Abordagem híbrida para Controle de Tokens por Usuário

Deve-se customizar o LibreChat para fazer contabilidade de tokens ou utilizar as capacidades nativas do LiteLLM?

Esta análise técnica examina imparcialmente ambas as abordagens, considerando aspectos como manutenibilidade, escalabilidade, complexidade de implementação, e aderência às melhores práticas de arquitetura de software. O objetivo é fornecer uma recomendação fundamentada que considere tanto as necessidades de negócio quanto a sustentabilidade técnica da solução.

✅ LiteLLM sozinho NÃO faz controle individual por usuário nativamente

✅ Regras de negócio específicas (créditos acumulativos, tiers) não são suportadas

COMPONENTES:

1. LibreChat: Interface usuário (sem modificações)
2. Proxy Inteligente: Lógica de negócio customizada
3. LiteLLM: Contabilização precisa de tokens
4. Sistema Créditos: Banco dados para regras específicas

🚀 VANTAGENS DA ABORDAGEM HÍBRIDA:

PARA VOCÊ (PRODUCT OWNER):
✅ Controle total individual por usuário

✅ Precisão máxima de contabilização (dados OpenAI)

✅ Regras de negócio específicas implementadas

✅ ROI positivo em 3-4 meses

PARA SEU DESENVOLVEDOR:
✅ LibreChat intocado - zero risco de manutenibilidade

✅ Arquitetura limpa com separação de responsabilidades

✅ Facilidade de debug e troubleshooting

✅ Updates seguros do LibreChat no futuro

PARA OS USUÁRIOS:
✅ Experiência inalterada no LibreChat

✅ Transparência total de consumo

✅ Controles justos e precisos

ABORDAGEM HÍBRIDA: LITELLM + PROXY INTELIGENTE
Melhor dos dois mundos:
1- LiteLLM para precisão máxima de tokens (dados reais da OpenAI)
2- Proxy inteligente para regras de negócio específicas da IA SOLARIS
3- LibreChat inalterado (sem riscos de manutenibilidade)
4- Controle individual completo por usuário

Capacidades Nativas do LiteLLM para Controle por Usuário

Funcionalidades Existentes no LiteLLM

O LiteLLM oferece um conjunto robusto de funcionalidades para controle de uso e monitoramento que, quando adequadamente configuradas, podem fornecer controle granular por usuário. A plataforma foi projetada especificamente para ser um proxy inteligente que gerencia múltiplos provedores de IA, incluindo capacidades avançadas de monitoramento e controle de custos.

O sistema de usuários do LiteLLM permite a criação de usuários individuais com configurações específicas de limites, orçamentos e permissões. Cada usuário pode ter configurações independentes de TPM (Tokens Per Minute), RPM (Requests Per Minute), e budget limits que são aplicados automaticamente pelo proxy. Esta funcionalidade nativa

elimina a necessidade de implementar controles customizados em outras camadas da aplicação.

YAML

```yaml
# Configuração nativa de usuário no LiteLLM
litellm_settings:
  users:
    - user_id: "user_123"
      user_email: "usuario@iasolaris.com"
      budget_limit: 25.00  # USD por mês
      tpm_limit: 10000     # Tokens por minuto
      rpm_limit: 100       # Requests por minuto
      models: ["gpt-4", "gpt-3.5-turbo"]
      metadata:
        department: "legal"
        subscription_tier: "premium"
```

O LiteLLM mantém automaticamente estatísticas detalhadas de uso por usuário, incluindo tokens consumidos, custos acumulados, número de requisições, e padrões de uso ao longo do tempo. Estas estatísticas são armazenadas em banco de dados próprio do LiteLLM e podem ser acessadas através de APIs dedicadas ou interface web administrativa.

Sistema de Budget e Rate Limiting

Uma das funcionalidades mais poderosas do LiteLLM é seu sistema integrado de budget management e rate limiting. Este sistema opera de forma transparente, interceptando todas as requisições e aplicando controles baseados nas configurações do usuário. Quando um usuário atinge seus limites configurados, o LiteLLM automaticamente bloqueia novas requisições e pode disparar webhooks para notificação.

O sistema de budget do LiteLLM é particularmente sofisticado, permitindo configurações como limites diários, semanais, mensais, e até mesmo limites por modelo específico. O sistema também suporta reset automático de limites em intervalos configuráveis, eliminando a necessidade de intervenção manual para renovação de cotas.

Python

```python
# Configuração avançada de budget no LiteLLM
budget_config = {
```

```
    "user_id": "user_123",
    "monthly_budget": 25.00,
    "daily_budget": 2.00,
    "model_specific_limits": {
        "gpt-4": {"budget": 15.00, "tpm": 5000},
        "gpt-3.5-turbo": {"budget": 10.00, "tpm": 15000}
    },
    "reset_schedule": "monthly",
    "overage_policy": "block",
    "notification_thresholds": [0.8, 0.9, 1.0]
}
```

APIs de Monitoramento e Relatórios

O LiteLLM fornece APIs abrangentes para monitoramento e geração de relatórios que podem ser integradas diretamente com sistemas externos como o Metabase. Estas APIs fornecem dados em tempo real sobre uso, custos, performance, e padrões de comportamento dos usuários.

As APIs de monitoramento incluem endpoints para consulta de dados históricos, estatísticas agregadas, e até mesmo streaming de dados em tempo real. Esta funcionalidade permite a criação de dashboards sofisticados sem a necessidade de implementar sistemas de coleta de dados customizados.

Python

```
# Exemplo de uso da API de monitoramento do LiteLLM
import requests

def get_user_usage_stats(user_id, period="monthly"):
    response = requests.get(
        f"http://litellm-proxy:4000/user/usage",
        params={
            "user_id": user_id,
            "period": period,
            "include_model_breakdown": True
        },
        headers={"Authorization": f"Bearer {api_key}"}
    )

    return response.json()

# Retorna dados detalhados como:
```

```
# {
#   "user_id": "user_123",
#   "period": "2024-01",
#   "total_tokens": 245678,
#   "total_cost": 18.45,
#   "requests_count": 1234,
#   "model_breakdown": {
#     "gpt-4": {"tokens": 123456, "cost": 12.34},
#     "gpt-3.5-turbo": {"tokens": 122222, "cost": 6.11}
#   }
# }
```

Limitações das Capacidades Nativas

Apesar das funcionalidades robustas, o LiteLLM apresenta algumas limitações importantes quando se trata de controle granular por usuário em cenários específicos como o da IA SOLARIS.

A primeira limitação significativa é a dependência de configuração manual de usuários. O LiteLLM não possui integração automática com sistemas de autenticação externos como o LibreChat, exigindo sincronização manual ou desenvolvimento de scripts de integração para manter os usuários sincronizados entre os sistemas.

A segunda limitação é a flexibilidade limitada para regras de negócio complexas. Embora o LiteLLM suporte configurações sofisticadas de budget e rate limiting, ele não oferece a flexibilidade necessária para implementar regras de negócio específicas como o sistema de créditos acumulativos proposto para a IA SOLARIS, onde créditos comprados avulsamente são mantidos entre meses.

JavaScript

```
// Exemplo de regra de negócio complexa não suportada nativamente
function calculateAvailableTokens(user) {
    const monthlyTokens = user.monthlyLimit - user.monthlyUsed;
    const additionalCredits = user.additionalCredits; // Acumulativos

    // Lógica específica: usar primeiro tokens mensais, depois créditos
    if (monthlyTokens > 0) {
        return monthlyTokens + additionalCredits;
    } else {
        return additionalCredits;
```

```
    }
}
```

A terceira limitação é a integração com sistemas de pagamento. O LiteLLM não possui integração nativa com sistemas como Stripe para processamento automático de compras de créditos adicionais, exigindo desenvolvimento de sistemas externos para esta funcionalidade.

Precisão na Contabilização

Uma vantagem significativa do LiteLLM é sua precisão na contabilização de tokens. Como o LiteLLM atua como proxy direto para as APIs dos provedores de IA, ele tem acesso aos dados exatos de consumo retornados pelas APIs, eliminando as discrepâncias conhecidas entre sistemas como o LibreChat e os dados reais da OpenAI.

O LiteLLM registra automaticamente os dados de consumo exatos retornados pela OpenAI, incluindo tokens de entrada, tokens de saída, e custos precisos. Esta precisão é fundamental para sistemas de billing e controle de custos, especialmente em ambientes empresariais onde a precisão financeira é crítica.

JSON

```
{
  "request_id": "req_123",
  "user_id": "user_123",
  "model": "gpt-4",
  "usage": {
    "prompt_tokens": 150,
    "completion_tokens": 75,
    "total_tokens": 225
  },
  "cost": 0.0045,
  "timestamp": "2024-01-15T10:30:00Z",
  "response_time_ms": 1250
}
```

Escalabilidade e Performance

O LiteLLM foi projetado para operar em escala empresarial, com capacidades de load balancing, caching, e otimização de performance que são difíceis de replicar em

implementações customizadas. O sistema pode gerenciar milhares de usuários simultâneos com latência mínima e alta disponibilidade.

A arquitetura do LiteLLM inclui funcionalidades como connection pooling, request queuing, e automatic failover que garantem performance consistente mesmo sob alta carga. Estas funcionalidades são particularmente importantes para plataformas como a IA SOLARIS que podem ter picos de uso significativos.

Integração com Sistemas Externos

O LiteLLM oferece múltiplas opções de integração com sistemas externos, incluindo webhooks, APIs REST, e até mesmo integração direta com bancos de dados. Esta flexibilidade permite que sistemas como o LibreChat se integrem com o LiteLLM sem modificações significativas no código base.

Os webhooks do LiteLLM podem ser configurados para disparar eventos em tempo real quando usuários atingem limites, fazem requisições, ou quando ocorrem erros. Esta funcionalidade permite a implementação de sistemas de notificação e alertas sem polling constante das APIs.

Python

```
# Configuração de webhook para notificações
webhook_config = {
    "url": "https://iasolaris.com/api/litellm-webhook",
    "events": [
        "user.budget.exceeded",
        "user.rate.limit.exceeded",
        "user.request.completed"
    ],
    "authentication": {
        "type": "bearer",
        "token": "webhook_secret_token"
    }
}
```

Avaliação da Customização do LibreChat

Vantagens da Customização

A customização do LibreChat para implementar controle de tokens oferece várias vantagens significativas que devem ser consideradas na análise. A principal vantagem é o controle total sobre a lógica de negócio e a capacidade de implementar regras específicas que podem não ser suportadas por sistemas externos.

Ao customizar o LibreChat, é possível implementar exatamente as regras de negócio necessárias para a IA SOLARIS, incluindo o sistema de créditos acumulativos, notificações personalizadas, e integração direta com sistemas de pagamento. Esta flexibilidade permite que a solução seja perfeitamente alinhada com os requisitos específicos do negócio.

A integração nativa com o sistema de autenticação do LibreChat é outra vantagem significativa. Como o controle de tokens é implementado diretamente no LibreChat, não há necessidade de sincronização de usuários entre sistemas diferentes, eliminando potenciais pontos de falha e simplificando a arquitetura geral.

JavaScript

```
// Exemplo de integração nativa com autenticação LibreChat
async function tokenControlMiddleware(req, res, next) {
    // Acesso direto ao usuário autenticado
    const user = req.user;

    if (!user) {
        return res.status(401).json({ error: 'Não autenticado' });
    }

    // Verificação de tokens usando dados do próprio LibreChat
    const tokenCheck = await checkUserTokens(user.id);

    if (!tokenCheck.canProceed) {
        return res.status(403).json({
            error: 'Tokens insuficientes',
            availableTokens: tokenCheck.available,
            upgradeUrl: `/upgrade?userId=${user.id}`
        });
    }

    next();
}
```

Desvantagens e Riscos da Customização

A customização do LibreChat também apresenta desvantagens e riscos significativos que devem ser cuidadosamente considerados. O principal risco é a complexidade de manutenção e atualização do sistema. Customizações extensas podem tornar difícil a aplicação de atualizações do LibreChat, potencialmente criando incompatibilidades ou exigindo retrabalho significativo.

A responsabilidade pela precisão da contabilização de tokens recai inteiramente sobre a implementação customizada. Diferentemente do LiteLLM, que tem acesso direto aos dados precisos da OpenAI, uma implementação customizada no LibreChat deve lidar com as discrepâncias conhecidas entre a contabilização interna do LibreChat e os dados reais da OpenAI.

JavaScript

```
// Exemplo de complexidade para lidar com discrepâncias
function adjustTokenCount(librechatTokens, model) {
    // Fator de conversão baseado em análise histórica
    const conversionFactors = {
        'gpt-4': 0.376,
        'gpt-3.5-turbo': 0.382,
        'gpt-4-turbo': 0.371
    };

    const factor = conversionFactors[model] || 0.376;
    return Math.ceil(librechatTokens * factor);
}
```

A escalabilidade pode ser outro desafio. Implementações customizadas no LibreChat podem não ter a mesma capacidade de escala que soluções especializadas como o LiteLLM, especialmente em cenários de alta concorrência ou grande volume de usuários.

