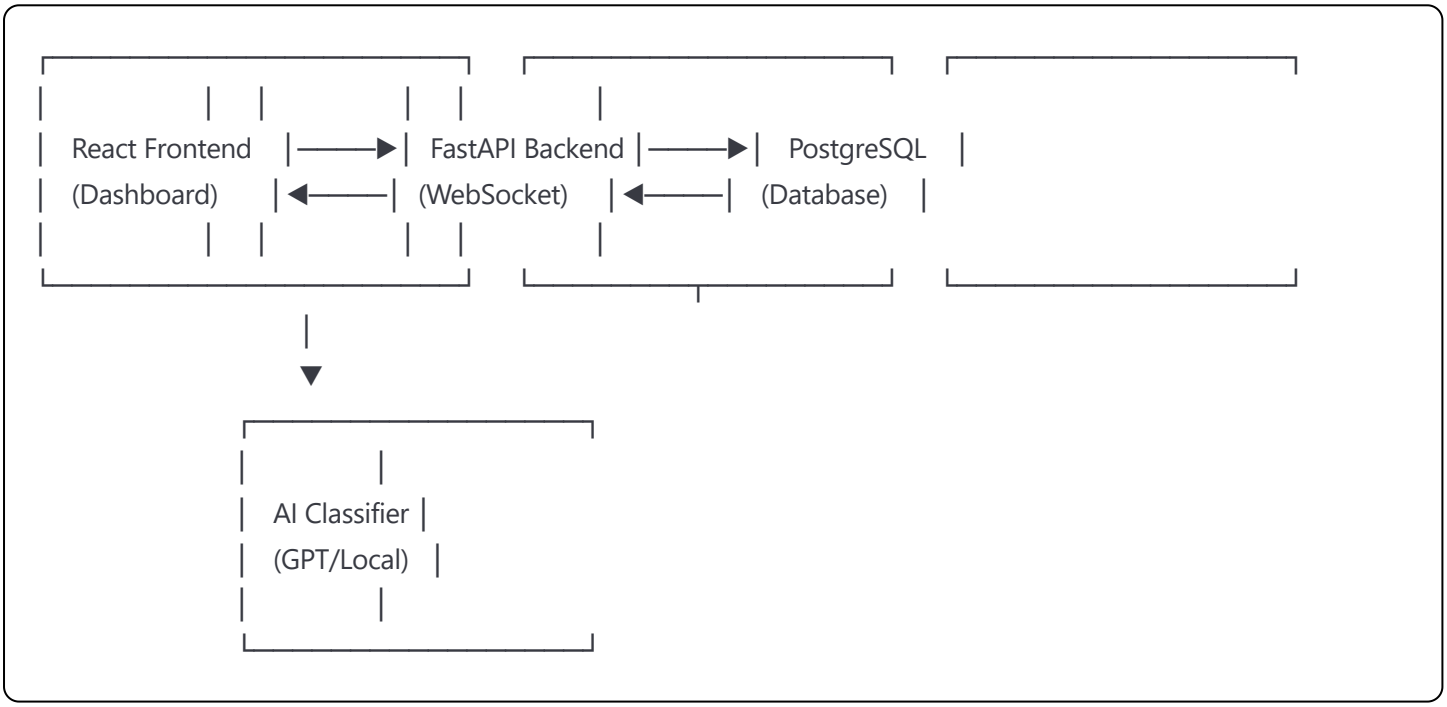
🎛️ Painel de Controle para Secretaria - Chatbot Clínica

📋 Visão Geral

O painel de controle permite que a secretaria:

- Visualize todas as conversas em tempo real
- Revise agendamentos feitos pelo bot
- Classifique conversas com etiquetas inteligentes
- Intervenha quando necessário
- Monitore métricas e qualidade

