Schriftliche Hausarbeit
in Informatik
Ludwig-Maximilians-Universität München

Migrationsstrategien von JavaScript zu TypeScript: Evaluation und Implementierung in BenchExec

Simon Hümmer

Prüfer:	Prof. Dr. Dirk Beyer
Mentor:	Dr. Philipp Wendler
Abgabedatum:	11.03.2026
Matrikelnummer:	12202995

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig angefertigt habe. Die aus fremden Quellen direkt und indirekt übernommenen Gedanken sind als solche kenntlich gemacht.
Zur sprachlichen Überarbeitung von Textpassagen (Korrekturlesen und stilistische Verbesserungen) wurde das KI-basierte Sprachmodell ChatGPT und der KI-basierte Schreibassistent Grammarly genutzt. Die inhaltliche Konzeption, Analyse, Bewertung der Ergebnisse sowie sämtliche fachlichen Entscheidungen wurden eigenständig getroffen.
Die Arbeit wurde weder einer anderen Prüfungsbehörde vorgelegt noch veröffentlicht.

München, den 11.03.2026

Simon Hümmer

.....

Simon Hümmer

Zusammenfassung

BenchExec ist ein Benchmarking-Framework des Software Systems Lab der LMU München. Zum Projekt gehört eine Tabellendarstellung, dessen Benutzeroberfläche bislang in JavaScript implementiert war. Ziel dieser Arbeit ist die Migration dieses Programmcodes zu TypeScript unter Wahrung des bestehenden Funktionsumfangs und ohne Umstrukturierungen.

Zur Vorbereitung wird eine Auswahl automatisierter Migrationswerkzeuge untersucht und anhand definierter Kriterien bewertet. Die Evaluation berücksichtigt dabei technische Korrektheit, Codequalität sowie Prozess- und Aufwandsaspekte und dient als Grundlage für die Auswahl des im Projekt eingesetzten Werkzeugs. Aufbauend auf diesem Ergebnis wird eine Migrationsstrategie abgeleitet, dokumentiert und praktisch umgesetzt. Die Durchführung folgt einer strukturierten, commit-basierten Vorgehensweise, bei der Tool-Output und manuelle Nacharbeit getrennt nachvollziehbar bleiben.

Anhand ausgewählter Beispiele wird gezeigt, wie TypeScript implizite Annahmen, inkonsistente Zustandsrepräsentationen und unklare Komponenten-Schnittstellen sichtbar macht und dadurch eine präzisere Modellierung von Datenflüssen ermöglicht. Abschließend werden die Ergebnisse der praktischen Migration im Kontext der Evaluation eingeordnet und Grenzen der Umsetzbarkeit diskutiert.

Inhaltsverzeichnis

Kapitel 1

Einleitung

BenchExec [3] ist ein Framework zur reproduzierbaren Durchführung und Auswertung von Benchmarks, das in der internationalen Forschung zur Softwareverifikation eingesetzt wird. Es bildet unter anderem die technische Grundlage des jährlichen Wettbewerbs *International Competition on Software Verification* [2], bei dem Verifikationswerkzeuge unter standardisierten Bedingungen ausgeführt und ausgewertet werden. Ein zentraler Bestandteil von BenchExec ist die webbasierte Ergebnisdarstellung, über die Nutzer große Mengen von Benchmarkdaten interaktiv analysieren können.

Die Ergebnisdarstellung ist in JavaScript implementiert und ermöglicht es, Ergebnisse zu filtern, zu sortieren sowie in unterschiedlichen Darstellungsformen wie Tabellen oder Diagrammen zu visualisieren. Abbildung 1.1 zeigt exemplarisch die interaktive Benutzeroberfläche mit einer tabellarischen Ergebnisübersicht, konfigurierbaren Spalten sowie dynamischen Filterelementen wie Bereichsreglern und Texteingaben.

Mit zunehmendem Funktionsumfang und wachsendem Quellcode stößt eine rein dynamisch typisierte Implementierung jedoch an ihre Grenzen. Insbesondere werden Wartbarkeit, Verständlichkeit des Codes und die frühzeitige Fehlererkennung dadurch erschwert, dass Typinkonsistenzen erst zur Laufzeit sichtbar werden. Statische Typprüfungen können dem entgegenwirken, indem potenzielle Fehler bereits in der Entwicklungsphase erkannt werden.

Für webbasierte Anwendungen hat sich TypeScript [4] als eine der meistgenutzten Programmiersprachen etabliert. Als Open-Source-Erweiterung von JavaScript [11] ergänzt TypeScript die Sprache um statische Typisierung und zusätzliche Strukturierungsmöglichkeiten, die insbesondere für größere und langfristig gepflegte Projekte von Vorteil sind. Dadurch können die Wartbarkeit und Robustheit komplexer Anwendungen verbessert werden.

Vor diesem Hintergrund stellt die Migration eine zentrale Strategie zur Modernisierung von bestehender Software dar. Dabei wird ein bestehendes System

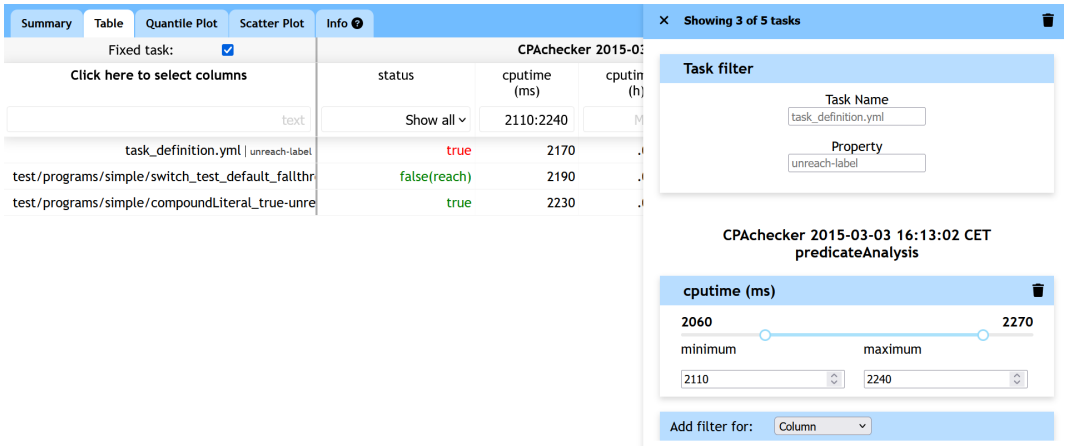

Abbildung 1.1: Webbasierte Benutzeroberfläche von BenchExec mit tabellarischer Ergebnisdarstellung und interaktiven Filterelementen

technisch weiterentwickelt, ohne es vollständig neu zu implementieren. Ziel ist es, die Vorteile moderner Technologien zu nutzen und gleichzeitig möglichst viel bestehenden Code wiederzuverwenden. In dieser Arbeit wird diese Strategie verwendet, um die mit JavaScript implementierte Benutzeroberfläche von BenchExec auf TypeScript umzustellen.

Migrationen sind in der Praxis jedoch mit erheblichem Aufwand verbunden. Um diesen zu strukturieren und zu reduzieren, existieren sowohl unterschiedliche Migrationsstrategien als auch verschiedene automatisierte Werkzeuge, deren Eignung stark vom jeweiligen Quellcode und Projektkontext abhängt. Ziel dieser Abhandlung ist es, geeignete Ansätze im Kontext von BenchExec zu untersuchen, automatisierte Migrationstools systematisch zu evaluieren und anschließend eine geeignete Migrationsvorgehensweise zu identifizieren und praktisch umzusetzen.

Um die technischen Hintergründe der Migration sowie die zugrunde liegenden Technologien zu verstehen, werden im Kapitel 2 zunächst die wichtigsten Grundlagen zu JavaScript, TypeScript und der in BenchExec verwendeten React-Architektur erläutert. Kapitel 3 stellt verschiedene Migrationsstrategien sowie ausgewählte Migrationstools vor. In Kapitel 4 wird die Bewertungsmethodik erläutert und die zugrunde liegenden Evaluationskriterien definiert. Kapitel 5 präsentiert die systematische Bewertung der Werkzeuge und fasst die zentralen Ergebnisse zusammen. Die detaillierte Analyse der einzelnen Migrationswerkzeuge ist im Anhang dokumentiert. Kapitel 6 beschreibt die praktische Umsetzung der Migration im Projekt BenchExec. Den Abschluss der Arbeit bildet Kapitel 7 mit einer Zusammenfassung der Ergebnisse sowie einem Ausblick auf weiterführende Arbeiten.

Kapitel 2

Grundlagen

2.1 JavaScript und TypeScript

JavaScript [11] ist eine dynamisch typisierte Programmiersprache, die primär zur Entwicklung interaktiver Webanwendungen eingesetzt wird. Typen werden erst zur Laufzeit festgelegt, was eine hohe Flexibilität ermöglicht. Gleichzeitig erschwert diese Dynamik jedoch die statische Überprüfung des Codes, da Typfehler häufig erst während der Ausführung des Projekts sichtbar werden.

TypeScript [4] setzt genau an diesem Punkt an. Als Obermenge von JavaScript erweitert es die Sprache um ein statisches Typsystem sowie zusätzliche Strukturierungsmechanismen. Durch explizite Typannotationen lassen sich viele Fehler bereits während der Entwicklungszeit erkennen. Der TypeScript-Code wird vor der Ausführung in JavaScript übersetzt und bleibt vollständig kompatibel mit bestehenden Laufzeitumgebungen.

Insbesondere in größeren und langfristig gepflegten Codebasen können diese Eigenschaften zu erhöhter Wartbarkeit und Robustheit beitragen. Für die Migration der BenchExec-Tabellen sind diese Unterschiede daher von zentraler Bedeutung. Tabelle 2.1 fasst die wesentlichen Merkmale beider Sprachen zusammen.

2.2 React, JSX, Hooks und Komponenten

Die Weboberfläche von BenchExec basiert auf der JavaScript-Bibliothek React [10]. React verfolgt einen komponentenbasierten Ansatz, bei dem Benutzeroberflächen aus wiederverwendbaren Bausteinen zusammengesetzt werden. Das bedeutet, jede Komponente besitzt sowohl ihre eigene Darstellungsstruktur als auch die dafür notwendige Logik. In modernen React-Anwendungen werden überwiegend funktionale Komponenten verwendet. Diese sind JavaScript-

Merkmal	**JavaScript**	**TypeScript**
Typisierung	Dynamisch	Statisch
Kompilierung	Direkt ausführbar im Browser	Muss vor der Ausführung in JavaScript kompiliert werden
Fehlererkennung	Zur Laufzeit	Zur Entwicklungszeit
Lernkurve	Einfach für Anfänger	Tieferes Verständnis nötig
Anwendungsbereiche	Kleine bis mittelgroße Projekte	Große und komplexe Projekte

Tabelle 2.1: Vergleich von JavaScript und TypeScript [1]

Funktionen, die Benutzeroberflächenelemente zurückgeben. Die Beschreibung dieser Benutzeroberfläche erfolgt dabei typischerweise mithilfe von JSX, einer Syntaxerweiterung von JavaScript, die HTML-ähnliche Strukturen direkt im Code ermöglicht. JSX verbessert die Lesbarkeit und Strukturierung von Komponenten und wird vor der Ausführung in reguläres JavaScript übersetzt.

Funktionale Komponenten sind in der Regel parametrisierbar und erhalten ihre Konfiguration über Eigenschaften (Properties, kurz: Props). Diese Eigenschaften stellen die Schnittstelle zwischen Komponenten dar und bestimmen, wie eine Komponente dargestellt wird und wie sie sich verhält. Im Rahmen einer Migration von JavaScript zu TypeScript ist eine korrekte Typisierung dieser Eigenschaften daher besonders relevant, da sie die explizite Definition der Komponentenschnittstellen erzwingt und damit zur strukturellen Absicherung der Anwendung beiträgt.

In modernen React-Versionen können die Funktionskomponenten durch Hooks [6] unabhängig von Klassenkomponenten ihren eigenen Zustand (State) und Lebenszyklusfunktionen verwalten. Zu den wichtigsten Hooks zählen:

- `useState`: Verwaltet den lokalen Zustand einer Komponente und ermöglicht dessen Aktualisierung.
- `useEffect`: Verwaltet Nebenwirkungen wie API-Aufrufe, Timer oder direkte DOM-Manipulationen.
- `useMemo` und `useCallback`: Optimieren die Performance, indem sie unnötige Neuberechnungen und Re-Renderings vermeiden.

2.3 Typisierung von React Hooks

Um die Notwendigkeit einer Typisierung von funktionalen Komponenten in BenchExec zu verdeutlichen wird im Folgenden ein vereinfachter Ausschnitt einer Datei zur Verarbeitung von Textfiltern betrachtet.

```
const ref = useRef(null);
let [typingTimer, setTypingTimer] = useState("");
let [value, setValue] = useState(initFilterValue);

useEffect(() => {
  if (focusedFilter === elementId) {
    ref.current.focus();
  }
}, [focusedFilter, elementId]);

const onChange = (event) => {
  const newValue = event.target.value;
  setValue(newValue);
  clearTimeout(typingTimer);
  setTypingTimer(
    setTimeout(() => {
      setCustomFilters({ id, value: newValue });
      document.getElementById(elementId).focus();
    }, 500),
  );
};
```

Listing 2.1: Vereinfachter, untypisierter Codeausschnitt aus BenchExec

Die dargestellte Komponente verwaltet mithilfe der Hooks `useRef`, `useState` und `useEffect` den aktuellen Wert eines Textfilters, einen Verzögerungstimer sowie den Fokus eines Eingabefeldes. Änderungen am Eingabefeld werden verzögert verarbeitet, um Aktualisierungen bei jeder Tastenbetätigung zu vermeiden und die Benutzerinteraktion mit großen Tabellen zu verbessern.

In der untypisierten JavaScript-Version bleiben mehrere zentrale Aspekte des Codes implizit. So wird der Zustand `typingTimer` zunächst mit einer Zeichenkette initialisiert, später jedoch mit dem Rückgabewert von `setTimeout` überschrieben. Die Variable nimmt somit im Verlauf der Ausführung unterschiedliche Typen an. Da JavaScript keine statische Typprüfung besitzt, bleibt diese Inkonsistenz zunächst unbemerkt und kann erst zur Laufzeit sichtbar werden.

Wird beispielsweise `clearTimeout` mit einem ungeeigneten Wert aufgerufen, kann der zuvor gesetzte Timer nicht korrekt gelöscht werden. In der Folge können mehrere Timer gleichzeitig aktiv bleiben und die verzögerte Aktualisierung des Filters mehrfach oder zu unerwarteten Zeitpunkten ausgeführt werden. Solche Fehler äußern sich häufig in inkonsistentem Verhalten der Benutzeroberfläche und sind im dynamisch typisierten Code nur schwer nachzuvollziehen.

Durch eine explizite Typisierung in TypeScript wird diese Problematik bereits während der Entwicklung erkennbar. Der Entwickler muss festlegen, welchen Typ der Zustand tatsächlich besitzen darf, wodurch inkonsistente Zuweisungen verhindert werden. Das folgende Codebeispiel zeigt eine typisierte Variante des Codeausschnittes.

```
const ref = useRef<HTMLInputElement | null>(null);
const [typingTimer, setTypingTimer] =
  useState<ReturnType<typeof setTimeout> | null>(null);
const [value, setValue] = useState<string>(
  initFilterValue,
);

useEffect(() => {
  if (focusedFilter === elementId) {
    ref.current?.focus();
  }
}, [focusedFilter, elementId]);

const onChange = (
  event: React.ChangeEvent<HTMLInputElement>,
) => {
  const newValue = event.target.value;
  setValue(newValue);

  if (typingTimer !== undefined) {
    clearTimeout(typingTimer);
  }
  setTypingTimer(
    setTimeout(() => {
      setCustomFilters({ id, value: newValue });
      document.getElementById(elementId).focus();
    }, 500),
  );
};
```

Listing 2.2: Typisierte Variante des Codebeispiels in TypeScript

Hier wird ersichtlich, dass `ref` auf ein `HTMLInputElement` oder `null` verweist, `typingTimer` entweder eine `setTimeout`-Rückgabe oder `null` speichert und `value` vom Typ `string` ist. Zudem wird durch die Typannotation des `event`-Parameters sichergestellt, dass ausschließlich gültige Eigenschaften eines `React.ChangeEvent<HTMLInputElement>` verwendet werden können.

2.4 BenchExec

2.4.1 Projektüberblick und Umfang

BenchExec [3] ist ein Open-Source-Framework zur automatisierten und reproduzierbaren Durchführung von Benchmarks. Neben der Ausführung und Auswertung von Programmen stellt BenchExec umfangreiche Werkzeuge zur Darstellung von Ergebnissen bereit. Die Benutzeroberfläche ist dabei ein zentraler Bestandteil, da sie als primäre Schnittstelle zur Analyse der Benchmark-Ergebnisse dient.

Der JavaScript-basierte Teil der Benutzeroberfläche umfasst eine Vielzahl von Komponenten zur Darstellung und Filterung von Ergebnissen. Der Code ist historisch gewachsen und wurde über mehrere Jahre hinweg schrittweise erweitert [5] [14]. Dadurch ergibt sich ein nicht-trivialer Projektumfang von ca. 11.000 Zeilen Code und ca. 50 JavaScript-Dateien.

2.4.2 Architektur und Ist-Zustand

Die Benutzeroberfläche von BenchExec ist klar von der Benchmark-Ausführung getrennt. Während die Erzeugung der Benchmarks durch Python-Komponenten von BenchExec ausgeführt und die entsprechenden Ergebnisdaten erzeugt werden, erfolgt die Interaktion mit den Ergebnissen vollständig lokal im Webbrowser. Die generierten HTML-Seiten enthalten sowohl die darzustellenden Daten als auch den JavaScript-Code zur Verarbeitung und Visualisierung, sodass keine serverseitige Komponente für die Analyse der Ergebnisse erforderlich ist.

Technisch basiert die Benutzeroberfläche auf React und verwendet sowohl funktionale als auch klassische Klassenkomponenten. Moderne React-Konzepte wie Hooks werden stellenweise eingesetzt, insbesondere zur Verwaltung lokaler Zustände, zur Reaktion auf Benutzerinteraktionen sowie zur Performance-Optimierung bei großen Tabellen. Diese Mischung unterschiedlicher Komponentenstile ist typisch für einen gewachsenen Quellcode und stellt eine zusätzliche Herausforderung für die Typisierung dar.

Die Komponenten arbeiten stark datengetrieben, da Filter-, Sortier- und Analysefunktionen ihre Struktur und ihr Verhalten aus den zur Laufzeit geladenen Benchmark-Ergebnissen ableiten. Da der Code vollständig in JavaScript implementiert ist, existieren bislang keine expliziten Typdefinitionen für diese Datenstrukturen, was die Nachvollziehbarkeit der Datenflüsse erschwert.

