# Credit Card Detection Application

## **Overview**

**Goal**: Build a user-friendly application (e.g., runnable via Docker or Python script) that scans a system’s disk/folders, extracts text from many file types, and detects credit-card numbers reliably, generating a report of file paths and positions where cards appear.

**Key components**:

* **Text extraction**: from PDFs, Word (.docx), Excel (.xlsx/.xls), plain text (.txt/.csv/.log), HTML/XML, PowerPoint (.pptx), email (.msg), archives (.zip/.jar), images (OCR), EPUB, ODT, etc.
* **Detection engine**: Microsoft Presidio analyzer for CREDIT\_CARD (regex + Luhn + context). Also custom context-aware scoring via spaCy.
* **Pipeline**: walk directories, prefilter files, extract text in chunks, run detection, write CSV report.
* **Optimizations**: prefilter with raw-byte regex, multiprocessing with process pools, batching with maxtasksperchild, memory monitoring and dynamic batch sizing, streaming/chunking large files, exception handling for corrupted files, suppress noisy warnings/errors, checkpointing to resume long scans, CLI options to tune thresholds and weights.

### Tools and Methods Researched

**Bulk Extractor**:

Explored Bulk Extractor (versions 1.5, 2.x): it is fast at carving from raw disk images or directories, but during tests it did **not detect credit-card numbers placed in dummy text files or PDFs**. Likely because it scans raw data and looks for patterns in binary sectors but may not extract text properly from modern formats like PDF/docx. Therefore, not reliable for our purpose.

**Apache Tika**:

* Tried using Apache Tika server to extract text from files, then feed text to Presidio.
* Setup: ran Tika server (tika-server.jar) separately, invoked via HTTP PUT for each file.

**Issues**:

* Slow: extracting each file via HTTP calls incurred overhead.
* Inaccurate or empty results for some files (Tika sometimes returned empty or errors on corrupted files).
* Required running a separate Tika server process alongside scanner.
* Handling long text chunks via Tika slowed detection pipeline.

**Conclusion**: Tika approach worked but was slower and more complex. We moved to Python-based extractors (PyMuPDF, python-docx, openpyxl/xlrd, BeautifulSoup, etc.), which ran in-process and allowed finer control (chunking, exception handling, logging).

**Microsoft Presidio Analyzer**:

* Used Microsoft Presidio’s built-in CreditCardRecognizer, which applies regex + Luhn checksum + some minimal context checks.
* Observed that Presidio sometimes detected random numeric sequences (that pass Luhn) in ML data files, causing false positives.
* Explored Presidio’s Context-Aware Enhancer (LemmaContextAwareEnhancer) and custom PatternRecognizer with context keywords to boost detection when real context words appear near numbers.
* Built a custom context-weighted scoring function combining Presidio base\_score and spaCy-based context similarity to penalize matches lacking context.

**Python text-extraction packages**:

* **PDF**: PyMuPDF (fitz). Allows per-page text extraction, but sometimes throws errors on corrupted PDFs. We wrapped calls to suppress stderr and catch exceptions per page.
* **Word (.docx)**: python-docx to read paragraphs and table cells.
* **Excel (.xlsx)**: openpyxl in read-only mode, iterating worksheets and rows. Suppressed openpyxl UserWarnings for invalid specs.
* **Excel (.xls)**: xlrd for old-format .xls files.
* **PowerPoint (.pptx)**: python-pptx.
* **HTML/XML**: BeautifulSoup with lxml or fallback to html.parser. Suppressed XMLParsedAsHTMLWarning.
* **Plain text (.txt, .log, .csv, .tmp)**: read in streaming mode via csv.reader for .csv, reading raw for others, in chunks.
* **EPUB**: ebooklib + BeautifulSoup.
* **ODT**: odfpy.
* **Email (.msg)**: extract\_msg.
* **Images**: PIL + pytesseract for OCR, wrapped to suppress stderr if needed.
* **Archives (.zip/.jar/etc.)**: Python zipfile, extracting entries to temp dir and applying extractors recursively.
* **Other binary files**: skipped by extension whitelist.

## Implementation and Issues

**Early pipeline using Tika + Presidio**:

* Wrote PowerShell script (run\_parallel\_scan.ps1) to enumerate files, start Python process\_one\_file.py jobs in parallel (2–4 at a time).
* A python script, process\_one\_file.py sent file bytes to Tika server via HTTP PUT, got extracted text, ran Presidio analyzer, appended hits to CSV.