📐 Arquitetura do Painel



🔧 Implementação Backend

1. Modelos Adicionais (models/dashboard.py)

```python
```

```python
from sqlalchemy import Column, String, DateTime, JSON, Boolean, Integer, Enum, ForeignKey
from sqlalchemy.orm import relationship
from datetime import datetime
import enum
from app.models.database import Base

class ConversationStatus(enum.Enum):
    PENDING = "pending"          # Aguardando revisão
    IN_PROGRESS = "in_progress"  # Em atendimento
    COMPLETED = "completed"      # Finalizada
    REQUIRES_ATTENTION = "requires_attention"  # Precisa atenção
    SPAM = "spam"                # Spam/Irrelevante

class ConversationTag(enum.Enum):
    AGENDAMENTO_SUCESSO = "agendamento_sucesso"
    AGENDAMENTO_ERRO = "agendamento_erro"
    CANCELAMENTO = "cancelamento"
    LISTA_ESPERA = "lista_espera"
    DUVIDA = "duvida"
    RECLAMACAO = "reclamacao"
    NOVO_PACIENTE = "novo_paciente"
    URGENTE = "urgente"
    REAGENDAMENTO = "reagendamento"
    CONFIRMACAO = "confirmacao"

class ConversationDashboard(Base):
    __tablename__ = "conversation_dashboard"

    id = Column(String, primary_key=True)
    conversation_id = Column(String, ForeignKey("conversations.id"))
    phone = Column(String, nullable=False)
    patient_name = Column(String)
    patient_cpf = Column(String)

    # Status e classificação
    status = Column(Enum(ConversationStatus), default=ConversationStatus.PENDING)
    tags = Column(JSON, default=[])
    priority = Column(Integer, default=0)  # 0-baixa, 1-média, 2-alta, 3-urgente

    # Análise de sentimento
    sentiment_score = Column(Integer)  # -100 a 100
    ai_summary = Column(String)
    ai_suggested_action = Column(String)

    # Métricas
    message_count = Column(Integer, default=0)
```

```python
    bot_resolution = Column(Boolean)
    human_intervention = Column(Boolean, default=False)
    resolution_time = Column(Integer)  # em minutos

    # Timestamps
    first_message_at = Column(DateTime)
    last_message_at = Column(DateTime)
    reviewed_at = Column(DateTime)
    reviewed_by = Column(String)

    # Relacionamentos
    messages = relationship("ConversationMessage", back_populates="dashboard")
    notes = relationship("ConversationNote", back_populates="dashboard")

class ConversationMessage(Base):
    __tablename__ = "conversation_messages"

    id = Column(String, primary_key=True)
    dashboard_id = Column(String, ForeignKey("conversation_dashboard.id"))

    sender = Column(String)  # "user" ou "bot"
    message = Column(String)
    timestamp = Column(DateTime, default=datetime.utcnow)
    message_type = Column(String)  # text, image, audio, etc

    dashboard = relationship("ConversationDashboard", back_populates="messages")

class ConversationNote(Base):
    __tablename__ = "conversation_notes"

    id = Column(String, primary_key=True)
    dashboard_id = Column(String, ForeignKey("conversation_dashboard.id"))

    note = Column(String)
    created_by = Column(String)
    created_at = Column(DateTime, default=datetime.utcnow)

    dashboard = relationship("ConversationDashboard", back_populates="notes")
```