2.4.3 Qualitätssicherung und Tests

BenchExec wird aktiv weiterentwickelt und legt großen Wert auf Qualitätssicherung. Für verschiedene Teile des Projekts existieren automatisierte Tests, die insbesondere die korrekte Verarbeitung und Darstellung von Benchmark-Ergebnissen absichern. Ebenfalls gibt es Tests für die Benutzeroberfläche, die garantieren, dass zentrale Funktionalitäten wie Filtermechanismen und Ergebnisdarstellungen bei Codeänderungen erwartungsgemäß funktionieren. Dabei kommen überwiegend Snapshot-Tests zum Einsatz, mit denen Änderungen an der gerenderten Benutzeroberfläche zuverlässig erkannt werden können.

Die vorhandene Testinfrastruktur bildet eine wichtige Grundlage für eine Migration auf TypeScript, da sie hilft unbeabsichtigte Verhaltensänderungen frühzeitig zu erkennen. Gleichzeitig zeigt sich, dass die vorhandenen Tests allein keine vollständige Absicherung gegen Typfehler bieten können, da viele potenzielle Probleme erst durch fehlende oder inkonsistente Datenstrukturen entstehen.

2.4.4 Projektentwicklung und Wartung

BenchExec wird als Open-Source-Projekt entwickelt und hauptsächlich von einer kleinen Anzahl von Entwicklern gepflegt. Neue Funktionen und Verbesserungen werden häufig im Rahmen einzelner, in sich abgeschlossener Projekte schrittweise integriert.

Unter diesen Rahmenbedingungen ist eine gute Code-Wartbarkeit von besonderer Bedeutung. Änderungen müssen nachvollziehbar sein und dürfen bestehende Funktionalitäten nicht unbeabsichtigt beeinträchtigen.

Kapitel 3

Migrationsstrategien und -tools

Die dargestellten Eigenschaften der Benutzeroberfläche – insbesondere die klare Abgrenzung vom Backend, die gewachsene React-Architektur sowie das Fehlen expliziter Typdefinitionen – prägen maßgeblich die Ausgangssituation für eine Migration auf TypeScript und werfen die Frage auf, welche Migrationsstrategie für ein Projekt wie BenchExec geeignet ist und in welchem Umfang automatisierte Werkzeuge dabei eingesetzt werden können.

3.1 Meta-Strategien der Migration

Die Migration von JavaScript zu TypeScript kann auf unterschiedliche Weise erfolgen. Eine Studie [12] untersuchte praxisnah die Vorgehensweise in verschiedenen Softwareprojekten und identifizierte drei Hauptstrategien: *Incomplete*, bei der TypeScript zwar eingeführt wird, JavaScript jedoch dauerhaft im Projekt verbleibt, *Gradual*, bei der die Migration schrittweise über längeren Zeitraum erfolgt, und *Sudden*, bei der große Teile des JavaScript-Codes in sehr kurzer Zeit durch TypeScript ersetzt werden. In nur wenigen Fällen wurde das Projekt vollständig auf TypeScript umgestellt. Die Ergebnisse zeigen, dass eine Koexistenz beider Sprachen möglich ist und eine gewisse Flexibilität bietet, sodass auch eine teilweise Konvertierung eines Projekts im Rahmen einer Migration sinnvoll sein kann. Für das Projekt BenchExec sind diese Strategien hinsichtlich ihrer Anwendbarkeit zu bewerten.

Die *Incomplete*-Strategie erscheint wenig zielführend, da sie dauerhaft einen heterogenen Codezustand etablieren würde.

Die *Gradual*-Strategie würde zu einer längeren Koexistenz von JavaScript- und TypeScript-Code führen. Ein solcher Mischbetrieb erhöht die Komplexität der Codebasis, erschwert die Wartung und reduziert den unmittelbaren Nutzen der statischen Typisierung.

Vor diesem Hintergrund bietet sich die *Sudden*-Strategie an. Der Quellcode der Benutzeroberfläche ist überschaubar und klar abgegrenzt, sodass eine vollständige Migration innerhalb eines begrenzten Zeitraums realistisch ist. Zudem wird BenchExec von einer kleinen Entwicklergruppe gepflegt, wodurch eine vorübergehende Einschränkung der Weiterentwicklung während der Migration vertretbar ist.

3.2 Allgemeiner Migrationsleitfaden

Unabhängig von der konkret gewählten Migrationsstrategie folgt die Migration von JavaScript auf TypeScript in der Praxis häufig einem grundlegenden Ablauf. Heiskanen [7] beschreibt einen allgemeinen Leitfaden mit neun Schritten, der als Orientierung für die Migration von Webanwendungen dient. Im Folgenden werden die einzelnen Schritte des Leitfadens kurz zusammengefasst:

1. **Migrationsstrategie festlegen:** Es wird entschieden, ob die Migration nur teilweise oder vollständig erfolgen soll. Dies hängt von der Größe und Komplexität des Projekts ab.
2. **Bibliotheksunterstützung prüfen:** Die verwendeten Pakete werden auf ihre Kompatibilität mit TypeScript überprüft. Gegebenenfalls müssen zusätzliche Typ-Pakete installiert werden.
3. **Konfigurationsdatei erstellen:** Eine *tsconfig*-Datei muss erstellt und konfiguriert werden.
4. **Integration in den Build-Prozess:** Die TypeScript-Kompilierung wird in den bestehenden Build-Prozess integriert.
5. **Linter einrichten:** Zur frühzeitigen Fehlererkennung wird ein Linter konfiguriert.
6. **Entwicklungsumgebung anpassen:** Entwicklungsumgebungen müssen auf TypeScript umgestellt werden.
7. (Optional) **API-Typen generieren:** Falls interne oder externe APIs vom Projekt benutzt werden, müssen die entsprechenden Typen generiert werden.
8. (Optional) **Migrationstools verwenden:** Bei umfangreichen Projekten kann der Einsatz unterstützender Tools den manuellen Aufwand deutlich verringern.
9. **Migration beginnen:** Idealerweise beginnt die Migration mit zentralen Modulen wie Hilfsfunktionen oder Basisklassen.

3.3 Manuelle Migration

Bei einer rein manuellen Migration wird der bestehende JavaScript-Code ohne automatisierte Unterstützung schrittweise oder vollständig in TypeScript umgewandelt. In der Praxis bedeutet dies insbesondere, Dateien umzubenennen, explizite Typannotationen für Variablen und Funktionen zu ergänzen sowie zentrale Datenstrukturen durch geeignete Typdefinitionen zu modellieren. Zudem müssen implizite Annahmen über Datenformen identifiziert und gegebenenfalls angepasst werden. Dementsprechend gibt es verschiedene Herangehensweisen an die Migration.

Eine mögliche Strategie ist der *Top-down*-Ansatz, bei dem zunächst zentrale Module, Schnittstellen oder Basiskomponenten typisiert werden. Diese dienen anschließend als Grundlage für die Migration abhängiger Komponenten. Alternativ kann der *Bottom-up*-Ansatz gewählt werden, bei dem zunächst kleinere, unabhängige Module migriert werden, bevor komplexere Komponenten angepasst werden.

Unabhängig von der gewählten Vorgehensweise ist der manuelle Aufwand der Migration in der Regel erheblich. Der Zeitbedarf lässt sich nur schwer abschätzen und hängt stark von der Codequalität, der Testabdeckung sowie der Erfahrung des Entwicklers mit TypeScript ab, weshalb in dieser Arbeit der Ansatz einer manuellen Migration nicht verfolgt wird.

3.4 Automatisierte Migrationstools

Aufgrund der Probleme einer manuellen Migration gibt es verschiedene Werkzeuge, die den Übergang auf TypeScript automatisiert unterstützen. Diese Tools übernehmen Aufgaben wie das Umbenennen von Dateien, das Generieren erster Typannotationen oder das Einfügen von Platzhaltern für noch unbestimmte Typen.

Ziel solcher Werkzeuge ist es, den initialen Migrationsaufwand zu reduzieren und eine strukturelle Grundlage für die weitere manuelle Verfeinerung der Typisierung zu schaffen. Im Folgenden werden ausgewählte Migrationstools vorgestellt und hinsichtlich ihrer Eigenschaften sowie ihrer Anwendbarkeit auf das Projekt BenchExec untersucht.

3.4.1 ts-migrate

ts-migrate[1] ist ein von Airbnb entwickeltes Tool zur automatischen Migration von JavaScript-Code auf TypeScript. Der Fokus liegt auf der schnellen Überführung eines bestehenden Projekts in einen kompilierbaren Zustand. Eine vollständige und präzise Typisierung wird jedoch nicht angestrebt und erfordert in der Regel manuelle Nacharbeit. Dank der Struktur des Tools ist außerdem eine flexible Anpassung der Migrationsprozesse über die entsprechenden Plugins möglich. Es ist als ein einzelnes Repository mit drei Packages organisiert, die jeweils unterschiedliche Funktionen übernehmen:

1. **ts-migrate:** Verantwortlich für die Nutzerinteraktion.
2. **ts-migrate-server:** Zentrale Komponente für Migrationsprozesse.
3. **ts-migrate-plugins:** Ermöglicht die Anpassung der Tool-Konfiguration an spezifische Anforderungen.

Aktuell stehen zwei Hauptkonfigurationen für die Anwendung zur Verfügung: `migration` konvertiert JavaScript-Code in TypeScript und `reignore` fügt Unterdrückungskommentare hinzu, um die Kompilierbarkeit des Projekts sicherzustellen [13].

Zu den wesentlichen Merkmalen von ts-migrate gehört unter anderem, dass eine Migration sowohl für das gesamte Projekt als auch für einzelne Dateien oder Verzeichnisse durchgeführt werden kann. Dabei versucht das Tool automatisch die richtigen Typen zu finden. Falls dies nicht möglich ist, wird der Typ `any` verwendet. Fehler, die dabei auftreten könnten, werden mit `@ts-expect-error` markiert. Darüber hinaus bietet ts-migrate explizite Unterstützung für React-Komponenten.

3.4.2 TypeStat

TypeStat[2] ist ein Open-Source-Tool von Joshua K. Goldberg. Zu den wichtigsten Funktionen gehören die Dateikonvertierung sowie das Hinzufügen von Typenannotationen zu den neu erzeugten Dateien. Außerdem kann das Programm Verstöße gegen Compiler-Optionen wie `--noImplicitAny` automatisch beheben. Dabei verändert TypeStat ausschließlich die Typinformationen durch das Hinzufügen oder Entfernen von Typen ohne das Laufzeitverhalten des Codes zu beeinflussen. Im Gegensatz zu ts-migrate liegt der Schwerpunkt von TypeStat weniger auf der initialen Projektmigration als auf der nachträglichen Verbesserung und Vervollständigung von Typinformationen.

[1] `https://github.com/airbnb/ts-migrate`
[2] `https://github.com/JoshuaKGoldberg/TypeStat`

3.4.3 js-to-ts-converter

Das Tool js-to-ts-converter[3] wurde von Gregory Jacobs entwickelt. Bei der Anwendung des Tools werden zum einen die Dateiendungen von `.js` auf `.ts` geändert und zum anderen die entsprechenden Konfigurationsdateien hinzugefügt. Im Gegensatz zu ts-migrate und TypeStat wird allerdings keine automatische Typerkennung durchgeführt. Stattdessen konzentriert sich das Tool auf die strukturelle Vorbereitung des Projekts, sodass alle erforderlichen Typen durch `any` ersetzt werden. Dadurch wird zwar eine schnelle Kompilierbarkeit des Projekts ermöglicht, jedoch entsteht zunächst kein tatsächlicher Sicherheitsgewinn im Sinne einer statischen Typprüfung.

3.4.4 Typify

Ein weiteres Programm wurde von Shivam Trivedi entwickelt und in der DEV-Community unter dem Namen Typify[4] veröffentlicht. Der Fokus liegt insbesondere auf der automatischen Erkennung und der Generierung von Typdefinitionen. Zunächst wird eine Analyse der bestehenden Codebasis durchgeführt, anschließend werden die entsprechenden Konfigurationsdateien erstellt, sofern diese noch nicht vorhanden sind. Danach werden automatisch Typdefinitionen erstellt und unbekannte Typen durch `any` ersetzt. Im Vergleich zu den zuvor genannten Tools ist Typify weniger verbreitet und gilt vor allem als experimentelles Tool zur automatischen Typgenerierung.

3.4.5 Online-Services

Neben den oben genannten CLI-Anwendungen gibt es auch verschiedene Webanwendungen, die die automatische Konvertierung von JavaScript-Code in TypeScript-Code anbieten. Diese Tools sind primär auf die Konvertierung einzelner Codeausschnitte oder Dateien ausgelegt. Eine Integration in bestehende Build-Prozesse oder eine projektweite Migration wird in der Regel nicht unterstützt. Im Vergleich zu CLI-basierten Werkzeugen unterscheiden sie sich somit sowohl hinsichtlich ihrer technischen Tiefe als auch ihrer Integrationsfähigkeit in bestehende Entwicklungsumgebungen. Im Folgenden werden 3 unterschiedliche Webanwendungen vorgestellt.

[3] `https://github.com/gregjacobs/js-to-ts-converter`
[4] `https://www.npmjs.com/package/@sliderzz/typify`

JS2TS

Die Webanwendung JS2TS[5] bietet ein einfaches Tool zur Konvertierung von JavaScript- in TypeScript-Code. Der Fokus des Tools liegt auf einer schnellen, unkomplizierten Umwandlung einzelner Codeabschnitte ohne zusätzliche Installationen [9]. Nutzer können ihren bestehenden JavaScript-Code kopieren und in ein Textfeld einfügen. Durch einen Convert-Button wird der Prozess gestartet und die Nutzer erhalten automatisch eine TypeScript-Version ihres Codes. Der Online-Service nutzt im Hintergrund künstliche Intelligenz um den Vorgang zu optimieren.

CodeConvert

CodeConvert[6] stellt ein KI-basiertes Online-Tool zur automatischen Konvertierung von Code zwischen verschiedenen Programmiersprachen, unter anderem von JavaScript zu TypeScript, zur Verfügung. Auch hier liegt der Fokus auf der schnellen Konvertierung einzelner Dateien. Die Webanwendung ermöglicht es gespeicherte Dateien hochzuladen oder den Code in ein Textfeld einzugeben. Für die Konvertierung können zusätzlich auch bestimmte Vorgaben angegeben werden.

AI-Powered Code Convertor

Die Plattform Workik bietet mit dem AI Code Converter[7] die Möglichkeit, Code zwischen zwei Programmiersprachen zu konvertieren. Die Webanwendung ermöglicht mehrere Codeausschnitte oder Dateien gleichzeitig hochzuladen und zu konvertieren. Neben der Umwandlung von JavaScript in TypeScript unterstützt das Tool auch zahlreiche weitere Sprachen. Wie der Name schon sagt wird der Prozess durch maschinelles Lernen unterstützt.

3.4.6 Large Language Model (ChatGPT)

In den letzten Jahren haben Large Language Models (LLMs) zunehmend an Bedeutung für die automatische Codegenerierung und -transformation gewonnen. Durch ihre Fähigkeit, Quellcode zu analysieren und kontextabhängig zu verändern, eröffnen sie neue Ansätze für die Migration von JavaScript nach TypeScript.

Im Gegensatz zu spezialisierten Migrationstools verfolgen LLM-basierte Ansätze keinen fest definierten Migrationsprozess, sondern erzeugen Code auf

[5] `https://js2ts.com/`
[6] `https://www.codeconvert.ai/javascript-to-typescript-converter`
[7] `https://workik.com/ai-code-convertor`

Basis statistischer Sprachmodelle. Dies ermöglicht flexible und kontextabhängige Anpassungen, bringt jedoch auch Herausforderungen hinsichtlich Nachvollziehbarkeit, Reproduzierbarkeit und Konsistenz mit sich.

In dieser Arbeit wird exemplarisch ChatGPT[8] (Version 5.1) als Vertreter von LLM-basierten Ansätzen betrachtet. Die Auswahl von ChatGPT erfolgt aufgrund der großen Verbreitung und der aktuellen Relevanz, wodurch es in der Praxis häufig als System für codebezogene Aufgaben eingesetzt wird. Die genannten Einschränkungen werden bewusst in Kauf genommen und im Rahmen der Evaluation berücksichtigt.

3.5 Hybridstrategien

Neben klar abgegrenzten Migrationsstrategien lassen sich in der Praxis auch hybride Ansätze beobachten, wie die Studie [12] zeigt. Der Begriff Hybridstrategie bezeichnet die bewusste Kombination unterschiedlicher Migrationsansätze innerhalb eines Projekts, insbesondere hinsichtlich des Automatisierungsgrades.

Ein hybrider Ansatz kann beispielsweise darin bestehen, automatisierte Migrationstools nur für ausgewählte Teile des Projekts einzusetzen, während andere Komponenten vollständig manuell migriert oder dauerhaft in JavaScript belassen werden. Charakteristisch für solche Strategien ist, dass kein einheitlicher Migrationsansatz für das gesamte Projekt verfolgt wird, sondern dass unterschiedliche Vorgehensweisen parallel existieren.

Die im Rahmen dieser Arbeit eingesetzte toolgestützte Migration ist hiervon klar zu unterscheiden. Obwohl automatisierte Tools in der Praxis keine vollständige Migration ohne manuelle Nachbearbeitung ermöglichen, stellt diese Nacharbeit keine hybride Migrationsstrategie dar. Sie ist vielmehr ein notwendiger Bestandteil nahezu aller automatisierten Migrationsansätze und dient der Korrektur und Präzisierung der erzeugten Typinformationen.

Hybridstrategien werden in dieser Arbeit daher nicht als eigenständiger Migrationsansatz evaluiert, sondern als praxisnahe Sonderform bestehender Strategien eingeordnet. Der Fokus liegt damit auf einem klar abgegrenzten, toolgestützten Migrationsprozess.

[8] `https://chatgpt.com/`

Kapitel 4

Evaluationskriterien

In diesem Kapitel wird die Bewertungsmethodik zur Evaluation der untersuchten Migrationstools vorgestellt und die zugrunde liegenden Evaluationskriterien definiert. Ziel ist eine transparente und nachvollziehbare Bewertung, auf deren Basis die Eignung der einzelnen Tools für die Migration von BenchExec von JavaScript zu TypeScript beurteilt werden kann.

4.1 Bewertungsmethodik und Gewichtung

Zur Bewertung der Migrationstools wird eine gewichtete Bewertungsmatrix[1] verwendet, die eine strukturierte und vergleichbare Einordnung der Ergebnisse ermöglicht. Die Bewertung erfolgt in drei Hauptkategorien: Technische Korrektheit, Codequalität sowie Prozess- und Aufwandsaspekte.