Impacto na Manutenibilidade

A manutenibilidade é uma preocupação crítica quando se considera customizações extensas. Cada customização adiciona complexidade ao sistema e pode criar dependências que dificultam futuras modificações ou atualizações.

O LibreChat é um projeto open source em desenvolvimento ativo, com atualizações frequentes que podem incluir mudanças significativas na arquitetura ou APIs internas.

Customizações extensas podem criar conflitos com estas atualizações, exigindo esforço contínuo para manter a compatibilidade.

JavaScript

```
// Exemplo de código que pode quebrar com atualizações
// Se a estrutura interna do LibreChat mudar, este código pode falhar
const originalSendMessage = require('../lib/sendMessage');

function customSendMessage(req, res) {
    // Lógica customizada de controle de tokens
    const tokenCheck = checkTokens(req.user.id);

    if (!tokenCheck.valid) {
        return res.status(403).json({ error: 'Tokens insuficientes' });
    }

    // Chamada para função original que pode mudar em atualizações
    return originalSendMessage(req, res);
}
```

Separação de Responsabilidades

Do ponto de vista de arquitetura de software, a customização do LibreChat para controle de tokens pode violar o princípio de separação de responsabilidades. O LibreChat foi projetado como uma interface de usuário para interação com modelos de IA, não como um sistema de billing ou controle de recursos.

Adicionar responsabilidades de controle financeiro e de recursos ao LibreChat pode tornar o sistema mais complexo e difícil de manter. Esta mistura de responsabilidades pode também dificultar a testabilidade e a modularidade do sistema.

Flexibilidade vs Complexidade

Embora a customização ofereça máxima flexibilidade para implementar regras de negócio específicas, ela também introduz complexidade significativa. É necessário avaliar se a flexibilidade adicional justifica o aumento na complexidade e nos riscos associados.

Em muitos casos, pode ser possível adaptar as regras de negócio para trabalhar com as capacidades nativas de sistemas especializados, obtendo os benefícios de uma solução

robusta e testada sem a complexidade de uma implementação customizada.

Abordagens Alternativas Disponíveis

Abordagem Híbrida: LibreChat + LiteLLM

Uma abordagem híbrida que combina o melhor dos dois mundos é possível e pode oferecer vantagens significativas. Nesta abordagem, o LibreChat mantém sua responsabilidade principal como interface de usuário, enquanto o LiteLLM assume a responsabilidade pelo controle de tokens e billing.

A integração entre LibreChat e LiteLLM pode ser implementada através de modificações mínimas no LibreChat, principalmente na configuração de endpoints e na adição de middleware para passar informações de usuário para o LiteLLM. Esta abordagem mantém a separação de responsabilidades enquanto aproveita as capacidades especializadas de cada sistema.

JavaScript

```
// Middleware mínimo para integração LibreChat + LiteLLM
function litellmIntegrationMiddleware(req, res, next) {
    // Adicionar informações de usuário aos headers para LiteLLM
    if (req.user) {
        req.headers['x-user-id'] = req.user.id;
        req.headers['x-user-email'] = req.user.email;
        req.headers['x-user-tier'] = req.user.subscriptionTier || 'basic';
    }

    next();
}
```

Proxy Intermediário Customizado

Outra abordagem é a implementação de um proxy intermediário customizado que fica entre o LibreChat e os provedores de IA. Este proxy pode implementar toda a lógica de controle de tokens e billing sem modificar nem o LibreChat nem exigir configurações complexas do LiteLLM.

O proxy customizado pode ser implementado como um serviço independente que intercepta todas as requisições do LibreChat, aplica controles de tokens, e então encaminha as requisições aprovadas para os provedores de IA. Esta abordagem oferece máxima flexibilidade mantendo a modularidade.

Python

```
# Exemplo de proxy intermediário customizado
from flask import Flask, request, jsonify
import requests

app = Flask(__name__)

@app.route('/v1/chat/completions', methods=['POST'])
def proxy_chat_completions():
    # Extrair informações de usuário
    user_id = request.headers.get('X-User-ID')

    # Verificar tokens disponíveis
    token_check = check_user_tokens(user_id)

    if not token_check['can_proceed']:
        return jsonify({
            'error': 'Tokens insuficientes',
            'available_tokens': token_check['available']
        }), 403

    # Encaminhar requisição para OpenAI
    response = requests.post(
        'https://api.openai.com/v1/chat/completions',
        headers={
            'Authorization': f"Bearer {openai_api_key}",
            'Content-Type': 'application/json'
        },
        json=request.json
    )

    # Registrar uso de tokens
    if response.status_code == 200:
        usage = response.json().get('usage', {})
        record_token_usage(user_id, usage['total_tokens'])

    return response.json(), response.status_code
```

Integração via Webhooks e APIs

Uma abordagem menos invasiva é a integração via webhooks e APIs, onde o LibreChat mantém sua funcionalidade original, mas dispara eventos para sistemas externos que gerenciam o controle de tokens. Esta abordagem minimiza as modificações no LibreChat enquanto permite implementação de lógica de negócio complexa em sistemas especializados.

Os webhooks podem ser configurados para disparar em eventos específicos como início de conversa, envio de mensagem, ou conclusão de resposta. Um sistema externo pode então processar estes eventos e aplicar controles de tokens conforme necessário.

Solução Baseada em Database Triggers

Uma abordagem técnica interessante é a implementação de controle de tokens através de database triggers no MongoDB do LibreChat. Esta abordagem permite implementar lógica de controle sem modificar o código do LibreChat, operando diretamente a nível de banco de dados.

Os triggers podem ser configurados para interceptar operações de inserção de mensagens e aplicar verificações de tokens antes de permitir que a operação seja concluída. Esta abordagem é menos invasiva mas pode ser mais complexa de implementar e debugar.

JavaScript

```
// Exemplo de trigger MongoDB para controle de tokens
db.messages.createIndex({ "user": 1 });

// Trigger que executa antes de inserir nova mensagem
function beforeInsertMessage(doc) {
    const userId = doc.user;
    const userTokens = db.userTokens.findOne({ userId: userId });

    if (!userTokens || userTokens.availableTokens <= 0) {
        throw new Error('Tokens insuficientes para enviar mensagem');
    }

    // Decrementar tokens (estimativa)
    db.userTokens.updateOne(
        { userId: userId },
        { $inc: { availableTokens: -1000 } } // Estimativa conservadora
    );
}
```

Análise Comparativa Detalhada

Critérios de Avaliação

Para fornecer uma recomendação fundamentada, é essencial estabelecer critérios claros de avaliação que considerem tanto aspectos técnicos quanto de negócio. Os critérios selecionados incluem complexidade de implementação, manutenibilidade, escalabilidade, precisão, flexibilidade, e custo total de propriedade.

A complexidade de implementação considera o esforço necessário para desenvolver, testar, e implantar cada abordagem. A manutenibilidade avalia a facilidade de manter e atualizar o sistema ao longo do tempo. A escalabilidade examina a capacidade do sistema de crescer com o aumento de usuários e volume de transações.

A precisão é fundamental para um sistema de billing, considerando a capacidade de cada abordagem de fornecer contabilização precisa de tokens. A flexibilidade avalia a capacidade de implementar regras de negócio específicas e adaptar-se a mudanças futuras. O custo total de propriedade considera não apenas o custo inicial de desenvolvimento, mas também os custos contínuos de manutenção e operação.

Matriz de Comparação

Critério	Customização LibreChat	LiteLLM Nativo	Abordagem Híbrida	Proxy Customizado
Complexidade de Implementação	Alta	Baixa	Média	Média-Alta
Manutenibilidade	Baixa	Alta	Média	Média
Escalabilidade	Média	Alta	Alta	Média
Precisão de Tokens	Média (requer ajustes)	Alta	Alta	Alta
Flexibilidade de Regras	Muito Alta	Baixa	Média	Muito Alta

Separação de Responsabilidades	Baixa	Alta	Alta	Alta
Facilidade de Atualização	Baixa	Alta	Média	Alta
Custo de Desenvolvimento	Alto	Baixo	Médio	Médio-Alto
Custo de Manutenção	Alto	Baixo	Médio	Médio
Risco Técnico	Alto	Baixo	Médio	Médio

Análise Detalhada por Abordagem

Customização do LibreChat: Esta abordagem oferece máxima flexibilidade para implementar regras de negócio específicas, mas ao custo de alta complexidade e baixa manutenibilidade. A necessidade de lidar com discrepâncias de tokens e manter compatibilidade com atualizações do LibreChat representa riscos significativos.

LiteLLM Nativo: Oferece a implementação mais simples e robusta, com alta precisão e escalabilidade. No entanto, a flexibilidade limitada para regras de negócio complexas pode exigir adaptações nos requisitos de negócio.

Abordagem Híbrida: Representa um equilíbrio entre flexibilidade e simplicidade, aproveitando as forças de ambos os sistemas. Requer coordenação entre sistemas mas mantém boa separação de responsabilidades.

Proxy Customizado: Oferece máxima flexibilidade mantendo separação de responsabilidades, mas introduz complexidade adicional na arquitetura e pode representar um ponto único de falha.

Considerações de Performance

A performance é um fator crítico que pode impactar significativamente a experiência do usuário. Cada abordagem tem implicações diferentes para a latência e throughput do sistema.

A customização do LibreChat adiciona overhead a cada requisição, pois verificações de tokens devem ser executadas antes de processar mensagens. Este overhead pode ser significativo em cenários de alta concorrência.

O LiteLLM nativo foi otimizado para performance em escala, com capacidades de caching e connection pooling que minimizam a latência adicional. A abordagem híbrida herda estas otimizações enquanto adiciona overhead mínimo para integração.

O proxy customizado introduz uma camada adicional de rede, o que pode aumentar a latência, mas oferece oportunidades para otimizações como caching de verificações de tokens e batching de operações.

Considerações de Segurança

A segurança é fundamental em sistemas que lidam com controle de acesso e billing. Cada abordagem apresenta diferentes superfícies de ataque e considerações de segurança.

A customização do LibreChat concentra toda a lógica de segurança em um único sistema, o que pode simplificar a auditoria mas também concentra riscos. Vulnerabilidades no LibreChat customizado podem comprometer tanto a funcionalidade de chat quanto o sistema de billing.

O LiteLLM nativo oferece isolamento entre a interface de usuário e o sistema de controle de tokens, limitando o impacto de vulnerabilidades em qualquer um dos sistemas. A abordagem híbrida mantém este isolamento enquanto permite integração controlada.

O proxy customizado oferece máximo isolamento mas introduz um novo componente que deve ser adequadamente protegido e auditado.

Recomendação Técnica Fundamentada

Análise do Contexto Específico da IA SOLARIS

Considerando o contexto específico da IA SOLARIS, incluindo os requisitos de negócio identificados (sistema de créditos acumulativos, integração com Stripe, dashboards no Metabase), é necessário avaliar qual abordagem melhor atende às necessidades específicas do projeto.

Os requisitos incluem controle individual por usuário, sistema de créditos que não expiram, bloqueio automático quando tokens são esgotados, notificações por email, e posterior integração com sistema de pagamentos automatizado. Estes requisitos apresentam complexidades que vão além das capacidades nativas do LiteLLM.

Recomendação Principal: Abordagem Híbrida Evolutiva

Com base na análise detalhada, a recomendação é uma **abordagem híbrida evolutiva** que combina as vantagens do LiteLLM com customizações mínimas no LibreChat, implementada em fases para reduzir riscos e permitir validação incremental.

Fase 1 - MVP com LiteLLM: Implementar controle básico de tokens utilizando as capacidades nativas do LiteLLM, com modificações mínimas no LibreChat apenas para passar informações de usuário. Esta fase permite validação rápida do conceito com baixo risco.

Fase 2 - Extensões Customizadas: Implementar um serviço auxiliar que estende as capacidades do LiteLLM para suportar regras de negócio específicas como créditos acumulativos e integração com Stripe.

Fase 3 - Otimizações e Dashboards: Implementar dashboards avançados no Metabase e otimizações de performance baseadas nos dados coletados nas fases anteriores.

Justificativa Técnica da Recomendação

A abordagem híbrida evolutiva oferece o melhor equilíbrio entre todos os critérios avaliados. Ela permite aproveitar a robustez e precisão do LiteLLM enquanto mantém flexibilidade para implementar regras de negócio específicas através de serviços auxiliares.

Esta abordagem minimiza os riscos associados à customização extensiva do LibreChat, mantendo a capacidade de aplicar atualizações e correções de segurança. Ao mesmo tempo, oferece um caminho claro para implementar todas as funcionalidades necessárias para a IA SOLARIS.

A implementação em fases permite validação incremental e reduz o risco de over-engineering. Cada fase entrega valor tangível e pode ser avaliada independentemente antes de prosseguir para a próxima.

