LAPORAN TUGAS BESAR 3
IF2211 STRATEGI ALGORITMA

oleh:

Kelompok 52 - K03

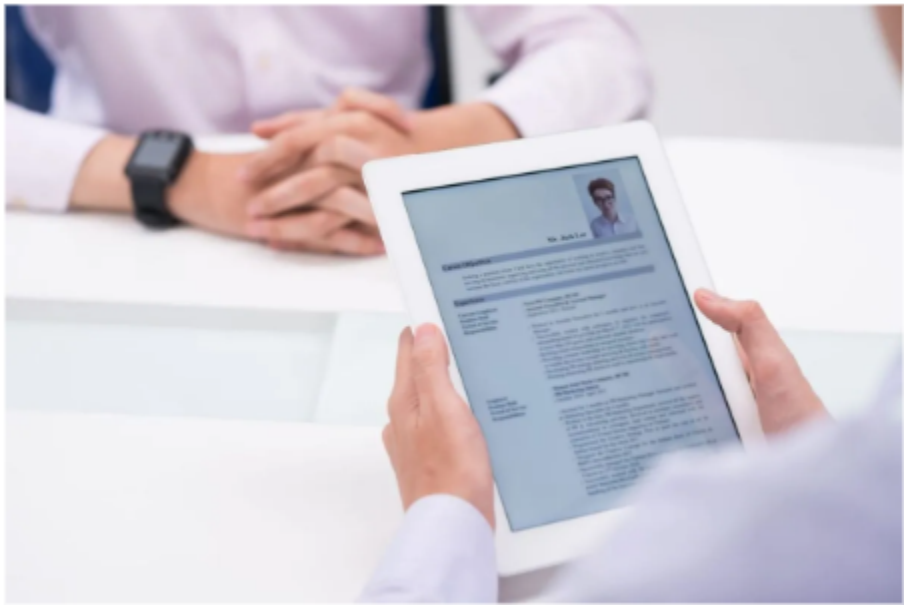
Rhio Bimo Prakoso Sugiyanto	**13523123**
Rafael Marchell Darmawijaya	**13523146**
Frederiko Eldad Mugiyono	**13523147**

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG

2025

DAFTAR ISI

BAB I
DESKRIPSI TUGAS



Gambar 1. CV ATS dalam Dunia Kerja 2
(sumber: https://www.antaranews.com/)

Di era digital ini, keamanan data dan akses menjadi semakin penting. Perkembangan proses rekrutmen tenaga kerja telah mengalami perubahan signifikan dengan memanfaatkan teknologi untuk meningkatkan efisiensi dan akurasi. Salah satu inovasi yang menjadi solusi utama adalah Applicant Tracking System (ATS), yang dirancang untuk mempermudah perusahaan dalam menyaring dan mencocokkan informasi kandidat dari berkas lamaran, khususnya Curriculum Vitae (CV). ATS memungkinkan perusahaan untuk mengelola ribuan dokumen lamaran secara otomatis dan memastikan kandidat yang relevan dapat ditemukan dengan cepat.

Meskipun demikian, salah satu tantangan besar dalam pengembangan sistem ATS adalah kemampuan untuk memproses dokumen CV dalam format PDF yang tidak selalu terstruktur. Dokumen seperti ini memerlukan metode canggih untuk mengekstrak informasi penting seperti identitas, pengalaman kerja, keahlian, dan riwayat pendidikan secara efisien. Pattern matching menjadi solusi ideal dalam menghadapi tantangan ini.

Pattern matching adalah teknik untuk menemukan dan mencocokkan pola tertentu dalam teks. Dalam konteks ini, algoritma Boyer-Moore dan Knuth-Morris-Pratt (KMP) sering digunakan karena keduanya menawarkan efisiensi tinggi untuk pencarian teks di dokumen besar. Algoritma ini memungkinkan sistem ATS untuk mengidentifikasi informasi penting dari CV pelamar dengan kecepatan dan akurasi yang optimal.

Di dalam Tugas Besar 3 ini, Anda diminta untuk mengimplementasikan sistem yang dapat melakukan deteksi informasi pelamar berbasis dokumen CV digital. Metode yang akan digunakan untuk melakukan deteksi pola dalam CV adalah Boyer-Moore dan Knuth-Morris-Pratt. Selain itu, sistem ini akan dihubungkan dengan identitas kandidat melalui basis data sehingga harapannya terbentuk sebuah sistem yang dapat mengenali profil pelamar secara lengkap hanya dengan menggunakan CV digital.

BAB II
LANDASAN TEORI

A. Algoritma Knuth-Morris-Pratt (KMP)

Algoritma Knuth-Morris-Pratt (KMP) merupakan salah satu metode pencocokan string yang dirancang untuk meningkatkan efisiensi pencarian pola dalam teks. Teknik ini menghindari perbandingan ulang karakter yang sebelumnya sudah dicocokkan, dengan memanfaatkan informasi internal dari pola itu sendiri. Inti dari KMP terletak pada tahapan preprocessing terhadap pola, di mana dibentuk sebuah array bantu bernama LPS (Longest Prefix Suffix). Array ini mencatat panjang prefiks terpanjang yang sekaligus merupakan sufiks hingga titik tertentu dalam pola.

Saat terjadi ketidakcocokan karakter dalam proses pencarian, algoritma tidak serta-merta memulai ulang pencocokan dari awal pola, melainkan menggunakan informasi dalam LPS untuk menentukan posisi baru di dalam pola tanpa perlu menggeser pointer dalam teks. Pendekatan ini menjaga kompleksitas waktu dalam pencarian tetap linear, yaitu O(n + m), dengan n adalah panjang teks dan m adalah panjang pola.

KMP sangat cocok diterapkan dalam pencarian kecocokan persis (exact matching) pada dokumen besar yang mengandung banyak pengulangan pola, karena efisiensi waktu dan minimnya backtracking.

B. Algoritma Boyer-Moore (BM)

Boyer-Moore adalah algoritma pencocokan string yang secara luas dianggap sangat efisien untuk pencarian satu pola dalam teks panjang, khususnya jika alfabet yang digunakan besar dan pola relatif panjang. Berbeda dari KMP, Boyer-Moore melakukan pencocokan karakter pola dari arah kanan ke kiri.

Salah satu komponen utama dalam algoritma ini adalah bad character heuristic, yaitu mekanisme yang memanfaatkan informasi tentang posisi terakhir kemunculan karakter tertentu dalam pola. Ketika ditemukan ketidakcocokan, algoritma menggunakan informasi ini untuk menggeser pola sejauh mungkin melewati bagian-bagian teks yang tidak mungkin menghasilkan kecocokan.

Hasilnya, Boyer-Moore mampu melompati sejumlah besar karakter dalam teks selama pencarian berlangsung. Namun, efisiensi optimal dari algoritma ini sangat dipengaruhi oleh keunikan karakter dalam pola. Jika pola mengandung banyak pengulangan karakter, efektivitas heuristik akan berkurang. Dengan demikian, Boyer-Moore sangat andal untuk teks panjang dan pola kompleks, namun perlu mempertimbangkan struktur karakter dari pola yang digunakan.

C. Algoritma Aho-Corasick

Aho-Corasick merupakan algoritma pencocokan string yang dirancang khusus untuk menangani kasus pencarian banyak pola (multi-pattern matching) dalam satu kali pemindaian teks. Algoritma ini menggabungkan struktur trie dengan mekanisme failure link yang serupa dengan pendekatan dalam KMP.

Langkah pertama dari Aho-Corasick adalah membentuk sebuah trie dari seluruh pola yang ingin dicari. Setiap node dalam trie merepresentasikan karakter, dan node yang merupakan akhir dari pola akan menyimpan output berupa pola tersebut. Setelah trie terbentuk, dibuatlah struktur failure function yang memungkinkan peralihan node secara efisien saat terjadi ketidakcocokan, tanpa harus kembali ke akar.

Selama pencarian, teks diproses satu arah, dan semua pola yang cocok akan langsung dikenali berdasarkan output node aktif dalam trie. Karena pemrosesan dilakukan satu kali untuk seluruh teks, algoritma ini sangat efisien dan stabil bahkan dalam pencarian terhadap ratusan hingga ribuan pola. Aho-Corasick ideal untuk sistem seperti ATS (Applicant Tracking System) yang perlu mencari banyak keyword secara simultan dari dokumen-dokumen besar.

D. Fuzzy Matching

Fuzzy matching adalah pendekatan dalam pencocokan string yang memungkinkan pendeteksian kecocokan tidak persis (inexact match) antara dua string berdasarkan tingkat kemiripan tertentu. Berbeda dengan algoritma pencocokan eksak yang hanya mengenali string yang benar-benar identik, fuzzy matching tetap dapat mengidentifikasi string yang memiliki kesamaan meskipun terdapat perbedaan kecil, seperti kesalahan ketik, perubahan urutan huruf, atau variasi bentuk kata.

Konsep utama dalam fuzzy matching didasarkan pada pengukuran jarak antar string, yang disebut sebagai string distance. Salah satu metode paling umum yang digunakan adalah *Levenshtein Distance*, yaitu jumlah minimum operasi edit (penyisipan, penghapusan, atau penggantian karakter) yang diperlukan untuk mengubah satu string menjadi string lainnya.

Sebagai contoh, kata "developer" dan "develper" memiliki *Levenshtein Distance* sebesar 1, karena hanya satu karakter yang perlu ditambahkan untuk menjadikannya identik. Semakin kecil nilai jarak ini, semakin mirip kedua string tersebut.

Selain *Levenshtein Distance*, metode lain seperti *Jaro-Winkler*, *Soundex*, dan *cosine similarity* juga digunakan tergantung pada konteks dan jenis data yang diolah. Dalam konteks aplikasi pencarian CV, fuzzy matching sangat berguna untuk menangani kasus di mana kandidat mungkin menuliskan keterampilan atau istilah pekerjaan dengan variasi

ejaan atau singkatan. Misalnya, pencarian kata kunci “JavaScript” tetap dapat menemukan dokumen yang menuliskan “Javascript”, “java script”, atau bahkan “JS”.

Penggunaan fuzzy matching memastikan sistem tetap toleran terhadap variasi penulisan, sehingga meningkatkan cakupan dan kualitas hasil pencarian. Dengan demikian, fuzzy matching menjadi solusi penting untuk pemrosesan teks dalam dunia nyata, di mana data input sering kali tidak sempurna, dan pencocokan eksak saja tidak cukup untuk menemukan informasi yang relevan secara semantik.

E. Aplikasi

Aplikasi yang dibuat adalah aplikasi GUI berbasis Python yang dibangun menggunakan *library* PyQt6, dengan tujuan utama melakukan pencarian kata kunci secara efisien pada berkas CV dalam format PDF. Ketika aplikasi dijalankan, seluruh berkas CV akan diproses menjadi representasi string (untuk proses pencocokan) dan format mentah (untuk keperluan tampilan dan ringkasan informasi). Proses ini dilakukan secara in-memory tanpa menyimpan hasil ekstraksi ke dalam berkas tambahan.

Pengguna diberikan opsi untuk memasukkan satu atau beberapa pola kata kunci, menentukan algoritma pencarian (KMP, Boyer-Moore, Aho-Corasick, fuzzy, dsb.), serta menetapkan jumlah hasil CV yang ingin ditampilkan berdasarkan tingkat kecocokan terbanyak. Sistem akan melakukan pencarian dengan pendekatan exact matching terlebih dahulu. Jika hasil relevan tidak ditemukan, maka algoritma fuzzy matching akan dijalankan sebagai alternatif.

Setelah proses pencarian selesai, pengguna dapat melihat waktu yang diperlukan untuk setiap metode pencarian yang dijalankan, baik exact maupun fuzzy. Selain fitur pencarian, aplikasi ini menyediakan antarmuka tambahan untuk melihat ringkasan informasi dari setiap CV yang ditemukan, termasuk nama, keahlian, riwayat pekerjaan, dan pendidikan, yang diperoleh dari database SQL terintegrasi. CV yang relevan juga dapat langsung dibuka dalam format PDF melalui tombol interaktif di antarmuka. Dengan seluruh fitur ini, aplikasi mendukung pencarian dokumen secara cepat, akurat, dan informatif.

BAB III
ANALISIS PEMECAHAN MASALAH

Pada bab ini akan dibahas langkah-langkah dan komponen utama dalam merancang sistem pencarian resep di aplikasi CV finder. Setiap poin menguraikan aspek penting mulai dari pemetaan masalah hingga penentuan strategi pencarian

A. Mapping Persoalan CV Finder

Sistem CV Finder dirancang untuk menyelesaikan permasalahan pencarian dan pencocokan CV digital secara otomatis menggunakan teknik pattern matching. Pemetaan persoalan dilakukan dengan mengidentifikasi komponen-komponen utama sistem dan hubungan antar komponen tersebut. Sistem CV Finder dirancang secara berlapis dengan empat lapisan utama. Pada lapisan ekstraksi data, dokumen diekstrak menggunakan pustaka PyPDF2. Lalu, dengan menerapkan pola regex khusus, sistem mengidentifikasi dan mengekstrak informasi terstruktur seperti nama, alamat, nomor telepon, keahlian, pengalaman kerja, dan riwayat pendidikan. Kemudian sistem akan mempersiapkan teks dalam dua format berbeda, format asli untuk regex dan format linear untuk pattern matching.

Pada lapisan penyimpanan data, sistem menyimpan profil pelamar dan detail aplikasi menggunakan skema relasional. Sistem juga menyediakan abstraksi untuk operasi CRUD dan manajemen koneksi database. Sebagai proteksi tambahan, sistem menggunakan metode enkripsi yang disebut sebagai Cellular Automata Encryption (CAE) yang memanfaatkan *cellular automaton* dengan aturan modifikasi dari Conway's Game of Life.

Pada lapisan pencarian, mengimplementasikan tiga algoritma pencarian string: Knuth-Morris-Pratt (KMP), Boyer-Moore, dan Aho-Corasick. Ketika tidak dapat ditemukan *exact match*, sistem akan menggunakan Levenshtein Distance untuk menangani variasi ejaan dan kesalahan ketik. Selain itu, *package* pencarian menyediakan pengaturan fleksibel untuk sensitivitas huruf, jumlah hasil maksimal, dan kemiripan.

Pada lapisan antarmuka, GUI dibangun menggunakan PyQt6 dengan komponen modern dan responsif. GUI juga menjalankan operasi berat (preprocessing dan pencarian) pada thread terpisah untuk menjaga responsivitas. GUI akan menampilkan hasil pencarian, statistik kinerja, dan ringkasan CV.

B. Langkah-langkah Pemecahan Persoalan

Masalah utama yang ingin diselesaikan dalam proyek ini adalah menemukan string tertentu menggunakan regex serta mencari keyword dalam kumpulan CV. Berikut langkah-langkah detailnya:

a. PDF Reader

PDF Reader berfungsi sebagai fondasi utama sistem CV-Finder dalam mengekstraksi dan mempersiapkan data teks dari dokumen CV digital. Proses ini dirancang dengan pendekatan *dual-format extraction* yang menghasilkan dua versi teks berbeda untuk memenuhi kebutuhan spesifik setiap tahap pemrosesan dalam sistem. Implementasi PDF Reader memanfaatkan pustaka PyPDF2 untuk membaca dan mengekstraksi konten teks dari berkas PDF, dengan penanganan khusus untuk berbagai format dan struktur dokumen CV yang beragam.

Tahapan ekstraksi dimulai dengan pembacaan file PDF secara langsung ke dalam memori menggunakan fungsi extract_text_from_pdf yang memproses seluruh halaman dokumen secara sekuensial. Hasil ekstraksi awal berupa teks mentah kemudian diproses melalui fungsi prepare_texts_from_pdf yang menghasilkan dua format teks berbeda. Format pertama adalah *regex text* yang mempertahankan struktur asli dokumen dengan format, spasi, dan tata letak yang utuh. Format kedua adalah *pattern text* yang merupakan representasi linear dari seluruh teks dengan normalisasi karakter, konversi ke huruf kecil, dan penghapusan pemisah baris untuk mengoptimalkan proses *pattern matching*.

Sistem *preprocessing* yang terintegrasi dalam PDF Reader menerapkan strategi *caching* melalui fungsi save_extracted_texts yang menyimpan kedua format teks sebagai file .txt terpisah. Mekanisme ini memungkinkan sistem untuk menghindari pemrosesan berulang pada dokumen yang sama.

b. KMP

Algoritma Knuth-Morris-Pratt (KMP) merupakan salah satu algoritma pencarian string untuk melakukan *exact pattern matching* dalam sistem CV Finder. Algoritma ini dirancang khusus untuk mengatasi kelemahan algoritma pencarian naif yang melakukan backtracking pada teks ketika terjadi ketidakcocokan karakter. Implementasi KMP dalam KMPSearcher menggunakan pendekatan preprocessing melalui pembangunan tabel Longest Proper Prefix which is also Suffix (LPS) untuk mengoptimalkan proses pencarian dan menghindari pembandingan karakter yang redundan.

Proses kerja algoritma KMP dapat dibagi menjadi dua tahap utama yang saling berkaitan. Tahap pertama adalah *preprocessing pattern* melalui fungsi _generate_lps_list yang membangun tabel LPS. Tabel LPS menyimpan panjang prefix terpanjang dari substring pattern[0..i] yang juga merupakan suffix dari substring yang sama. Sebagai contoh, untuk pattern "ABCDABCE", tabel LPS akan berisi [0, 0, 0, 0, 1, 2, 3, 0] dimana nilai pada posisi ke-6 adalah 3 karena "ABC" merupakan prefix dan suffix terpanjang dari "ABCDABC".

Tahap kedua adalah pencarian yang dilakukan oleh fungsi _kmp_search yang memanfaatkan informasi LPS untuk melakukan pencarian yang efisien.

Algoritma ini menggunakan dua pointer, yaitu text_index untuk posisi dalam teks dan pattern_index untuk posisi dalam pattern. Ketika kedua karakter pada posisi pointer cocok, kedua pointer bergerak maju secara bersamaan. Namun ketika terjadi ketidakcocokan, algoritma menggunakan informasi dari tabel LPS untuk menentukan posisi selanjutnya dalam pattern yang perlu dibandingkan. Jika pattern_index tidak berada di awal pattern, algoritma akan mengatur pattern_index ke nilai lps[pattern_index - 1], yang secara efektif memanfaatkan kecocokan parsial yang telah ditemukan sebelumnya.

Kompleksitas waktu algoritma KMP terdiri dari dua komponen utama yang berjalan secara berurutan. Fase preprocessing untuk membangun tabel LPS memiliki kompleksitas O(m) dimana m adalah panjang pattern, dimana setiap karakter dalam pattern dikunjungi maksimal dua kali. Fase pencarian utama memiliki kompleksitas O(n) dimana n adalah panjang teks, karena text_index hanya bergerak maju dan tidak pernah mundur, sementara pattern_index dapat mundur tetapi total pergerakan mundurnya dibatasi oleh kemajuan text_index. Dengan demikian, kompleksitas waktu total untuk pencarian single pattern adalah O(n + m), yang merupakan kompleksitas optimal untuk string matching.

Untuk pencarian multiple pattern dengan k pattern, kompleksitas waktu menjadi O(k × (n + m_avg)) dimana m_avg adalah rata-rata panjang pattern. Kompleksitas ruang algoritma KMP adalah O(m) untuk menyimpan tabel LPS, ditambah O(1) untuk variabel tambahan, menjadikan algoritma ini sangat memory-efficient.

c. Boyer-Moore

Algoritma Boyer-Moore merupakan salah satu algoritma pencarian string yang dirancang untuk mengoptimalkan pencarian pattern yang relatif panjang dalam teks berukuran besar. Dalam sistem CV Finder, algoritma Boyer-Moore berfungsi sebagai alternatif exact matching yang memberikan keunggulan signifikan dibandingkan algoritma pencarian konvensional melalui pendekatan pencarian dari kanan ke kiri (right-to-left scanning) dan pemanfaatan heuristik untuk melompati bagian teks yang tidak relevan. Implementasi Boyer-Moore dalam BoyerMooreSearcher mengadopsi teknik preprocessing yang membangun tabel heuristik yang memungkinkan algoritma melakukan skip ketika menemukan ketidakcocokan karakter.

Algoritma Boyer-Moore menggunakan pencarian yang berbeda dengan melakukan pembandingan karakter dari kanan pattern ke kiri, berlawanan dengan pendekatan left-to-right yang umum digunakan. Proses ini dimulai dengan fase preprocessing yang membangun dua tabel heuristik utama melalui fungsi seperti _build_bad_character_table. Bad Character Heuristic menyimpan informasi posisi

terakhir setiap karakter dalam pattern, memungkinkan algoritma untuk menentukan seberapa jauh pattern dapat digeser ketika menemukan karakter yang tidak cocok. Sebagai contoh, jika pattern "EXAMPLE" sedang dicari dalam teks dan terjadi mismatch pada karakter 'T' yang tidak ada dalam pattern, algoritma dapat langsung menggeser pattern sejauh panjang pattern karena tidak ada kemungkinan match di posisi tersebut.

Proses pencarian aktual yang dilakukan oleh fungsi _boyer_moore_search dimulai dengan menempatkan pattern pada awal teks dan membandingkan karakter dari posisi terakhir pattern. Ketika semua karakter cocok dari kanan ke kiri, algoritma mencatat posisi sebagai match dan melanjutkan pencarian. Namun ketika terjadi ketidakcocokan, algoritma memanfaatkan informasi dari bad character table untuk menentukan shift yang optimal. Jika karakter yang mismatch ada dalam pattern, algoritma akan menggeser pattern sehingga karakter tersebut dalam sejajar dengan karakter mismatch dalam teks. Jika karakter tidak ada dalam pattern, algoritma dapat melakukan shift yang lebih agresif dengan memindahkan pattern melewati posisi karakter tersebut sepenuhnya. Implementasi dalam sistem CV Finder juga mengintegrasikan Good Suffix Heuristic sebagai optimisasi tambahan, meskipun bad character heuristic saja sudah memberikan peningkatan kinerja yang baik untuk sebagian besar kasus pencarian CV.

Kompleksitas waktu algoritma Boyer-Moore memiliki karakteristik yang unik dibandingkan algoritma string matching lainnya. Dalam best case scenario, ketika pattern dan teks memiliki distribusi karakter yang sparse dan pattern relatif panjang, algoritma dapat mencapai kompleksitas O(n/m) dimana n adalah panjang teks dan m adalah panjang pattern. Hal ini terjadi karena algoritma dapat melakukan skip sebesar panjang pattern pada setiap iterasi ketika menemukan karakter yang tidak ada dalam pattern. Namun dalam worst case, kompleksitas dapat mencapai O(n × m), terutama ketika teks dan pattern memiliki banyak karakter berulang yang menyebabkan minimal skip pada setiap mismatch.

Kompleksitas preprocessing untuk membangun bad character table adalah O(m + σ) dimana σ adalah ukuran alphabet, yang dalam konteks teks CV biasanya mencakup karakter ASCII standar. Untuk pencarian multiple pattern dengan k pattern, total kompleksitas menjadi O(k × (m_avg + σ) + total_search_time) dimana preprocessing dilakukan terpisah untuk setiap pattern. Kompleksitas ruang algoritma Boyer-Moore adalah O(m + σ) untuk menyimpan pattern dan bad character table.

d. Aho-Corasick

Algoritma Aho-Corasick merupakan algoritma pencarian string yang paling optimal untuk multi-pattern matching, dirancang khusus untuk menangani skenario pencarian dimana banyak pattern harus dicari secara sekaligus dalam teks yang sama. Dalam sistem CV Finder, algoritma ini menjadi pilihan utama ketika pengguna memasukkan multiple keywords sekaligus. Implementasi Aho-Corasick dalam AhoCorasickSearcher mengadopsi pendekatan tiga fase: pembangunan trie, konstruksi failure links, dan pencarian dengan finite automaton.

Fondasi algoritma Aho-Corasick terletak pada struktur data TrieNode yang merupakan upgrade dari trie tradisional dengan penambahan failure links. Setiap simpul dalam trie menyimpan dictionary children untuk memetakan karakter ke simpul anak, failure pointer yang menunjuk ke state alternatif ketika terjadi mismatch, output list yang berisi semua pattern yang berakhir di simpul tersebut, dan flag is_end untuk menandai terminasi pattern.

Kelas AhoCorasick mengimplementasikan finite automaton yang menyimpan simpul root, daftar pattern asli untuk referensi, dan flag untuk tracking status pembangunan failure links. Fungsi add_pattern membangun trie dengan menambahkan setiap pattern, karakter demi karakter, menciptakan simpul baru bila diperlukan dan menandai simpul terminal dengan pattern yang sesuai. Proses ini menghasilkan trie dimana pattern-pattern dengan prefix yang sama berbagi jalur yang sama di awal.

Kompleksitas algoritma Aho-Corasick dapat dianalisis dalam tiga fase terpisah yang masing-masing memiliki karakteristik yang berbeda. Fase pembangunan trie memiliki kompleksitas $O(\Sigma m_i)$ dimana Σm_i adalah total panjang semua pattern, karena setiap karakter dari setiap pattern harus diproses tepat satu kali. Fase pembangunan failure links memiliki kompleksitas $O(\Sigma m_i)$ meskipun tampak seperti $O(\Sigma m_i^2)$ karena loop while dalam BFS, tetapi setiap simpul dikunjungi maksimal sebanyak panjang path dari root, dan total semua path tidak melebihi total panjang pattern.

Fase pencarian memiliki kompleksitas $O(n + z)$ dimana n adalah panjang teks dan z adalah jumlah total match yang ditemukan. Kompleksitas ini optimal untuk multi-pattern matching karena tidak bergantung pada jumlah pattern yang dicari, berbeda dengan pendekatan naif yang akan memiliki kompleksitas $O(k \times n)$ untuk k pattern. Kompleksitas ruang adalah $O(\Sigma m_i + \sigma \times |states|)$ dimana σ adalah ukuran alphabet dan |states| adalah jumlah state dalam automaton, yang proporsional dengan total panjang pattern karena trie “sharing” prefix yang sama.

e. Fuzzy Matching

Algoritma Fuzzy Matching merupakan komponen penting dalam sistem CV Finder yang dirancang untuk mengatasi keterbatasan exact matching. Implementasi fuzzy matching dalam FuzzySearcher mengadopsi Levenshtein Distance sebagai parameter utama untuk mengukur kemiripan antar string, dengan mekanisme fallback melalui integrasi dengan SearchEngine. Algoritma ini menjadi solusi vital ketika recruiter mencari skill seperti "JavaScript" tetapi kandidat menulisnya sebagai "Java Script", "Javascript", atau bahkan dengan typo minor seperti "JavaScrpit", memastikan bahwa kandidat yang relevan tidak terlewatkan.

Levenshtein Distance, yang diimplementasikan dalam fungsi _levenshtein_distance, merupakan parameter edit distance yang mengukur jumlah minimum operasi single-character editing yang diperlukan untuk mengubah satu string menjadi string lain. Tiga jenis operasi yang diizinkan adalah insertion (penyisipan karakter baru), deletion (penghapusan karakter), dan substitution (penggantian karakter). Sebagai contoh, untuk mengubah "PYTHON" menjadi "PYTHN", diperlukan satu operasi deletion untuk menghilangkan karakter 'O', sehingga Levenshtein Distance-nya adalah 1. Algoritma ini menggunakan pendekatan dynamic programming dengan membangun matrix dua dimensi dimana setiap sel [i,j] merepresentasikan edit distance minimum antara prefix pertama i karakter dari string pertama dengan prefix pertama j karakter dari string kedua.

Implementasi yang digunakan dalam sistem CV-Finder mengadopsi optimisasi space-efficient dimana hanya dua row dari matrix yang disimpan dalam memori pada satu waktu, mengurangi kompleksitas ruang dari O(m×n) menjadi O(min(m,n)). Algoritma juga memastikan bahwa string yang lebih pendek selalu menjadi string pertama untuk mengoptimalkan penggunaan memori. Proses perhitungan dilakukan dengan mengisi matrix secara row-by-row, dimana setiap sel dihitung berdasarkan minimum dari tiga kemungkinan operasi: mengambil nilai dari cell diagonal kiri-atas (substitution), cell atas (deletion), atau cell kiri (insertion), dengan menambahkan cost 1 untuk setiap operasi kecuali ketika karakter pada posisi tersebut sama.

Fungsi _similarity_ratio bertanggung jawab untuk mengkonversi raw edit distance menjadi similarity score antara 0.0 hingga 1.0, dimana 1.0 menunjukkan kecocokan sempurna dan 0.0 menunjukkan tidak ada kemiripan sama sekali. Formula yang digunakan adalah 1.0 - (edit_distance / max_length), dimana max_length adalah panjang string terpanjang dari kedua string yang dibandingkan.

Mekanisme pencarian fuzzy yang sesungguhnya diimplementasikan dalam _fuzzy_search_text menggunakan sliding window approach. Algoritma tidak hanya mencari exact substring matches, tetapi mengevaluasi semua possible substring dalam teks dengan panjang yang bervariasi berdasarkan similarity threshold. Strategi window sizing menghitung max_window berdasarkan rumus query_length × (1.0 / min_similarity), memungkinkan pencarian substring yang lebih panjang ketika similarity threshold lebih rendah. Sebaliknya, min_window dihitung sebagai query_length × min_similarity untuk memastikan tidak ada pemborosan komputasi pada substring yang terlalu pendek untuk mencapai threshold yang ditetapkan.

Kompleksitas waktu algoritma Levenshtein Distance yang diimplementasikan adalah O(m×n) dimana m dan n adalah panjang kedua string yang dibandingkan, meskipun optimisasi space-efficient mengurangi konstanta yang terlibat. Untuk fuzzy search keseluruhan, kompleksitas menjadi O(T×W×P×max(Q,W)) dimana T adalah panjang teks, W adalah rata-rata window size, P adalah jumlah patterns, dan Q adalah rata-rata panjang pattern. Window size sendiri bergantung pada similarity threshold dan panjang pattern, sehingga threshold yang lebih rendah akan menghasilkan window yang lebih besar dan computational cost yang lebih tinggi.

Kompleksitas ruang dioptimalkan menjadi O(min(m,n)) untuk setiap operasi Levenshtein Distance calculation, ditambah O(R) untuk menyimpan hasil sementara dimana R adalah jumlah matches yang ditemukan.

f. Cellular Automata Encryption

Cellular Automata Encryption (CAE) merupakan sistem enkripsi yang mengimplementasikan konsep cellular automata sebagai generator keystream untuk enkripsi simetris dalam sistem CV Finder. Algoritma ini dirancang untuk mengamankan informasi sensitif pelamar, khususnya path file CV yang disimpan dalam database, dengan memanfaatkan sifat chaos dan unpredictability dari cellular automata untuk menghasilkan keystream yang cryptographically strong. Implementasi CAE dalam CAE mengadopsi Counter (CTR) mode operation yang memungkinkan enkripsi data dengan panjang arbitrary.

Sistem CAE dibangun dengan arsitektur modular yang terdiri dari beberapa komponen. Komponen pertama adalah key derivation mechanism yang diimplementasikan dalam _stretch_key, menggunakan PBKDF2-like approach dengan SHA-256 sebagai hash function untuk mengkonversi password user menjadi master key dengan panjang yang sesuai dengan block size (256 bytes). Proses key stretching melibatkan iterative hashing dengan kombinasi password, salt, dan block index untuk menghasilkan multiple blocks yang kemudian

di-concatenate untuk membentuk master key. Parameter iterations yang dapat dikonfigurasi (default 1,000,000) memberikan protection terhadap brute force attacks dengan meningkatkan computational cost untuk setiap password attempt.

Komponen kedua adalah counter-based key diversification melalui _apply_counter yang mengimplementasikan CTR mode semantic. Setiap block data yang dienkripsi menggunakan unique keystream yang digenerate dengan mengaplikasikan counter value ke master key. Counter di-encode sebagai 8-byte big-endian integer dan di-XOR dengan 8 byte terakhir dari master key, memastikan bahwa setiap block memiliki seed yang berbeda untuk cellular automata simulation.

Inti dari sistem CAE terletak pada cellular automata simulation dengan custom rule set. Proses dimulai dengan konversi seed bytes menjadi 16x16 grid melalui _convert_bytes_to_grid, dimana setiap byte dalam 256-byte block dipetakan ke cell dalam grid secara row-major order. Grid ini kemudian menjadi initial state untuk cellular automata yang akan berevolusi selama sejumlah generasi yang telah ditentukan (default 10 generations).

Rule set yang diimplementasikan dalam _apply_ca_rules merupakan hybrid antara Conway's Game of Life dan custom cryptographic rules yang dirancang untuk memaksimalkan diffusion dan confusion properties. Algoritma mengevaluasi 3x3 neighborhood kernel untuk setiap cell, dengan core value sebagai cell center dan 8 neighboring cells. Rule set membedakan antara "live" cells (value > 128) dan "dead" cells (value ≤ 128), dengan transition rules yang berbeda untuk masing-masing state. Untuk live cells, survival bergantung pada jumlah live neighbors (2-3 neighbors untuk survival, otherwise decay), sementara birth rules untuk dead cells dimodifikasi dengan factor generation number untuk meningkatkan temporal complexity.

Proses evolusi cellular automata dijalankan melalui _run_ca_simulation yang mengimplementasikan synchronous update dengan boundary conditions yang wrapped (toroidal topology). Setiap generasi, seluruh grid di-update berdasarkan state generasi sebelumnya. Hasil akhir setelah 10 generasi adalah grid yang telah mengalami sufficient mixing dan evolution untuk menghasilkan keystream yang cryptographically secure.

