NORDAKADEMIE
HOCHSCHULE DER WIRTSCHAFT

File Explorer

Ein Projekt für den Informaticup

Autoren:
Brandt Marco (Matrikel-Nr. 13779)
Lauritz Wiebusch (Matrikel-Nr. 13771)
Sören Panten(Matrikel-Nr. 13766)
Schatz Daniel (Matrikel-Nr. 13796)

Projekt: Informaticup
Abgabedatum: 30.Juni.2025

Inhaltsverzeichnis

1 Einleitung **1**

2 Datenverarbeitung **2**

2.1 Datenaustausch Backend - Frontend 2

2.2 Nutzerdaten 3

3 Performanceoptimierung **4**

4 Search-Algorithmus **5**

4.1 Architektonischer Aufbau 5

5 Design Philosophie und Inspiration **8**

5.1 Die Wahl des macOS Finders als Designgrundlage 8

5.2 Integration bewährter Konzepte aus verschiedenenSystemen 8

5.3 Nutzerorientierte Anpassungsmöglichkeiten 8

5.4 Technische Umsetzung und eigenständige Entwicklung 8

6 Projektplanung und Kommunikation **9**

6.1 Zusammenarbeit mit Community und dem Ersteller 9

6.2 Kommunikation im Team 9

Anhang **II**

Literaturverzeichnis **X**

Abbildungsverzeichnis **X**

1 Einleitung

Die Entwicklung einer eigenen Software mit genau jenen Funktionen, welche man sich im Alltag immer gewünscht hat, bietet sowohl kreativ als auch praktisch spannende Möglichkeiten. Insbesondere im Bereich der Dateiverwaltung, die täglich genutzt wird, bieten Standard-File-Explorer, wie der Windows-Explorer oder auf macOS der Finder, meist nur das nötigste: Dateien anzeigen und einfache Bedienoberflächen bereitstellen. Dabei bleiben viele Anforderungen des modernen Nutzers meist unbeachtet.

Insbesondere deshalb war es für uns interessant einen File Explorer nach unseren eigenen Vorstellungen zu entwickeln. Dieses Projekt gab uns nicht nur die Freiheit, lange gewünschte Funktionen einzubauen, sondern ermöglichte auch auszuprobieren, welche Ansätze tatsächlich hilfreich sind: Wie gestaltet man eine Benutzeroberfläche, die Menschen wirklich unterstützt, entlastet und informativ ist? Welche Strukturen und Automatismen sorgen im Alltag für echte Effizienz?

Die Idee zu diesem Projekt entstand durch ein YouTube-Video von Conaticus [1], in dem er seinen Ansatz für einen schnellen File Explorer vorstellt. Da wir persönlichen Kontakt mit ihm aufnehmen konnten und er von unserer Beteiligung begeistert war, entschieden wir uns für dieses Projekt. Schon beim ersten Blick auf das Konzept hatten wir konkrete Vorstellungen für Erweiterungen, da auch wir selbst bisher viele Funktionen bei existierenden Explorern vermisst hatten. Außerdem schildert Conaticus in einem weiteren Video ausführlich die Herausforderungen rund um die Suchfunktion [2], was wir ebenfalls als interessante Aufgabe ansahen.

Technisch basiert unser Projekt auf dem Framework Tauri [5], das eine klare Trennung der Anwendung ermöglicht: Einerseits gibt es ein Frontend, welches mit React entwickelt wurde und andererseits die Verarbeitungslogik, die in Rust implementiert ist. Zum Zeitpunkt unserer Projektstartes war das Frontend noch nicht vollständig fertiggestellt, da der ursprüngliche Fokus vor allem auf Funktionalität und weniger auf Design und Nutzerfreundlichkeit lag. Im Austausch mit Conaticus bestätigte er uns ebenfalls Interesse an einer attraktiven Benutzeroberfläche. Aus diesem Grund haben wir uns entschieden, das Frontend mit React neu und ansprechend zu gestalten.

2 Datenverarbeitung

Die Verarbeitung und Speicherung von Daten bildet das Herzstück jeder Software, denn sie entscheidet maßgeblich darüber, wie einfach und effizient eine Anwendung genutzt werden kann. Insbesondere bei Anwendungen, die eng mit der Dateiverwaltung verbunden sind, spielt der schnelle und problemlose Datenaustausch zwischen Frontend und Backend eine wesentliche Rolle.

2.1 Datenaustausch Backend - Frontend

Die Datenverarbeitung findet hauptsächlich im Backend statt, das im Framework Tauri [5] durch in Rust geschriebenen Code realisiert wird. Das Frontend ruft die Methoden auf, deren Daten anschließend über die asynchrone Runtime übermittelt werden. Dies erfolgt meist über JSON oder auch mit primitiven Datentypen. Wenn jedoch benutzerdefinierte Objekte übergeben werden sollen, müssen die Felder im Frontend dieselben Namen tragen und identisch strukturiert sein. Für dieses Projekt haben wir uns entschieden, sämtliche Daten ausschließlich über JSON bereitzustellen. Diese einheitliche Lösung verhindert Probleme im Zusammenhang mit unterschiedlichen Datentypen und fehleranfälligen Konvertierungen.

Die Methoden, die vom Frontend aus aufgerufen werden können, werden mit dem Attribut `#[tauri::command]` markiert und dem Application-Builder hinzugefügt. Jeder Endpoint muss explizit registriert werden. Danach können die Methoden über ihren Namen im Frontend aufgerufen werden. Wichtig ist dabei, dass alle Methoden stand asynchron sind.

```
#[tauri::command]
pub async fn open_file(path: &str) -> Result<String, String> {
let path_obj = Path::new(path);
// Check if path exists
if !path_obj.exists() {
return Err(format!(//TODO));
}
// Check if path is a file
if !path_obj.is_file() {
return Err(format!(//TODO));
}
// Read the file
fs::read_to_string(path).map_err(|err| format!("Failed to read file: {}", err))
}
```

Eine weitere Möglichkeit des Datentransports an das Frontend bietet der sogenannte State [4]. Der Vorteil dieser Methode liegt in der zentralisierten Datenspeicherung für das Frontend, welche über die gesamte Laufzeit der Applikation hinweg bestehen bleibt. Der State kann jederzeit vom Frontend abgefragt werden. Ein Beispiel für die Verwendung des State ist im Anhang zu finden unter State Beispiel.

Im File Explorer wird unter anderem folgende Datenstruktur über einen solchen State bereitgestellt:

```
pub struct MetaData {
version: String,
abs_file_path_buf: PathBuf,
abs_file_path_for_settings_json: PathBuf,
pub abs_folder_path_buf_for_templates: PathBuf,
pub template_paths: Vec<PathBuf>,
all_volumes_with_information: Vec<VolumeInformation>,
current_running_os: String,
current_cpu_architecture: String,
user_home_dir: String
}
```

Das oben gezeigte `struct` beinhaltet grundlegende Informationen, die entweder über die Applikation selbst gespeichert oder aus dem System, auf dem das Programm läuft, ausgelesen werden. Es wird beim Start der Anwendung initialisiert und mithilfe von Tauri dem Frontend zur Verfügung gestellt.