Implementação Detalhada da Recomendação

Modificações Mínimas no LibreChat:

JavaScript

```
// Único middleware necessário no LibreChat
function userContextMiddleware(req, res, next) {
    if (req.user) {
        req.headers['x-user-id'] = req.user.id;
        req.headers['x-user-email'] = req.user.email;
    }
    next();
}

// Configuração de endpoint para usar LiteLLM
const litellmEndpoint = {
    name: "IA SOLARIS",
    apiKey: process.env.LITELLM_API_KEY,
    baseURL: "http://litellm-proxy:4000/v1",
    models: {
        default: ["gpt-4", "gpt-3.5-turbo"],
        fetch: false
    }
};
```

Configuração do LiteLLM:

YAML

```
# litellm.config.yaml
model_list:
  - model_name: gpt-4
    litellm_params:
      model: openai/gpt-4
      api_key: os.environ/OPENAI_API_KEY

general_settings:
  database_url: "postgresql://user:pass@localhost/litellm"

router_settings:
  enable_user_auth: true
  enable_budget_tracking: true

user_api_key_auth:
```

```
  budget_duration: "1mo"
  default_budget: 25.0
```

Serviço Auxiliar para Regras de Negócio:

Python

```
# Serviço que estende LiteLLM para regras específicas
class IASOLARISTokenService:
    def __init__(self):
        self.litellm_client = LiteLLMClient()
        self.stripe_client = StripeClient()

    async def check_user_tokens(self, user_id):
        # Verificar tokens no LiteLLM
        litellm_data = await self.litellm_client.get_user_usage(user_id)

        # Aplicar regras específicas (créditos acumulativos)
        additional_credits = await self.get_additional_credits(user_id)

        return {
            'can_proceed': litellm_data['remaining_budget'] > 0 or
additional_credits > 0,
            'monthly_remaining': litellm_data['remaining_budget'],
            'additional_credits': additional_credits,
            'total_available': litellm_data['remaining_budget'] +
additional_credits
        }
```

Benefícios da Abordagem Recomendada

1. **Baixo Risco**: Modificações mínimas no LibreChat reduzem riscos de incompatibilidade
2. **Alta Precisão**: Aproveita a precisão nativa do LiteLLM para contabilização
3. **Flexibilidade**: Permite implementação de regras específicas através de serviços auxiliares
4. **Escalabilidade**: Herda as capacidades de escala do LiteLLM
5. **Manutenibilidade**: Mantém separação clara de responsabilidades
6. **Evolutiva**: Permite crescimento incremental das funcionalidades

Resposta ao Questionamento do Desenvolvedor

O desenvolvedor está correto em questionar a customização extensiva do LibreChat para controle de tokens. A separação de responsabilidades é um princípio fundamental de arquitetura de software, e misturar funcionalidades de interface de usuário com lógica de billing pode criar problemas de manutenibilidade.

No entanto, a solução não é necessariamente usar apenas o LiteLLM nativo, pois ele não suporta todas as regras de negócio específicas necessárias. A abordagem híbrida recomendada oferece o melhor dos dois mundos: aproveita as capacidades especializadas do LiteLLM enquanto permite implementação de regras específicas através de arquitetura modular.

A chave é encontrar o equilíbrio certo entre flexibilidade e simplicidade, implementando apenas as customizações mínimas necessárias no LibreChat e delegando a lógica complexa para sistemas especializados.

Avaliação Aprofundada da Customização do LibreChat

Arquitetura Atual do LibreChat e Pontos de Integração

O LibreChat possui uma arquitetura modular baseada em Node.js e Express, com separação clara entre frontend React e backend API. A estrutura atual inclui sistemas de autenticação, gerenciamento de conversas, integração com múltiplos provedores de IA, e persistência de dados em MongoDB. Para implementar controle de tokens, seria necessário intervir em múltiplos pontos desta arquitetura.

Os principais pontos de integração para controle de tokens incluem o middleware de autenticação, onde seria necessário verificar o status de tokens do usuário antes de permitir acesso às funcionalidades de chat. O sistema de roteamento de mensagens também precisaria ser modificado para interceptar requisições e aplicar verificações de quota antes de encaminhar para os provedores de IA.

A camada de persistência exigiria extensões significativas para armazenar dados de consumo de tokens, histórico de transações, e configurações de limites por usuário. Estas

modificações impactariam diretamente o schema do banco de dados e poderiam criar incompatibilidades com futuras atualizações do LibreChat.

JavaScript

```
// Exemplo de modificação necessária no middleware de autenticação
const originalAuthMiddleware = require('./middleware/auth');

async function enhancedAuthMiddleware(req, res, next) {
    // Executar autenticação original
    await originalAuthMiddleware(req, res, async () => {
        if (req.user) {
            // Verificação adicional de tokens
            const tokenStatus = await checkUserTokenStatus(req.user.id);

            if (!tokenStatus.canProceed) {
                return res.status(403).json({
                    error: 'Limite de tokens excedido',
                    availableTokens: tokenStatus.remaining,
                    upgradeRequired: true,
                    upgradeUrl: `/billing/upgrade?userId=${req.user.id}`
                });
            }

            // Adicionar informações de tokens ao contexto da requisição
            req.tokenContext = {
                available: tokenStatus.remaining,
                used: tokenStatus.used,
                limit: tokenStatus.limit,
                resetDate: tokenStatus.resetDate
            };
        }

        next();
    });
}
```

Complexidade de Implementação e Manutenção

A implementação de controle de tokens no LibreChat introduz complexidade significativa em múltiplas camadas da aplicação. No nível de banco de dados, seria necessário criar novas coleções para armazenar dados de consumo, implementar índices otimizados para consultas frequentes, e garantir consistência transacional entre operações de chat e contabilização de tokens.

A camada de aplicação exigiria modificações extensas nos controladores existentes, implementação de novos serviços para gerenciamento de tokens, e integração com sistemas externos como Stripe para processamento de pagamentos. Cada uma dessas modificações representa um ponto potencial de falha e aumenta a superfície de ataque do sistema.

A manutenção contínua seria particularmente desafiadora devido à natureza evolutiva do LibreChat. O projeto recebe atualizações frequentes que podem incluir refatorações significativas, mudanças de API, ou alterações na estrutura de dados. Customizações extensas podem criar conflitos com essas atualizações, exigindo esforço contínuo para manter compatibilidade.

JavaScript

```
// Exemplo de serviço de tokens que precisaria ser mantido
class TokenManagementService {
    constructor() {
        this.db = require('./database');
        this.stripe = require('./stripe-client');
        this.emailService = require('./email-service');
    }

    async consumeTokens(userId, estimatedTokens) {
        const session = await this.db.startSession();

        try {
            await session.withTransaction(async () => {
                // Verificar tokens disponíveis
                const userTokens = await this.db.collection('userTokens')
                    .findOne({ userId }, { session });

                if (!userTokens || userTokens.available < estimatedTokens) {
                    throw new Error('Tokens insuficientes');
                }

                // Decrementar tokens
                await this.db.collection('userTokens').updateOne(
                    { userId },
                    {
                        $inc: {
                            available: -estimatedTokens,
                            used: estimatedTokens
                        },
                        $push: {
```

```
                            transactions: {
                                type: 'consumption',
                                amount: estimatedTokens,
                                timestamp: new Date(),
                                description: 'Chat message processing'
                            }
                        }
                    },
                    { session }
                );

                // Verificar se precisa enviar alertas
                const updatedTokens = await this.db.collection('userTokens')
                    .findOne({ userId }, { session });

                await this.checkAndSendAlerts(userId, updatedTokens);
            });
        } finally {
            await session.endSession();
        }
    }

    async checkAndSendAlerts(userId, tokenData) {
        const usagePercentage = tokenData.used / tokenData.limit;

        if (usagePercentage >= 0.9 && !tokenData.alert90Sent) {
            await this.emailService.sendTokenAlert(userId, '90%');
            await this.db.collection('userTokens').updateOne(
                { userId },
                { $set: { alert90Sent: true } }
            );
        }

        if (usagePercentage >= 1.0) {
            await this.emailService.sendTokenExhausted(userId);
            await this.blockUserAccess(userId);
        }
    }
}
```

Impacto na Performance e Escalabilidade

A adição de verificações de tokens em cada requisição introduz overhead significativo que pode impactar a performance geral do sistema. Cada verificação requer consultas ao banco de dados, cálculos de consumo, e potencialmente operações de escrita para atualizar

contadores. Em cenários de alta concorrência, este overhead pode se tornar um gargalo significativo.

A escalabilidade horizontal também pode ser comprometida pela necessidade de manter consistência de dados entre instâncias. Operações de contabilização de tokens exigem atomicidade para evitar condições de corrida, o que pode limitar a capacidade de distribuir carga entre múltiplos servidores.

O caching de dados de tokens apresenta desafios adicionais, pois os dados devem ser mantidos atualizados em tempo real para evitar over-consumption. Estratégias de cache inadequadas podem resultar em usuários consumindo mais tokens do que deveriam, criando problemas financeiros para a plataforma.

JavaScript

```
// Exemplo de otimização com cache Redis para melhorar performance
class OptimizedTokenService {
    constructor() {
        this.redis = require('./redis-client');
        this.db = require('./database');
    }

    async checkTokensWithCache(userId) {
        // Tentar buscar do cache primeiro
        const cacheKey = `user_tokens:${userId}`;
        let tokenData = await this.redis.get(cacheKey);

        if (tokenData) {
            tokenData = JSON.parse(tokenData);

            // Verificar se cache não está muito desatualizado
            const cacheAge = Date.now() - tokenData.lastUpdated;
            if (cacheAge < 30000) { // 30 segundos
                return tokenData;
            }
        }

        // Buscar dados atualizados do banco
        tokenData = await this.db.collection('userTokens')
            .findOne({ userId });

        if (tokenData) {
            tokenData.lastUpdated = Date.now();
```

```
            // Atualizar cache com TTL de 60 segundos
            await this.redis.setex(
                cacheKey,
                60,
                JSON.stringify(tokenData)
            );
        }

        return tokenData;
    }

    async consumeTokensOptimized(userId, tokens) {
        // Usar operação atômica Redis para verificação rápida
        const available = await this.redis.decrby(
            `user_tokens_available:${userId}`,
            tokens
        );

        if (available < 0) {
            // Reverter operação se não há tokens suficientes
            await this.redis.incrby(
                `user_tokens_available:${userId}`,
                tokens
            );
            throw new Error('Tokens insuficientes');
        }

        // Agendar sincronização com banco de dados
        await this.scheduleDbSync(userId, tokens);

        return { success: true, remaining: available };
    }
}
```

Questões de Segurança e Auditoria

A implementação de controle de tokens no LibreChat introduz novas superfícies de ataque que devem ser cuidadosamente consideradas. A lógica de verificação de tokens deve ser à prova de manipulação, evitando cenários onde usuários possam contornar as verificações através de requisições malformadas ou ataques de timing.

A auditoria torna-se mais complexa quando a lógica de billing está integrada ao sistema de chat. Discrepâncias entre tokens consumidos e tokens cobrados podem ser difíceis de

rastrear e resolver, especialmente quando múltiplos sistemas estão envolvidos na cadeia de processamento.

A segregação de dados também é uma preocupação importante. Dados financeiros e de billing devem ser adequadamente protegidos e isolados de outros dados do sistema. A mistura de responsabilidades pode tornar mais difícil implementar controles de acesso granulares e políticas de retenção de dados específicas.

JavaScript

```
// Exemplo de implementação segura com auditoria
class SecureTokenService {
    constructor() {
        this.auditLogger = require('./audit-logger');
        this.encryption = require('./encryption-service');
    }

    async secureTokenConsumption(userId, requestData, estimatedTokens) {
        const auditId = this.generateAuditId();

        try {
            // Log início da operação
            await this.auditLogger.log({
                auditId,
                action: 'token_consumption_start',
                userId,
                estimatedTokens,
                requestHash: this.hashRequest(requestData),
                timestamp: new Date(),
                ipAddress: requestData.ip,
                userAgent: requestData.userAgent
            });

            // Verificação com múltiplas camadas de validação
            const tokenCheck = await this.multiLayerTokenCheck(userId);

            if (!tokenCheck.valid) {
                await this.auditLogger.log({
                    auditId,
                    action: 'token_consumption_denied',
                    userId,
                    reason: tokenCheck.reason,
                    availableTokens: tokenCheck.available,
                    timestamp: new Date()
                });
```

```
                throw new Error('Acesso negado: ' + tokenCheck.reason);
            }

            // Consumir tokens com verificação de integridade
            const result = await this.atomicTokenConsumption(
                userId,
                estimatedTokens,
                auditId
            );

            await this.auditLogger.log({
                auditId,
                action: 'token_consumption_success',
                userId,
                tokensConsumed: result.actualTokens,
                remainingTokens: result.remaining,
                timestamp: new Date()
            });

            return result;

        } catch (error) {
            await this.auditLogger.log({
                auditId,
                action: 'token_consumption_error',
                userId,
                error: error.message,
                timestamp: new Date()
            });

            throw error;
        }
    }

    async multiLayerTokenCheck(userId) {
        // Verificação primária no cache
        const cacheCheck = await this.checkTokensCache(userId);

        // Verificação secundária no banco de dados
        const dbCheck = await this.checkTokensDatabase(userId);

        // Verificação de consistência
        if (Math.abs(cacheCheck.available - dbCheck.available) > 1000) {
            await this.auditLogger.log({
                action: 'token_inconsistency_detected',
                userId,
                cacheValue: cacheCheck.available,
                dbValue: dbCheck.available,
```

```
                timestamp: new Date()
            });

            // Usar valor mais conservador em caso de inconsistência
            return {
                valid: Math.min(cacheCheck.available, dbCheck.available) > 0,
                available: Math.min(cacheCheck.available, dbCheck.available),
                reason: cacheCheck.available !== dbCheck.available ?
                    'Inconsistência detectada, usando valor conservador' :
null
            };
        }

        return {
            valid: dbCheck.available > 0,
            available: dbCheck.available
        };
    }
}
```

Compatibilidade com Atualizações do LibreChat

Uma das maiores preocupações com customizações extensas é a compatibilidade com futuras atualizações do LibreChat. O projeto está em desenvolvimento ativo, com releases frequentes que podem incluir mudanças breaking changes na API, refatorações de código, ou alterações na estrutura de dados.

