometry
             )
        ORDER BY 1;
        """,
        (request_id, request_id, spatial_buffer_m, spatial_buffer_m),
    )
    busy = cursor.fetchall()

    if not busy:
        return (desired_start, desired_end, 0)

    # 2) Acolchoa e mescla intervalos ocupados
    intervals: List[Tuple[datetime, datetime]] = []
    for s_min, e_max in busy:
        s = s_min - timedelta(seconds=pad_seconds)
        e = e_max + timedelta(seconds=pad_seconds)
        intervals.append((s, e))
    intervals.sort(key=lambda x: x[0])

    merged: List[Tuple[datetime, datetime]] = []
    cur_s, cur_e = intervals[0]
    for s, e in intervals[1:]:
        if s <= cur_e:
            cur_e = max(cur_e, e)
        else:
            merged.append((cur_s, cur_e))
            cur_s, cur_e = s, e
```

```
366     merged.append((cur_s, cur_e))
367
368     # 3) Varre shifts possíveis conforme direção/limite
369     def fits_at(t: datetime) -> Optional[Tuple[datetime, datetime]]:
370         end = t + duration
371         for s, e in merged:
372             if not (end <= s or t >= e):
373                 return None
374         return (t, end)
375
376     fit = fits_at(desired_start)
377     if fit:
378         return (*fit, 0)
379
380     max_shift_s = int(max_shift_ratio * duration_s)
381     step = 15  # segundos
382
383     candidates: List[Tuple[int, datetime, datetime]] = []
384     for d in range(step, max_shift_s + step, step):
385         if direction in ("both", "forward"):
386             t = desired_start + timedelta(seconds=d)
387             fwd = fits_at(t)
388             if fwd:
389                 candidates.append((d, *fwd))
390         if direction in ("both", "backward"):
391             t = desired_start - timedelta(seconds=d)
392             bwd = fits_at(t)
393             if bwd:
394                 candidates.append((-d, *bwd))
395
396     if not candidates:
397         return (None, None, None)
398
399     candidates.sort(key=lambda x: abs(x[0]))
400     best_shift, new_start, new_end = candidates[0]
401     return (new_start, new_end, int(best_shift))
```

Código A.6 – Rotinas de detecção e resolução de conflitos de rota, segmentos e altitude, com fallback temporal

A.7 Geração de segmentos e janelas temporais

```
"""
register_segment.py
-------------------

Divide a trajetória de uma requisição em segmentos (~100 m cada),
calcula as janelas temporais proporcionais e registra em ‘segmentos‘.

Opcionalmente, consulta a Google Elevation API para obter a altitude
nos pontos inicial/final de cada segmento.

Segurança:
- NÃO armazene a chave no código. Use parâmetro explicitamente
  na chamada ou variável de ambiente GOOGLE_ELEVATION_API_KEY.

Dependências:
- shapely, pyproj, requests
- get_db_connection (PostgreSQL/PostGIS)
- logger (get_logger)
"""

from __future__ import annotations

import binascii
import math
import os
from datetime import timedelta
from typing import Iterable, List, Optional, Tuple

import requests
from pyproj import CRS, Transformer
from shapely import geometry, wkb
from shapely.ops import snap, split

from db_connection import get_db_connection
from logger import get_logger

# ---------------------------------
# Constantes e configuração
# ---------------------------------
SEG_LEN_M = 100.0         # comprimento alvo de cada segmento (m)
```