**Issues faced**:

* CSV remained empty: debugging revealed process\_one\_file.py sometimes didn’t write hits because detection threshold or code issues; PS jobs suppressed print statements; path or writing logic flawed.
* Tika sometimes returned empty text or slow performance.
* No obvious errors, but no hits due to threshold or analyzer not loaded with custom recognizer.
* Parallel jobs: output from print statements not visible; logs redirected to temp files.
* Fixed code unpacking return values, ensured analyzer loaded with correct recognizer, lowered thresholds, added logging to stderr for debugging. But overall Tika overhead remained high.

**Switch to Python-based extractors**:

* Created text\_extractor.py with extraction functions for each file type.
* Built initial scanner run.py and optimized\_pipeline.py using single-thread or ProcessPoolExecutor.

**Issues faced**:

* DOCX not detected sometimes: fixed code to iterate paragraphs and tables correctly.
* CSV parsing errors: some CSV files malformed; fixed by reading via csv.reader line-by-line with error handling, fallback to raw read.
* Excel warnings: openpyxl printed many UserWarnings about invalid specs; suppressed via warnings.filterwarnings.
* PyMuPDF printed many “MuPDF error: …” lines to stderr for corrupted PDFs, cluttering logs and slowing I/O; implemented context manager to suppress stderr around fitz.open and page.get\_text().
* Large or corrupted files: extraction hung or threw errors; added per-page and per-file exception handling and timeouts.
* .xls files: openpyxl could not open; added xlrd extractor for .xls.
* Archive scanning: nested zips/ODF packages; ensured recursion and skip corrupted entries.
* Image OCR: sometimes slow; only run when extension indicates image; wrapped to catch errors.
* After some time scanning large folder (e.g. tens of thousands of files), pipeline slowed or stuck: identified memory leaks or accumulation in worker processes; fixed by using maxtasksperchild in multiprocessing.Pool, calling gc.collect(), and periodically restarting pools.
* Directory walk: os.walk sometimes slower; switched to os.scandir recursion for efficiency.
* Checkpointing: long scans needed resume capability; added pickle-based checkpoint of processed files.
* Prefilter: raw-byte regex on first/last chunks to skip files unlikely to contain CC patterns, avoiding heavy extraction when unnecessary.
* Progress reporting: printed timestamps, file counts, ETA calculations; later replaced with tqdm progress bars in optimized version.
* Subprocess thread errors: certain libraries or custom subprocess calls produced “ValueError: read of closed file” in background threads; fixed by redirecting subprocess stdout/stderr to DEVNULL or using communicate() properly.

**Detection logic and Presidio**:

* Initially used Presidio’s built-in AnalyzerEngine().analyze(...) with default threshold. Observed many false positives (random numeric sequences) or misses (threshold too high).
* Added custom PatternRecognizer for full and partial CC patterns; ran into recognizer init errors (argument ordering); fixed by correct PatternRecognizer signature.
* Tested default vs custom: default built-in sometimes adequate, sometimes too permissive.
* Explored context-aware enhancers: added context keywords to PatternRecognizer, used LemmaContextAwareEnhancer; saw improved boosting when real context present but did not penalize lack of context enough.
* Designed combined scoring: use Presidio base detection with low threshold to gather candidates, then compute spaCy-based context similarity on windows around matches, combine scores with weights (context weight > pattern weight) to lower final score for matches lacking context. Implemented detect\_credit\_cards\_custom(...) accordingly.
* Tested custom scoring on sample texts: verified that sequences in context passed, standalone sequences filtered. But in practice, tuning parameters (weights, thresholds, window size) still needed: sometimes false positives or negatives remain. Observed default Presidio sometimes works better out-of-box, so default scanner remains primary.

### Optimization Techniques Implemented

**Prefilter with raw-byte regex**:

* For large files (>100 MB), read only first and last ~64 KB, run a compiled regex for 16-digit-like patterns. If no match, skip heavy parsing. This “fail-fast” avoids processing huge irrelevant files.

**Extension list & size filters**:

* Maintain a set of supported extensions (.pdf, .docx, .xlsx, .xls, .csv, .txt, .log, .html, .xml, .pptx, .epub, .odt, .msg, .png/.jpg/.jpeg, .zip/.jar, etc.).
* Skip unsupported extensions immediately.
* Skip or quickly prefilter very large files.