2. Sistema de Classificação Inteligente (services/classifier.py)

```python
```

```python
import re
from typing import List, Dict, Tuple
from datetime import datetime
import httpx
from app.config import settings
import logging

logger = logging.getLogger(__name__)

class ConversationClassifier:
    def __init__(self):
        self.patterns = self._load_patterns()

    def _load_patterns(self) -> Dict:
        """Carrega padrões para classificação baseada em regras"""
        return {
            'agendamento_sucesso': [
                r'consulta agendada com sucesso',
                r'agendamento confirmado',
                r'confirmou.*agendamento'
            ],
            'agendamento_erro': [
                r'erro ao agendar',
                r'não.*consegui.*agendar',
                r'problema.*agendamento'
            ],
            'cancelamento': [
                r'cancelar.*consulta',
                r'desmarcar',
                r'não.*poder.*comparecer'
            ],
            'urgente': [
                r'urgente',
                r'emergência',
                r'dor.*forte',
                r'preciso.*hoje'
            ],
            'reclamacao': [
                r'reclamar',
                r'insatisfeito',
                r'péssimo',
                r'horrível',
                r'demora'
            ],
            'novo_paciente': [
                r'primeira vez',
```

```
            r'novo paciente',
            r'não.*cadastrado',
            r'realizar.*cadastro'
        ],
        'duvida': [
            r'quanto custa',
            r'qual.*valor',
            r'convênio',
            r'aceita.*plano',
            r'como.*funciona'
        ]
    }

async def analyze_conversation(self, messages: List[Dict]) -> Dict:
    """Analisa conversa completa e retorna classificação"""

    # Juntar todas as mensagens
    full_text = " ".join([msg['message'].lower() for msg in messages])

    # Análise baseada em regras
    tags = self._extract_tags(full_text)
    priority = self._calculate_priority(tags, messages)
    sentiment = self._analyze_sentiment(messages)

    # Análise com IA (se configurado)
    ai_analysis = await self._ai_analysis(messages) if settings.use_ai_classifier else {}

    # Resumo e ação sugerida
    summary = ai_analysis.get('summary', self._generate_summary(messages))
    suggested_action = ai_analysis.get('action', self._suggest_action(tags))

    return {
        'tags': tags,
        'priority': priority,
        'sentiment_score': sentiment,
        'ai_summary': summary,
        'ai_suggested_action': suggested_action,
        'requires_attention': self._requires_human_attention(tags, sentiment)
    }

def _extract_tags(self, text: str) -> List[str]:
    """Extrai tags baseadas em padrões"""
    tags = []

    for tag, patterns in self.patterns.items():
        for pattern in patterns:
            if re.search(pattern, text, re.IGNORECASE):
```

```
                tags.append(tag)
                break

    return list(set(tags))

def _calculate_priority(self, tags: List[str], messages: List[Dict]) -> int:
    """Calcula prioridade da conversa"""

    # Urgente
    if 'urgente' in tags or 'reclamacao' in tags:
        return 3

    # Alta - problemas ou novos pacientes
    if 'agendamento_erro' in tags or 'novo_paciente' in tags:
        return 2

    # Média - ações pendentes
    if 'cancelamento' in tags or 'reagendamento' in tags:
        return 1

    # Baixa - sucesso ou dúvidas simples
    return 0

def _analyze_sentiment(self, messages: List[Dict]) -> int:
    """Analisa sentimento da conversa (-100 a 100)"""

    positive_words = ['obrigado', 'obrigada', 'ótimo', 'excelente', 'perfeito', 'maravilhoso']
    negative_words = ['ruim', 'péssimo', 'horrível', 'demora', 'problema', 'erro', 'cancelar']

    score = 0
    for msg in messages:
        if msg['sender'] == 'user':
            text = msg['message'].lower()

            for word in positive_words:
                if word in text:
                    score += 20

            for word in negative_words:
                if word in text:
                    score -= 20

    return max(-100, min(100, score))

def _generate_summary(self, messages: List[Dict]) -> str:
    """Gera resumo simples da conversa"""
    if not messages:
```

```
            return "Conversa vazia"

        user_messages = [m for m in messages if m['sender'] == 'user']

        if len(user_messages) == 0:
            return "Sem mensagens do usuário"

        first_msg = user_messages[0]['message'][:100]
        return f"Conversa iniciada com: {first_msg}..."

    def _suggest_action(self, tags: List[str]) -> str:
        """Sugere ação baseada nas tags"""

        if 'agendamento_erro' in tags:
            return "Verificar problema no agendamento e entrar em contato"

        if 'reclamacao' in tags:
            return "Priorizar atendimento e resolver insatisfação"

        if 'novo_paciente' in tags:
            return "Orientar sobre processo de cadastro"

        if 'cancelamento' in tags:
            return "Confirmar cancelamento e liberar horário"

        if 'urgente' in tags:
            return "Atendimento imediato necessário"

        return "Monitorar conversa"

    def _requires_human_attention(self, tags: List[str], sentiment: int) -> bool:
        """Determina se precisa de atenção humana"""

        attention_tags = ['urgente', 'reclamacao', 'agendamento_erro', 'novo_paciente']

        # Precisa atenção se tem tags críticas ou sentimento muito negativo
        return any(tag in attention_tags for tag in tags) or sentiment < -50

    async def _ai_analysis(self, messages: List[Dict]) -> Dict:
        """Análise usando IA (OpenAI/Local)"""
        try:
            # Preparar contexto
            conversation = "\n".join([
                f"{msg['sender']}: {msg['message']}"
                for msg in messages[-10:]  # Últimas 10 mensagens
            ])
```

```
        prompt = f"""
        Analise esta conversa de WhatsApp de uma clínica médica:

        {conversation}

        Retorne um JSON com:
        1. summary: Resumo em uma linha
        2. action: Ação recomendada para secretaria
        3. category: Categoria principal (agendamento/cancelamento/dúvida/reclamação)
        4. urgency: true/false se é urgente
        """

        # Aqui você pode integrar com OpenAI ou outro modelo
        # Por enquanto, retornamos análise baseada em regras
        return {}

    except Exception as e:
        logger.error(f"Erro na análise IA: {str(e)}")
        return {}
```

