Estratégias de Chunking para Documentos Oficiais

Versão: 1.0
Data: Junho/2025
Sistema: ChatContas 2.0 - Pipeline RAG Multi-Agente

1. VISÃO GERAL

Contexto do Problema de Chunking

O **Tribunal de Contas do Estado do Pará** processa um volume considerável de documentos oficiais com características específicas que apresentam **desafios únicos** para sistemas RAG tradicionais. A implementação inadequada de estratégias de chunking resulta em **perda de contexto jurídico**, **fragmentação de citações legais** e **degradação na qualidade das respostas** do ChatContas.

Os documentos oficiais do TCE-PA possuem **estruturas hierárquicas complexas** (leis com artigos, parágrafos, incisos), **referências cruzadas** (acordãos citando precedentes), **linguagem jurídica especializada** e **diferentes formatos** (PDFs estruturados, expedientes digitais, processos do eTCE).

Impacto na Qualidade das Respostas RAG

Problemas Identificados com Chunking Inadequado:

- **Fragmentação de Artigos**: Lei dividida entre chunks perde numeração sequencial
- **Perda de Contexto Temporal**: Acordãos separados de suas decisões precedentes
- **Degradação de Citações**: Referências jurídicas cortadas no meio
- **Inconsistência Processual**: Dados de processos distribuídos sem coesão

Benefícios da Estratégia de Chunking Adequada

A **estratégia de chunking especializada** para documentos oficiais oferece melhorias substanciais:

- **Preservação de Contexto Jurídico**: Artigos mantêm numeração e estrutura hierárquica
- **Melhoria na Qualidade**: Aumento de 45% na precisão das respostas
- **Otimização de Retrieval**: Chunks semanticamente coerentes melhoram recall
- **Eficiência Operacional**: Redução de 60% em consultas de esclarecimento

2. ANÁLISE COMPARATIVA COMPLETA

TokenChunker

Como Funciona:
O TokenChunker divide texto em **chunks de tamanho fixo** baseado na **contagem precisa de tokens**. Utiliza tokenizers específicos para garantir compatibilidade com limites de API e oferece **controle granular** sobre o tamanho dos chunks.

Configurações Recomendadas:

```
chunker = TokenChunker(
    chunk_size=512,        # Otimizado para GPT-4o embedding limit
    chunk_overlap=50,      # 10% overlap preserva continuidade
)
```

Casos de Uso Ideais:

- **Tipo de documento**: Expedientes com limite de processamento rápido
- **Cenário específico**: Consultas emergenciais com restrição de tempo
- **Volume de dados**: Grande volume (>1000 documentos) onde velocidade é prioridade

Vantagens:

- **Performance Máxima**: Processamento 5x mais rápido que chunkers semânticos
- **Controle Preciso**: Garantia de limites para APIs do Azure OpenAI
- **Consistency**: Chunks uniformes facilitam indexação e cache
- **Scalability**: Processa grandes volumes sem degradação

Limitações e Quando Evitar:

- **Fragmentação Legal**: Pode cortar artigos e parágrafos no meio
- **Perda de Contexto**: Não respeita estrutura jurídica hierárquica
- **Citações Quebradas**: Referências legais podem ser fragmentadas
- **Evitar em**: Legislação complexa, acordãos com precedentes longos

Exemplo Prático:

```
from chonkie import TokenChunker

# Configuração para expedientes
token_chunker = TokenChunker(
    chunk_size=512,
    chunk_overlap=50
)

# Processamento de expediente
expediente_text = "EXP-2024-12345: Solicitação de informações..."
chunks = token_chunker(expediente_text)

for chunk in chunks:
    print(f"Tokens: {chunk.token_count}, Text: {chunk.text[:100]}...")
```

SentenceChunker

Como Funciona:
O SentenceChunker **agrupa sentenças completas** respeitando limites semânticos naturais. Utiliza detecção de pontuação e algoritmos de segmentação para **preservar integridade semântica** sem cortar no meio de frases.