Untuk setiap plaintext block, sistem melakukan sequence operations yang deterministik namun cryptographically complex. Counter di-apply ke master key untuk menghasilkan unique seed, seed ini kemudian di-convert menjadi initial grid, cellular automata simulation dijalankan untuk menghasilkan evolved grid, dan grid final di-flatten menjadi keystream bytes. Plaintext block kemudian di-XOR dengan keystream untuk menghasilkan ciphertext block. Proses ini diulangi untuk semua blocks, dengan hasil akhir berupa concatenation dari salt

dan seluruh ciphertext blocks yang di-encode menggunakan Base64 untuk safe transmission dan storage.

Fungsi decrypt mengimplementasikan proses reverse yang identik, dimana Base64-encoded data di-decode, salt di-extract, dan master key di-regenerate menggunakan salt dan password yang sama. Setiap ciphertext block kemudian di-decrypt dengan me-regenerate keystream yang identik menggunakan counter yang sesuai, memastikan bahwa XOR operation menghasilkan original plaintext. Symmetry dari XOR operation memastikan bahwa dekripsi menghasilkan plaintext yang identik dengan input asli.

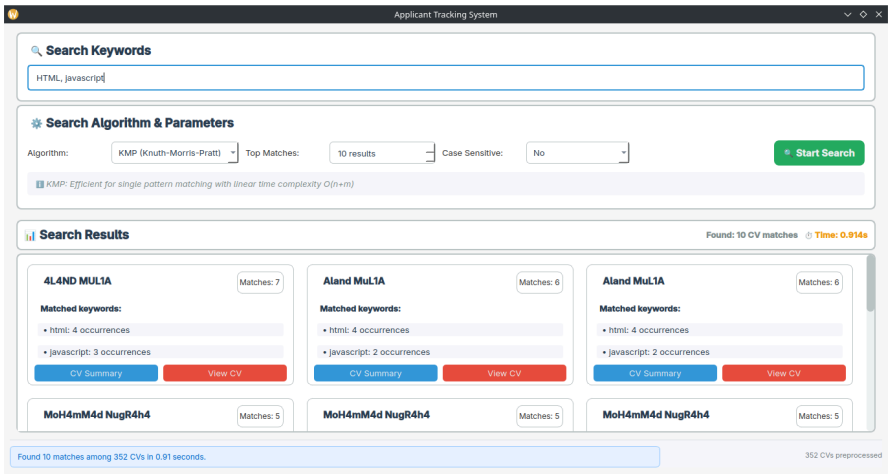
C. Fitur Fungsional dan Arsitektur Aplikasi yang Dibangun.

Dalam aplikasi kami, arsitektur yang digunakan berbasis model Single Page Application (SPA) dibangun menggunakan bahasa Python beserta pengimplementasian basis data menggunakan MySql. SPA memungkinkan halaman aplikasi dimuat satu kali, lalu semua interaksi dan navigasi dilakukan secara dinamis tanpa perlu memuat ulang seluruh halaman. Backend memproses permintaan pencarian *keyword(s)* dengan algoritma pencarian string matching yang sudah dipilih dan merespons dengan data CV dengan nilai string matching tertinggi. Arsitektur ini memungkinkan responsivitas yang tinggi serta pemrosesan yang efisien.

Fitur-fitur fungsional yang terdapat pada aplikasi kami di antaranya:

1. Pencarian CV dengan Berbagai Algoritma

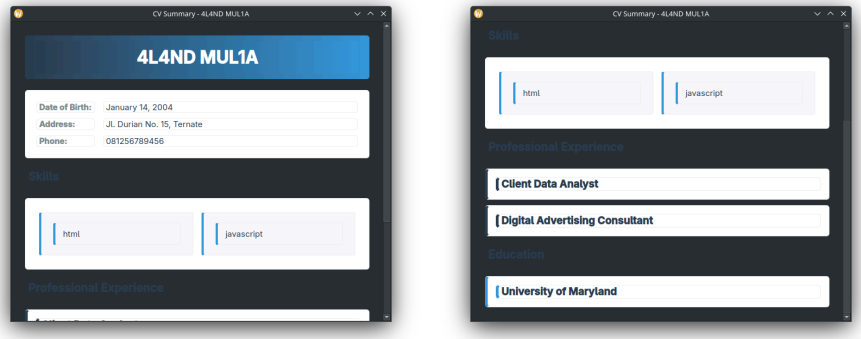
   Fitur utama dari aplikasi ini adalah kemampuannya untuk membantu pengguna dalam menemukan kandidat yang relevan dari database CV yang tersedia. Pengguna diberikan fleksibilitas untuk memasukkan satu atau lebih kata kunci yang dipisahkan koma. Untuk optimasi dan adaptasi terhadap berbagai skenario pencarian, pengguna dapat memilih metode pencarian yang diinginkan dari serangkaian algoritma string matching yang efisien.



Gambar 6. Tampilan laman pencarian
(sumber: Arsip kelompok)

2. Ringkasan CV Otomatis Berbasis Regex

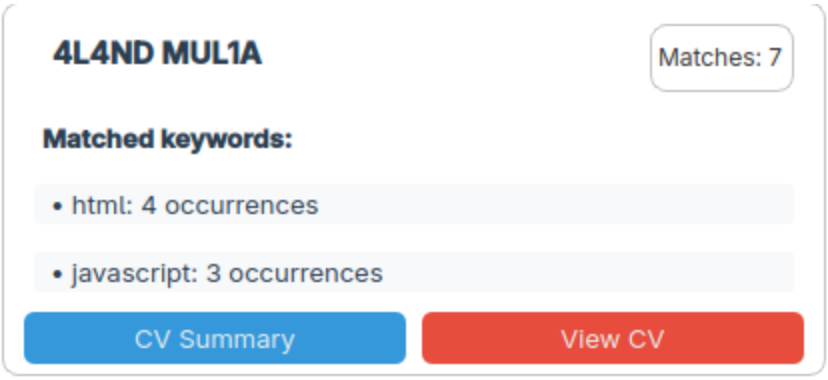
Salah satu fitur inovatif dari aplikasi ini adalah kemampuan untuk menghasilkan ringkasan otomatis dari CV yang dipilih. Fitur ini dirancang untuk mempercepat proses screening kandidat dengan menyajikan poin-poin penting CV tanpa perlu membaca dokumen lengkap.



Gambar 7. CV Summary
(sumber: Arsip kelompok)

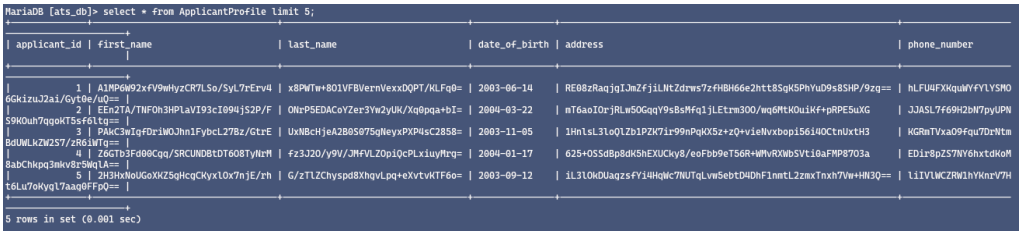
3. Peninjauan CV Langsung dalam Format PDF

Melengkapi fitur pencarian dan ringkasan, aplikasi ini juga menyediakan fungsionalitas untuk meninjau dokumen CV asli secara langsung.



Gambar 8. Tombol View CV
(sumber: Arsip kelompok)

4. Enkripsi Data Applicant
   Untuk menjadikan aplikasi ATS yang berintegritas, tentunya perlu disertakan suatu enkripsi data agar, apabila data jatuh ke tangan yang tidak tepat. Data tersebut tetap aman karena sudah terenkripsi. Enkripsi data dilakukan ketika startup aplikasi dilakukan secara *background process*.



Gambar 9. Tangkapan data yang sudah terenkripsi
(sumber: Arsip kelompok)

D. Ilustrasi Kasus

Ilustrasi kasus:

1. HR ingin mencari pelamar yang memiliki kata kunci HTML, Javascript
2. HR memilih salah satu dari metode pencocokan (KMP, BM, BM-complex, AHO Corasick, dan Fuzzy)
3. HR memilih berapa banyak hasil pencocokan yang ingin ditampilkan
4. HR memilih apakah ingin mencocokan dengan memperhatikan huruf kapital atau tidak
5. HR menekan tombol “Search"

6. Program menjalankan proses pencocokan dengan metode KMP:
    i. Pecah input kata kunci yang dipisahkan dengan koma “,”.
    ii. Untuk setiap kata kunci yang telah dipecahkan:
        1. Jalankan *worker thread* yang akan mencocokan hasil dari text cv yang disimpan dalam memori secara KMP.
        2. Apabila seluruh *worker thread* sudah memproses, *notify main thread*.
    iii. *Main thread* akan mengurutkan hasil berdasarkan skor kemiripan
7. Program menampilkan CV yang memiliki kata kunci HTML, Javascript sebanyak yang diminta HR.
8. HR menekan tombol *view* CV
9. Program membuka file PDF CV
10. HR menekan tombol *view* CV *summary*
11. Program mengolah data CV menggunakan RegEx dan menampilkan kepada HR.

BAB IV
IMPLEMENTASI DAN PENGUJIAN

A. Spesifikasi Teknis Program

Program dibuat dengan arsitektur aplikasi web pada umumnya yaitu *gui component*, *search algorithms*, *service*, dan sistem basis data. Seluruh program dibangun menggunakan bahasa Python dan basis data-nya menggunakan MySQL.

```
TUBES3_QTPY_COBA/
├── data/
├── src/
│   ├── config/
│   │   └── config.py
│   ├── database/
│   │   ├── ingest.py
│   │   ├── models.py
│   │   ├── parser.py
│   │   ├── pdf_utils.py
│   │   ├── setup.py
│   │   ├── <ats_db>.sql
│   ├── encryption/
│   │   ├── CAE.py
│   ├── gui_components/
│   │   ├── cv_summary_window.py
│   │   ├── general_config.py
│   │   ├── header.py
│   │   ├── input.py
│   │   ├── result.py
│   │   └── search.py
│   ├── search_algorithms/
│   │   ├── aho_corasick.py
│   │   ├── boyer_moore.py
│   │   ├── fuzzy_searcher.py
│   │   ├── kmp_searcher.py
│   │   ├── pattern_searcher.py
│   │   └── search_engine.py
│   └── service/
│       ├── encryptservice.py
│       ├── searchservice.py
│       ├── service_provider.py
│       └── threadservice.py
├── main.py
└── ...
```

Bagian *search-algorithms* dipisah menjadi beberapa file untuk meningkatkan modularitas program dan mempermudah proses implementasi.

a. Config

Secara keseluruhan, modul ini memusatkan semua parameter lingkungan dan path penting agar komponen lain—seperti bagian ingestion, model, atau service—cukup mengimpor DB_CONN, CV_FOLDER, dan variabel-variabel lain tanpa perlu hard-code path atau kredensial di beberapa tempat (dapat melihat README repository).

config.py

```
import os
from dotenv import load_dotenv

PROJECT_ROOT =
os.path.abspath(os.path.join(os.path.dirname(__file__), "../.."))

load_dotenv()

DB_CONFIG = {
    "user": os.getenv("DB_USER", "root"),
    "password": os.getenv("DB_PASSWORD", ""),
    "host": os.getenv("DB_HOST", "localhost"),
    "port": os.getenv("DB_PORT", "3306"),
    "database": os.getenv("DB_NAME", "ats_db"),
}

DB_CONN =
f"mysql+pymysql://{DB_CONFIG['user']}:{DB_CONFIG['password']}@" \

f"{DB_CONFIG['host']}:{DB_CONFIG['port']}/{DB_CONFIG['database']}"

CV_FOLDER = os.path.join(PROJECT_ROOT, "./data")

KAGGLE_USER = os.getenv("KAGGLE_USERNAME", "")
KAGGLE_KEY = os.getenv("KAGGLE_KEY", "")

DB_PATH = os.path.join(PROJECT_ROOT, "./src/database")

ENCRYPTION_PASSWORD = os.getenv("ENCRYPT_PASSWORD", "admin")
```

Secara singkat, config bekerja dengan membaca file '*.env*' di root proyek dan memasukkan variabel-variabelnya ke dalam environment Python.

b. Database

Database ini berisi struktur-struktur data yang digunakan dalam program beserta program seeding dan SQL database yang dimiliki preogram.

ingest.py

```
import pandas as pd
import kagglehub
import os
from datetime import datetime, timedelta
import random
from faker import Faker
from .models import ApplicationDetail, ApplicantProfile,
SessionLocal
from ..config.config import KAGGLE_USER, KAGGLE_KEY

fake = Faker('id_ID')

def load_kaggle():
    """Load Kaggle dataset with caching to avoid redownloading"""
    print("Loading data from Kaggle...")

    # Define the known path to the CSV file
    home_dir = os.path.expanduser("~")
    cached_csv_path = os.path.join(home_dir,
".cache/kagglehub/datasets/snehaanbhawal/resume-dataset/versions/1
/Resume/Resume.csv")

    # Check if the CSV file already exists
    if os.path.exists(cached_csv_path):
        print(f"Using cached file: {cached_csv_path}")
        df = pd.read_csv(cached_csv_path)
        print(f"Loaded {len(df)} records from cached file")
        return df

    # If not found, download from Kaggle
    print("Cached file not found. Downloading from Kaggle...")
    path =
kagglehub.dataset_download("snehaanbhawal/resume-dataset")
    print(f"Dataset downloaded to: {path}")

    # Find CSV files in the downloaded path
    csv_files = []
    for root, _, files in os.walk(path):
```

```
        for file in files:
            if file.endswith('.csv'):
                csv_files.append(os.path.join(root, file))

    if not csv_files:
        raise FileNotFoundError(f"No CSV files found in {path}")

    # Load the first CSV file found
    csv_path = csv_files[0]
    df = pd.read_csv(csv_path)
    print(f"Loaded {len(df)} records from
{os.path.basename(csv_path)}")

    return df

def generate_random_date(start_year=1980, end_year=2002):
    start_date = datetime(start_year, 1, 1)
    end_date = datetime(end_year, 12, 31)
    time_between_dates = end_date - start_date
    days_between_dates = time_between_dates.days
    random_days = random.randrange(days_between_dates)
    return start_date + timedelta(days=random_days)

def seed_from_csv():
    try:
        print("Loading data from Kaggle...")
        try:
            df = load_kaggle()
        except Exception as e:
            print(f"Failed to load data from Kaggle: {str(e)}")
            return False

        db = None
        try:
            db = SessionLocal()
        except Exception as e:
            print(f"Failed to create database session: {str(e)}")
            return False

        applicant_map = {}

        try:
            print("Seeding Applicant Profiles...")
            profile_count = random.randint(50, 100)
```

```
            for i in range(profile_count):
                try:
                    profile = ApplicantProfile(
                        first_name = fake.first_name(),
                        last_name  = fake.last_name(),
                        date_of_birth = generate_random_date(),
                        address    = fake.address().replace('\n',
', '),
                        phone_number =
fake.numerify(text="+62-8##-####-####")
                    )

                    db.add(profile)
                    db.flush()

                    applicant_map[i] = profile.applicant_id
                except Exception as e:
                    print(f"Error creating profile
{i+1}/{profile_count}: {str(e)}")

            db.commit()
            print(f"Created {len(applicant_map)} applicant
profiles.")

            if len(applicant_map) == 0:
                print("No applicant profiles were created. Cannot
continue.")
                return False

            print("Seeding Application Details...")
            application_count = 0
            for i, (_, row) in enumerate(df.iterrows()):
                try:
                    r = ApplicationDetail(
                        applicant_id =
applicant_map[random.randint(0, len(applicant_map) - 1)],
                        application_role = row["Category"],
                        cv_path = row["ID"]
                    )
                    db.add(r)
                    application_count += 1
                except Exception as e:
                    print(f"Error creating application
{i+1}/{len(df)}: {str(e)}")
```

```
            db.commit()
            print(f"Created {application_count} application
details.")

            return True

        except Exception as e:
            print(f"An unexpected error occurred during database
seeding: {str(e)}")

            if db:
                db.rollback()
            return False

    finally:
        if 'db' in locals() and db:
            db.close()
            print("Database connection closed.")

if __name__ == "__main__":
    print("Starting database seeding...")
    if seed_from_csv():
        print("Database successfully seeded!")
    else:
        print("Database seeding failed")
```

models.py

```
from sqlalchemy import (
    Column, ForeignKey, Integer, String, Date, Text, text,
create_engine
)
import os
import subprocess
from datetime import datetime
from sqlalchemy.orm import declarative_base, relationship,
sessionmaker
from ..config.config import DB_CONN

Base = declarative_base()
```

```
class ApplicantProfile(Base):
    __tablename__ = "ApplicantProfile"

    applicant_id     = Column(Integer, primary_key=True,
autoincrement=True)
    first_name       = Column(String(50))
    last_name        = Column(String(50))
    date_of_birth    = Column(Date)
    address          = Column(String(255))
    phone_number     = Column(String(20))

    applications = relationship(
        "ApplicationDetail",
        back_populates="applicant",
        cascade="all, delete-orphan"
    )

class ApplicationDetail(Base):
    __tablename__ = "ApplicationDetail"
    detail_id        = Column(Integer, primary_key=True,
autoincrement=True)
    applicant_id     = Column(Integer,
ForeignKey('ApplicantProfile.applicant_id'), nullable=False)
    application_role = Column(String(100))
    cv_path          = Column(Text)

    applicant = relationship("ApplicantProfile",
back_populates="applications")



# create an engine and session factory
engine = create_engine(DB_CONN, echo=True, future=True)
SessionLocal = sessionmaker(bind=engine, expire_on_commit=False)

def init_db() -> bool:
    try:
        Base.metadata.create_all(bind=engine)
        # Test connection by executing a simple query
        with SessionLocal() as session:
            # Simple query just to verify connection works
            session.execute(text('SELECT 1'))
        print("Database initialized successfully!")
```

```
        return True
    except Exception as e:
        print(f"Database initialization failed: {str(e)}")
        return False

def dump_db(output_path=None, schema = True) -> bool:
    """
    Export the database to a SQL file
    """
    try:
        # Extract database configuration from connection string
        from urllib.parse import urlparse
        parsed = urlparse(DB_CONN)

        # Extract components from the connection string
        username = parsed.username
        password = parsed.password
        host = parsed.hostname
        port = parsed.port or 3306
        database = parsed.path.strip('/')

        # Default output path
        if not output_path:
            timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
            output_path = f"{database}_backup_{timestamp}.sql"

        # Build mysqldump command
        cmd = f"mysqldump -h {host} -P {port} -u {username}
--password={password} {'-d' if schema else ''} --databases
{database} > {output_path}"

        # Execute command
        process = subprocess.Popen(
            cmd,
            shell=True,
            stdout=subprocess.PIPE,
            stderr=subprocess.PIPE
        )
        stdout, stderr = process.communicate()

        if process.returncode != 0:
            print(f"Export failed: {stderr.decode()}")
            return False
```

```
        print(f"Database {"schema " if schema else ""}exported
successfully to: {output_path}")
        print(f"File size: {os.path.getsize(output_path) /
(1024*1024):.2f} MB")
        return True

    except Exception as e:
        print(f"Database export failed: {str(e)}")
        return False
```

parser.py

```
from typing import List

import re

class SectionScraper:
    '''Scrapes sections out of PDF dumps'''

    skill_sections = [
        "skills",
        "skill highlights",
        "summary of skills",
    ]

    experience_sections = [
        "work history",
        "work experience",
        "experience",
        "professional experience",
        "professional history",
    ]

    education_sections = [
        "education",
        "education and training",
        "educational background",
        "teaching experience",
        "corporate experience"
    ]

    sections = skill_sections + experience_sections +
```

```
education_sections + [
        "summary",
        "highlights",
        "professional summary",
        "core qualifications",
        "languages",
        "professional profile",
        "relevant experience",
        "affiliations",
        "certifications",
        "qualifications",
        "accomplishments",
        "additional information",
        "core accomplishments",
        "career overview",
        "core strengths",
        "interests",
        "professional affiliations",
        "online profile",
        "certifications and trainings",
        "credentials",
        "personal information",
        "career focus",
        "executive profile",
        "military experience",
        "community service",
        "presentasions",
        "publications",
        "community leadership positions",
        "license",
        "computer skills",
        "presentations",
        "volunteer work",
        "awards and publications",
        "activities and honors",
        "volunteer associations",
    ]

    def remove_prefix(text: str, prefix: str) -> str:
        if prefix:
            if text.lower().startswith(prefix.lower()):
                return text[len(prefix):]
        return text
```

```
    def remove_suffix(text: str, suffix: str) -> str:
        if suffix:
            if text.lower().endswith(suffix.lower()):
                return text[:-len(suffix)]
        return text

    def _read(self, source) -> str:
        text = source

        return text

    def scrape_skills(self, source: str) -> str:
        text = self._read(source)
        res =
re.search(f"\n({'|'.join(self.skill_sections)})(\n.*)+?(\n({'|'.jo
in(self.sections)})\n|$)", text, re.IGNORECASE)
        if res:
            i, j = res.span()
            content: str =
SectionScraper.remove_prefix(text[i:j].strip(), res.groups()[0])
            for header in SectionScraper.sections:
                content = SectionScraper.remove_suffix(content,
header)

            # Transform bullet points to comma separated list
            output = ", ".join(content.strip().split("\n"))

            return output
        else:
            return "Not Found"

    def scrape_experience(self, source: str) -> str:
        text = self._read(source)
        res =
re.search(f"\n({'|'.join(self.experience_sections)})(\n.*)+?(\n({'
|'.join(self.sections)})\n|$)", text, re.IGNORECASE)
        if res:
            i, j = res.span()
            content: str =
SectionScraper.remove_prefix(text[i:j].strip(), res.groups()[0])
            for header in SectionScraper.sections:
                content = SectionScraper.remove_suffix(content,
header)
```

```
            # cheating by detecting "company name" to detect lines
containing jobs
            lines = content.splitlines()
            output = []
            keywords = [
                "Director",
                "Manager",
                "Analyst",
                "Specialist",
                "Recruiter",
                "Representative",
                "Coordinator",
                "Lead",
                "Consultant",
                "Volunteer",
                "Assistant",
                "Technician",
                "Supervisor",
                "Associate",
                "Intern",
                "Counselor",
                "Advocate"
            ]
            for line in lines:
                for keyword in keywords:
                    if keyword in line.strip():
                        output.extend(re.findall(f"[A-Z][a-zA-Z
]*{keyword}[a-zA-Z ]*", line.strip()))
                        break
                    elif "company name" != line.lower().strip()
and "company name" in line.lower().strip():
                        output.append(line)

            if output:
                return ", ".join(list(set(output)))
            else:
                return "Not Found"
        else:
            return "Not Found"

    def scrape_education(self, source: str) -> str:
        text = self._read(source)
        res =
re.search(f"\n({'|'.join(self.education_sections)})(\n.*)+?(\n({'|
```

```
'.join(self.sections)})\n|$)", text, re.IGNORECASE)
        if res:
            i, j = res.span()
            content =
SectionScraper.remove_prefix(text[i:j].strip(), res.groups()[0])
            for header in SectionScraper.sections:
                content = SectionScraper.remove_suffix(content,
header)

            # scrape universities
            regex: List[str] = []
            regex.extend(re.findall("university of [a-zA-Z ]+",
content, re.IGNORECASE))
            regex.extend(re.findall("[a-zA-Z ]+ university",
content, re.IGNORECASE))
            regex.extend(re.findall("[a-zA-Z ]+ college", content,
re.IGNORECASE))
            regex.extend(re.findall("college of [a-zA-Z ]+",
content, re.IGNORECASE))
            regex.extend(re.findall("([a-zA-Z ]* institute of
[a-zA-Z ]+|[a-zA-Z ]+ institute)", content, re.IGNORECASE))
            regex.extend(re.findall("[a-zA-Z ]+ high school",
content, re.IGNORECASE))
            regex.extend(re.findall("[a-zA-Z ]+ seminary",
content, re.IGNORECASE))
            regex.extend(re.findall("[a-zA-Z ]+ center", content,
re.IGNORECASE))
            regex.extend(re.findall("[a-zA-Z ]+ training program",
content, re.IGNORECASE))

            if regex:
                return ", ".join(list(set(regex)))
            else:
                return "Not Found"
        else:
            return "Not Found"

SectionScraper.sections.sort(key=lambda s: len(s), reverse=True)
```

pdf_utils.py

```
import PyPDF2
```

```python
import os
import re

def extract_text_from_pdf(pdf_path):
    """Extract text from PDF file using PyPDF2"""
    text = ""
    try:
        with open(pdf_path, 'rb') as file:
            pdf_reader = PyPDF2.PdfReader(file)
            for page in pdf_reader.pages:
                text += page.extract_text() + "\n"
    except Exception as e:
        print(f"Error reading PDF {pdf_path}: {e}")
        return ""
    return text

def clean_text(text):
    text = text.strip().replace("\n", " ").replace("\r",
"").replace("Â", "").replace("ï1/4", "")  # Remove unwanted
characters
    text = re.sub(r'\s+', ' ', text)
    return text

def prepare_texts_from_pdf(pdf_path: str) -> tuple[str, str] |
None:
    """
    Fungsi utama yang baru: Mengekstrak dan mempersiapkan teks
dari PDF.
    Langsung mengembalikan dua versi teks tanpa menyimpan ke file.

    Returns:
        Sebuah tuple (full_text_for_regex, text_for_pattern) jika
berhasil,
        atau None jika gagal.
    """
    raw_text = extract_text_from_pdf(pdf_path)

    if not raw_text:
        return None

    regex_text = clean_text(raw_text)

    text_for_pattern = raw_text.replace('\n', ' ').replace('\r', '
').strip().lower()
```

```
    return regex_text, text_for_pattern

def save_extracted_texts(pdf_path, output_regex, output_pattern):
    """Extract and save text in two formats: raw and linear"""
    # Create directories if they don't exist
    os.makedirs(os.path.dirname(output_regex), exist_ok=True)
    os.makedirs(os.path.dirname(output_pattern), exist_ok=True)

    text = prepare_texts_from_pdf(pdf_path)

    if not text:
        print(f"No text extracted from {pdf_path}")
        return False

    regex_text, pattern_text = text

    # Simpan untuk regex (APA ADANYA, persis aslinya)
    with open(output_regex, 'w', encoding='utf-8') as f:
        f.write(regex_text)

    with open(output_pattern, 'w', encoding='utf-8') as f:
        f.write(pattern_text)

    print(f"Text extracted and saved: {output_regex},
{output_pattern}")
    return True
```

setup.py

```
import os
import mysql.connector
from mysql.connector import Error
from dotenv import load_dotenv
import sys

load_dotenv()

def setup_db() -> bool:
    DB_HOST = os.getenv("DB_HOST", "localhost")
    DB_USER = os.getenv("DB_USER", "root")
    DB_PASSWORD = os.getenv("DB_PASSWORD", "")
    DB_NAME = os.getenv("DB_NAME", "ats_db")
```

```
    DB_PORT = os.getenv("DB_PORT", "3306")

    try:
        conn = mysql.connector.connect(
            host=DB_HOST,
            user=DB_USER,
            password=DB_PASSWORD,
            port=DB_PORT
        )

        if conn.is_connected():
            cursor = conn.cursor()
            cursor.execute(f"DROP DATABASE IF EXISTS {DB_NAME}")
            cursor.execute(f"CREATE DATABASE {DB_NAME}")
            cursor.execute(f"USE {DB_NAME}")
            print(f"Database '{DB_NAME}' is ready.")

            cursor.close()
            conn.close()

            return True
    except Error as e:

        print(f"Error setting up database: {e}")
        return False

if __name__ == "__main__":
    print("Setting up the database...")
    if setup_db():
        print("Database setup completed successfully.")
    else:
        print("Database setup failed.")
        sys.exit(1)
```

c. Encryption

CAE.py

```
import base64
import hashlib
from math import ceil
import os

class CAE:
```

```
    """
    A class to encapsulate the Cellular Automata Encryption (CAE)
system.
    This implementation uses Counter (CTR) mode for secure
encryption of any length.
    """
    def __init__(self, iterations=1_000, grid_size=16,
generations=10):
        self.iterations = iterations
        self.grid_size = grid_size
        self.block_size = grid_size * grid_size  # 16x16 = 256
bytes
        self.master_key_length = self.block_size # Master key must
match block size
        self.generations = generations
        self.salt_bytes = 16

    def _stretch_key(self, password: bytes, salt: bytes,
iteration: int, key_length: int) -> bytes:
        """Stretches the password and salt to create a master key
of the desired length."""
        master_key = b''
        digest_size = hashlib.sha256().digest_size
        num_blocks = ceil(key_length / digest_size)

        for block_index in range(1, num_blocks + 1):
            stretched_key_block = hashlib.sha256(password + salt +
block_index.to_bytes(4, 'big')).digest()
            for _ in range(iteration):
                stretched_key_block =
hashlib.sha256(stretched_key_block + password + salt).digest()
            master_key += stretched_key_block

        return master_key[:key_length]

    def _apply_counter(self, key: bytes, counter: int) -> bytes:
        """Applies a counter to a key to produce a unique seed for
each block (CTR mode)."""
        temp_key = bytearray(key)
        counter_bytes = counter.to_bytes(8, 'big')

        # XOR the counter into the last 8 bytes of the key copy
        for i in range(8):
            temp_key[self.block_size - 8 + i] ^= counter_bytes[i]
```

```
        return bytes(temp_key)

    def _convert_bytes_to_grid(self, byte_chunk: bytes) ->
list[list[int]]:
        """Converts a 256-byte chunk into a 16x16 grid of
integers."""
        return [[byte_chunk[i * self.grid_size + j] for j in
range(self.grid_size)] for i in range(self.grid_size)]

    def _apply_ca_rules(self, kernel: list[list[int]],
generation_num: int) -> int:
        """Calculates a cell's next state based on its 3x3 kernel
and custom rules."""
        core_value = kernel[1][1]

        live_neighbors = 0
        for i in range(3):
            for j in range(3):
                if i == 1 and j == 1:
                    continue
                if kernel[i][j] > 128:
                    live_neighbors += 1

        if core_value > 128:
            if live_neighbors in [2, 3]:
                new_value = min(255, core_value + live_neighbors)
            else:
                new_value = max(0, core_value - live_neighbors *
2)
        else:
            if live_neighbors == 3:
                new_value = min(255, core_value + live_neighbors *
10)
            else:
                new_value = (core_value + live_neighbors *
generation_num) % 256

        return new_value

    def _run_ca_simulation(self, seed_grid: list[list[int]]) ->
list[list[int]]:
        """Runs the CA simulation for a fixed number of
generations."""
```

```
        current_grid = seed_grid

        for generation_num in range(1, self.generations + 1):
            new_grid = [[0 for _ in range(self.grid_size)] for _
in range(self.grid_size)]
            for r in range(self.grid_size):
                for c in range(self.grid_size):
                    kernel = [[0 for _ in range(3)] for _ in
range(3)]
                    for i in range(-1, 2):
                        for j in range(-1, 2):
                            neighbor_r = (r + i) % self.grid_size
                            neighbor_c = (c + j) % self.grid_size
                            kernel[i + 1][j + 1] =
current_grid[neighbor_r][neighbor_c]

                    new_grid[r][c] = self._apply_ca_rules(kernel,
generation_num)
            current_grid = new_grid

        return current_grid

    def _flatten_grid_to_bytes(self, grid: list[list[int]]) ->
bytes:
        """Converts a 16x16 grid back into a 256-byte
keystream."""
        return bytes([cell for row in grid for cell in row])

    def encrypt(self, plaintext: str, password: str) -> str:
        """Encrypts a plaintext string using CTR mode."""
        # 1. Setup
        plaintext_bytes = plaintext.encode('utf-8')
        salt = os.urandom(self.salt_bytes)
        master_key = self._stretch_key(password.encode('utf-8'),
salt, self.iterations, self.master_key_length)

        ciphertext = b''
        # 2. Loop through plaintext in blocks
        for i in range(0, len(plaintext_bytes), self.block_size):
            plaintext_block = plaintext_bytes[i:i +
self.block_size]
            block_counter = i // self.block_size

            # 3. Generate unique keystream for this block
```

```
            seed = self._apply_counter(master_key, block_counter)
            initial_grid = self._convert_bytes_to_grid(seed)
            final_grid = self._run_ca_simulation(initial_grid)
            keystream = self._flatten_grid_to_bytes(final_grid)

            # 4. XOR the plaintext block with the keystream
            encrypted_block = bytes([p_byte ^ k_byte for p_byte,
k_byte in zip(plaintext_block, keystream)])
            ciphertext += encrypted_block

        # 5. Return the salt concatenated with the final
ciphertext
        return base64.b64encode(salt + ciphertext).decode('utf-8')

    def decrypt(self, encrypted_text: str, password: str) -> str:
        """Decrypts data encrypted with this system."""
        # 1. Setup
        byte_data = base64.b64decode(encrypted_text)
        salt = byte_data[:self.salt_bytes]
        ciphertext = byte_data[self.salt_bytes:]
        master_key = self._stretch_key(password.encode('utf-8'),
salt, self.iterations, self.master_key_length)

        decrypted_bytes = b''
        # 2. Loop through ciphertext in blocks (same process as
encryption)
        for i in range(0, len(ciphertext), self.block_size):
            ciphertext_block = ciphertext[i:i + self.block_size]
            block_counter = i // self.block_size

            # 3. Re-generate the exact same unique keystream for
this block
            seed_for_block = self._apply_counter(master_key,
block_counter)
            initial_grid =
self._convert_bytes_to_grid(seed_for_block)
            final_grid = self._run_ca_simulation(initial_grid)
            keystream_block =
self._flatten_grid_to_bytes(final_grid)

            # 4. XOR the ciphertext block with the keystream to
get the original plaintext
            decrypted_block = bytes([c_byte ^ k_byte for c_byte,
k_byte in zip(ciphertext_block, keystream_block)])
```

```
            decrypted_bytes += decrypted_block

        # 5. Decode the final bytes back to a string
        return decrypted_bytes.decode('utf-8')
```