```
41 SNAP_TOL = 0.01           # tolerância para snap (m, no sistema projetado)
42 POINT_BUF = 0.001         # pequeno buffer p/ evitar degenerescência na divisão
43 FT_PER_M = 3.281          # conversão aproximada m -> ft
44 HTTP_TIMEOUT_S = 5        # timeout para requisições HTTP
45
46 logger = get_logger("RegisterSegment")
47
48
49 def _utm_proj_from_lonlat(lon: float, lat: float) -> str:
50     """Retorna string proj para a zona UTM correspondente ao lon/lat."""
51     zone = math.floor((lon + 180) / 6) + 1
52     hemi = "south" if lat < 0 else "north"
53     return f"+proj=utm +zone={zone} +{hemi} +ellps=WGS84"
54
55
56 def _get_transformers(lon: float, lat: float) -> Tuple[Transformer, Transformer]:
57     """Cria transformers WGS84 <-> UTM na zona apropriada ao centróide da rota."""
58     wgs84 = CRS("EPSG:4326")
59     utm_crs = CRS(_utm_proj_from_lonlat(lon, lat))
60     to_utm = Transformer.from_crs(wgs84, utm_crs, always_xy=True)
61     to_wgs = Transformer.from_crs(utm_crs, wgs84, always_xy=True)
62     return to_utm, to_wgs
63
64
65 def _load_line_from_wkb(geom_wkb: bytes) -> geometry.LineString:
66     """Carrega LineString a partir de WKB (aceita bytes ou hex)."""
67     try:
68         return wkb.loads(bytes(geom_wkb))
69     except Exception:
70         hex_wkb = binascii.hexlify(geom_wkb).decode("utf-8")
71         return wkb.loads(hex_wkb, hex=True)
72
73
74 def _split_line_every(line_utm: geometry.LineString, step_m: float) -> List[
    geometry.LineString]:
75     """Divide uma LineString em segmentos aproximados de 'step_m' metros (no CRS
    projetado)."""
76     total_len = line_utm.length
77     if total_len <= 0:
78         return []
79
```

```
    # pontos de divisão ao longo da linha
    n = max(1, math.ceil(total_len / step_m))
    dists = [min(step_m * i, total_len) for i in range(1, n)]
    pts = [line_utm.interpolate(d).buffer(POINT_BUF).centroid for d in dists]

    splitter = geometry.MultiPoint(pts).union(line_utm.boundary)

    # ajusta nós e divide
    snapped = snap(line_utm, splitter, SNAP_TOL)
    parts = split(snapped, splitter)

    # remove segmentos degenerate e ordena na direção da linha original
    segs = [s for s in parts.geoms if s.length > 0]
    segs.sort(key=lambda s: line_utm.project(geometry.Point(s.coords[0])))
    return segs


def _resolve_api_key(explicit_key: Optional[str]) -> Optional[str]:
    """
    Prioriza a chave passada como parâmetro; se ausente, tenta a env var
    GOOGLE_ELEVATION_API_KEY. Retorna None se nenhuma existir.
    """
    if explicit_key:
        return explicit_key.strip()
    env = os.getenv("GOOGLE_ELEVATION_API_KEY", "").strip()
    return env or None


def _get_elevation(lat: float, lon: float, api_key: Optional[str]) -> Optional[
    float]:
    """
    Consulta a Google Elevation API (se houver chave). Retorna elevação em metros
    ou None se indisponível (erro/chave ausente).
    """
    if not api_key:
        return None
    url = (
        "https://maps.googleapis.com/maps/api/elevation/json"
        f"?locations={lat},{lon}&key={api_key}"
    )
    try:
```

```
        resp = requests.get(url, timeout=HTTP_TIMEOUT_S)
        resp.raise_for_status()
        data = resp.json()
        if data.get("status") == "OK" and data.get("results"):
            return float(data["results"][0]["elevation"])
        logger.warning("Elevation API status != OK: %s", data.get("status"))
    except Exception as e:
        logger.error("Erro na Elevation API: %s", e)
    return None


def process_request(
    request_id: int,
    tol_percent: float = 0.10,
    google_api_key: Optional[str] = None,
) -> None:
    """
    Gera segmentos (~100 m) para a requisição e registra janelas temporais.

