Exemplos Práticos de Configuração - COBOL AI Engine v2.1.4

Cenários Reais de Configuração

Cenário 1: Ambiente de Produção Crítico

Situação: Sistema em produção com alta disponibilidade, processamento de programas críticos do mainframe.

Configuração Recomendada:

YAML

```yaml
# config/config_producao_critico.yaml
ai:
  primary_provider: "luzia"
  enable_fallback: true

  providers:
    luzia:
      enabled: true
      client_id: "${LUZIA_CLIENT_ID}"
      client_secret: "${LUZIA_CLIENT_SECRET}"
      model: "aws-claude-3-5-sonnet"
      temperature: 0.05              # Muito baixo para consistência
      max_tokens: 8000               # Alto para análises completas
      timeout: 300                   # 5 minutos de timeout

      retry:
        max_attempts: 10             # Muitas tentativas
        base_delay: 5.0              # Delay conservador
        max_delay: 180.0             # Até 3 minutos entre tentativas
        backoff_multiplier: 1.5      # Crescimento suave
        requests_per_minute: 4       # Muito conservador

    enhanced_mock:
      enabled: true                  # Fallback garantido

performance:
  token_management:
    enable_chunking: false           # Análises completas sempre

logging:
```

```yaml
  level: INFO
  file: "logs/producao_critico.log"
```

Comando de Execução:

```bash
python main.py --config config/config_producao_critico.yaml \
               --fontes programas_criticos.txt \
               --output documentacao_critica \
               --provider luzia
```

Características:

- Máxima confiabilidade
- Análises sempre completas
- Fallback garantido
- Logs detalhados para auditoria

Cenário 2: Desenvolvimento Ágil

Situação: Equipe de desenvolvimento testando e iterando rapidamente.

Configuração Recomendada:

```yaml
# config/config_desenvolvimento.yaml
ai:
  primary_provider: "enhanced_mock"  # Mock para velocidade
  enable_fallback: true

  providers:
    enhanced_mock:
      enabled: true
      response_delay: 0.05           # Muito rápido
      simulate_realistic_tokens: true
      cobol_analysis_enabled: true

    luzia:
      enabled: true
      retry:
        max_attempts: 2              # Poucas tentativas
        base_delay: 0.5              # Delay mínimo
        max_delay: 10.0              # Máximo 10s
```

```yaml
        backoff_multiplier: 2.0
        requests_per_minute: 15      # Mais permissivo
      timeout: 30                    # Timeout baixo

performance:
  token_management:
    enable_chunking: true            # Pode dividir para testes
    max_tokens_per_chunk: 50000      # Chunks menores

logging:
  level: DEBUG                       # Logs detalhados
  file: "logs/desenvolvimento.log"
```

Comando de Execução:

Bash

```bash
python main.py --config config/config_desenvolvimento.yaml \
               --fontes exemplos_teste.txt \
               --output teste_rapido
```

Características:

- Velocidade máxima
- Feedback imediato
- Logs detalhados para debug
- Flexibilidade para testes

Cenário 3: Processamento em Lote (Batch)

Situação: Análise de centenas de programas COBOL em lote durante a madrugada.

Configuração Recomendada:

YAML

```yaml
# config/config_batch.yaml
ai:
  primary_provider: "luzia"
  enable_fallback: true

  providers:
    luzia:
      enabled: true
      model: "azure-gpt-4o-mini"     # Modelo eficiente
```

```yaml
      temperature: 0.1
      max_tokens: 6000
      timeout: 240                    # 4 minutos

      retry:
        max_attempts: 8               # Muitas tentativas
        base_delay: 3.0               # Delay moderado
        max_delay: 120.0              # Até 2 minutos
        backoff_multiplier: 1.8       # Crescimento moderado
        requests_per_minute: 6        # Sustentável para lote

performance:
  token_management:
    enable_chunking: false            # Análises completas

logging:
  level: INFO
  file: "logs/batch_processing.log"
  max_size: "50MB"                    # Logs maiores
  backup_count: 10                    # Mais backups
```

Script de Execução:

Bash

```bash
#!/bin/bash
# batch_analysis.sh

echo "Iniciando processamento em lote - $(date)"

python main.py --config config/config_batch.yaml \
               --fontes programas_lote_1.txt \
               --output batch_resultado_1 \
               --provider luzia

python main.py --config config/config_batch.yaml \
               --fontes programas_lote_2.txt \
               --output batch_resultado_2 \
               --provider luzia

echo "Processamento concluído - $(date)"
```

Características:

- Otimizado para volume
- Rate limiting sustentável

- Logs robustos
- Recuperação automática de falhas

Cenário 4: Ambiente Híbrido (Multi-Provedor)

Situação: Uso de múltiplos provedores conforme disponibilidade e necessidade.