**Directory traversal via os.scandir**:

Replaced os.walk with a stack-based os.scandir recursion for faster file enumeration.

**Multiprocessing with process pools**:

* Use multiprocessing.Pool (or ProcessPoolExecutor) with initializer=init\_worker to load heavy models (spaCy, Presidio) once per worker.
* Use maxtasksperchild to restart workers periodically (e.g., every 100 tasks) to release leaked memory.
* Tune chunksize so that tasks are distributed efficiently without too much overhead.

**Chunked extraction**:

For large text segments (e.g., long extracted text from large files), split into chunks of ~50,000 characters, process each chunk separately. Limits memory per detection call and isolates matches.

**Timeouts per file/page**:

* For PDF: skip corrupted pages quietly, suppress stderr, catch exceptions.
* Enforce a max time per file (e.g., 60 seconds); if exceeded, abort processing that file to avoid hanging the entire scan.

**Suppress noisy warnings and stderr messages**:

* For PyMuPDF: wrap fitz.open and page.get\_text() in context manager that redirects sys.stderr to /dev/null, so MuPDF error messages do not flood console.
* For openpyxl: use warnings.filterwarnings("ignore", category=UserWarning, module="openpyxl") or wrap workbook loading in warnings.catch\_warnings().
* For BeautifulSoup XML warnings: suppress XMLParsedAsHTMLWarning.
* For subprocess calls (e.g., Tika, OCR): redirect stdout/stderr to DEVNULL or use communicate() to avoid background reader thread errors.

**Memory monitoring & dynamic batch sizing**:

* Use psutil to check system memory usage; if usage exceeds threshold (e.g., 80%), reduce batch size for subsequent batches to avoid OOM.
* After each batch, call gc.collect() to free Python-level garbage.
* Restrict number of worker processes based on available memory (~1 GB per worker as estimate).

**Checkpointing**:

* Maintain a pickle file of processed file paths; after each batch (or every N files), save checkpoint.
* On restart, load checkpoint and skip already processed files.

**Progress reporting**:

* Use tqdm progress bars to show number of files processed, estimated time remaining.
* Log per-batch and per-file timing (optional) for diagnostics.

**File-type-specific robust extractors**:

* Each extractor function catches exceptions per file/page/sheet/row to skip problematic parts without crashing pipeline.
* For CSV: use csv.reader to avoid pandas errors on malformed lines; fallback to raw read if parsing fails.
* For Excel .xls: use xlrd in streaming fashion, with per-sheet chunking and timeouts.
* For images: catch UnidentifiedImageError; wrap OCR calls in try/except and suppress errors.
* For archives: skip bad zip entries; use temporary directories for extraction.
* For HTML/XML: try lxml, fallback to html.parser if parsing errors occur.
* For epub/odt: catch errors on malformed packages.

**CLI usability**:

Both scanners (optimized\_scanner.py for default Presidio, custom\_scanner.py for context-aware) accept CLI arguments to:

* Set scan root directory.
* Enable/disable checkpointing.
* Exclude certain file extensions, e.g., exclude image files if not necessary to scan images.
* Tune detection parameters: thresholds, weights, window size.
  + Set batch size, max workers, max time per file.
* Logs written to a file (pipeline.log, custom\_scanner.log) for debugging.
* CSV report path fixed or configurable; includes headers.
* After scan, user can sort CSV by final\_score descending to see highest-confidence hits first.

## Final Implementation

**Default scanner (optimized\_scanner.py)**

* Uses Presidio built-in detect\_credit\_cards\_default:
  + Calls AnalyzerEngine.analyze(text, entities=["CREDIT\_CARD"], language="en", score\_threshold=threshold).
  + Default threshold can be tuned via CLI (e.g. --threshold 0.5).
* All the optimizations listed above apply (prefilter, chunking, multiprocessing, memory monitoring, suppression of warnings, checkpointing, robust extractors).
* CSV columns: timestamp, file, label, start, end, match.
* In tests, default scanner generally detected valid CC numbers reliably; tuning threshold helps reduce false positives or false negatives.
* Custom mode option was removed in final default scanner; always use default recognizer.