d. GUI Components

cv_summary_window.py

```
from PyQt6.QtWidgets import (
    QMainWindow, QWidget, QLabel, QVBoxLayout, QHBoxLayout,
QGridLayout,
    QPushButton, QScrollArea, QFrame, QFileDialog
)
from PyQt6.QtCore import Qt
from PyQt6.QtGui import QFont
from .general_config import gui_config
import datetime
import re

class CVSummaryWindow(QMainWindow):
    def __init__(
        self,
        name: str,
        birthdate,
        address: str,
        phone: str,
        skills,  # Can be string or list
        jobs,    # Can be string or list of dicts
        education  # Can be string or list of dicts
    ):
        super().__init__()
        self.setWindowTitle(f"CV Summary - {name}")
        self.setMinimumSize(700, 550)

        if isinstance(birthdate, datetime.date):
            birthdate = birthdate.strftime("%B %d, %Y")
        elif birthdate is None:
            birthdate = "Not provided"

        # Process inputs to ensure consistent format
        self.processed_skills = self._process_skills(skills)
        self.processed_jobs = self._process_jobs(jobs)
        self.processed_education =
```

```
self._process_education(education)

        # Main scrollable area
        scroll = QScrollArea()
        scroll.setWidgetResizable(True)
        container = QWidget()
        scroll.setWidget(container)
        self.setCentralWidget(scroll)

        main_layout = QVBoxLayout(container)
        main_layout.setContentsMargins(25, 25, 25, 25)
        main_layout.setSpacing(20)

        # Create each section
        self.create_header_section(main_layout, name, birthdate,
address, phone)
        self.create_skills_section(main_layout,
self.processed_skills)
        self.create_experience_section(main_layout,
self.processed_jobs)
        self.create_education_section(main_layout,
self.processed_education)

        # Add final spacer
        main_layout.addStretch()

    def _process_skills(self, skills):
        """Process skills input to ensure list format"""
        if isinstance(skills, str):
            # Split semicolon or comma-separated skills
            if ';' in skills:
                return [s.strip() for s in skills.split(';') if
s.strip()]
            elif ',' in skills:
                return [s.strip() for s in skills.split(',') if
s.strip()]
            else:
                return [skills]
        elif isinstance(skills, list):
            return skills
        else:
            return ["No skills found"]

    def _process_jobs(self, jobs):
```

```
        """Process jobs input to ensure consistent format"""
        processed_jobs = []

        if isinstance(jobs, str) and jobs != "Not Found":
            # Split by commas if it's a string
            job_items = [j.strip() for j in jobs.split(',') if
j.strip()]

            for item in job_items:
                job = {
                    'title': item,
                    'period': 'Not specified',
                    'company': 'Not specified',
                    'location': 'Not specified',
                    'description': ''
                }
                processed_jobs.append(job)

        elif isinstance(jobs, list):
            # If it's already a list, ensure each item is a dict
            for job in jobs:
                if isinstance(job, dict):
                    processed_jobs.append(job)
                else:
                    processed_jobs.append({
                        'title': str(job),
                        'period': 'Not specified',
                        'company': 'Not specified',
                        'location': 'Not specified',
                        'description': ''
                    })

        return processed_jobs

    def _process_education(self, education):
        """Process education input to ensure consistent format"""
        processed_education = []

        if isinstance(education, str) and education != "Not
Found":
            # Split by commas if it's a string
            edu_items = [e.strip() for e in education.split(',')
if e.strip()]
```

```
            for item in edu_items:
                edu = {
                    'degree': 'Degree not specified',
                    'institution': item,
                    'period': '',
                    'details': ''
                }
                processed_education.append(edu)

        elif isinstance(education, list):
            # If it's already a list, ensure each item is a dict
            for edu in education:
                if isinstance(edu, dict):
                    processed_education.append(edu)
                else:
                    processed_education.append({
                        'degree': 'Degree not specified',
                        'institution': str(edu),
                        'period': '',
                        'details': ''
                    })

        return processed_education

    def create_header_section(self, parent_layout, name,
birthdate, address, phone):
        """Create the profile header section with contact info"""
        # Profile header with gradient background
        header_container = QFrame()
        header_container.setFixedHeight(80)  # Increased height
for better appearance
        header_container.setStyleSheet(f"""
            background: qlineargradient(x1:0, y1:0, x2:1, y2:0,
                        stop:0 {gui_config.colors.primary},
                        stop:1 {gui_config.colors.secondary});
            border-radius:
{gui_config.spacing.border_radius_medium}px;
        """)

        header_layout = QVBoxLayout(header_container)
        header_layout.setContentsMargins(20, 10, 20, 10)

        # Name label
        name_label = QLabel(name)
```

```
        name_label.setStyleSheet("""
            color: white;
            font-size: 24pt;
            font-weight: bold;
        """)
        name_label.setAlignment(Qt.AlignmentFlag.AlignCenter)
        header_layout.addWidget(name_label)

        parent_layout.addWidget(header_container)

        # Contact details section
        contact_box = QFrame()
        contact_box.setFrameShape(QFrame.Shape.StyledPanel)
        contact_box.setStyleSheet(f"""
            background-color: {gui_config.colors.bg_primary};
            border-radius:
{gui_config.spacing.border_radius_medium}px;
            border: 1px solid {gui_config.colors.border_light};
        """)

        contact_layout = QHBoxLayout(contact_box)
        contact_layout.setContentsMargins(20, 20, 20, 20)

        # Create a grid for contact details
        details_grid = QGridLayout()
        details_grid.setColumnStretch(1, 1)  # Make value column
stretch
        details_grid.setHorizontalSpacing(15)
        details_grid.setVerticalSpacing(10)

        # Detail fields
        detail_fields = [
            ("Date of Birth", birthdate or "Not provided"),
            ("Address", address or "Not provided"),
            ("Phone", phone or "Not provided")
        ]

        for row, (field, value) in enumerate(detail_fields):
            field_label = QLabel(f"{field}:")
            field_label.setStyleSheet(f"""
                color: {gui_config.colors.text_secondary};
                font-weight: bold;
                font-size: 11pt;
            """)
```

```
            value_label = QLabel(str(value))
            value_label.setStyleSheet(f"""
                color: {gui_config.colors.text_primary};
                font-size: 11pt;
            """)
            value_label.setWordWrap(True)

            details_grid.addWidget(field_label, row, 0)
            details_grid.addWidget(value_label, row, 1)

        contact_layout.addLayout(details_grid)
        parent_layout.addWidget(contact_box)

    def create_skills_section(self, parent_layout, skills):
        """Create the skills section with max 3 items per row"""
        # Section header
        header_label = self.create_section_header("Skills")
        parent_layout.addWidget(header_label)

        if not skills or skills == ["No skills found"]:
            empty_label = self.create_empty_section_label("No
skills found in resume")
            parent_layout.addWidget(empty_label)
            return

        # Skills container
        skills_box = QFrame()
        skills_box.setFrameShape(QFrame.Shape.StyledPanel)
        skills_box.setStyleSheet(f"""
            background-color: {gui_config.colors.bg_primary};
            border-radius:
{gui_config.spacing.border_radius_medium}px;
            border: 1px solid {gui_config.colors.border_light};
            padding: 10px;
        """)

        # Grid layout for skills with max 3 per row
        skills_layout = QGridLayout(skills_box)
        skills_layout.setContentsMargins(15, 15, 15, 15)
        skills_layout.setHorizontalSpacing(15)
        skills_layout.setVerticalSpacing(15)

        # Place skills in grid, 3 per row
```

```
        for i, skill in enumerate(skills):
            row = i // 3
            col = i % 3

            skill_chip = self.create_skill_chip(skill)
            skills_layout.addWidget(skill_chip, row, col)

        parent_layout.addWidget(skills_box)

    def create_skill_chip(self, skill_text):
        """Create a styled skill chip"""
        chip = QFrame()
        chip.setStyleSheet(f"""
            background-color: {gui_config.colors.bg_secondary};
            border: 1px solid {gui_config.colors.border_light};
            border-left: 4px solid {gui_config.colors.secondary};
/* Use secondary for accent */
            border-radius:
{gui_config.spacing.border_radius_small}px;
        """)

        layout = QHBoxLayout(chip)
        layout.setContentsMargins(12, 8, 12, 8)

        label = QLabel(skill_text)
        label.setStyleSheet(f"""
            color: {gui_config.colors.text_primary};
            font-size: 11pt;
        """)
        layout.addWidget(label)

        return chip

    def create_experience_section(self, parent_layout, jobs):
        """Create the job experience section"""
        # Section header
        header_label = self.create_section_header("Professional
Experience")
        parent_layout.addWidget(header_label)

        if not jobs:
            empty_label = self.create_empty_section_label("No work
experience found in resume")
            parent_layout.addWidget(empty_label)
```

```
            return

        # Experience container
        experience_container = QWidget()
        experience_layout = QVBoxLayout(experience_container)
        experience_layout.setContentsMargins(0, 0, 0, 0)
        experience_layout.setSpacing(10)

        for job in jobs:
            job_card = self.create_job_card(job)
            experience_layout.addWidget(job_card)

        parent_layout.addWidget(experience_container)

    def create_job_card(self, job):
        """Create a card for a job entry"""
        job_box = QFrame()
        job_box.setFrameShape(QFrame.Shape.StyledPanel)
        job_box.setStyleSheet(f"""
            background-color: {gui_config.colors.bg_primary};
            border-radius:
{gui_config.spacing.border_radius_medium}px;
            border-left: 5px solid {gui_config.colors.primary};
/* Use primary as accent */
            border-top: 1px solid
{gui_config.colors.border_light};
            border-right: 1px solid
{gui_config.colors.border_light};
            border-bottom: 1px solid
{gui_config.colors.border_light};
        """)

        job_layout = QVBoxLayout(job_box)
        job_layout.setContentsMargins(15, 15, 15, 15)

        # Title row
        title_label = QLabel(job['title'])
        title_label.setStyleSheet(f"""
            font-size: 14pt;
            font-weight: bold;
            color: {gui_config.colors.primary};  /* Use primary
for main text */
        """)
        title_label.setWordWrap(True)
```

```
        job_layout.addWidget(title_label)

        # Company and period row (if available)
        if job.get('company') and job['company'] != 'Not
specified':
            company_period_layout = QHBoxLayout()

            company_label = QLabel(job['company'])
            company_label.setStyleSheet(f"""
                font-size: 12pt;
                color: {gui_config.colors.text_secondary};
                font-weight: bold;
            """)
            company_period_layout.addWidget(company_label)

            company_period_layout.addStretch()

            if job.get('period') and job['period'] != 'Not
specified':
                period_label = QLabel(job['period'])
                period_label.setStyleSheet(f"""
                    font-size: 11pt;
                    font-style: italic;
                    color: {gui_config.colors.text_secondary};
                """)
                company_period_layout.addWidget(period_label)

            job_layout.addLayout(company_period_layout)

        # Description (if available)
        if job.get('description') and job['description']:
            separator = QFrame()
            separator.setFrameShape(QFrame.Shape.HLine)
            separator.setFrameShadow(QFrame.Shadow.Sunken)
            separator.setStyleSheet(f"background-color:
{gui_config.colors.border_light};")
            job_layout.addWidget(separator)

            desc_label = QLabel(job['description'])
            desc_label.setWordWrap(True)
            desc_label.setStyleSheet(f"""
                font-size: 11pt;
                color: {gui_config.colors.text_primary};
                margin-top: 5px;
```

```
            """)
            job_layout.addWidget(desc_label)

        return job_box

    def create_education_section(self, parent_layout, education):
        """Create the education section"""
        # Section header
        header_label = self.create_section_header("Education")
        parent_layout.addWidget(header_label)

        if not education:
            empty_label = self.create_empty_section_label("No
education information found in resume")
            parent_layout.addWidget(empty_label)
            return

        # Education container
        education_container = QWidget()
        education_layout = QVBoxLayout(education_container)
        education_layout.setContentsMargins(0, 0, 0, 0)
        education_layout.setSpacing(10)

        for edu in education:
            edu_card = self.create_education_card(edu)
            education_layout.addWidget(edu_card)

        parent_layout.addWidget(education_container)

    def create_education_card(self, edu):
        """Create a card for an education entry"""
        edu_box = QFrame()
        edu_box.setFrameShape(QFrame.Shape.StyledPanel)
        edu_box.setStyleSheet(f"""
            background-color: {gui_config.colors.bg_primary};
            border-radius:
{gui_config.spacing.border_radius_medium}px;
            border-left: 5px solid {gui_config.colors.secondary};
/* Use secondary as accent */
            border-top: 1px solid
{gui_config.colors.border_light};
            border-right: 1px solid
{gui_config.colors.border_light};
            border-bottom: 1px solid
```

```
{gui_config.colors.border_light};
        """)

        edu_layout = QVBoxLayout(edu_box)
        edu_layout.setContentsMargins(15, 15, 15, 15)

        # Institution
        institution_label = QLabel(edu['institution'])
        institution_label.setStyleSheet(f"""
            font-size: 14pt;
            font-weight: bold;
            color: {gui_config.colors.primary};  /* Use primary
for main text */
        """)
        institution_label.setWordWrap(True)
        edu_layout.addWidget(institution_label)

        # Degree and period (if available)
        if edu.get('degree') and edu['degree'] != 'Degree not
specified':
            degree_period_layout = QHBoxLayout()

            degree_label = QLabel(edu['degree'])
            degree_label.setStyleSheet(f"""
                font-size: 12pt;
                color: {gui_config.colors.text_secondary};
                font-weight: bold;
            """)
            degree_period_layout.addWidget(degree_label)

            degree_period_layout.addStretch()

            if edu.get('period') and edu['period']:
                period_label = QLabel(edu['period'])
                period_label.setStyleSheet(f"""
                    font-size: 11pt;
                    font-style: italic;
                    color: {gui_config.colors.text_secondary};
                """)
                degree_period_layout.addWidget(period_label)

            edu_layout.addLayout(degree_period_layout)

        # Details (if available)
```

```
        if edu.get('details') and edu['details']:
            separator = QFrame()
            separator.setFrameShape(QFrame.Shape.HLine)
            separator.setFrameShadow(QFrame.Shadow.Sunken)
            separator.setStyleSheet(f"background-color:
{gui_config.colors.border_light};")
            edu_layout.addWidget(separator)

            details_label = QLabel(edu['details'])
            details_label.setWordWrap(True)
            details_label.setStyleSheet(f"""
                font-size: 11pt;
                color: {gui_config.colors.text_primary};
                margin-top: 5px;
            """)
            edu_layout.addWidget(details_label)

        return edu_box

    def create_section_header(self, title):
        """Create a styled section header"""
        header = QLabel(title)
        header.setStyleSheet(f"""
            font-size: 16pt;
            font-weight: bold;
            color: {gui_config.colors.primary};
            padding-bottom: 5px;
            border-bottom: 2px solid
{gui_config.colors.secondary};
        """)
        return header

    def create_empty_section_label(self, text):
        """Create an empty state label for a section"""
        frame = QFrame()
        frame.setStyleSheet(f"""
            background-color: {gui_config.colors.bg_primary};
            border-radius:
{gui_config.spacing.border_radius_medium}px;
            border: 1px solid {gui_config.colors.border_light};
        """)

        layout = QVBoxLayout(frame)
        layout.setContentsMargins(20, 15, 20, 15)
```

```
        label = QLabel(text)
        label.setAlignment(Qt.AlignmentFlag.AlignCenter)
        label.setStyleSheet(f"""
            color: {gui_config.colors.text_muted};
            font-style: italic;
            font-size: 12pt;
            padding: 10px;
        """)
        layout.addWidget(label)

        return frame
```

general_config.py

```
from typing import Dict, Any, Optional
from dataclasses import dataclass, field
from PyQt6.QtGui import QFont # type: ignore


@dataclass
class ColorTheme:
    """Color theme configuration."""
    primary: str = "#2c3e50"
    secondary: str = "#3498db"
    accent: str = "#e74c3c"
    success: str = "#27ae60"
    warning: str = "#f39c12"
    danger: str = "#e74c3c"

    # Text colors
    text_primary: str = "#2c3e50"
    text_secondary: str = "#7f8c8d"
    text_muted: str = "#95a5a6"
    text_light: str = "#ecf0f1"

    # Background colors
    bg_primary: str = "#ffffff"
    bg_secondary: str = "#f8f9fa"
    bg_dark: str = "#34495e"

    # Border colors
    border_light: str = "#ecf0f1"
    border_medium: str = "#bdc3c7"
```

```
    border_dark: str = "#95a5a6"


@dataclass
class FontConfig:
    """Font configuration."""
    family_primary: str = "Segoe UI"
    family_monospace: str = "Consolas, Monaco, monospace"

    # Font sizes
    size_small: int = 9
    size_normal: int = 10
    size_medium: int = 11
    size_large: int = 14
    size_xlarge: int = 18
    size_title: int = 24

    # Font weights
    weight_normal: int = QFont.Weight.Normal.value
    weight_bold: int = QFont.Weight.Bold.value


@dataclass
class SpacingConfig:
    """Spacing and sizing configuration."""
    margin_small: int = 5
    margin_medium: int = 10
    margin_large: int = 20

    padding_small: int = 5
    padding_medium: int = 10
    padding_large: int = 15

    border_radius_small: int = 3
    border_radius_medium: int = 5
    border_radius_large: int = 8

    border_width_thin: int = 1
    border_width_medium: int = 2
    border_width_thick: int = 3


@dataclass
class HeaderConfig:
```

```
    """Header-specific configuration."""
    show_icon: bool = True
    icon_size: int = 32
    title_alignment: str = "center"  # left, center, right
    subtitle_alignment: str = "center"
    background_gradient: bool = False
    shadow_enabled: bool = True

    # Custom styling
    custom_title_style: Optional[str] = None
    custom_subtitle_style: Optional[str] = None
    custom_background_style: Optional[str] = None

@dataclass
class InputConfig:
    """Input section specific configuration."""
    show_file_controls: bool = True
    show_character_count: bool = True
    show_file_info: bool = True
    min_height: int = 200
    max_height: int = 400
    placeholder_text: str = "Enter your text here..."

    # File dialog settings
    supported_extensions: str = "Text Files (*.txt *.md *.py *.js
*.html *.css *.json *.xml);;All Files (*)"
    default_directory: str = ""

    # Button configurations
    load_button_text: str = "📁 Load File"
    clear_button_text: str = "🗑️ Clear"
    show_load_button: bool = True
    show_clear_button: bool = True

    # Text area settings
    word_wrap: bool = True
    font_family: str = "Consolas, Monaco, monospace"
    font_size: int = 10
    line_height: float = 1.4

    # Custom styling
    custom_frame_style: Optional[str] = None
    custom_textarea_style: Optional[str] = None
    custom_button_style: Optional[str] = None
```

```python
@dataclass
class SearchConfig:
    """Search section specific configuration."""
    # Keywords section
    show_keywords_section: bool = True
    keywords_placeholder: str = "Enter keywords separated by
commas (e.g., python, algorithm, search)"
    max_keywords_display_height: int = 80

    # Algorithm section
    show_algorithm_section: bool = True
    default_algorithm: str = "KMP (Knuth-Morris-Pratt)"
    available_algorithms: tuple = (
        "KMP (Knuth-Morris-Pratt)",
        "Boyer-Moore Simple",
        "Boyer-Moore Complex",
        "Aho-Corasick",
        "Fuzzy Search"
    )

    # Search parameters
    default_top_matches: int = 10
    max_top_matches: int = 1000
    default_case_sensitive: bool = False

    # Button configurations
    search_button_text: str = "🔍 Start Search"
    searching_button_text: str = "🔄 Searching..."
    add_keyword_text: str = "+ Add"
    clear_keywords_text: str = "Clear All"

    # Display settings
    show_algorithm_info: bool = True
    show_quick_add_buttons: bool = True
    show_search_parameters: bool = True

    # Custom styling
    custom_frame_style: Optional[str] = None
    custom_button_style: Optional[str] = None
    custom_input_style: Optional[str] = None

@dataclass
class ResultConfig:
```

```
    """Result section specific configuration."""
    # Display settings
    show_statistics: bool = True
    show_export_button: bool = True
    show_clear_button: bool = True
    show_progress_bar: bool = True

    # Card settings
    max_snippet_length: int = 200
    snippet_height: int = 80
    card_hover_effect: bool = True
    show_pattern_highlighting: bool = True
    show_similarity_score: bool = True

    # Export settings
    default_export_format: str = "txt"  # txt, csv, json
    supported_export_formats: str = "Text Files (*.txt);;CSV Files
(*.csv);;JSON Files (*.json)"

    # Button configurations
    export_button_text: str = "💾 Export Results"
    clear_button_text: str = "🗑️ Clear Results"

    # Display texts
    title_text: str = "📊 Search Results"
    empty_state_message: str = "🔍 No search results yet\n\nEnter
your text and keywords above, then click 'Start Search' to find
patterns."
    no_results_message: str = "❌ No matches found\n\nTry
different keywords or switch to fuzzy search for approximate
matches."

    # Animation and interaction
    enable_animations: bool = True
    enable_card_selection: bool = True
    auto_scroll_to_results: bool = True

    # Custom styling
    custom_header_style: Optional[str] = None
    custom_card_style: Optional[str] = None
    custom_empty_state_style: Optional[str] = None

class GUIConfig:
    """Central GUI configuration manager."""
```

```
    def __init__(self):
        self.colors = ColorTheme()
        self.fonts = FontConfig()
        self.spacing = SpacingConfig()
        self.header = HeaderConfig()
        self.search = SearchConfig()
        self.result = ResultConfig()
        self._custom_themes: Dict[str, ColorTheme] = {}

    def set_theme(self, theme_name: str) -> None:
        """Set a predefined theme."""
        themes = {
            "light": ColorTheme(),
            "dark": ColorTheme(
                primary="#ecf0f1",
                text_primary="#ecf0f1",
                text_secondary="#bdc3c7",
                bg_primary="#2c3e50",
                bg_secondary="#34495e",
                border_light="#34495e",
                border_medium="#7f8c8d"
            ),
            "blue": ColorTheme(
                primary="#3498db",
                secondary="#2980b9",
                bg_secondary="#ebf3fd"
            )
        }

        if theme_name in themes:
            self.colors = themes[theme_name]
        elif theme_name in self._custom_themes:
            self.colors = self._custom_themes[theme_name]

    def register_custom_theme(self, name: str, theme: ColorTheme)
-> None:
        """Register a custom color theme."""
        self._custom_themes[name] = theme

    def get_style_dict(self) -> Dict[str, Any]:
        """Get configuration as a dictionary for easy template
substitution."""
        return {
```

```
            'colors': self.colors.__dict__,
            'fonts': self.fonts.__dict__,
            'spacing': self.spacing.__dict__,
            'header': self.header.__dict__
        }


# Global configuration instance
gui_config = GUIConfig()
```

header.py

```
from PyQt6.QtWidgets import QWidget, QVBoxLayout, QHBoxLayout,
QLabel, QFrame # type: ignore
from PyQt6.QtCore import Qt, pyqtSignal # type: ignore
from PyQt6.QtGui import QFont, QPixmap, QIcon # type: ignore
from typing import Optional, Union, Literal
from .general_config import gui_config, HeaderConfig


class ConfigurableHeaderComponent(QWidget):
    """Highly configurable header component for the
application."""

    # Signals
    title_clicked = pyqtSignal()
    subtitle_clicked = pyqtSignal()

    def __init__(self,
                 title: str = "CV PROCESSOR",
                 subtitle: str = "Advanced pattern matching with
KMP, Boyer-Moore, Aho-Corasick, and Fuzzy Search",
                 icon_path: Optional[str] = None,
                 config: Optional[HeaderConfig] = None,
                 parent: Optional[QWidget] = None) -> None:
        super().__init__(parent)

        # Configuration
        self.config = config or gui_config.header
        self.gui_config = gui_config

        # Content
```

```
        self._title = title
        self._subtitle = subtitle
        self._icon_path = icon_path

        # UI Elements
        self.container_frame: Optional[QFrame] = None
        self.icon_label: Optional[QLabel] = None
        self.title_label: Optional[QLabel] = None
        self.subtitle_label: Optional[QLabel] = None

        self.setup_ui()

    def setup_ui(self) -> None:
        """Set up the header UI components."""
        main_layout = QVBoxLayout(self)
        main_layout.setContentsMargins(0, 0, 0, 0)
        main_layout.setSpacing(0)

        # Container frame for styling
        self.container_frame = QFrame()
        self.container_frame.setObjectName("headerContainer")

        # Main content layout
        content_layout = QVBoxLayout(self.container_frame)
        content_layout.setContentsMargins(
            self.gui_config.spacing.margin_large,
            self.gui_config.spacing.margin_large,
            self.gui_config.spacing.margin_large,
            self.gui_config.spacing.margin_medium
        )

content_layout.setSpacing(self.gui_config.spacing.margin_small)

        # Header content layout (for icon + text)
        header_content_layout = QHBoxLayout()

        # Icon (if enabled and provided)
        if self.config.show_icon and self._icon_path:
            self.setup_icon(header_content_layout)

        # Text content layout
        text_layout = QVBoxLayout()

text_layout.setSpacing(self.gui_config.spacing.margin_small)
```

```
        # Title
        self.setup_title(text_layout)

        # Subtitle
        if self._subtitle:
            self.setup_subtitle(text_layout)

        # Add text layout to header content
        header_content_layout.addLayout(text_layout)

        # Add header content to main content
        content_layout.addLayout(header_content_layout)

        # Add container to main layout
        main_layout.addWidget(self.container_frame)

        # Apply styling
        self.apply_styling()

    def setup_icon(self, layout: QHBoxLayout) -> None:
        """Set up the header icon."""
        self.icon_label = QLabel()
        self.icon_label.setObjectName("headerIcon")

        # Load and set icon
        pixmap = QPixmap(self._icon_path)
        if not pixmap.isNull():
            scaled_pixmap = pixmap.scaled(
                self.config.icon_size,
                self.config.icon_size,
                Qt.AspectRatioMode.KeepAspectRatio,
                Qt.TransformationMode.SmoothTransformation
            )
            self.icon_label.setPixmap(scaled_pixmap)

        self.icon_label.setAlignment(Qt.AlignmentFlag.AlignCenter)
        layout.addWidget(self.icon_label)

    def setup_title(self, layout: QVBoxLayout) -> None:
        """Set up the title label."""
        self.title_label = QLabel(self._title)
        self.title_label.setObjectName("headerTitle")
```

```
        # Alignment
        alignment_map = {
            "left": Qt.AlignmentFlag.AlignLeft,
            "center": Qt.AlignmentFlag.AlignCenter,
            "right": Qt.AlignmentFlag.AlignRight
        }

self.title_label.setAlignment(alignment_map.get(self.config.title_
alignment, Qt.AlignmentFlag.AlignCenter))

        # Font
        title_font = QFont(self.gui_config.fonts.family_primary)
        title_font.setPointSize(self.gui_config.fonts.size_title)

title_font.setWeight(QFont.Weight(self.gui_config.fonts.weight_bol
d))
        self.title_label.setFont(title_font)

        # Make clickable if needed
        self.title_label.mousePressEvent = lambda e:
self.title_clicked.emit()

        layout.addWidget(self.title_label)

    def setup_subtitle(self, layout: QVBoxLayout) -> None:
        """Set up the subtitle label."""
        self.subtitle_label = QLabel(self._subtitle)
        self.subtitle_label.setObjectName("headerSubtitle")

        # Alignment
        alignment_map = {
            "left": Qt.AlignmentFlag.AlignLeft,
            "center": Qt.AlignmentFlag.AlignCenter,
            "right": Qt.AlignmentFlag.AlignRight
        }

self.subtitle_label.setAlignment(alignment_map.get(self.config.sub
title_alignment, Qt.AlignmentFlag.AlignCenter))

        # Font
        subtitle_font =
QFont(self.gui_config.fonts.family_primary)

subtitle_font.setPointSize(self.gui_config.fonts.size_medium)
```

```
subtitle_font.setWeight(QFont.Weight(self.gui_config.fonts.weight_
normal))
        self.subtitle_label.setFont(subtitle_font)

        # Word wrap for long subtitles
        self.subtitle_label.setWordWrap(True)

        # Make clickable if needed
        self.subtitle_label.mousePressEvent = lambda e:
self.subtitle_clicked.emit()

        layout.addWidget(self.subtitle_label)

    def apply_styling(self) -> None:
        """Apply CSS styling to the header components."""
        # Base container style
        container_style = f"""
        QFrame#headerContainer {{
            background-color: {self.gui_config.colors.bg_primary};
            border: none;
            {self._get_background_style()}
        }}
        """

        # Title style
        title_style = f"""
        QLabel#headerTitle {{
            color: {self.gui_config.colors.text_primary};
            margin: {self.gui_config.spacing.margin_medium}px 0;
            {self._get_title_effects()}
        }}
        QLabel#headerTitle:hover {{
            color: {self.gui_config.colors.secondary};
        }}
        """

        # Subtitle style
        subtitle_style = f"""
        QLabel#headerSubtitle {{
            color: {self.gui_config.colors.text_secondary};
            margin-bottom:
{self.gui_config.spacing.margin_large}px;
            line-height: 1.4;
```

```
        }}
        QLabel#headerSubtitle:hover {{
            color: {self.gui_config.colors.text_primary};
        }}
        """

        # Combine and apply styles
        full_style = container_style + title_style +
subtitle_style

        # Apply custom styles if provided
        if self.config.custom_title_style and self.title_label:
            full_style += f"QLabel#headerTitle {{
{self.config.custom_title_style} }}"

        if self.config.custom_subtitle_style and
self.subtitle_label:
            full_style += f"QLabel#headerSubtitle {{
{self.config.custom_subtitle_style} }}"

        if self.config.custom_background_style:
            full_style += f"QFrame#headerContainer {{
{self.config.custom_background_style} }}"

        self.setStyleSheet(full_style)

    def _get_background_style(self) -> str:
        """Get background styling based on configuration."""
        if self.config.background_gradient:
            return f"""
            background: qlineargradient(x1:0, y1:0, x2:0, y2:1,
                stop:0 {self.gui_config.colors.bg_primary},
                stop:1 {self.gui_config.colors.bg_secondary});
            """
        return ""

    def _get_title_effects(self) -> str:
        """Get title effects based on configuration."""
        effects = ""
        if self.config.shadow_enabled:
            effects += "text-shadow: 1px 1px 2px rgba(0,0,0,0.1);"
        return effects

    # Public API methods
```

```
    def set_title(self, title: str) -> None:
        """Set the header title."""
        self._title = title
        if self.title_label:
            self.title_label.setText(title)

    def set_subtitle(self, subtitle: str) -> None:
        """Set the header subtitle."""
        self._subtitle = subtitle
        if self.subtitle_label:
            self.subtitle_label.setText(subtitle)
        elif subtitle:  # Create subtitle if it doesn't exist
            self.setup_ui()  # Rebuild UI

    def set_icon(self, icon_path: str) -> None:
        """Set the header icon."""
        self._icon_path = icon_path
        if self.icon_label:
            pixmap = QPixmap(icon_path)
            if not pixmap.isNull():
                scaled_pixmap = pixmap.scaled(
                    self.config.icon_size,
                    self.config.icon_size,
                    Qt.AspectRatioMode.KeepAspectRatio,
                    Qt.TransformationMode.SmoothTransformation
                )
                self.icon_label.setPixmap(scaled_pixmap)

    def update_config(self, config: HeaderConfig) -> None:
        """Update header configuration and refresh UI."""
        self.config = config
        self.setup_ui()

    def set_theme(self, theme_name: str) -> None:
        """Apply a different theme to the header."""
        self.gui_config.set_theme(theme_name)
        self.apply_styling()


# Backward compatibility - keep your original simple header
class HeaderComponent(ConfigurableHeaderComponent):
    """Simple header component for backward compatibility."""

    def __init__(self, parent: Optional[QWidget] = None) -> None:
```

```
        super().__init__(parent=parent)
```