Die Kategorie Technische Korrektheit umfasst insbesondere die Frage, inwieweit das erzeugte TypeScript semantisch korrekt ist und ohne zusätzliche Fehler kompiliert werden kann. Codequalität bewertet die strukturelle Qualität und die Wartbarkeit des generierten Codes. Prozess- und Aufwandsaspekte berücksichtigen hingegen Faktoren wie Installationsaufwand, Bedienbarkeit und Integrationsfähigkeit in bestehende Entwicklungsprozesse.

Da das Ziel der Migration eine langfristig wartbare und korrekte Codebasis ist, werden die Kategorien Technische Korrektheit und Codequalität jeweils mit 45% gewichtet. Prozess- und Aufwandsaspekte sind für die Durchführung der Migration relevant, wirken sich jedoch primär kurzfristig aus und erhalten daher ein geringeres Gewicht von 10%. Die konkrete Gewichtung ist in Tabelle 4.1 veranschaulicht.

[1] Die hier verwendete Struktur wurde durch einen Hinweis von Prof. Dr. Sven Strickroth und einem KI-gestützten Assistenzsystem (ChatGPT) angeregt. Die inhaltliche Ausgestaltung wurde vollständig eigenständig vorgenommen.

Kategorie	Gewichtung
Technische Korrektheit	45%
Codequalität	45%
Prozess- und Aufwandsaspekte	10%

Tabelle 4.1: Gewichtung der Hauptkategorien

Jedes einzelne Kriterium wird auf einer ordinalen Skala von 1 (sehr schlecht) bis 5 (sehr gut) bewertet. Die Bewertung erfolgt dabei anhand klar definierter Beobachtungsmerkmale um eine möglichst nachvollziehbare und konsistente Einordnung zu gewährleisten. Für jede Hauptkategorie wird anschließend der arithmetische Mittelwert der zugehörigen Kriterien berechnet und gemäß der festgelegten Gewichtung in die Gesamtbewertung einbezogen. Die Gesamtpunktzahl eines Tools ergibt sich dementsprechend wie folgt:

$$\text{Gesamtpunktzahl} = 0{,}45 \cdot \overline{x}_{\text{Korrektheit}} + 0{,}45 \cdot \overline{x}_{\text{Codequalität}} + 0{,}10 \cdot \overline{x}_{\text{Prozess}}$$

wobei $\overline{x}$ den Mittelwert der Kriterien innerhalb einer Kategorie bezeichnet.

4.2 Technische Korrektheit

4.2.1 Funktionalität

Die Funktionalität beschreibt, inwieweit der migrierte Code nach der Anwendung des Tools lauffähig ist. Bewertet wird dabei nicht ausschließlich ein vollständig funktionsfähiger Zustand, sondern auch, wie nah das Ergebnis an einem lauffähigen Zustand liegt. Berücksichtigt werden unter anderem, ob der Code kompiliert, ob zentrale Build-Schritte ausführbar sind und ob bestehende Tests grundsätzlich gestartet werden können.

Bewertungsschema:

- **1 Punkt:** Weder kompilierbar noch lauffähig
- **3 Punkte:** Nach manueller Nachbearbeitung grundsätzlich lauffähig
- **5 Punkte:** Ohne manuelle Eingriffe vollständig kompilierbar und lauffähig

4.2.2 Typisierungsgrad

Der Typisierungsgrad beschreibt, wie vollständig und präzise das jeweilige Tool Typinformationen ergänzt. Zur Bewertung werden unter anderem der Anteil unspezifischer Typen wie `any` oder `unknown`, die Häufigkeit von Compiler-Direktiven wie `@ts-expect-error` sowie das Vorhandensein expliziter Strukturdefinitionen berücksichtigt. Eine hohe statische Modellierung liegt vor, wenn zentrale Datenstrukturen klar typisiert sind und nur wenige unspezifische Typen verwendet werden.

Bewertungsschema:

- **1 Punkt:** Typisierung kaum vorhanden
- **3 Punkte:** Viele `any` und `unknown`, aber konsistent
- **5 Punkte:** Überwiegend präzise Typen

4.2.3 Manueller Nachbearbeitungsaufwand

Dieses Kriterium erfasst den manuellen Aufwand, der nach der automatischen Migration erforderlich ist, um einen technisch nutzbaren Zustand zu erreichen. Dazu zählen ausschließlich notwendige Anpassungen zur Behebung von Typfehlern und zur Wiederherstellung der Kompilierbarkeit des Codes. Weitergehende Maßnahmen zur qualitativen Verbesserung der Typgenauigkeit, zur Modellierung domänenspezifischer Typen oder zur strukturellen Überarbeitung des Codes werden in diesem Kriterium ausdrücklich nicht berücksichtigt, sondern fließen in die Bewertung der Typisierungs- und Codequalität ein. Der Aufwand wird qualitativ bewertet und anhand der Anzahl der notwendigen Codeänderungen abgeschätzt.

Bewertungsschema:

- **1 Punkt:** Sehr hoher Aufwand (> 150 Zeilen)
- **3 Punkte:** Mittlerer Aufwand (ca. 50–150 Zeilen)
- **5 Punkte:** Keine oder nur wenige Änderungen nötig (< 50 Zeilen)

4.3 Codequalität

4.3.1 Struktur und Abstraktion

Dieses Kriterium bewertet, ob das Tool sinnvolle Typstrukturen erzeugt und bestehende Strukturen des Projekts berücksichtigt. Dabei wird berücksichtigt, ob komplexere Datenstrukturen angemessen abstrahiert werden oder ob lediglich oberflächliche Typannotationen ergänzt werden. Ein qualitativ hochwertiges Ergebnis zeichnet sich dadurch aus, dass wiederkehrende Strukturmuster erkannt und in geeignete Typkonstrukte überführt werden, statt identischer Strukturen mehrfach lokal zu definieren. Negativ bewertet wird die Einführung unnötiger oder redundanter Typen

Bewertungsschema:

- **1 Punkt:** Keine erkennbare Struktur oder stark redundante Typen
- **3 Punkte:** Grundlegende Typstrukturen vorhanden
- **5 Punkte:** Klare, wiederverwendbare Typstrukturen (z. B. Interfaces)

4.3.2 Semantik und Benennung

Bewertet wird, ob die erzeugten Typ-, Variablen- und Funktionsnamen semantisch aussagekräftig und konsistent sind. Hierbei wird geprüft, ob die gewählten Bezeichnungen bestehende Namenskonventionen des Projekts aufgreifen und sich konsistent in die vorhandene Codebasis einfügen. Eine gute semantische Qualität liegt insbesondere dann vor, wenn neue Typ- oder Variablennamen die fachliche Bedeutung der zugrunde liegenden Daten widerspiegeln und nicht lediglich aus der Struktur des Ausgangscodes abgeleitet werden.

Bewertungsschema:

- **1 Punkt:** Kaum semantische Aussagekraft
- **3 Punkte:** Mischung aus guten und generischen Namen
- **5 Punkte:** Ausschließlich sinnvolle Namen

4.3.3 Nachvollziehbarkeit

Dieses Kriterium beschreibt, wie transparent und überprüfbar die durch das Tool erzeugten Änderungen sind. Bewertet werden insbesondere die Granularität und Struktur der erzeugten Änderungen, beispielsweise anhand der Anzahl und Größe der veränderten Codeabschnitte sowie – sofern vorhanden – der Commit-Struktur.

Bewertungsschema:

- **1 Punkt:** Umfangreiche Änderungen ohne klare Struktur
- **3 Punkte:** Größere Änderungen mit erhöhter Prüfkomplexität
- **5 Punkte:** Klar abgegrenzte, logisch nachvollziehbare Änderungen

4.4 Prozess- und Aufwandsaspekte

4.4.1 Zeitbedarf

Der Zeitbedarf beschreibt ausschließlich den zeitlichen Aufwand, der für die Ausführung des Migrationstools selbst anfällt. Berücksichtigt werden die Laufzeit des Tools sowie der unmittelbare Bedienaufwand, etwa durch notwendige Konfigurationsschritte. Die bewusste Trennung zwischen Tool-Laufzeit und manueller Nachbearbeitung dient der isolierten Bewertung der technischen Leistungsfähigkeit des jeweiligen Werkzeugs. Ziel dieses Kriteriums ist es, die reine Ausführungs- und Bedienkomplexität vergleichbar zu machen, unabhängig von der Qualität oder dem Nachbearbeitungsbedarf des erzeugten Codes. Der tatsächliche Gesamtaufwand einer Migration wird daher nicht vollständig durch dieses Kriterium abgebildet, sondern in der Gesamtevaluation kontextualisiert.

Bewertungsschema:

- **1 Punkt:** Sehr lange Tool-Laufzeit oder hoher Bedienaufwand
- **3 Punkte:** Überschaubare Tool-Laufzeit, geringer Bedienaufwand
- **5 Punkte:** Sehr kurze Tool-Laufzeit, keine Interaktion notwendig

4.4.2 Installationsaufwand

Dieses Kriterium beschreibt den Aufwand für die Einrichtung und Nutzung des Migrationstools selbst. Berücksichtigt werden zusätzliche Abhängigkeiten und toolspezifische Konfigurationsschritte. Projektweite Anpassungen, die bei jeder Migration auf TypeScript unabhängig vom Tool erforderlich sind, werden nicht einbezogen.

Bewertungsschema:

- **1 Punkt:** Komplexe Einrichtung mit tiefen Eingriffen
- **3 Punkte:** Mehrere Konfigurationsschritte
- **5 Punkte:** Installation ohne zusätzliche Konfiguration

4.4.3 Automatisierungsgrad

Der Automatisierungsgrad beschreibt, in welchem Umfang das Tool den Migrationsprozess eigenständig durchführt. Bewertet wird, ob zentrale Schritte wie Dateikonvertierung, Typinferenz und Fehlerbehandlung automatisiert werden oder ob häufige manuelle Eingriffe notwendig sind. Tools mit hohem Automatisierungsgrad ermöglichen eine weitgehend selbstständige Migration.

Bewertungsschema:

- **1 Punkt:** Kaum Automatisierung, Migration überwiegend manuell
- **3 Punkte:** Teilautomatisierte Migration mit manuellen Eingriffen
- **5 Punkte:** Weitgehend automatisierte Migration

Kapitel 5

Evaluation der Migrationstools

Das Ziel der Evaluation ist es, die Eignung der einzelnen Tools für die Migration des Projekts BenchExec (Version 3.31) [15] systematisch zu vergleichen. Dieses Kapitel konzentriert sich auf die wesentlichen Bewertungsergebnisse, während die vollständige Detailanalyse einschließlich exemplarischer Codeauszüge im Anhang dokumentiert ist.

Um eine hohe Vergleichbarkeit zwischen den Tools sicherzustellen, wurde die Evaluation anhand derselben beiden Referenzdateien durchgeführt. Beide Dateien sind repräsentativ für typische Strukturen des Projekts. Dabei handelt es sich um zwei React-Komponenten: eine funktionale Komponente (`FilterInputField.js`, 69 Codezeilen) sowie eine klassenbasierte Komponente (`FilterBox.js`, 185 Codezeilen). Diese Dateien wurden für jedes ausgewählte Tool migriert und anschließend anhand der definierten Bewertungskriterien analysiert.

Nicht evaluiert wurden die Online-Dienste CodeConvert und AI-Powered Code Converter. Beide Angebote sind methodisch mit JS2TS vergleichbar, da sie auf einer dateibasierten Copy-&-Paste-Konvertierung einzelner Dateien beruhen und keine projektweite Automatisierung oder reproduzierbare Toolausführung ermöglichen. Eine zusätzliche Evaluation hätte daher keinen wesentlichen Erkenntnisgewinn erbracht.

Neben der Betrachtung der automatisch erzeugten Migrationsergebnisse wurden auch die notwendigen manuellen Anpassungen dokumentiert, um die migrierten Dateien wieder in einen lauffähigen Zustand zu überführen. Die jeweils verwendeten Tool-Versionen, Migrationsbefehle sowie die daraus resultierenden Code-Stände sind im Reproduction Package [8] hinterlegt, sodass alle Schritte der Evaluation nachvollziehbar und reproduzierbar sind.

5.1 Evaluation des Tools ts-migrate

5.1.1 Durchführung der Migration

Die Migration wurde mit Version 0.1.35 von ts-migrate durchgeführt. Das Tool wurde im Verzeichnis `benchexec/tablegenerator/react-table` angewendet und über die folgenden Befehle ausgeführt:

```
npx ts-migrate rename .

npx ts-migrate migrate .
```

Nach Abschluss der automatisierten Migration lagen beide Referenzdateien sowie sämtliche weiteren JavaScript-Dateien des Verzeichnisses als `.ts`/`.tsx`-Dateien vor. Damit wurde für den betrachteten Projektbereich eine vollständige syntaktische Umstellung auf TypeScript erreicht.

5.1.2 Zwischenfazit

Die Bewertung (siehe Tabelle 5.1) zeigt, dass ts-migrate insbesondere hinsichtlich Zeit- und Installationsaufwand überzeugt. Die syntaktische Umstellung erfolgt schnell und zuverlässig. Gleichzeitig bleibt der inhaltliche Mehrwert der Migration begrenzt, da zentrale Typinformationen häufig durch generische Konstrukte oder Fehlerunterdrückungen ersetzt werden. Der erzeugte Code ist formal TypeScript-kompatibel, erfordert jedoch umfangreiche manuelle Nachbearbeitung, um einen qualitativ hochwertigen und langfristig wartbaren Zustand zu erreichen.

Insgesamt eignet sich ts-migrate daher primär als vorbereitender Schritt innerhalb einer mehrstufigen Migrationsstrategie, nicht jedoch als eigenständige Lösung für eine tiefgreifende TypeScript-Migration.

Kategorie	**Bewertung**
Technische Korrektheit	2,00
Codequalität	2,33
Prozess- und Aufwandsaspekte	4,33
Gesamtbewertung	**2,38**

Tabelle 5.1: Zusammenfassende Bewertung von ts-migrate

5.2 Evaluation des Tools TypeStat

5.2.1 Durchführung der Migration

Die Migration wurde mit Version 0.8.18 von TypeStat durchgeführt. Das Tool wurde zunächst im Verzeichnis `benchexec/tablegenerator/react-table` über den Befehl

```
npx typestat
```

ausgeführt. Dabei erzeugt TypeStat automatisch eine Konfigurationsdatei, deren Inhalt von den während der Ausführung gewählten Einstellungen abhängt. Unter Windows war eine manuelle Anpassung dieser Konfigurationsdatei erforderlich, da die zu analysierenden Verzeichnisse nicht korrekt erkannt wurden. Anschließend wurde die eigentliche Migration mithilfe der angepassten Konfiguration über den Befehl

```
npx typestat --config typestat.json
```

durchgeführt. Am Ende lagen die beiden Referenzdateien als `.tsx`-Dateien vor.

5.2.2 Zwischenfazit

Die Bewertung (siehe Tabelle 5.2) zeigt, dass TypeStat im Vergleich zu rein syntaktischen Konvertierungswerkzeugen einen höheren inhaltlichen Mehrwert bietet. Insbesondere die automatisch generierten Interfaces für Eigenschaften schaffen eine strukturierte Ausgangsbasis für die weitere Migration.

Gleichzeitig bleiben die erzeugten Typen teilweise ungenau oder semantisch unpassend, sodass eine manuelle Nachbearbeitung erforderlich ist, um einen stabilen und konsistenten TypeScript-Zustand zu erreichen.

Insgesamt eignet sich TypeStat eher als unterstützendes Werkzeug für eine manuelle Migration, bei der automatisiert erste Typstrukturen erzeugt und anschließend gezielt verfeinert werden. Eine vollständig automatisierte Migration kann damit nicht erreicht werden.

Kategorie	**Bewertung**
Technische Korrektheit	3,00
Codequalität	3,00
Prozess- und Aufwandsaspekte	3,33
Gesamtbewertung	**3,03**

Tabelle 5.2: Zusammenfassende Bewertung von TypeStat

5.3 Evaluation des Tools js-to-ts-converter

5.3.1 Durchführung der Migration

Die Migration wurde mit Version 0.18.2 von js-to-ts-converter durchgeführt. Das Tool wurde im Verzeichnis `benchexec/tablegenerator/react-table` über folgenden Befehl ausgeführt:

```
npx js-to-ts-converter . --exclude node_modules --exclude build
```

Die Parameter `--exclude node_modules` und `--exclude build` wurden verwendet, um die zu analysierende Dateimenge auf den relevanten Quellcode zu begrenzen. Ohne diese Ausschlüsse konnte die Tool-Ausführung nicht erfolgreich abgeschlossen werden. Nach Abschluss der Konvertierung lagen alle Dateien des Projektbereichs als `.ts`/`.tsx`-Dateien vor.

5.3.2 Zwischenfazit

Die Bewertung (siehe Tabelle 5.3) verdeutlicht, dass js-to-ts-converter insbesondere hinsichtlich der schnellen und automatisierten syntaktischen Umstellung überzeugt. Der Installations- und Ausführungsaufwand ist gering.

In Bezug auf die inhaltliche Qualität des erzeugten Codes bleibt das Tool jedoch deutlich hinter den Anforderungen einer nachhaltigen Migration zurück. Zentrale Typinformationen werden nicht explizit modelliert, sondern häufig durch generische `any`-Deklarationen ersetzt. Der resultierende Code ist zwar formal TypeScript-kompatibel, bietet jedoch kaum zusätzlichen strukturellen oder semantischen Mehrwert gegenüber der ursprünglichen JavaScript-Version.

Insgesamt eignet sich js-to-ts-converter daher primär als mechanisches Umstellungstool zur schnellen Dateikonvertierung, jedoch nicht als eigenständige Lösung für eine qualitativ hochwertige TypeScript-Migration.

Kategorie	**Bewertung**
Technische Korrektheit	2,33
Codequalität	2,00
Prozess- und Aufwandsaspekte	3,66
Gesamtbewertung	**2,32**

Tabelle 5.3: Zusammenfassende Bewertung von js-to-ts-converter

5.4 Evaluation des Tools Typify

5.4.1 Durchführung der Migration

Die Migration wurde mit Version 1.3.2 von Typify durchgeführt. Das Tool wurde jeweils gezielt über den folgenden Befehl auf einzelne Dateien angewendet:

```
npx @sliderzz/typify <pfad>
```

Typify erzeugt für React-Komponenten neue Dateien mit der Endung `.ts`. Für die Integration in das Projekt ist anschließend eine manuelle Umbenennung in `.tsx` erforderlich. Zusätzlich werden die ursprünglichen `.js`-Dateien entfernt, um doppelte Modulvarianten zu vermeiden.