**Custom scanner (custom\_scanner.py)**

* Uses detect\_credit\_cards\_custom(...) from cc\_detector.py, which:
  + Runs Presidio built-in detection with a low base threshold to collect all Luhn-valid candidates.
  + For each match, extracts a text window around the match and computes spaCy-based semantic similarity against a “context\_doc” built from keywords (e.g., “credit”, “card”, “payment”, etc.).
  + Computes final\_score = w\_context \* context\_score + w\_pattern \* base\_score, with weights tunable via CLI.
  + Only returns matches whose final\_score >= final\_threshold.
* CSV columns: timestamp, file, label, start, end, match, base\_score, context\_score, final\_score.
* CLI options: --base-threshold, --final-threshold, --w-context, --w-pattern, --window-size, etc.
* All the same extraction and pipeline optimizations apply (prefilter, chunking, multiprocessing, memory monitoring, suppression of warnings, checkpointing).
* **Notes on tuning**: In practice, initial experiments showed custom scanner still had false positives (e.g., data files with numeric sequences) or false negatives (some real CC mentions missed if context words too far). Tuning base\_threshold, final\_threshold, and weights is necessary. The default scanner often worked better out-of-box, so custom scanner is available for further experimentation and fine-tuning.

### Issues Faced and Resolutions

1. **Bulk Extractor limitations**: Could not detect CC numbers in testing text/docx/pdf. Decided not to use it.
2. **Apache Tika overhead**: Tika server approach was slow and required separate process. Moved to Python in-process extractors.
3. **Empty report / CSV writing issues**: Early code did not unpack detection results or had threshold mismatches, so CSV stayed empty. Fixed by lowering thresholds and ensuring hits are written in scanner loop.
4. **DOCX, PDF, Excel parsing errors**:
   * DOCX extractor sometimes missed tables; updated to iterate paragraphs and table cells.
   * PDF extractor threw many MuPDF errors on corrupted pages; added stderr suppression and exception handling per page.
   * Excel .xlsx warnings cluttered logs; suppressed openpyxl warnings. .xls needed xlrd; added extractor.
5. **Slowdowns over time / pipeline getting stuck**:
   * Initially scanning large folders was fast, but after many files, memory grew and processes hung or slowed dramatically.
   * Added process pooling with maxtasksperchild, gc.collect(), memory monitoring, dynamic batch sizing to prevent leaks and overuse.
   * Prefilter to skip irrelevant large files.
   * Chunking large text to avoid huge memory spikes.
6. **CSV parsing errors**:
   * Some CSV files had malformed rows causing pandas.read\_csv errors; moved to csv.reader with fallback to raw read.
7. **Subprocess thread errors**:
   * Some libraries or code parts invoked subprocess incorrectly, causing “ValueError: read of closed file” in background threads. Fixed by redirecting stdout/stderr to DEVNULL or using communicate().
8. **Presidio detection accuracy**:
   * Default Presidio sometimes detected random Luhn-valid sequences. Added custom context-aware scoring in detect\_credit\_cards\_custom, though tuning remains ongoing.
   * Also experimented with adding PatternRecognizer for partial CC patterns (first 12 digits) to flag incomplete numbers.
9. **Parallel job debugging**:
   * Early PowerShell parallel jobs suppressed print output; added logging to file, used proper error handling so that exceptions are visible.
10. **File system nuances**:
    * Long file paths on Windows sometimes caused errors; handle path lengths or use long-path prefixes if needed.

**List of File Types Supported in the Final Implementation:**

* **Plain text**: .txt, .log, .ini, .cfg, .bak, .tmp, etc.
* **CSV**: .csv (streamed via csv.reader; fallback raw read).
* **PDF**: .pdf via PyMuPDF, per-page extraction.
* **Word**: .docx via python-docx.
* **Excel**: .xlsx via openpyxl; .xls via xlrd.
* **PowerPoint**: .pptx via python-pptx.
* **HTML/XML**: .html, .htm, .xml via BeautifulSoup.
* **EPUB**: .epub via ebooklib + BeautifulSoup.
* **ODT**: .odt via odfpy.
* **Email**: .msg via extract\_msg.
* **Image OCR**: .png, .jpg, .jpeg via PIL + pytesseract.
* **Archives**: .zip, .jar, etc.; recursively extract entries and apply extractors.
* **Other binary files**: skipped by extension whitelist or optionally prefilter raw bytes.