Da das `struct` unter anderem den Typ `PathBuf` verwendet, ergeben sich bei der Umsetzung in `TypeScript` gewisse Einschränkungen. Diese Typen lassen sich dort nicht ohne Weiteres abbilden. Ein möglicher Ansatz zur Abbildung besteht im Einsatz einer äquivalenten Struktur, wobei jedoch zu beachten ist, dass diese nicht automatisch generiert werden kann.

2.2 Nutzerdaten

Die vom Nutzer erzeugten Daten werden in unserem Projekt persistent im Anwendungsverzeichnis neben der ausführbaren Binärdatei abgelegt. Konkret bedeutet das, dass die Daten dauerhaft gespeichert sind und bei jedem Neustart der Applikation unverändert wieder zur Verfügung stehen. Diese Vorgehensweise bietet für Nutzer große Vorteile, da sämtliche gespeicherten Daten jederzeit problemlos eingesehen werden können. Anders als bei versteckten oder komplex strukturierten Datenbanken ermöglicht die Speicherung in direkt lesbaren JSON-Dateien, dass Nutzer bei eventuellen Problemen oder Unklarheiten eigenständig Anpassungen vornehmen können. Dadurch steigt nicht nur die Transparenz, sondern auch die Kontrolle des Nutzers über die eigenen Daten. Besonders hilfreich ist dieser Ansatz auch bei der Fehlersuche, da Konfigurationsfehler oder inkonsistente Daten leicht entdeckt und behoben werden können. Insgesamt entsteht somit eine besonders benutzerfreundliche, flexible und robuste Datenverwaltung.

3 Performanceoptimierung

Performance-Optimierung stellt einen essenziellen Bestandteil Softwareentwicklung dar, besonders bei Anwendungen, die häufig mit großen Datenvolumina oder komplexen Berechnungen arbeiten müssen. Ziel einer solchen Optimierung ist die effiziente Nutzung von Ressourcen wie CPU-Leistung, Arbeitsspeicher und Eingabe-/Ausgabesystemen (I/O), um die Reaktionszeiten und die allgemeine Performance einer Applikation maßgeblich zu verbessern. Neben einer Steigerung des Nutzerkomforts tragen performante Anwendungen auch zu einer Verringerung des Energieverbrauchs und zu niedrigeren Anforderungen an die Hardware bei.

Ein besonders wirkungsvolles Hilfsmittel zur Analyse und Optimierung der Performance sind sogenannte Flamegraphs. Diese Diagramme visualisieren detailliert die Aufrufhierarchie eines Programms und veranschaulichen, wie lange und wie oft bestimmte Funktionen ausgeführt wurden. Im Flamegraph wird die Dauer der Funktionsausführung durch die Größe eines Balkens repräsentiert. Je größer der Balken, desto größer ist auch der Anteil der Rechenzeit, die diese Funktion beansprucht hat.

Mithilfe von Flamegraphs lassen sich sogenannte Hotspots im Programmcode präzise identifizieren. Hotspots sind jene Funktionen oder Codeabschnitte, die überproportional viel Rechenleistung verbrauchen. Durch eine gezielte Untersuchung dieser kritischen Bereiche lassen sich Leistungsengpässe aufdecken und mittels gezielter algorithmischer Verbesserungen, effizientem Caching oder Parallelisierung beheben. Wir haben diese verwendet um während der Entwicklung Methoden zu überprüfen, welche mit größeren Datenmengen hantieren. Hierbei war es uns wichtig, dass das Filesystem so performant wie möglich gestaltet ist. An vielen Stellen ist weiterhin Enwicklungspotenzial für Verbesserungen durch verbesserte Speicherverwaltung. Um sich einen Eindruck über den Aufbau zu verschaffen, befinden sich im Anhang Beispiele (5 6) für die von uns erstellten Graphen. Erstellt haben wir diese mit einem Flamegraph Tool [3]. Für dieses haben wir uns entschieden, da es direkt über cargo erreichbar ist und reibungslos mit Rust funktioniert.

Die dargestellten Graphen zeigen Spitzen, welche durch die rekursive Suche in den Unterverzeichnissen entstehen. Da diese Verzeichnisse von Testläufen generiert wurden, ist es wichtig, zwischen Setup-Code und dem eigentlichen Anwendungscode zu differenzieren. Die gezeigten Graphen dienen ausschließlich der Veranschaulichung typischer Merkmale und sollten daher nicht als reale Performance-Metriken interpretiert werden.

4 Search-Algorithmus

Der initiale Algorithmus sah vor, das gesamte Dateisystem vor dem Start der Anwendung in einen Cache zu laden, welcher während der Laufzeit fortlaufend aktualisiert wurde. Dieses Verfahren ermöglichte eine schnelle Verfügbarkeit von Dateinamen, ging jedoch mit einem erheblichen Ressourcen- und Speicherverbrauch einher. Besonders kritisch wirkte sich die Notwendigkeit aus, die Cache-Datei persistent zu speichern und nachträgliche Änderungen am Dateisystem zu erkennen, sofern die Anwendung in der Zwischenzeit nicht aktiv war. Eine gezieltere Strategie, etwa das Laden einzelner Unterverzeichnisse, wurde im ursprünglichen Konzept nicht berücksichtigt. Aufgrund dieser Herausforderungen wurde das System grundlegend überarbeitet und durch ein effizienteres Verfahren ersetzt, das im Folgenden beschrieben wird.

Die neue Suchmaschine bietet eine performante Lösung für die Dateisuche in großen Verzeichnissen. Nachfolgend wird ihre Architektur von der Kommunikation mit dem Frontend bis hin zu den intern verwendeten Datenstrukturen erläutert.

4.1 Architektonischer Aufbau

Die Suchmaschine folgt einem mehrschichtigen Modell zur Gewährleistung von Modularität, Erweiterbarkeit und Performanz. Die Interaktion erfolgt über `Tauri:Commands`, die in `search_engine_commands.rs` implementiert sin. Wichtige Endpunkte sind:

- `search` – Durchführen einer Suchanfrage
- `add_paths_recursive` – Rekursives Indizieren eines Verzeichnisses
- `remove_paths_recursive` – Entfernen eines Verzeichnisses
- `get_search_engine_info` – Statusinformationen der Suchmaschine

Der zentrale Zustand wird durch `SearchEngineState` in `searchengine_data.rs` durch ein threadsicheres `Arc<Mutex<>>` verwaltet. Dieser umfasst die Statusinformationen (z. B. Indexing, Searching), Fortschrittsmetriken, Leistungsdaten und Konfigurationen. Zudem enthält er die `AutocompleteEngine`, welche die eigentliche Such- und Indexierungslogik kapselt und alle Kernkomponenten koordiniert.