input.py

```
from PyQt6.QtWidgets import (QWidget, QVBoxLayout, QHBoxLayout,
QLabel,  # type: ignore
                             QTextEdit, QPushButton, QFileDialog,
QMessageBox,
                             QFrame, QProgressBar)
from PyQt6.QtCore import Qt, pyqtSignal # type: ignore
from PyQt6.QtGui import QFont, QIcon # type: ignore
from typing import Optional, List
from .general_config import gui_config, InputConfig
import os


class ConfigurableInputSection(QWidget):
    """Highly configurable input section for text input and file
loading."""

    # Signals
    text_changed = pyqtSignal(str)  # Emitted when text content
changes
    file_loaded = pyqtSignal(str, str)  # Emitted when file is
loaded (filename, content)
    file_load_started = pyqtSignal()  # Emitted when file loading
starts
    file_load_finished = pyqtSignal()  # Emitted when file loading
finishes
    clear_requested = pyqtSignal()  # Emitted when clear is
requested

    def __init__(self,
                 title: str = "📄 Input Text",
                 config: Optional[InputConfig] = None,
                 parent: Optional[QWidget] = None) -> None:
        super().__init__(parent)

        # Configuration
        self.config = config or gui_config.input
        self.gui_config = gui_config

        # Content
```

```
        self._title = title
        self.current_file_path: Optional[str] = None

        # UI Elements
        self.frame: Optional[QFrame] = None
        self.section_label: Optional[QLabel] = None
        self.load_file_btn: Optional[QPushButton] = None
        self.clear_btn: Optional[QPushButton] = None
        self.file_info_label: Optional[QLabel] = None
        self.text_input: Optional[QTextEdit] = None
        self.char_count_label: Optional[QLabel] = None
        self.progress_bar: Optional[QProgressBar] = None

        self.setup_ui()
        self.connect_signals()
        self.apply_styling()

    def setup_ui(self) -> None:
        """Set up the input section UI components."""
        layout = QVBoxLayout(self)
        layout.setContentsMargins(
            self.gui_config.spacing.margin_large,
            self.gui_config.spacing.margin_medium,
            self.gui_config.spacing.margin_large,
            self.gui_config.spacing.margin_medium
        )
        layout.setSpacing(self.gui_config.spacing.margin_medium)

        # Section frame
        self.frame = QFrame()
        self.frame.setObjectName("inputFrame")
        self.frame.setFrameStyle(QFrame.Shape.Box)

        frame_layout = QVBoxLayout(self.frame)

frame_layout.setSpacing(self.gui_config.spacing.margin_medium)
        frame_layout.setContentsMargins(
            self.gui_config.spacing.margin_medium,
            self.gui_config.spacing.margin_medium,
            self.gui_config.spacing.margin_medium,
            self.gui_config.spacing.margin_medium
        )

        # Section header
```

```
        self.setup_header(frame_layout)

        # File info label
        if self.config.show_file_info:
            self.setup_file_info(frame_layout)

        # Text input area
        self.setup_text_input(frame_layout)

        # Character count and progress
        self.setup_footer(frame_layout)

        layout.addWidget(self.frame)

    def setup_header(self, layout: QVBoxLayout) -> None:
        """Set up the header section with title and controls."""
        header_layout = QHBoxLayout()

        # Section title
        self.section_label = QLabel(self._title)
        self.section_label.setObjectName("inputTitle")

        title_font = QFont(self.gui_config.fonts.family_primary)
        title_font.setPointSize(self.gui_config.fonts.size_large)

title_font.setWeight(QFont.Weight(self.gui_config.fonts.weight_bol
d))
        self.section_label.setFont(title_font)

        header_layout.addWidget(self.section_label)
        header_layout.addStretch()

        # File controls
        if self.config.show_file_controls:
            self.setup_file_controls(header_layout)

        layout.addLayout(header_layout)

    def setup_file_controls(self, layout: QHBoxLayout) -> None:
        """Set up file control buttons."""
        file_controls_layout = QHBoxLayout()

        if self.config.show_load_button:
            self.load_file_btn =
```

```
QPushButton(self.config.load_button_text)
            self.load_file_btn.setObjectName("loadButton")
            file_controls_layout.addWidget(self.load_file_btn)

        if self.config.show_clear_button:
            self.clear_btn =
QPushButton(self.config.clear_button_text)
            self.clear_btn.setObjectName("clearButton")
            file_controls_layout.addWidget(self.clear_btn)

        layout.addLayout(file_controls_layout)

    def setup_file_info(self, layout: QVBoxLayout) -> None:
        """Set up file information label."""
        self.file_info_label = QLabel("Type or paste your text
below, or load from a file")
        self.file_info_label.setObjectName("fileInfo")
        layout.addWidget(self.file_info_label)

    def setup_text_input(self, layout: QVBoxLayout) -> None:
        """Set up the text input area."""
        self.text_input = QTextEdit()
        self.text_input.setObjectName("textInput")

        # Configure text area

self.text_input.setPlaceholderText(self._build_placeholder_text())
        self.text_input.setMinimumHeight(self.config.min_height)
        self.text_input.setMaximumHeight(self.config.max_height)

        # Font configuration
        text_font = QFont(self.config.font_family)
        text_font.setPointSize(self.config.font_size)
        self.text_input.setFont(text_font)

        # Word wrap
        if self.config.word_wrap:

self.text_input.setLineWrapMode(QTextEdit.LineWrapMode.WidgetWidth
)
        else:

self.text_input.setLineWrapMode(QTextEdit.LineWrapMode.NoWrap)
```

```
        layout.addWidget(self.text_input)

    def setup_footer(self, layout: QVBoxLayout) -> None:
        """Set up footer with character count and progress bar."""
        footer_layout = QHBoxLayout()

        # Progress bar (initially hidden)
        self.progress_bar = QProgressBar()
        self.progress_bar.setObjectName("progressBar")
        self.progress_bar.setVisible(False)
        self.progress_bar.setMaximumHeight(5)

        # Character count
        if self.config.show_character_count:
            self.char_count_label = QLabel("Characters: 0")
            self.char_count_label.setObjectName("charCount")

            count_font =
QFont(self.gui_config.fonts.family_primary)

count_font.setPointSize(self.gui_config.fonts.size_small)
            self.char_count_label.setFont(count_font)

self.char_count_label.setAlignment(Qt.AlignmentFlag.AlignRight)

            footer_layout.addStretch()
            footer_layout.addWidget(self.char_count_label)

        layout.addWidget(self.progress_bar)
        layout.addLayout(footer_layout)

    def _build_placeholder_text(self) -> str:
        """Build placeholder text based on configuration."""
        base_text = self.config.placeholder_text + "\n\n"

        if self.config.show_file_controls:
            base_text += "You can type directly or load a text
file using the 'Load File' button above.\n"

        # Add supported extensions info
        extensions =
self.config.supported_extensions.split(';;')[0]
        extensions = extensions.replace('Text Files (',
'').replace(')', '')
```

```
        base_text += f"Supported formats: {extensions}"

        return base_text

    def connect_signals(self) -> None:
        """Connect internal signals."""
        if self.load_file_btn:
            self.load_file_btn.clicked.connect(self.load_file)
        if self.clear_btn:
            self.clear_btn.clicked.connect(self.clear_text)
        if self.text_input:

self.text_input.textChanged.connect(self.on_text_changed)

    def apply_styling(self) -> None:
        """Apply CSS styling to all components."""
        # Frame style
        frame_style = f"""
        QFrame#inputFrame {{
            border:
{self.gui_config.spacing.border_width_medium}px solid
{self.gui_config.colors.border_medium};
            border-radius:
{self.gui_config.spacing.border_radius_large}px;
            background-color: {self.gui_config.colors.bg_primary};
            padding: {self.gui_config.spacing.padding_medium}px;
        }}
        """

        # Title style
        title_style = f"""
        QLabel#inputTitle {{
            color: {self.gui_config.colors.text_primary};
            margin-bottom:
{self.gui_config.spacing.margin_small}px;
        }}
        """

        # Button styles
        load_button_style = f"""
        QPushButton#loadButton {{
            background-color: {self.gui_config.colors.secondary};
            color: {self.gui_config.colors.text_light};
            border: none;
```

```
            border-radius:
{self.gui_config.spacing.border_radius_medium}px;
            padding: {self.gui_config.spacing.padding_small}px
{self.gui_config.spacing.padding_medium}px;
            font-weight: bold;
        }}
        QPushButton#loadButton:hover {{
            background-color: {self.gui_config.colors.primary};
        }}
        QPushButton#loadButton:pressed {{
            background-color:
{self.gui_config.colors.text_primary};
        }}
        """

        clear_button_style = f"""
        QPushButton#clearButton {{
            background-color: {self.gui_config.colors.danger};
            color: {self.gui_config.colors.text_light};
            border: none;
            border-radius:
{self.gui_config.spacing.border_radius_medium}px;
            padding: {self.gui_config.spacing.padding_small}px
{self.gui_config.spacing.padding_medium}px;
            font-weight: bold;
        }}
        QPushButton#clearButton:hover {{
            background-color: {self.gui_config.colors.accent};
        }}
        """

        # Text input style
        text_input_style = f"""
        QTextEdit#textInput {{
            border: {self.gui_config.spacing.border_width_thin}px
solid {self.gui_config.colors.border_medium};
            border-radius:
{self.gui_config.spacing.border_radius_medium}px;
            padding: {self.gui_config.spacing.padding_medium}px;
            background-color: {self.gui_config.colors.bg_primary};
            color: {self.gui_config.colors.text_primary};
            line-height: {self.config.line_height};
        }}
        QTextEdit#textInput:focus {{
```

```
            border:
{self.gui_config.spacing.border_width_medium}px solid
{self.gui_config.colors.secondary};
        }}
        """

        # File info style
        file_info_style = f"""
        QLabel#fileInfo {{
            color: {self.gui_config.colors.text_secondary};
            font-style: italic;
        }}
        """

        # Character count style
        char_count_style = f"""
        QLabel#charCount {{
            color: {self.gui_config.colors.text_secondary};
        }}
        """

        # Progress bar style
        progress_style = f"""
        QProgressBar#progressBar {{
            border: none;
            border-radius:
{self.gui_config.spacing.border_radius_small}px;
            background-color:
{self.gui_config.colors.border_light};
        }}
        QProgressBar#progressBar::chunk {{
            background-color: {self.gui_config.colors.secondary};
            border-radius:
{self.gui_config.spacing.border_radius_small}px;
        }}
        """

        # Combine all styles
        full_style = (frame_style + title_style +
load_button_style +
                      clear_button_style + text_input_style +
file_info_style +
                      char_count_style + progress_style)
```

```
        # Apply custom styles if provided
        if self.config.custom_frame_style:
            full_style += f"QFrame#inputFrame {{
{self.config.custom_frame_style} }}"

        if self.config.custom_textarea_style:
            full_style += f"QTextEdit#textInput {{
{self.config.custom_textarea_style} }}"

        if self.config.custom_button_style:
            full_style += f"QPushButton {{
{self.config.custom_button_style} }}"

        self.setStyleSheet(full_style)

    def load_file(self) -> None:
        """Load text from a file with progress indication."""
        self.file_load_started.emit()

        if self.progress_bar:
            self.progress_bar.setVisible(True)
            self.progress_bar.setRange(0, 0)  # Indeterminate
progress

        file_path, _ = QFileDialog.getOpenFileName(
            self,
            "Select Text File",
            self.config.default_directory,
            self.config.supported_extensions
        )

        if file_path:
            try:
                with open(file_path, 'r', encoding='utf-8') as
file:
                    content = file.read()

                self.text_input.setPlainText(content)
                self.current_file_path = file_path
                filename = os.path.basename(file_path)

                if self.file_info_label:
                    self.file_info_label.setText(f"📁 Loaded:
{filename} ({len(content):,} characters)")
```

```
                    self.file_info_label.setStyleSheet(f"color:
{self.gui_config.colors.success}; font-weight: bold;")

                self.file_loaded.emit(filename, content)

            except Exception as e:
                QMessageBox.critical(
                    self,
                    "File Load Error",
                    f"Could not load file: {str(e)}"
                )

        if self.progress_bar:
            self.progress_bar.setVisible(False)

        self.file_load_finished.emit()

    def clear_text(self) -> None:
        """Clear the text input."""
        self.clear_requested.emit()

        if self.text_input:
            self.text_input.clear()

        self.current_file_path = None

        if self.file_info_label:
            self.file_info_label.setText("Type or paste your text
below, or load from a file")
            self.file_info_label.setStyleSheet(f"color:
{self.gui_config.colors.text_secondary}; font-style: italic;")

    def on_text_changed(self) -> None:
        """Handle text change events."""
        if not self.text_input:
            return

        text = self.text_input.toPlainText()
        char_count = len(text)

        # Update character count
        if self.char_count_label:
            self.char_count_label.setText(f"Characters:
{char_count:,}")
```

```
        # Update file info if text was modified
        if self.file_info_label:
            if self.current_file_path and text:
                filename =
os.path.basename(self.current_file_path)
                self.file_info_label.setText(f"📁 {filename}
(modified) - {char_count:,} characters")
                self.file_info_label.setStyleSheet(f"color:
{self.gui_config.colors.warning}; font-weight: bold;")
            elif not text:
                self.file_info_label.setText("Type or paste your
text below, or load from a file")
                self.file_info_label.setStyleSheet(f"color:
{self.gui_config.colors.text_secondary}; font-style: italic;")

        # Emit signal
        self.text_changed.emit(text)

    # Public API methods
    def set_title(self, title: str) -> None:
        """Set the section title."""
        self._title = title
        if self.section_label:
            self.section_label.setText(title)

    def get_text(self) -> str:
        """Get the current text content."""
        return self.text_input.toPlainText() if self.text_input
else ""

    def set_text(self, text: str) -> None:
        """Set the text content."""
        if self.text_input:
            self.text_input.setPlainText(text)

    def get_file_path(self) -> Optional[str]:
        """Get the current file path."""
        return self.current_file_path

    def set_readonly(self, readonly: bool) -> None:
        """Set the text input as readonly."""
        if self.text_input:
            self.text_input.setReadOnly(readonly)
```

```
    def update_config(self, config: InputConfig) -> None:
        """Update input configuration and refresh UI."""
        self.config = config
        self.setup_ui()
        self.apply_styling()

    def set_theme(self, theme_name: str) -> None:
        """Apply a different theme to the input section."""
        self.gui_config.set_theme(theme_name)
        self.apply_styling()


# Backward compatibility - keep your original simple input
class InputSection(ConfigurableInputSection):
    """Simple input section for backward compatibility."""

    def __init__(self, parent: Optional[QWidget] = None) -> None:
        super().__init__(parent=parent)
```

result.py

```
from PyQt6.QtWidgets import (QWidget, QVBoxLayout, QHBoxLayout,
QLabel,  # type: ignore
                             QScrollArea, QFrame, QPushButton,
QTextEdit,
                             QProgressBar, QSplitter, QMessageBox,
QFileDialog, QGridLayout, QDialog, QFormLayout, QGroupBox,
QDialogButtonBox)  # type: ignore
from PyQt6.QtCore import Qt, QTimer, pyqtSignal,
QPropertyAnimation, QEasingCurve # type: ignore
from PyQt6.QtGui import QFont, QPalette, QTextCharFormat, QColor #
type: ignore
from typing import Optional, List, Dict, Any
from ..service.searchservice import CVMatch
from ..database.models import SessionLocal, ApplicantProfile
from .general_config import gui_config, ResultConfig
import time
import subprocess
import platform
import os
from ..service.service_provider import get_search_service
```

```
class ConfigurableResultCard(QFrame):
    """Highly configurable individual result card widget."""

    # Signals - remove card_clicked & card_double_clicked signals
    # Keep only the signal needed for the "View PDF" functionality
    view_pdf_requested = pyqtSignal(str)  # Emit the PDF path when
view button clicked

    def __init__(self,
                 match_data: CVMatch,
                 config: Optional[ResultConfig] = None,
                 parent: Optional[QWidget] = None) -> None:
        super().__init__(parent)

        # Configuration
        self.config = config or gui_config.result
        self.gui_config = gui_config

        # Data
        self.match_data = match_data
        self.db = SessionLocal()
        self.profile =
self.db.query(ApplicantProfile).get(self.match_data.applicant_id)
        self.db.close()
        self.name = self.profile.first_name + " " +
self.profile.last_name if self.profile else "Unknown Applicant"
        self.cv_path = self.match_data.cv_path
        self.score = self.match_data.score
        self.occurrences = self.match_data.occurrences

        # Remove is_selected property, as cards won't be
selectable
        # self.is_selected = False

        # Animation
        self.animation: Optional[QPropertyAnimation] = None

        self.setup_ui()
        self.apply_styling()

    def setup_ui(self) -> None:
        """Set up the result card UI with better horizontal space
usage."""
```

```
        self.setFrameStyle(QFrame.Shape.Box)
        self.setCursor(Qt.CursorShape.ArrowCursor)

        layout = QVBoxLayout(self)
        layout.setContentsMargins(
            self.gui_config.spacing.margin_medium,
            self.gui_config.spacing.margin_small,
            self.gui_config.spacing.margin_medium,
            self.gui_config.spacing.margin_small
        )
        layout.setSpacing(self.gui_config.spacing.margin_small)

        # Header section with name and match count
        header_layout = QHBoxLayout()

        # Applicant name header
        name_label = QLabel(self.name)
        name_label.setObjectName("applicantNameLabel")
        name_font = QFont(self.gui_config.fonts.family_primary)
        name_font.setBold(True)
        name_font.setPointSize(self.gui_config.fonts.size_medium)
# Slightly smaller
        name_label.setFont(name_font)

        # Matches count
        matches_label = QLabel(f"Matches: {self.score}")
        matches_label.setObjectName("matchesLabel")
        matches_font = QFont(self.gui_config.fonts.family_primary)

matches_font.setPointSize(self.gui_config.fonts.size_small)
        matches_label.setFont(matches_font)

        header_layout.addWidget(name_label)
        header_layout.addStretch(1)
        header_layout.addWidget(matches_label)

        # Matched keywords section with better styling
        keywords_label = QLabel("Matched keywords:")
        keywords_label.setObjectName("matchedKeywordsLabel")

keywords_label.setFont(QFont(self.gui_config.fonts.family_primary,
weight=QFont.Weight.Bold))

        # Create keyword chips in a flow layout
```

```
        keywords_widget = QWidget()
        keywords_layout = QVBoxLayout(keywords_widget)
        keywords_layout.setContentsMargins(0, 0, 0, 0)
        keywords_layout.setSpacing(4)

        # Add each keyword with its occurrence count as a styled
label
        for key in self.occurrences:
            occurrence_text = f"{self.occurrences.get(key)}
occurrence{'s' if self.occurrences.get(key) > 1 else ''}"
            keyword_item = QLabel(f"• {key}: {occurrence_text}")
            keyword_item.setObjectName("keywordItem")
            keyword_item.setStyleSheet(f"""
                padding: 2px 5px;
                color: {self.gui_config.colors.text_primary};
                background-color:
{self.gui_config.colors.bg_secondary};
                border-radius: 3px;
            """)
            keywords_layout.addWidget(keyword_item)

        # Buttons section
        buttons_layout = QHBoxLayout()
        buttons_layout.setSpacing(8)

        details_btn = QPushButton("CV Summary")
        details_btn.setObjectName("detailsBtn")

        view_cv_btn = QPushButton("View CV")
        view_cv_btn.setObjectName("viewCVBtn")

        buttons_layout.addWidget(details_btn)
        buttons_layout.addWidget(view_cv_btn)

        # Add all elements to the main layout
        layout.addLayout(header_layout)
        layout.addWidget(keywords_label)
        layout.addWidget(keywords_widget)
        layout.addStretch(1)
        layout.addLayout(buttons_layout)

        # Connect button signals
        details_btn.clicked.connect(self.show_details)
        view_cv_btn.clicked.connect(self.open_pdf)
```

```
    def show_details(self) -> None:
        """Show details for this CV using CV Summary Window"""
        from .cv_summary_window import CVSummaryWindow

        # Get service directly
        service = get_search_service()

        # Now use the service directly
        cv_details =
service.get_cv_details(self.match_data.resume_id)

        # Extract profile data with fallbacks
        name = self.name
        birthdate = self.profile.date_of_birth if
self.profile.date_of_birth else "Not available"
        address = self.profile.address if self.profile.address
else "Not available"
        phone = self.profile.phone_number if
self.profile.phone_number else "Not available"

        # Placeholder data - in a real app, you'd extract these
from the CV or database
        # You could add functions to parse CV text from
self.match_data.cv_path
        skills = list(self.occurrences.keys())  # Use matched
keywords as skills for demo

        jobs = cv_details.get("jobs", [])

        education = cv_details.get("education", [])
        # Create and show the summary window
        self.summary_window = CVSummaryWindow(
            name=name,
            birthdate=birthdate,
            address=address,
            phone=phone,
            skills=skills,
            jobs=jobs,
            education=education
        )

        # Show the window as non-modal
        self.summary_window.show()
```

```
    def open_pdf(self) -> None:
        """Open the CV PDF file with the system's default PDF
viewer"""
        if not self.cv_path or not os.path.exists(self.cv_path):
            QMessageBox.warning(
                self,
                "File Not Found",
                f"The PDF file could not be found
at:\n{self.cv_path}"
            )
            return

        try:
            # Use the appropriate command based on the operating
system
            if platform.system() == 'Windows':
                os.startfile(self.cv_path)
            elif platform.system() == 'Darwin':  # macOS
                subprocess.run(['open', self.cv_path])
            else:  # Linux and other Unix-like systems
                subprocess.run(['xdg-open', self.cv_path])
        except Exception as e:
            QMessageBox.critical(
                self,
                "Error Opening PDF",
                f"Could not open the PDF file:\n{str(e)}"
            )

    def _highlight_pattern_in_snippet(self, text_edit: QTextEdit,
pattern: str) -> None:
        """Highlight the pattern in the snippet text."""
        try:

            cursor = text_edit.textCursor()
            format = QTextCharFormat()

format.setBackground(QColor(self.gui_config.colors.warning))

format.setForeground(QColor(self.gui_config.colors.text_primary))

            # Find and highlight all occurrences
            text = text_edit.toPlainText()
            start = 0
```

```
            while True:
                pos = text.lower().find(pattern.lower(), start)
                if pos == -1:
                    break

                cursor.setPosition(pos)
                cursor.setPosition(pos + len(pattern),
cursor.MoveMode.KeepAnchor)
                cursor.mergeCharFormat(format)
                start = pos + 1

        except Exception:
            # Fallback: just display without highlighting
            pass

    def apply_styling(self) -> None:
        """Apply CSS styling to the card."""
        base_style = f"""
        QFrame {{
            border: {self.gui_config.spacing.border_width_thin}px
solid {self.gui_config.colors.border_medium};
            border-radius:
{self.gui_config.spacing.border_radius_large}px;
            background-color: {self.gui_config.colors.bg_primary};
            margin: {self.gui_config.spacing.margin_small}px;
        }}
        """

        # if self.config.card_hover_effect:
        #     base_style += f"""
        #     QFrame:hover {{
        #         border:
{self.gui_config.spacing.border_width_medium}px solid
{self.gui_config.colors.secondary};
        #         background-color:
{self.gui_config.colors.bg_secondary};
        #     }}
        #     """

        # Label styles
        label_styles = f"""
        QLabel#applicantNameLabel {{
            color: {self.gui_config.colors.text_primary};
            font-weight: bold;
```

```
            margin-bottom:
{self.gui_config.spacing.margin_medium}px;
            background-color: transparent;
            border: none;
            border-radius:
{self.gui_config.spacing.border_radius_medium}px;
            padding: {self.gui_config.spacing.padding_small}px;
        }}
        QLabel#matchedKeywordsLabel {{
            color: {self.gui_config.colors.text_primary};
            background-color: transparent;
            border: none;
            font-weight: bold;
        }}
        QLabel#keywordItem {{
            background-color: transparent;
            border: none;
            padding-left:
{self.gui_config.spacing.padding_small}px;
        }}
        """

        # Button styles
        button_styles = f"""
        QPushButton {{
            background-color: {self.gui_config.colors.secondary};
            color: {self.gui_config.colors.text_light};
            border: none;
            border-radius:
{self.gui_config.spacing.border_radius_medium}px;
            padding: {self.gui_config.spacing.padding_small}px
{self.gui_config.spacing.padding_medium}px;
        }}
        QPushButton:hover {{
            background-color: {self.gui_config.colors.primary};
        }}
        QPushButton#detailsBtn {{
            background-color: {self.gui_config.colors.secondary};
        }}
        QPushButton#viewCVBtn {{
            background-color: {self.gui_config.colors.accent};
        }}
        """
```

```
        full_style = base_style + label_styles + button_styles

        # Apply custom card style if provided
        if self.config.custom_card_style:
            full_style += f"QFrame {{
{self.config.custom_card_style} }}"

        self.setStyleSheet(full_style)

    # def mousePressEvent(self, event) -> None:
    #     """Handle mouse press events."""
    #     if event.button() == Qt.MouseButton.LeftButton and
self.config.enable_card_selection:
    #         self.set_selected(not self.is_selected)
    #         self.card_clicked.emit(self.match_data)
    #     super().mousePressEvent(event)

    # def mouseDoubleClickEvent(self, event) -> None:
    #     """Handle mouse double-click events."""
    #     if event.button() == Qt.MouseButton.LeftButton:
    #         self.card_double_clicked.emit(self.match_data)
    #     super().mouseDoubleClickEvent(event)

    # def set_selected(self, selected: bool) -> None:
    #     """Set the selection state of the card."""
    #     self.is_selected = selected
    #     if selected:
    #         self.setStyleSheet(self.styleSheet() + f"""
    #         QFrame {{
    #             border:
{self.gui_config.spacing.border_width_medium}px solid
{self.gui_config.colors.primary} !important;
    #             background-color:
{self.gui_config.colors.bg_secondary} !important;
    #         }}
    #         """)
    #     else:
    #         self.apply_styling()  # Reset to normal styling

    def animate_entry(self) -> None:
        """Animate card entry if animations are enabled."""
        if not self.config.enable_animations:
            return
```

```
        try:

            self.animation = QPropertyAnimation(self, b"geometry")
            self.animation.setDuration(300)

self.animation.setEasingCurve(QEasingCurve.Type.OutCubic)

            # Start from slightly smaller size
            start_rect = self.geometry()
            start_rect.setHeight(int(start_rect.height() * 0.8))

            end_rect = self.geometry()

            self.animation.setStartValue(start_rect)
            self.animation.setEndValue(end_rect)
            self.animation.start()

        except Exception:
            # Fallback: no animation
            pass

    def get_match_data(self) -> Dict[str, Any]:
        """Get the match data for this card."""
        return self.match_data.copy()


class ConfigurableResultsSection(QWidget):
    """Highly configurable results section displaying search
results and statistics."""

    # Remove the selection-related signals
    export_requested = pyqtSignal(str)  # Emitted when export is
requested with format
    results_cleared = pyqtSignal()  # Emitted when results are
cleared

    def __init__(self,
                 config: Optional[ResultConfig] = None,
                 parent: Optional[QWidget] = None) -> None:
        super().__init__(parent)

        # Configuration
        self.config = config or gui_config.result
        self.gui_config = gui_config
```

```
        # Data
        self.current_results: List[CVMatch] = []
        self.search_time: float = 0.0

        # Remove selected cards tracking
        # self.selected_cards: List[ConfigurableResultCard] = []

        # UI Elements
        self.results_title: Optional[QLabel] = None
        self.results_count_label: Optional[QLabel] = None
        self.search_time_label: Optional[QLabel] = None
        self.progress_bar: Optional[QProgressBar] = None
        self.export_btn: Optional[QPushButton] = None
        self.clear_results_btn: Optional[QPushButton] = None
        self.scroll_area: Optional[QScrollArea] = None
        self.scroll_content: Optional[QWidget] = None
        self.scroll_layout: Optional[QVBoxLayout] = None
        self.empty_state_label: Optional[QLabel] = None

        self.setup_ui()
        self.apply_styling()

    def setup_ui(self) -> None:
        """Set up the results section UI."""
        layout = QVBoxLayout(self)
        layout.setContentsMargins(
            self.gui_config.spacing.margin_medium,
            self.gui_config.spacing.margin_small,
            self.gui_config.spacing.margin_medium,
            self.gui_config.spacing.margin_small
        )
        layout.setSpacing(self.gui_config.spacing.margin_small)

        # Results header section
        if self.config.show_statistics:
            self.setup_results_header(layout)

        # Results content area
        self.setup_results_content(layout)

    def setup_results_header(self, parent_layout: QVBoxLayout) ->
None:
        """Set up the results header with statistics."""
```

```
        # Header frame
        header_frame = QFrame()
        header_frame.setObjectName("resultsHeaderFrame")
        header_frame.setFrameStyle(QFrame.Shape.Box)

        header_layout = QVBoxLayout(header_frame)

header_layout.setSpacing(self.gui_config.spacing.margin_medium)
        header_layout.setContentsMargins(
            self.gui_config.spacing.padding_medium,
            self.gui_config.spacing.padding_medium,
            self.gui_config.spacing.padding_medium,
            self.gui_config.spacing.padding_medium
        )

        # Title and stats row
        title_row = QHBoxLayout()

        # Results title
        self.results_title = QLabel(self.config.title_text)
        self.results_title.setObjectName("resultsTitle")

        title_font = QFont(self.gui_config.fonts.family_primary)
        title_font.setPointSize(self.gui_config.fonts.size_large)
        title_font.setBold(True)
        self.results_title.setFont(title_font)

        # Results count
        self.results_count_label = QLabel("No results")
        self.results_count_label.setObjectName("resultsCount")

        # Search time
        self.search_time_label = QLabel("⏱️ Time: --")
        self.search_time_label.setObjectName("searchTime")

        title_row.addWidget(self.results_title)
        title_row.addStretch()
        title_row.addWidget(self.results_count_label)
        title_row.addWidget(self.search_time_label)

        # Progress bar (for search indication)
        if self.config.show_progress_bar:
            self.progress_bar = QProgressBar()
            self.progress_bar.setObjectName("resultsProgressBar")
```

```
            self.progress_bar.setVisible(False)

        # Action buttons
        actions_row = QHBoxLayout()

        if self.config.show_export_button:
            self.export_btn =
QPushButton(self.config.export_button_text)
            self.export_btn.setObjectName("exportBtn")
            self.export_btn.setEnabled(False)
            actions_row.addWidget(self.export_btn)

        if self.config.show_clear_button:
            self.clear_results_btn =
QPushButton(self.config.clear_button_text)

self.clear_results_btn.setObjectName("clearResultsBtn")
            self.clear_results_btn.setEnabled(False)
            actions_row.addWidget(self.clear_results_btn)

        actions_row.addStretch()

        header_layout.addLayout(title_row)
        if self.config.show_progress_bar:
            header_layout.addWidget(self.progress_bar)
        # header_layout.addLayout(actions_row)

        parent_layout.addWidget(header_frame)

        # Connect signals
        if self.export_btn:
            self.export_btn.clicked.connect(self.export_results)
        if self.clear_results_btn:

self.clear_results_btn.clicked.connect(self.clear_results)

    def setup_results_content(self, parent_layout: QVBoxLayout) ->
None:
        """Set up the scrollable results content area."""
        # Results scroll area
        self.scroll_area = QScrollArea()
        self.scroll_area.setObjectName("resultsScrollArea")
        self.scroll_area.setWidgetResizable(True)
```

```
        # Scrollable content widget
        self.scroll_content = QWidget()
        self.scroll_layout = QVBoxLayout(self.scroll_content)
        self.scroll_content.setObjectName("resultsScrollContent")
        self.scroll_layout.setObjectName("resultsScrollLayout")
        self.scroll_layout.setContentsMargins(
            self.gui_config.spacing.margin_medium,
            self.gui_config.spacing.margin_medium,
            self.gui_config.spacing.margin_medium,
            self.gui_config.spacing.margin_medium
        )

self.scroll_layout.setSpacing(self.gui_config.spacing.margin_small
)

        # Empty state label
        self.empty_state_label =
QLabel(self.config.empty_state_message)
        self.empty_state_label.setObjectName("emptyStateLabel")

self.empty_state_label.setAlignment(Qt.AlignmentFlag.AlignCenter)

        self.scroll_layout.addWidget(self.empty_state_label)
        self.scroll_layout.addStretch()

        self.scroll_area.setWidget(self.scroll_content)
        parent_layout.addWidget(self.scroll_area)

    def apply_styling(self) -> None:
        """Apply CSS styling to all components."""
        # Header frame style
        header_style = f"""
        QFrame#resultsHeaderFrame {{
            border:
{self.gui_config.spacing.border_width_medium}px solid
{self.gui_config.colors.border_medium};
            border-radius:
{self.gui_config.spacing.border_radius_large}px;
            background-color: {self.gui_config.colors.bg_primary};
        }}
        """

        # Title and label styles
        title_styles = f"""
```

```
        QLabel#resultsTitle {{
            color: {self.gui_config.colors.text_primary};
        }}
        QLabel#resultsCount {{
            color: {self.gui_config.colors.text_secondary};
            font-weight: bold;
        }}
        QLabel#searchTime {{
            color: {self.gui_config.colors.warning};
            font-weight: bold;
        }}
        """

        # Button styles
        button_styles = f"""
        QPushButton#exportBtn {{
            background-color: {self.gui_config.colors.secondary};
            color: {self.gui_config.colors.text_light};
            border: none;
            border-radius:
{self.gui_config.spacing.border_radius_medium}px;
            padding: {self.gui_config.spacing.padding_small}px
{self.gui_config.spacing.padding_medium}px;
            font-weight: bold;
        }}
        QPushButton#exportBtn:hover {{
            background-color: {self.gui_config.colors.primary};
        }}
        QPushButton#exportBtn:disabled {{
            background-color:
{self.gui_config.colors.border_medium};
        }}

        QPushButton#clearResultsBtn {{
            background-color: {self.gui_config.colors.danger};
            color: {self.gui_config.colors.text_light};
            border: none;
            border-radius:
{self.gui_config.spacing.border_radius_medium}px;
            padding: {self.gui_config.spacing.padding_small}px
{self.gui_config.spacing.padding_medium}px;
            font-weight: bold;
        }}
        QPushButton#clearResultsBtn:hover {{
```

```
            background-color: {self.gui_config.colors.accent};
        }}
        QPushButton#clearResultsBtn:disabled {{
            background-color:
{self.gui_config.colors.border_medium};
        }}
        """

        # Scroll area style
        scroll_style = f"""
        QScrollArea#resultsScrollArea {{
            border:
{self.gui_config.spacing.border_width_medium}px solid
{self.gui_config.colors.border_medium};
            border-radius:
{self.gui_config.spacing.border_radius_large}px;
            background-color: {self.gui_config.colors.bg_primary};
        }}

        QWidget#resultsScrollContent {{
            border-radius:
{self.gui_config.spacing.border_radius_large}px;
            background-color: {self.gui_config.colors.bg_primary};
        }}

        QScrollBar:vertical, QScrollBar:horizontal {{
            border: none;
            background-color:
{self.gui_config.colors.border_light};
            width: 12px;
            border-radius: 6px;
        }}
        QScrollBar::handle:vertical, QScrollBar::handle:horizontal
{{
            background-color:
{self.gui_config.colors.border_medium};
            border-radius: 6px;
            min-height: 20px;
        }}
        QScrollBar::handle:vertical:hover,
QScrollBar::handle:horizontal:hover {{
            background-color:
{self.gui_config.colors.text_secondary};
        }}
```

```
        """

        # Progress bar style
        progress_style = f"""
        QProgressBar#resultsProgressBar {{
            border: {self.gui_config.spacing.border_width_thin}px
solid {self.gui_config.colors.border_medium};
            border-radius:
{self.gui_config.spacing.border_radius_medium}px;
            text-align: center;
            font-weight: bold;
        }}
        QProgressBar#resultsProgressBar::chunk {{
            background-color: {self.gui_config.colors.secondary};
            border-radius:
{self.gui_config.spacing.border_radius_medium}px;
        }}
        """

        # Empty state style
        empty_state_style = f"""
        QLabel#emptyStateLabel {{
            color: {self.gui_config.colors.text_secondary};
            font-size: {self.gui_config.fonts.size_large}pt;
            font-style: italic;
            padding: {self.gui_config.spacing.padding_large *
2}px;
            background-color: {self.gui_config.colors.bg_primary};
            border-radius:
{self.gui_config.spacing.border_radius_large}px;
            border:
{self.gui_config.spacing.border_width_medium}px dashed
{self.gui_config.colors.border_medium};
        }}
        """

        # Combine all styles
        full_style = header_style + title_styles + button_styles +
scroll_style + progress_style + empty_state_style

        # Apply custom styles if provided
        if self.config.custom_header_style:
            full_style += f"QFrame#resultsHeaderFrame {{
{self.config.custom_header_style} }}"
```

```
        if self.config.custom_empty_state_style:
            full_style += f"QLabel#emptyStateLabel {{
{self.config.custom_empty_state_style} }}"

        self.setStyleSheet(full_style)

    def show_search_progress(self, message: str = "Searching...")
-> None:
        """Show search progress indication."""
        if self.progress_bar:
            self.progress_bar.setVisible(True)
            self.progress_bar.setRange(0, 0) # Indeterminate
progress

        if self.results_count_label:
            self.results_count_label.setText(message)
        if self.search_time_label:
            self.search_time_label.setText("⏱️ Time: --")

    def hide_search_progress(self) -> None:
        """Hide search progress indication."""
        if self.progress_bar:
            self.progress_bar.setVisible(False)

    def display_results(self, results: List[CVMatch], search_time:
float, search_params: Dict[str, Any]) -> None:
        """Display search results with adaptive card sizing."""
        if self.progress_bar:
            self.progress_bar.setVisible(False)

        # Clear previous results
        self.clear_results_display()
        self.current_results = results
        self.search_time = search_time

        # Update statistics
        if self.results_count_label:
            self.results_count_label.setText(f"Found:
{len(results)} CV matches")

        if self.search_time_label:
            self.search_time_label.setText(f"⏱️ Time:
{search_time:.3f}s")
```

```
        # Handle no results case by recreating the empty state
label if needed
        if not results:
            # Always recreate the empty state label to avoid
deleted widget issues
            # This ensures we have a fresh widget that hasn't been
deleted
            self.empty_state_label =
QLabel(self.config.no_results_message)

self.empty_state_label.setObjectName("emptyStateLabel")

self.empty_state_label.setAlignment(Qt.AlignmentFlag.AlignCenter)
            self.empty_state_label.setStyleSheet(f"""
                color: {self.gui_config.colors.text_secondary};
                font-size: {self.gui_config.fonts.size_large}pt;
                font-style: italic;
                padding: {self.gui_config.spacing.padding_large *
2}px;
                background-color:
{self.gui_config.colors.bg_secondary};
                border-radius:
{self.gui_config.spacing.border_radius_large}px;
                border:
{self.gui_config.spacing.border_width_medium}px dashed
{self.gui_config.colors.border_medium};
            """)

            # Add to layout
            self.scroll_layout.addWidget(self.empty_state_label)
            return

        # Create a container widget for all results
        results_widget = QWidget()
        results_layout = QVBoxLayout(results_widget)
        results_layout.setContentsMargins(0, 0, 0, 0)

results_layout.setSpacing(self.gui_config.spacing.margin_medium)

        # Get screen dimensions for responsive layout
        from PyQt6.QtWidgets import QApplication
        screen_width = QApplication.primaryScreen().size().width()
```

```
        # Calculate card width based on screen width
        card_width = min(500, max(300, int((screen_width - 100) /
3)))

        # Adaptive column count based on screen width
        max_cols = max(1, min(4, int(screen_width / 400)))  # 1-4
columns based on width

        # Create rows of cards
        row_idx = 0
        while row_idx < len(results):
            # Create a new row container with fixed size policy
            row_widget = QWidget()
            row_layout = QHBoxLayout(row_widget)
            row_layout.setContentsMargins(0, 0, 0, 0)

row_layout.setSpacing(self.gui_config.spacing.margin_medium)
            row_layout.setAlignment(Qt.AlignmentFlag.AlignLeft)  #
Left align cards in row

            # Add cards to this row
            cards_in_row = 0
            for i in range(max_cols):
                if row_idx + i < len(results):
                    result = results[row_idx + i]
                    card = ConfigurableResultCard(result,
self.config)
                    card.setMinimumWidth(card_width)
                    card.setMaximumWidth(card_width)
                    row_layout.addWidget(card)
                    cards_in_row += 1
                else:
                    break

            # Don't add any stretch to the row - let cards keep
their size
            row_idx += cards_in_row
            results_layout.addWidget(row_widget)

        # Add the results container to the scroll area
        results_layout.addStretch()
        self.scroll_layout.addWidget(results_widget)

        # Update button states
```

```
        if self.export_btn:
            self.export_btn.setEnabled(True)
        if self.clear_results_btn:
            self.clear_results_btn.setEnabled(True)

        # Auto-scroll to results if enabled
        if self.config.auto_scroll_to_results and
self.scroll_area:
            self.scroll_area.verticalScrollBar().setValue(0)

    def clear_results_display(self) -> None:
        """Clear the results display area."""
        # Remove all widgets except stretch
        while self.scroll_layout.count() > 0:
            child = self.scroll_layout.takeAt(0)
            if child.widget():
                child.widget().deleteLater()

        # Re-add empty state if no results
        if not self.current_results and self.empty_state_label:
            self.scroll_layout.addWidget(self.empty_state_label)

        self.scroll_layout.addStretch()

    def clear_results(self) -> None:
        """Clear all results and reset the display."""
        self.current_results.clear()
        self.search_time = 0.0

        # Reset UI
        if self.results_count_label:
            self.results_count_label.setText("No results")
        if self.search_time_label:
            self.search_time_label.setText("⏱️ Time: --")
        if self.export_btn:
            self.export_btn.setEnabled(False)
        if self.clear_results_btn:
            self.clear_results_btn.setEnabled(False)

        # Clear display
        self.clear_results_display()

        # Show empty state
        if self.empty_state_label and not
```

```
self.empty_state_label.parent():
            self.scroll_layout.insertWidget(0,
self.empty_state_label)

        self.results_cleared.emit()

    def export_results(self) -> None:
        """Export search results to a file."""
        if not self.current_results:
            return

        file_path, selected_filter = QFileDialog.getSaveFileName(
            self,
            "Export Search Results",

f"search_results_{int(time.time())}.{self.config.default_export_fo
rmat}",
            self.config.supported_export_formats
        )

        if file_path:
            try:
                if file_path.endswith('.json'):
                    self._export_json(file_path)
                elif file_path.endswith('.csv'):
                    self._export_csv(file_path)
                else:  # .txt
                    self._export_txt(file_path)

                QMessageBox.information(
                    self,
                    "Export Successful",
                    f"Results exported successfully
to:\n{file_path}"
                )

                # Get format from file extension
                export_format = file_path.split('.')[-1].lower()
                self.export_requested.emit(export_format)

            except Exception as e:
                QMessageBox.critical(
                    self,
                    "Export Error",
```

```
                    f"Could not export results:\n{str(e)}"
                )

    def _export_json(self, file_path: str) -> None:
        """Export results as JSON."""
        import json
        with open(file_path, 'w', encoding='utf-8') as f:
            json.dump({
                'search_time': self.search_time,
                'result_count': len(self.current_results),
                'results': self.current_results
            }, f, indent=2, ensure_ascii=False)

    def _export_csv(self, file_path: str) -> None:
        """Export results as CSV."""
        import csv
        with open(file_path, 'w', newline='', encoding='utf-8') as
f:
            writer = csv.writer(f)
            writer.writerow(['Pattern', 'Start Position', 'End
Position', 'Snippet', 'Similarity'])
            for result in self.current_results:
                writer.writerow([
                    result.get('pattern', ''),
                    result.get('start_pos', ''),
                    result.get('end_pos', ''),
                    result.get('snippet', '').replace('\n', ' '),
                    result.get('similarity', '')
                ])

    def _export_txt(self, file_path: str) -> None:
        """Export results as plain text."""
        with open(file_path, 'w', encoding='utf-8') as f:
            f.write(f"Search Results Export\n")
            f.write(f"{'='*50}\n")
            f.write(f"Search Time: {self.search_time:.3f}
seconds\n")
            f.write(f"Total Matches:
{len(self.current_results)}\n\n")

            for i, result in enumerate(self.current_results, 1):
                f.write(f"Match #{i}\n")
                f.write(f"Pattern: {result.get('pattern',
'N/A')}\n")
```

```
                f.write(f"Position: {result.get('start_pos',
0)}-{result.get('end_pos', 0)}\n")
                if 'similarity' in result:
                    f.write(f"Similarity:
{result['similarity']:.1%}\n")
                f.write(f"Context: {result.get('snippet',
'N/A')}\n")
                f.write(f"{'-'*30}\n\n")

    # Public API methods
    def get_current_results(self) -> List[Dict[str, Any]]:
        """Get the current search results."""
        return self.current_results.copy()

    def get_search_time(self) -> float:
        """Get the last search time."""
        return self.search_time

    def update_config(self, config: ResultConfig) -> None:
        """Update result configuration and refresh UI."""
        self.config = config
        self.setup_ui()
        self.apply_styling()

    def set_theme(self, theme_name: str) -> None:
        """Apply a different theme to the results section."""
        self.gui_config.set_theme(theme_name)
        self.apply_styling()


# Backward compatibility - keep your original simple results
class ResultsSection(ConfigurableResultsSection):
    """Simple results section for backward compatibility."""

    def __init__(self, parent: Optional[QWidget] = None) -> None:
        super().__init__(parent=parent)


class ResultCard(ConfigurableResultCard):
    """Simple result card for backward compatibility."""

    def __init__(self, match_data: Dict[str, Any], parent:
Optional[QWidget] = None) -> None:
        super().__init__(match_data, parent=parent)
```