**Efficiency & Memory Improvements**

* **Prefilter raw bytes**: skip large irrelevant files quickly.
* **Chunking text**: process in ~50k-character chunks to limit memory per detection call.
* **Multiprocessing**: use worker pools, preload heavy models, restart periodically to free memory.
* **Dynamic batch sizing**: reduce batch size under high memory pressure.
* **Checkpoints**: resume long scans without restarting from scratch.
* **Suppress warnings/errors**: avoid I/O slowdown from printing many messages.
* **Timeouts**: abort overly slow file processing.
* **os.scandir**: faster directory traversal.
* **Selective OCR**: only on image extensions.
* **CSV streaming**: avoid reading entire large CSV into memory.
* **Excel streaming**: openpyxl in read-only mode; xlrd on .xls in on\_demand mode.
* **Garbage collection**: call gc.collect() after batches.
* **Logging to file**: avoid console clutter, inspect logs offline.

These combined made scans of thousands of files complete reliably without hanging, and kept memory usage stable

**CLI Usability**

* Both scanners (optimized\_scanner.py and custom\_scanner.py) accept arguments:
  + **scan root** directory.
  + **Checkpoint** toggle.
  + **Exclude extensions** list.
  + **Thresholds & weights**: base\_threshold, final\_threshold, w\_context, w\_pattern, window\_size.
  + **Batch size**, **max workers**, **max time per file**.
* **Logs** written to a log file for quiet operation.
* **Progress bars** show live progress.
* **Report** saved to CSV in working directory; can be sorted by confidence score after scan.
* Docker usage: wrap Python environment and scanner script in a container for users; minimal requirements: Docker installed, ~2 GB free RAM, Python 3 environment with required packages.

**Testing & Validation**

**Unit tests**: small scripts to verify:

* Extraction functions produce non-empty text for known files.
* Presidio built-in detection catches known test card numbers.
* Custom detection filters out standalone numeric sequences lacking context.

**Integration tests**: place sample files in a test folder:

* .txt, .pdf, .docx containing valid CC numbers in sentences with context words.
* Files with random numeric sequences that pass Luhn but lack context.
* Verify default scanner picks up true cases and not picks up false ones (tune threshold).
* Verify custom scanner behaviour and tune weights/thresholds.

**Performance tests**: scan a large directory tree (thousands of files) and observe memory, CPU, throughput. Ensure pipeline does not hang or consume too much RAM.

**Logging**: check logs for skipped files, timeouts, errors on corrupted files, to ensure coverage.

**Notes & Next Steps**

* **Custom scoring** works but requires careful tuning of base\_threshold, final\_threshold, w\_context, and w\_pattern per actual data. In experiments, default Presidio often performed better; custom mode remains for further tuning or specific contexts.
* **Further improvements** could include:
  + Logging summary statistics (e.g., number of candidate matches vs accepted matches).
  + Integration with a database or UI for reviewing flagged hits.
  + More sophisticated context models (e.g., fine-tune spaCy or use transformers for deeper semantics).
* **Dockerization**: prepare a Dockerfile installing Python, required packages, copying scanner scripts; users run container with volume mount to scan host directory. Ensure resource limits and permissions handled properly.
* **Security & Privacy**: ensure the scanner runs locally; extracted text may contain sensitive data, report only stores matches. Consider encrypting reports or handling them securely.

### Conclusion

Built a robust, efficient, and configurable file-scanner application for credit-card number detection. Key achievements:

* **Researched** multiple tools (Bulk Extractor, Apache Tika, Presidio) and settled on Python extractors + Presidio.
* **Implemented** robust extractors for many file types, handling corrupted files gracefully.
* **Built** scanning pipelines (optimized\_scanner.py for default detection; custom\_scanner.py for context-weighted detection).
* **Faced and resolved** issues: empty reports, parsing errors, slowdowns, memory leaks, noisy warnings.
* **Optimized** for speed and memory: prefilter, chunking, multiprocessing with worker restarts, memory monitoring, suppressed noise.
* **Enabled** CLI tuning of thresholds, weights, batch sizes, and checkpointing for long runs.
* **Tested** detection accuracy and performance; default scanner currently more reliable, custom scanner available for further tuning.
* **Prepared** for production use: logs, reports in CSV, potential Docker packaging, and documentation.