5.4.2 Zwischenfazit

Die Bewertung (siehe Tabelle 5.4) zeigt, dass Typify eine schnelle, dateibasierte Umwandlung einzelner JavaScript-Dateien ermöglicht und damit für einfache oder isolierte Dateien einen pragmatischen Einstieg in eine TypeScript-Migration bietet.

Für eine strukturierte oder projektweite Migration ist das Tool jedoch nur eingeschränkt geeignet. Die fehlende Unterstützung für JSX-haltige Dateien, der ausschließlich dateibasierte Arbeitsmodus sowie der geringe Typisierungsgrad führen dazu, dass zentrale Schritte der Migration weiterhin manuell erfolgen müssen. Eine konsistente und projektübergreifende Typstruktur entsteht dadurch nicht.

Insgesamt eignet sich Typify daher eher als punktuelles Hilfsmittel für einzelne Dateien. Es bietet keine tragfähige Lösung für eine systematische Migration größerer React-Projekte.

Kategorie	**Bewertung**
Technische Korrektheit	2.33
Codequalität	2.33
Prozess- und Aufwandsaspekte	3.33
Gesamtbewertung	**2,43**

Tabelle 5.4: Zusammenfassende Bewertung von Typify

5.5 Evaluation des Online-Services JS2TS

5.5.1 Durchführung der Migration

Die Migration erfolgte über die Webanwendung JS2TS (Stand: 07.12.2025). Dabei wird der Quellcode einzelner Dateien manuell in eine browserbasierte Oberfläche eingefügt. Der von JS2TS erzeugte TypeScript-Code wird anschließend aus der Webanwendung kopiert, in das Projekt übernommen und manuell als `.ts` bzw. `.tsx` gespeichert. Eine automatische Anpassung von Konfigurationsdateien oder Build-Prozessen findet im Rahmen der Webanwendung nicht statt. Die Integration der konvertierten Dateien in das Projekt erfolgt daher getrennt vom eigentlichen Transformationsprozess.

5.5.2 Zwischenfazit

Die Bewertung (siehe Tabelle 5.5) zeigt, dass JS2TS bei einzelnen Dateien qualitativ durchaus überzeugende TypeScript-Ergebnisse liefern kann, insbesondere bei funktionalen React-Komponenten. In diesen Fällen entstehen teilweise strukturierte und semantisch sinnvolle Typannotationen.

Die Qualität der erzeugten Ergebnisse ist jedoch inkonsistent und bei komplexeren, klassenbasierten Komponenten deutlich eingeschränkt. Hinzu kommt der vollständig manuelle und nicht reproduzierbare Migrationsprozess, der einen sehr geringen Automatisierungsgrad aufweist.

Insgesamt eignet sich JS2TS daher lediglich als punktuelles Hilfsmittel für einzelne Dateien, nicht jedoch als belastbares Werkzeug für eine systematische oder projektweite Migration.

Kategorie	**Bewertung**
Technische Korrektheit	3,66
Codequalität	2,00
Prozess- und Aufwandsaspekte	2,66
Gesamtbewertung	**2,82**

Tabelle 5.5: Zusammenfassende Bewertung von JS2TS

5.6 Evaluation des LLMs ChatGPT

5.6.1 Durchführung der Migration

Die Migration erfolgte durch Copy-&-Paste der jeweiligen Datei in das LLM und anschließendes Übernehmen des generierten TypeScript-Codes in das Projekt. Die Referenzdateien `FilterBox.tsx` und `FilterInputField.tsx` werden auf diese Weise migriert. Die Anwendung geschah dateibasiert. Die Generierung fand jeweils in einem einzelnen Durchlauf ohne iterative Nachfragen oder nachträgliche Verfeinerungen der Ausgabe statt. Dadurch wird sichergestellt, dass der erzeugte Code als einzigartiges Migrationsergebnis betrachtet wird und methodisch mit den nicht interaktiven Werkzeugen vergleichbar bleibt. Der verwendete Prompt ist im Reproduction Package [8] in der Datei `chatgpt.txt` dokumentiert.

5.6.2 Zwischenfazit

Die Bewertung (siehe Tabelle 5.6) zeigt, dass für die betrachteten Referenzdateien strukturierte und semantisch sinnvolle Typdefinitionen erzeugt werden. Der resultierende Code ist in der Regel kompilierbar und erfordert nur geringe manuelle Nachbearbeitung.

Gleichzeitig ist der Migrationsprozess vollständig manuell und nicht reproduzierbar. Eine projektweite oder automatisierte Anwendung ist nicht vorgesehen, sodass jeder Migrationsschritt einzeln durchgeführt werden muss.

Insgesamt eignet sich ChatGPT insbesondere für die qualitative Migration einzelner Dateien, bei denen eine strukturierte und inhaltlich fundierte TypeScript-Fassung erzeugt werden soll. Für eine systematische oder skalierbare Migration größerer Projekte ist der Ansatz hingegen nur eingeschränkt geeignet.

Kategorie	**Bewertung**
Technische Korrektheit	4,00
Codequalität	3,66
Prozess- und Aufwandsaspekte	3,00
Gesamtbewertung	**3,75**

Tabelle 5.6: Zusammenfassende Bewertung von ChatGPT

5.7 Gesamtfazit und Tool-Empfehlung

Die Tabelle 5.7 fasst die Ergebnisse der Evaluation übersichtlich zusammen und ermöglicht eine direkte Gegenüberstellung der untersuchten Migrationstools. Die Ergebnisse zeigen, dass kein Tool in allen Kategorien gleichermaßen überzeugt, sondern jeweils unterschiedliche Stärken und Schwächen aufweist.

	ts-migrate	TypeStat	js-to-ts-converter	Typify	JS2TS	ChatGPT
Technische Korrektheit (45%)						
Funktionalität	3	3	3	3	3	3
Typisierungsgrad	2	3	1	1	3	4
Manueller Nachbearbeitungsaufwand	1	3	3	3	5	5
Codequalität (45%)						
Struktur und Abstraktion	2	3	1	1	3	4
Semantik und Benennung	2	3	2	2	1	4
Nachvollziehbarkeit	3	3	3	4	2	3
Prozess- und Aufwandsaspekte (10%)						
Zeitbedarf	5	4	4	3	2	2
Installationsaufwand	5	3	4	5	5	5
Automatisierungsgrad	3	3	3	2	1	2
Gesamt	**2,38**	**3,03**	**2,32**	**2,43**	**2,82**	**3,75**

Tabelle 5.7: Bewertungsergebnisse der untersuchten Migrationstools

5.7.1 Einordnung der einzelnen Tool-Kategorien

CLI-Tools (ts-migrate, TypeStat, js-to-ts-converter, Typify)

Die regelbasierten Tools zeigen insgesamt solide, aber begrenzte Ergebnisse:

- **ts-migrate** überzeugt durch sehr geringen Zeit- und Installationsaufwand, erreicht jedoch nur niedrige Werte bei Typisierungsgrad, Struktur und Codequalität. Das Tool eignet sich primär als mechanische Vorstufe, jedoch nicht als qualitativ hochwertige Migration.
- **TypeStat** erzielt die insgesamt beste Bewertung unter den klassischen CLI-Tools. Es erzeugt erste Typstrukturen und Interfaces und bietet damit eine brauchbare Ausgangsbasis. Gleichzeitig bleiben Typen häufig ungenau, sodass ein erheblicher manueller Nachbearbeitungsaufwand erforderlich ist.

- **js-to-ts-converter** und **Typify** schneiden insbesondere bei Typisierungsgrad, Struktur und Abstraktion schwach ab. Beide Werkzeuge erzeugen zwar formal TypeScript-kompatiblen Code, nutzen die Möglichkeiten von TypeScript jedoch kaum aus und liefern damit nur einen begrenzten Mehrwert gegenüber dem ursprünglichen JavaScript-Code.

Insgesamt eignen sich diese Tools vor allem für grobe, syntaktische Migrationen, bei denen eine schnelle Umstellung im Vordergrund steht. Für ein komplexes, langlebiges Projekt wie BenchExec reichen die erzielten Ergebnisse jedoch nicht aus, um eine nachhaltige TypeScript-Codebasis zu schaffen.

Online-Services (JS2TS)

JS2TS nimmt eine Sonderrolle ein. Der Dienst zeigt bei einzelnen Dateien ein durchaus hohes Potenzial hinsichtlich Typisierungsgrad und Struktur, leidet jedoch unter mangelnder Zuverlässigkeit, fehlender Reproduzierbarkeit sowie syntaktischen Fehlern in komplexeren Komponenten. Die vergleichsweise hohe Bewertung beim manuellen Nachbearbeitungsaufwand erklärt sich daraus, dass zwar nur wenige Codezeilen geändert werden müssen, diese Änderungen jedoch zwingend erforderlich sind, um den Code überhaupt kompilierbar zu machen.

Für eine wissenschaftlich nachvollziehbare und reproduzierbare Migration eines größeren Projekts ist JS2TS daher nicht geeignet, kann jedoch punktuell als Inspirations- oder Vergleichswerkzeug dienen.

Large Language Model (ChatGPT)

Mit einer Gesamtpunktzahl von 3,75 erreicht ChatGPT deutlich das beste Ergebnis unter allen untersuchten Ansätzen. Besonders hervorzuheben sind:

- der hohe Typisierungsgrad,
- die klare Strukturierung und sinnvolle Abstraktion,
- die sehr geringe notwendige manuelle Nachbearbeitung.

Im Gegensatz zu den regelbasierten CLI-Tools gelingt es ChatGPT, kontextbezogene Typen zu erzeugen und React-spezifische Konzepte idiomatisch abzubilden. Damit wird ein Niveau erreicht, das deutlich näher an handgeschriebenem TypeScript liegt als bei allen anderen untersuchten Werkzeugen.

Die Schwächen von ChatGPT liegen klar im prozessualen Bereich: fehlende Automatisierung, keine projektweite Anwendung und eine nicht deterministisch reproduzierbare Ausführung. Diese Aspekte wirken sich jedoch aufgrund der geringen Gewichtung der Prozesskategorie nur begrenzt auf das Gesamtergebnis aus.

Zusätzlich ist bei der Nutzung von LLM-basierten Diensten wie ChatGPT die Datensicherheit zu berücksichtigen. Die Migration erfordert die Übermittlung des Quellcodes an einen externen Dienst, wodurch potenziell vertrauliche Informationen offengelegt werden könnten. Im vorliegenden Fall ist dieser Aspekt jedoch nicht ausschlaggebend, da es sich bei BenchExec um ein frei zugängliches Open-Source-Projekt handelt, dessen Quellcode öffentlich verfügbar ist. Für Projekte mit sensiblen oder nicht veröffentlichten Programmcode müsste dieser Faktor hingegen gesondert bewertet werden.

5.7.2 Tool-Empfehlung für die Migration von BenchExec

Auf Basis der Evaluation lässt sich eine klare Empfehlung treffen:

Für die Migration von BenchExec von JavaScript auf TypeScript ist ChatGPT das am besten geeignete Tool.

Die Gründe hierfür sind:

- BenchExec ist ein komplexes, langlebiges Projekt, bei dem Codequalität, Typensicherheit und Wartbarkeit entscheidend sind.
- Die Migration ist als einmaliger Vorgang geplant, sodass ein höherer manueller Bedienaufwand gegenüber der langfristigen Codequalität klar in den Hintergrund tritt.
- ChatGPT liefert bereits im initialen Migrationsschritt strukturierten, gut typisierten und weitgehend kompilierbaren Code, der nur minimal nachbearbeitet werden muss.

Die Empfehlung von ChatGPT gilt allerdings nur unter der Annahme, dass keine projektspezifischen Datenschutz- oder Compliance-Vorgaben einer externen Codeverarbeitung entgegenstehen.

Für Teilaspekte oder unterstützende Zwecke kann TypeStat ergänzend eingesetzt werden, etwa um projektweite Typinformationen zu extrahieren oder als Vergleichsbasis. Eine rein regelbasierte Migration mit ts-migrate, js-to-ts-converter oder Typify ist hingegen für BenchExec nicht zu empfehlen, da der resultierende Code die Vorteile von TypeScript nur unzureichend ausschöpft.

5.7.3 Abschließende Bewertung

Die Evaluation zeigt, dass ein KI-gestützter Ansatz in Form von ChatGPT gegenüber den untersuchten regelbasierten Migrationstools einen deutlichen qualitativen Vorteil bieten kann. Insbesondere bei der kontextsensitiven Typmodellierung, der strukturellen Aufbereitung des Codes und der Reduktion manueller Nacharbeiten erzielt ChatGPT Ergebnisse, die näher an einer handgeschriebenen TypeScript-Migration liegen als die der übrigen betrachteten Tools.

Für Projekte mit hoher Komplexität und langfristigem Wartungsanspruch stellt ChatGPT damit eine sehr geeignete Option dar, auch wenn prozedurale Einschränkungen wie fehlende Automatisierung und eingeschränkte Reproduzierbarkeit bestehen. Aussagen über die Leistungsfähigkeit anderer KI-basierter Migrationstools lassen sich aus dieser Arbeit jedoch nicht ableiten.

Kapitel 6

Umsetzung an BenchExec

Die Migration der Benutzeroberfläche von JavaScript zu TypeScript erfolgte auf Basis einer vorab definierten und dokumentierten Migrationsstrategie. Ziel war es, einen strukturierten, nachvollziehbaren und reproduzierbaren Migrationsprozess sicherzustellen. Zu diesem Zweck wurde eine separate Dokumentationsdatei (`MIGRATION.md`) erstellt, die Bestandteil des Reproduction Packages [8] ist und sämtliche strategischen und organisatorischen Entscheidungen der Migration festhält.

6.1 Einordnung in den Migrationsleitfaden

Die praktische Migration der Benutzeroberfläche von BenchExec orientierte sich grundsätzlich am in Kapitel 3.2 dargestellten allgemeinen Migrationsleitfaden nach Heiskanen [7]. Die dort beschriebenen neun Schritte dienten als strukturelle Orientierung für die Planung und Durchführung der Migration.

Eine vollständige Vorabprüfung sämtlicher verwendeter Bibliotheken auf TypeScript-Kompatibilität (Schritt 2 des Leitfadens) erfolgte hingegen nicht systematisch. Stattdessen wurden fehlende Typdefinitionen bedarfsorientiert während der Migration ergänzt, beispielsweise durch Installation entsprechender `@types`-Pakete.

Diese Abweichung zeigt, dass der theoretische Leitfaden als Orientierungsrahmen dient, in der praktischen Umsetzung jedoch flexibel an projektspezifische Gegebenheiten angepasst werden kann.

6.2 Migrationsstrategie

6.2.1 Wahl der Migrationsstrategie

Die in Kapitel 3.1 hergeleitete Entscheidung für die *Sudden*-Strategie bildet die strategische Grundlage der praktischen Umsetzung der Migration. Ziel dieser Strategie ist die vollständige Überführung der Benutzeroberfläche in TypeScript-basierten Programmcode, ohne einen langfristigen Mischbetrieb von JavaScript- und TypeScript-Dateien zu etablieren.

Obwohl die Migration als *Sudden*-Strategie klassifiziert wird, wurde technisch eine temporäre Übergangsphase zugelassen. Hierzu wurde in der Konfigurationsdatei (`tsconfig.json`) die Option `allowJs` aktiviert, um eine schrittweise Konvertierung einzelner Dateien innerhalb eines Projekts zu ermöglichen. Diese Übergangsphase diente ausschließlich der technischen Umsetzung der Migration und war nicht als dauerhafter Mischbetrieb vorgesehen.

Die Migrationsstrategie bezog sich primär auf den funktionsrelevanten Anwendungscode der Benutzeroberfläche, einschließlich der zugehörigen Worker-Komponenten sowie der bestehenden Testfälle. Unterstützende Build- und Hilfsskripte im Verzeichnis scripts/ wurden hingegen bewusst nicht migriert. Sie sind Teil der Infrastruktur, aber kein Bestandteil der Anwendung. Eine Typisierung dieser Skripte hätte die Einführung zusätzlicher Runtime-Transpiler zur direkten Ausführung von TypeScript oder einen separaten Kompilierungsschritt erfordert.

6.2.2 Bottom-up-Migrationsreihenfolge

Die Migration folgte einem *Bottom-up*-Ansatz. Dabei wurden zunächst grundlegende Utility-Module, anschließend zentrale Hilfsfunktionen sowie einfache Komponenten und schließlich komplexere, zusammengesetzte Komponenten migriert.

Diese Reihenfolge wurde gewählt, um Abhängigkeiten frühzeitig zu typisieren und gleichzeitig die Anzahl der auftretenden Typfehler zu reduzieren. Durch die Migration der unteren Ebenen konnten darauf aufbauende Komponenten schrittweise angepasst werden, ohne umfangreiche, projektweite Typkorrekturen vornehmen zu müssen.

Die Typisierung erfolgte dabei überwiegend lokal innerhalb der jeweiligen Module. Gemeinsame Typdefinitionen wurden nur dort eingeführt, wo sie zur Beschreibung von Modulgrenzen oder bestehenden Datenstrukturen passten. Dieses Vorgehen reduzierte die Komplexität der Migration und verhinderte eine vorzeitige Zentralisierung der Typdefinitionen.

6.2.3 Strukturierte Commit-Strategie

Ein zentrales Element der Migration war die strikte Trennung zwischen automatisierten und manuellen Änderungen in den Commits. Jede Dateimigration folgte einer standardisierten Commit-Sequenz mit einem bestimmten Präfix in der Commit-Beschreibung:

1. `rename:` Umbenennung der Datei
2. `tool:` Einfügen des generierten Tool-Outputs
3. `fix:` Behebung von Kompilierungsfehlern (falls erforderlich)
4. `refactor:` Manuelle Bereinigung und Typverbesserung

Diese Trennung der Commits ermöglicht eine präzise Nachvollziehbarkeit der durch die Werkzeuge erzeugten Änderungen. Insbesondere kann die Leistungsfähigkeit des eingesetzten Migrationstools isoliert betrachtet und anhand ihres erzeugten Outputs bewertet werden. Darüber hinaus entsteht eine klare Abgrenzung zwischen automatisch generiertem und nachträglich überarbeitetem Code. Dies erhöht die Transparenz des Migrationsprozesses und ermöglicht eine reproduzierbare Analyse der Migrationsqualität.

6.2.4 Einsatz von Migrationstools

Zur Unterstützung der Migration wurden automatisierte Werkzeuge eingesetzt, um die initiale Konvertierung von JavaScript-Dateien nach TypeScript zu erleichtern. Aufgrund der Größe und Struktur des Projekts erschien eine vollständig manuelle Migration ineffizient, insbesondere für repetitive Transformationsschritte wie die Einführung grundlegender Typannotationen oder die Anpassung von Funktionssignaturen.