3. API do Dashboard (handlers/dashboard.py)

python

```python
from fastapi import APIRouter, Depends, HTTPException, WebSocket, WebSocketDisconnect
from sqlalchemy.orm import Session
from typing import List, Optional
from datetime import datetime, timedelta
import json

from app.models.database import get_db
from app.models.dashboard import (
    ConversationDashboard, ConversationMessage,
    ConversationNote, ConversationStatus, ConversationTag
)
from app.services.classifier import ConversationClassifier

router = APIRouter()
classifier = ConversationClassifier()

# WebSocket manager para real-time updates
class ConnectionManager:
    def __init__(self):
        self.active_connections: List[WebSocket] = []

    async def connect(self, websocket: WebSocket):
        await websocket.accept()
        self.active_connections.append(websocket)

    def disconnect(self, websocket: WebSocket):
        self.active_connections.remove(websocket)

    async def broadcast(self, message: dict):
        for connection in self.active_connections:
            try:
                await connection.send_json(message)
            except:
                pass

manager = ConnectionManager()

@router.get("/conversations")
async def list_conversations(
    status: Optional[ConversationStatus] = None,
    priority: Optional[int] = None,
    date_from: Optional[datetime] = None,
    date_to: Optional[datetime] = None,
    search: Optional[str] = None,
    db: Session = Depends(get_db)
):
```

```
    """Lista todas as conversas com filtros"""

    query = db.query(ConversationDashboard)

    if status:
        query = query.filter(ConversationDashboard.status == status)

    if priority is not None:
        query = query.filter(ConversationDashboard.priority == priority)

    if date_from:
        query = query.filter(ConversationDashboard.first_message_at >= date_from)

    if date_to:
        query = query.filter(ConversationDashboard.last_message_at <= date_to)

    if search:
        query = query.filter(
            ConversationDashboard.phone.contains(search) |
            ConversationDashboard.patient_name.contains(search) |
            ConversationDashboard.patient_cpf.contains(search)
        )

    # Ordenar por prioridade e data
    conversations = query.order_by(
        ConversationDashboard.priority.desc(),
        ConversationDashboard.last_message_at.desc()
    ).all()

    return conversations

@router.get("/conversations/{conversation_id}")
async def get_conversation_detail(
    conversation_id: str,
    db: Session = Depends(get_db)
):
    """Detalhes completos de uma conversa"""

    conversation = db.query(ConversationDashboard).filter(
        ConversationDashboard.id == conversation_id
    ).first()

    if not conversation:
        raise HTTPException(status_code=404, detail="Conversa não encontrada")

    # Carregar mensagens
    messages = db.query(ConversationMessage).filter(
```

```
        ConversationMessage.dashboard_id == conversation_id
    ).order_by(ConversationMessage.timestamp).all()

    # Carregar notas
    notes = db.query(ConversationNote).filter(
        ConversationNote.dashboard_id == conversation_id
    ).order_by(ConversationNote.created_at.desc()).all()

    return {
        "conversation": conversation,
        "messages": messages,
        "notes": notes
    }

@router.patch("/conversations/{conversation_id}")
async def update_conversation(
    conversation_id: str,
    status: Optional[ConversationStatus] = None,
    tags: Optional[List[str]] = None,
    priority: Optional[int] = None,
    reviewed_by: Optional[str] = None,
    db: Session = Depends(get_db)
):
    """Atualiza status e tags de uma conversa"""

    conversation = db.query(ConversationDashboard).filter(
        ConversationDashboard.id == conversation_id
    ).first()

    if not conversation:
        raise HTTPException(status_code=404, detail="Conversa não encontrada")

    if status:
        conversation.status = status

    if tags is not None:
        conversation.tags = tags

    if priority is not None:
        conversation.priority = priority

    if reviewed_by:
        conversation.reviewed_by = reviewed_by
        conversation.reviewed_at = datetime.utcnow()

    db.commit()
```

```
    # Broadcast update
    await manager.broadcast({
        "type": "conversation_updated",
        "conversation_id": conversation_id,