search.py

```
from PyQt6.QtWidgets import (QWidget, QVBoxLayout, QHBoxLayout,
QLabel,  # type: ignore
                             QLineEdit, QComboBox, QSpinBox,
QPushButton,
                             QFrame, QTextEdit, QScrollArea)
from PyQt6.QtCore import Qt, pyqtSignal # type: ignore
from PyQt6.QtGui import QFont # type: ignore
from typing import Optional, List, Dict, Any
from .general_config import gui_config, SearchConfig


class ConfigurableSearchControls(QWidget):
    """Highly configurable search controls section for keywords,
algorithm selection, and search parameters."""

    # Signals
    search_requested = pyqtSignal(dict)  # Emitted when search is
requested with parameters
    keyword_added = pyqtSignal(str)  # Emitted when a keyword is
added
    keywords_cleared = pyqtSignal()  # Emitted when keywords are
cleared
    algorithm_changed = pyqtSignal(str)  # Emitted when algorithm
selection changes

    def __init__(self,
                 config: Optional[SearchConfig] = None,
                 parent: Optional[QWidget] = None) -> None:
        super().__init__(parent)

        # Configuration
        self.config = config or gui_config.search
        self.gui_config = gui_config

        # UI Elements
        self.keywords_input: Optional[QLineEdit] = None
        self.keywords_display: Optional[QTextEdit] = None
        self.add_keyword_btn: Optional[QPushButton] = None
        self.clear_keywords_btn: Optional[QPushButton] = None
        self.algorithm_combo: Optional[QComboBox] = None
        self.algorithm_info: Optional[QLabel] = None
        self.top_matches_spin: Optional[QSpinBox] = None
        self.case_sensitive_combo: Optional[QComboBox] = None
```

```
        self.search_btn: Optional[QPushButton] = None

        self.setup_ui()
        self.connect_signals()
        self.apply_styling()

    def setup_ui(self) -> None:
        """Set up the search controls UI components."""
        layout = QVBoxLayout(self)
        layout.setContentsMargins(
            self.gui_config.spacing.margin_medium,
            self.gui_config.spacing.margin_small,
            self.gui_config.spacing.margin_medium,
            self.gui_config.spacing.margin_small
        )
        layout.setSpacing(self.gui_config.spacing.margin_small)

        # Keywords section
        if self.config.show_keywords_section:
            self.setup_keywords_section(layout)

        # Algorithm and parameters section (now includes search
button)
        if self.config.show_algorithm_section:
            self.setup_algorithm_section(layout)

    def setup_keywords_section(self, parent_layout: QVBoxLayout)
-> None:
        """Set up the keywords input section."""
        # Keywords frame
        keywords_frame = QFrame()
        keywords_frame.setObjectName("keywordsFrame")
        keywords_frame.setFrameStyle(QFrame.Shape.Box)

        keywords_layout = QVBoxLayout(keywords_frame)

keywords_layout.setSpacing(self.gui_config.spacing.margin_small)
        keywords_layout.setContentsMargins(
            self.gui_config.spacing.margin_small,
            self.gui_config.spacing.margin_small,
            self.gui_config.spacing.margin_small,
            self.gui_config.spacing.margin_small
        )
```

```
        # Keywords header
        keywords_header = QHBoxLayout()

        keywords_label = QLabel("🔍 Search Keywords")
        keywords_label.setObjectName("keywordsTitle")

        keywords_font =
QFont(self.gui_config.fonts.family_primary)

keywords_font.setPointSize(self.gui_config.fonts.size_large)

keywords_font.setWeight(QFont.Weight(self.gui_config.fonts.weight_
bold))
        keywords_label.setFont(keywords_font)

        keywords_header.addWidget(keywords_label)
        keywords_header.addStretch()

        # Keywords input
        self.keywords_input = QLineEdit()
        self.keywords_input.setObjectName("keywordsInput")

self.keywords_input.setPlaceholderText(self.config.keywords_placeh
older)

        # Add everything to the layout
        keywords_layout.addLayout(keywords_header)
        keywords_layout.addWidget(self.keywords_input)

        parent_layout.addWidget(keywords_frame)

    def setup_algorithm_section(self, parent_layout: QVBoxLayout)
-> None:
        """Set up the algorithm selection and parameters
section."""
        # Algorithm frame
        algorithm_frame = QFrame()
        algorithm_frame.setObjectName("algorithmFrame")
        algorithm_frame.setFrameStyle(QFrame.Shape.Box)

        algorithm_layout = QVBoxLayout(algorithm_frame)

algorithm_layout.setSpacing(self.gui_config.spacing.margin_medium)
        algorithm_layout.setContentsMargins(
```

```
            self.gui_config.spacing.margin_small,
            self.gui_config.spacing.margin_small,
            self.gui_config.spacing.margin_small,
            self.gui_config.spacing.margin_small
        )

        # Algorithm header
        algorithm_label = QLabel("⚙️ Search Algorithm &
Parameters")
        algorithm_label.setObjectName("algorithmTitle")

        algorithm_font =
QFont(self.gui_config.fonts.family_primary)

algorithm_font.setPointSize(self.gui_config.fonts.size_large)

algorithm_font.setWeight(QFont.Weight(self.gui_config.fonts.weight
_bold))
        algorithm_label.setFont(algorithm_font)

        # Parameters row - create a single row for all parameters
        params_row = QHBoxLayout()

        # Algorithm selection (now part of params_row)
        algo_select_label = QLabel("Algorithm:")
        algo_select_label.setObjectName("algorithmSelectLabel")
        algo_select_label.setMinimumWidth(120)

        self.algorithm_combo = QComboBox()
        self.algorithm_combo.setObjectName("algorithmCombo")

self.algorithm_combo.addItems(list(self.config.available_algorithm
s))

        # Set default algorithm
        default_index =
list(self.config.available_algorithms).index(self.config.default_a
lgorithm)
        self.algorithm_combo.setCurrentIndex(default_index)

        params_row.addWidget(algo_select_label)
        params_row.addWidget(self.algorithm_combo)

        # Add other parameters to the same row
```

```
        if self.config.show_search_parameters:
            self.add_search_parameters_to_row(params_row)

        params_row.addStretch()

        # Algorithm info
        if self.config.show_algorithm_info:
            self.algorithm_info = QLabel()
            self.algorithm_info.setObjectName("algorithmInfo")
            self.algorithm_info.setWordWrap(True)

self.update_algorithm_info(self.config.default_algorithm)

        algorithm_layout.addWidget(algorithm_label)
        algorithm_layout.addLayout(params_row)

        if self.config.show_algorithm_info:
            algorithm_layout.addWidget(self.algorithm_info)

        # Add some spacing between the algorithm info and search
button

algorithm_layout.addSpacing(self.gui_config.spacing.margin_small)

        parent_layout.addWidget(algorithm_frame)

    def add_search_parameters_to_row(self, params_row:
QHBoxLayout) -> None:
        """Add search parameters to an existing row."""
        # Top matches count
        matches_label = QLabel("Top Matches:")
        matches_label.setObjectName("matchesLabel")
        matches_label.setMinimumWidth(120)

        self.top_matches_spin = QSpinBox()
        self.top_matches_spin.setObjectName("topMatchesSpin")
        self.top_matches_spin.setRange(1,
self.config.max_top_matches)

self.top_matches_spin.setValue(self.config.default_top_matches)
        self.top_matches_spin.setSuffix(" results")

        # Case sensitive option
        case_label = QLabel("Case Sensitive:")
```

```
        case_label.setObjectName("caseLabel")
        case_label.setMinimumWidth(120)

        self.case_sensitive_combo = QComboBox()

self.case_sensitive_combo.setObjectName("caseSensitiveCombo")
        self.case_sensitive_combo.addItems(["No", "Yes"])

        if self.config.default_case_sensitive:
            self.case_sensitive_combo.setCurrentText("Yes")

        # Add search button to algorithm frame
        button_layout = QHBoxLayout()
        self.search_btn =
QPushButton(self.config.search_button_text)
        self.search_btn.setObjectName("searchBtn")
        self.search_btn.setMaximumHeight(50)

        button_layout.addStretch()
        button_layout.addWidget(self.search_btn)
        button_layout.addStretch()

        params_row.addWidget(matches_label)
        params_row.addWidget(self.top_matches_spin)
        params_row.addWidget(case_label)
        params_row.addWidget(self.case_sensitive_combo)
        params_row.addStretch(1)
        params_row.addLayout(button_layout)


    def apply_styling(self) -> None:
        """Apply CSS styling to all components."""
        # Frame styles
        frame_style = f"""
        QFrame#keywordsFrame, QFrame#algorithmFrame {{
            border:
{self.gui_config.spacing.border_width_medium}px solid
{self.gui_config.colors.border_medium};
            border-radius:
{self.gui_config.spacing.border_radius_large}px;
            background-color: {self.gui_config.colors.bg_primary};
            padding: {self.gui_config.spacing.padding_medium}px;
        }}
        """
```

```
        # Title styles
        title_style = f"""
        QLabel#keywordsTitle, QLabel#algorithmTitle {{
            color: {self.gui_config.colors.text_primary};
            margin-bottom:
{self.gui_config.spacing.margin_small}px;
        }}
        """

        # Input styles
        input_style = f"""
        QLineEdit#keywordsInput {{
            border: {self.gui_config.spacing.border_width_thin}px
solid {self.gui_config.colors.border_medium};
            border-radius:
{self.gui_config.spacing.border_radius_medium}px;
            background-color: {self.gui_config.colors.bg_primary};
            color: {self.gui_config.colors.text_primary};
            padding: {self.gui_config.spacing.padding_medium}px;
            font-size: {self.gui_config.fonts.size_normal}pt;
        }}
        QLineEdit#keywordsInput:focus {{
            border:
{self.gui_config.spacing.border_width_medium}px solid
{self.gui_config.colors.secondary};
        }}
        """

        # Button styles
        button_style = f"""
        QPushButton#addKeywordBtn {{
            background-color: {self.gui_config.colors.success};
            color: {self.gui_config.colors.text_light};
            border: none;
            border-radius:
{self.gui_config.spacing.border_radius_medium}px;
            padding: {self.gui_config.spacing.padding_small}px
{self.gui_config.spacing.padding_medium}px;
            font-weight: bold;
        }}
        QPushButton#addKeywordBtn:hover {{
            background-color: {self.gui_config.colors.primary};
        }}
```

```
        QPushButton#clearKeywordsBtn {{
            background-color: {self.gui_config.colors.danger};
            color: {self.gui_config.colors.text_light};
            border: none;
            border-radius:
{self.gui_config.spacing.border_radius_medium}px;
            padding: {self.gui_config.spacing.padding_small}px
{self.gui_config.spacing.padding_medium}px;
            font-weight: bold;
        }}
        QPushButton#clearKeywordsBtn:hover {{
            background-color: {self.gui_config.colors.accent};
        }}

        QPushButton#searchBtn {{
            background-color: {self.gui_config.colors.success};
            color: {self.gui_config.colors.text_light};
            border: none;
            border-radius:
{self.gui_config.spacing.border_radius_large}px;
            font-size: {self.gui_config.fonts.size_medium}pt;
            font-weight: bold;
            padding: {self.gui_config.spacing.padding_medium}px
{self.gui_config.spacing.padding_large}px;
        }}
        QPushButton#searchBtn:hover {{
            background-color: {self.gui_config.colors.primary};
        }}
        QPushButton#searchBtn:pressed {{
            background-color:
{self.gui_config.colors.text_primary};
        }}
        QPushButton#searchBtn:disabled {{
            background-color:
{self.gui_config.colors.border_medium};
            color: {self.gui_config.colors.text_secondary};
        }}
        """

        # ComboBox and SpinBox styles
        control_style = f"""
        QComboBox#algorithmCombo, QComboBox#caseSensitiveCombo,
QSpinBox#topMatchesSpin  {{
```

```
            border: {self.gui_config.spacing.border_width_thin}px
solid {self.gui_config.colors.border_medium};
            border-radius:
{self.gui_config.spacing.border_radius_medium}px;
            padding: {self.gui_config.spacing.padding_medium}px;
            font-size: {self.gui_config.fonts.size_normal}pt;
            max-height: 12px;
            background-color: {self.gui_config.colors.bg_primary};
            color: {self.gui_config.colors.text_primary};
            min-width: 120px;
        }}

        QComboBox#algorithmCombo QAbstractItemView,
QComboBox#caseSensitiveCombo QAbstractItemView {{
            background-color: {self.gui_config.colors.bg_primary};
            selection-background-color:
{self.gui_config.colors.bg_primary};
            selection-color:
{self.gui_config.colors.text_primary};
        }}

        QComboBox#algorithmCombo:focus,
QComboBox#caseSensitiveCombo:focus, QSpinBox#topMatchesSpin:focus
{{
            border:
{self.gui_config.spacing.border_width_medium}px solid
{self.gui_config.colors.secondary};
        }}

        """

        # Text display styles
        display_style = f"""


        QLabel#algorithmInfo {{
            color: {self.gui_config.colors.text_secondary};
            font-style: italic;
            background-color:
{self.gui_config.colors.bg_secondary};
            border-radius:
{self.gui_config.spacing.border_radius_medium}px;
            padding: {self.gui_config.spacing.padding_medium}px;
        }}
```

```
        """

        # Combine all styles
        full_style = frame_style + title_style + input_style +
button_style + control_style + display_style

        # Apply custom styles if provided
        if self.config.custom_frame_style:
            full_style += f"QFrame {{
{self.config.custom_frame_style} }}"

        if self.config.custom_button_style:
            full_style += f"QPushButton {{
{self.config.custom_button_style} }}"

        if self.config.custom_input_style:
            full_style += f"QLineEdit, QComboBox, QSpinBox {{
{self.config.custom_input_style} }}"

        self.setStyleSheet(full_style)

    def connect_signals(self) -> None:
        """Connect internal signals."""
        if self.keywords_input:

self.keywords_input.textChanged.connect(self.on_keywords_changed)
        if self.algorithm_combo:

self.algorithm_combo.currentTextChanged.connect(self.update_algori
thm_info)

self.algorithm_combo.currentTextChanged.connect(self.algorithm_cha
nged.emit)
        if self.search_btn:
            self.search_btn.clicked.connect(self.request_search)

    def on_keywords_changed(self) -> None:
        """Handle keyword input changes."""
        has_keywords = bool(self.keywords_input.text().strip())
        if self.search_btn:
            self.search_btn.setEnabled(has_keywords)

    def get_keywords(self) -> List[str]:
        """Get the current keywords string."""
```

```
        if not self.keywords_input:
            return ""
        return self.keywords_input.text().strip()

    def update_algorithm_info(self, algorithm: str) -> None:
        """Update algorithm information display."""
        if not self.algorithm_info:
            return

        info_map = {
            "KMP (Knuth-Morris-Pratt)": "ℹ️ KMP: Efficient for
single pattern matching with linear time complexity O(n+m)",
            "Boyer-Moore Simple": "ℹ️ Boyer-Moore Simple: Uses bad
character heuristic for fast pattern matching",
            "Boyer-Moore Complex": "ℹ️ Boyer-Moore Complex: Uses
both bad character and good suffix heuristics for optimal
performance",
            "Aho-Corasick": "ℹ️ Aho-Corasick: Optimal for multiple
pattern matching, finds all patterns simultaneously",
            "Fuzzy Search": "ℹ️ Fuzzy Search: Finds approximate
matches using edit distance, great for typos and variations"
        }

        self.algorithm_info.setText(info_map.get(algorithm, "ℹ️
Select an algorithm to see information"))

    def request_search(self) -> None:
        """Request search with current parameters."""
        keywords = self.get_keywords()

        if not keywords:
            from PyQt6.QtWidgets import QMessageBox # type: ignore
            QMessageBox.warning(
                self,
                "No Keywords",
                "Please enter at least one keyword to search for."
            )
            return

        # Prepare search parameters
        search_params = {
            'keywords': keywords,
            'algorithm': self.algorithm_combo.currentText() if
self.algorithm_combo else self.config.default_algorithm,
```

```
            'top_matches': self.top_matches_spin.value() if
self.top_matches_spin else self.config.default_top_matches,
            'case_sensitive':
(self.case_sensitive_combo.currentText() == "Yes") if
self.case_sensitive_combo else self.config.default_case_sensitive
        }

        self.search_requested.emit(search_params)

    def set_search_enabled(self, enabled: bool) -> None:
        """Enable or disable the search button."""
        if not self.search_btn:
            return

        self.search_btn.setEnabled(enabled)
        if not enabled:

self.search_btn.setText(self.config.searching_button_text)
        else:

self.search_btn.setText(self.config.search_button_text)

    def get_search_parameters(self) -> Dict[str, Any]:
        """Get current search parameters."""
        return {
            'keywords': self.get_keywords(),
            'algorithm': self.algorithm_combo.currentText() if
self.algorithm_combo else self.config.default_algorithm,
            'top_matches': self.top_matches_spin.value() if
self.top_matches_spin else self.config.default_top_matches,
            'case_sensitive':
(self.case_sensitive_combo.currentText() == "Yes") if
self.case_sensitive_combo else self.config.default_case_sensitive
        }

    def set_keywords(self, keywords: str) -> None:
        """Set keywords programmatically."""
        if self.keywords_input:
            self.keywords_input.setText(keywords)

    def set_algorithm(self, algorithm: str) -> None:
        """Set the selected algorithm."""
        if self.algorithm_combo and algorithm in
self.config.available_algorithms:
```

```
            self.algorithm_combo.setCurrentText(algorithm)

    def update_config(self, config: SearchConfig) -> None:
        """Update search configuration and refresh UI."""
        self.config = config
        self.setup_ui()
        self.apply_styling()

    def set_theme(self, theme_name: str) -> None:
        """Apply a different theme to the search section."""
        self.gui_config.set_theme(theme_name)
        self.apply_styling()


# Backward compatibility - keep your original simple search
controls
class SearchControls(ConfigurableSearchControls):
    """Simple search controls for backward compatibility."""

    def __init__(self, parent: Optional[QWidget] = None) -> None:
        super().__init__(parent=parent)
```

e. Search Algorithms

aho_corasick.py

```
from collections import deque
from typing import Dict, List, Optional

from .pattern_searcher import PatternSearcher, SearchMatch


class TrieNode:
    """Node class for the Trie structure used in Aho-Corasick
algorithm."""

    def __init__(self) -> None:
        self.children: Dict[str, "TrieNode"] = {}  # Dictionary to
store child nodes
        self.failure: Optional["TrieNode"] = None  # Failure link
for Aho-Corasick
        self.output: List[str] = []  # List of patterns that end
at this node
        self.is_end: bool = False  # Flag to mark end of a pattern
```

```python
class AhoCorasick:
    """
    Aho-Corasick algorithm implementation for multiple pattern
matching.

    The algorithm works in three phases:
    1. Build a trie of all patterns
    2. Construct failure links (similar to KMP failure function)
    3. Search for all patterns in the text using the automaton
    """

    def __init__(self) -> None:
        self.root: TrieNode = TrieNode()
        self.patterns: List[str] = []  # Store original patterns
for reference
        self._failure_links_built: bool = False

    def add_pattern(self, pattern: str) -> None:
        """
        Add a pattern to the trie.

        Args:
            pattern (str): The pattern to add
        """
        if not pattern:
            return

        self.patterns.append(pattern)
        node: TrieNode = self.root

        # Build trie by adding each character
        for char in pattern:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]

        # Mark end of pattern and store the pattern
        node.is_end = True
        node.output.append(pattern)

        self._failure_links_built = False
```

```
    def _build_failure_links(self) -> None:
        """
        Build failure links for the Aho-Corasick automaton.

        This creates failure links that allow efficient
backtracking
        when a mismatch occurs during pattern matching.
        """
        if self._failure_links_built:
            return

        queue: deque[TrieNode] = deque()

        # Initialize failure links for root's children
        for child in self.root.children.values():
            child.failure = self.root
            queue.append(child)

        # BFS to build failure links
        while queue:
            current: TrieNode = queue.popleft()

            for char, child in current.children.items():
                queue.append(child)

                # Find failure link for this child
                failure_node: Optional[TrieNode] = current.failure

                # Follow failure links until we find a match or
reach root
                while failure_node is not None and char not in
failure_node.children:
                    failure_node = failure_node.failure

                if failure_node is not None:
                    child.failure = failure_node.children[char]
                else:
                    child.failure = self.root

                # Copy output from failure node (for overlapping
patterns)
                if child.failure is not None:
                    child.output.extend(child.failure.output)
```

```
        self._failure_links_built = True

    def search(self, text: str) -> List[tuple[int, int, str]]:
        """
        Search for all patterns in the given text.

        Args:
            text (str): The text to search in

        Returns:
            List[tuple[int, int, str]]: List of tuples
(start_index, end_index, pattern)
                                        representing matches found
in the text
        """
        if not text or not self.patterns:
            return []

        # Build the automaton if not already built
        self._build_failure_links()

        matches: List[tuple[int, int, str]] = []
        current_node: Optional[TrieNode] = self.root

        for i, char in enumerate(text):
            # Follow failure links until we find a match or reach
root
            while current_node is not None and char not in
current_node.children:
                current_node = current_node.failure

            if current_node is None:
                current_node = self.root
                continue

            # Move to next state
            current_node = current_node.children[char]

            # Check for pattern matches at current position
            for pattern in current_node.output:
                start_index: int = i - len(pattern) + 1
                end_index: int = i
                matches.append((start_index, end_index, pattern))
```

```
        return matches


class AhoCorasickSearcher(PatternSearcher):
    """Aho-Corasick algorithm with caching for repeated pattern
sets."""

    def __init__(self):
        self._ac_instance: Optional[AhoCorasick] = None
        self._cached_patterns: List[str] = []

    def search_multiple(self, text: str, patterns: List[str]) ->
List[SearchMatch]:
        """Search for multiple patterns using Aho-Corasick
algorithm."""
        if not text or not patterns:
            return []

        # Filter out empty patterns
        valid_patterns = [p for p in patterns if p and
len(p.strip()) > 0]
        if not valid_patterns:
            return []

        # Check if we need to rebuild the automaton
        if self._ac_instance is None or set(self._cached_patterns)
!= set(
            valid_patterns
        ):
            self._build_automaton(valid_patterns)

        # Perform search
        raw_matches = self._ac_instance.search(text)  # type:
ignore
        matches: List[SearchMatch] = []

        for start_pos, end_pos, pattern in raw_matches:
            match = SearchMatch(
                pattern=pattern, start_pos=start_pos,
end_pos=end_pos, similarity=1.0
            )
            matches.append(match)

        # Sort by position, then by pattern for deterministic
```

```
results
        matches.sort(key=lambda x: (x.start_pos, x.pattern))
        return matches

    def _build_automaton(self, patterns: List[str]) -> None:
        """Build or rebuild the Aho-Corasick automaton."""
        self._ac_instance = AhoCorasick()
        for pattern in patterns:
            self._ac_instance.add_pattern(pattern)
        self._cached_patterns = patterns.copy()

    def clear_cache(self) -> None:
        """Clear the cached automaton."""
        self._ac_instance = None
        self._cached_patterns = []

    @property
    def algorithm_name(self) -> str:
        return "Aho-Corasick"

    @property
    def is_exact_match(self) -> bool:
        return True
```