Der Ablauf des Suchprozesses ist in Abbildung 7 schematisch dargestellt und verdeutlicht das Zusammenspiel der einzelnen Komponenten innerhalb der AutocompleteEngine vom Eingang der Suchanfrage bis zur finalen Ergebnisliste.

Die `AutocompleteEngine` setzt sich aus mehreren zentralen Komponenten zusammen: dem Adaptive Radix Trie (ART) als primäre Suchstruktur, einer Fuzzy-Suche als Fallback-Mechanismus, einem LRU-basierten Cache zur Zwischenspeicherung häufiger Anfragen sowie einem kontextbasierten Ranker zur situationsabhängigen Gewichtung der Ergebnisse.

Der ART in `art_v5.rs` ist eine spezialisierte, baumbasierte Datenstruktur zur effizienten Verwaltung und Suche von Pfad-Strings. Im Gegensatz zu klassischen Trie-Implementierungen, bei denen jeder Buchstabe einen eigenen Knoten erhält, nutzt der ART adaptive Knotentypen (Node4 bis Node256), die sich dynamisch an die Anzahl der Kindknoten anpassen. Dadurch werden Speicherverbrauch

und Zugriffszeiten deutlich reduziert. Pfade werden vor dem Einfügen normalisiert, sodass unterschiedliche Schreibweisen und Plattformunterschiede konsistent behandelt werden. Beim Einfügen werden gemeinsame Präfixe erkannt und gemeinsam genutzt, was insbesondere bei vielen ähnlichen Verzeichnissen zu hoher Effizienz führt.

Die Suche nach Präfixen ist besonders performant, da der ART direkt bis zum gesuchten Präfix navigiert und von dort aus alle relevanten Pfade extrahiert. Jeder Eintrag kann mit einem Score versehen werden, der für die Sortierung und das Ranking der Suchergebnisse genutzt wird. Die Ergebnisse werden nach Score sortiert und bei Bedarf dedupliziert.

Das Entfernen von Pfaden ist ebenfalls effizient: Wenn Einträge entfert werden, dann schrumpft der ART automatisch, indem Knotentypen verkleinert oder zusammengeführt werden. Die Implementierung unterstützt zudem das vollständige Leeren des Tries sowie die Begrenzung der maximalen Ergebnisanzahl bei Suchanfragen.

Die wichtigsten Vorteile des ART sind die hohe Performance beim Suchen, Einfügen und Löschen, die weitgehend unabhängig von der Gesamtzahl der gespeicherten Pfade ist. Die adaptive Knotenstruktur sorgt für eine optimale Nutzung des Speichers und ermöglicht eine nahezu logarithmische Skalierung der Zugriffszeiten.

Benchmarks belegen, dass selbst bei einer Steigerung der Pfadanzahl von 10 auf über 170.000 die durchschnittliche Suchzeit lediglich von 1,6 µs auf 22,9 µs anwächst. Dies entspricht einer sublinearen Skalierung, die im praktischen Verhalten zwischen $\mathcal{O}(\log n)$ und $\mathcal{O}(n^a)$ (mit $a \ll 1$) liegt (Abbildung 8).

Im Vergleich zu typischen ART-Implementierungen oder alternativen Strukturen wie einem linearem Scan oder naiven Tries zeigt sich die Implementierung in `art_v5.rs` als besonders effizient. Sie skaliert in der Praxis deutlich besser als $\mathcal{O}(n)$ oder $\mathcal{O}(n \log n)$ und liefert selbst bei großen Datenmengen in Millisekundenbruchteilen konsistente und sortierte Ergebnisse. Damit bildet sie das performante Rückgrat der Suchmaschine für sämtliche Präfix- und Komponentensuchen.

Zur Ergänzung der Präfixsuche kommt ein auf Trigrammen basierender Fuzzy-Suchalgorithmus zum Einsatz, der in `fast_fuzzy_v2.rs` implementiert ist. Der sogenannte Fast Fuzzy Matcher wird aktiv, wenn die durch die Präfixsuche gelieferten Ergebnisse nicht ausreichen oder uneindeutige Anfragen mit Tippfehlern, Auslassungen oder Transpositionen gestellt werden.

Technisch basiert der Matcher auf einem Trigram-Index, in dem für jeden Pfad sämtliche 3-Zeichen-Kombinationen gespeichert werden. Dieser erlaubt eine schnelle Einschränkung des Suchraums auf Kandidaten mit hoher trigrammatischer Ähnlichkeit. Zur Verbesserung der Fehlertoleranz werden zudem Varianten der Suchanfrage generiert, um auch bei unvollständigen oder fehlerhaften Eingaben sinnvolle Treffer zu identifizieren. Die Auswahl und Sortierung der Ergebnisse erfolgt über ein gewichtetes Scoring, das u. a. die Überlappung der Trigramme und die relative Position der Übereinstimmungen berücksichtigt.

Dabei zeigt sich eine empirisch bestätigte sub-lineare Zeitkomplexität von $\mathcal{O}(n^a)$ mit $a \approx 0{,}5$ bis 0,7, was selbst bei stark wachsendem Datenbestand eine praktikable Performance garantiert (Abbildung 9). Im Vergleich zu klassischen Algorithmen wie Levenshtein ($O(N \cdot M^2)$) oder regulären

Ausdrücken ($O(N \cdot Q)$) bietet der Trigram-Ansatz eine deutlich bessere Skalierbarkeit und eignet sich damit besonders für große Dateisammlungen.

Zur Beschleunigung häufig wiederkehrender Anfragen kommt ein **PathCache** mit **LRU-Strategie** zum Einsatz. Er kombiniert eine HashMap mit einer doppelt verketteten Liste für Zugriffszeiten in $\mathcal{O}(1)$ (Abbildung 10) und unterstützt optional TTL-basierte Einträge. Zugriffszeiten liegen je nach Cache-Größe im Bereich von 57 ns bis 265 ns, wobei der Cache mutex-geschützt ist.

Ein kontextbasiertes Ranking priorisiert die Ergebnisse zusätzlich anhand situativer Merkmale wie dem aktuellem Arbeitsverzeichnis, bevorzugten Dateitypen oder Nutzungshistorie. Die finale Bewertung erfolgt durch ein mehrstufiges, normalisiertes Scoring-Modell (Sigmoid), wodurch besonders relevante Ergebnisse nach oben sortiert werden.

Diese Komponenten arbeiten eng zusammen und ermöglichen eine robuste, fehlertolerante und gleichzeitig performante Dateisuche auch bei sehr großen und dynamischen Verzeichnisstrukturen...

Im Ergebnis bildet die Kombination aus ART als Hauptindex, Fast Fuzzy Matcher zur Fehlerkompensation, LRU-Cache für wiederholte Anfragen und kontextabhängiger Ergebnisgewichtung eine leistungsfähige Gesamtlösung. Die Implementierung erreicht durchgehend kurze Antwortzeiten und lässt sich flexibel anpassen, was sie zur geeigneten Basis für moderne Dateiexplorer macht.