        "status": status.value if status else None
    })

    return conversation

@router.post("/conversations/{conversation_id}/notes")
async def add_note(
    conversation_id: str,
    note: str,
    created_by: str,
    db: Session = Depends(get_db)
):
    """Adiciona nota a uma conversa"""

    new_note = ConversationNote(
        dashboard_id=conversation_id,
        note=note,
        created_by=created_by
    )

    db.add(new_note)
    db.commit()

    return new_note

@router.get("/analytics/summary")
async def get_analytics_summary(
    date_from: Optional[datetime] = None,
    date_to: Optional[datetime] = None,
    db: Session = Depends(get_db)
):
    """Resumo analítico do período"""

    if not date_from:
        date_from = datetime.now() - timedelta(days=7)

    if not date_to:
        date_to = datetime.now()

    # Total de conversas
    total_conversations = db.query(ConversationDashboard).filter(
        ConversationDashboard.first_message_at >= date_from,
        ConversationDashboard.first_message_at <= date_to
```

```
    ).count()

    # Por status
    status_counts = {}
    for status in ConversationStatus:
        count = db.query(ConversationDashboard).filter(
            ConversationDashboard.status == status,
            ConversationDashboard.first_message_at >= date_from
        ).count()
        status_counts[status.value] = count

    # Taxa de resolução do bot
    bot_resolved = db.query(ConversationDashboard).filter(
        ConversationDashboard.bot_resolution == True,
        ConversationDashboard.first_message_at >= date_from
    ).count()

    # Tempo médio de resolução
    avg_resolution = db.query(
        func.avg(ConversationDashboard.resolution_time)
    ).filter(
        ConversationDashboard.resolution_time.isnot(None),
        ConversationDashboard.first_message_at >= date_from
    ).scalar()

    # Tags mais comuns
    # (Implementar agregação de tags JSON)

    return {
        "period": {
            "from": date_from.isoformat(),
            "to": date_to.isoformat()
        },
        "total_conversations": total_conversations,
        "status_distribution": status_counts,
        "bot_resolution_rate": (bot_resolved / total_conversations * 100) if total_conversations > 0 else 0,
        "avg_resolution_time_minutes": avg_resolution or 0
    }

@router.websocket("/ws")
async def websocket_endpoint(websocket: WebSocket):
    """WebSocket para atualizações em tempo real"""
    await manager.connect(websocket)
    try:
        while True:
            # Manter conexão viva
            await websocket.receive_text()
```

```python
    except WebSocketDisconnect:
        manager.disconnect(websocket)

# Função para processar nova conversa (chamada pelo bot)
async def process_new_conversation(phone: str, messages: List[Dict], db: Session):
    """Processa e classifica nova conversa"""

    # Analisar conversa
    analysis = await classifier.analyze_conversation(messages)

    # Criar entrada no dashboard
    dashboard_entry = ConversationDashboard(
        phone=phone,
        tags=analysis['tags'],
        priority=analysis['priority'],
        sentiment_score=analysis['sentiment_score'],
        ai_summary=analysis['ai_summary'],
        ai_suggested_action=analysis['ai_suggested_action'],
        status=ConversationStatus.REQUIRES_ATTENTION if analysis['requires_attention'] else ConversationStatus.PENDIN
        message_count=len(messages),
        first_message_at=messages[0]['timestamp'] if messages else datetime.utcnow(),
        last_message_at=messages[-1]['timestamp'] if messages else datetime.utcnow()
    )

    db.add(dashboard_entry)

    # Adicionar mensagens
    for msg in messages:
        conversation_msg = ConversationMessage(
            dashboard_id=dashboard_entry.id,
            sender=msg['sender'],
            message=msg['message'],
            timestamp=msg['timestamp'],
            message_type=msg.get('type', 'text')
        )
        db.add(conversation_msg)

    db.commit()

    # Notificar dashboard via WebSocket
    await manager.broadcast({
        "type": "new_conversation",
        "conversation": {
            "id": dashboard_entry.id,
            "phone": phone,
            "priority": analysis['priority'],
            "tags": analysis['tags'],
```

```
        "summary": analysis['ai_summary']
    }
})
```