Auf Grundlage der in Kapitel 5 dargestellten Evaluation wurde das Large Language Model ChatGPT als am besten geeignetes Migrationstool für BenchExec identifiziert. Die generierten Ergebnisse wurden nicht ungeprüft übernommen. Jeder Tool-Output wurde in einem separaten Commit festgehalten und anschließend systematisch überprüft. Eventuell auftretende Typfehler wurden behoben, unklare oder unspezifische Typannotationen präzisiert und strukturelle Anpassungen manuell vorgenommen. Erst nach dieser Überarbeitung wurde der Code als final betrachtet. Durch diese Vorgehensweise wurde sichergestellt, dass der Einsatz automatisierter Werkzeuge den Migrationsprozess unterstützt, ohne die Kontrolle über die Codequalität oder die Typstruktur zu verlieren.

6.2.5 Ziele und Abgrenzungen

Die Migration verfolgte das Ziel, die Benutzeroberfläche vollständig in einen typisierten Quellcode zu überführen. Dabei sollte eine möglichst umfassende statische Typprüfung implementiert werden, um potenzielle Fehlerquellen frühzeitig zu identifizieren und die langfristige Wartbarkeit des Codes zu verbessern. Insbesondere wurde Wert auf die Verwendung strenger Typprüfungen gelegt, um implizite `any`-Typen zu vermeiden.

Gleichzeitig war die Migration ausdrücklich nicht darauf ausgelegt, funktionale Änderungen am Verhalten der Anwendung vorzunehmen. Die bestehende Logik sowie das Laufzeitverhalten sollten unverändert bleiben. Ebenso waren keine Umstrukturierungen vorgesehen, sofern diese nicht unmittelbar zur Herstellung oder Verbesserung der Typsicherheit erforderlich waren.

Nicht Bestandteil der Migration waren zudem die Einführung neuer Funktionalitäten oder umfassende architektonische Umgestaltungen. Die Arbeiten beschränkten sich bewusst auf die Überführung in TypeScript. Dadurch wurde sichergestellt, dass die beobachtbaren Änderungen primär auf die Typisierung zurückzuführen sind und nicht durch funktionale Erweiterungen oder Designänderungen beeinflusst werden.

6.2.6 Grenzen der praktischen Umsetzbarkeit

Im Verlauf der Migration zeigte sich, dass vier zentrale Komponenten der Benutzeroberfläche zwar grundsätzlich ebenfalls nach TypeScript überführt werden könnten, eine vollständige und saubere Typisierung unter strengen Compiler-Prüfungen jedoch mit erheblichem Zusatzaufwand verbunden gewesen wäre. Betroffen waren insbesondere Komponenten auf oberster Ebene der Anwendung, in denen zahlreiche Datenstrukturen, Bibliotheken und Benutzerinteraktionen zusammenlaufen.

Diese Komponenten übernehmen primär Integrationslogik und verbinden verschiedene Teile der Anwendung, etwa Tabellenansichten, Filtermechanismen, Visualisierungen sowie den in der URL gespeicherten Zustand der Anwendung. Dabei werden viele Datenstrukturen dynamisch erzeugt und abhängig vom aktuellen Zustand der Benutzeroberfläche unterschiedlich weiterverarbeitet. Während solche Muster in JavaScript problemlos funktionieren, da die Typprüfung erst zur Laufzeit erfolgt, erfordert eine präzise statische Typisierung in TypeScript häufig eine explizite Modellierung der zugrunde liegenden Datenstrukturen sowie zusätzliche Validierungsschritte.

In mehreren der betrachteten Komponenten wären daher zusätzliche Zwischenrepräsentationen, Hilfstypen oder strukturelle Anpassungen der bestehenden Zustands- und Datenmodelle erforderlich gewesen, um eine konsisten-

te und wartbare Typisierung zu erreichen. Dies hätte die Migration von einer primär syntaktischen Überführung nach TypeScript hin zu einer weitergehenden Umstrukturierung der betroffenen Komponenten verschoben.

Da die Migration im Rahmen dieser Arbeit bewusst darauf ausgerichtet war keine größeren architektonischen Änderungen vorzunehmen, wurde die Migration an dieser Stelle abgebrochen. Der funktionsrelevante Großteil der Anwendung, insbesondere untergeordnete Komponenten und Utility-Dateien, konnte dennoch erfolgreich nach TypeScript überführt werden. Die verbleibenden Dateien markieren damit keinen grundsätzlich untypisierbaren Teil der Anwendung, sondern einen Bereich, dessen Überführung zusätzliche strukturelle Anpassungen erfordern würde und daher als eigenständiger Arbeitsschritt betrachtet werden kann.

6.3 Konkrete Migrationsschritte

Die praktische Umsetzung der Migration gliederte sich in zwei Phasen. Zunächst wurde die Projektumgebung so vorbereitet, dass TypeScript-Dateien im bestehenden Build- und Testsystem verarbeitet werden konnten. Darauf aufbauend erfolgte die eigentliche Umstellung schrittweise und dateibasiert, indem einzelne JavaScript-Dateien nacheinander auf TypeScript umgestellt wurden.

6.3.1 Projektweite Konfigurationsanpassungen

Vor Beginn der eigentlichen Dateimigration waren projektweite Anpassungen erforderlich, um TypeScript in die bestehende Toolchain zu integrieren. Hierzu wurde eine `tsconfig.json`-Datei erstellt und schrittweise so konfiguriert, dass eine strikte statische Typprüfung möglich war, ohne den bestehenden Build-Prozess strukturell zu verändern.

Für die Übergangsphase wurde die Option `allowJs` aktiviert, um JavaScript- und TypeScript-Dateien parallel im Projekt zu behalten. Zusätzlich wurde `resolveJsonModule` aktiviert, um bestehende JSON-Importe weiterhin typsicher verwenden zu können. Ergänzend wurden die Linting- und Formatierungsprüfungen so eingebunden, dass TypeScript-Dateien im gleichen Umfang wie JavaScript-Dateien geprüft werden.

6.3.2 Dateibasierte Durchführung der Migration

Im Anschluss an die vorbereitenden Konfigurationsanpassungen erfolgte die Migration schrittweise und dateibasiert. Jede JavaScript-Datei wurde einzeln in eine TypeScript-Datei umgewandelt. Der Migrationsprozess begann jeweils mit der Umbenennung der Datei von `.js` auf `.ts` bzw. `.tsx`. Anschließend wurde eine erste automatisierte Konvertierung durchgeführt, die als Ausgangspunkt für die weitere Typisierung diente. Darauf aufbauend wurden die vom TypeScript-Compiler gemeldeten Fehler systematisch behoben und die generierten Typen manuell präzisiert.

Diese Vorgehensweise stellte sicher, dass jede Datei isoliert migriert und nachvollziehbar überprüft werden konnte. Durch die schrittweise Umstellung war eine klare Zuordnung der auftretenden Typfehler zu einzelnen Dateien möglich. Dadurch konnten Probleme gezielt analysiert und behoben werden, ohne dass sich Fehlerquellen über mehrere Komponenten hinweg überlagerten.

6.3.3 Typisierungsarbeit im Quellcode

Im Zuge der Migration wurden insbesondere Funktionsparameter und Rückgabewerte explizit typisiert. Bei React-Komponenten wurden klar definierte Interfaces für Komponenten-Eigenschaften eingeführt sowie der jeweilige Komponenten-Zustand typisiert. Implizite `any`-Typen wurden systematisch entfernt und durch präzisere Typkonstrukte oder Utility-Typen ersetzt. Die Typdefinitionen wurden primär modulintern vorgenommen.

Im Verlauf der Migration zeigte sich wiederholt die Notwendigkeit, implizite Annahmen des ursprünglichen JavaScript-Codes explizit zu modellieren. Insbesondere mögliche `null`- und `undefined`-Zustände wurden durch geeignete Typannotationen sowie Guard-Konstrukte berücksichtigt. Dies führte zu einer insgesamt defensiveren Ausgestaltung einzelner Codepfade, ohne das funktionale Verhalten der Anwendung zu verändern. An mehreren Stellen wurden ergänzend Fallback-Mechanismen oder explizite Standardwerte eingesetzt, um die Anforderungen des TypeScript-Typsystems zu erfüllen und potenzielle Laufzeitfehler frühzeitig abzusichern.

6.4 Beispiele ausgewählter Migrationen

6.4.1 Typisierung von Komponenten-Schnittstellen

Ein zentrales Muster der Migration betrifft die explizite Typisierung von Komponenten-Schnittstellen. In der ursprünglichen JavaScript-Version werden die Properties einer React-Komponente zwar aufgeführt, ihre Typen bleiben jedoch implizit. Die erwarteten Eingaben ergeben sich ausschließlich aus der Verwendung der Eigenschaften innerhalb der Komponente sowie aus der korrekten Nutzung durch die Aufrufer.

Das nachfolgende Codebeispiel zeigt die ursprüngliche Signatur der Komponente `FilterInputFieldComponent`.

```
function FilterInputFieldComponent({
  id,
  setFilter,
  setCustomFilters,
  disableTaskText,
  focusedFilter,
  setFocusedFilter,
}) {
  // ...
}
```

Listing 6.1: Implizite Struktur der Komponenten-Eigenschaften

In dieser Form existiert keine statische Prüfung der Typen zwischen Komponente und ihren Aufrufern. Insbesondere bleibt unklar, welche Typen für die einzelnen Eigenschaften erwartet werden, ob bestimmte Werte optional sind und welche strukturellen Anforderungen an Funktionsparameter bestehen.

In der TypeScript-Version wird die implizite Struktur durch einen expliziten Prop-Typ ersetzt. Dabei werden bereits im Projekt definierte Typen (z.B. `FilterValueState`, `SetCustomFilters`) wiederverwendet, um Konsistenz mit bestehenden Typdefinitionen sicherzustellen und Redundanz zu vermeiden.

```
type FilterInputFieldProps = Readonly<{
  id: string;
  setFilter: FilterValueState;
  setCustomFilters: SetCustomFilters;
  disableTaskText: boolean;
  focusedFilter: string;
  setFocusedFilter: SetFocusedFilter;
}>;
```

```
function FilterInputFieldComponent({
  id,
  setFilter,
  setCustomFilters,
  disableTaskText,
  focusedFilter,
  setFocusedFilter,
}: FilterInputFieldProps) {
  // ...
}
```

Listing 6.2: Explizite Typisierung der Komponenten-Schnittstellen

Die funktionale Implementierung der Komponente bleibt unverändert. Die Migration beschränkt sich auf die statische Beschreibung der Schnittstelle. Durch die Verwendung von `Readonly` wird zusätzlich klargestellt, dass die Komponenten-Eigenschaften nicht mutiert werden dürfen, was der üblichen Verwendung von React-Eigenschaften entspricht.

Der unmittelbare Effekt dieser Typisierung zeigt sich an den Aufrufstellen der Komponente. Während in JavaScript inkonsistente oder strukturell unvollständige Eigenschaften erst zur Laufzeit auffallen würden, überprüft TypeScript nun bereits während der Kompilierung die Übereinstimmung zwischen übergebenen und erwarteten Werten. Dadurch entsteht eine explizite, statisch überprüfbare Verbindung zwischen den Komponenten.

Ein konkretes Beispiel ist die Ermittlung des `setFilter`-Wertes:

```
const id = filterProps.column.id;
const setFilter = props.filters.find(
  (filter) => filter.id === id,
);

return (
  <FilterInputField
    id={id}
    setFilter={setFilter}
    disableTaskText={disableTaskText}
    setCustomFilters={setCustomFilters}
    focusedFilter={focusedFilter}
    setFocusedFilter={setFocusedFilter}
  />
);
```

Listing 6.3: Implizite Existenzannahme eines Filters an der Aufrufstelle trotz potentiellen Rückgabewert undefined

In der JavaScript-Version wird implizit angenommen, dass stets ein passendes Element existiert. Der Rückgabewert von `find` wird ohne weitere Prüfung an die Komponente weitergereicht. In TypeScript hingegen erhält der Ausdruck den Typ `FilterValueState | undefined`, da das Fehlschlagen der Suche statisch modelliert wird. Die zuvor fehlerhafte, implizite Annahme eines stets vorhandenen Filters wird somit explizit sichtbar.

Um diesen Fehler zu beheben, muss entweder eine entsprechende Absicherung erfolgen oder die Annahme explizit dokumentiert werden, beispielsweise durch einen Guard. Die Migration zwingt somit zur formalen Auseinandersetzung mit bereits vorhandenen, jedoch zuvor nicht dokumentierten Annahmen im Datenfluss.

Dieses Beispiel verdeutlicht ein wiederkehrendes Migrationsmuster: Die Migration führt nicht zu einer fachlichen Änderung des Programmverhaltens, sondern zu einer expliziten Modellierung bereits vorhandener Annahmen über die Datenstruktur. Implizite Erwartungen werden in formale Typdefinitionen überführt, wodurch die Nachvollziehbarkeit der Komponenten-Schnittstellen erhöht und potenzielle Fehlerquellen früher sichtbar gemacht werden.

6.4.2 Präzisierung interner Zustände

Ein komplexeres Migrationsbeispiel betrifft die Behandlung numerischer Werte in der Komponente `FilterCard`. In der JavaScript-Version existieren für den numerischen Bereichsfilter faktisch zwei Wertepaare: `sliderMin` / `sliderMax` und `numericMin` / `numericMax`. Diese vier Zustandswerte werden jedoch nicht konsistent typisiert. Je nach Interaktionspfad erhalten sie formatierte Strings, numerische Werte oder `null`. Die implizite Struktur lässt sich vereinfacht wie folgt charakterisieren:

```
1 sliderMin: number | string
2 sliderMax: number | string
3 numericMin: number | string | null
4 numericMax: number | string | null
```

Listing 6.4: Implizite Typstruktur der Zustandsvariablen in der JavaScript-Version

Diese Vermischung ist nicht nur eine Eigenschaft der unterschiedlichen UI-Quellen (Slider vs. Eingabefeld), sondern wird durch die interne Weiterverarbeitung zusätzlich verstärkt: Mehrere Codepfade schreiben Werte in dieselben Zustandsfelder, wobei je nach Pfad Strings, Zahlen oder `null` verwendet werden. Die Methode `handleNumberChange` stellt dabei eine zentrale Stelle dar, an der diese unterschiedlichen Repräsentationen zusammengeführt werden:

```
handleNumberChange(min, max) {
  const newState = {};
  newState.sliderMin = Number(
    this.state.numericMin ?? this.state.sliderMin,
  );
  newState.sliderMax = Number(
    this.state.numericMax ?? this.state.sliderMax,
  );
  // ...

  this.setState(newState);
}
```

Listing 6.5: Umwandlung von string- und null-Werten zu Zahlen

In dieser Implementierung werden Zustandswerte, die zuvor als Strings oder `null` vorliegen konnten, explizit mittels `Number(...)` in numerische Werte überführt. Anschließend werden diese Werte wieder im Komponenten-Zustand gespeichert. Damit kann dasselbe Zustandsfeld je nach Interaktionspfad sowohl Strings als auch Zahlen enthalten. Diese implizite Union-Struktur wird im JavaScript-Code nicht dokumentiert, sondern ergibt sich lediglich aus der Laufzeitlogik.

In der TypeScript-Version wurde diese implizite Vermischung aufgelöst. Die Zustände werden nun explizit getrennt modelliert:

```
interface FilterCardState {
  // ...
  sliderMin: string;
  sliderMax: string;

  // Raw user input buffer
  inputMin: string;
  inputMax: string;
}
```

Listing 6.6: Explizite Typisierung und Trennung der Zustandswerte

Die Unterscheidung ist nun eindeutig: `sliderMin` / `sliderMax` repräsentieren formatierte, gültige Werte, `inputMin` / `inputMax` dienen als Eingabepuffer. Dementsprechend musste auch die Methode `handleNumberChange` etwas angepasst werden. Dadurch entfällt die implizite Mehrdeutigkeit der ursprünglichen Implementierung. Jeder Zustand besitzt nun eine klar definierte semantische Rolle und einen eindeutigen Typ.

In der JavaScript-Version war diese Trennung nicht klar ersichtlich. Obwohl die Bezeichnungen `numericMin` und `numericMax` suggerieren, dass es sich um numerische Werte handelt, konnten diese Felder tatsächlich auch Strings oder `null` enthalten. Gleichzeitig wurden `sliderMin` und `sliderMax` sowohl als formatierte Strings als auch als numerisch interpretierte Werte verwendet. Damit war aus der Benennung nicht ableitbar, welches Wertepaar für welche Phase der Verarbeitung (Eingabe, Formatierung, Vergleich) verantwortlich ist.

Dieses Beispiel zeigt exemplarisch, wie die Typisierung implizite Mehrdeutigkeiten sichtbar machte. Zwei Wertepaare, die zuvor implizit verschiedene Typen annehmen konnten, wurden in ein klar strukturiertes Zustandsmodell überführt. Die funktionale Logik blieb erhalten, jedoch wurde die semantische Trennung zwischen Anzeige- und Eingabewerten explizit modelliert.

Die Migration führte hier somit nicht nur zu formalen Typannotationen, sondern zu einer strukturellen Präzisierung des internen Zustandsmodells.

6.4.3 Präzisierung von Datenstrukturen

Ein weiteres Migrationsmuster zeigt sich in Utility-Funktionen zur Datenaufbereitung. Die Funktion `splitColumnsWithMeta` in der Datei `utils/stats` transformiert das Datenset von einem zeilenbasierten Layout in ein spaltenbasiertes Layout, um statistische Berechnungen pro Spalte zu ermöglichen. Dabei werden zusätzlich Metainformationen an die einzelnen Zellwerte angefügt.

In der JavaScript-Version wird mittels `for...in` iteriert:

```
const splitColumnsWithMeta =
  (tools) => (preppedRows, toolIdx) => {
    const out = [];
    for (const {
      row,
      categoryType,
      resultType,
    } of preppedRows) {
      for (const columnIdx in row) {
        const column = row[columnIdx].raw;
        const curr = out[columnIdx] || [];
        // ...
      }
    }
    return out;
  };
```

Listing 6.7: Verwendung von for...in zur Iteration über eine arraybasierte Datenstruktur

Obwohl `row` konzeptionell ein Array von Zellen darstellt, behandelt `for...in` das Array als Objekt. Der Index `columnIdx` besitzt daher zur Laufzeit den Typ `string`. Diese implizite Typisierung bleibt in JavaScript folgenlos, da Array-Zugriffe auch mit String-Indizes funktionieren.

In TypeScript wird diese implizite Annahme jedoch sichtbar: Für den Zugriff auf arraybasierte Strukturen wie `columns[cIdx]` wird ein `number`-Index erwartet. Die Verwendung von `for...in` würde daher eine explizite Konvertierung des Index erfordern.