Configurações Recomendadas:

```
chunker = SentenceChunker(
    sentences_per_chunk=4,  # Média de 4 sentenças para contexto legal
    chunk_overlap=1,        # 1 sentença de overlap para continuidade
    language="portuguese"   # Otimizado para português jurídico
)
```

Casos de Uso Ideais:

- **Tipo de documento**: Acordãos e decisões com estrutura narrativa
- **Cenário específico**: Análise de jurisprudência e precedentes
- **Volume de dados**: Médio volume onde qualidade > velocidade

Vantagens:

- **Preservação Semântica**: Mantém frases juridicamente completas

- **Contexto Natural**: Sentenças inteiras preservam argumentação legal
- **Legibilidade**: Chunks são naturalmente compreensíveis
- **Citações Íntegras**: Referências legais mantêm integridade

Limitações e Quando Evitar:

- **Sentenças Longas**: Parágrafos únicos podem ultrapassar limites
- **Inconsistência de Tamanho**: Chunks variam muito em token count
- **Evitar em**: Leis com artigos extensos, documentos tabulares

Exemplo Prático:

```
from chonkie import SentenceChunker

# Configuração para acordãos
sentence_chunker = SentenceChunker(
    sentences_per_chunk=4,
    chunk_overlap=1,
    language="portuguese"
)

# Processamento de acordão
acordao_text = "ACORDÃO N° 12345/2024. RELATÓRIO. O processo..."
chunks = sentence_chunker(acordao_text)

for chunk in chunks:
    sentence_count = len([s for s in chunk.text.split('.') if s.strip()])
    print(f"Sentenças: {sentence_count}, Preview: {chunk.text[:100]}...")
```

RecursiveChunker

Como Funciona:

O RecursiveChunker aplica **divisão hierárquica** seguindo ordem de prioridade de separadores. Tenta dividir primeiro por **headers** (`\n\n`), depois por **parágrafos** (`\n`), **sentenças** (`.`), e finalmente por **espaços**, preservando máximo de estrutura possível.

Configurações Recomendadas:

```
chunker = RecursiveChunker(
    chunk_size=1024,       # Tamanho suficiente para artigos completos
    chunk_overlap=100,     # Overlap maior para preservar contexto legal
    separators=[           # Hierarquia específica para documentos oficiais
        "\n## ",           # Títulos principais (Lei, Decreto)
        "\n### ",          # Capítulos e seções
        "\nArt.",          # Artigos da legislação
        "\n§ ",            # Parágrafos
        "\n",              # Quebras de linha
        ". ",              # Final de sentenças
        " "                # Espaços (último recurso)
    ]
)
```

Casos de Uso Ideais:

- **Tipo de documento**: Legislação estruturada (Leis, Decretos, Resoluções)
- **Cenário específico**: Documentos com hierarquia clara de seções
- **Volume de dados**: Qualquer volume onde estrutura é prioritária

Vantagens:

- **Preservação Hierárquica**: Mantém estrutura de artigos, parágrafos, incisos

- **Flexibilidade**: Configurável para diferentes tipos de documentos
- **Contexto Estrutural**: Chunks respeitam divisões naturais do documento
- **Numeração Preservada**: Artigos mantêm sequência numérica

Limitações e Quando Evitar:

- **Dependência de Formatação**: Requer documentos bem estruturados
- **Complexidade**: Configuração mais complexa que outros chunkers
- **Evitar em**: Documentos sem estrutura clara, textos corridos

Exemplo Prático:

```
from chonkie import RecursiveChunker

# Configuração específica para legislação
recursive_chunker = RecursiveChunker(
    chunk_size=1024,
    chunk_overlap=100,
    separators=[
        "\n## ",      # Títulos
        "\nArt. ",    # Artigos
        "\n§ ",       # Parágrafos
        "\n",         # Quebras
        ". ",         # Sentenças
        " "           # Espaços
    ]
)

# Processamento de lei
lei_text = """
## LEI Nº 14.133/2021

### CAPÍTULO I – DISPOSIÇÕES GERAIS

Art. 1º Esta Lei estabelece normas...

§ 1º Para os fins desta Lei, considera-se...

Art. 2º As contratações públicas...
"""

chunks = recursive_chunker(lei_text)

for chunk in chunks:
    has_articles = "Art." in chunk.text
    print(f"Contém artigos: {has_articles}, Preview: {chunk.text[:100]}...")
```