Customizações que modificam arquivos core do LibreChat são particularmente vulneráveis a conflitos durante atualizações. Mesmo modificações aparentemente simples podem se tornar incompatíveis se os desenvolvedores do LibreChat decidirem refatorar as áreas modificadas.

A estratégia de merge durante atualizações torna-se complexa quando há modificações extensas. Conflitos de merge podem exigir análise detalhada para determinar como integrar mudanças upstream com customizações locais, processo que pode ser propenso a erros e consumir tempo significativo.

Bash

```
# Exemplo de processo complexo de atualização com customizações
#!/bin/bash
```

```
# Script de atualização com customizações
echo "Iniciando processo de atualização do LibreChat..."

# Backup das customizações atuais
git stash push -m "Customizações antes da atualização $(date)"

# Fetch das últimas mudanças
git fetch upstream

# Tentar merge automático
if git merge upstream/main; then
    echo "Merge automático bem-sucedido"
else
    echo "Conflitos detectados, resolução manual necessária"

    # Listar arquivos com conflito
    git diff --name-only --diff-filter=U

    # Aplicar patches customizados se necessário
    for patch in patches/*.patch; do
        if git apply --check "$patch" 2>/dev/null; then
            git apply "$patch"
            echo "Patch $patch aplicado com sucesso"
        else
            echo "ERRO: Patch $patch não pode ser aplicado"
            echo "Revisão manual necessária"
        fi
    done
fi

# Restaurar customizações específicas
git stash pop

# Executar testes para verificar compatibilidade
npm test

# Verificar funcionalidades customizadas
npm run test:custom-features

echo "Atualização concluída. Revisar logs para possíveis problemas."
```

Análise de Custo-Benefício da Customização

O custo total de implementar e manter customizações extensas no LibreChat deve ser cuidadosamente avaliado contra os benefícios obtidos. O custo inicial de desenvolvimento inclui não apenas a implementação das funcionalidades, mas também o tempo necessário

para entender profundamente a arquitetura do LibreChat e identificar os pontos de integração adequados.

Os custos contínuos de manutenção podem ser significativos, incluindo tempo para resolver conflitos durante atualizações, debugging de problemas específicos das customizações, e adaptação a mudanças na arquitetura upstream. Estes custos tendem a aumentar ao longo do tempo conforme as customizações se tornam mais complexas e o código base do LibreChat evolui.

O risco de vendor lock-in também deve ser considerado. Customizações extensas podem tornar difícil migrar para outras soluções no futuro, criando dependência técnica que pode limitar opções estratégicas. Este risco é particularmente relevante em projetos open source onde a direção futura do desenvolvimento pode mudar.

JavaScript

```
// Exemplo de análise de custo-benefício quantitativa
class CustomizationCostAnalysis {
    calculateImplementationCost() {
        return {
            // Custos de desenvolvimento inicial
            analysis: 40, // horas para análise da arquitetura
            implementation: 120, // horas de desenvolvimento
            testing: 60, // horas de testes
            documentation: 20, // horas de documentação

            // Custos de infraestrutura
            additionalServers: 200, // USD/mês para recursos extras
            monitoring: 50, // USD/mês para ferramentas de monitoramento

            // Total inicial
            totalInitial: 240 * 100 + 250 // horas * taxa + infraestrutura
        };
    }

    calculateMaintenanceCost() {
        return {
            // Custos mensais de manutenção
            updateConflicts: 8, // horas/mês para resolver conflitos
            bugFixes: 12, // horas/mês para correções
            monitoring: 4, // horas/mês para monitoramento

            // Custos anuais
            majorUpdates: 40, // horas/ano para atualizações grandes
```

```
            securityPatches: 20, // horas/ano para patches de segurança

            // Total anual
            totalAnnual: (8 + 12 + 4) * 12 * 100 + (40 + 20) * 100
        };
    }

    calculateBenefits() {
        return {
            // Benefícios quantificáveis
            revenueControl: 5000, // USD/mês de receita controlada
            costSavings: 1500, // USD/mês economizados em over-usage
            customerRetention: 2000, // USD/mês em retenção melhorada

            // Benefícios qualitativos (difíceis de quantificar)
            brandControl: 'Alto',
            customerExperience: 'Melhorado',
            competitiveAdvantage: 'Médio'
        };
    }

    calculateROI(timeHorizon = 12) {
        const implementation = this.calculateImplementationCost();
        const maintenance = this.calculateMaintenanceCost();
        const benefits = this.calculateBenefits();

        const totalCosts = implementation.totalInitial +
                           (maintenance.totalAnnual * timeHorizon / 12);

        const totalBenefits = (benefits.revenueControl +
                               benefits.costSavings +
                               benefits.customerRetention) * timeHorizon;

        return {
            totalCosts,
            totalBenefits,
            netBenefit: totalBenefits - totalCosts,
            roi: ((totalBenefits - totalCosts) / totalCosts) * 100,
            breakEvenMonths: totalCosts / (benefits.revenueControl +
                                           benefits.costSavings +
                                           benefits.customerRetention)
        };
    }
}
```

Alternativas de Implementação Menos Invasivas

Existem abordagens para implementar controle de tokens no LibreChat que são menos invasivas e mantêm melhor compatibilidade com atualizações futuras. Uma estratégia é utilizar o sistema de plugins ou hooks do LibreChat, se disponível, para interceptar requisições sem modificar o código core.

Outra abordagem é implementar o controle de tokens como um middleware externo que opera a nível de proxy reverso, interceptando requisições HTTP antes que cheguem ao LibreChat. Esta abordagem mantém o LibreChat completamente inalterado enquanto adiciona a funcionalidade necessária.

A implementação através de database triggers ou stored procedures também pode ser considerada, movendo a lógica de controle para o nível de banco de dados. Esta abordagem pode ser menos invasiva no código da aplicação, mas introduz complexidade na camada de dados.

JavaScript

```
// Exemplo de middleware externo não-invasivo
const express = require('express');
const httpProxy = require('http-proxy-middleware');

class TokenControlProxy {
    constructor() {
        this.app = express();
        this.tokenService = new TokenService();
        this.setupMiddleware();
    }

    setupMiddleware() {
        // Middleware para verificação de tokens
        this.app.use('/api/ask/*', async (req, res, next) => {
            try {
                const userId = this.extractUserId(req);

                if (userId) {
                    const tokenCheck = await
this.tokenService.checkTokens(userId);

                    if (!tokenCheck.canProceed) {
                        return res.status(403).json({
                            error: 'Limite de tokens excedido',
                            available: tokenCheck.available,
                            limit: tokenCheck.limit,
```

```
                            resetDate: tokenCheck.resetDate
                        });
                    }

                    // Adicionar headers para rastreamento
                    req.headers['x-token-tracking'] = 'enabled';
                    req.headers['x-user-tokens-available'] =
tokenCheck.available;
                }

                next();
            } catch (error) {
                console.error('Erro na verificação de tokens:', error);
                // Em caso de erro, permitir requisição (fail-open)
                next();
            }
        });

        // Proxy para LibreChat
        this.app.use('/', httpProxy({
            target: 'http://librechat:3080',
            changeOrigin: true,
            onProxyRes: (proxyRes, req, res) => {
                // Interceptar resposta para contabilizar tokens
                if (req.path.startsWith('/api/ask/')) {
                    this.handleTokenConsumption(req, proxyRes);
                }
            }
        }));
    }

    extractUserId(req) {
        // Extrair user ID do token JWT ou session
        const token = req.headers.authorization?.replace('Bearer ', '');
        if (token) {
            try {
                const decoded = jwt.verify(token, process.env.JWT_SECRET);
                return decoded.userId;
            } catch (error) {
                console.error('Erro ao decodificar token:', error);
            }
        }
        return null;
    }

    async handleTokenConsumption(req, proxyRes) {
        const userId = this.extractUserId(req);
```

```
        if (userId && proxyRes.statusCode === 200) {
            // Aguardar resposta completa para calcular tokens
            let responseBody = '';

            proxyRes.on('data', (chunk) => {
                responseBody += chunk;
            });

            proxyRes.on('end', async () => {
                try {
                    const response = JSON.parse(responseBody);

                    if (response.usage && response.usage.total_tokens) {
                        await this.tokenService.consumeTokens(
                            userId,
                            response.usage.total_tokens
                        );
                    }
                } catch (error) {
                    console.error('Erro ao processar consumo de tokens:',
error);
                }
            });
        }
    }
}

// Inicializar proxy
const proxy = new TokenControlProxy();
proxy.app.listen(3000, () => {
    console.log('Token Control Proxy rodando na porta 3000');
});
```

Comparação Detalhada de Abordagens Alternativas

Abordagem 1: LiteLLM Puro com Configuração Avançada

A utilização do LiteLLM em sua forma pura, aproveitando ao máximo suas capacidades nativas, representa uma abordagem conservadora que prioriza estabilidade e manutenibilidade. Esta estratégia envolve configurar o LiteLLM como proxy principal para todas as requisições de IA, implementando controle de usuários através de suas APIs nativas e sistemas de budget management.

A configuração avançada do LiteLLM permite implementar a maioria dos requisitos de controle de tokens através de suas funcionalidades built-in. O sistema de usuários pode ser configurado para mapear diretamente os usuários do LibreChat, com limites individuais de budget, TPM (Tokens Per Minute), e RPM (Requests Per Minute). O LiteLLM oferece APIs robustas para criação, atualização, e monitoramento de usuários, permitindo sincronização automática com o sistema de autenticação do LibreChat.

YAML

```yaml
# Configuração avançada do LiteLLM para IA SOLARIS
model_list:
  - model_name: gpt-4-solaris
    litellm_params:
      model: openai/gpt-4
      api_key: os.environ/OPENAI_API_KEY
      max_tokens: 4096
      temperature: 0.7

  - model_name: gpt-3.5-turbo-solaris
    litellm_params:
      model: openai/gpt-3.5-turbo
      api_key: os.environ/OPENAI_API_KEY
      max_tokens: 4096

litellm_settings:
  # Configurações de banco de dados para persistência
  database_url: "postgresql://user:pass@localhost:5432/litellm"

  # Configurações de cache para performance
  redis_host: "localhost"
  redis_port: 6379
  redis_password: os.environ/REDIS_PASSWORD

  # Configurações de logging e auditoria
  success_callback: ["langfuse", "posthog"]
  failure_callback: ["sentry"]

  # Configurações de rate limiting global
  rpm_limit: 1000
  tpm_limit: 100000

  # Configurações de timeout
  request_timeout: 600

router_settings:
```

```
  # Habilitar autenticação de usuários
  enable_user_auth: true

  # Habilitar tracking de budget
  enable_budget_tracking: true

  # Configurações de fallback
  enable_fallbacks: true
  fallback_models: ["gpt-3.5-turbo-solaris"]

  # Configurações de load balancing
  enable_loadbalancing: true

general_settings:
  # Configurações de usuário padrão
  default_user_budget: 25.0  # USD por mês
  budget_duration: "1mo"

  # Configurações de alertas
  alert_webhooks:
    - url: "https://iasolaris.com/api/webhooks/budget-alert"
      events: ["budget_crossed_80", "budget_crossed_90", "budget_exceeded"]

  # Configurações de UI administrativa
  ui_username: os.environ/LITELLM_UI_USERNAME
  ui_password: os.environ/LITELLM_UI_PASSWORD
```

O sistema de webhooks do LiteLLM pode ser configurado para notificar sistemas externos quando eventos importantes ocorrem, como usuários atingindo limites de budget ou consumindo tokens. Estes webhooks podem ser integrados com sistemas de email, Slack, ou outras ferramentas de notificação para alertar administradores e usuários sobre status de consumo.