boyer_moore.py

```
from typing import Dict, List

from .pattern_searcher import PatternSearcher, SearchMatch


class BoyerMooreSearcher(PatternSearcher):
    """Boyer-Moore algorithm with unified multi-pattern
interface."""

    def __init__(self, use_complex: bool = True):
        """
        Args:
            use_complex (bool): True if using good suffix
heuristic.
        """

        self.use_complex = use_complex
```

```
    def _build_bad_character_table(self, pattern: str) ->
Dict[str, int]:
        """
        Build the bad character table for Boyer-Moore algorithm.

        Args:
            pattern (str): The pattern for which to build the bad
character table

        Returns:
            Dict[str, int]: Dictionary mapping characters to their
rightmost position in pattern
        """
        bad_char: Dict[str, int] = {}
        for i in range(len(pattern)):
            bad_char[pattern[i]] = i
        return bad_char

    def _build_good_suffix_table(self, pattern: str) -> List[int]:
        """
        Build the good suffix table for Boyer-Moore algorithm.

        Args:
            pattern (str): The pattern for which to build the good
suffix table

        Returns:
            List[int]: Array where good_suffix[i] contains the
shift value for position i
        """
        pattern_length: int = len(pattern)
        good_suffix: List[int] = [0] * (pattern_length + 1)
        border: List[int] = [0] * (pattern_length + 1)

        # Compute border array (similar to KMP failure function)
        pattern_index: int = pattern_length
        suffix_index: int = pattern_length + 1
        border[pattern_index] = suffix_index

        while pattern_index > 0:
            while (
                suffix_index <= pattern_length
                and pattern[pattern_index - 1] !=
pattern[suffix_index - 1]
```

```
            ):
                if good_suffix[suffix_index] == 0:
                    good_suffix[suffix_index] = suffix_index -
pattern_index
                suffix_index = border[suffix_index]
            pattern_index -= 1
            suffix_index -= 1
            border[pattern_index] = suffix_index

        # Fill remaining entries
        suffix_index = border[0]
        for pattern_index in range(pattern_length + 1):
            if good_suffix[pattern_index] == 0:
                good_suffix[pattern_index] = suffix_index
            if pattern_index == suffix_index:
                suffix_index = border[suffix_index]

        return good_suffix

    def _boyer_moore_simple(self, text: str, pattern: str) ->
List[int]:
        """
        Boyer-Moore algorithm using only bad character heuristic.

        Args:
            text (str): The text to search in
            pattern (str): The pattern to search for

        Returns:
            List[int]: List of starting indices where pattern is
found in text
        """
        text_length: int = len(text)
        pattern_length: int = len(pattern)
        matches: List[int] = []

        if not pattern or not text:
            return matches

        if pattern_length > text_length:
            return matches

        # Build bad character table
        bad_char: Dict[str, int] =
```

```
self._build_bad_character_table(pattern)

        shift: int = 0
        while shift <= text_length - pattern_length:
            pattern_index: int = pattern_length - 1

            # Match pattern from right to left
            while (
                pattern_index >= 0
                and pattern[pattern_index] == text[shift +
pattern_index]
            ):
                pattern_index -= 1

            if pattern_index < 0:
                # Pattern found
                matches.append(shift)
                # Shift to next possible position
                shift += (
                    pattern_length - bad_char.get(text[shift +
pattern_length], -1)
                    if shift + pattern_length < text_length
                    else 1
                )
            else:
                # Mismatch occurred, use bad character heuristic
                bad_char_shift = pattern_index - bad_char.get(
                    text[shift + pattern_index], -1
                )
                shift += max(1, bad_char_shift)

        return matches

    def _boyer_moore_complex(self, text: str, pattern: str) ->
List[int]:
        """
        Boyer-Moore algorithm using both bad character and good
suffix heuristics.

        Args:
            text (str): The text to search in
            pattern (str): The pattern to search for

        Returns:
```

```
            List[int]: List of starting indices where pattern is
found in text
        """
        text_length: int = len(text)
        pattern_length: int = len(pattern)
        matches: List[int] = []

        if not pattern or not text:
            return matches

        if pattern_length > text_length:
            return matches

        # Build bad character and good suffix tables
        bad_char: Dict[str, int] =
self._build_bad_character_table(pattern)
        good_suffix: List[int] =
self._build_good_suffix_table(pattern)

        shift: int = 0
        while shift <= text_length - pattern_length:
            pattern_index: int = pattern_length - 1

            # Match pattern from right to left
            while (
                pattern_index >= 0
                and pattern[pattern_index] == text[shift +
pattern_index]
            ):
                pattern_index -= 1

            if pattern_index < 0:
                # Pattern found
                matches.append(shift)
                shift += good_suffix[0]
            else:
                # Mismatch occurred, use both heuristics
                bad_char_shift = pattern_index - bad_char.get(
                    text[shift + pattern_index], -1
                )
                good_suffix_shift = good_suffix[pattern_index + 1]
                shift += max(bad_char_shift, good_suffix_shift)

        return matches
```

```
    def search_multiple(self, text: str, patterns: List[str]) ->
List[SearchMatch]:
        """Search for multiple patterns using Boyer-Moore
algorithm."""
        if not text or not patterns:
            return []

        # Filter out empty patterns
        valid_patterns = [p for p in patterns if p and
len(p.strip()) > 0]
        if not valid_patterns:
            return []

        matches: List[SearchMatch] = []
        search_func = (
            self._boyer_moore_complex if self.use_complex else
self._boyer_moore_simple
        )

        # Use Boyer-Moore for each pattern
        for pattern in valid_patterns:
            positions = search_func(text, pattern)
            for start_pos in positions:
                end_pos = start_pos + len(pattern) - 1
                match = SearchMatch(
                    pattern=pattern,
                    start_pos=start_pos,
                    end_pos=end_pos,
                    similarity=1.0,
                )
                matches.append(match)

        # Sort by position, then by pattern for deterministic
results
        matches.sort(key=lambda x: (x.start_pos, x.pattern))
        return matches

    @property
    def algorithm_name(self) -> str:
        return "Boyer-Moore Complex" if self.use_complex else
"Boyer-Moore Simple"

    @property
```

```
    def is_exact_match(self) -> bool:
        return True
```

fuzzy_searcher.py

```
from typing import List, Tuple

from .pattern_searcher import PatternSearcher, SearchMatch


class FuzzySearcher(PatternSearcher):
    """Fuzzy search algorithm with configurable similarity
threshold."""

    def __init__(self, min_similarity: float = 0.6,
max_results_per_pattern: int = 100):
        self.min_similarity = min_similarity
        self.max_results_per_pattern = max_results_per_pattern

    def _levenshtein_distance(self, str1: str, str2: str) -> int:
        """
        Calculate the Levenshtein distance between two strings.

        The Levenshtein distance is the minimum number of
single-character edits
        (insertions, deletions, or substitutions) required to
change one string
        into another.

        Args:
            str1 (str): First string
            str2 (str): Second string

        Returns:
            int: The Levenshtein distance between the two strings
        """
        if not str1:
            return len(str2)
        if not str2:
            return len(str1)

        # Ensure str1 is the shorter string for space optimization
        if len(str1) > len(str2):
            str1, str2 = str2, str1
```

```
        len1: int = len(str1)
        len2: int = len(str2)

        prev_row: List[int] = list(range(len1 + 1))
        curr_row: List[int] = [0] * (len1 + 1)

        for j in range(1, len2 + 1):
            curr_row[0] = j

            for i in range(1, len1 + 1):
                if str1[i - 1] == str2[j - 1]:
                    curr_row[i] = prev_row[i - 1]
                else:
                    curr_row[i] = min(
                        prev_row[i] + 1,  # Deletion
                        curr_row[i - 1] + 1,  # Insertion
                        prev_row[i - 1] + 1,  # Substitution
                    )

            prev_row, curr_row = curr_row, prev_row

        return prev_row[len1]

    def _similarity_ratio(self, str1: str, str2: str) -> float:
        """
        Calculate similarity ratio between two strings (0.0 to
1.0).

        Args:
            str1 (str): First string
            str2 (str): Second string

        Returns:
            float: Similarity ratio (1.0 = identical, 0.0 =
completely different)
        """
        if not str1 and not str2:
            return 1.0
        if not str1 or not str2:
            return 0.0

        max_len: int = max(len(str1), len(str2))
        distance: int = self._levenshtein_distance(str1, str2)
```

```
        return 1.0 - (distance / max_len)

    def _fuzzy_search_text(
        self, query: str, text: str, min_similarity: float,
max_results: int
    ) -> List[Tuple[int, int, str, float]]:
        """
        Fuzzy search in text using similarity ratio threshold.

        Args:
            query (str): The search query
            text (str): The text to search in
            min_similarity (float): Minimum similarity ratio (0.0
to 1.0)
            max_results (int): Maximum number of results to return

        Returns:
            List[Tuple[int, int, str, float]]: List of tuples
(start_index, end_index, substring, similarity)
                                               sorted by similarity
(highest to lowest)
        """
        if not query or not text:
            return []

        query_len: int = len(query)
        text_len: int = len(text)
        matches: List[Tuple[int, int, str, float]] = []

        # Calculate reasonable window size based on similarity
threshold
        # Lower similarity means need to check longer substrings
        max_length_ratio: float = 1.0 / min_similarity if
min_similarity > 0 else 3.0
        max_window: int = min(int(query_len * max_length_ratio),
text_len)

        # Also set a minimum window size
        min_window: int = max(1, int(query_len * min_similarity))

        for start in range(text_len):
            for length in range(min_window, min(max_window + 1,
text_len - start + 1)):
```

```
                end: int = start + length
                substring: str = text[start:end]

                similarity: float = self._similarity_ratio(query,
substring)

                if similarity >= min_similarity:
                    matches.append((start, end - 1, substring,
similarity))

        # Sort by similarity (highest to lowest), then by start
position for ties
        matches.sort(key=lambda x: (-x[3], x[0]))

        # Return only the requested number of results
        return matches[:max_results]

    def search_multiple(self, text: str, patterns: List[str]) ->
List[SearchMatch]:
        """Search for multiple patterns using fuzzy matching."""
        if not text or not patterns:
            return []

        # Filter out empty patterns
        valid_patterns = [p for p in patterns if p and
len(p.strip()) > 0]
        if not valid_patterns:
            return []

        matches: List[SearchMatch] = []

        # Use fuzzy search for each pattern
        for pattern in valid_patterns:
            fuzzy_matches = self._fuzzy_search_text(
                query=pattern,
                text=text,
                min_similarity=self.min_similarity,
                max_results=self.max_results_per_pattern,
            )

            for start_pos, end_pos, snippet, similarity in
fuzzy_matches:
                match = SearchMatch(
                    pattern=pattern,
```

```
                    start_pos=start_pos,
                    end_pos=end_pos,
                    similarity=similarity,
                )
                matches.append(match)

        # Sort by similarity (highest first), then by position
        matches.sort(key=lambda x: (-x.similarity, x.start_pos))
        return matches

    def set_similarity_threshold(self, min_similarity: float) ->
None:
        """Update the minimum similarity threshold."""
        if not 0.0 <= min_similarity <= 1.0:
            raise ValueError("Similarity threshold must be between
0.0 and 1.0")
        self.min_similarity = min_similarity

    @property
    def algorithm_name(self) -> str:
        return f"Fuzzy Search (min_sim={self.min_similarity:.2f})"

    @property
    def is_exact_match(self) -> bool:
        return False
```

kmp_searcher.py

```
from typing import List

from .pattern_searcher import PatternSearcher, SearchMatch


class KMPSearcher(PatternSearcher):
    """KMP algorithm with unified multi-pattern interface."""

    def _generate_lps_list(self, pattern: str) -> List[int]:
        """
        Generate the Longest Proper Prefix which is also Suffix
(LPS) array.

        This array is used by the KMP algorithm to determine how
```

```
much to skip
        when a mismatch occurs.

        Args:
            pattern (str): The pattern for which to generate the
LPS array

        Returns:
            List[int]: LPS array where lps[i] contains the length
of the longest
                       proper prefix of pattern[0..i] which is
also a suffix of
                       pattern[0..i]
        """
        lps: List[int] = [0] * len(pattern)
        length: int = 0
        pointer: int = 1  # lsp[0] is always 0, start from index 1

        while pointer < len(pattern):
            if pattern[pointer] == pattern[length]:
                length += 1
                lps[pointer] = length
                pointer += 1
            else:
                if length == 0:
                    lps[pointer] = 0
                    pointer += 1
                else:
                    length = lps[length - 1]

        return lps

    def _kmp_search(self, text: str, pattern: str) -> List[int]:
        """
        Knuth-Morris-Pratt string searching algorithm.

        Args:
            text (str): The text to search in
            pattern (str): The pattern to search for

        Returns:
            List[int]: List of starting indices where pattern is
found in text
        """
```

```
        matches: List[int] = []
        lps: List[int] = self._generate_lps_list(pattern)
        text_length: int = len(text)
        pattern_length: int = len(pattern)

        # Check for impossible case
        if not pattern or not text:
            return matches
        if pattern_length > text_length:
            return matches

        text_index: int = 0
        pattern_index: int = 0

        while text_index < text_length:
            if text[text_index] == pattern[pattern_index]:
                text_index += 1
                pattern_index += 1

                if pattern_index == pattern_length:
                    matches.append(text_index - pattern_index)
                    pattern_index = lps[pattern_index - 1]
            else:
                if pattern_index != 0:
                    pattern_index = lps[pattern_index - 1]
                else:
                    text_index += 1

        return matches

    def search_multiple(self, text: str, patterns: List[str]) ->
List[SearchMatch]:
        """Search for multiple patterns using KMP algorithm."""

        if not text or not patterns:
            return []

        valid_patterns = [p for p in patterns if p and
len(p.strip()) > 0]
        if not valid_patterns:
            return []

        matches: List[SearchMatch] = []
```

```
        for pattern in valid_patterns:
            positions: List[int] = self._kmp_search(text, pattern)
            for start_pos in positions:
                end_pos: int = start_pos + len(pattern) - 1
                match: SearchMatch = SearchMatch(
                    pattern=pattern,
                    start_pos=start_pos,
                    end_pos=end_pos,
                    similarity=1.0,
                )
                matches.append(match)

        matches.sort(key=lambda x: (x.start_pos, x.pattern))

        return matches

    @property
    def algorithm_name(self) -> str:
        return "KMP"

    @property
    def is_exact_match(self) -> bool:
        return True
```

pattern_searcher.py

```
from abc import ABC, abstractmethod
from dataclasses import dataclass
from enum import Enum
from typing import List


@dataclass(frozen=True)
class SearchMatch:
    """Unified match result structure."""

    pattern: str
    start_pos: int
    end_pos: int
    similarity: float = 1.0  # Always 1.0 for exact matches,
variable for fuzzy
```

```
    @property
    def length(self) -> int:
        return self.end_pos - self.start_pos + 1


class SearchStrategy(Enum):
    """Search strategy types."""

    EXACT = "exact"
    FUZZY = "fuzzy"


class PatternSearcher(ABC):
    """Abstract base class for all pattern search algorithms."""

    @abstractmethod
    def search_multiple(self, text: str, patterns: List[str]) ->
List[SearchMatch]:
        """
        Search for multiple patterns in text.

        Args:
            text: Text to search in
            patterns: List of patterns to search for

        Returns:
            List of SearchMatch objects sorted by position
        """
        pass

    def search_single(self, text: str, pattern: str) ->
List[SearchMatch]:
        """
        Search for a single pattern in text.
        Default implementation delegates to search_multiple.
        """
        return self.search_multiple(text, [pattern])

    @property
    @abstractmethod
    def algorithm_name(self) -> str:
        """Return the name of the algorithm."""
        pass
```

```
    @property
    @abstractmethod
    def is_exact_match(self) -> bool:
        """Return True if this algorithm performs exact
matching."""
        pass
```

search_engine.py

```
import time
from dataclasses import dataclass
from enum import Enum
from typing import Dict, List, Optional, Union

from .aho_corasick import AhoCorasickSearcher
from .boyer_moore import BoyerMooreSearcher
from .fuzzy_searcher import FuzzySearcher
from .kmp_searcher import KMPSearcher
from .pattern_searcher import PatternSearcher, SearchMatch,
SearchStrategy


class AlgorithmType(Enum):
    """Available exact match algorithms."""

    KMP = "kmp"
    BOYER_MOORE_SIMPLE = "boyer_moore_simple"
    BOYER_MOORE_COMPLEX = "boyer_moore_complex"
    AHO_CORASICK = "aho_corasick"


@dataclass
class SearchConfig:
    """Search configuration parameters."""

    case_sensitive: bool = True
    max_results: int = 100
    exact_algorithm: AlgorithmType = AlgorithmType.AHO_CORASICK
    fuzzy_min_similarity: float = 0.6
    use_fuzzy_fallback: bool = True
```

```
@dataclass
class SearchStats:
    """Search execution statistics."""

    total_time: float
    exact_matches_count: int
    fuzzy_matches_count: int
    strategy_used: SearchStrategy
    algorithm_used: str
    patterns_searched: int


class SearchEngine:
    """
    Search engine that tries exact matching first,
    then falls back to fuzzy search if needed.
    """

    def __init__(self, config: SearchConfig = SearchConfig()):
        self.config = config
        self._exact_searchers: Dict[AlgorithmType,
PatternSearcher] = {
            AlgorithmType.KMP: KMPSearcher(),
            AlgorithmType.BOYER_MOORE_SIMPLE:
BoyerMooreSearcher(use_complex=False),
            AlgorithmType.BOYER_MOORE_COMPLEX:
BoyerMooreSearcher(use_complex=True),
            AlgorithmType.AHO_CORASICK: AhoCorasickSearcher(),
        }
        self._fuzzy_searcher =
FuzzySearcher(min_similarity=config.fuzzy_min_similarity)

    def search(
        self,
        text: str,
        patterns: Union[str, List[str]],
        config: Optional[SearchConfig] = None,
    ) -> tuple[List[SearchMatch], SearchStats]:
        """
        Unified search method with exact matching and fuzzy
fallback.

        Args:
            text: Text to search in
```

```
            patterns: Pattern(s) to search for
            config: Optional search configuration (uses default if
None)

        Returns:
            Tuple of (matches, search statistics)
        """
        # Use provided config or default
        search_config = config or self.config

        # Normalize patterns input
        if isinstance(patterns, str):
            pattern_list = [patterns]
        else:
            pattern_list = list(patterns)

        # Handle case sensitivity
        search_text = text if search_config.case_sensitive else
text.lower()
        search_patterns = (
            pattern_list
            if search_config.case_sensitive
            else [p.lower() for p in pattern_list]
        )

        start_time = time.perf_counter()

        # Step 1: Try exact matching
        exact_searcher =
self._exact_searchers[search_config.exact_algorithm]
        exact_matches =
exact_searcher.search_multiple(search_text, search_patterns)

        # Step 2: Use fuzzy search if no exact matches and
fallback is enabled
        fuzzy_matches: List[SearchMatch] = []
        strategy_used = SearchStrategy.EXACT

        if not exact_matches and search_config.use_fuzzy_fallback:
            self._fuzzy_searcher.set_similarity_threshold(
                search_config.fuzzy_min_similarity
            )
            fuzzy_matches = self._fuzzy_searcher.search_multiple(
                search_text, search_patterns
```

```
            )
            strategy_used = SearchStrategy.FUZZY

        end_time = time.perf_counter()

        # Combine results (exact matches take priority)
        all_matches = exact_matches + fuzzy_matches

        # Restore original case for patterns if needed
        if not search_config.case_sensitive:
            all_matches = self._restore_original_patterns(
                all_matches, pattern_list, search_patterns
            )

        # Limit results
        limited_matches = all_matches[: search_config.max_results]

        # Create statistics
        stats = SearchStats(
            total_time=end_time - start_time,
            exact_matches_count=len(exact_matches),
            fuzzy_matches_count=len(fuzzy_matches),
            strategy_used=strategy_used,
            algorithm_used=exact_searcher.algorithm_name,
            patterns_searched=len(pattern_list),
        )

        return limited_matches, stats

    def _restore_original_patterns(
        self,
        matches: List[SearchMatch],
        original_patterns: List[str],
        search_patterns: List[str],
    ) -> List[SearchMatch]:
        """Restore original case for patterns in search
results."""
        pattern_mapping = dict(zip(search_patterns,
original_patterns))

        restored_matches = []
        for match in matches:
            original_pattern = pattern_mapping.get(match.pattern,
match.pattern)
```

```
            restored_match = SearchMatch(
                pattern=original_pattern,
                start_pos=match.start_pos,
                end_pos=match.end_pos,
                similarity=match.similarity,
            )
            restored_matches.append(restored_match)

        return restored_matches

    def search_exact_only(
        self,
        text: str,
        patterns: Union[str, List[str]],
        algorithm: AlgorithmType = AlgorithmType.AHO_CORASICK,
    ) -> tuple[List[SearchMatch], SearchStats]:
        """Search using only exact matching algorithms."""
        config = SearchConfig(exact_algorithm=algorithm,
use_fuzzy_fallback=False)
        return self.search(text, patterns, config)

    def search_fuzzy_only(
        self, text: str, patterns: Union[str, List[str]],
min_similarity: float = 0.6
    ) -> tuple[List[SearchMatch], SearchStats]:
        """Search using only fuzzy matching."""
        start_time = time.perf_counter()

        # Normalize patterns
        if isinstance(patterns, str):
            pattern_list = [patterns]
        else:
            pattern_list = list(patterns)

        # Set similarity and search

self._fuzzy_searcher.set_similarity_threshold(min_similarity)
        matches = self._fuzzy_searcher.search_multiple(text,
pattern_list)

        end_time = time.perf_counter()

        stats = SearchStats(
            total_time=end_time - start_time,
```

```
            exact_matches_count=0,
            fuzzy_matches_count=len(matches),
            strategy_used=SearchStrategy.FUZZY,
            algorithm_used=self._fuzzy_searcher.algorithm_name,
            patterns_searched=len(pattern_list),
        )

        return matches, stats

    def benchmark_algorithms(self, text: str, patterns: List[str])
-> Dict[str, float]:
        """Benchmark all exact matching algorithms."""
        results = {}

        for algo_type, searcher in self._exact_searchers.items():
            start_time = time.perf_counter()
            searcher.search_multiple(text, patterns)
            end_time = time.perf_counter()
            results[searcher.algorithm_name] = end_time -
start_time

        return results

    def update_config(self, config: SearchConfig) -> None:
        """Update the search configuration."""
        self.config = config

self._fuzzy_searcher.set_similarity_threshold(config.fuzzy_min_sim
ilarity)

    def get_available_algorithms(self) -> List[str]:
        """Get list of available exact matching algorithms."""
        return [searcher.algorithm_name for searcher in
self._exact_searchers.values()]


# Convenience functions for quick searches
def search_text(
    text: str,
    patterns: Union[str, List[str]],
    algorithm: str = "aho_corasick",
    use_fuzzy_fallback: bool = True,
) -> List[SearchMatch]:
    """Quick search function with sensible defaults."""
```

```
    config = SearchConfig(
        exact_algorithm=AlgorithmType(algorithm.lower()),
        use_fuzzy_fallback=use_fuzzy_fallback,
    )
    engine = SearchEngine(config)
    matches, _ = engine.search(text, patterns)
    return matches


def fuzzy_search(
    text: str, patterns: Union[str, List[str]], min_similarity:
float = 0.6
) -> List[SearchMatch]:
    """Quick fuzzy search function."""
    engine = SearchEngine()
    matches, _ = engine.search_fuzzy_only(text, patterns,
min_similarity)
    return matches
```

f. Service

encryptservice.py

```
from ..database.models import SessionLocal, ApplicationDetail,
ApplicantProfile
from ..encryption.CAE import CAE
from ..config.config import ENCRYPTION_PASSWORD
import base64
from sqlalchemy import text

class EncryptService:
    def __init__(self):
        self.cae = CAE()
        self.password = ENCRYPTION_PASSWORD
        self.db = None

    def is_encrypted(self, text):
        """Check if text is already encrypted"""
        if not text:
            return False
        try:
            decoded = base64.b64decode(text)
            if len(decoded) >= self.cae.salt_bytes:
```

```
                return True
            except Exception:
                return False
            return False

    def get_db(self):
        """Get database session, creating one if needed"""
        if self.db is None:
            self.db = SessionLocal()
        return self.db

    def close_db(self):
        """Close database connection if open"""
        if self.db:
            self.db.close()
            self.db = None

    def encrypt_text(self, text):
        """Encrypt a single text value if not already encrypted"""
        if not text or self.is_encrypted(text):
            return text
        return self.cae.encrypt(text, self.password)

    def encrypt_batch(self, batch_size=100,
progress_callback=None):
        """Encrypt in smaller batches to avoid memory issues"""
        db = self.get_db()
        try:
            encryption_count = 0
            encrypted_count = 0

            # Process application details in batches
            total_apps = db.query(ApplicationDetail).count()
            for offset in range(0, total_apps, batch_size):
                applications =
db.query(ApplicationDetail).limit(batch_size).offset(offset).all()

                for app in applications:
                    if app.cv_path:
                        if not self.is_encrypted(app.cv_path):
                            app.cv_path =
self.encrypt_text(app.cv_path)
                            encryption_count += 1
                        else:
```

```
                            encrypted_count += 1

                db.commit()
                if progress_callback:
                    progress_callback(offset + len(applications),
total_apps,
                                      f"Encrypted {offset +
len(applications)}/{total_apps} applications")

            # Process applicant profiles in batches
            total_applicants = db.query(ApplicantProfile).count()
            fields_to_encrypt = ['first_name', 'last_name',
'address', 'phone_number']

            for offset in range(0, total_applicants, batch_size):
                applicants =
db.query(ApplicantProfile).limit(batch_size).offset(offset).all()

                for applicant in applicants:
                    for field in fields_to_encrypt:
                        value = getattr(applicant, field, None)
                        if value:
                            if not self.is_encrypted(value):
                                setattr(applicant, field,
self.encrypt_text(value))
                                encryption_count += 1
                            else:
                                encrypted_count += 1
                db.execute(text(f"ALTER TABLE ApplicantProfile
MODIFY COLUMN {field} TEXT"))
                db.commit()
                if progress_callback:
                    progress_callback(offset + len(applicants),
total_applicants,
                                      f"Encrypted {offset +
len(applicants)}/{total_applicants} applicants")

            print(f"Encryption completed: {encryption_count} items
encrypted, {encrypted_count} items were already encrypted.")
            return encryption_count, encrypted_count

        except Exception as e:
            db.rollback()
            print(f"Encryption failed: {str(e)}")
```

```
            raise
        finally:
            self.close_db()

    def encrypt(self, progress_callback=None):
        """Legacy method for compatibility"""
        return
self.encrypt_batch(progress_callback=progress_callback)

    def decrypt(self, text):
        """Safely decrypt text if it's encrypted"""
        if not text or not self.is_encrypted(text):
            return text
        try:
            return self.cae.decrypt(text, self.password)
        except Exception:
            # Return original if decryption fails
            return text
```

searchservice.py

```
import os
import time
import concurrent.futures
from dataclasses import dataclass
from typing import List, Tuple, Optional, Dict
from collections import Counter
from ..database.models import SessionLocal, ApplicationDetail
from ..database.pdf_utils import prepare_texts_from_pdf,
save_extracted_texts
from ..database.parser import SectionScraper
from ..database.parser import SectionScraper
from ..search_algorithms.search_engine import SearchEngine,
AlgorithmType, SearchMatch
from ..config.config import CV_FOLDER

@dataclass
class CVMatch:
    applicant_id: int
    resume_id: str
    score: int
    cv_path: str
    occurrences: Dict[str, int]
```

```python
# ini gajadi dipake tapi gpp dh keep aja dl
@dataclass
class ApplicantResult:
    applicant_id: int
    resumes: List[CVMatch]

class SearchService:
    def __init__(self, max_workers: int = None):
        self.max_workers = max_workers
        self.engine = SearchEngine()
        # Keep both caches
        self.text_cache_pattern = {}  # For searching
        self.text_cache_regex = {}
        self.section = SectionScraper()    # For structured data
extraction
        self.decryptor = None

    def preprocess_cvs(self, progress_callback=None):
        """Use pdf_utils functions directly to avoid redundant
processing"""
        try:
            db = SessionLocal()
            resumes = db.query(ApplicationDetail).all()
            from .service_provider import get_encrypt_service
            self.decryptor = get_encrypt_service()
            for resume in resumes:
                resume.cv_path =
self.decryptor.decrypt(resume.cv_path)
        finally:
            db.close()

        start = time.time()
        total = len(resumes)
        processed = 0

        if progress_callback and total > 0: progress_callback(0)

        # Create cache directory
        cache_dir = os.path.join(os.path.dirname(CV_FOLDER),
"cache")
        os.makedirs(cache_dir, exist_ok=True)

        def process_cv(resume):
```

```
            pdf_path = resume.cv_path
            if not os.path.exists(pdf_path):
                return None

            # Define cache files for both formats
            cache_regex = os.path.join(cache_dir,
f"{resume.cv_path}_regex.txt")
            cache_pattern = os.path.join(cache_dir,
f"{resume.cv_path}_pattern.txt")

            # Check if we need to parse the PDF
            need_parsing = True
            if os.path.exists(cache_pattern) and
os.path.exists(cache_regex):
                pdf_mtime = os.path.getmtime(pdf_path)
                cache_mtime = max(os.path.getmtime(cache_pattern),
os.path.getmtime(cache_regex))

                # Use cache if it's newer than the PDF
                if cache_mtime > pdf_mtime:
                    need_parsing = False
                    # Read both formats from cache
                    with open(cache_regex, 'r', encoding='utf-8')
as f:
                        text_regex = f.read()
                    with open(cache_pattern, 'r',
encoding='utf-8') as f:
                        text_pattern = f.read()
                    return resume.cv_path, (text_regex,
text_pattern)

            # Parse PDF if needed
            if need_parsing:
                # Use prepare_texts_from_pdf which handles both
extraction and formatting
                result = prepare_texts_from_pdf(pdf_path)
                if result is None:
                    return None

                text_regex, text_pattern = result

                # Save both formats to cache
                save_extracted_texts(pdf_path, cache_regex,
cache_pattern)
```

```
                return resume.cv_path, (text_regex, text_pattern)

            return None

        # Process CVs in parallel
        with
concurrent.futures.ThreadPoolExecutor(max_workers=self.max_workers
) as executor:
            for result in executor.map(process_cv, resumes):
                if result:
                    cv_id, (text_regex, text_pattern) = result
                    self.text_cache_regex[cv_id] = text_regex
                    self.text_cache_pattern[cv_id] = text_pattern

                processed += 1
                if progress_callback:
                    progress = min(100, int((processed / total) *
100))
                    progress_callback(progress)

        if progress_callback:
            progress_callback(100)

        elapsed = time.time() - start
        return elapsed, len(self.text_cache_pattern)

    def search(self, keywords: str, algorithm: str, top_k: int,
case: bool, progress_callback=None) -> Tuple[int, float,
List[CVMatch]]:
        """
        :param keywords: pattern(s) to search, space-separated
        :param algorithm: 'fuzzy' or one of exact algorithm names
        :param top_k: number of top CVs to return
        :return: total_scanned, search_time, list of CVMatch
        """
        # Split keywords by spaces
        keyword_list = [kw.strip() if case else kw.strip().lower()
for kw in keywords.replace(',', ' ').split() if kw.strip()]
        if not keyword_list:
            return 0, 0.0, []

        # Load resumes
        try:
```

```
            db = SessionLocal()
            resumes = db.query(ApplicationDetail).all()
            for resume in resumes:
                resume.cv_path =
self.decryptor.decrypt(resume.cv_path)
        finally:
            db.close()

        total_scanned = len(self.text_cache_pattern)
        start_time = time.time()
        all_matches: List[CVMatch] = []

        # Initialize the processed variable before using it
        processed = 0  # Add this line

        def process(resume) -> Optional[CVMatch]:
            pdf_path = resume.cv_path
            if not os.path.exists(pdf_path):
                return None

            # Use text_cache_pattern if available
            if resume.cv_path in self.text_cache_pattern:
                text = self.text_cache_pattern[resume.cv_path]
            else:
                # If not cached in memory, use
prepare_texts_from_pdf
                result = prepare_texts_from_pdf(pdf_path)
                if result is None:
                    return None

                # Get the pattern text (second item)
                _, text = result
                self.text_cache_pattern[resume.cv_path] = text

            all_matches = []

            # Search for each keyword separately
            for keyword in keyword_list:
                # Choose search mode
                if algorithm.lower() == 'fuzzy':
                    matches, _ =
self.engine.search_fuzzy_only(text, keyword)
                else:
                    try:
```

```
                        algo = AlgorithmType(algorithm.lower())
                    except ValueError:
                        algo = self.engine.config.exact_algorithm
                    matches, _ =
self.engine.search_exact_only(text, keyword, algo)

                all_matches.extend(matches)

            if not all_matches:
                return None

            # Count occurrences of each matched pattern
            counts = Counter(match.pattern for match in
all_matches)
            score = sum(counts.values())

            return CVMatch(
                applicant_id=resume.applicant_id,
                resume_id=resume.cv_path,
                score=score,
                cv_path=pdf_path,
                occurrences=dict(counts)
            )

        with
concurrent.futures.ThreadPoolExecutor(max_workers=self.max_workers
) as executor:
            for match in executor.map(process, resumes):
                if match:
                    all_matches.append(match)

                # Move this outside the if condition so it counts
all processed items
                processed += 1
                if progress_callback and total_scanned > 0:
                    # Calculate percentage progress (0-100)
                    progress = min(100, int((processed /
len(resumes)) * 100))
                    progress_callback(progress)

        # Sort CVs directly by score
        if progress_callback:
            progress_callback(100)
```

```
        all_matches.sort(key=lambda m: m.score, reverse=True)
        elapsed = time.time() - start_time

        return total_scanned, elapsed, all_matches[:top_k]

    def get_cv_details(self, cv_id: str) -> Dict:
        """Get structured information from a CV using regex
text"""
        if cv_id not in self.text_cache_regex:
            # Try to load from cache file
            cache_regex = os.path.join(os.path.dirname(CV_FOLDER),
"cache", f"{cv_id}_regex.txt")
            if os.path.exists(cache_regex):
                with open(cache_regex, 'r', encoding='utf-8') as
f:
                    self.text_cache_regex[cv_id] = f.read()
            else:
                # If not in cache, try to extract it from PDF
                pdf_path = os.path.join(CV_FOLDER, f"{cv_id}.pdf")
                if os.path.exists(pdf_path):
                    result = prepare_texts_from_pdf(pdf_path)
                    if result:
                        self.text_cache_regex[cv_id] = result[0]
# Store regex text

        # Extract structured information if we have the regex text
        if cv_id in self.text_cache_regex:
            regex_text = self.text_cache_regex[cv_id]
            jobs = self.section.scrape_experience(regex_text)
            education = self.section.scrape_education(regex_text)
            skills = self.section.scrape_skills(regex_text)

            return {
                "jobs": jobs,
                "education": education,
                "skills": skills
            }

        return {"jobs": [], "education": [], "skills": []}
```