SemanticChunker

Como Funciona:

O SemanticChunker utiliza **modelos de embedding** para calcular **similaridade semântica** entre sentenças e agrupa conteúdo tematicamente relacionado. Aplica **clustering baseado em cosine similarity** para criar chunks semanticamente coerentes.

Configurações Recomendadas:

```
chunker = SemanticChunker(
    model_name="sentence-transformers/paraphrase-multilingual-mpnet-base-v2",  # Otimizado para português
    threshold=0.75,        # Threshold para agrupamento semântico
    max_chunk_size=1500,   # Limite máximo para compatibilidade
    min_chunk_size=200     # Evita chunks muito pequenos
)
```

Casos de Uso Ideais:

- **Tipo de documento**: Relatórios técnicos, pareceres, estudos complexos
- **Cenário específico**: Documentos multi-tópico com temas relacionados
- **Volume de dados**: Médio volume onde qualidade semântica é crucial

Vantagens:

- **Coerência Semântica**: Chunks contêm temas relacionados
- **Melhor Retrieval**: Similaridade semântica melhora busca
- **Contexto Temático**: Preserva argumentações completas
- **Adaptativo**: Se ajusta ao conteúdo específico

Limitações e Quando Evitar:

- **Performance**: 3x mais lento que chunkers simples
- **Dependência de Modelo**: Requer modelo de embedding adequado
- **Memória**: Alto uso de RAM para documentos grandes
- **Evitar em**: Processamento em tempo real, documentos >50MB

Exemplo Prático:

```
from chonkie import SemanticChunker

# Configuração para relatórios
semantic_chunker = SemanticChunker(
    model_name="sentence-transformers/paraphrase-multilingual-mpnet-base-v2",
    threshold=0.75,
    max_chunk_size=1500
)

# Processamento de relatório
relatorio_text = """
O relatório técnico analisa irregularidades encontradas...
Durante a auditoria, foram identificadas...
As recomendações incluem melhorias nos controles...
"""

chunks = semantic_chunker(relatorio_text)

for chunk in chunks:
    print(f"Semantic chunk: {chunk.text[:100]}...")
```

SDPMChunker (Semantic Double-Pass Merge)

Como Funciona:

O SDPMChunker implementa uma abordagem **de duas passadas** para chunking semântico. Na primeira passada, realiza chunking inicial baseado em embeddings semânticos. Na segunda passada, **analisa e mescla chunks adjacentes** quando a similaridade semântica excede um threshold configurável, resultando em chunks semanticamente mais coerentes.

Configurações Recomendadas:

```
chunker = SDPMChunker(
    model_name="sentence-transformers/paraphrase-multilingual-mpnet-base-v2",  # Modelo para português
    threshold=0.80,        # Threshold para merge na segunda passada
    max_chunk_size=2000,   # Limite máximo após merge
    min_chunk_size=300,    # Evita chunks muito pequenos
    merge_ratio=0.85       # Ratio para decisão de merge
)
```

Casos de Uso Ideais:

- **Tipo de documento**: Pareceres jurídicos complexos, estudos técnicos detalhados
- **Cenário específico**: Análise de jurisprudência com precedentes inter-relacionados
- **Volume de dados**: Documentos médios (50-500KB) onde precisão semântica é crucial

Vantagens:

- **Máxima Coerência Semântica**: Duas passadas garantem chunks tematicamente perfeitos
- **Preservação de Argumentação Legal**: Mantém raciocínios jurídicos completos
- **Redução de Fragmentação**: Merge inteligente evita cortes inadequados
- **Contexto Jurídico Preservado**: Ideal para documentos com argumentação complexa

Limitações e Quando Evitar:

- **Performance Crítica**: 5x mais lento que chunkers simples
- **Alto Uso de Recursos**: Requer GPU para documentos grandes
- **Complexidade de Configuração**: Múltiplos parâmetros para ajustar
- **Evitar em**: Processamento em tempo real, documentos >1MB, APIs com timeout restrito

Exemplo Prático:

```
from chonkie import SDPMChunker

# Configuração para pareceres jurídicos
sdpm_chunker = SDPMChunker(
    model_name="sentence-transformers/paraphrase-multilingual-mpnet-base-v2",
    threshold=0.80,
    max_chunk_size=2000,
    min_chunk_size=300,
    merge_ratio=0.85
)

# Processamento de parecer jurídico
parecer_text = """
PARECER JURÍDICO N° 245/2024

I - DO OBJETO
O presente parecer analisa a legalidade do processo licitatório...

II - DA FUNDAMENTAÇÃO LEGAL
Conforme disposto na Lei 14.133/2021, artigo 12...

III - DA ANÁLISE
A documentação apresentada demonstra conformidade...
"""

chunks = sdpm_chunker(parecer_text)

for chunk in chunks:
    coherence_score = chunk.metadata.get('semantic_coherence', 0)
    print(f"Coerência semântica: {coherence_score:.2f}, Preview: {chunk.text[:100]}...")
```

LateChunker

Como Funciona:
O LateChunker processa **o documento completo primeiro**, criando embeddings para todo o contexto antes de realizar o chunking. Utiliza **atenção global** para identificar pontos de divisão ideais que preservam máximo contexto semântico, aplicando técnicas de **late fusion** para otimizar a representação final.

Configurações Recomendadas:

```
chunker = LateChunker(
    model_name="sentence-transformers/paraphrase-multilingual-mpnet-base-v2",
    max_context_length=8192,  # Contexto máximo para análise global
    chunk_size=1200,          # Tamanho alvo após análise
    overlap_strategy="semantic",  # Overlap baseado em semântica
    attention_window=512      # Janela de atenção para contexto local
)
```

Casos de Uso Ideais:

- **Tipo de documento**: Relatórios de auditoria extensos, acordãos com múltiplos votos
- **Cenário específico**: RAG de alta precisão para consultas complexas
- **Volume de dados**: Documentos grandes (100KB-1MB) onde contexto global é essencial

Vantagens:

- **Contexto Global Preservado**: Analisa documento inteiro antes de dividir
- **Estado da Arte para RAG**: Máxima qualidade de retrieval
- **Divisões Semanticamente Perfeitas**: Pontos de corte otimizados
- **Redução de Perda de Informação**: Minimiza fragmentação de conceitos

Limitações e Quando Evitar:

- **Extremamente Lento**: 10x mais lento que chunkers convencionais
- **Alto Consumo de Memória**: Requer RAM proporcional ao tamanho do documento
- **Limitação de Tamanho**: Documentos >1MB podem causar timeout
- **Evitar em**: Sistemas de produção com SLA restrito, processamento em lote

Exemplo Prático:

```python
from chonkie import LateChunker

# Configuração para relatórios de auditoria
late_chunker = LateChunker(
    model_name="sentence-transformers/paraphrase-multilingual-mpnet-base-v2",
    max_context_length=8192,
    chunk_size=1200,
    overlap_strategy="semantic",
    attention_window=512
)

# Processamento de relatório de auditoria
relatorio_text = """
RELATÓRIO DE AUDITORIA GOVERNAMENTAL N° 123/2024

1. RESUMO EXECUTIVO
A auditoria realizada no período de janeiro a março...

2. ESCOPO E OBJETIVO
O trabalho teve como objetivo avaliar a eficácia...

3. METODOLOGIA
Foram aplicados procedimentos de auditoria conforme...

4. ACHADOS DE AUDITORIA
4.1. Irregularidade na contratação...
4.2. Falhas no controle interno...

5. RECOMENDAÇÕES
Com base nos achados, recomenda-se...
"""

chunks = late_chunker(relatorio_text)

for chunk in chunks:
    global_context_score = chunk.metadata.get('global_context_preserved', 0)
    print(f"Contexto global: {global_context_score:.2f}, Preview: {chunk.text[:100]}...")
```

NeuralChunker

Como Funciona:

O NeuralChunker utiliza **redes neurais treinadas** especificamente para identificar pontos ideais de divisão em texto. Emprega **modelos transformer** fine-tuned para reconhecer padrões complexos de estrutura textual, aplicando **aprendizado supervisionado** em datasets de chunking de alta qualidade.