A precisão na contabilização de tokens é uma vantagem significativa desta abordagem. O LiteLLM tem acesso direto aos dados de usage retornados pelas APIs dos provedores, eliminando as discrepâncias conhecidas entre sistemas como o LibreChat e os dados reais da OpenAI. Esta precisão é fundamental para sistemas de billing onde a exatidão financeira é crítica.

Python

```
# Exemplo de integração com LiteLLM para sincronização de usuários
import asyncio
```

```
import aiohttp
from typing import List, Dict

class LiteLLMUserManager:
    def __init__(self, litellm_base_url: str, api_key: str):
        self.base_url = litellm_base_url
        self.api_key = api_key
        self.session = None

    async def __aenter__(self):
        self.session = aiohttp.ClientSession(
            headers={"Authorization": f"Bearer {self.api_key}"}
        )
        return self

    async def __aexit__(self, exc_type, exc_val, exc_tb):
        if self.session:
            await self.session.close()

    async def sync_users_from_librechat(self, librechat_users: List[Dict]):
        """Sincronizar usuários do LibreChat com LiteLLM"""
        for user in librechat_users:
            await self.create_or_update_user(
                user_id=user['_id'],
                email=user['email'],
                name=user.get('name', ''),
                subscription_tier=user.get('subscriptionTier', 'basic')
            )

    async def create_or_update_user(self, user_id: str, email: str,
                                    name: str, subscription_tier: str):
        """Criar ou atualizar usuário no LiteLLM"""

        # Determinar budget baseado no tier de assinatura
        budget_mapping = {
            'basic': 25.0,      # 1M tokens ≈ $25
            'premium': 50.0,    # 2M tokens ≈ $50
            'enterprise': 100.0 # 4M tokens ≈ $100
        }

        user_data = {
            "user_id": user_id,
            "user_email": email,
            "user_alias": name,
            "budget_limit": budget_mapping.get(subscription_tier, 25.0),
            "budget_duration": "1mo",
            "tpm_limit": 10000,
            "rpm_limit": 100,
```

```
            "models": ["gpt-4-solaris", "gpt-3.5-turbo-solaris"],
            "metadata": {
                "subscription_tier": subscription_tier,
                "source": "librechat_sync",
                "sync_timestamp": asyncio.get_event_loop().time()
            }
        }

        async with self.session.post(
            f"{self.base_url}/user/new",
            json=user_data
        ) as response:
            if response.status == 200:
                result = await response.json()
                print(f"Usuário {user_id} sincronizado com sucesso")
                return result
            else:
                error = await response.text()
                print(f"Erro ao sincronizar usuário {user_id}: {error}")
                raise Exception(f"Falha na sincronização: {error}")

    async def get_user_usage(self, user_id: str) -> Dict:
        """Obter dados de uso de um usuário"""
        async with self.session.get(
            f"{self.base_url}/user/usage",
            params={"user_id": user_id}
        ) as response:
            if response.status == 200:
                return await response.json()
            else:
                raise Exception(f"Erro ao obter dados de uso: {await
response.text()}")

    async def update_user_budget(self, user_id: str, additional_budget:
float):
        """Adicionar budget adicional para um usuário (compra de créditos)"""
        current_data = await self.get_user_usage(user_id)
        new_budget = current_data['budget_limit'] + additional_budget

        update_data = {
            "user_id": user_id,
            "budget_limit": new_budget,
            "metadata": {
                "additional_purchase": additional_budget,
                "purchase_timestamp": asyncio.get_event_loop().time()
            }
        }
```

```
        async with self.session.post(
            f"{self.base_url}/user/update",
            json=update_data
        ) as response:
            if response.status == 200:
                return await response.json()
            else:
                raise Exception(f"Erro ao atualizar budget: {await
response.text()}")

# Exemplo de uso
async def sync_users_example():
    # Buscar usuários do LibreChat (exemplo)
    librechat_users = [
        {
            "_id": "user_123",
            "email": "usuario@iasolaris.com",
            "name": "João Silva",
            "subscriptionTier": "premium"
        }
    ]

    async with LiteLLMUserManager(
        "http://litellm:4000",
        "sk-litellm-api-key"
    ) as manager:
        await manager.sync_users_from_librechat(librechat_users)

        # Verificar uso atual
        usage = await manager.get_user_usage("user_123")
        print(f"Uso atual: {usage}")
```

Limitações da Abordagem LiteLLM Puro

Apesar das vantagens significativas, a abordagem LiteLLM puro apresenta limitações importantes que devem ser consideradas no contexto específico da IA SOLARIS. A principal limitação é a flexibilidade restrita para implementar regras de negócio complexas que não são suportadas nativamente pelo LiteLLM.

O sistema de créditos acumulativos proposto para a IA SOLARIS, onde créditos comprados avulsamente são mantidos entre meses enquanto o limite mensal é resetado, não é suportado nativamente pelo LiteLLM. O sistema de budget do LiteLLM opera com períodos fixos e não oferece a granularidade necessária para distinguir entre diferentes tipos de créditos.

A integração com sistemas de pagamento como Stripe também não é nativa no LiteLLM. Embora seja possível implementar esta integração através de webhooks e APIs, ela requer desenvolvimento adicional de sistemas auxiliares que podem adicionar complexidade à arquitetura geral.

Python

```
# Exemplo de limitação: sistema de créditos complexo não suportado
nativamente
class ComplexCreditSystem:
    """
    Sistema que seria necessário para implementar regras de negócio
    específicas não suportadas pelo LiteLLM nativo
    """

    def __init__(self, litellm_client, database):
        self.litellm = litellm_client
        self.db = database

    async def calculate_available_tokens(self, user_id: str) -> Dict:
        """
        Calcular tokens disponíveis considerando regras específicas:
        1. Tokens mensais (resetam todo mês)
        2. Créditos adicionais (acumulativos, não expiram)
        3. Prioridade: usar tokens mensais primeiro
        """

        # Obter dados do LiteLLM (apenas budget mensal)
        litellm_data = await self.litellm.get_user_usage(user_id)
        monthly_remaining = litellm_data.get('budget_remaining', 0)

        # Obter créditos adicionais do banco próprio
        additional_credits = await self.db.get_additional_credits(user_id)

        # Aplicar regras de negócio específicas
        if monthly_remaining > 0:
            # Ainda há tokens mensais disponíveis
            total_available = monthly_remaining +
additional_credits['balance']
            next_source = 'monthly'
        else:
            # Tokens mensais esgotados, usar apenas créditos adicionais
            total_available = additional_credits['balance']
            next_source = 'additional'

        return {
```

```
            'total_available': total_available,
            'monthly_remaining': monthly_remaining,
            'additional_credits': additional_credits['balance'],
            'next_consumption_source': next_source,
            'can_proceed': total_available > 0
        }

    async def consume_tokens(self, user_id: str, tokens_needed: int) -> Dict:
        """
        Consumir tokens seguindo regras de prioridade específicas
        """
        availability = await self.calculate_available_tokens(user_id)

        if not availability['can_proceed']:
            raise InsufficientTokensError("Tokens insuficientes")

        if availability['next_consumption_source'] == 'monthly':
            # Consumir do budget mensal primeiro
            monthly_consumption = min(tokens_needed,
availability['monthly_remaining'])
            remaining_needed = tokens_needed - monthly_consumption

            # Registrar consumo no LiteLLM (budget mensal)
            await self.litellm.record_usage(user_id, monthly_consumption)

            if remaining_needed > 0:
                # Consumir restante dos créditos adicionais
                await self.db.consume_additional_credits(user_id,
remaining_needed)
        else:
            # Consumir apenas dos créditos adicionais
            await self.db.consume_additional_credits(user_id, tokens_needed)

        return {
            'consumed': tokens_needed,
            'monthly_consumed': monthly_consumption if 'monthly_consumption'
in locals() else 0,
            'additional_consumed': remaining_needed if 'remaining_needed' in
locals() else tokens_needed,
            'remaining_total': availability['total_available'] -
tokens_needed
        }
```

Abordagem 2: Arquitetura de Microserviços com Separação de Responsabilidades

Uma abordagem arquitetural mais sofisticada envolve a criação de microserviços especializados que lidam com diferentes aspectos do controle de tokens. Esta estratégia mantém o LibreChat e LiteLLM em suas funções principais enquanto adiciona serviços dedicados para billing, user management, e business logic específica.

O Token Management Service seria responsável por toda a lógica de controle de tokens, incluindo verificações de limites, aplicação de regras de negócio complexas, e integração com sistemas de pagamento. Este serviço operaria como uma camada intermediária entre o LibreChat e o LiteLLM, interceptando requisições e aplicando controles conforme necessário.

O User Billing Service gerenciaria todos os aspectos financeiros, incluindo integração com Stripe, processamento de pagamentos, geração de faturas, e manutenção de histórico de transações. Este serviço seria completamente independente dos sistemas de chat, permitindo evolução independente e reutilização em outros contextos.

Python

```
# Arquitetura de microserviços para controle de tokens
from fastapi import FastAPI, HTTPException, Depends
from pydantic import BaseModel
from typing import Optional, List
import asyncio

# Token Management Service
class TokenManagementService:
    def __init__(self):
        self.app = FastAPI(title="Token Management Service")
        self.setup_routes()

    def setup_routes(self):
        @self.app.post("/tokens/check")
        async def check_tokens(request: TokenCheckRequest):
            """Verificar se usuário pode consumir tokens"""
            try:
                user_data = await self.get_user_token_data(request.user_id)

                # Aplicar regras de negócio específicas
                can_proceed = await self.apply_business_rules(
                    user_data,
                    request.estimated_tokens
                )
```

```
                return TokenCheckResponse(
                    can_proceed=can_proceed['allowed'],
                    available_tokens=can_proceed['available'],
                    monthly_remaining=can_proceed['monthly'],
                    additional_credits=can_proceed['additional'],
                    next_reset_date=can_proceed['reset_date']
                )

            except Exception as e:
                raise HTTPException(status_code=500, detail=str(e))

        @self.app.post("/tokens/consume")
        async def consume_tokens(request: TokenConsumptionRequest):
            """Registrar consumo de tokens"""
            try:
                result = await self.process_token_consumption(
                    request.user_id,
                    request.actual_tokens,
                    request.model_used,
                    request.request_metadata
                )

                return TokenConsumptionResponse(
                    success=True,
                    consumed=result['consumed'],
                    remaining=result['remaining'],
                    transaction_id=result['transaction_id']
                )

            except Exception as e:
                raise HTTPException(status_code=500, detail=str(e))

    async def apply_business_rules(self, user_data: dict, estimated_tokens:
int) -> dict:
        """Aplicar regras de negócio específicas da IA SOLARIS"""

        # Regra 1: Verificar tokens mensais primeiro
        monthly_available = user_data['monthly_limit'] -
user_data['monthly_used']

        # Regra 2: Verificar créditos adicionais
        additional_available = user_data['additional_credits']

        # Regra 3: Calcular total disponível
        total_available = monthly_available + additional_available

        # Regra 4: Verificar se pode prosseguir
        can_proceed = total_available >= estimated_tokens
```

```
        # Regra 5: Determinar próxima data de reset
        next_reset =
self.calculate_next_reset_date(user_data['subscription_start'])

        return {
            'allowed': can_proceed,
            'available': total_available,
            'monthly': monthly_available,
            'additional': additional_available,
            'reset_date': next_reset
        }

# User Billing Service
class UserBillingService:
    def __init__(self):
        self.app = FastAPI(title="User Billing Service")
        self.stripe_client = StripeClient()
        self.setup_routes()

    def setup_routes(self):
        @self.app.post("/billing/purchase-credits")
        async def purchase_credits(request: CreditPurchaseRequest):
            """Processar compra de créditos adicionais"""
            try:
                # Criar sessão de checkout no Stripe
                checkout_session = await
self.stripe_client.create_checkout_session(
                    user_id=request.user_id,
                    credit_package=request.package_id,
                    success_url=request.success_url,
                    cancel_url=request.cancel_url
                )

                return CreditPurchaseResponse(
                    checkout_url=checkout_session.url,
                    session_id=checkout_session.id
                )

            except Exception as e:
                raise HTTPException(status_code=500, detail=str(e))

        @self.app.post("/billing/webhook/stripe")
        async def stripe_webhook(request: StripeWebhookRequest):
            """Processar webhooks do Stripe"""
            try:
                # Verificar assinatura do webhook
                event = self.stripe_client.verify_webhook(
```

```
                    request.payload,
                    request.signature
                )

                if event.type == 'checkout.session.completed':
                    await self.process_successful_payment(event.data.object)
                elif event.type == 'payment_intent.payment_failed':
                    await self.process_failed_payment(event.data.object)

                return {"status": "success"}

            except Exception as e:
                raise HTTPException(status_code=400, detail=str(e))

    async def process_successful_payment(self, checkout_session):
        """Processar pagamento bem-sucedido"""
        user_id = checkout_session.metadata.user_id
        credit_amount = int(checkout_session.metadata.credit_amount)

        # Adicionar créditos à conta do usuário
        await self.add_credits_to_user(user_id, credit_amount)

        # Enviar email de confirmação
        await self.send_purchase_confirmation(user_id, credit_amount)

        # Reativar conta se estava bloqueada
        await self.reactivate_user_if_blocked(user_id)

# Notification Service
class NotificationService:
    def __init__(self):
        self.app = FastAPI(title="Notification Service")
        self.email_client = EmailClient()
        self.setup_routes()

    def setup_routes(self):
        @self.app.post("/notifications/token-alert")
        async def send_token_alert(request: TokenAlertRequest):
            """Enviar alerta de tokens"""
            try:
                if request.alert_type == "80_percent":
                    await self.send_80_percent_alert(request.user_id)
                elif request.alert_type == "90_percent":
                    await self.send_90_percent_alert(request.user_id)
                elif request.alert_type == "exhausted":
                    await self.send_exhausted_alert(request.user_id)

                return {"status": "sent"}
```

```
            except Exception as e:
                raise HTTPException(status_code=500, detail=str(e))

# Models para APIs
class TokenCheckRequest(BaseModel):
    user_id: str
    estimated_tokens: int
    model_name: str

class TokenCheckResponse(BaseModel):
    can_proceed: bool
    available_tokens: int
    monthly_remaining: int
    additional_credits: int
    next_reset_date: str

class TokenConsumptionRequest(BaseModel):
    user_id: str
    actual_tokens: int
    model_used: str
    request_metadata: dict

class TokenConsumptionResponse(BaseModel):
    success: bool
    consumed: int
    remaining: int
    transaction_id: str
```