    Lógica:
      1) Carrega waypoints e janela [start_time, end_time] de `requisicoes`.
      2) Projeta para UTM, mede comprimento e divide em ~100 m.
      3) Para cada segmento, calcula janela proporcional e tolerâncias:
         - start_time_min/max e end_time_min/max usando `tol_percent`.
      4) (Opcional) Consulta elevação (pés) via Google Elevation (média entre
         início/fim do segmento). Se indisponível, usa 0 ft e registra no log.
      5) Insere linhas em `segmentos`.

    Parâmetros:
        request_id: ID em `requisicoes`.
        tol_percent: fração da duração do segmento usada como tolerância (0–1).
        google_api_key: **NÃO** coloque chave fixa aqui. Passe-a em runtime
                        ou configure `GOOGLE_ELEVATION_API_KEY`.

    Efeitos colaterais:
        INSERT em `segmentos` (request_id, segmento, alt, start/end *_min/_max)

    Observações:
        - Unidades de tempo preservam os tipos do banco (timestamp/timestamptz).
        - Altitude gravada em pés (ft). Quando não disponível, grava 0.0.
    """
```

```
161     api_key = _resolve_api_key(google_api_key)
162 
163     conn = None
164     cursor = None
165     try:
166         # ---------------- Banco: carrega geometria e janela da requisição
167         conn = get_db_connection()
168         cursor = conn.cursor()
169         cursor.execute(
170             """
171             SELECT ST_AsBinary(waypoints), start_time, end_time
172             FROM requisicoes
173             WHERE request_id = %s;
174             """,
175             (request_id,),
176         )
177         row = cursor.fetchone()
178         if not row:
179             logger.warning("request_id %s não encontrado em requisicoes.",
    request_id)
180             return
181 
182         geom_wkb, start_time, end_time = row
183         original_line = _load_line_from_wkb(geom_wkb)
184 
185         # ---------------- CRS / projeção
186         centroid = original_line.centroid
187         to_utm, to_wgs = _get_transformers(centroid.x, centroid.y)
188 
189         # linha no CRS projetado (m) para medir/computar
190         utm_line = geometry.LineString([to_utm.transform(x, y) for x, y in
    original_line.coords])
191         total_length = float(utm_line.length)
192         total_duration = end_time - start_time  # timedelta
193 
194         if total_length <= 0 or total_duration.total_seconds() <= 0:
195             logger.error("Requisição %s com geometria/duração inválida.",
    request_id)
196             return
197 
198         # ---------------- Divide a linha
```

```
199         segments = _split_line_every(utm_line, SEG_LEN_M)
200         if not segments:
201             logger.warning("Sem segmentos gerados (request_id=%s).", request_id)
202             return
203 
204         # ---------------- Para cada segmento, calcula janela e altitude
205         cumulative = timedelta(seconds=0)
206         inserted = 0
207 
208         for seg in segments:
209             seg_len = float(seg.length)
210             seg_frac = seg_len / total_length
211             seg_dur = total_duration * seg_frac
212 
213             seg_start = start_time + cumulative
214             seg_end = seg_start + seg_dur
215             cumulative += seg_dur
216 
217             tol = seg_dur * tol_percent
218             seg_start_min = max(start_time, seg_start - tol)
219             seg_start_max = seg_start + tol
220             seg_end_min = seg_end - tol
221             seg_end_max = min(end_time, seg_end + tol)
222 
223             # segmento para WGS84 (gravação no banco)
224             wgs_segment = geometry.LineString([to_wgs.transform(x, y) for x, y in
    seg.coords])
225             segment_wkt = wgs_segment.wkt
226 
227             # elevação (opcional) nos extremos (média, convertida para pés)
228             (lon0, lat0) = wgs_segment.coords[0]
229             (lon1, lat1) = wgs_segment.coords[-1]
230             elev0_m = _get_elevation(lat0, lon0, api_key)
231             elev1_m = _get_elevation(lat1, lon1, api_key)
232 
233             if elev0_m is None or elev1_m is None:
234                 alt_ft = 0.0
235                 logger.info(
236                     "Elevação indisponível (req=%s). Gravando 0 ft para o segmento.
    ",
237                     request_id,
```