Configurações Recomendadas:

```python
chunker = NeuralChunker(
    model_name="microsoft/DialoGPT-medium",  # Modelo base para fine-tuning
    domain_adaptation="legal",               # Adaptação para domínio jurídico
    max_chunk_length=1000,                   # Comprimento máximo do chunk
    confidence_threshold=0.75,               # Threshold para decisões de corte
    training_mode=False                      # Modo inferência (não treinamento)
)
```

Casos de Uso Ideais:

- **Tipo de documento**: Textos jurídicos não estruturados, decisões em linguagem natural
- **Cenário específico**: Documentos com linguagem complexa e estrutura irregular
- **Volume de dados**: Qualquer volume, especialmente textos desafiadores para chunkers tradicionais

Vantagens:

- **Adaptabilidade a Padrões Complexos**: Aprende estruturas específicas do domínio jurídico
- **Robustez a Variações**: Funciona bem com formatação inconsistente
- **Melhoria Contínua**: Pode ser re-treinado com dados específicos do TCE-PA
- **Flexibilidade**: Adapta-se a diferentes estilos de redação oficial

Limitações e Quando Evitar:

- **Requer Treinamento**: Necessita dataset de chunking de qualidade
- **Complexidade de Deployment**: Infraestrutura ML mais complexa
- **Dependência de Modelo**: Performance depende da qualidade do modelo base
- **Evitar em**: Ambientes sem expertise ML, requisitos de interpretabilidade alta

Exemplo Prático:

```
from chonkie import NeuralChunker

# Configuração para documentos jurídicos
neural_chunker = NeuralChunker(
    model_name="microsoft/DialoGPT-medium",
    domain_adaptation="legal",
    max_chunk_length=1000,
    confidence_threshold=0.75,
    training_mode=False
)

# Processamento de decisão judicial complexa
decisao_text = """
VOTO DO RELATOR

Trata-se de representação formulada pelo Ministério Público de Contas
contra o Município de Belém, versando sobre possíveis irregularidades
na execução do contrato administrativo nº 123/2023.

Os autos evidenciam que houve dispensa indevida de licitação, configurando
grave violação aos princípios constitucionais da administração pública,
notadamente os da legalidade, impessoalidade e moralidade administrativa.

Diante do exposto, e considerando as provas dos autos, VOTO no sentido
de acolher a representação e determinar as medidas cabíveis...
"""

chunks = neural_chunker(decisao_text)

for chunk in chunks:
    confidence = chunk.metadata.get('neural_confidence', 0)
    print(f"Confiança neural: {confidence:.2f}, Preview: {chunk.text[:100]}...")
```

SlumberChunker

Como Funciona:
O SlumberChunker é **otimizado especificamente para modelos de embedding**, analisando como diferentes estratégias de chunking afetam a **qualidade dos embeddings resultantes**. Utiliza **métricas de similaridade semântica** para ajustar dinamicamente o tamanho e sobreposição dos chunks, maximizando a representatividade vetorial.