In der TypeScript-Version wird stattdessen eine arraytypische Iteration verwendet:

```
const splitColumnsWithMeta =
  (tools: Tool[]) =>
  (
    preppedRows: PreppedRow[],
    toolIdx: number,
  ): SplitColumnItem[][] => {
    const out: SplitColumnItem[][] = [];
    for (const {
      row,
      categoryType,
      resultType,
    } of preppedRows) {
      row.forEach((rawCell, cIdx) => {
        const column = rawCell?.raw;
        const curr = out[cIdx] || [];
        // ...
      });
    }
    return out;
  };
```

Listing 6.8: Typkonsistente Array-Iteration mittels forEach in TypeScript

Durch die Verwendung von `forEach` wird der Index `cIdx` explizit als `number` modelliert und entspricht damit der tatsächlichen Struktur von `row` als Array. Die Transformation von zeilen- zu spaltenbasiertem Layout wird somit typkonsistent zur zugrunde liegenden Datenstruktur implementiert.

Dieses Beispiel zeigt, dass die Migration nicht nur Typannotationen ergänzt, sondern implizite Strukturannahmen im Code offenlegt. Die Iterationsform wird an die tatsächliche Datenstruktur angepasst, wodurch die Typkonsistenz verbessert und implizite String-Indizes vermieden werden.

6.5 Herausforderungen und Lösungen

Die Migration von JavaScript nach TypeScript erwies sich in der praktischen Umsetzung als komplexer als zunächst angenommen. Obwohl der Einsatz automatisierter Werkzeuge einen erheblichen Teil der initialen Konvertierungsarbeit übernahm, zeigte sich, dass der Gesamtaufwand nicht primär im Ergänzen von Typannotationen lag, sondern in der Überprüfung, Präzisierung und Absicherung der generierten Änderungen.

6.5.1 Automatisierte Migration und manuelle Reviews

Das eingesetzte Migrationstool erzeugte für die meisten Dateien eine syntaktisch korrekte und weitgehend lauffähige TypeScript-Version. Funktionsparameter, Rückgabewerte sowie einfache Objektstrukturen wurden häufig automatisch und konsistent typisiert. Auch grundlegende React-Komponenten konnten in vielen Fällen ohne größere strukturelle Eingriffe migriert werden.

Der eigentliche Mehraufwand entstand jedoch bei der anschließenden manuellen Überprüfung des Tool-Outputs. Jede generierte Datei musste daraufhin analysiert werden,

- ob eingeführte Typen semantisch korrekt waren,
- ob implizite Annahmen des ursprünglichen Codes angemessen modelliert wurden,
- ob Union Types oder unknown-Typen weiter präzisiert werden konnten,
- und ob die Typisierung langfristig wartbar und konsistent war.

Insbesondere bei komplexeren Datenstrukturen oder verschachtelten Objekten waren die vom Tool erzeugten Typen zwar formal korrekt, jedoch nicht immer optimal modelliert. In solchen Fällen war eine gezielte Nachbearbeitung erforderlich. Der zeitliche Aufwand verlagerte sich somit vom eigentlichen Schreiben von Code hin zur Bewertung und strukturellen Einordnung des generierten Codes.

Die Migration stellte sich daher weniger als rein technische Übersetzungsaufgabe dar, sondern vielmehr als kontinuierlicher Review- und Entscheidungsprozess.

Auf Ebene einzelner Dateien erwies sich der Einsatz des Large Language Models insgesamt als effektiv. Eine strukturelle Grenze zeigte sich jedoch bei projektübergreifenden Typabhängigkeiten. Während lokal definierte Funktionen und Komponenten häufig konsistent typisiert werden konnten, traten

Schwierigkeiten insbesondere dann auf, wenn Typdefinitionen in anderen Dateien oder Modulen definiert waren.

Da die Migration dateibasiert erfolgte, stand dem Modell jeweils nur ein begrenzter Kontext zur Verfügung. Funktionssignaturen, generische Typen oder gemeinsam genutzte Schnittstellen, die außerhalb der aktuell migrierten Datei definiert waren, konnten daher nicht zuverlässig in ihrer vollständigen Semantik erfasst werden. Dies führte teilweise zu unpräzisen Typannahmen, etwa zur Verwendung allgemeiner Union Types oder weniger spezifischer Platzhaltertypen.

Eine Verbesserung der Typqualität zeigte sich insbesondere dann, wenn relevante Typdefinitionen explizit gemeinsam mit der zu migrierenden Datei bereitgestellt wurden. Der Umfang des verfügbaren Kontexts beeinflusste somit unmittelbar die Präzision der erzeugten Typmodelle.

6.5.2 Typisierung impliziter Annahmen

Ein weiterer zentraler Aspekt war die explizite Modellierung impliziter Laufzeitannahmen. Der ursprüngliche JavaScript-Code enthielt zahlreiche Stellen, an denen von der Existenz bestimmter Eigenschaften oder bestimmter Datentypen ausgegangen wurde. Während JavaScript solche Annahmen zur Laufzeit toleriert, verlangt TypeScript eine explizite Beschreibung möglicher Zustände. Dies führte zu:

- der Einführung zusätzlicher Guard-Konstrukte,
- der Absicherung potenziell undefinierter Werte,
- der Modellierung alternativer Objektformen mittels Union Types,
- sowie der Ergänzung von Fallback-Mechanismen.

Diese Änderungen dienten ausschließlich der Herstellung von Typsicherheit und veränderten nicht das funktionale Verhalten der Anwendung. Dennoch stellten sie strukturelle Anpassungen dar, die über das bloße Ergänzen von Typannotationen hinausgingen.

6.5.3 Abweichung erwarteter vs. tatsächlicher Aufwand

Zu Beginn der Migration wurde angenommen, dass der Hauptaufwand bei der mechanischen Ergänzung fehlender Typen liegen würde. Der Einsatz eines leistungsfähigen Migrationstools schien diese Annahme zu bestätigen, da große Teile des Codes automatisiert konvertiert werden konnten.

In der praktischen Umsetzung zeigte sich jedoch, dass die entscheidende Phase nicht die Generierung, sondern die Validierung der Änderungen war. Insbesondere die Bewertung der Tool-Entscheidungen sowie die Einhaltung der strikten Compiler- und Linting-Regeln erforderten zusätzliche Zeit.

Die Migration erwies sich somit nicht als rein syntaktische Transformation, sondern als strukturelle Qualitätssicherung des bestehenden Codes. Die statische Typprüfung machte implizite Annahmen sichtbar, die zuvor unbemerkt geblieben waren, und zwang zu einer expliziten Modellierung dieser Zustände.

6.5.4 Build- und CI-bezogene Herausforderungen

Zusätzlich zur Codeebene traten Herausforderungen im Build- und Continuous-Integration-Prozess auf. Trotz Angleichung der Node-Versionen sowie des Einsatzes von Containern kam es zu minimalen Abweichungen in den erzeugten Build-Artefakten. Diese Differenzen waren funktional unproblematisch, konnten jedoch nicht vollständig aufgelöst werden.

Kapitel 7

Fazit und Ausblick

7.1 Zusammenfassung der Migration

Ziel dieser Arbeit war die Evaluation von Migrationswerkzeugen zur Bestimmung eines geeigneten Ansatzes zur Überführung des Programmcodes der Benutzeroberfläche von BenchExec von JavaScript nach TypeScript. Das identifizierte Werkzeug wurde anschließend im Rahmen einer praktischen Migration verwendet, wobei der Migrationsprozess strukturiert dokumentiert wurde.

Die Migration wurde weitgehend vollständig durchgeführt, sodass überwiegend ein einheitlicher Programmcode entstand. Ein dauerhafter Parallelbetrieb von JavaScript- und TypeScript-Dateien wurde dabei weitgehend vermieden, jedoch verblieben vier Dateien im JavaScript-Format.

Der Einsatz eines Large Language Models unterstützte die initiale Konvertierung erheblich. Ein Großteil der syntaktischen Umstellung sowie grundlegende Typannotationen konnten automatisiert erzeugt werden. Die generierten Dateien waren häufig bereits kompilierbar oder mit überschaubarem Anpassungsaufwand lauffähig.

Gleichzeitig zeigte sich jedoch, dass der zentrale Aufwand der Migration nicht im Ergänzen von Typannotationen lag, sondern in der Überprüfung und Präzisierung des generierten Codes. Die statische Typprüfung machte implizite Annahmen des ursprünglichen JavaScript-Codes sichtbar und erforderte deren explizite Modellierung. Insbesondere die Validierung automatisiert erzeugter Typstrukturen sowie die Modellierung impliziter Laufzeitannahmen erforderten zusätzliche Analysearbeit.

Insgesamt führte die Migration zu einer transparenteren Typstruktur und zu einer verbesserten strukturellen Nachvollziehbarkeit des Quellcodes von BenchExec.

7.2 Bewertung der gewählten Strategie

Die gewählte *Sudden*-Strategie erwies sich im betrachteten Projektkontext grundsätzlich als geeignet. Die klare Entscheidung für eine vollständige Umstellung auf Type-Script schuf ein eindeutiges Zielbild. Dadurch blieb der Migrationsprozess strukturell fokussiert und zeitlich begrenzt.

Auch die *Bottom-up*-Vorgehensweise erwies sich als sinnvoll. Durch die frühzeitige Typisierung von Utilities und Basisfunktionen konnten darauf aufbauende Komponenten schrittweise angepasst werden. Diese Reihenfolge erleichterte insbesondere die Migration komplexerer Komponenten, da übergreifende Datenstrukturen und Hilfsfunktionen bereits typisiert vorlagen.

In der praktischen Umsetzung zeigte sich jedoch, dass die ursprünglich angestrebte vollständige Migration im Rahmen dieser Arbeit nicht vollständig umgesetzt werden konnte. Vier zentrale Komponenten der Benutzeroberfläche verblieben in JavaScript, da ihre Migration weitergehende strukturelle Anpassungen erfordert hätte. Damit blieb das Ergebnis der Migration teilweise in einem Zustand, der einer *Incomplete*-Strategie entspricht.

Ein weiterer Bestandteil der gewählten Strategie war der Einsatz eines automatisierten Migrationstools. Das im Rahmen der Evaluation ausgewählte Large Language Model erwies sich in der praktischen Umsetzung als zweckmäßige Unterstützung. Insbesondere die initiale syntaktische Umstellung sowie grundlegende Typannotationen konnten automatisiert erzeugt werden, wodurch sich der initiale Konvertierungsaufwand deutlich reduzierte.

Gleichzeitig zeigte die praktische Anwendung, dass solche Werkzeuge primär als Assistenzsysteme zu verstehen sind. Sie können strukturelle Vorarbeit leisten, ersetzen jedoch keine sorgfältige Überprüfung und Präzisierung der erzeugten Typstrukturen. Ob alternative Werkzeuge im konkreten Projektkontext vergleichbare oder möglicherweise bessere Ergebnisse erzielt hätten, lässt sich auf Grundlage der praktischen Migration nicht abschließend beurteilen.

7.3 Grenzen der Arbeit

Die Ergebnisse dieser Arbeit sind projektspezifisch zu betrachten. Die Migration wurde exemplarisch an der Benutzeroberfläche von BenchExec umgesetzt. Rückschlüsse auf andere Projekte mit abweichender Architektur, Größe oder Codequalität sind nur eingeschränkt möglich.

Zudem erfolgte keine systematische quantitative Messung des Zeitaufwands einzelner Migrationsschritte. Die Bewertung des Werkzeugeinsatzes basiert daher primär auf qualitativer Analyse und praktischer Erfahrung.

Darüber hinaus wurde die Migration bewusst ohne tiefgreifende archi-

tektonische Umstrukturierungen durchgeführt. Dadurch konnten funktionale Änderungen vermieden werden. Gleichzeitig blieb jedoch Potenzial für weitergehende strukturelle Verbesserungen ungenutzt.

Neben den methodischen Grenzen des eingesetzten Migrationstools bestehen auch projektbezogene Grenzen der praktischen Umsetzbarkeit. Die Migration wurde unter der Prämisse durchgeführt, keine funktionalen Änderungen vorzunehmen und die bestehende Architektur weitgehend beizubehalten. Dadurch konnten strukturelle Inkonsistenzen der ursprünglichen Codebasis nur insoweit angepasst werden, wie es zur Herstellung von Typsicherheit erforderlich war.

Insbesondere projektübergreifende Typabhängigkeiten, historisch gewachsene Zustandsmodelle sowie externe Bibliotheken mit eingeschränkter oder ungenauer Typunterstützung setzten der vollständigen formalen Modellierung natürliche Grenzen. In einzelnen Fällen verblieben JavaScript-Dateien im Projekt, da deren Migration ohne weitergehende strukturelle Eingriffe nicht konsistent umsetzbar gewesen wäre.

7.4 Ausblick und weiterführende Arbeiten

Die vorliegende Migration stellt einen strukturellen Übergang zu TypeScript dar, jedoch keinen abschließenden Optimierungsschritt des Programmcodes. Mehrere weiterführende Maßnahmen bieten Potenzial für zukünftige Arbeiten.

Ein erster Ansatz betrifft die Zentralisierung und Vereinheitlichung der Typdefinitionen. Während der Migration wurde bewusst eine überwiegend modulinterne Typisierung gewählt, um die Komplexität zu reduzieren. In einem nächsten Schritt könnten wiederkehrende Datenstrukturen systematisch analysiert und in gemeinsame Domänenmodelle überführt werden. Dies würde die Konsistenz zwischen den Komponenten weiter erhöhen.

Ein weiterer Ansatz betrifft die Präzisierung von Komponenten-Schnittstellen. Im Rahmen der Migration wurden die Strukturen der Komponenten-Eigenschaften weitgehend unverändert übernommen, um die funktionale Stabilität sicherzustellen. Teilweise werden jedoch Parameter an Komponenten übergeben, die innerhalb dieser Komponenten nicht verwendet werden. Eine gezielte Nachbearbeitung könnte solche Schnittstellen konkretisieren, Verantwortlichkeiten klarer trennen und die Wartbarkeit weiter verbessern.

Die Erfahrungen dieser Migration lassen sich darüber hinaus auch auf ähnliche Projekte übertragen. Insbesondere zeigt sich, dass eine Migration von JavaScript nach TypeScript nicht ausschließlich als technische Umstellung der Programmiersprache verstanden werden sollte. Vielmehr handelt es sich um einen schrittweisen Prozess, in dem bestehende Datenstrukturen, implizi-

te Annahmen und Schnittstellen explizit modelliert werden müssen. Projekte mit gewachsener Codebasis sollten daher ausreichend Zeit für Analyse und Nachbearbeitung der automatisch erzeugten Typstrukturen einplanen.

Insgesamt zeigt die Arbeit, dass die Migration eines gewachsenen JavaScript-Projekts nach TypeScript technisch realisierbar ist, jedoch sorgfältige Planung, ein strukturiertes Vorgehen und eine kontinuierliche Qualitätskontrolle erfordert. Automatisierte Werkzeuge können den Prozess signifikant unterstützen, ersetzen jedoch nicht die fachliche Verantwortung für Architektur, Typmodellierung und Codequalität.

Anhang A

Detailbewertung ts-migrate

A.1 Funktionalität

Nach der automatisierten Migration mit ts-migrate liegen beide Referenzdateien formal als `.tsx`-Dateien vor, der resultierende Code ist jedoch nicht unmittelbar kompilierbar. In beiden Komponenten verhindern unvollständige Typinformationen eine erfolgreiche TypeScript-Kompilierung ohne zusätzliche Maßnahmen.Insbesondere Zugriffe auf Komponenten-Eigenschaften und den Zustand sowie Event-Handler führen zu wiederholten TypeScript-Fehlermeldungen, die vom Tool mithilfe von `@ts-expect-error` umfangreich unterdrückt werden. Erst nach manuellen Nachbesserungen konnte ein stabiler Zustand erreicht werden.

Bewertung: 3/5

A.2 Typisierungsgrad

Der von ts-migrate erzeugte Typisierungsgrad ist insgesamt gering. In beiden Referenzdateien werden zentrale Typinformationen nicht explizit modelliert, sondern entweder durch `any` ersetzt oder mithilfe von Fehlerunterdrückungen umgangen. Dies betrifft insbesondere die Typisierung von Eigenschaften und Event-Handlern sowie die Modellierung interner Zustands- und Hilfsstrukturen.

In der Klassenkomponente `FilterBox.tsx` werden weder die erwarteten Eigenschaften der Komponente noch deren interner Zustand durch konkrete Typen beschrieben. Stattdessen bleiben sowohl Klassenfelder als auch der Konstruktorparameter unspezifisch typisiert, wie im folgenden Codebeispiel zu sehen ist.

```
export default class FilterBox
  extends React.PureComponent
{
  listeners: any;
  resetFilterHook: any;

  constructor(props: any) {
    super(props);
    // ...
  }
}
```

Durch `props: any` bleibt die Schnittstelle der Komponenten-Eigenschaften unsichtbar. Durch `listeners: any` und `resetFilterHook: any` bleiben zentrale interne Invarianten und Funktionssignaturen unprüfbar.

Auch in der funktionalen Komponente `FilterInputField.tsx` werden zentrale Interaktionen nicht präzise typisiert. Ereignisparameter werden pauschal als `any` behandelt, obwohl ihre Struktur im weiteren Verlauf des Codes vorausgesetzt wird.

```
const onChange = (event: any) => {
  const newValue = event.target.value;
  // ...
}
```

Der resultierende Code ist damit zwar syntaktisch gültig, bietet gegenüber der ursprünglichen JavaScript-Version jedoch nur einen sehr begrenzten zusätzlichen Informationsgewinn.

Bewertung: 2/5

A.3 Manueller Nachbearbeitungsaufwand

Für die betrachteten Referenzdateien waren insgesamt **+141/−73 Codezeilen** an manuellen Anpassungen notwendig, um einen technisch nutzbaren und kompilierbaren Zustand herzustellen. Die Änderungen betreffen insbesondere das Ergänzen von Typen für die Eigenschaften und den Zustand in Klassen- und Funktionskomponenten, die Typisierung von Event-Handlern sowie Korrekturen an React-spezifischen Stellen (z.,B. Refs und Timer). Zusätzlich war es erforderlich, vom Tool eingefügte Compiler-Direktiven wie `@ts-expect-error` zu entfernen.

Bewertung: 1/5

A.4 Struktur und Abstraktion

Die durch ts-migrate erzeugte Struktur orientiert sich weitgehend an der bestehenden JavaScript-Struktur. Neue Abstraktionen oder wiederverwendbare Typmodelle werden nicht eingeführt. Stattdessen bleiben zentrale interne Datenstrukturen lediglich implizit im Code verankert. Dies zeigt sich exemplarisch an der Initialisierung des internen Zustands, bei der die erwartete Struktur ausschließlich aus der konkreten Zuweisung hervorgeht.

```
this.state = {
  filters:
    this.createFiltersFromReactTableStructure(filtered),
  idFilters: this.retrieveIdFilters(filtered),
};
```

Eine explizite Typbeschreibung dieser Struktur fehlt, sodass weder die erwarteten Eigenschaften noch deren Typen formal festgelegt sind. Die resultierende Struktur bleibt damit nur implizit nachvollziehbar und bietet keine zusätzliche Unterstützung bei der statischen Analyse oder bei späteren Änderungen.