🎨 Frontend - React Dashboard

4. Estrutura do Frontend

bash

```
dashboard-frontend/
├── src/
│   ├── components/
│   │   ├── ConversationList.jsx
│   │   ├── ConversationDetail.jsx
│   │   ├── StatusBadge.jsx
│   │   ├── TagSelector.jsx
│   │   ├── Analytics.jsx
│   │   └── Header.jsx
│   ├── services/
│   │   ├── api.js
│   │   └── websocket.js
│   ├── hooks/
│   │   ├── useConversations.js
│   │   └── useWebSocket.js
│   ├── App.jsx
│   └── index.js
├── package.json
└── tailwind.config.js
```

5. Componente Principal (App.jsx)

jsx

```
import React, { useState, useEffect } from 'react';
import ConversationList from './components/ConversationList';
import ConversationDetail from './components/ConversationDetail';
import Analytics from './components/Analytics';
import Header from './components/Header';
import { useWebSocket } from './hooks/useWebSocket';
import { useConversations } from './hooks/useConversations';

function App() {
  const [selectedConversation, setSelectedConversation] = useState(null);
  const [activeTab, setActiveTab] = useState('conversations');
  const [filters, setFilters] = useState({
    status: null,
    priority: null,
    search: ''
  });

  const { conversations, loading, refetch } = useConversations(filters);
  const { lastMessage } = useWebSocket(refetch);

  return (
    <div className="min-h-screen bg-gray-50">
      <Header />

      <div className="flex h-[calc(100vh-64px)]">
        {/* Sidebar com lista de conversas */}
        <div className="w-96 bg-white border-r overflow-hidden flex flex-col">
          {/* Tabs */}
          <div className="flex border-b">
            <button
              onClick={() => setActiveTab('conversations')}
              className={`flex-1 py-3 px-4 text-sm font-medium ${
                activeTab === 'conversations'
                  ? 'text-blue-600 border-b-2 border-blue-600'
                  : 'text-gray-500'
              }`}
            >
              Conversas
            </button>
            <button
              onClick={() => setActiveTab('analytics')}
              className={`flex-1 py-3 px-4 text-sm font-medium ${
                activeTab === 'analytics'
                  ? 'text-blue-600 border-b-2 border-blue-600'
                  : 'text-gray-500'
              }`}
```

```
  >
    Análise
  </button>
</div>

{/* Filtros */}
<div className="p-4 border-b">
  <input
    type="text"
    placeholder="Buscar por telefone, nome ou CPF..."
    className="w-full px-3 py-2 border rounded-lg"
    value={filters.search}
    onChange={(e) => setFilters({ ...filters, search: e.target.value })}
  />

  <div className="flex gap-2 mt-2">
    <select
      className="flex-1 px-3 py-1 border rounded text-sm"
      value={filters.status || ''}
      onChange={(e) => setFilters({ ...filters, status: e.target.value || null })}
    >
      <option value="">Todos Status</option>
      <option value="pending">Pendente</option>
      <option value="in_progress">Em Atendimento</option>
      <option value="completed">Finalizada</option>
      <option value="requires_attention">Atenção</option>
    </select>

    <select
      className="flex-1 px-3 py-1 border rounded text-sm"
      value={filters.priority || ''}
      onChange={(e) => setFilters({ ...filters, priority: e.target.value || null })}
    >
      <option value="">Todas Prioridades</option>
      <option value="3">Urgente</option>
      <option value="2">Alta</option>
      <option value="1">Média</option>
      <option value="0">Baixa</option>
    </select>
  </div>
</div>

{/* Conteúdo da tab */}
<div className="flex-1 overflow-y-auto">
  {activeTab === 'conversations' ? (
    <ConversationList
      conversations={conversations}
```

```
            loading={loading}
            selectedId={selectedConversation?.id}
            onSelect={setSelectedConversation}
          />
        ) : (
          <Analytics />
        )}
      </div>
    </div>

    {/* Área de detalhes */}
    <div className="flex-1">
      {selectedConversation ? (
        <ConversationDetail
          conversation={selectedConversation}
          onUpdate={refetch}
        />
      ) : (
        <div className="h-full flex items-center justify-center text-gray-400">
          <div className="text-center">
            <svg className="w-16 h-16 mx-auto mb-4" fill="none" stroke="currentColor" viewBox="0 0 24 24">
              <path strokeLinecap="round" strokeLinejoin="round" strokeWidth={2} d="M8 12h.01M12 12h.01M16 12h.
            </svg>
            <p>Selecione uma conversa para ver detalhes</p>
          </div>
        </div>
      )}
    </div>
  </div>
</div>
);
}

export default App;
```