Configuração Recomendada:

YAML

```yaml
# config/config_hibrido.yaml
ai:
  primary_provider: "luzia"
  enable_fallback: true

  providers:
    luzia:
      enabled: true
      retry:
        max_attempts: 5
        base_delay: 2.0
        max_delay: 60.0
        backoff_multiplier: 2.0
        requests_per_minute: 8
      timeout: 180

    openai:
      enabled: true
      api_key: "${OPENAI_API_KEY}"
      model: "gpt-4"
      temperature: 0.1
      max_tokens: 4000
      timeout: 60

    databricks:
      enabled: true
      workspace_url: "${DATABRICKS_WORKSPACE_URL}"
      access_token: "${DATABRICKS_ACCESS_TOKEN}"
      model: "llama-3.1-405b-instruct"
      timeout: 120

    enhanced_mock:
      enabled: true                  # Fallback final

performance:
  token_management:
```

```yaml
    provider_specific:
      luzia:
        enable_chunking: false
        max_tokens_per_request: 200000
      openai:
        enable_chunking: true
        max_tokens_per_request: 120000
      databricks:
        enable_chunking: true
        max_tokens_per_request: 100000
```

Comandos de Execução:

Bash

```bash
# Usar LuzIA preferencialmente
python main.py --config config/config_hibrido.yaml \
               --fontes programas.txt \
               --output resultado_luzia \
               --provider luzia

# Usar OpenAI como alternativa
python main.py --config config/config_hibrido.yaml \
               --fontes programas.txt \
               --output resultado_openai \
               --provider openai

# Fallback automático (tenta todos)
python main.py --config config/config_hibrido.yaml \
               --fontes programas.txt \
               --output resultado_auto
```

Características:

- Flexibilidade máxima
- Redundância de provedores
- Configuração específica por provedor
- Fallback inteligente

Cenário 5: Análise de Programas Muito Grandes

Situação: Programas COBOL com mais de 10.000 linhas que excedem limites de token.

Configuração Recomendada:

YAML

```yaml
# config/config_programas_grandes.yaml
ai:
  primary_provider: "luzia"

  providers:
    luzia:
      enabled: true
      max_tokens: 8000                # Máximo de tokens
      timeout: 600                    # 10 minutos

      retry:
        max_attempts: 3               # Menos tentativas (programas grandes)
        base_delay: 10.0              # Delay maior
        max_delay: 300.0              # Até 5 minutos
        backoff_multiplier: 2.0
        requests_per_minute: 2        # Muito conservador

performance:
  token_management:
    enable_chunking: true             # OBRIGATÓRIO para programas grandes
    max_tokens_per_chunk: 80000       # Chunks grandes
    overlap_tokens: 2000              # Sobreposição maior

    provider_specific:
      luzia:
        enable_chunking: true
        max_tokens_per_request: 150000

logging:
  level: DEBUG                        # Debug para acompanhar divisão
```

Comando de Execução:

Bash

```bash
python main.py --config config/config_programas_grandes.yaml \
               --fontes programas_gigantes.txt \
               --output analise_programas_grandes \
               --provider luzia
```

Características:

- Divisão automática de programas grandes
- Timeouts estendidos
- Rate limiting muito conservador

- Logs detalhados para acompanhamento

Configurações de Monitoramento

Configuração com Métricas Detalhadas

YAML

```yaml
# config/config_monitoramento.yaml
ai:
  primary_provider: "luzia"

  providers:
    luzia:
      retry:
        max_attempts: 5
        base_delay: 2.0
        max_delay: 60.0
        backoff_multiplier: 2.0
        requests_per_minute: 8

logging:
  level: INFO
  format: "%(asctime)s - %(name)s - %(levelname)s - [%(funcName)s:%
(lineno)d] - %(message)s"
  file: "logs/monitoramento.log"

monitoring:
  enable_metrics: true
  metrics_file: "metrics/performance.json"
  track_token_usage: true
  track_response_times: true
  track_success_rates: true
```