Aufgrund von Zeitmangel konnte der Searchalgorithmus im Frontend nicht komplett eingebettet werden. Daran wird jedoch aktiv gearbeitet und insbesondere im Backend sind hierfür alle Konnektoren vorhanden. Ein Punkt der Verbesserung ist das lösen der fehlenden Zugriffsrechte, sobald das Programm selbstständig auf das Filesystem des Users zugreifen möchte.

5 Design Philosophie und Inspiration

5.1 Die Wahl des macOS Finders als Designgrundlage

Bei der Entwicklung unseres File Explorers diente der macOS Finder bewusst als primäre Designinspiration. Ausschlaggebend waren dessen klare Strukturen, minimalistisches Design und eine aufgeräumte Benutzeroberfläche ohne überflüssige Elemente. Besonders der gezielte Einsatz von Whitespace und die klare Hierarchie der Interface-Elemente überzeugten uns. Diese Designsprache unterstützt eine intuitive Navigation und effizientes Arbeiten, ohne durch visuelle Ablenkungen zu stören.

5.2 Integration bewährter Konzepte aus verschiedenenSystemen

Obwohl der macOS Finder unsere Hauptinspiration war, haben wir uns nicht ausschließlich darauf beschränkt. Stattdessen haben wir einen selektiven Ansatz verfolgt und nützliche Features aus anderen Dateiverwaltungssystemen integriert. Ein besonders gelungenes Beispiel ist die Übernahme der "Dieser PCAnsicht aus dem Windows Explorer. Diese Ansicht bietet einen praktischen Überblick über alle verfügbaren Laufwerke und Speichermedien. Wir haben sie jedoch nicht unverändert übernommen, sondern entsprechend unserer minimalistischen Designphilosophie angepasst. Das Ergebnis ist eine funktionale Lösung, die sich nahtlos in unser Gesamtdesign einfügt und die Vorteile beider Systeme kombiniert.

5.3 Nutzerorientierte Anpassungsmöglichkeiten

Ein wichtiger Aspekt unserer Designphilosophie ist die Berücksichtigung individueller Nutzerpräferenzen. Deshalb haben wir verschiedene Anpassungsmöglichkeiten implementiert, die es jedem Nutzer ermöglichen, den File Explorer nach seinen Bedürfnissen zu konfigurieren. Der Dark Mode ist dabei eine grundlegende Option, die modernen Erwartungen entspricht und bei längerer Nutzung die Augenbelastung reduziert. Darüber hinaus können Nutzer die Akzentfarbe individuell anpassen, um eine persönliche Note einzubringen, ohne die Designkohärenz zu beeinträchtigen. Weitere Anpassungen umfassen die Schriftgröße, was besonders bei verschiedenen Bildschirmgrößen und individuellen Sehgewohnheiten hilfreich ist, sowie verschiedene Ansichtsmodi, die je nach Arbeitskontext gewählt werden können.

5.4 Technische Umsetzung und eigenständige Entwicklung

Die technische Basis des Frontends bildet React, das uns die notwendige Flexibilität für eine moderne und reaktive Benutzeroberfläche bietet. Eine bewusste Entscheidung war die Entwicklung eines eigenen CSS-Frameworks anstelle der Nutzung etablierter Lösungen wie Bootstrap oder Tailwind CSS. Diese Entscheidung ermöglicht uns vollständige Kontrolle über das visuelle Erscheinungsbild und stellt sicher, dass jedes Element exakt unseren Designvorstellungen entspricht. Wir sind nicht durch externe Framework-Beschränkungen limitiert und können das Design optimal auf unsere spezifischen Anforderungen abstimmen. Auch die verwendeten Icons haben wir als eigene SVG-Grafiken entwickelt. Dies gewährleistet eine perfekte Integration in das Gesamtdesign und optimiert sowohl die Ladezeiten als auch die Skalierbarkeit der Anwendung.

6 Projektplanung und Kommunikation

Gerade bei umfangreicheren Projekten ist es entscheidend, eine klare und strukturierte Organisation sicherzustellen. Dazu gehört nicht nur eine interne Absprache im Team, sondern auch die Kommunikation mit externen Beteiligten. So können frühzeitig etwaige Verbesserungen identifiziert und gemeinsam umgesetzt werden, um von der ursprünglichen Vision nicht abzuweichen.

6.1 Zusammenarbeit mit Community und dem Ersteller

Um unser Projekt von Anfang an eng an die ursprüngliche Vision von Conaticus anzulehnen und gleichzeitig sicherzustellen, dass unsere Ideen und Umsetzungen in seinem Sinne sind, haben wir frühzeitig persönlichen Kontakt zu ihm aufgenommen. Hierfür sind wir seinem Discord-Server beigetreten, auf dem ein Austausch mit seiner Community stattfand. Durch diesen Dialog hatten wir die Möglichkeit, konkrete Fragen zu stellen und konnten von Conaticus selbst explizit die Erlaubnis und Zustimmung einholen, Änderungen und Erweiterungen an seinem bestehenden Projekt vorzunehmen. Diese Genehmigung wurde uns schriftlich erteilt und kann jederzeit auf Nachfrage eingesehen werden. Zusätzlich haben wir auch die Pull Requests auf GitHub sowie die Kommentarsektionen unter seinen YouTube-Videos ausführlich betrachtet und mögliche Features durchgesprochen. Dadurch konnten wir erkennen, welche Funktionen besonders gefragt sind, und den Programmcode so strukturieren, dass zukünftige Features einfacher integriert werden können. Etwa zur Halbzeit des Projektes führten wir ein Online-Meeting mit Conaticus persönlich durch, um direktes Feedback zu unseren bisherigen Umsetzungen einzuholen. Das Gespräch verlief ausgesprochen positiv und Conaticus war bezüglich des neuen Interfaces und der Funktionen sehr erfreut.

6.2 Kommunikation im Team

Zu Beginn des Projekts haben wir überlegt, wie wir unsere Zusammenarbeit möglichst klar und strukturiert gestalten können. Dabei war uns wichtig, den Code sinnvoll aufzubauen und gleichzeitig den Überblick über Aufgaben und Fortschritte zu behalten. Zur Organisation haben wir Trello genutzt: Dort haben wir das Projekt in einzelne Aufgaben aufgeteilt, diese den Teammitgliedern zugewiesen und den Fortschritt über Listen wie „To Do“, „In Bearbeitung“ und „Erledigt“ festgehalten. So konnten wir effizient zusammenarbeiten und jederzeit nachvollziehen, was bereits umgesetzt wurde und noch zu implementieren galt.