Vantagens da Arquitetura de Microserviços

A arquitetura de microserviços oferece várias vantagens significativas para o controle de tokens na IA SOLARIS. A separação clara de responsabilidades permite que cada serviço seja desenvolvido, testado, e implantado independentemente, reduzindo a complexidade geral do sistema e facilitando a manutenção.

A escalabilidade independente é outra vantagem importante. Serviços que recebem mais carga, como o Token Management Service, podem ser escalados horizontalmente sem afetar outros componentes. Esta flexibilidade é particularmente valiosa em cenários onde diferentes aspectos do sistema têm padrões de uso distintos.

A testabilidade é significativamente melhorada com microserviços. Cada serviço pode ser testado isoladamente, com mocks para dependências externas. Esta abordagem facilita a

implementação de testes automatizados abrangentes e reduz a complexidade de debugging quando problemas ocorrem.

A reutilização de componentes também é facilitada. O User Billing Service, por exemplo, pode ser reutilizado em outros produtos da empresa que requerem funcionalidades de billing, maximizando o retorno sobre o investimento em desenvolvimento.

Desvantagens e Complexidades da Arquitetura de Microserviços

Apesar das vantagens, a arquitetura de microserviços introduz complexidades significativas que devem ser cuidadosamente consideradas. A comunicação entre serviços adiciona latência e pontos potenciais de falha. Cada chamada de API entre serviços representa uma oportunidade para timeouts, falhas de rede, ou inconsistências de dados.

A consistência de dados torna-se mais desafiadora em arquiteturas distribuídas. Transações que envolvem múltiplos serviços requerem padrões como Saga ou Two-Phase Commit, que são mais complexos de implementar e debugar do que transações locais.

O overhead operacional também é significativo. Cada microserviço requer sua própria infraestrutura, monitoramento, logging, e deployment pipeline. Esta complexidade pode ser excessiva para equipes pequenas ou projetos com recursos limitados.

YAML

```yaml
# Exemplo de complexidade de deployment com microserviços
# docker-compose.yml para IA SOLARIS com microserviços
version: '3.8'

services:
  # Serviços principais
  librechat:
    image: librechat:latest
    environment:
      - TOKEN_SERVICE_URL=http://token-management:8000
      - BILLING_SERVICE_URL=http://billing:8001
    depends_on:
      - token-management
      - mongodb
    networks:
      - iasolaris-network

  litellm:
```

```
    image: litellm:latest
    environment:
      - DATABASE_URL=postgresql://user:pass@postgres:5432/litellm
      - REDIS_URL=redis://redis:6379
    depends_on:
      - postgres
      - redis
    networks:
      - iasolaris-network

  # Microserviços customizados
  token-management:
    build: ./services/token-management
    environment:
      - DATABASE_URL=postgresql://user:pass@postgres:5432/tokens
      - LITELLM_URL=http://litellm:4000
      - BILLING_SERVICE_URL=http://billing:8001
    depends_on:
      - postgres
      - redis
    networks:
      - iasolaris-network
    deploy:
      replicas: 3  # Múltiplas instâncias para alta disponibilidade

  billing:
    build: ./services/billing
    environment:
      - DATABASE_URL=postgresql://user:pass@postgres:5432/billing
      - STRIPE_SECRET_KEY=${STRIPE_SECRET_KEY}
      - STRIPE_WEBHOOK_SECRET=${STRIPE_WEBHOOK_SECRET}
    depends_on:
      - postgres
    networks:
      - iasolaris-network
    deploy:
      replicas: 2

  notification:
    build: ./services/notification
    environment:
      - EMAIL_SERVICE_URL=${EMAIL_SERVICE_URL}
      - EMAIL_API_KEY=${EMAIL_API_KEY}
    networks:
      - iasolaris-network

  # Infraestrutura
  postgres:
```

```
    image: postgres:14
    environment:
      - POSTGRES_DB=iasolaris
      - POSTGRES_USER=user
      - POSTGRES_PASSWORD=pass
    volumes:
      - postgres_data:/var/lib/postgresql/data
    networks:
      - iasolaris-network

  redis:
    image: redis:7
    networks:
      - iasolaris-network

  mongodb:
    image: mongo:6
    volumes:
      - mongodb_data:/data/db
    networks:
      - iasolaris-network

  # Monitoramento e observabilidade
  prometheus:
    image: prometheus:latest
    volumes:
      - ./monitoring/prometheus.yml:/etc/prometheus/prometheus.yml
    networks:
      - iasolaris-network

  grafana:
    image: grafana:latest
    environment:
      - GF_SECURITY_ADMIN_PASSWORD=admin
    networks:
      - iasolaris-network

  # API Gateway
  nginx:
    image: nginx:latest
    volumes:
      - ./nginx/nginx.conf:/etc/nginx/nginx.conf
    ports:
      - "80:80"
      - "443:443"
    depends_on:
      - librechat
      - token-management
```

```
      - billing
    networks:
      - iasolaris-network

volumes:
  postgres_data:
  mongodb_data:

networks:
  iasolaris-network:
    driver: bridge
```

Abordagem 3: Proxy Inteligente com Lógica de Negócio Embarcada

Uma terceira abordagem envolve a implementação de um proxy inteligente que fica entre o LibreChat e os provedores de IA, embarcando toda a lógica de controle de tokens e regras de negócio. Este proxy opera de forma transparente, interceptando todas as requisições e aplicando controles sem modificar os sistemas upstream ou downstream.

O proxy inteligente pode ser implementado usando tecnologias como Envoy, Kong, ou uma solução customizada em Python/Node.js. A vantagem desta abordagem é a capacidade de implementar lógica complexa de negócio mantendo os sistemas existentes completamente inalterados.

Python

```
# Implementação de proxy inteligente para controle de tokens
from fastapi import FastAPI, Request, HTTPException
from fastapi.responses import StreamingResponse
import httpx
import asyncio
import json
from typing import Dict, Any

class IntelligentTokenProxy:
    def __init__(self):
        self.app = FastAPI(title="IA SOLARIS Token Proxy")
        self.openai_client = httpx.AsyncClient(
            base_url="https://api.openai.com",
            timeout=httpx.Timeout(60.0)
        )
        self.token_service = TokenService()
        self.billing_service = BillingService()
```

```
        self.setup_routes()

    def setup_routes(self):
        @self.app.api_route("/{path:path}", methods=["GET", "POST", "PUT",
"DELETE"])
        async def proxy_request(request: Request, path: str):
            """Proxy inteligente para todas as requisições"""

            # Verificar se é requisição de chat que precisa controle de
tokens
            if self.is_chat_request(path, request.method):
                return await self.handle_chat_request(request, path)
            else:
                # Proxy transparente para outras requisições
                return await self.transparent_proxy(request, path)

    async def handle_chat_request(self, request: Request, path: str):
        """Lidar com requisições de chat com controle de tokens"""

        # Extrair informações do usuário
        user_info = await self.extract_user_info(request)

        if not user_info:
            raise HTTPException(status_code=401, detail="Usuário não
autenticado")

        # Estimar tokens necessários
        request_body = await request.json()
        estimated_tokens = await self.estimate_tokens(request_body)

        # Verificar se usuário pode prosseguir
        token_check = await self.token_service.check_user_tokens(
            user_info['user_id'],
            estimated_tokens
        )

        if not token_check['can_proceed']:
            return await self.handle_insufficient_tokens(user_info,
token_check)

        # Prosseguir com a requisição
        response = await self.forward_to_openai(request, path, request_body)

        # Processar resposta e contabilizar tokens reais
        if response.status_code == 200:
            response_data = await response.json()
            actual_tokens = response_data.get('usage',
{}).get('total_tokens', estimated_tokens)
```

```
            # Registrar consumo real
            await self.token_service.consume_tokens(
                user_info['user_id'],
                actual_tokens,
                request_body.get('model', 'unknown')
            )

            # Verificar se precisa enviar alertas
            await self.check_and_send_alerts(user_info['user_id'])

        return response

    async def estimate_tokens(self, request_body: Dict[str, Any]) -> int:
        """Estimar tokens necessários baseado na requisição"""

        # Implementação simplificada de estimativa de tokens
        messages = request_body.get('messages', [])
        total_chars = sum(len(msg.get('content', '')) for msg in messages)

        # Estimativa conservadora: 1 token ≈ 4 caracteres
        estimated_input_tokens = total_chars // 4

        # Estimar tokens de saída baseado em max_tokens ou padrão
        max_tokens = request_body.get('max_tokens', 1000)
        estimated_output_tokens = min(max_tokens, estimated_input_tokens * 2)

        return estimated_input_tokens + estimated_output_tokens

    async def handle_insufficient_tokens(self, user_info: Dict, token_check:
Dict):
        """Lidar com usuários sem tokens suficientes"""

        # Verificar se usuário está completamente bloqueado
        if token_check['total_available'] <= 0:
            # Enviar para página de compra de créditos
            purchase_url = await self.billing_service.generate_purchase_url(
                user_info['user_id']
            )

            return {
                "error": "tokens_exhausted",
                "message": "Seus tokens foram esgotados. Compre créditos
adicionais para continuar.",
                "purchase_url": purchase_url,
                "available_packages": await
self.billing_service.get_credit_packages()
            }
```

```
        else:
            # Usuário tem alguns tokens, mas não suficientes para esta
requisição
            return {
                "error": "insufficient_tokens",
                "message": f"Tokens insuficientes para esta requisição.
Disponível: {token_check['total_available']}",
                "available_tokens": token_check['total_available'],
                "estimated_needed": token_check['estimated_needed']
            }

    async def forward_to_openai(self, request: Request, path: str, body:
Dict):
        """Encaminhar requisição para OpenAI"""

        headers = dict(request.headers)
        headers['authorization'] = f"Bearer {self.get_openai_api_key()}"

        response = await self.openai_client.post(
            f"/{path}",
            json=body,
            headers=headers
        )

        return response

    async def check_and_send_alerts(self, user_id: str):
        """Verificar se precisa enviar alertas de consumo"""

        user_status = await self.token_service.get_user_status(user_id)
        usage_percentage = user_status['used'] / user_status['total_limit']

        # Alertas em 80% e 90%
        if usage_percentage >= 0.8 and not user_status.get('alert_80_sent'):
            await self.send_usage_alert(user_id, 80)
            await self.token_service.mark_alert_sent(user_id, '80')

        if usage_percentage >= 0.9 and not user_status.get('alert_90_sent'):
            await self.send_usage_alert(user_id, 90)
            await self.token_service.mark_alert_sent(user_id, '90')

        # Bloqueio automático em 100%
        if usage_percentage >= 1.0:
            await self.token_service.block_user(user_id)
            await self.send_exhaustion_notification(user_id)

class TokenService:
    """Serviço para gerenciamento de tokens"""
```

```
    async def check_user_tokens(self, user_id: str, estimated_tokens: int) ->
Dict:
        """Verificar tokens disponíveis para um usuário"""

        # Buscar dados do usuário
        user_data = await self.get_user_data(user_id)

        # Aplicar regras de negócio da IA SOLARIS
        monthly_available = user_data['monthly_limit'] -
user_data['monthly_used']
        additional_available = user_data['additional_credits']
        total_available = monthly_available + additional_available

        return {
            'can_proceed': total_available >= estimated_tokens,
            'total_available': total_available,
            'monthly_available': monthly_available,
            'additional_available': additional_available,
            'estimated_needed': estimated_tokens
        }

    async def consume_tokens(self, user_id: str, actual_tokens: int, model:
str):
        """Registrar consumo de tokens"""

        user_data = await self.get_user_data(user_id)

        # Aplicar regra de prioridade: consumir tokens mensais primeiro
        monthly_available = user_data['monthly_limit'] -
user_data['monthly_used']

        if monthly_available >= actual_tokens:
            # Consumir apenas dos tokens mensais
            await self.update_monthly_usage(user_id, actual_tokens)
        else:
            # Consumir todos os tokens mensais e o restante dos créditos
adicionais
            if monthly_available > 0:
                await self.update_monthly_usage(user_id, monthly_available)
                remaining = actual_tokens - monthly_available
                await self.update_additional_credits(user_id, -remaining)
            else:
                # Consumir apenas dos créditos adicionais
                await self.update_additional_credits(user_id, -actual_tokens)
```

```
        # Registrar transação para auditoria
        await self.record_transaction(user_id, actual_tokens, model)
```

Comparação Quantitativa das Abordagens

Para fornecer uma análise objetiva, é essencial comparar quantitativamente as diferentes abordagens considerando métricas específicas relevantes para o contexto da IA SOLARIS. Esta comparação inclui aspectos como complexidade de implementação, custo de desenvolvimento, tempo de implementação, e risco técnico.