6. Lista de Conversas (ConversationList.jsx)

jsx

```
import React from 'react';
import StatusBadge from './StatusBadge';
import { formatDistanceToNow } from 'date-fns';
import { ptBR } from 'date-fns/locale';

function ConversationList({ conversations, loading, selectedId, onSelect }) {
 if (loading) {
  return (
   <div className="p-4">
    <div className="animate-pulse space-y-3">
     {[...Array(5)].map((_, i) => (
      <div key={i} className="bg-gray-200 h-20 rounded"></div>
     ))}
    </div>
   </div>
  );
 }

 if (conversations.length === 0) {
  return (
   <div className="p-4 text-center text-gray-500">
    Nenhuma conversa encontrada
   </div>
  );
 }

 const getPriorityColor = (priority) => {
  switch (priority) {
   case 3: return 'bg-red-500';
   case 2: return 'bg-orange-500';
   case 1: return 'bg-yellow-500';
   default: return 'bg-green-500';
  }
 };

 const getPriorityText = (priority) => {
  switch (priority) {
   case 3: return 'Urgente';
   case 2: return 'Alta';
   case 1: return 'Média';
   default: return 'Baixa';
  }
 };

 return (
  <div>
```

```
{conversations.map((conversation) => (
  <div
   key={conversation.id}
   onClick={() => onSelect(conversation)}
   className={`p-4 border-b cursor-pointer hover:bg-gray-50 transition-colors ${
    selectedId === conversation.id ? 'bg-blue-50' : ''
   }`}
  >
   {/* Header da conversa */}
   <div className="flex items-start justify-between mb-2">
    <div className="flex-1">
     <div className="flex items-center gap-2">
      <h3 className="font-semibold text-gray-900">
       {conversation.patient_name || conversation.phone}
      </h3>
      <span className={`w-2 h-2 rounded-full ${getPriorityColor(conversation.priority)}`} />
     </div>
     <p className="text-sm text-gray-500">{conversation.phone}</p>
    </div>
    <StatusBadge status={conversation.status} />
   </div>

   {/* Resumo da conversa */}
   <p className="text-sm text-gray-600 mb-2 line-clamp-2">
    {conversation.ai_summary}
   </p>

   {/* Tags */}
   <div className="flex flex-wrap gap-1 mb-2">
    {conversation.tags?.slice(0, 3).map((tag, index) => (
     <span
      key={index}
      className="px-2 py-1 text-xs bg-gray-100 text-gray-600 rounded"
     >
      {tag}
     </span>
    ))}
    {conversation.tags?.length > 3 && (
     <span className="text-xs text-gray-500">
      +{conversation.tags.length - 3}
     </span>
    )}
   </div>

   {/* Footer */}
   <div className="flex items-center justify-between text-xs text-gray-500">
    <span>{conversation.message_count} mensagens</span>
```

```
            <span>
              {formatDistanceToNow(new Date(conversation.last_message_at), {
                addSuffix: true,
                locale: ptBR
              })}
            </span>
          </div>

          {/* Indicador de atenção necessária */}
          {conversation.status === 'requires_attention' && (
            <div className="mt-2 px-2 py-1 bg-red-100 text-red-700 text-xs rounded">
              ⚠️ Requer atenção imediata
            </div>
          )}
        </div>
      ))}
    </div>
  );
}

export default ConversationList;
```