Anhang

Vorher und Nachher Bilder

Vorher

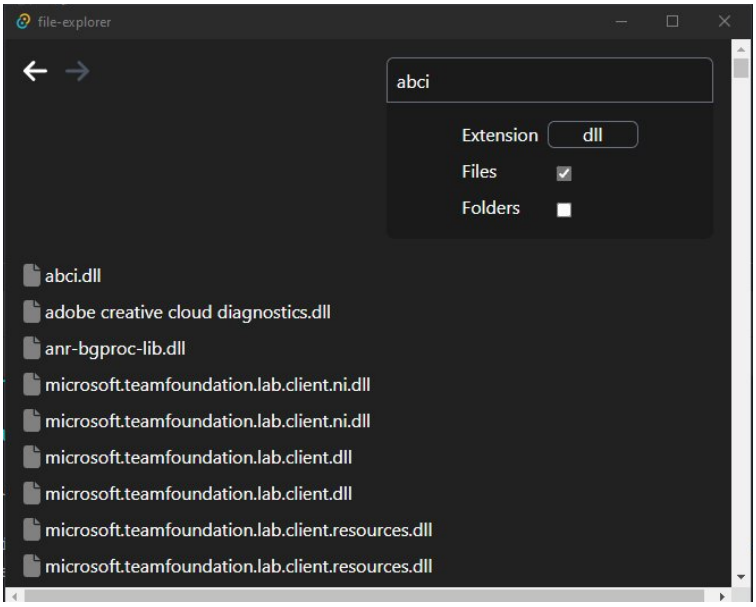

Abbildung 1: Vorher. Quelle ist https://github.com/conaticus/FileExplorer

Nachher

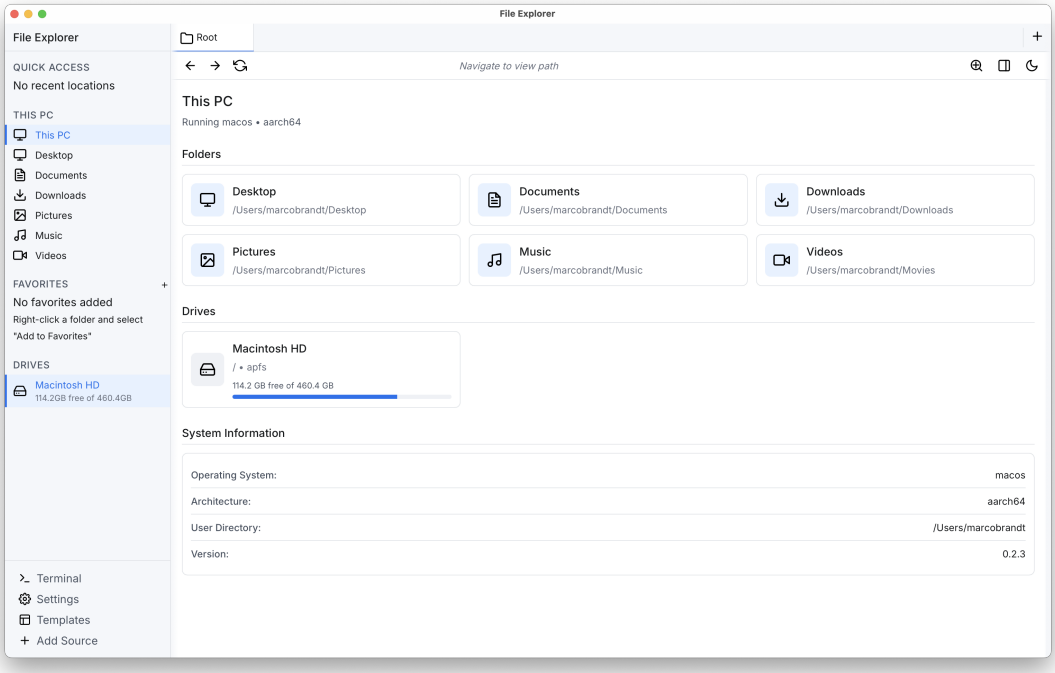

Abbildung 2: Übersicht danach

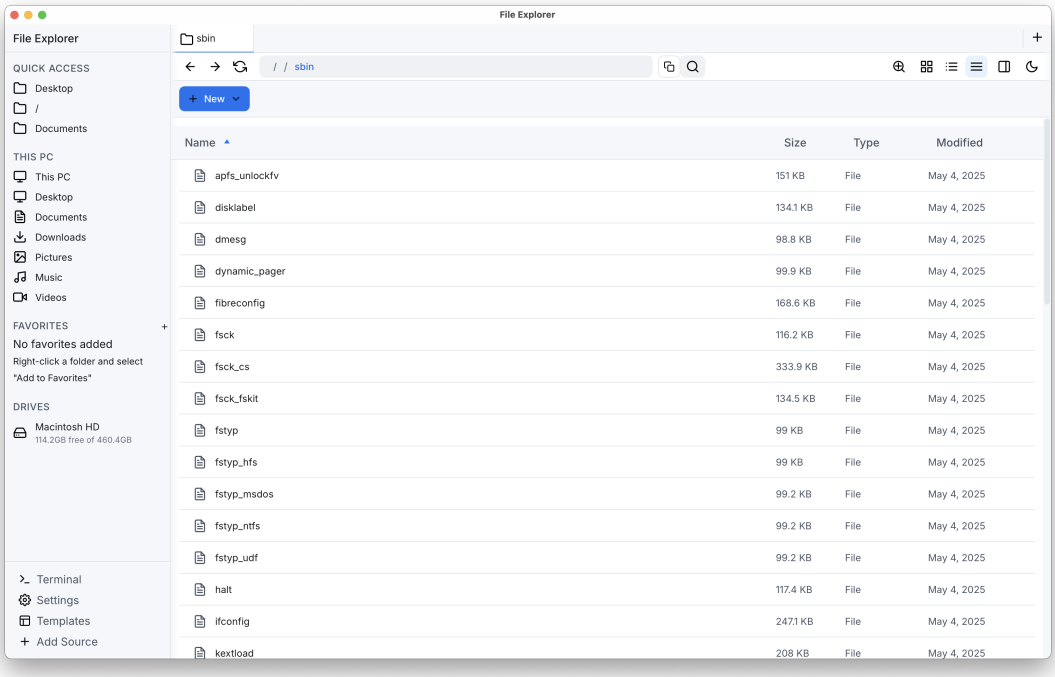

Abbildung 3: Files danach

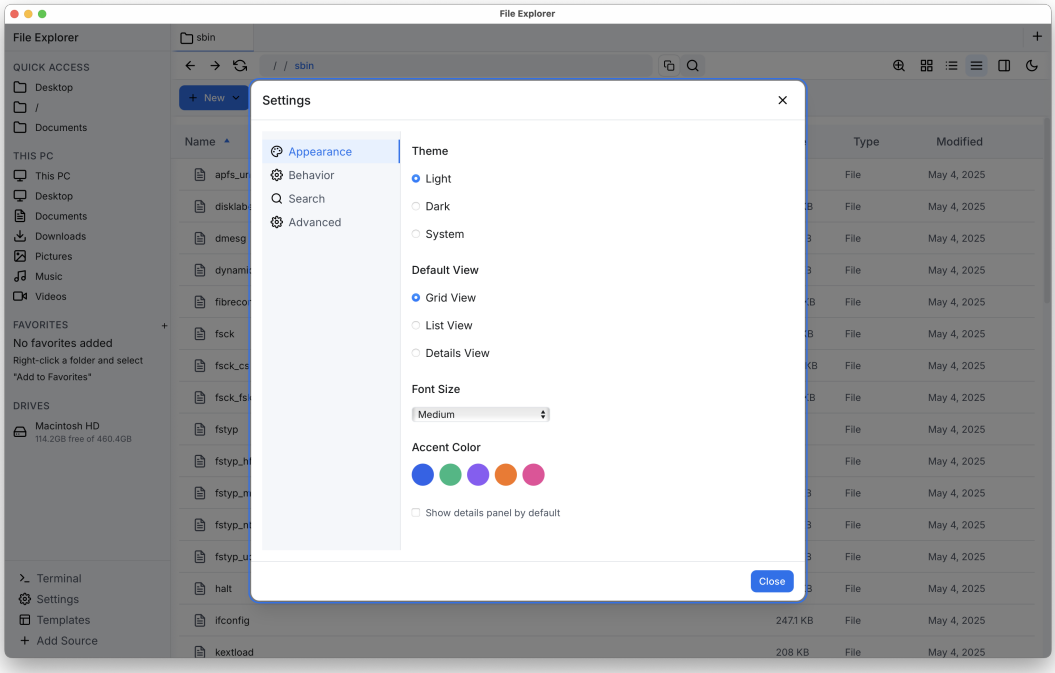

Abbildung 4: Settings

State Beispiel

Rust Code:

```
use tauri::State;
struct MyState(String);
#[tauri::command]
fn get_value(state: State<'_, MyState>) -> String {
    state.0.clone()
}
fn main() {
    tauri::Builder::default()
        .manage(MyState("Hello from backend".to_string())) // Initial value
        .invoke_handler(tauri::generate_handler![get_value])
        .run(tauri::generate_context!())
        .expect("error while running tauri application");
}
```

Frontend in Typescript

```
'use client';
import { useEffect, useState } from 'react';
import { invoke } from '@tauri-apps/api/tauri';
export default function GetStateExample() {
  const [value, setValue] = useState<string>('');
  useEffect(() => {
    async function fetchValue() {
      const result = await invoke<string>('get_value');
      setValue(result);
    }
    fetchValue();
  }, []);
  return (
    <div>
      <h1>Backend State</h1>
      <p>{value}</p>
    </div>
  );
}
```