Bewertung: 2/5

A.5 Semantik und Benennung

Die semantische Qualität der Typen und Bezeichner wird durch ts-migrate kaum verbessert. Typen sind entweder gar nicht oder nur generisch vorhanden, sodass kein zusätzlicher Bedeutungsgehalt entsteht. Die Verständlichkeit bleibt damit weitgehend auf dem Niveau der ursprünglichen Version.

Bewertung: 2/5

A.6 Nachvollziehbarkeit

Die durch ts-migrate erzeugten Änderungen sind überwiegend mechanisch (Umbenennung der Dateiendungen, minimale Anpassungen an die TypeScript-Syntax) und im Ergebnis nur eingeschränkt überprüfbar. Das Tool löst zahlreiche Typprobleme nicht durch Typmodellierung, sondern verschleiert sie durch Compiler-Direktiven (`@ts-expect-error`) sowie sehr grobe Typen.

Dies zeigt sich beispielsweise in der Klassenkomponente, in der Zugriffe auf die Eigenschaften und den Zustand nicht typisiert werden, sondern durch Fehlerunterdrückung stabilisiert werden.

```
function componentDidUpdate(prevProps: any) {
  // @ts-expect-error: Property 'filtered' ...
  if (!equals(prevProps.filtered, this.props.filtered)) {
    // ...
  }
}
```

Damit ist der Änderungsumfang zwar nachvollziehbar und lokalisierbar, die tatsächlichen Annahmen zu Datenstrukturen und Schnittstellen bleiben jedoch verborgen und sind nur durch zusätzliche Analyse rekonstruierbar. Insgesamt erhöht dies die Prüfkomplexität der Migrationsergebnisse.

Bewertung: 3/5

A.7 Zeitbedarf

Die Ausführung von ts-migrate erfolgt vollständig automatisiert mit nur einem Befehl und kurzer Laufzeit. Während der Ausführung ist keine Interaktion erforderlich und zusätzliche Konfigurationsschritte sind für die Tool-Ausführung nicht notwendig.

Bewertung: 5/5

A.8 Installationsaufwand

Der Installationsaufwand ist gering. ts-migrate kann direkt über `npx` ausgeführt werden und erfordert keine umfangreiche Vorabkonfiguration.

Bewertung: 5/5

A.9 Automatisierungsgrad

ts-migrate bietet einen hohen Automatisierungsgrad bei der syntaktischen Umstellung von JavaScript auf TypeScript. Die Typmodellierung sowie die Behebung komplexerer Typfehler erfolgen jedoch kaum automatisiert, sodass ein wesentlicher Teil der TypeScript-spezifischen Arbeit manuell bleibt.

Bewertung: 3/5

Anhang B

Detailbewertung TypeStat

B.1 Funktionalität

Nach der automatischen Ausführung von TypeStat ist der resultierende Code nicht direkt kompilierbar. In den generierten Typdefinitionen treten mehrfach zu enge oder inhaltlich unpassende Typannahmen auf, die nicht mit der tatsächlichen Nutzung der entsprechenden Variablen und Parameter im Code übereinstimmen. Zusätzlich verbleiben an mehreren Stellen vollständig untypisierte Funktionsparameter. In Kombination mit einer strikten TypeScript-Konfiguration führt dies zu Kompilierfehlern.

Bewertung: 3/5

B.2 Typisierungsgrad

TypeStat erzeugt explizite Interfaces für die Eigenschaften und setzt punktuelle Typannotationen. Der Typisierungsgrad bleibt jedoch insgesamt mittelmäßig, da viele Felder weiterhin über `any` abgebildet werden und zudem inhaltlich fehlerhafte Typen entstehen. Das folgende Beispiel zeigt eine unpassende Null-Typisierung.

```
interface FilterInputFieldComponentProps {
  // ...
  focusedFilter: null;
  setFocusedFilter: Dispatch<SetStateAction<null>>;
}
useEffect(() => {
  if (focusedFilter === elementId) {
    ref.current.focus();
  }
}, [focusedFilter, elementId]);
```

Im Code wird `focusedFilter` mit einem String (`elementId`) verglichen, wodurch die Typdefinition nicht zur tatsächlichen Nutzung passt.

Bewertung: 3/5

B.3 Manueller Nachbearbeitungsaufwand

Der manuelle Nachbearbeitungsaufwand ist als mittelhoch zu bewerten. Insgesamt waren **+43/−31 Codezeilen** an manuellen Anpassungen erforderlich, um die durch TypeStat erzeugten Typdefinitionen in einen technisch nutzbaren und kompilierbaren Zustand zu überführen. Die notwendigen Änderungen betreffen vor allem die Korrektur inhaltlich unpassender Typannotationen sowie das Ergänzen fehlender Typannotationen an Funktionsparametern.

Bewertung: 3/5

B.4 Struktur und Abstraktion

TypeStat erzeugt für beide Referenzdateien explizite Interfaces, die als Einstiegspunkt für eine formale Typisierung dienen. Damit wird grundsätzlich eine strukturelle Grundlage für TypeScript geschaffen. Gleichzeitig bleibt diese Struktur häufig oberflächlich, da viele Felder weiterhin mit `any` deklariert sind und insbesondere interne Klassenfelder sowie komplexere Zustands- und Datenstrukturen nicht konsequent typisiert werden. Wiederverwendbare oder projektweit konsistente Typmodelle entstehen nur eingeschränkt, sodass der strukturelle Mehrwert gegenüber der ursprünglichen JavaScript-Version begrenzt bleibt.

Bewertung: 3/5

B.5 Semantik und Benennung

Die von TypeStat eingeführten Interface- und Typnamen sind überwiegend verständlich und folgen einer konsistenten Benennung (z. B. `FilterBoxProps`). Dadurch wird die grundsätzliche Lesbarkeit des Codes verbessert. Die semantische Aussagekraft dieser Bezeichnungen wird jedoch häufig durch den Einsatz unspezifischer Typen wie `any` abgeschwächt, da die benannten Strukturen inhaltlich nicht näher modelliert sind. Insgesamt ergibt sich eine Mischung aus formal sinnvollen Bezeichnern und generischen Typisierungen mit begrenztem semantischem Mehrwert.

Bewertung: 3/5

B.6 Nachvollziehbarkeit

Die von TypeStat vorgenommenen Änderungen sind überwiegend lokal und gut auffindbar. Interfaces werden explizit eingefügt, und ergänzte Typannotationen lassen sich einzelnen Funktionen oder Komponenten klar zuordnen. Gleichzeitig erfordert das Ergebnis eine inhaltliche Prüfung der erzeugten Typen, da an mehreren Stellen unpassende oder zu enge Typisierungen auftreten (z.B. Null-Typisierungen). Der Änderungsumfang bleibt damit grundsätzlich gut prüfbar, ist jedoch nicht vollständig selbsterklärend.

Bewertung: 3/5

B.7 Zeitbedarf

Die eigentliche Tool-Laufzeit von TypeStat ist kurz. Der zeitliche Aufwand entsteht nicht durch lange Ausführungszeiten, sondern durch die notwendige mehrstufige Ausführung des Tools. Im Vergleich zu vollständig automatisierten Tools mit nur einer notwendigen Ausführung ist der Bedienaufwand leicht erhöht, bleibt jedoch insgesamt überschaubar.

Bewertung: 4/5

B.8 Installationsaufwand

TypeStat ist grundsätzlich direkt über `npx` nutzbar und erfordert keine separate Installation. Der Einrichtungsaufwand entsteht jedoch durch die toolspezifische Konfiguration über `typestat.json` und deren manuelle Anpassung (insbesondere unter Windows), bevor ein reproduzierbarer Lauf möglich ist.

Bewertung: 3/5

B.9 Automatisierungsgrad

TypeStat automatisiert die Erzeugung von Interfaces für die Eigenschaften und einzelnen Typannotationen, erzeugt jedoch teilweise unpassende oder zu ungenaue Typen und lässt viele Bereiche weiterhin unpräzise. Der Migrationsprozess ist damit teilautomatisiert: Das Tool schafft eine erste Basis, die jedoch durch gezielte manuelle Nachbearbeitung ergänzt und korrigiert werden muss.

Bewertung: 3/5

Anhang C

Detailbewertung js-to-ts-converter

C.1 Funktionalität

Der js-to-ts-converter ermöglicht eine grundlegende syntaktische Umstellung der betrachteten JavaScript-Dateien auf TypeScript. Die resultierenden Dateien liegen nach der Tool-Ausführung als `.ts/.tsx` vor, jedoch ist der erzeugte Code nicht unmittelbar kompilierbar. Ursachen hierfür sind insbesondere fehlende Typannotationen für Funktionsparameter sowie unvollständig typisierte Komponenten. Erst nach gezielten manuellen Korrekturen konnten die Referenzdateien in einen lauffähigen Zustand überführt werden.

Bewertung: 3/5

C.2 Typisierungsgrad

Der von js-to-ts-converter erzeugte Typisierungsgrad ist insgesamt sehr niedrig. In der Klassenkomponente werden weder Interfaces für die Eigenschaften noch für den Zustand eingeführt. Stattdessen ergänzt das Tool eine Reihe von Klassenfeldern mit dem Typ `any`, um TypeScript-Fehler zu unterdrücken:

```
export default class FilterBox
  extends React.PureComponent
{
  public listeners: any;
  public resetFilterHook: any;
  public state: any;
  public props: any;
}
```

Dieses Vorgehen umgeht die eigentliche Typisierung, statt sie zu modellieren. In der funktionalen Komponente werden darüber hinaus keinerlei Typanpassungen vorgenommen, sodass diese faktisch auf dem Stand der ursprünglichen JavaScript-Version verbleibt.

Bewertung: 1/5

C.3 Manueller Nachbearbeitungsaufwand

Um einen technisch nutzbaren und kompilierbaren Zustand zu erreichen, waren für die beiden Referenzdateien insgesamt **+43/−37 Codezeilen** an manuellen Anpassungen erforderlich. Diese Änderungen betreffen vor allem das nachträgliche Ergänzen elementarer Typannotationen für Funktionsparameter sowie Korrekturen an React-spezifischen Stellen (z. B. Refs und Timer). Der Aufwand ist damit nicht außergewöhnlich hoch, entsteht jedoch primär, weil das Tool selbst kaum verwertbare Typinformationen liefert und zentrale Probleme manuell behoben werden müssen.

Bewertung: 3/5

C.4 Struktur und Abstraktion

Der js-to-ts-converter erzeugt keine sinnvollen Typstrukturen oder wiederverwendbaren Abstraktionen. Die bestehende JavaScript-Struktur wird nahezu unverändert übernommen und lediglich durch `any`-Deklarationen für TypeScript kompatibel gemacht. Insbesondere das systematische Ergänzen von Klassenfeldern mit dem Typ `any` verhindert eine explizite Modellierung und führt dazu, dass strukturelle Eigenschaften der Daten lediglich implizit bleiben.

Bewertung: 1/5

C.5 Semantik und Benennung

Da das Tool kaum echte Typdefinitionen einführt, bleibt auch die semantische Qualität des Codes weitgehend unverändert. Die zusätzlich eingefügten `any`-Felder haben keinen eigenständigen Bedeutungsgehalt, sondern dienen primär dazu, den TypeScript-Compiler zu umgehen. Dadurch entsteht keine verbesserte Dokumentation der Datenstrukturen, und die Lesbarkeit sowie die Verständlichkeit des Codes profitieren nur minimal von der Migration.

Bewertung: 2/5

C.6 Nachvollziehbarkeit

Die durch js-to-ts-converter vorgenommenen Änderungen sind überwiegend mechanisch und daher grundsätzlich gut nachvollziehbar. Allerdings erschweren die eingefügten `any`-Deklarationen die inhaltliche Nachvollziehbarkeit erheblich, da zentrale Annahmen zu den Datenstrukturen nicht explizit gemacht werden. Zwar ist der Änderungsumfang überschaubar, jedoch bleibt unklar, welche Typen tatsächlich korrekt sind.

Bewertung: 3/5

C.7 Zeitbedarf

Die Tool-Ausführung selbst ist kurz und erfordert während des Laufs keine Interaktion. Allerdings sind die `--exclude`-Parameter notwendig, da sich das Tool ohne diese Einschränkung während der Ausführung aufhängt.

Bewertung: 4/5

C.8 Installationsaufwand

Die Nutzung ist grundsätzlich unkompliziert (direkt mit `npx`), und es sind keine zusätzlichen Konfigurationsdateien nötig. Die notwendige Verwendung der `--exclude`-Parameter ist ein zusätzlicher, aber kleiner Konfigurationsschritt.

Bewertung: 4/5

C.9 Automatisierungsgrad

Der Automatisierungsgrad bei der syntaktischen Umstellung von JavaScript auf TypeScript ist hoch, da Dateien projektweit automatisch konvertiert werden. Die eigentliche TypeScript-Migration im Sinne einer inhaltlichen Typmodellierung wird jedoch nur sehr eingeschränkt automatisiert: Zentrale Stellen bleiben ungenau typisiert, und React-spezifische Typen werden kaum abgeleitet. Dadurch entsteht zwar ein formal TypeScript-kompatibler Code, jedoch ohne substanzielle automatisierte Typanpassungen.

Bewertung: 3/5

Anhang D

Detailbewertung Typify

D.1 Funktionalität

Der von Typify erzeugte Code ist im Ergebniszustand zunächst nicht direkt in das React-Setup integrierbar, da React-Komponenten von Typify als `.ts`-Dateien generiert werden. Ohne manuelle Umbenennung nach `.tsx` kann JSX nicht korrekt verarbeitet werden, wodurch der Tool-Output keinen unmittelbar kompilierbaren Zustand darstellt. Nach der notwendigen Umbenennung der Dateiendung sowie weiteren minimalen Korrekturen kann ein technisch nutzbarer Zustand erreicht werden.

Bewertung: 3/5

D.2 Typisierungsgrad

Der Typisierungsgrad ist gering. Typify fügt zwar Typannotationen hinzu, diese bleiben jedoch überwiegend unspezifisch und liefern kaum zusätzliche Typinformationen. Besonders häufig treten `any` und grobe Platzhaltertypen wie `Record<string, any>` auf. Ein erstes Beispiel zeigt den Verlust struktureller Typinformationen bei Objekten.

```
componentDidUpdate(prevProps: Record<string, any>) {
  if (!equals(prevProps.filtered, this.props.filtered)) {
    // ...
  }
}
```

Es wird hier implizit vorausgesetzt, dass `prevProps` die Eigenschaft `filtered` besitzt. Diese strukturelle Annahme wird jedoch nicht typisiert, sodass TypeScript keine Unterstützung für fehlerhafte Zugriffe oder Refactorings leisten

kann. Ein weiteres Beispiel verdeutlicht den Verlust semantischer Typinformationen bei Funktionsparametern.

```
updateFilters(
  toolIdx: Record<string, any>,
  columnIdx: Record<string, any>,
  data: any
) {
  // ...
}
```

Die Parameter `toolIdx` und `columnIdx` werden im Code als numerische Indizes verwendet, sind jedoch als `Record<string, any>` typisiert. Diese Typisierung ist semantisch unpassend und vermittelt keinerlei Informationen über die tatsächliche Bedeutung der Parameter. Eine einfache Typisierung als `number` wäre hier naheliegend gewesen.

Bewertung: 1/5

D.3 Manueller Nachbearbeitungsaufwand

Für die betrachteten Referenzdateien waren insgesamt **+59/−28 Codezeilen** an manuellen Anpassungen erforderlich, um einen technisch nutzbaren und kompilierbaren Zustand zu erreichen. Der Aufwand ergibt sich dabei sowohl aus der Korrektur von TypeScript-Fehlern als auch aus projektspezifischen Anpassungen an React-Komponenten. Zusätzlich ist eine manuelle Umbenennung der durch Typify erzeugten `.ts`-Dateien in `.tsx` notwendig, da JSX andernfalls nicht korrekt verarbeitet werden kann. Dies erhöht den Nachbearbeitungsaufwand weiter.

Bewertung: 3/5

D.4 Struktur und Abstraktion

Typify führt keine erkennbaren wiederverwendbaren Typstrukturen ein. Weder Interfaces für Komponenten-Eigenschaften noch für den Zustand werden erzeugt. Stattdessen bleibt die Struktur des ursprünglichen JavaScript-Codes weitgehend unverändert. Die Ergänzungen beschränken sich auf grobe Typannotationen, die primär der formalen TypeScript-Kompatibilität dienen, ohne die strukturelle Qualität oder Modularität des Codes zu verbessern. Eine Abstraktion zentraler Konzepte oder eine Typstrukturierung fehlt.

Bewertung: 1/5

D.5 Semantik und Benennung

Die semantische Aussagekraft der eingefügten Typen ist gering. Typify verwendet überwiegend generische Platzhaltertypen wie `any` oder sehr allgemeine Konstrukte, die keinen zusätzlichen Bedeutungsgehalt haben. Dadurch bleibt unklar, welche inhaltlichen Annahmen über Datenstrukturen oder Funktionsparameter tatsächlich gelten.

Bewertung: 2/5

D.6 Nachvollziehbarkeit

Die durch Typify erzeugten Änderungen sind grundsätzlich gut nachvollziehbar, da das Tool dateibasiert arbeitet und hauptsächlich lokale Typannotationen ergänzt. Die Modifikationen lassen sich klar den jeweiligen Dateien zuordnen und sind in ihrem Umfang überschaubar. Allerdings wird die inhaltliche Nachvollziehbarkeit durch den hohen Anteil generischer Typen eingeschränkt, da die zugrunde liegenden Annahmen zu Datenstrukturen nicht explizit dokumentiert oder modelliert werden.

Bewertung: 4/5

D.7 Zeitbedarf

Die Ausführung von Typify ist pro Datei sehr kurz. Der zeitliche Aufwand entsteht jedoch durch den wiederholten manuellen Bedienprozess, da jede zu migrierende Datei einzeln über einen separaten Kommandozeilenaufruf verarbeitet werden muss. Mit zunehmender Anzahl an Dateien steigt dieser Aufwand entsprechend an. Zusätzlich fallen manuelle Nacharbeiten wie das Umbenennen der erzeugten Dateien sowie das Entfernen der ursprünglichen JavaScript-Dateien an.