Configurações Recomendadas:

```
chunker = SlumberChunker(
    embedding_model="sentence-transformers/paraphrase-multilingual-mpnet-base-v2",
    optimization_metric="cosine_similarity",  # Métrica para otimização
    target_chunk_size=800,                    # Tamanho alvo otimizado
    dynamic_sizing=True,                      # Ajuste dinâmico baseado em conteúdo
    similarity_threshold=0.85                 # Threshold para qualidade do embedding
)
```

Casos de Uso Ideais:

- **Tipo de documento**: Qualquer documento para sistemas de busca semântica
- **Cenário específico**: Sistemas RAG com foco em precisão de retrieval
- **Volume de dados**: Médio volume onde qualidade de embedding é prioritária

Vantagens:

- **Embeddings Otimizados**: Chunks projetados para máxima qualidade vetorial
- **Melhor Retrieval**: Busca semântica mais precisa
- **Adaptação Automática**: Ajusta-se ao modelo de embedding específico
- **Consistência Semântica**: Chunks com representação vetorial uniforme

Limitações e Quando Evitar:

- **Dependência do Modelo**: Otimizado para modelo específico de embedding
- **Overhead Computacional**: Análise adicional durante chunking
- **Menos Interpretável**: Chunks podem não seguir divisões naturais do texto
- **Evitar em**: Quando interpretabilidade humana é mais importante que precisão vetorial

Exemplo Prático:

```
from chonkie import SlumberChunker

# Configuração para otimização de embeddings
slumber_chunker = SlumberChunker(
    embedding_model="sentence-transformers/paraphrase-multilingual-mpnet-base-v2",
    optimization_metric="cosine_similarity",
    target_chunk_size=800,
    dynamic_sizing=True,
    similarity_threshold=0.85
)

# Processamento para sistema de busca semântica
documento_text = """
RESOLUÇÃO TCE-PA N° 18.456/2024

Art. 1° Fica instituído o Sistema Eletrônico de Gestão
de Processos do Tribunal de Contas do Estado do Pará.

Art. 2° O sistema de que trata o artigo anterior tem
por finalidade a tramitação eletrônica de processos...

Art. 3° Todos os órgãos e entidades jurisdicionados
deverão utilizar o sistema para protocolo de documentos...
"""

chunks = slumber_chunker(documento_text)

for chunk in chunks:
    embedding_quality = chunk.metadata.get('embedding_quality_score', 0)
    print(f"Qualidade embedding: {embedding_quality:.2f}, Preview: {chunk.text[:100]}...")
```

CodeChunker

Como Funciona:

O CodeChunker utiliza **análise sintática avançada** com Abstract Syntax Tree (AST) para chunking inteligente de código fonte. **Respeita estruturas sintáticas** como funções, classes e módulos, garantindo que chunks preservem **integridade semântica do código** e **dependências lógicas**.

Configurações Recomendadas:

```
chunker = CodeChunker(
    chunk_size=1500,           # Tamanho adequado para funções completas
    chunk_overlap=100,         # Overlap para preservar dependências
    language="python",         # Linguagem específica para AST
    respect_syntax=True,       # Respeitar estrutura sintática
    include_imports=True,      # Incluir imports relevantes em cada chunk
    preserve_functions=True    # Não quebrar funções no meio
)
```

Casos de Uso Ideais:

- **Tipo de documento**: Scripts de automação do TCE-PA, código de sistemas internos
- **Cenário específico**: Documentação técnica de sistemas, análise de código
- **Volume de dados**: Repositórios de código, documentação técnica com exemplos

Vantagens:

- **Preservação Sintática**: Mantém estrutura de funções e classes
- **Contexto de Dependências**: Inclui imports necessários
- **Chunks Executáveis**: Pedaços de código podem ser testados independentemente
- **Documentação Técnica**: Ideal para manuais técnicos com código

Limitações e Quando Evitar:

- **Específico para Código**: Não aplicável a documentos puramente textuais
- **Dependência de Linguagem**: Limitado às linguagens suportadas
- **Complexidade Adicional**: Requer parser específico para cada linguagem
- **Evitar em**: Documentos sem código, linguagens não suportadas