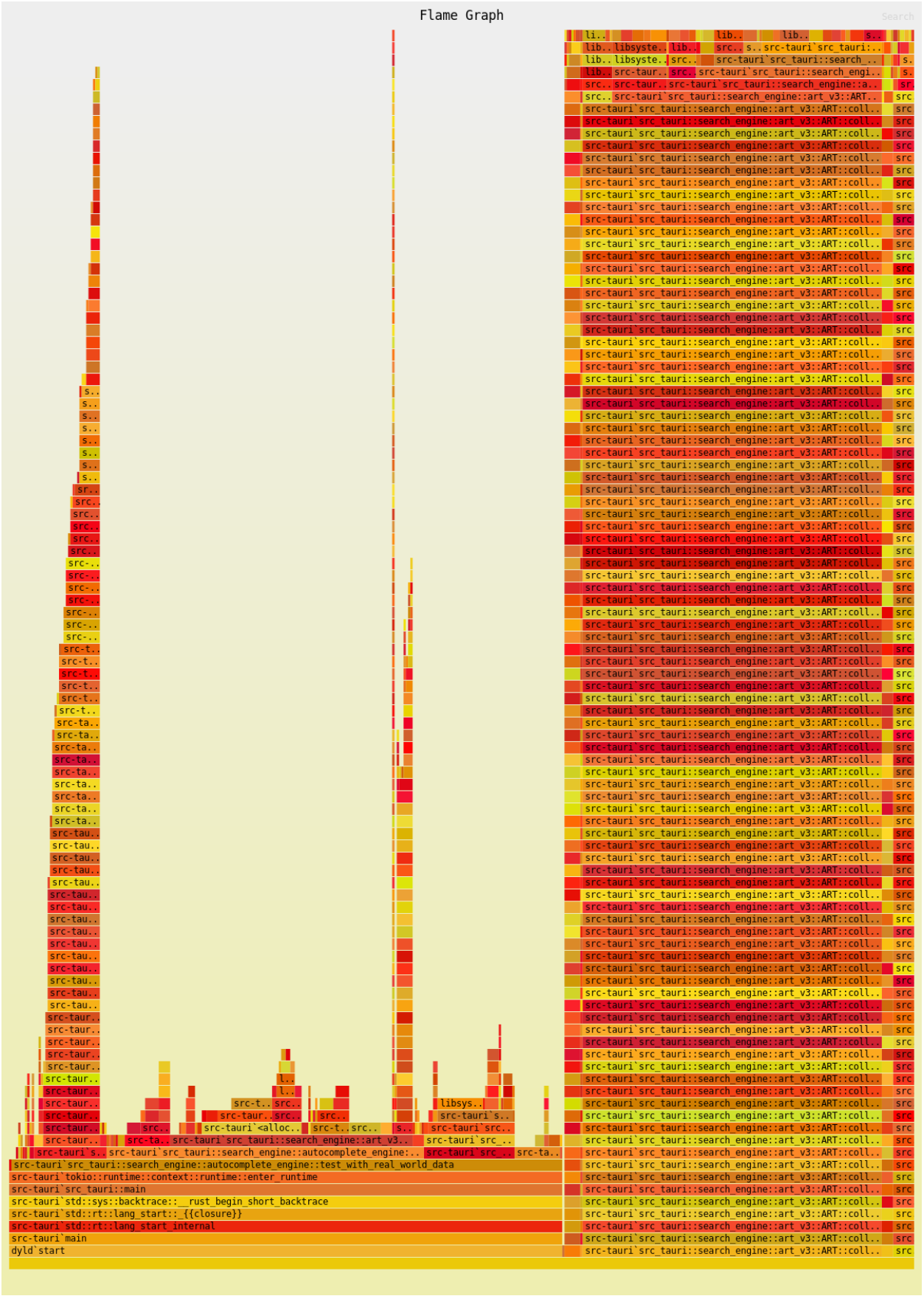

Abbildung 5: Flamegraph von einem Searchengine Durchlauf mit 500 Einträgen

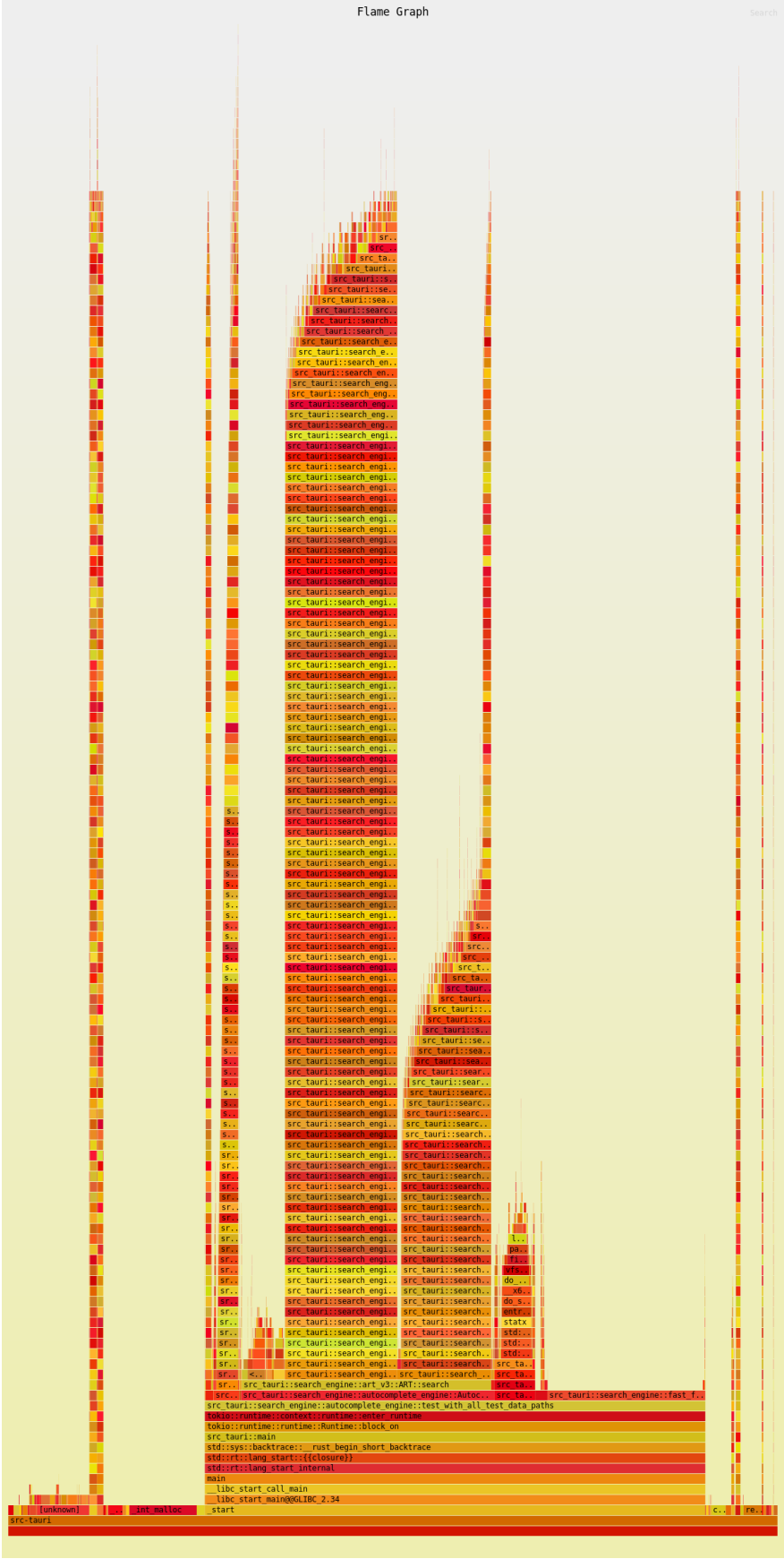

Abbildung 6: Flamegraph von einem Searchengine Durchlauf mit allen Einträgen der Testdaten