service_provider.py

```python
"""Global service provider module"""
from .searchservice import SearchService
from .encryptservice import EncryptService

# Global service instance
_search_service = None
_encrypt_service = None

def get_search_service():
    """Get the global search service instance"""
    global _search_service
    if _search_service is None:
        _search_service = SearchService()
    return _search_service

def get_encrypt_service():
    global _encrypt_service
    if _encrypt_service is None:
        _encrypt_service = EncryptService()
    return _encrypt_service

def set_search_service(service):
    """Set the global search service instance"""
    global _search_service
    _search_service = service

def set_encrypt_service(service):
    global _encrypt_service
    _encrypt_service = service
```

threadservice.py

```python
from PyQt6.QtCore import QThread, pyqtSignal
from typing import Callable, Any, Optional

class PreprocessThread(QThread):
    """Thread for preprocessing CVs in background"""
    progress = pyqtSignal(int)
    finished = pyqtSignal(int, float)  # count, elapsed_time

    def __init__(self, service):
        super().__init__()
```

```
        self.service = service

    def run(self):
        elapsed, count =
self.service.preprocess_cvs(progress_callback=self.progress.emit)
        self.finished.emit(count, elapsed)

class SearchThread(QThread):
    """Thread for searching CVs in background"""
    progress = pyqtSignal(int)
    results_ready = pyqtSignal(int, float, list)  # total_docs,
elapsed_time, results

    def __init__(self, service, keywords, algorithm, top_k,
case_sensitive):
        super().__init__()
        self.service = service
        self.keywords = keywords
        self.algorithm = algorithm
        self.top_k = top_k
        self.case_sensitive = case_sensitive

    def run(self):
        total, elapsed, results = self.service.search(
            self.keywords,
            self.algorithm,
            self.top_k,
            self.case_sensitive,
            progress_callback=self.progress.emit
        )
        self.results_ready.emit(total, elapsed, results)
```

g. Main.py

Main.py

```
import sys
import traceback
import os
import time
from PyQt6.QtWidgets import (QApplication, QMainWindow,
QVBoxLayout, QWidget,
                             QLabel, QSizePolicy, QFrame,
QMessageBox, QProgressDialog,
```

```
                                QHBoxLayout)
from PyQt6.QtCore import Qt

from src.gui_components.header import HeaderComponent
from src.gui_components.search import SearchControls
from src.gui_components.result import ResultsSection
from src.database.models import SessionLocal, ApplicationDetail
from src.service.searchservice import SearchService
from src.service.threadservice import PreprocessThread,
SearchThread
from src.service.encryptservice import EncryptService
from src.service.service_provider import set_search_service,
set_encrypt_service

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("Applicant Tracking System")
        self.setGeometry(100, 100, 1280, 720)
        self.setup_ui()
        self.db = SessionLocal()
        self.cv_count = self.db.query(ApplicationDetail).count()
        self.db.close()

        # Create service and set it as global instance
        self.encryptor = EncryptService()
        set_encrypt_service(self.encryptor)
        self.encryptor.encrypt()
        self.service = SearchService()
        set_search_service(self.service)

        self.preprocess_cvs()

    def setup_ui(self):
        """Set up the main UI components"""
        # Create central widget and main layout
        central_widget = QWidget()
        self.setCentralWidget(central_widget)
        main_layout = QVBoxLayout(central_widget)
        # main_layout.setContentsMargins(20, 20, 20, 20)
        # main_layout.setSpacing(15)

        # Initialize components with better styling
        self.header = HeaderComponent(
```

```
        )

        self.search_section = SearchControls()
        self.result_section = ResultsSection()

        # Create a better status bar container
        status_container = QWidget()
        status_container.setObjectName("statusContainer")
        status_layout = QHBoxLayout(status_container)
        status_layout.setContentsMargins(0, 5, 0, 0)
        status_layout.setSpacing(8)

        # Left status section (main message)
        self.status_label = QLabel("Ready. Enter keywords to begin
searching.")
        self.status_label.setObjectName("statusLabel")

        # Right status section (CV count)
        self.cv_count_label = QLabel()
        self.cv_count_label.setObjectName("cvCountLabel")

self.cv_count_label.setAlignment(Qt.AlignmentFlag.AlignRight |
Qt.AlignmentFlag.AlignVCenter)

        # Add to layout
        status_layout.addWidget(self.status_label, 3)  # Give more
space to main status
        status_layout.addWidget(self.cv_count_label, 1)

        # Add components to layout with proper spacing
        # main_layout.addWidget(self.header)
        main_layout.addWidget(self.search_section)
        main_layout.addWidget(self.result_section)
        main_layout.addWidget(status_container)

        # Set proper size policies
        self.header.setSizePolicy(QSizePolicy.Policy.Preferred,
QSizePolicy.Policy.Fixed)

self.search_section.setSizePolicy(QSizePolicy.Policy.Preferred,
QSizePolicy.Policy.Fixed)
        self.search_section.setMinimumHeight(250)

self.result_section.setSizePolicy(QSizePolicy.Policy.Preferred,
```

```
QSizePolicy.Policy.Expanding)

self.status_label.setSizePolicy(QSizePolicy.Policy.Preferred,
QSizePolicy.Policy.Fixed)

        # Initialize search service and connect signals

self.search_section.search_requested.connect(self.on_search)

        # Apply application-wide styling
        self.setStyleSheet("""
            QMainWindow {
                background-color: #f5f5f7;
            }

            QScrollArea#resultsScrollArea {
                border: 1px solid #e0e0e0;
                border-radius: 8px;
                background-color: white;
            }

            QScrollBar:vertical {
                border: none;
                background-color: #f0f0f0;
                width: 10px;
                border-radius: 5px;
                margin: 0px;
            }

            QScrollBar::handle:vertical {
                background-color: #bbbbbb;
                border-radius: 5px;
                min-height: 30px;
            }

            QScrollBar::handle:vertical:hover {
                background-color: #999999;
            }

            QScrollBar::add-line:vertical,
QScrollBar::sub-line:vertical {
                height: 0px;
            }
```

```
            QScrollBar::add-page:vertical,
QScrollBar::sub-page:vertical {
                background: none;
            }

            /* Make labels and buttons more modern */
            QLabel {
                color: #333333;
            }

            QPushButton {
                border-radius: 6px;
            }

                        #statusContainer {
                background-color: #f8f9fa;
                border-top: 1px solid #dee2e6;
                padding: 8px 12px;
            }

            QLabel#statusLabel {
                color: #495057;
                font-size: 11px;
            }

            QLabel#cvCountLabel {
                color: #6c757d;
                font-size: 11px;
            }

            /* Status styles for different states */
            QLabel#statusLabel[state="info"] {
                color: #0d6efd;
            }

            QLabel#statusLabel[state="success"] {
                color: #198754;
            }

            QLabel#statusLabel[state="warning"] {
                color: #fd7e14;
            }

            QLabel#statusLabel[state="error"] {
```

```
                color: #dc3545;
            }
        """)

    def create_separator(self):
        """Create a horizontal separator line"""
        line = QFrame()
        line.setFrameShape(QFrame.Shape.HLine)
        line.setFrameShadow(QFrame.Shadow.Sunken)
        return line

    def on_search(self, search_params):
        """Handle search button click with progress dialog"""
        keywords = search_params['keywords']
        algorithm = search_params['algorithm']
        top_k = search_params['top_matches']
        case_sensitive = search_params['case_sensitive']

        # Update UI with visual feedback
        self.search_section.set_search_enabled(False)

        # Animated status update
        self.status_label.setStyleSheet("""
            QLabel#statusLabel {
                color: #0066cc;
                font-size: 12px;
                padding: 8px;
                background-color: #e6f2ff;
                border-radius: 6px;
                border: 1px solid #99ccff;
                margin-top: 5px;
            }
        """)
        self.update_status("Searching CVs... Please wait", "info")

        # Create a more attractive progress dialog
        from PyQt6.QtWidgets import QProgressDialog
        self.search_progress = QProgressDialog("Searching CVs...",
"Cancel", 0, 100, self)
        self.search_progress.setWindowTitle(f"Searching for:
{keywords}")
        self.search_progress.setMinimumDuration(0)

self.search_progress.setWindowModality(Qt.WindowModality.WindowMod
```

```
al)
        self.search_progress.setStyleSheet("""
            QProgressDialog {
                background-color: white;
                border-radius: 10px;
            }
            QProgressBar {
                border: 1px solid #cccccc;
                border-radius: 5px;
                text-align: center;
                color: #333333;
                background-color: #f5f5f5;

            }
            QProgressBar::chunk {
                background-color: #4c72b0;
                border-radius: 5px;
            }
        """)

        # Create and start search thread using the imported class
        self.search_thread = SearchThread(
            self.service, keywords, algorithm, top_k,
case_sensitive
        )

self.search_thread.progress.connect(self.search_progress.setValue)

self.search_thread.results_ready.connect(self.on_search_completed)
        self.search_thread.start()

    def on_search_completed(self, total, elapsed, results):
        """Handle completion of search"""
        self.search_section.set_search_enabled(True)

        # Close the progress dialog
        if hasattr(self, 'search_progress'):
            self.search_progress.close()

        # Display results
        self.result_section.display_results(
            results, elapsed,
            {'keywords': self.search_section.get_keywords()}
        )
```

```
        # Update status
        if results:
            self.update_status(
                f"Found {len(results)} matches among {total} CVs
in {elapsed:.2f} seconds.",
                "success"
            )
        else:
            self.update_status(
                f"No matches found among {total} CVs in
{elapsed:.2f} seconds.",
                "warning"
            )

    def update_status(self, message, status_type="info"):
        """Update status message with proper styling"""
        # Set the status message
        self.status_label.setText(message)

        # Apply appropriate status style
        self.status_label.setProperty("state", status_type)

        # Force style refresh (needed for dynamic property
changes)
        self.status_label.style().unpolish(self.status_label)
        self.status_label.style().polish(self.status_label)

    def update_cv_count(self, count, preprocessed=False):
        """Update CV count in status bar"""
        if preprocessed:
            self.cv_count_label.setText(f"{count} CVs
preprocessed")
        else:
            self.cv_count_label.setText(f"{count} CVs available")

    def preprocess_cvs(self):
        """Preprocess CVs in background thread with progress
dialog"""

        # Create progress dialog
        from PyQt6.QtWidgets import QProgressDialog
        self.preprocess_progress = QProgressDialog("Preprocessing
CVs...", "Cancel", 0, self.cv_count, self)
```

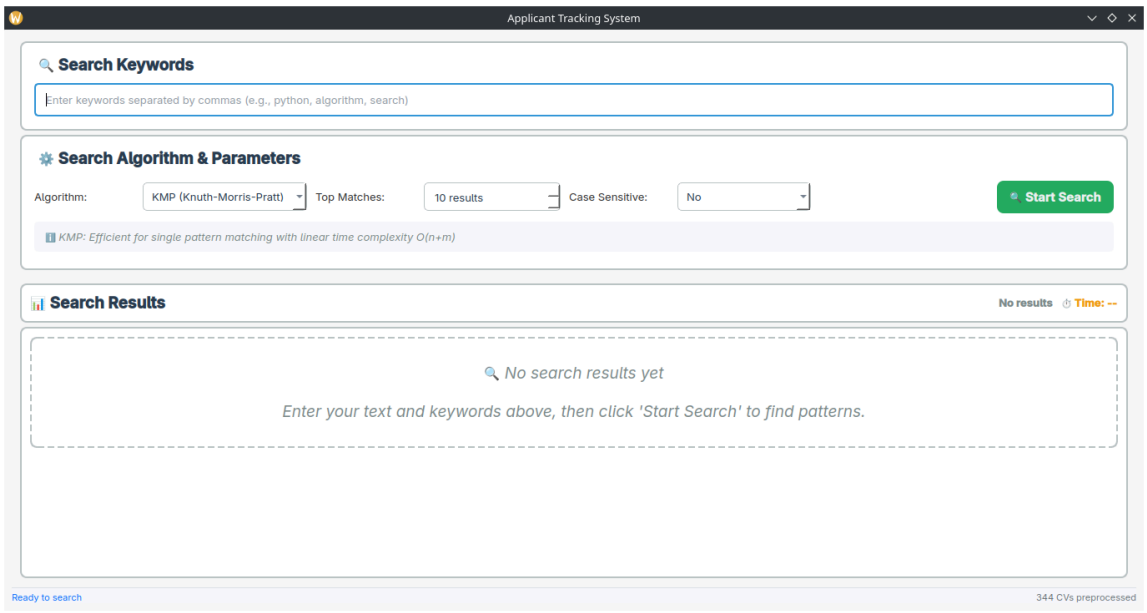
```
        self.preprocess_progress.setWindowTitle("CV
Preprocessing")
        self.preprocess_progress.setMinimumDuration(0)  # Show
immediately

self.preprocess_progress.setWindowModality(Qt.WindowModality.Windo
wModal)

        # Create and start thread using the imported class
        self.preprocess_thread = PreprocessThread(self.service)

self.preprocess_thread.progress.connect(self.preprocess_progress.s
etValue)

self.preprocess_thread.finished.connect(self.on_preprocessing_fini
shed)
        self.preprocess_thread.start()

    def on_preprocessing_finished(self, count, elapsed):
        # Close the progress dialog
        if hasattr(self, 'preprocess_progress'):
            self.preprocess_progress.close()

        self.update_status(f"Ready to search", "info")
        self.update_cv_count(count, True)

if __name__ == "__main__":
    try:
        # Set application-wide style
        import sys
        from PyQt6.QtGui import QPalette, QColor
        from PyQt6.QtWidgets import QApplication

        app = QApplication(sys.argv)

        # Set app-wide font
        from PyQt6.QtGui import QFont
        font = QFont("Segoe UI", 10)  # Modern font
        app.setFont(font)

        # Create main window
        main_win = MainWindow()
        main_win.show()
        sys.exit(app.exec())
```

```
    except Exception as e:
        print(f"Application error: {str(e)}")
        traceback.print_exc()
        # Show error as message box
        if 'app' in locals():
            QMessageBox.critical(None, "Critical Error",
                                 f"The application could not start:
{str(e)}")
            sys.exit(1)
```

B. Penjelasan Tata Cara Penggunaan Program

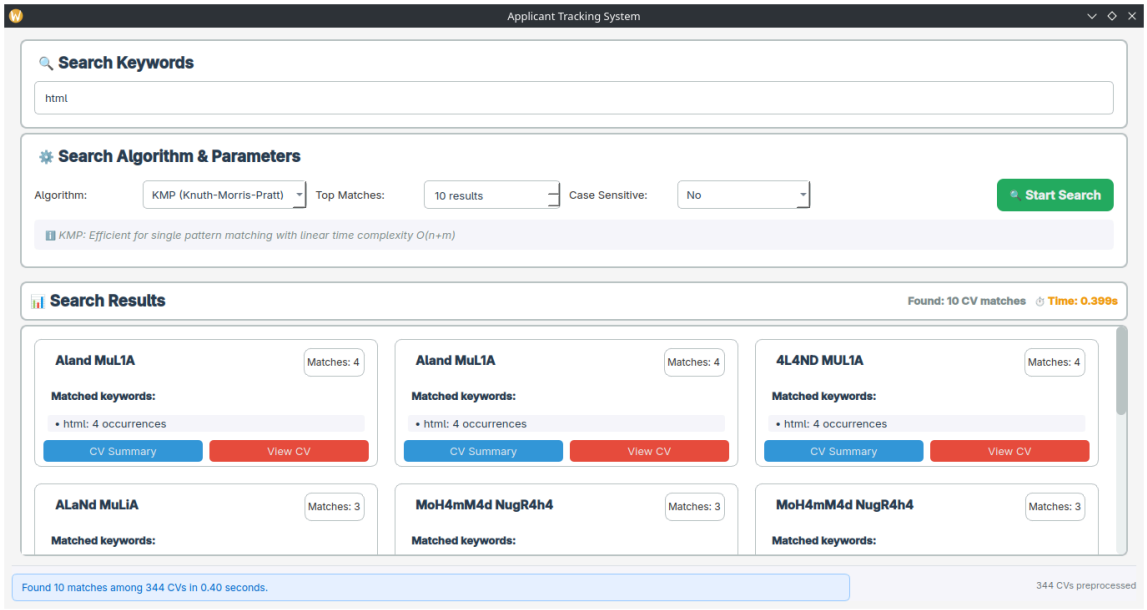
Pengguna bisa mengakses aplikasi dengan menjalankan aplikasi dalam uv. Langkah umum untuk membuka dan menjalankan aplikasi dengan uv adalah sebagai berikut:

1. Lakukan uv pip install -r requirements.txt
2. Lakukan `uv run main.py`

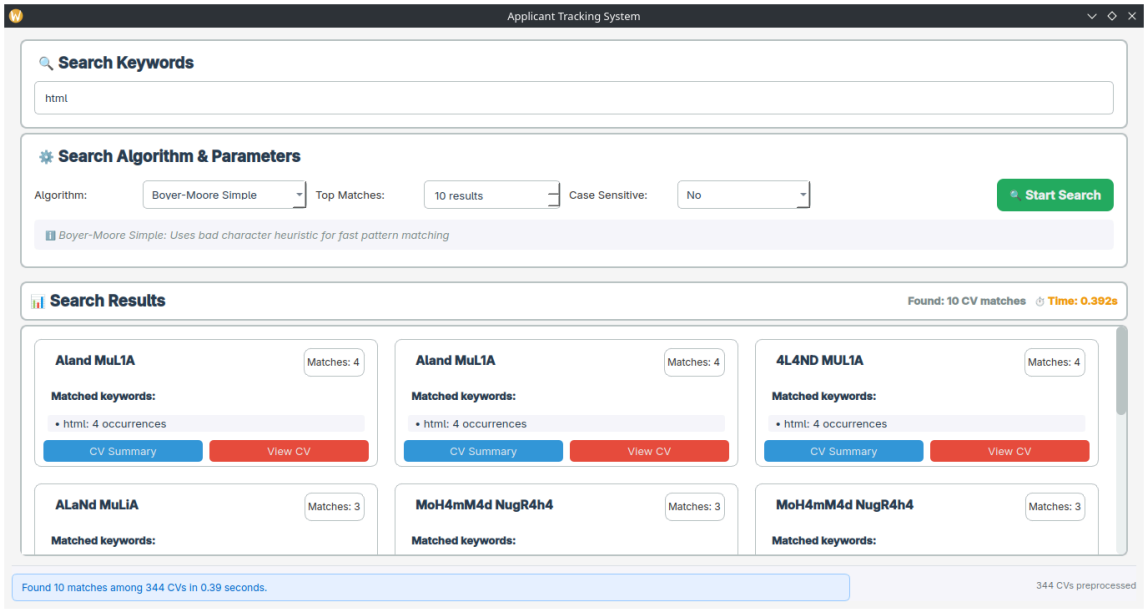


Gambar 11. Tampilan antarmuka halaman utama.
(sumber: Arsip kelompok)

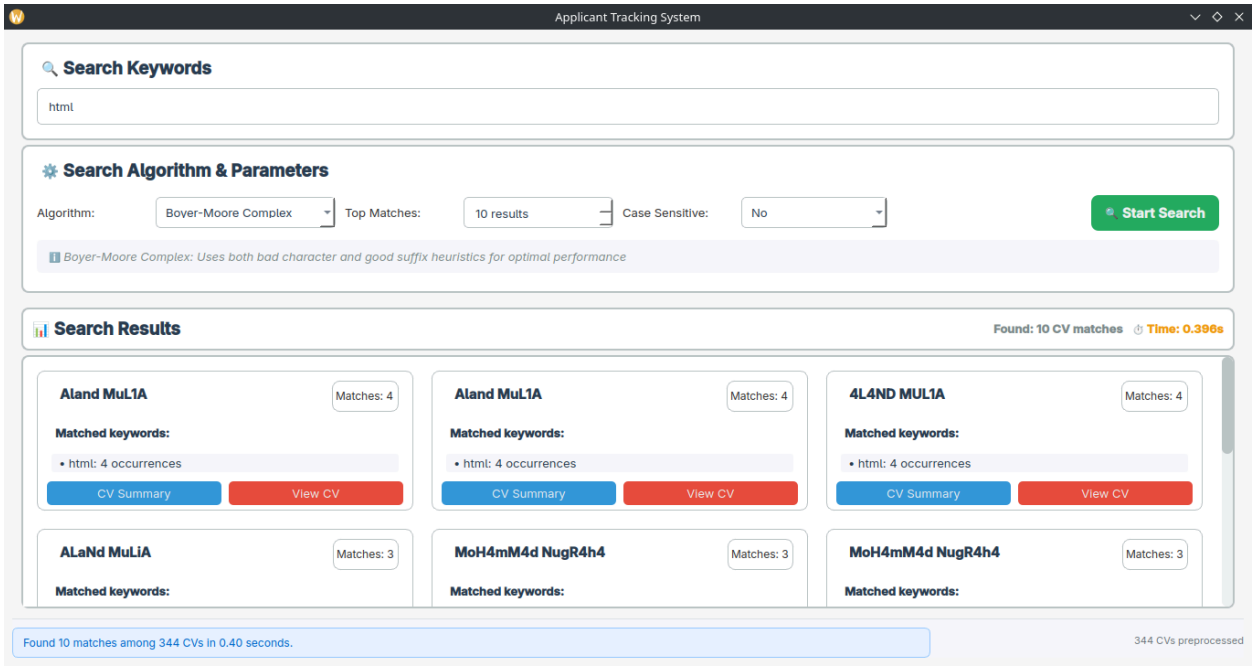
C. Hasil Pengujian



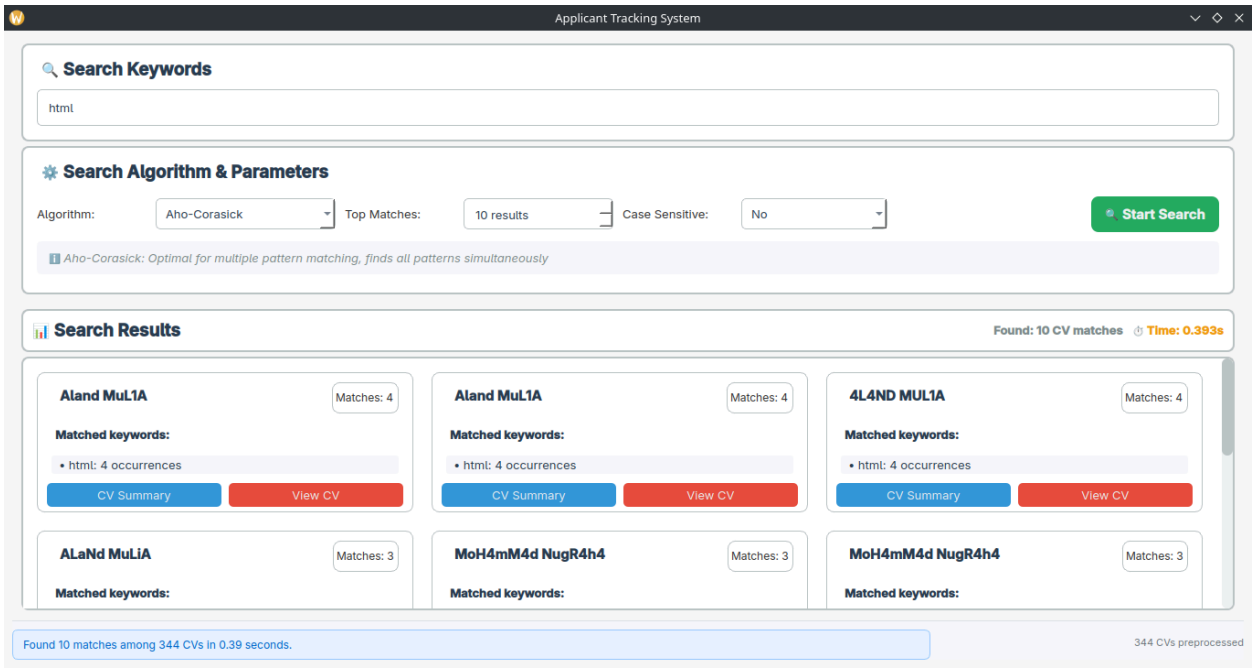
Gambar 12. Pengujian algoritma KMP dalam pencarian kata kunci 'html (Single Keyword)



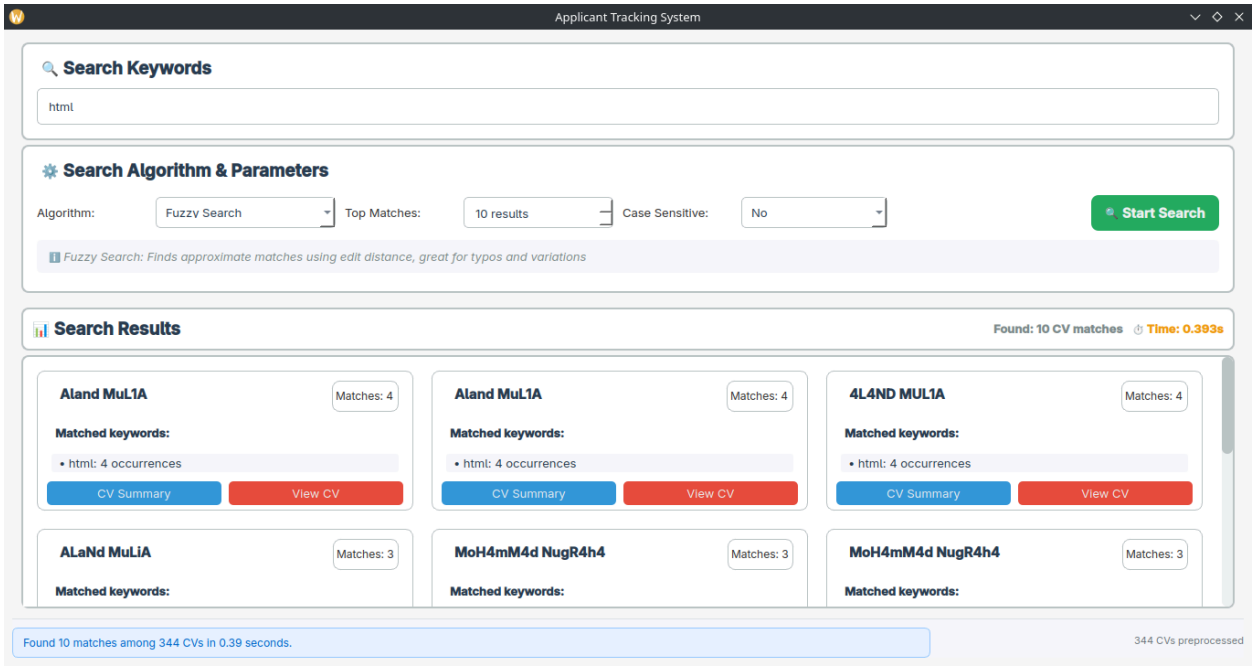
Gambar 13. Pengujian algoritma BM Simple dalam pencarian kata kunci 'html (Single Keyword)



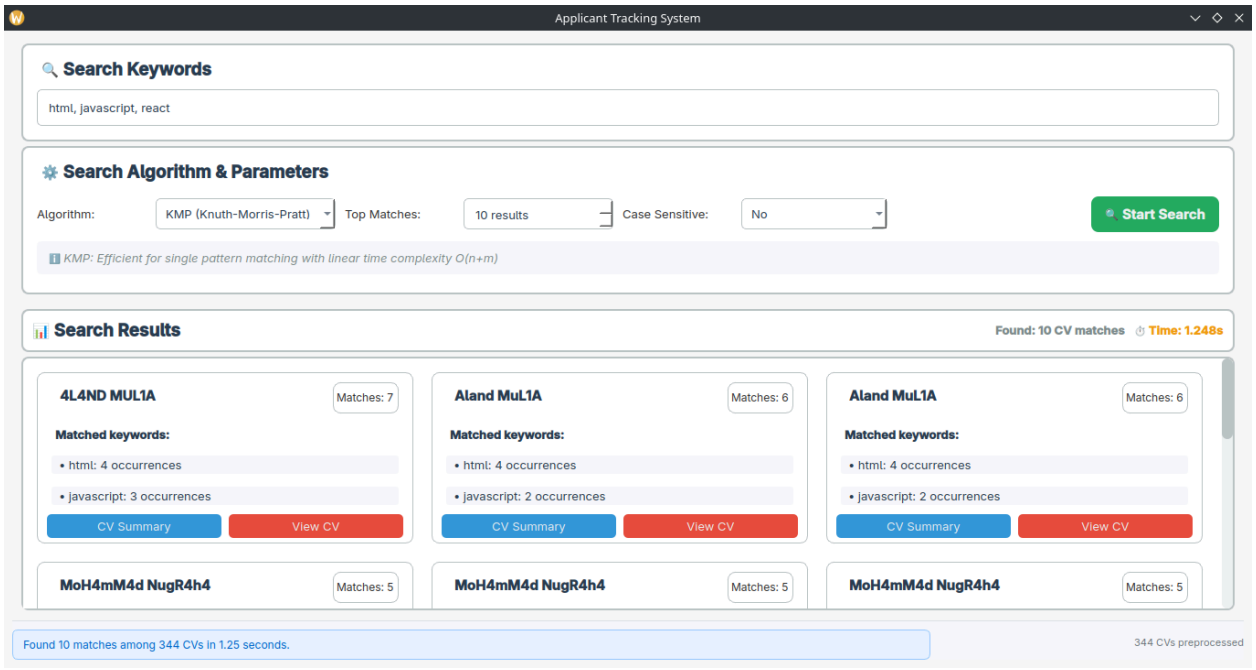
Gambar 14. Pengujian algoritma BM Complex dalam pencarian kata kunci 'html (Single Keyword)