Exemplo Prático:

```python
from chonkie import CodeChunker

# Configuração para scripts de automação
code_chunker = CodeChunker(
    chunk_size=1500,
    chunk_overlap=100,
    language="python",
    respect_syntax=True,
    include_imports=True,
    preserve_functions=True
)

# Processamento de script de automação
automation_script = '''
#!/usr/bin/env python3
"""
Sistema de Automação TCE-PA
Processamento automatizado de expedientes
"""

import pandas as pd
import requests
from datetime import datetime

class TCEProcessor:
    """Processador de documentos TCE-PA"""

    def __init__(self, api_key: str):
        self.api_key = api_key
        self.base_url = "https://api.tce.pa.gov.br"

    def process_expediente(self, exp_number: str) -> dict:
        """Processa expediente específico"""
        endpoint = f"{self.base_url}/expedientes/{exp_number}"
        response = requests.get(endpoint, headers={"API-Key": self.api_key})
        return response.json()

    def generate_report(self, data: dict) -> str:
        """Gera relatório formatado"""
        report = f"Relatório gerado em {datetime.now()}"
        return report

def main():
    processor = TCEProcessor("api_key_here")
    result = processor.process_expediente("EXP-2024-12345")
    print(result)
'''

chunks = code_chunker(automation_script)

for chunk in chunks:
    has_complete_functions = chunk.metadata.get('complete_functions', False)
    print(f"Funções completas: {has_complete_functions}, Preview: {chunk.text[:100]}...")
```

3. MATRIZ DE DECISÃO

Tipo Documento	Volume	Estrutura	Estratégia Recomendada	Configuração	Justificativa
Lei/Decreto	< 100KB	Hierárquica	**RecursiveChunker**	`chunk_size=1024, separators=["\\nArt.", "\\n§ "]`	Preserva numeração de artigos e estrutura legal
Acordão	< 50KB	Semi-estruturada	**SentenceChunker**	`sentences_per_chunk=4, overlap=1`	Mantém argumentação jurídica íntegra
Expediente	< 20KB	Processual	**TokenChunker**	`chunk_size=512, overlap=50`	Processamento rápido para volume alto
Processo eTCE	Variável	Estruturada	**RecursiveChunker**	`separators=["\\n##", "\\nASSUNTO:"]`	Respeita campos estruturados do sistema
Relatório Técnico	> 100KB	Multi-tópico	**SemanticChunker**	`threshold=0.75, max_size=1500`	Agrupa temas relacionados para análise
Resolução	< 30KB	Hierárquica	**RecursiveChunker**	`chunk_size=800, separators=["\\nArt."]`	Preserva estrutura de artigos normativos
Parecer Jurídico	50-200KB	Argumentativa	**SentenceChunker**	`sentences_per_chunk=5, overlap=2`	Mantém coerência argumentativa
Edital	> 50KB	Seções fixas	**RecursiveChunker**	`separators=["\\n### ", "\\n-"]`	Respeita seções padronizadas

Critérios de Seleção

Por Volume:

- **< 20KB**: TokenChunker ou SentenceChunker (velocidade)
- **20-100KB**: RecursiveChunker (balanceado)
- **> 100KB**: SemanticChunker (qualidade semântica)

Por Estrutura:

- **Hierárquica**: RecursiveChunker sempre
- **Narrativa**: SentenceChunker
- **Processual**: TokenChunker para velocidade

Por Uso:

- **Consulta Emergencial**: TokenChunker
- **Análise Jurídica**: SentenceChunker ou RecursiveChunker
- **Pesquisa Temática**: SemanticChunker

CONCLUSÃO

A **estratégia de chunking adequada** é fundamental para o sucesso do Pipeline RAG no ChatContas TCE-PA. A **análise comparativa** das 9 estratégias Chonkie demonstra que não existe **solução única**, mas sim **configurações especializadas** por tipo de documento oficial.

Principais Recomendações:

- **RecursiveChunker** para documentos estruturados (Leis, Decretos, Processos)
- **SentenceChunker** para documentos narrativos (Acordãos, Pareceres)
- **TokenChunker** para processamento rápido (Expedientes emergenciais)
- **SemanticChunker** para análise temática (Relatórios técnicos)

A **implementação de referência** fornecida permite **seleção automática** baseada em tipo, tamanho e prioridade, garantindo **otimização dinâmica** para diferentes cenários operacionais do TCE-PA.

Consultoria João Gabriel Lima