Critério	LiteLLM Puro	Microserviços	Proxy Inteligente	Customização LibreChat
Complexidade de Implementação	Baixa (2/10)	Alta (8/10)	Média (6/10)	Alta (7/10)
Flexibilidade de Regras	Baixa (3/10)	Muito Alta (10/10)	Alta (8/10)	Muito Alta (9/10)
Manutenibilidade	Alta (9/10)	Média (6/10)	Alta (8/10)	Baixa (4/10)
Escalabilidade	Alta (9/10)	Muito Alta (10/10)	Alta (8/10)	Média (6/10)
Precisão de Tokens	Muito Alta (10/10)	Alta (8/10)	Alta (8/10)	Média (6/10)
Risco de Vendor Lock-in	Médio (5/10)	Baixo (2/10)	Baixo (3/10)	Alto (8/10)
Facilidade de Debugging	Alta (8/10)	Baixa (4/10)	Média (6/10)	Baixa (4/10)
Compatibilidade Futura	Alta (9/10)	Alta (8/10)	Alta (8/10)	Baixa (3/10)

Overhead Operacional	Baixo (2/10)	Alto (9/10)	Médio (5/10)	Médio (6/10)

Análise de Cenários de Uso

Diferentes cenários de uso podem favorecer diferentes abordagens. Para startups ou projetos com recursos limitados, a abordagem LiteLLM puro pode ser mais apropriada devido ao menor custo e complexidade. Para empresas estabelecidas com equipes técnicas robustas, a arquitetura de microserviços pode oferecer maior flexibilidade e escalabilidade.

O proxy inteligente representa um meio-termo interessante, oferecendo flexibilidade significativa sem a complexidade operacional dos microserviços. Esta abordagem pode ser particularmente adequada para organizações que querem implementar controles sofisticados mantendo a simplicidade operacional.

A customização do LibreChat pode ser justificada apenas em cenários onde os requisitos de negócio são tão específicos que não podem ser atendidos por nenhuma outra abordagem, e onde a organização tem recursos dedicados para manutenção contínua das customizações.

Considerações de Performance e Latência

A performance é um fator crítico que pode impactar significativamente a experiência do usuário. Cada abordagem introduz diferentes overheads que devem ser cuidadosamente considerados.

A abordagem LiteLLM puro introduz latência mínima, pois opera como proxy otimizado com caching e connection pooling. A arquitetura de microserviços pode introduzir latência significativa devido à comunicação entre serviços, especialmente se não for adequadamente otimizada.

O proxy inteligente adiciona uma camada de processamento que pode impactar a latência, mas oferece oportunidades para otimizações como caching de verificações de tokens e batching de operações. A customização do LibreChat pode ter impacto variável na performance dependendo da qualidade da implementação.

```
Python
```

```
# Exemplo de análise de performance para diferentes abordagens
import asyncio
import time
from typing import Dict, List

class PerformanceAnalyzer:
    def __init__(self):
        self.metrics = {}

    async def benchmark_litellm_approach(self, num_requests: int = 100) ->
Dict:
        """Benchmark da abordagem LiteLLM puro"""

        start_time = time.time()
        latencies = []

        for i in range(num_requests):
            request_start = time.time()

            # Simular verificação de tokens no LiteLLM
            await self.simulate_litellm_token_check()

            # Simular requisição para OpenAI
            await self.simulate_openai_request()

            latency = time.time() - request_start
            latencies.append(latency)

        total_time = time.time() - start_time

        return {
            'approach': 'LiteLLM Puro',
            'total_time': total_time,
            'avg_latency': sum(latencies) / len(latencies),
            'p95_latency': sorted(latencies)[int(0.95 * len(latencies))],
            'p99_latency': sorted(latencies)[int(0.99 * len(latencies))],
            'throughput': num_requests / total_time
        }

    async def benchmark_microservices_approach(self, num_requests: int = 100)
-> Dict:
        """Benchmark da abordagem de microserviços"""

        start_time = time.time()
        latencies = []

        for i in range(num_requests):
```

```
            request_start = time.time()

            # Simular chamada para Token Management Service
            await self.simulate_service_call('token-management', 50)

            # Simular chamada para User Billing Service
            await self.simulate_service_call('billing', 30)

            # Simular requisição para OpenAI
            await self.simulate_openai_request()

            # Simular registro de consumo
            await self.simulate_service_call('token-management', 25)

            latency = time.time() - request_start
            latencies.append(latency)

        total_time = time.time() - start_time

        return {
            'approach': 'Microserviços',
            'total_time': total_time,
            'avg_latency': sum(latencies) / len(latencies),
            'p95_latency': sorted(latencies)[int(0.95 * len(latencies))],
            'p99_latency': sorted(latencies)[int(0.99 * len(latencies))],
            'throughput': num_requests / total_time
        }

    async def benchmark_proxy_approach(self, num_requests: int = 100) ->
Dict:
        """Benchmark da abordagem de proxy inteligente"""

        start_time = time.time()
        latencies = []

        for i in range(num_requests):
            request_start = time.time()

            # Simular processamento no proxy
            await self.simulate_proxy_processing()

            # Simular requisição para OpenAI
            await self.simulate_openai_request()

            # Simular pós-processamento
            await self.simulate_proxy_postprocessing()

            latency = time.time() - request_start
```

```
            latencies.append(latency)

        total_time = time.time() - start_time

        return {
            'approach': 'Proxy Inteligente',
            'total_time': total_time,
            'avg_latency': sum(latencies) / len(latencies),
            'p95_latency': sorted(latencies)[int(0.95 * len(latencies))],
            'p99_latency': sorted(latencies)[int(0.99 * len(latencies))],
            'throughput': num_requests / total_time
        }

    async def simulate_litellm_token_check(self):
        """Simular verificação de tokens no LiteLLM"""
        await asyncio.sleep(0.010)  # 10ms

    async def simulate_openai_request(self):
        """Simular requisição para OpenAI"""
        await asyncio.sleep(0.800)  # 800ms (tempo típico de resposta)

    async def simulate_service_call(self, service: str, latency_ms: int):
        """Simular chamada para microserviço"""
        await asyncio.sleep(latency_ms / 1000)

    async def simulate_proxy_processing(self):
        """Simular processamento no proxy"""
        await asyncio.sleep(0.020)  # 20ms

    async def simulate_proxy_postprocessing(self):
        """Simular pós-processamento no proxy"""
        await asyncio.sleep(0.015)  # 15ms

    async def run_comprehensive_benchmark(self) -> Dict:
        """Executar benchmark abrangente de todas as abordagens"""

        results = {}

        print("Executando benchmark LiteLLM...")
        results['litellm'] = await self.benchmark_litellm_approach()

        print("Executando benchmark Microserviços...")
        results['microservices'] = await
self.benchmark_microservices_approach()

        print("Executando benchmark Proxy...")
        results['proxy'] = await self.benchmark_proxy_approach()
```

```
        return results

# Exemplo de uso
async def main():
    analyzer = PerformanceAnalyzer()
    results = await analyzer.run_comprehensive_benchmark()

    print("\n=== RESULTADOS DO BENCHMARK ===")
    for approach, metrics in results.items():
        print(f"\n{metrics['approach']}:")
        print(f"  Latência Média: {metrics['avg_latency']:.3f}s")
        print(f"  P95 Latência: {metrics['p95_latency']:.3f}s")
        print(f"  P99 Latência: {metrics['p99_latency']:.3f}s")
        print(f"  Throughput: {metrics['throughput']:.1f} req/s")

# Resultados esperados:
# LiteLLM Puro: ~810ms latência média, ~12 req/s throughput
# Microserviços: ~905ms latência média, ~11 req/s throughput
# Proxy Inteligente: ~835ms latência média, ~12 req/s throughput
```

Recomendação Técnica Fundamentada

Após análise abrangente das diferentes abordagens disponíveis para implementar controle de tokens por usuário na IA SOLARIS, a recomendação técnica deve considerar não apenas aspectos puramente técnicos, mas também o contexto organizacional, recursos disponíveis, e objetivos de negócio específicos.

Análise do Contexto Específico da IA SOLARIS

O contexto da IA SOLARIS apresenta características únicas que influenciam significativamente a escolha da abordagem mais adequada. A plataforma já possui uma base de usuários estabelecida utilizando LibreChat, o que significa que qualquer solução deve minimizar disruptions na experiência do usuário existente. Adicionalmente, os requisitos específicos de negócio, incluindo o sistema de créditos acumulativos e diferentes tiers de assinatura, adicionam complexidade que nem todas as abordagens podem acomodar adequadamente.

A equipe técnica disponível e sua experiência com diferentes tecnologias também é um fator determinante. Uma solução que requer expertise em microserviços e orquestração de

containers pode não ser viável se a equipe não possui essa experiência, independentemente dos benefícios técnicos teóricos.

O orçamento disponível e timeline para implementação são considerações práticas que podem eliminar certas opções. Uma solução que requer 12 semanas de desenvolvimento pode não ser viável se existe pressão para implementar controles de custo rapidamente devido a gastos crescentes com APIs.

Recomendação Primária: Abordagem Híbrida LiteLLM + Proxy Inteligente

Baseado na análise completa, a recomendação primária é uma abordagem híbrida que combina as forças do LiteLLM para precisão de contabilização com um proxy inteligente para implementar regras de negócio específicas. Esta abordagem oferece o melhor equilíbrio entre funcionalidade, complexidade, e viabilidade de implementação.

A arquitetura híbrida recomendada utiliza o LiteLLM como sistema principal de contabilização e controle de tokens, aproveitando sua precisão e integração nativa com provedores de IA. Um proxy inteligente customizado é implementado como camada adicional para lidar com regras de negócio específicas que não são suportadas nativamente pelo LiteLLM.