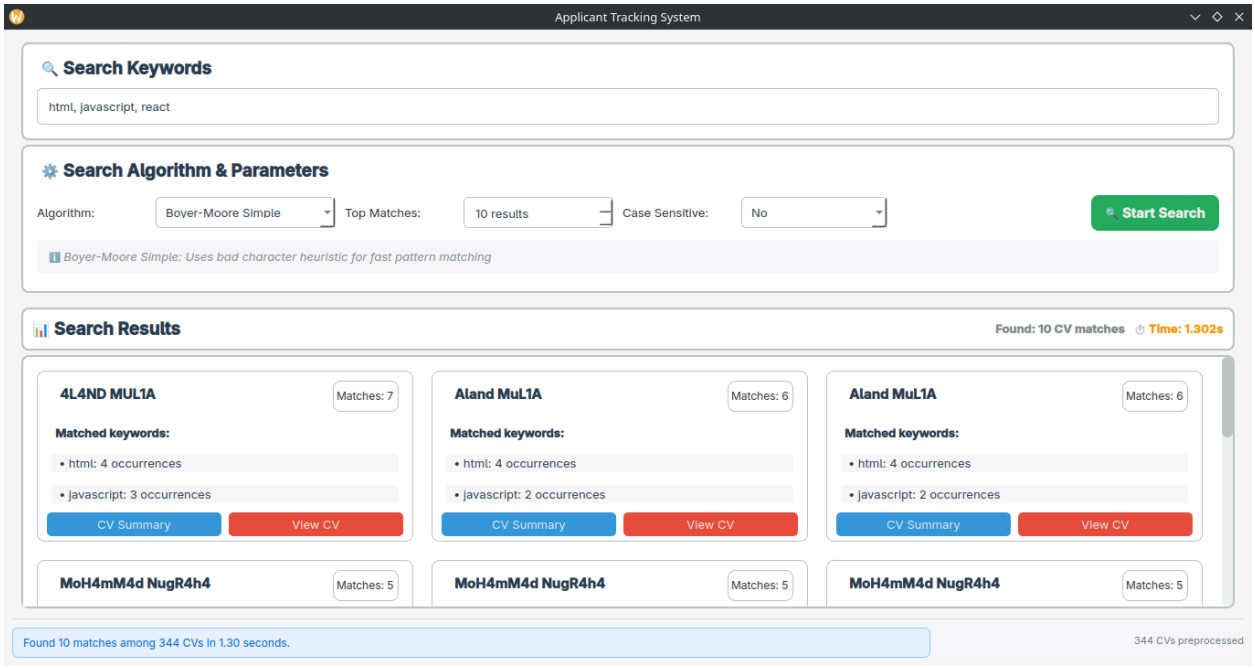
Gambar 15. Pengujian algoritma AHO Corasick dalam pencarian kata kunci 'html (Single Keyword)



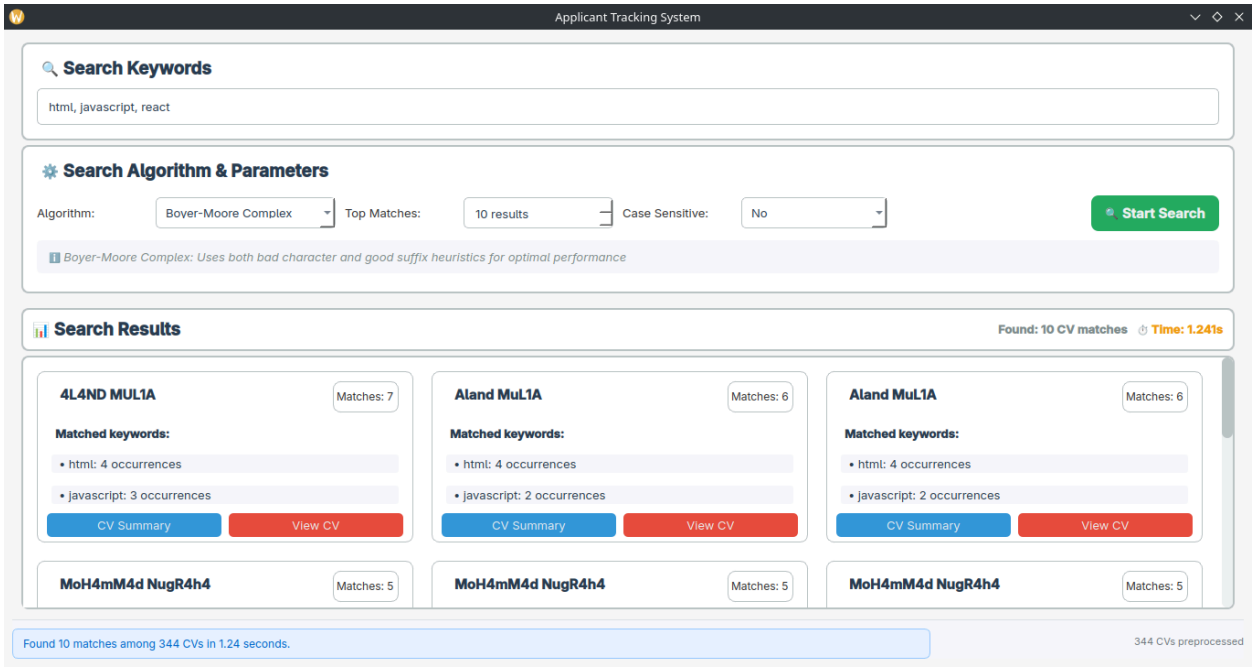
Gambar 16. Pengujian algoritma Fuzzy Search dalam pencarian kata kunci 'html (Single Keyword)



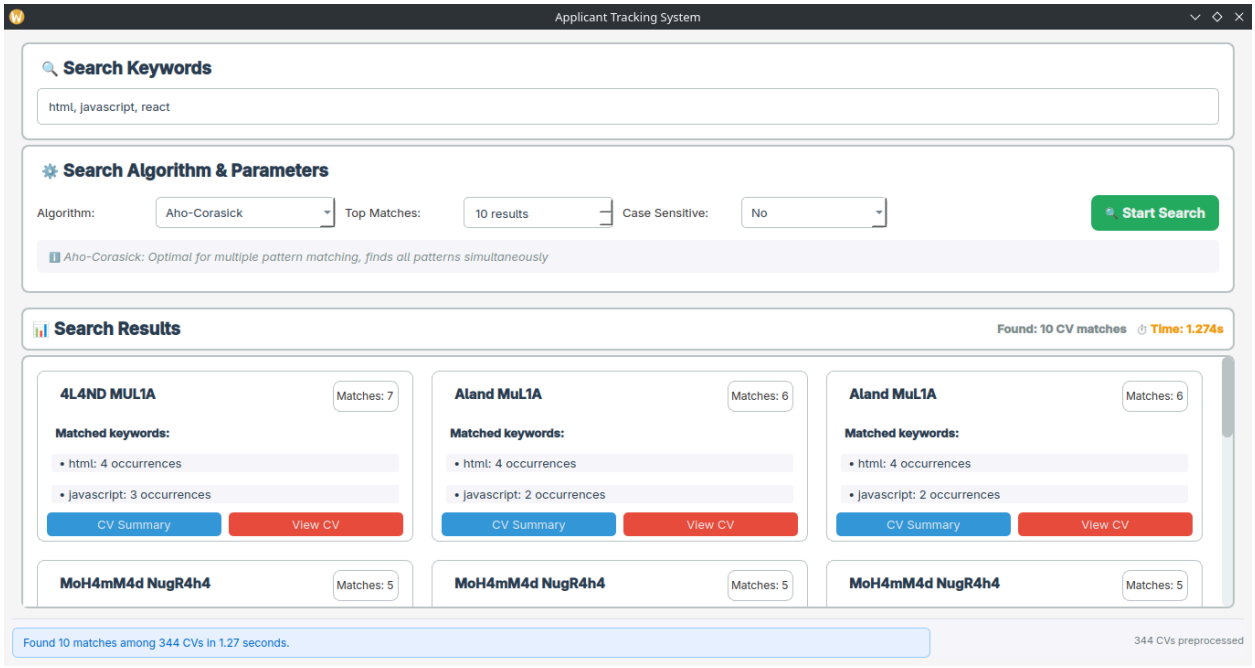
Gambar 17. Pengujian algoritma KMP dalam pencarian kata kunci 'html, javascript, react' (Multiple Keywords)



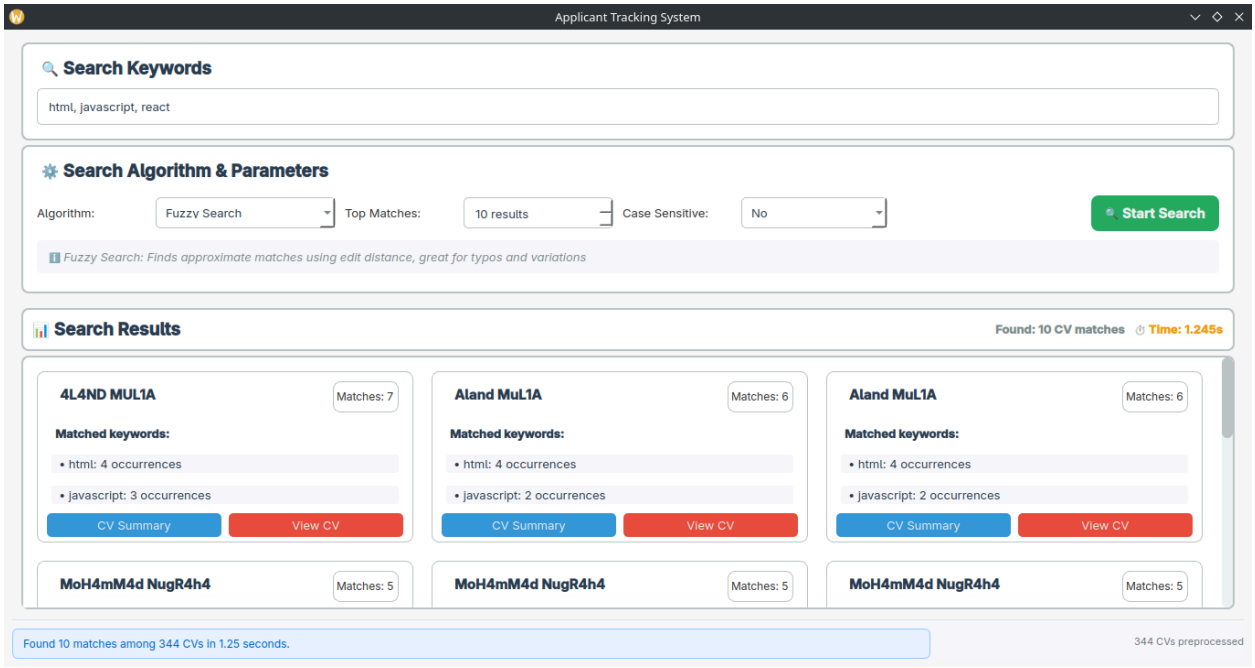
Gambar 18. Pengujian algoritma BM Simple dalam pencarian kata kunci 'html, javascript, react' (Multiple Keywords)



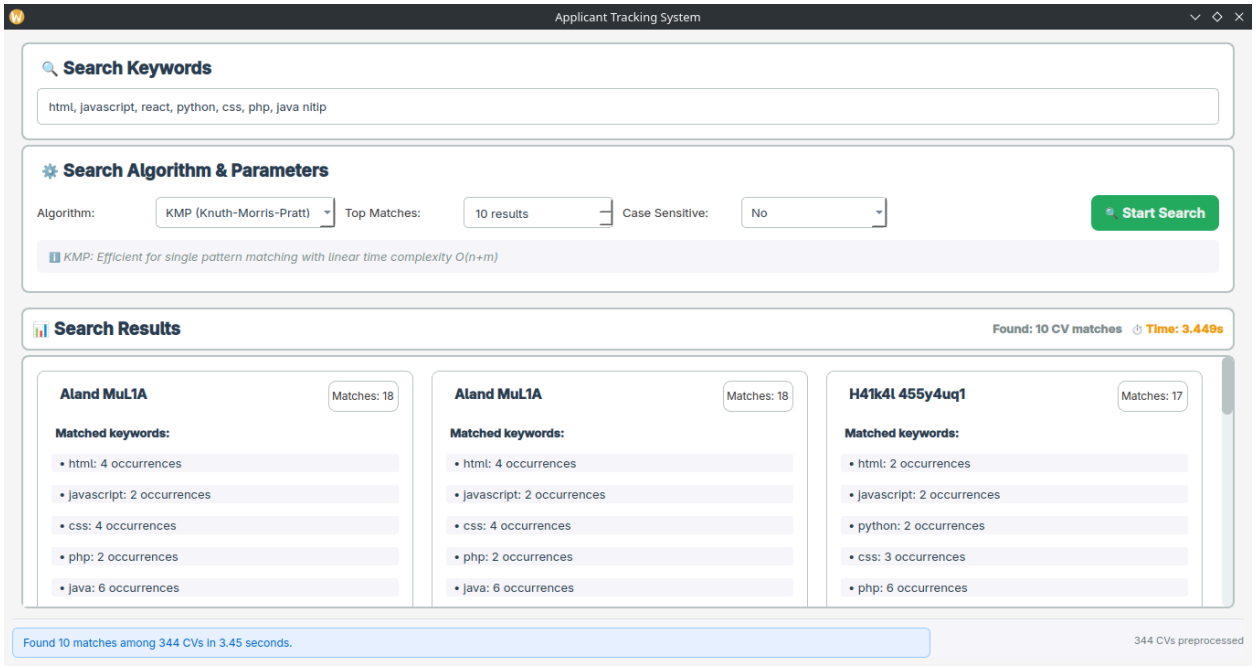
Gambar 19. Pengujian algoritma BM Complex dalam pencarian kata kunci 'html, javascript, react' (Multiple Keywords)



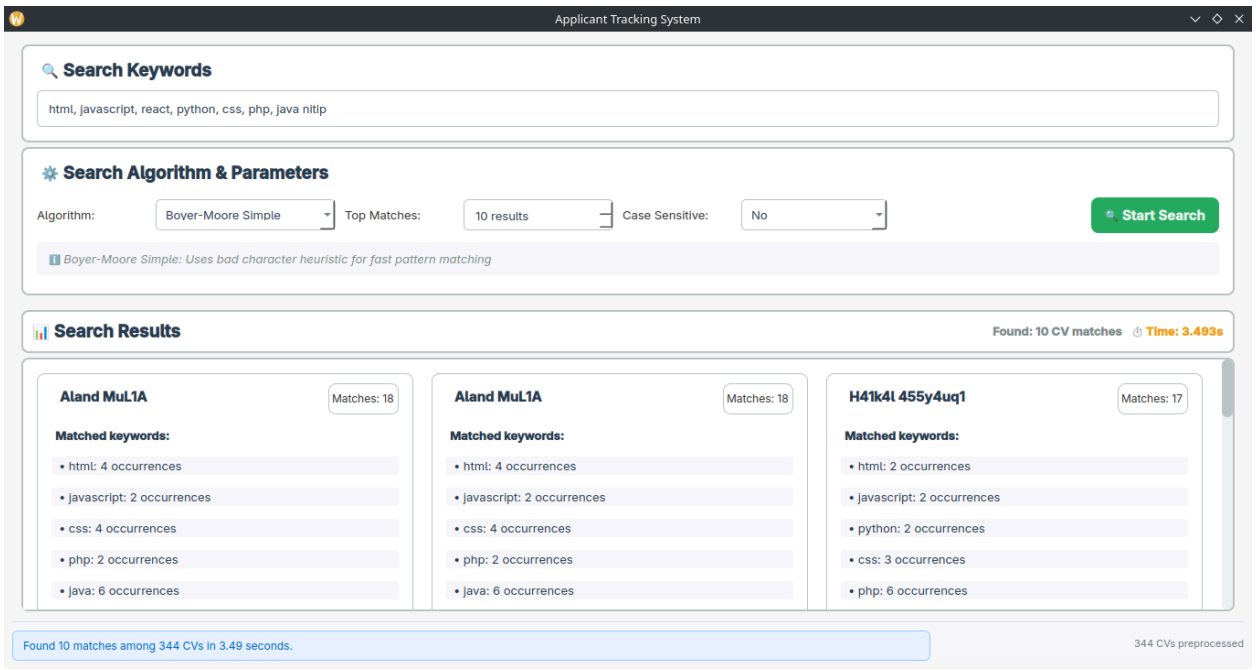
Gambar 20. *Pengujian algoritma AHO Corasick dalam pencarian kata kunci 'html, javascript, react' (Multiple Keywords)*



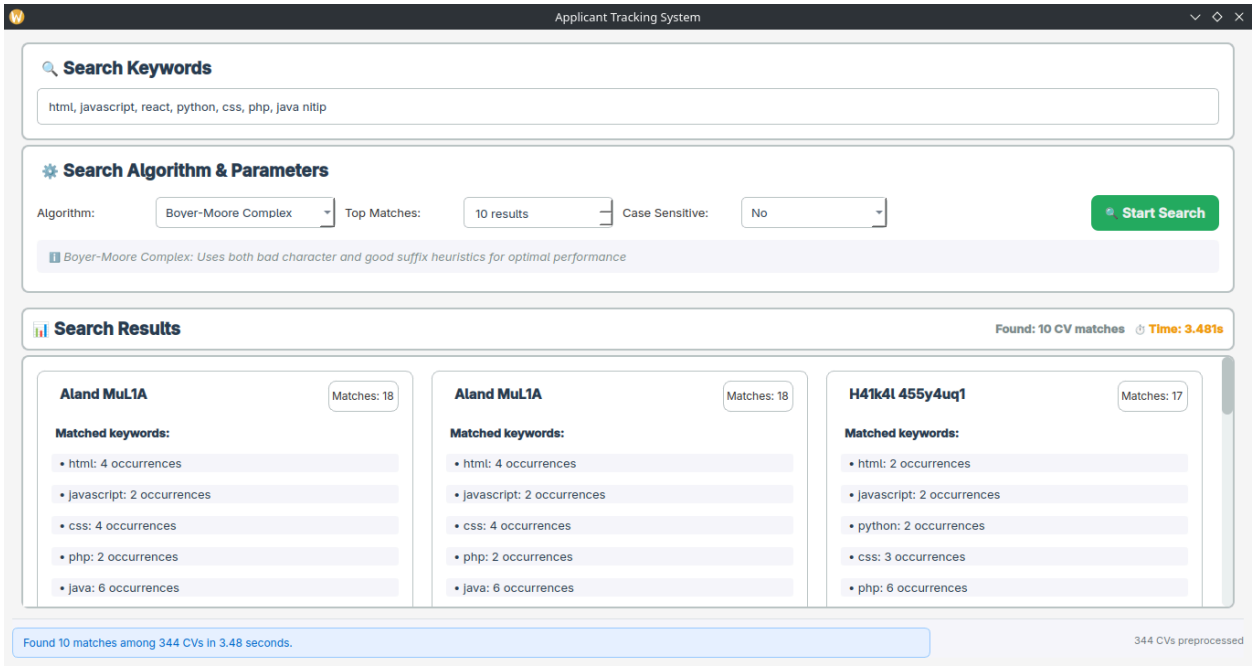
Gambar 21. Pengujian algoritma Fuzzy Search dalam pencarian kata kunci 'html, javascript, react' (Multiple Keywords)



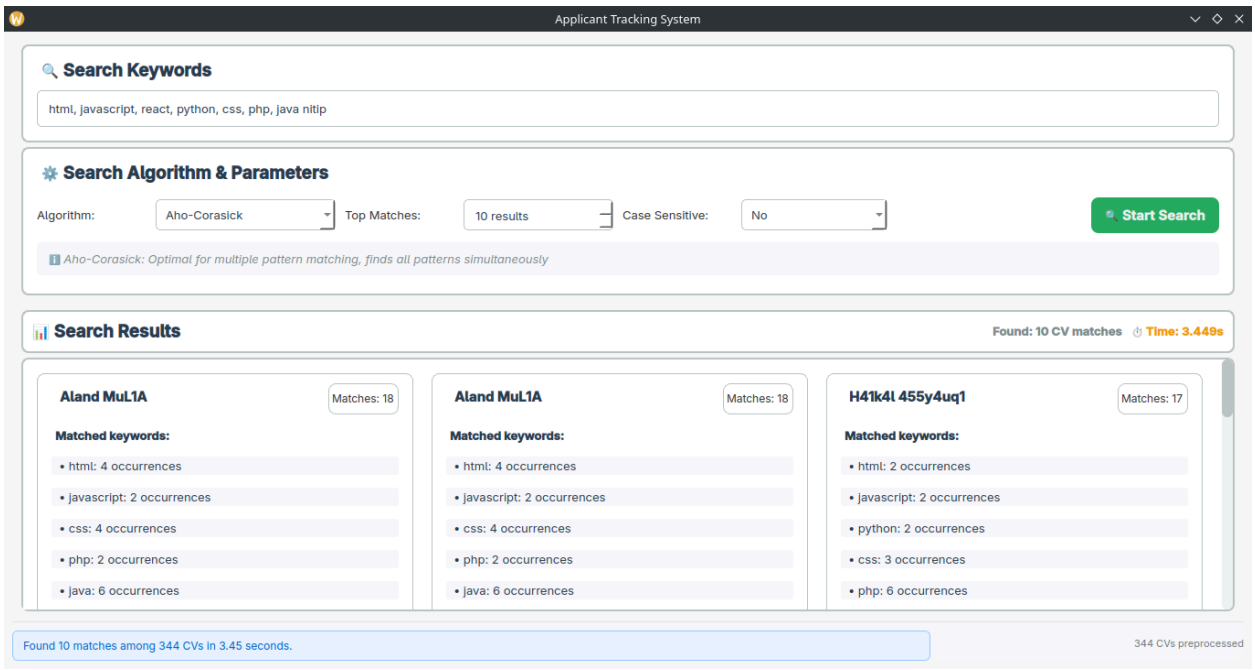
Gambar 22. Pengujian algoritma KMP dalam pencarian kata kunci 'html, javascript, react, python, css, php, java nitip' (Multiple Keywords)



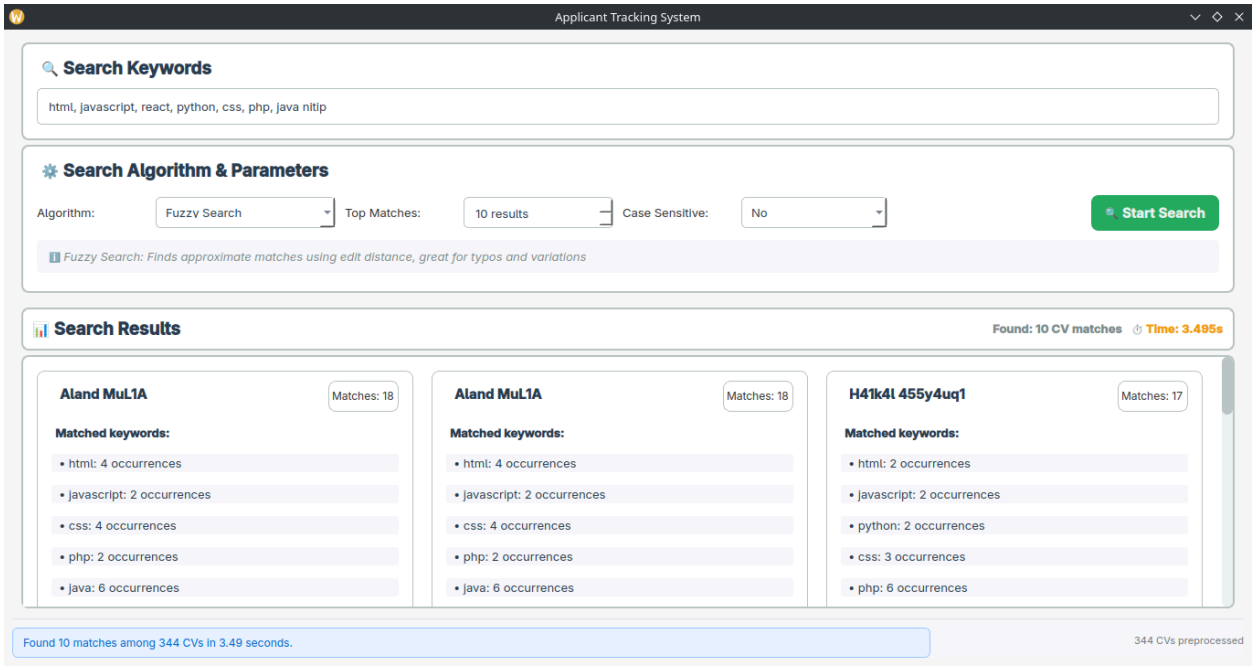
Gambar 23. Pengujian algoritma BM Simple dalam pencarian kata kunci 'html, javascript, react, python, css, php, java nitip' (Multiple Keywords)



Gambar 24. Pengujian algoritma BM Complex dalam pencarian kata kunci 'html, javascript, react, python, css, php, java nitip' (Multiple Keywords)



Gambar 25. Pengujian algoritma AHO Corasick dalam pencarian kata kunci 'html, javascript, react, python, css, php, java nitip' (Multiple Keywords)



Gambar 26. Pengujian algoritma Fuzzy Search dalam pencarian kata kunci 'html, javascript, react, python, css, php, java nitip' (Multiple Keywords)

D. Analisis Hasil

Berdasarkan waktu yang dibutuhkan dalam proses pencarian kata kunci, didapatkan bahwa KMP dan AHO Corasick relatif membutuhkan waktu yang lebih singkat dan konsisten dibandingkan algoritma lainnya, terutama saat jumlah kata kunci meningkat. Hal ini sangat terlihat ketika kita membandingkan performa pada 7 kata kunci, di mana kedua algoritma tersebut mencatatkan waktu tercepat (3449 ms). Sementara itu, BM-Simple menunjukkan peningkatan waktu yang paling signifikan dari 1 ke 3 kata kunci, mengindikasikan skalabilitas yang kurang optimal dibandingkan BM-Complex atau algoritma lainnya pada perangkat low-end ini.

Penyebab utamanya adalah karena AHO Corasick dirancang khusus untuk pencocokan multi-pola atau banyak kata kunci secara efisien. Secara internal, Aho-Corasick membangun sebuah automata (mesin keadaan hingga) dari seluruh kumpulan kata kunci yang akan dicari. Ini memungkinkannya untuk memindai teks target (misalnya, CV) hanya dalam satu kali pass, secara simultan mencari semua kata kunci yang telah di-pre-process. Hal ini menjadikannya sangat ringan secara waktu eksekusi saat menghadapi banyak pola sekaligus. Sementara itu, KMP juga sangat efisien untuk pencocokan exact dengan kompleksitas linier, meminimalkan perbandingan yang tidak perlu melalui fungsi prefixnya, yang membuatnya tetap cepat bahkan saat jumlah kata kunci bertambah.

Namun, dari sisi kemampuan pencarian, hasil pengujian menunjukkan bahwa Fuzzy Search mampu mempertahankan performa yang sangat kompetitif (bahkan lebih cepat dari KMP dan Aho-Corasick pada 3 kata kunci) meskipun dengan kompleksitas tambahan untuk melakukan pencocokan perkiraan. Hal ini terjadi karena Fuzzy Search tidak memerlukan pencocokan karakter yang sama persis, melainkan dapat menemukan kesamaan meskipun ada salah ketik atau variasi. Ini berarti ia dapat "menangkap" lebih banyak kandidat yang relevan yang mungkin saja terlewat oleh algoritma pencocokan exact karena typo atau perbedaan ejaan. Sebaliknya, algoritma seperti KMP dan Boyer-Moore sangat bergantung pada pencocokan karakter yang tepat, sehingga bisa berhenti terlalu dini atau melewatkan hasil jika ada sedikit perbedaan.

Dengan demikian, terdapat trade-off antara kecepatan/akurasi exact dan keluasan eksplorasi: KMP dan AHO Corasick lebih cepat dan efisien untuk pencocokan kata kunci yang persis sama, sedangkan Fuzzy Search lebih komprehensif dan toleran terhadap kesalahan, namun dengan potensi overhead komputasi yang sedikit lebih tinggi, terutama pada kasus fuzziness yang ekstrem atau jumlah kata kunci yang sangat banyak. Pada pengujian ini, Fuzzy Search menunjukkan bahwa fleksibilitas tersebut dapat dicapai tanpa penalti performa yang signifikan bahkan pada perangkat low-end, menjadikannya pilihan berharga untuk ATS yang membutuhkan pencarian yang lebih "memaafkan".

BAB V
KESIMPULAN, SARAN, DAN REFLEKSI

A. Kesimpulan

Pada Tugas Besar 3 ini, kami membangun sebuah aplikasi GUI yang mampu mencari dan mencocokkan pola teks dalam sekumpulan CV berformat PDF dengan kecepatan dan akurasi tinggi. Aplikasi ini mendukung berbagai metode exact matching—termasuk KMP, Boyer-Moore, dan Aho-Corasick—serta dilengkapi mekanisme fuzzy matching berbasis Levenshtein Distance untuk menangani variasi ejaan atau kesalahan ketik.

Saat dijalankan, semua CV diunggah dan diproses otomatis: konten PDF diubah menjadi string untuk pencocokan, kemudian informasi terstruktur diekstrak untuk tampilan ringkasan. Hasil pencarian diurutkan menurut tingkat relevansi, dan pengguna dapat langsung membuka CV asli atau melihat ringkasan profilnya. Fitur tambahan memungkinkan penambahan CV baru dengan penandaan kategori pekerjaan.

Dengan penyelesaian proyek ini, kami tidak hanya berhasil memenuhi seluruh kebutuhan fungsional, tetapi juga memperoleh pemahaman yang lebih mendalam tentang algoritma string matching dan langkah-langkah pengembangan antarmuka GUI menggunakan Python.

B. Saran

Untuk pengembangan lanjutan, berikut beberapa rekomendasi yang bisa dipertimbangkan:

1. Mengintegrasikan algoritma tambahan untuk pattern matching, baik pada mode exact maupun fuzzy, agar cakupan pencocokan menjadi lebih luas.
2. Menyediakan modul visualisasi yang memperlihatkan langkah-langkah algoritma dalam proses pencocokan pola, sehingga pengguna dapat memahami mekanisme kerja secara interaktif.
3. Menambahkan opsi penyaringan hasil berdasarkan kriteria seperti bidang pekerjaan, riwayat jabatan, atau latar belakang pendidikan untuk membantu pengguna menemukan kandidat yang lebih tepat.

C. Refleksi

Pelaksanaan Tugas Besar 3 menghadirkan tantangan nyata dalam menerapkan algoritma string matching pada data dokumen yang tidak seragam. Setiap berkas CV memiliki variasi format, gaya penulisan, dan struktur informasi, sehingga kami perlu menyesuaikan pola regex dan praproses teks agar ekstraksi data—seperti nama, kontak, dan riwayat pekerjaan—dapat berjalan dengan andal. Proses ini memperkuat pemahaman kami tentang pentingnya tahap pembersihan dan normalisasi data sebelum algoritma pencocokan dijalankan.

Selanjutnya, integrasi beberapa algoritma exact matching (KMP, Boyer-Moore, Aho-Corasick) sekaligus mekanisme fuzzy matching berbasis Levenshtein Distance mengajarkan kami perbedaan performa dan trade-off antara kecepatan versus toleransi kesalahan ketik. Kami belajar bahwa pemilihan ambang kemiripan fuzzy yang tepat sangat krusial agar sistem mampu menangkap kesalahan kecil tanpa menghasilkan terlalu banyak "false positive." Pengukuran waktu eksekusi di setiap mode matching juga memberikan wawasan langsung mengenai efisiensi masing-masing algoritma pada skala data yang berbeda.

Dari sisi antarmuka pengguna, membangun aplikasi GUI dengan Tkinter menantang kami untuk merancang alur interaksi yang intuitif—dari pemilihan algoritma hingga tampilan ringkasan hasil pencarian. Kami belajar pentingnya umpan balik visual, seperti indikator progres dan notifikasi kesalahan, agar pengguna dapat memahami status proses secara real-time. Keseluruhan pengalaman ini tidak hanya meningkatkan keterampilan teknis dalam pemrograman Python dan algoritma, tetapi juga memupuk kemampuan desain perangkat lunak yang fokus pada kebutuhan dan kenyamanan pengguna.

LAMPIRAN

Tabel Implementasi Program:

No	Poin	Ya	Tidak
1	Aplikasi dapat dijalankan.	✓	
2	Aplikasi menggunakan basis data berbasis SQL dan berjalan dengan lancar.	✓	
3	Aplikasi dapat mengekstrak informasi penting menggunakan Regular Expression (Regex).	✓	
4	Algoritma *Knuth-Morris-Pratt (KMP)* dan *Boyer-Moore (BM)* dapat menemukan kata kunci dengan benar.	✓	
5	Algoritma Levenshtein Distance dapat mengukur kemiripan kata kunci dengan benar.	✓	
6	Aplikasi dapat menampilkan *summary CV applicant*.	✓	
7	Aplikasi dapat menampilkan *CV applicant* secara keseluruhan.	✓	
8	Membuat laporan sesuai dengan spesifikasi.	✓	
9	Membuat bonus enkripsi data profil *applicant*.	✓	
10	Membuat bonus algoritma Aho-Corasick.	✓	
11	Membuat bonus video dan diunggah pada Youtube.		✓

Pembagian Tugas :

NIM	Nama	Tugas
13523123	Rhio Bimo P S	Laporan, Database, Service, Testing
13523146	Rafael Marchell D	Laporan, GUI, Search Algorithm, Encryption
13523147	Frederiko Eldad M	Laporan, GUI, Database Migration, Service, Finishing

Link Github Repository :

https://github.com/susTuna/Tubes3_Qtpy_coba/

Link Video : Sayangnya Tidak Membuat, Rhio Sedang Terlalu Sibuk :v

Find the secret!! : AuVjtPcxlfE

DAFTAR PUSTAKA

Munir, Rinaldi. 2025. BFS-DFS-Bag1. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2023-2024/BFS-DFS-2021-Bag1-2024.pdf . Diakses pada 1 Maret 2025.