# Source: C:\Users\Admin\Desktop\CMC\ms1_emailIngestion\core\polling_service.py

```
"""
Polling Service - Updated
Fetch emails và ■■y vào queue thay vì x■ lý tr■c ti■p
"""
import time
import threading
import requests
from typing import List, Dict, Optional
from datetime import datetime, timezone
from utils.config import MAX_POLL_PAGES as max_pages
from core.session_manager import session_manager, SessionState, TriggerMode
from core.queue_manager import get_email_queue
from core.token_manager import get_token


class PollingService:
    """
    Polling service - optimized with queue
    Ch■ fetch và enqueue, không x■ lý
    """

    GRAPH_URL = "https://graph.microsoft.com/v1.0"

    def __init__(self):
        self.active = False
        self.mode = TriggerMode.MANUAL
        self.interval = 300
        self.thread: Optional[threading.Thread] = None
        self.queue = get_email_queue()
        self._stop_event = threading.Event()

    def start(self, mode: TriggerMode = TriggerMode.SCHEDULED, interval: int = 300):
        """Kh■i ■■ng polling service"""
        if self.active:
            print(f"[PollingService] Already active in {self.mode.value} mode")
            return False

        self.mode = mode
        self.interval = interval
        self.active = True
        self._stop_event.clear()

        print(f"[PollingService] Starting in {mode.value} mode")
        print(f"[PollingService] Interval: {interval}s ({interval/60:.1f}min)")

        if mode == TriggerMode.SCHEDULED or mode == TriggerMode.FALLBACK:
            self.thread = threading.Thread(target=self._polling_loop, daemon=True)
            self.thread.start()

        return True

    def stop(self):
        """D■ng polling service"""
        if not self.active:
            return

        print(f"[PollingService] Stopping...")
        self.active = False
        self._stop_event.set()

        if (self.thread and self.thread.is_alive() and
            threading.current_thread() != self.thread):
            self.thread.join(timeout=5)

        print(f"[PollingService] Stopped")

    def poll_once(self) -> Dict:
        """
        Fetch emails và enqueue (không x■ lý)
        Processing s■ do BatchProcessor ■■m nh■n
        """
        try:
```

```
            print(f"[PollingService] Fetching unread emails...")
            start_time = time.time()

            # Fetch emails
            messages = self._fetch_unread_emails()
            fetch_time = time.time() - start_time

            if not messages:
                print(f"[PollingService] No unread emails found")
                return {
                    "status": "success",
                    "emails_found": 0,
                    "enqueued": 0,
                    "skipped": 0,
                    "fetch_time": fetch_time
                }

            print(f"[PollingService] Found {len(messages)} unread emails (took {fetch_time:.2f}s)")

            # Batch enqueue
            enqueue_start = time.time()
            emails_to_enqueue = [
                (msg.get("id"), msg, None)  # (id, data, priority)
                for msg in messages
            ]

            enqueued_ids = self.queue.enqueue_batch(emails_to_enqueue)
            enqueue_time = time.time() - enqueue_start

            enqueued = len(enqueued_ids)
            skipped = len(messages) - enqueued

            print(f"[PollingService] Enqueued {enqueued} emails (took {enqueue_time:.2f}s)")
            if skipped > 0:
                print(f"[PollingService] Skipped {skipped} emails (already processed/queued)")

            # Mark enqueued emails as read immediately
            if enqueued_ids:
                self._batch_mark_as_read(enqueued_ids)

            # Update session stats
            for msg in messages:
                msg_id = msg.get("id")
                if not session_manager.is_email_processed(msg_id):
                    session_manager.register_pending_email(msg_id)

            return {
                "status": "success",
                "emails_found": len(messages),
                "enqueued": enqueued,
                "skipped": skipped,
                "fetch_time": fetch_time,
                "enqueue_time": enqueue_time,
                "total_time": time.time() - start_time
            }

        except Exception as e:
            print(f"[PollingService] Poll error: {e}")
            session_manager.increment_polling_errors()
            return {
                "status": "error",
                "error": str(e),
                "emails_found": 0,
                "enqueued": 0,
                "skipped": 0
            }

    def _polling_loop(self):
        """Background loop cho scheduled/fallback polling"""
        print(f"[PollingService] Background polling started")

        while self.active and not self._stop_event.is_set():
            try:
                session_status = session_manager.get_session_status()
                current_state = SessionState(session_status["state"])

                # Vòng l■p này ch■ dành cho FALLBACK mode
```

```
                if self.mode != TriggerMode.FALLBACK:
                    print(f"[PollingService] Loop paused (mode: {self.mode.value}). Only runs in FALL
                    time.sleep(self.interval)
                    continue

                # Poll
                print(f"[PollingService] Fallback poll running...")