```
                )
            else:
                alt_ft = ((float(elev0_m) + float(elev1_m)) / 2.0) * FT_PER_M

            # INSERT
            cursor.execute(
                """
                INSERT INTO segmentos (
                    request_id,
                    segmento,
                    alt,
                    start_time_min,
                    start_time_max,
                    end_time_min,
                    end_time_max
                )
                VALUES (%s, ST_GeomFromText(%s, 4326), %s, %s, %s, %s, %s);
                """,
                (
                    request_id,
                    segment_wkt,
                    alt_ft,
                    seg_start_min,
                    seg_start_max,
                    seg_end_min,
                    seg_end_max,
                ),
            )
            inserted += 1

        conn.commit()
        logger.info("%s segmentos inseridos para request_id=%s.", inserted,
    request_id)

    except Exception as e:
        if conn:
            conn.rollback()
        logger.error("Erro em process_request(%s): %s", request_id, e)
    finally:
        if cursor:
            cursor.close()
```

```
278         if conn:
279             conn.close()
```

Código A.7 – Divisão da rota em segmentos, cálculo de janelas e (opcional) consulta de elevação

A.8 Conversão de arquivos GeoJSON

```
"""
geojson_conversion.py
---------------------

Módulo utilitário para conversão de arquivos GeoJSON em geometrias WKT com SRID
    =4326.

Funções principais:
    - trajetoria(arq): lê um arquivo GeoJSON contendo uma LineString e retorna
      (start_point, end_point, waypoints) prontos para inserção em banco.
    - area(arq): lê um arquivo GeoJSON contendo um Polygon e retorna o WKT
      equivalente (SRID=4326).

Dependências:
    - geopandas (gpd)
"""

from __future__ import annotations

import os
import geopandas as gpd

# Diretórios base (compatíveis com Windows/Linux)
ROUTES_DIR = os.path.join(".", "geojsons", "routes")
PVRS_DIR = os.path.join(".", "geojsons", "pvrs")


def trajetoria(nome_arquivo: str):
    """
    Lê um GeoJSON de rota (LineString) e retorna pontos e linha em formato WKT.

    Args:
        nome_arquivo (str): Nome do arquivo (sem extensão) dentro de geojsons/
    routes.
```

```
33
34      Returns:
35          tuple[str, str, str]:
36              (start_point, end_point, waypoints)  todos com SRID=4326.
37      """
38      caminho = os.path.join(ROUTES_DIR, f"{nome_arquivo}.geojson")
39      gdf = gpd.read_file(caminho)
40
41      if gdf.empty:
42          raise ValueError(f"O arquivo {caminho} está vazio ou inválido.")
43      if gdf.geometry.iloc[0].geom_type != "LineString":
44          raise ValueError("O arquivo deve conter uma geometria do tipo LineString.")
45
46      linha = gdf.geometry.iloc[0]
47      coords = list(linha.coords)
48
49      # primeiro e último ponto
50      first_point, last_point = coords[0], coords[-1]
51      start_point = f"SRID=4326;POINT({first_point[0]} {first_point[1]})"
52      end_point = f"SRID=4326;POINT({last_point[0]} {last_point[1]})"
53
54      # todos os waypoints (LINESTRING)
55      waypoints = (
56          f"SRID=4326;LINESTRING({', '.join([f'{x} {y}' for x, y in coords])})"
57      )
58
59      return start_point, end_point, waypoints
60
61
62  def area(nome_arquivo: str):
63      """
64      Lê um GeoJSON contendo um Polygon e retorna sua geometria WKT (SRID=4326).
65
66      Args:
67          nome_arquivo (str): Nome do arquivo (sem extensão) dentro de geojsons/pvrs.
68
69      Returns:
70          str: Polígono formatado como WKT com SRID=4326.
71      """
72      caminho = os.path.join(PVRS_DIR, f"{nome_arquivo}.geojson")
73      gdf = gpd.read_file(caminho)
```