Bewertung: 3/5

D.8 Installationsaufwand

Typify ist direkt über `npx` nutzbar und erfordert keine zusätzliche Konfiguration oder Anpassung bestehender Projektdateien. Das Tool kann ohne vorbereitende Setup-Schritte ausgeführt werden, wodurch der Installationsaufwand gering ist.

Bewertung: 5/5

D.9 Automatisierungsgrad

Typify automatisiert die syntaktische Umwandlung einzelner JavaScript-Dateien, arbeitet jedoch ausschließlich dateibasiert. Zudem erkennt das Tool JSX-haltige Dateien nicht automatisch, sodass React-Komponenten als `.ts`-Dateien erzeugt und manuell angepasst werden müssen. Wesentliche Schritte des Migrationsprozesses finden damit außerhalb der Automatisierung statt.

Bewertung: 2/5

Anhang E

Detailbewertung JS2TS

E.1 Funktionalität

Die Funktionalität ist insgesamt eingeschränkt. Während die funktionale Komponente in der generierten Version syntaktisch korrekt ist und eine plausible TypeScript-Integration aufweist, enthält die Klassenkomponente zahlreiche syntaktische Fehler (z. B. inkonsistente Bezeichner, ungültige JSX-Eigenschaften und fehlerhafte Methodennamen). Der migrierte Code ist daher zunächst nicht kompilierbar. Nach manueller Korrektur dieser rein syntaktischen Probleme kann jedoch ein technisch nutzbarer und lauffähiger Zustand erreicht werden, ohne dass eine grundlegende Überarbeitung der Komponentenlogik erforderlich ist.

Bewertung: 3/5

E.2 Typisierungsgrad

Der Typisierungsgrad des Outputs von JS2TS ist insgesamt uneinheitlich. In der funktionalen Referenzkomponente `FilterInputField.tsx` erzeugt der Dienst eine vergleichsweise idiomatische TypeScript-Fassung, etwa durch ein explizites Interface für die Eigenschaften, die Nutzung von `React.FC` sowie konkrete Event- und Ref-Typen. Dadurch erhöht sich die Typsicherheit dieser Datei.

In der klassenbasierten Komponente `FilterBox.tsx` sind zwar ebenfalls Typstrukturen vorhanden, deren tatsächlicher Nutzen jedoch gering bleibt. Zentrale Datenstrukturen werden überwiegend sehr grob typisiert, wie man im folgenden Beispiel an `any` und `Record<string, any>` sieht.

```
interface FilterBoxProps {
  filtered: any[];
  ids: Record<string, any>;
  // ...
  filterable: Array<{ name: string; columns: any[] }>;
  hiddenCols?: any[];
}
```

Obwohl formale Typdefinitionen existieren, beschreiben sie die verwendeten Datenstrukturen nur unzureichend. Wichtige Konzepte wie Filter, Spalten oder Zustandsobjekte bleiben unmodelliert. Zusätzlich wird die Verwertbarkeit der vorhandenen Typen durch inkonsistente Bezeichner und Methodennamen eingeschränkt, sodass Teile der Typisierung in der Praxis wirkungslos bleiben und bereits einfache Inkonsistenzen die Kompilierbarkeit verhindern.

Bewertung: 3/5

E.3 Manueller Nachbearbeitungsaufwand

Diese manuellen Änderungen betreffen ausschließlich die klassenbasierte Komponente. Die funktionale Komponente kann ohne zusätzliche Nachbearbeitung übernommen werden. Insgesamt waren **+19/−15 Codezeilen** an Änderungen notwendig. Der Aufwand beschränkt sich dabei im Wesentlichen auf das Beheben von Syntax- und Schreibfehlern (z. B. inkonsistente Bezeichner und Methodenaufrufe), die eine erfolgreiche Kompilierung zunächst verhindern.

Bewertung: 5/5

E.4 Struktur und Abstraktion

JS2TS erzeugt stellenweise sinnvolle Typstrukturen, etwa durch die Einführung von Interfaces für die Eigenschaften und in Ansätzen auch für den Zustand der Komponente. Diese strukturellen Ansätze sind jedoch nicht durchgängig konsistent. Insbesondere in `FilterBox.tsx` bleiben zentrale Datenstrukturen weiterhin ungenau typisiert, und inkonsistente Bezeichner sowie fehlerhafte Methodennamen schränken die Konsistenz und Wiederverwendbarkeit der erzeugten Typstrukturen erheblich ein. Die Strukturen können daher nur eingeschränkt und eher als Grundlage für eine weiterführende Typmodellierung genutzt werden.

Bewertung: 3/5

E.5 Semantik und Benennung

In der funktionalen Komponente `FilterInputField.tsx` sind Bezeichner und Typnamen überwiegend konsistent und verständlich. In der klassenbasierten Komponente `FilterBox.tsx` treten hingegen zahlreiche Tippfehler und inkonsistente Schreibweisen auf, die sowohl die Lesbarkeit als auch die funktionale Korrektheit des Codes beeinträchtigen.

Die inkonsistente Benennung betrifft nicht nur vereinzelte Randbereiche, sondern auch zentrale Methodenaufrufe und Zustandszugriffe innerhalb der Komponente. Dadurch entstehen nicht nur Lesbarkeitsprobleme, sondern auch potenzielle funktionale Fehlerquellen, da bereits geringfügige Abweichungen in der Schreibweise zu nicht auflösbaren Bezeichnern oder zu unerwartetem Verhalten führen können.

Der folgende Codeausschnitt illustriert exemplarisch mehrere solcher Inkonsistenzen, darunter vertauschte oder doppelte Buchstaben in Variablennamen sowie fehlerhafte Funktionsaufrufe.

```
updateFilters(
  toolIdx: number,
  columnIdx: number,
  data: any,
): void {
  const newFilters = [...this.state.filters];
  const idFilte = this.state.idFilters;
  // ...
  this.setState({ filters: newFilters });
  this.sendFilters({ filter: newFilters, idFilte });
}

updateIdFilters(data: any): void {
  // ...
  const newFIlter = /* ... */;

  this.setState({ idFilte: newFIlter });
  this.sendFiltters({
    filter: this.state.filters,
    idFilte: newFIlter,
  });
}
```

Die gezeigten Inkonsistenzen führen dazu, dass Variablen- und Methodennamen nicht eindeutig referenzierbar sind. Bereits geringfügige Abweichungen wie `idFilte` statt `idFilter` oder `sendFiltters` statt `sendFilters` können Kompilierfehler verursachen oder die statische Analyse beeinträchtigen. Da-

durch entsteht zusätzlicher manueller Prüf- und Korrekturaufwand, der den Nutzen der automatisierten Migration erheblich relativiert.

Bewertung: 1/5

E.6 Nachvollziehbarkeit

Die Nachvollziehbarkeit der durch JS2TS erzeugten Änderungen ist insgesamt begrenzt. Der Online-Service bietet keine reproduzierbare Tool-Ausführung. Die Migration erfolgt ausschließlich über einen manuellen Copy-&-Paste-Prozess. Dadurch entsteht kein eindeutig zuordenbarer Transformationsschritt. Zusätzlich wird die Überprüfbarkeit des Ergebnisses durch syntaktische und semantische Fehler erschwert, da vor einer Prüfung auf korrekte Typannotationen zunächst grundlegende Schreib- und Syntaxfehler behoben werden müssen.

Bewertung: 2/5

E.7 Zeitbedarf

Der Zeitbedarf bei JS2TS ergibt sich weniger aus der eigentlichen Konvertierung, die im Browser sehr schnell erfolgt, als aus dem vollständig manuellen Ablauf. Für jede Datei müssen der Quellcode manuell übertragen, der generierte Output zurückkopiert und lokal gespeichert werden. Dieser Prozess ist nicht automatisierbar und muss für jede Datei erneut durchgeführt werden.

Bewertung: 2/5

E.8 Installationsaufwand

Eine lokale Installation ist nicht erforderlich, da JS2TS als browserbasierter Online-Dienst genutzt wird. Es sind weder zusätzliche Abhängigkeiten noch projektspezifische Konfigurationsschritte notwendig.

Bewertung: 5/5

E.9 Automatisierungsgrad

Der Automatisierungsgrad ist sehr gering. JS2TS bietet weder eine projektweite Anwendung noch eine Integration in bestehende Build- oder CI-Prozesse. Die Migration erfolgt ausschließlich manuell pro Datei, ohne reproduzierbare oder wiederholbare Tool-Ausführung.

Bewertung: 1/5

Anhang F

Detailbewertung ChatGPT

F.1 Funktionalität

Der durch ChatGPT erzeugte Code ist nach manueller Übernahme und Umbenennung der Dateien grundsätzlich in das bestehende React-Projekt integrierbar. Die funktionale Komponente ist nach der Migration ohne weitere Korrekturen kompilierbar und lauffähig. JSX, Imports sowie die grundlegende Logik der Komponente sind konsistent umgesetzt. In der klassenbasierten Komponente treten hingegen vereinzelt funktionale und semantische Probleme auf, etwa im Umgang mit internen Zuständen. Diese führen zwar nicht zu grundlegenden Strukturbrüchen, verhindern jedoch einen unmittelbar fehlerfreien Einsatz. Nach gezielten manuellen Korrekturen lassen sich beide Komponenten in einen technisch nutzbaren und kompilierbaren Zustand überführen.

Bewertung: 3/5

F.2 Typisierungsgrad

Der Typisierungsgrad ist im Vergleich zu den übrigen betrachteten Werkzeugen hoch. Es werden explizite Interfaces für Komponenten-Eigenschaften eingeführt, und React- sowie Event-Typen werden überwiegend korrekt verwendet. In der funktionalen Referenzkomponente `FilterInputField.tsx` sind insbesondere Ref- und Event-Typen präzise modelliert.

```
const ref = useRef<HTMLInputElement | null>(null);
const [typingTimer, setTypingTimer] =
  useState<ReturnType<typeof setTimeout | null>>(null);
const [value, setValue] = useState<string>(
  initFilterValue,
);
```

```
const onChange = (
  event: ChangeEvent<HTMLInputElement>,
) => {
  const newValue = event.target.value;
  setValue(newValue);
  // ...
};
```

In der klassenbasierten Komponente `FilterBox.tsx` werden ebenfalls zentrale Typstrukturen eingeführt, etwa für Filterelemente und Spaltenbeschreibungen. Gleichzeitig zeigt sich hier die Grenze der automatischen Typmodellierung: Komplexe Datenflüsse werden weiterhin nur ungenau typisiert.

```
interface FilterBoxState {
  filters: ToolFilter[];
  idFilters: any[] | null | undefined;
}

function updateFilters(
  toolIdx: number,
  columnIdx: number,
  data: any,
) {
  const newFilters = [...this.state.filters];
  const idFilter = this.state.idFilters;
  // ...
}
```

Während grundlegende Strukturen vorhanden sind, bleiben zentrale Felder wie `idFilters` sowie Parameter wie `data` unspezifisch. Dadurch wird zwar die Kompilierbarkeit sichergestellt, eine vollständig Typmodellierung jedoch nicht erreicht. Insgesamt ergibt sich damit eine relativ hohe Typqualität.

Bewertung: 4/5

F.3 Manueller Nachbearbeitungsaufwand

Der manuelle Nachbearbeitungsaufwand ist sehr gering. Für beide Referenzdateien waren nur wenige gezielte Anpassungen erforderlich, um einen kompilierbaren Zustand zu erreichen. Änderungen betrafen ausschließlich die klassenbasierte Komponente. Insgesamt beschränkten sich die notwendigen Anpassungen auf **+7/−3 Codezeilen** und betrafen kleinere Typpräzisierungen sowie Integrationsdetails.

Bewertung: 5/5

F.4 Struktur und Abstraktion

Die Struktur des generierten Codes ist klar gegliedert und gut nachvollziehbar. Typdefinitionen sind vom eigentlichen Komponenten-Code getrennt angeordnet, und funktionale Bereiche innerhalb der Komponenten sind logisch gruppiert (z. B. Abschnitte für Reset-Verhalten, Filter-Persistenz, Update-Funktionen und Rendering). Zudem werden wiederverwendbare Typen eingeführt, die als Grundlage für weitere Verfeinerungen dienen.

Beispielhaft zeigt sich dies an der Einführung eigenständiger Typen zur Beschreibung der Filterlogik.

```
export interface ReactTableFilter {
  id: string;
  value?: string;
  values?: string[];
}

export interface FilterableColumn {
  name: string;
  columns: { title: string; column: number }[];
}
```

Diese Typen abstrahieren wesentliche Konzepte der Anwendung und lassen sich konsistent in mehreren Komponenten verwenden. Dadurch entsteht eine klare strukturelle Grundlage, die die Verständlichkeit des Codes erhöht und spätere Erweiterungen erleichtert.

An einzelnen komplexeren Stellen bleiben die Typen ungenau, sodass keine vollständige Modellierung erreicht wird. Insgesamt ist die strukturelle Qualität jedoch hoch und bietet eine Basis für weiterführende Typverfeinerungen.

Bewertung: 4/5

F.5 Semantik und Benennung

Die Benennung von Typen, Variablen und Funktionen ist überwiegend konsistent und semantisch sinnvoll. Insbesondere die eingeführten Typnamen (z. B. `ReactTableFilter`, `FilterableColumn`, `FilterBoxState`) spiegeln die intendierte Bedeutung der jeweiligen Datenstrukturen klar wider und verbessern die Lesbarkeit des Codes.

An einzelnen Stellen wird die semantische Aussagekraft jedoch durch sehr generische Typen eingeschränkt. Insgesamt ist die Benennung überzeugend, wird jedoch nicht durchgängig durch gleichwertig präzise Typdefinitionen gestützt.

Bewertung: 4/5

F.6 Nachvollziehbarkeit

Die durchgeführten Änderungen sind im Code gut nachvollziehbar: Neue Interfaces, Typannotationen und strukturelle Anpassungen sind klar erkennbar und lokal umgesetzt. Einschränkend ist jedoch der Copy-&-Paste-Ansatz, da kein automatisch reproduzierbarer Transformationsschritt existiert, der die Änderungen eindeutig dokumentiert.

Bewertung: 3/5

F.7 Zeitbedarf

Der Zeitbedarf wird maßgeblich durch den manuellen Arbeitsaufwand (Kopieren, Übernehmen und Integrieren des Codes) bestimmt. Für einzelne Dateien ist die Migration schnell und unkompliziert durchführbar. Mit zunehmender Dateianzahl nimmt die Effizienz jedoch deutlich ab, da keine projektweite Automatisierung möglich ist. Der Zeitbedarf steigt dabei nicht linear mit der Dateigröße, sondern mit der Anzahl der manuell durchzuführenden Schritte.

Bewertung: 2/5

F.8 Installationsaufwand

Es ist weder eine Installation noch eine lokale Konfiguration erforderlich. Die Migration kann unmittelbar durchgeführt werden, da das LLM vollständig extern und ohne projektspezifisches Setup genutzt werden kann. Abhängigkeiten, Build-Konfigurationen oder projektspezifische Anpassungen sind für die Nutzung nicht erforderlich.

Bewertung: 5/5

F.9 Automatisierungsgrad

Die Codegenerierung selbst ist weitgehend automatisiert, die Anwendung erfolgt jedoch manuell und dateibasiert. Eine reproduzierbare, projektweite oder deterministische Ausführung ist nicht gegeben.

Bewertung: 2/5

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Literaturverzeichnis

[1] D. Ahirav. *JavaScript vs TypeScript: A Comprehensive Comparison*. URL: https://dev.to/dipakahirav/javascript-vs-typescript-a-comprehensive-comparison-c97 (besucht am 28. 09. 2025).

[2] D. Beyer. „Competition on Software Verification“. In: *Tools and Algorithms for the Construction and Analysis of Systems* (2012), S. 504–524. DOI: 10.1007/978-3-642-28756-5_38.

[3] D. Beyer, S. Löwe und P. Wendler. „Reliable benchmarking: requirements and solutions“. In: *International Journal on Software Tools for Technology Transfer* 21 (2019), S. 1–29. DOI: 10.1007/s10009-017-0469-y.

[4] G. Bierman, M. Abadi und M. Torgersen. „Understanding TypeScript“. In: *ECOOP 2014 – Object-Oriented Programming* (2014), S. 257–281. DOI: 10.1007/978-3-662-44202-9_11.

[5] L. S. Bschor. „Modern Architecture and Improved UI for Tables of BenchExec“. Bachelorarbeit. LMU Munich, 2019. URL: https://www.sosy-lab.org/research/bsc/2019.Bschor.Modern_Architecture_and_Improved_UI_for_Tables_of_BenchExec.pdf.

[6] React Documentation. *Built-in React Hooks*. URL: https://react.dev/reference/react/hooks (besucht am 29. 09. 2025).

[7] M. Heiskanen. „Migrating a large JavaScript web UI to TypeScript to improve Developer Experience“. Masterarbeit. LUT School of Engineering Science, 2022. URL: https://urn.fi/URN:NBN:fi-fe2022102763371.

[8] S. Hümmer. *Reproduction Package for the Evaluation of JavaScript-to-TypeScript Migration Tools*. Zenodo, 2026. DOI: 10.5281/zenodo.18841209.

[9] S. Jain. *Effortlessly Convert JavaScript to TypeScript with JS2TS Tool — 2025 Guide*. URL: https://js2ts.com/blog/effortlessly-convert-javascript-to-typescript-using-js2ts-tool (besucht am 08. 11. 2025).

[10] V. Komperla, P. Deenadhayalan, P. Ghuli und R. Pattar. „React: A detailed survey“. In: *Indonesian Journal of Electrical Engineering and Computer Science* 26 (2022), S. 1710–1717. DOI: 10.11591/ijeecs.v26.i3.pp1710-1717.

[11] MDN Web Docs. *JavaScript*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript (besucht am 28. 09. 2025).

[12] C. Politowski, J. E. Montandon, D. Costa und G. El Boussaidi. „Understanding JavaScript to TypeScript Migration: Strategies and Quality Assessment“. In: *Social Science Research Network (SSRN)* (2025). DOI: 10.2139/ssrn.5614285.

[13] S. Rudenko. *ts-migrate: A Tool for Migrating to TypeScript at Scale*. URL: https://medium.com/airbnb-engineering/ts-migrate-a-tool-for-migrating-to-typescript-at-scale-cd23bfeb5cc (besucht am 23. 10. 2025).

[14] D. Simon. „Shareable Benchmarking Reports with Enhanced Filters and Dynamic Statistics for BenchExec“. Bachelorarbeit. LMU Munich, 2021. URL: https://www.sosy-lab.org/research/bsc/2021.Simon.Shareable_Benchmarking_Reports_with_Enhanced_Filters_and_Dynamic_Statistics_for_BenchExec.pdf.

[15] P. Wendler und D. Beyer. *sosy-lab/benchexec: Release 3.31*. Zenodo. 2025. DOI: 10.5281/zenodo.17695991.