                self.poll_once()

                # Wait before next poll
                print(f"[PollingService] Waiting {self.interval}s until next poll...")
                self._stop_event.wait(timeout=self.interval)

            except Exception as e:
                print(f"[PollingService] Loop error: {e}")
                time.sleep(30)

        print(f"[PollingService] Background polling stopped")

    def _fetch_unread_emails(self, max_results: int = 100) -> List[Dict]:
        """
        Fetch unread emails from Graph API

        This method now supports full pagination to retrieve all unread emails,
        respecting the `MAX_POLL_PAGES` environment variable as a safeguard.
        """
        token = get_token()
        headers = {"Authorization": f"Bearer {token}"}
        all_messages = []
        page_count = 0

        # Initial URL and parameters
        url = f"{self.GRAPH_URL}/me/messages"
        params = {
            "$filter": "isRead eq false",
            "$top": max_results,
            "$orderby": "receivedDateTime desc"
        }

        while url and page_count < max_pages:
            try:
                resp = requests.get(url, headers=headers, params=params, timeout=30)
                # Params are only needed for the first request. Subsequent requests use the full next
                if params:
                    params = None

                if resp.status_code != 200:
                    print(f"[PollingService] API error during pagination: {resp.status_code} - {resp
                    break  # Exit loop on API error

                data = resp.json()
                messages = data.get("value", [])
                all_messages.extend(messages)

                page_count += 1
                url = data.get("@odata.nextLink")  # Get the next page link

                if url:
                    print(f"[PollingService] Fetched page {page_count}, more emails available...")
                else:
                    print(f"[PollingService] Fetched final page ({page_count}). No more pages.")
            except requests.exceptions.RequestException as e:
                print(f"[PollingService] Network error during pagination: {e}")
                break  # Exit loop on network error

        if page_count >= max_pages and url:
            print(f"[PollingService] WARN: Reached max poll pages limit ({max_pages}). More emails ma

        return all_messages

    def _batch_mark_as_read(self, email_ids: List[str]):
        """
        Mark a batch of emails as read using Microsoft Graph batching.
        This is more efficient than sending individual requests.
        """
        if not email_ids:
            return
```

```
        print(f"[PollingService] Marking {len(email_ids)} emails as read...")
        token = get_token()
        headers = {
            "Authorization": f"Bearer {token}",
            "Content-Type": "application/json"
        }

        batch_payload = {
            "requests": [
                {
                    "id": str(i + 1),
                    "method": "PATCH",
                    "url": f"/me/messages/{email_id}",
                    "body": {"isRead": True},
                    "headers": {"Content-Type": "application/json"}
                } for i, email_id in enumerate(email_ids)
            ]
        }

        try:
            response = requests.post(
                f"{self.GRAPH_URL}/$batch",
                headers=headers,
                json=batch_payload,
                timeout=60
            )
            response.raise_for_status()
            print(f"[PollingService] ✓ Successfully marked {len(email_ids)} as read.")
        except requests.exceptions.RequestException as e:
            print(f"[PollingService] ERROR: Failed to batch mark as read: {e}")

    def get_status(self) -> Dict:
        """L■y tr■ng thái hi■n t■i"""
        queue_stats = self.queue.get_stats()

        return {
            "active": self.active,
            "mode": self.mode.value if self.mode else None,
            "interval": self.interval,
            "thread_alive": self.thread.is_alive() if self.thread else False,
            "queue_size": queue_stats["queue_size"]
        }


# Singleton instance
polling_service = PollingService()
```