Python

```
# Arquitetura híbrida recomendada para IA SOLARIS
from fastapi import FastAPI, HTTPException, Request
from fastapi.middleware.cors import CORSMiddleware
import httpx
import asyncio
from typing import Dict, Optional
import json
from datetime import datetime, timedelta

class IASOLARISTokenProxy:
    """
    Proxy inteligente que combina LiteLLM para precisão de tokens
    com lógica customizada para regras de negócio específicas
    """

    def __init__(self):
        self.app = FastAPI(title="IA SOLARIS Token Control Proxy")
```

```
        self.litellm_client = httpx.AsyncClient(
            base_url="http://litellm:4000",
            timeout=httpx.Timeout(60.0)
        )
        self.setup_middleware()
        self.setup_routes()

    def setup_middleware(self):
        """Configurar middleware necessário"""
        self.app.add_middleware(
            CORSMiddleware,
            allow_origins=["*"],
            allow_credentials=True,
            allow_methods=["*"],
            allow_headers=["*"],
        )

    def setup_routes(self):
        """Configurar rotas do proxy"""

        @self.app.middleware("http")
        async def token_control_middleware(request: Request, call_next):
            """Middleware para controle de tokens em todas as requisições"""

            # Verificar se é requisição que consome tokens
            if self.requires_token_control(request):
                # Aplicar controle de tokens antes de prosseguir
                control_result = await self.apply_token_control(request)

                if not control_result['allowed']:
                    return self.create_blocked_response(control_result)

            # Prosseguir com a requisição
            response = await call_next(request)

            # Pós-processamento se necessário
            if self.requires_token_control(request) and response.status_code
== 200:
                await self.post_process_token_usage(request, response)

            return response

        @self.app.get("/health")
        async def health_check():
            """Verificação de saúde do proxy"""
            return {"status": "healthy", "timestamp": datetime.utcnow()}

        @self.app.get("/user/{user_id}/token-status")
```

```
        async def get_user_token_status(user_id: str):
            """Obter status detalhado de tokens do usuário"""
            try:
                status = await self.get_comprehensive_token_status(user_id)
                return status
            except Exception as e:
                raise HTTPException(status_code=500, detail=str(e))

        @self.app.post("/admin/user/{user_id}/add-credits")
        async def add_user_credits(user_id: str, credits: Dict):
            """Adicionar créditos para um usuário (uso administrativo)"""
            try:
                result = await self.add_additional_credits(
                    user_id,
                    credits['amount'],
                    credits.get('reason', 'admin_addition')
                )
                return result
            except Exception as e:
                raise HTTPException(status_code=500, detail=str(e))

    def requires_token_control(self, request: Request) -> bool:
        """Determinar se a requisição requer controle de tokens"""

        # Verificar se é requisição para endpoints de chat/completion
        token_consuming_paths = [
            '/v1/chat/completions',
            '/v1/completions',
            '/chat/completions',
            '/completions'
        ]

        return any(path in str(request.url) for path in
token_consuming_paths)

    async def apply_token_control(self, request: Request) -> Dict:
        """Aplicar controle de tokens específico da IA SOLARIS"""

        # Extrair informações do usuário
        user_info = await self.extract_user_from_request(request)

        if not user_info:
            return {'allowed': False, 'reason': 'user_not_authenticated'}

        # Obter dados atuais do usuário no LiteLLM
        litellm_data = await self.get_litellm_user_data(user_info['user_id'])

        # Obter dados de créditos adicionais do sistema customizado
```

```
        additional_data = await
self.get_additional_credits_data(user_info['user_id'])

        # Estimar tokens necessários para a requisição
        request_body = await self.get_request_body(request)
        estimated_tokens = await
self.estimate_token_consumption(request_body)

        # Aplicar regras de negócio específicas da IA SOLARIS
        control_result = await self.apply_iasolaris_business_rules(
            user_info,
            litellm_data,
            additional_data,
            estimated_tokens
        )

        return control_result

    async def apply_iasolaris_business_rules(self, user_info: Dict,
                                             litellm_data: Dict,
                                             additional_data: Dict,
                                             estimated_tokens: int) -> Dict:
        """Aplicar regras de negócio específicas da IA SOLARIS"""

        # Regra 1: Calcular tokens mensais disponíveis
        monthly_budget_usd = litellm_data.get('budget_limit', 25.0)
        monthly_used_usd = litellm_data.get('budget_used', 0.0)
        monthly_remaining_usd = monthly_budget_usd - monthly_used_usd

        # Converter USD para tokens (aproximadamente $25 = 1M tokens)
        monthly_remaining_tokens = int(monthly_remaining_usd * 40000)  # 40k
tokens por USD

        # Regra 2: Obter créditos adicionais (acumulativos)
        additional_tokens = additional_data.get('balance', 0)

        # Regra 3: Calcular total disponível
        total_available = monthly_remaining_tokens + additional_tokens

        # Regra 4: Verificar se pode prosseguir
        can_proceed = total_available >= estimated_tokens

        # Regra 5: Determinar fonte de consumo (prioridade: mensal primeiro)
        if can_proceed:
            if monthly_remaining_tokens >= estimated_tokens:
                consumption_source = 'monthly'
                monthly_consumption = estimated_tokens
                additional_consumption = 0
```

```
            else:
                consumption_source = 'mixed'
                monthly_consumption = max(0, monthly_remaining_tokens)
                additional_consumption = estimated_tokens -
monthly_consumption
        else:
            consumption_source = 'insufficient'
            monthly_consumption = 0
            additional_consumption = 0

        # Regra 6: Verificar alertas necessários
        usage_percentage = (monthly_used_usd / monthly_budget_usd) if
monthly_budget_usd > 0 else 0
        alerts_needed = []

        if usage_percentage >= 0.8 and not
additional_data.get('alert_80_sent'):
            alerts_needed.append('80_percent')
        if usage_percentage >= 0.9 and not
additional_data.get('alert_90_sent'):
            alerts_needed.append('90_percent')
        if not can_proceed:
            alerts_needed.append('exhausted')

        return {
            'allowed': can_proceed,
            'total_available': total_available,
            'monthly_remaining': monthly_remaining_tokens,
            'additional_available': additional_tokens,
            'estimated_consumption': estimated_tokens,
            'consumption_plan': {
                'source': consumption_source,
                'monthly_consumption': monthly_consumption,
                'additional_consumption': additional_consumption
            },
            'alerts_needed': alerts_needed,
            'user_id': user_info['user_id']
        }

    async def get_litellm_user_data(self, user_id: str) -> Dict:
        """Obter dados do usuário do LiteLLM"""
        try:
            response = await self.litellm_client.get(
                f"/user/info",
                params={"user_id": user_id}
            )

            if response.status_code == 200:
```

```
                return response.json()
            else:
                # Usuário não existe no LiteLLM, criar
                return await self.create_litellm_user(user_id)

        except Exception as e:
            print(f"Erro ao obter dados do LiteLLM: {e}")
            return {'budget_limit': 25.0, 'budget_used': 0.0}

    async def create_litellm_user(self, user_id: str) -> Dict:
        """Criar usuário no LiteLLM se não existir"""

        # Obter dados do usuário do LibreChat
        user_data = await self.get_librechat_user_data(user_id)

        # Determinar budget baseado no tier de assinatura
        budget_mapping = {
            'basic': 25.0,      # 1M tokens
            'premium': 50.0,    # 2M tokens
            'enterprise': 100.0 # 4M tokens
        }

        budget = budget_mapping.get(user_data.get('subscription_tier',
'basic'), 25.0)

        litellm_user_data = {
            "user_id": user_id,
            "user_email": user_data.get('email', ''),
            "budget_limit": budget,
            "budget_duration": "1mo",
            "models": ["gpt-4", "gpt-3.5-turbo"],
            "metadata": {
                "created_by": "iasolaris_proxy",
                "subscription_tier": user_data.get('subscription_tier',
'basic')
            }
        }

        try:
            response = await self.litellm_client.post(
                "/user/new",
                json=litellm_user_data
            )

            if response.status_code == 200:
                return response.json()
            else:
                raise Exception(f"Falha ao criar usuário no LiteLLM:
```

```
{response.text}")

        except Exception as e:
            print(f"Erro ao criar usuário no LiteLLM: {e}")
            return {'budget_limit': budget, 'budget_used': 0.0}

    async def post_process_token_usage(self, request: Request, response):
        """Pós-processar uso de tokens após resposta bem-sucedida"""

        try:
            # Extrair informações do usuário
            user_info = await self.extract_user_from_request(request)

            if not user_info:
                return

            # Obter dados reais de uso da resposta
            response_data = await self.extract_response_data(response)
            actual_tokens = response_data.get('usage',
{}).get('total_tokens', 0)

            if actual_tokens > 0:
                # Registrar consumo no sistema de créditos adicionais se
necessário
                await self.process_additional_credits_consumption(
                    user_info['user_id'],
                    actual_tokens
                )

                # Enviar alertas se necessário
                await self.check_and_send_alerts(user_info['user_id'])

        except Exception as e:
            print(f"Erro no pós-processamento: {e}")

    async def get_additional_credits_data(self, user_id: str) -> Dict:
        """Obter dados de créditos adicionais do sistema customizado"""

        # Esta função se conectaria ao banco de dados customizado
        # para obter informações sobre créditos adicionais

        # Implementação simplificada para exemplo
        return {
            'balance': 0,  # Tokens de créditos adicionais disponíveis
            'total_purchased': 0,  # Total de créditos já comprados
            'alert_80_sent': False,
            'alert_90_sent': False,
            'last_purchase_date': None
```

```
        }

    async def estimate_token_consumption(self, request_body: Dict) -> int:
        """Estimar consumo de tokens baseado na requisição"""

        if not request_body:
            return 1000  # Estimativa conservadora padrão

        # Implementação simplificada de estimativa
        messages = request_body.get('messages', [])
        total_chars = sum(len(str(msg.get('content', ''))) for msg in
messages)

        # Estimativa: 1 token ≈ 4 caracteres para input
        input_tokens = total_chars // 4

        # Estimativa de output baseada em max_tokens ou padrão
        max_tokens = request_body.get('max_tokens', 1000)
        output_tokens = min(max_tokens, input_tokens * 2)

        return input_tokens + output_tokens

# Configuração de deployment da arquitetura híbrida
class IASOLARISDeploymentConfig:
    """Configuração de deployment para a arquitetura híbrida"""

    @staticmethod
    def generate_docker_compose() -> str:
        """Gerar configuração Docker Compose para deployment"""

        return """
version: '3.8'

services:
  # LibreChat (mantido como está)
  librechat:
    image: librechat:latest
    environment:
      - ENDPOINTS=openAI
      - OPENAI_API_KEY=${OPENAI_API_KEY}
      - PROXY_URL=http://iasolaris-proxy:8000
    depends_on:
      - mongodb
      - iasolaris-proxy
    networks:
      - iasolaris-network

  # LiteLLM para precisão de contabilização
```

```
  litellm:
    image: ghcr.io/berriai/litellm:main-latest
    environment:
      - DATABASE_URL=postgresql://user:pass@postgres:5432/litellm
      - REDIS_URL=redis://redis:6379
    volumes:
      - ./litellm.config.yaml:/app/config.yaml
    command: ["--config", "/app/config.yaml", "--port", "4000"]
    depends_on:
      - postgres
      - redis
    networks:
      - iasolaris-network

  # Proxy inteligente customizado
  iasolaris-proxy:
    build: ./iasolaris-proxy
    environment:
      - LITELLM_URL=http://litellm:4000
      - MONGODB_URL=mongodb://mongodb:27017/librechat
      - POSTGRES_URL=postgresql://user:pass@postgres:5432/iasolaris
      - REDIS_URL=redis://redis:6379
    depends_on:
      - litellm
      - postgres
      - redis
    networks:
      - iasolaris-network
    ports:
      - "8000:8000"

  # Infraestrutura
  postgres:
    image: postgres:14
    environment:
      - POSTGRES_DB=iasolaris
      - POSTGRES_USER=user
      - POSTGRES_PASSWORD=pass
    volumes:
      - postgres_data:/var/lib/postgresql/data
    networks:
      - iasolaris-network

  redis:
    image: redis:7
    networks:
      - iasolaris-network
```

```
  mongodb:
    image: mongo:6
    volumes:
      - mongodb_data:/data/db
    networks:
      - iasolaris-network

volumes:
  postgres_data:
  mongodb_data:

networks:
  iasolaris-network:
    driver: bridge
"""

    @staticmethod
    def generate_litellm_config() -> str:
        """Gerar configuração do LiteLLM"""

        return """
model_list:
  - model_name: gpt-4
    litellm_params:
      model: openai/gpt-4
      api_key: os.environ/OPENAI_API_KEY
      max_tokens: 4096

  - model_name: gpt-3.5-turbo
    litellm_params:
      model: openai/gpt-3.5-turbo
      api_key: os.environ/OPENAI_API_KEY
      max_tokens: 4096

litellm_settings:
  database_url: os.environ/DATABASE_URL
  redis_host: redis
  redis_port: 6379

  # Callbacks para logging e monitoramento
  success_callback: ["langfuse"]
  failure_callback: ["sentry"]

  # Rate limiting global
  rpm_limit: 1000
  tpm_limit: 100000

router_settings:
```

```
  enable_user_auth: true
  enable_budget_tracking: true
  enable_fallbacks: true

general_settings:
  # Budget padrão para novos usuários
  default_user_budget: 25.0
  budget_duration: "1mo"

  # Configurações de UI administrativa
  ui_username: os.environ/LITELLM_UI_USERNAME
  ui_password: os.environ/LITELLM_UI_PASSWORD

  # Webhooks para integração com sistema customizado
  alert_webhooks:
    - url: "http://iasolaris-proxy:8000/webhooks/litellm-alert"
      events: ["budget_crossed_80", "budget_crossed_90", "budget_exceeded"]
"""
```

Justificativa da Recomendação Híbrida

A abordagem híbrida recomendada oferece vantagens significativas que abordam diretamente as limitações identificadas em cada abordagem individual. Ao utilizar o LiteLLM como sistema principal de contabilização, a solução garante precisão máxima na contagem de tokens, eliminando as discrepâncias conhecidas entre sistemas como LibreChat e dados reais da OpenAI.

O proxy inteligente customizado permite implementar regras de negócio específicas da IA SOLARIS que não são suportadas nativamente pelo LiteLLM. Isso inclui o sistema de créditos acumulativos, diferentes tiers de assinatura, e lógica de priorização de consumo (tokens mensais antes de créditos adicionais).

A arquitetura mantém a simplicidade operacional evitando a complexidade de microserviços distribuídos, enquanto oferece flexibilidade suficiente para implementar funcionalidades avançadas. O proxy opera como uma camada transparente que não requer modificações no LibreChat existente, minimizando riscos de disruption.