```
74
75      if gdf.empty:
76          raise ValueError(f"O arquivo {caminho} está vazio ou inválido.")
77      geom = gdf.geometry.iloc[0]
78
79      if geom.geom_type != "Polygon":
80          raise ValueError("O arquivo GeoJSON deve conter uma geometria do tipo
        Polygon.")
81
82      polygon_coords = list(geom.exterior.coords)
83      coords_str = ", ".join([f"{x} {y}" for x, y in polygon_coords])
84      polygon_wkt = f"SRID=4326;POLYGON(({coords_str}))"
85
86      return polygon_wkt
```

Código A.8 – Conversão de arquivos GeoJSON em geometrias WKT (rotas e áreas PVR)

FOLHA DE REGISTRO DO DOCUMENTO

1. CLASSIFICAÇÃO/TIPO	2. DATA	3. DOCUMENTO Nº	4. Nº DE PÁGINAS
TC	13 de novembro de 2025	DCTA/ITA/TC-064/2025	135

5. TÍTULO E SUBTÍTULO:

Modelagem de arquitetura de gerenciamento do tráfego não tripulado em espaço aéreo abaixo de 400 pés

6. AUTORA(ES):

Bruna Belarmino Silva

7. INSTITUIÇÃO(ÕES)/ÓRGÃO(S) INTERNO(S)/DIVISÃO(ÕES):

Instituto Tecnológico de Aeronáutica – ITA

8. PALAVRAS-CHAVE SUGERIDAS PELA AUTORA:

UTM; Drones; Requisição de voo; Resolução de conflitos; Verificação estática; Modelagem de arquitetura; Gestão de tráfego não tripulado.

9. PALAVRAS-CHAVE RESULTANTES DE INDEXAÇÃO:

Aeronave não-tripulada; Planejamentp de trajetória; Rotas; Espaço aéreo; Segurança nacional; Brasil; Gerenciamento de riscos; Aeronaves teleguiadas; Engenharia aeronáutica.

10. APRESENTAÇÃO: (**X**) **Nacional** ( ) **Internacional**

ITA, São José dos Campos. Curso de Graduação em Engenharia Aeronáutica. Orientador: Maj. Av. Lucas Oliveira Barbacovi. Coorientador: Prof. Dr. Christopher Schneider Cerqueira. Publicado em 2025.

11. RESUMO:

O presente trabalho propõe uma arquitetura distribuída para o gerenciamento de tráfego aéreo de drones, com foco em operações abaixo de 400 pés (aproximadamente 120 metros) em espaço aéreo de baixa altitude. A motivação central é atender à crescente demanda por voos não tripulados, garantindo segurança, eficiência e integração com o ecossistema aeronáutico nacional.

A metodologia baseia-se na modelagem de um sistema modular composto por operadores, provedores de serviço e autoridade aeronáutica simulada, que interagem por meio do protocolo MQTT e banco de dados PostgreSQL/-PostGIS. O sistema realiza verificações automáticas e hierárquicas, compreendendo análises estáticas, resolução de conflito espacial e temporal, com aplicação de *buffers*, segmentação de rotas 4D e simulação de PVR.

Foram conduzidos dez cenários de simulação para avaliar o desempenho da arquitetura, abrangendo casos de rotas fora do limite de controle, áreas restritas, reservas ativas e múltiplas missões simultâneas. Os resultados confirmaram que o sistema é capaz de detectar conflitos, ajustar rotas e reagendar janelas temporais automaticamente, reproduzindo a lógica de um ambiente UTM federado e escalável.

Conclui-se que a arquitetura proposta representa um passo significativo rumo à implementação de um UTM nacional compatível com as diretrizes do DECEA e da FAA, promovendo um modelo de cooperação, interoperabilidade e segurança operacional para integração de drones no espaço aéreo brasileiro.

12. GRAU DE SIGILO:

(**X**) **OSTENSIVO** ( ) **RESERVADO** ( ) **SECRETO**