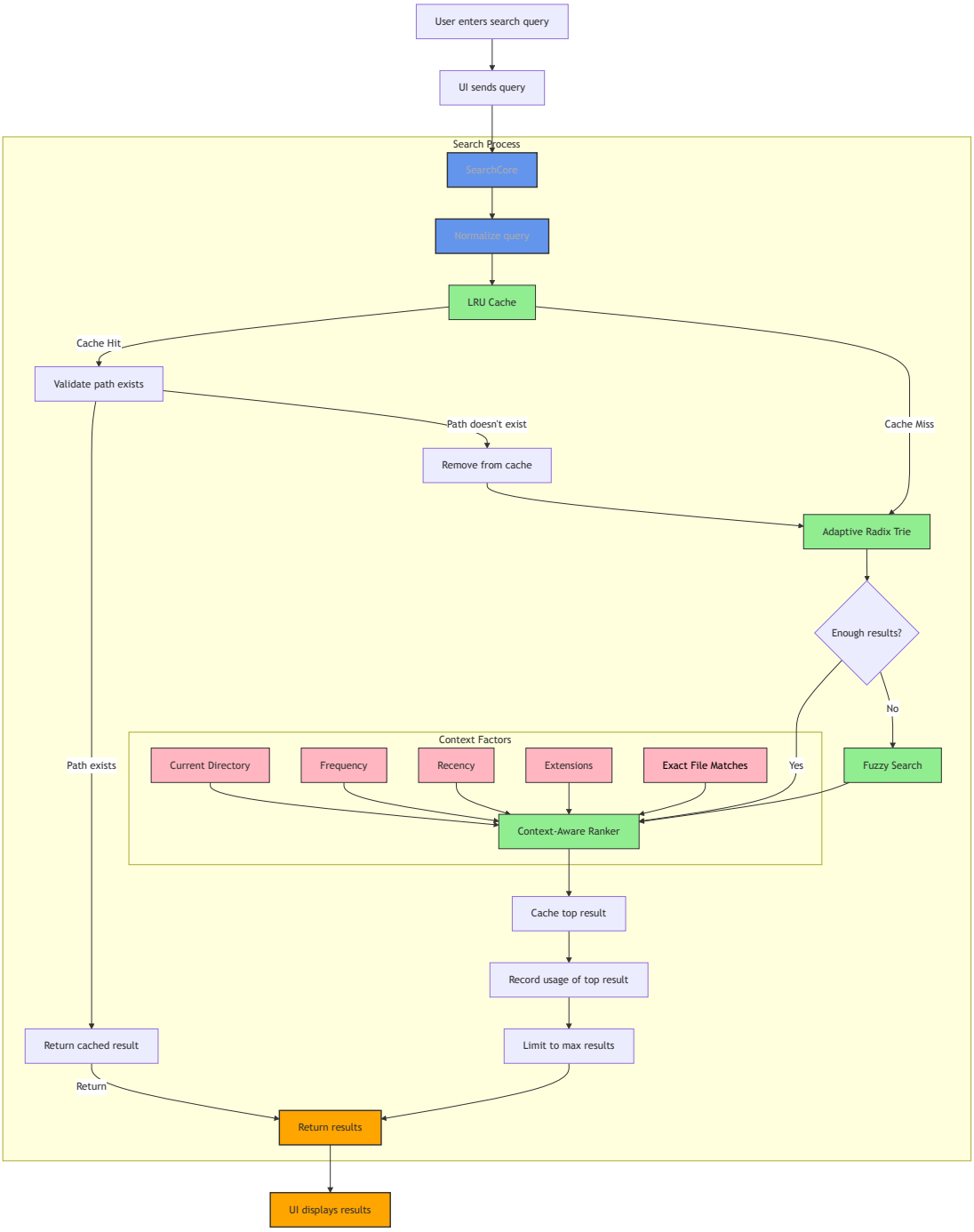

Abbildung 7: Search engine Prozessdiagram

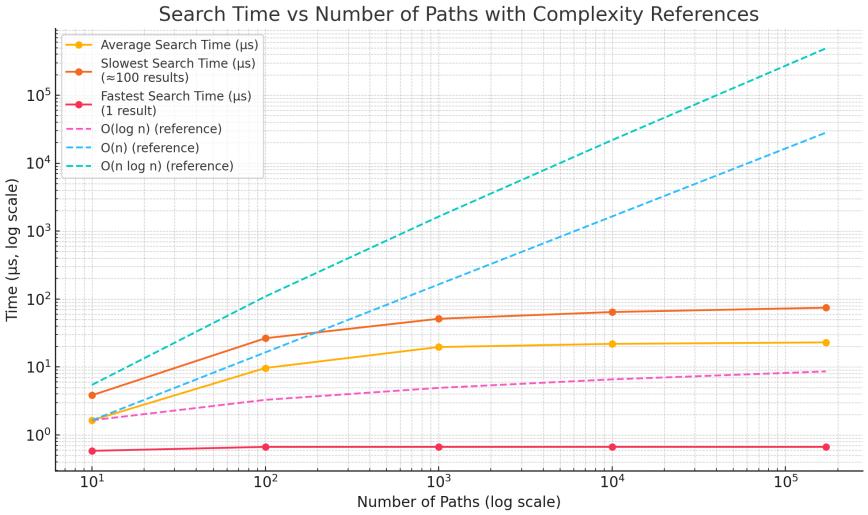

Abbildung 8: Adaptive Radix Trie Zeitkomplexität

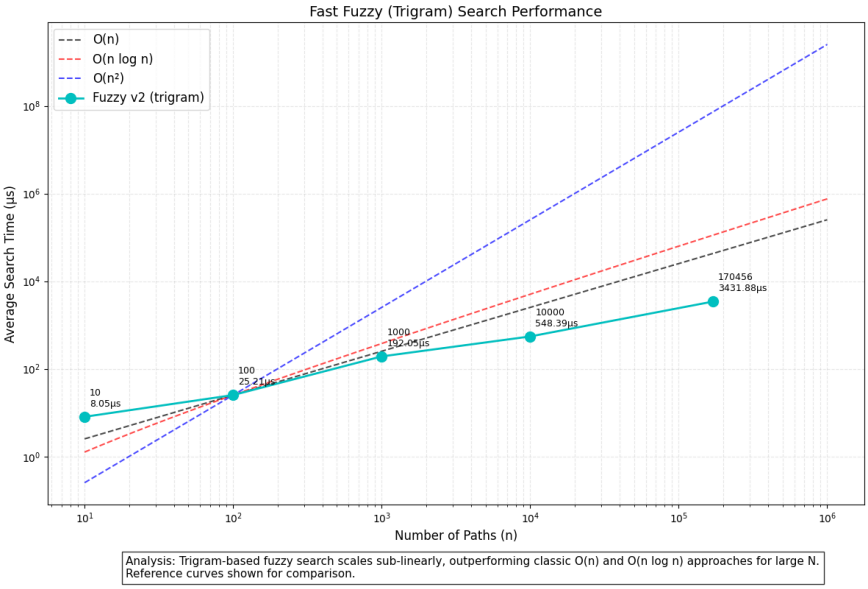

Abbildung 9: Fuzzy Suche Zeitkomplexität

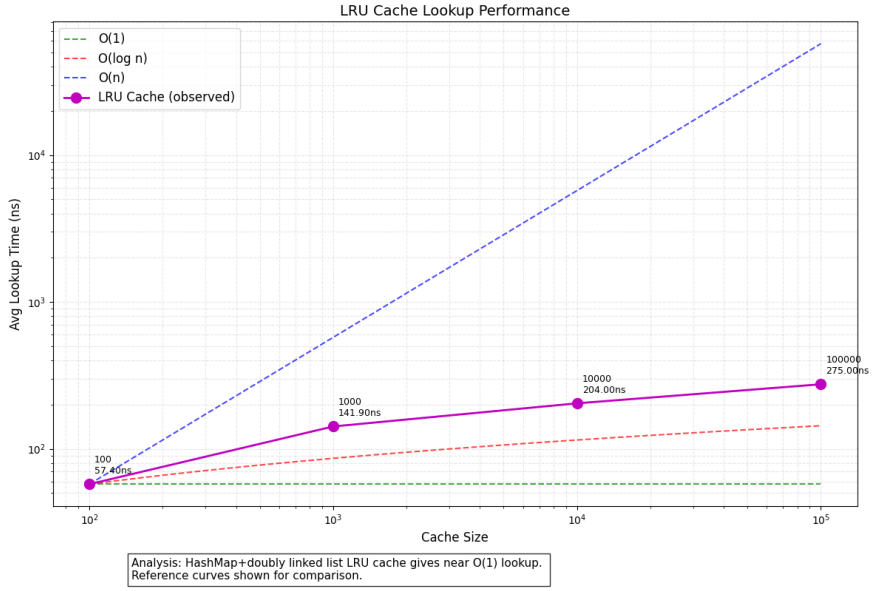

Abbildung 10: LRU Cache Zeitkomplexität

Literaturverzeichnis

[1] conaticus01. *I Made a FAST File Explorer (in Rust)*. https://www.youtube.com/watch?v=Z60f2g-COJY. YouTube video. Juni 2023.

[2] conaticus02. *How I RUINED My Rust Project*. https://youtu.be/4wdAZQROc4A?si=ucYtPULZfrVvErBy. YouTube video. Juli 2023.

[3] Johann Hofmann und contributors. *flamegraph-rs: A Rust implementation of Flamegraph*. https://github.com/flamegraph-rs/flamegraph. Accessed: 2025-05-22. 2024.

[4] Tauri Project. *State Management*. Accessed: 2025-05-10. 2025. URL: https://v2.tauri.app/develop/state-management/.

[5] Tauri Project. *Tauri - Build Smaller, Faster, and More Secure Desktop Applications with a Web Frontend*. Accessed: 2025-05-10. 2025. URL: https://v2.tauri.app/.

Abbildungsverzeichnis

1 Vorher. Quelle ist https://github.com/conaticus/FileExplorer II
2 Übersicht danach II
3 Files danach III
4 Settings III
5 Flamegraph von einem Searchengine Durchlauf mit 500 Einträgen V
6 Flamegraph von einem Searchengine Durchlauf mit allen Einträgen der Testdaten VI
7 Search engine Prozessdiagram VII
8 Adaptive Radix Trie Zeitkomplexität VIII
9 Fuzzy Suche Zeitkomplexität VIII
10 LRU Cache Zeitkomplexität IX