Script de Monitoramento

Bash

```bash
#!/bin/bash
# monitor_performance.sh

echo "=== Monitoramento COBOL AI Engine ===" > monitor.log
echo "Data: $(date)" >> monitor.log
echo "" >> monitor.log
```

```
# Executar análise com monitoramento
python main.py --config config/config_monitoramento.yaml \
               --fontes examples/fontes.txt \
               --output resultado_monitorado \
               2>&1 | tee -a monitor.log

# Extrair métricas dos logs
echo "" >> monitor.log
echo "=== MÉTRICAS EXTRAÍDAS ===" >> monitor.log
grep "Taxa de sucesso" logs/monitoramento.log | tail -1 >> monitor.log
grep "Total de tokens" logs/monitoramento.log | tail -1 >> monitor.log
grep "Tempo de processamento" logs/monitoramento.log | tail -1 >> monitor.log

echo "Monitoramento concluído. Ver monitor.log"
```

Configurações de Segurança

Configuração Segura para Produção

YAML

```yaml
# config/config_seguro.yaml
ai:
  primary_provider: "luzia"

  providers:
    luzia:
      # USAR APENAS VARIÁVEIS DE AMBIENTE
      client_id: "${LUZIA_CLIENT_ID}"
      client_secret: "${LUZIA_CLIENT_SECRET}"
      auth_url: "${LUZIA_AUTH_URL}"
      api_url: "${LUZIA_API_URL}"

      retry:
        max_attempts: 5
        base_delay: 2.0
        max_delay: 60.0
        backoff_multiplier: 2.0
        requests_per_minute: 8

security:
  mask_credentials_in_logs: true
  enable_audit_trail: true
  audit_file: "audit/access.log"
```

```yaml
logging:
  level: INFO
  # NÃO INCLUIR CREDENCIAIS NOS LOGS
  format: "%(asctime)s - %(name)s - %(levelname)s - %(message)s"
```

Script de Configuração Segura

Bash

```bash
#!/bin/bash
# setup_secure_env.sh

# Configurar variáveis de ambiente de forma segura
read -s -p "Digite o Client ID do LuzIA: " LUZIA_CLIENT_ID
echo
read -s -p "Digite o Client Secret do LuzIA: " LUZIA_CLIENT_SECRET
echo

# Exportar variáveis
export LUZIA_CLIENT_ID="$LUZIA_CLIENT_ID"
export LUZIA_CLIENT_SECRET="$LUZIA_CLIENT_SECRET"
export LUZIA_AUTH_URL="https://login.azure.pass.santanderbr.pre.corp"
export LUZIA_API_URL="https://prd-api-
aws.santanderbr.dev.corp/genai_services/v1"

echo "Variáveis configuradas com segurança!"

# Executar análise
python main.py --config config/config_seguro.yaml \
               --fontes programas_producao.txt \
               --output resultado_seguro
```

Troubleshooting por Cenário

Problema: Rate Limiting Muito Frequente

Sintomas:

Plain Text

```
Rate limit atingido. Aguardando 45s...
Rate limit detectado. Aguardando 8s...
```

Solução:

```yaml
retry:
  requests_per_minute: 4    # Reduzir de 8 para 4
  base_delay: 3.0           # Aumentar delay base
```

Problema: Timeouts Frequentes

Sintomas:

```
Timeout na requisição LuzIA (>180s)
```

Solução:

```yaml
timeout: 300                # Aumentar para 5 minutos
retry:
  max_attempts: 7           # Mais tentativas
  max_delay: 180.0          # Mais tempo entre tentativas
```

Problema: Análises Incompletas

Sintomas:

- Respostas muito curtas
- Análises superficiais

Solução:

```yaml
max_tokens: 8000            # Aumentar limite
temperature: 0.05           # Reduzir para mais consistência
model: "aws-claude-3-5-sonnet"  # Modelo mais capaz
```

Problema: Programas Muito Grandes

Sintomas:

```
```

```
Programa excede limite de tokens
```

Solução:

```yaml
performance:
  token_management:
    enable_chunking: true
    max_tokens_per_chunk: 100000
    overlap_tokens: 1000
```

Conclusão

O COBOL AI Engine v2.1.4 oferece configuração flexível para qualquer cenário:

- **Produção crítica**: Máxima confiabilidade
- **Desenvolvimento**: Velocidade e flexibilidade
- **Processamento em lote**: Sustentabilidade
- **Ambiente híbrido**: Multi-provedor
- **Programas grandes**: Divisão automática
- **Monitoramento**: Métricas detalhadas
- **Segurança**: Configuração segura

Escolha a configuração adequada ao seu cenário e ajuste conforme necessário.

COBOL AI Engine v2.1.4 - Exemplos de Configuração

Data: 15 de Setembro de 2025