Architektur Check Your Work

Inhaltsverzeichnis

1. Einführung und Ziele

1.1. Aufgabenstellung

Mit der Applikation `CheckYourWork` werden einfache Programmieraufgaben automatisch geprüft und bewertet. Der Quellcode, der die Programmieraufgabe löst, wird vom Studenten hochgeladen und vom System ausgeführt. Ebenfalls erfolgt eine interne und externe Plagiatsprüfung des Quellcodes. Der Quellcode, sowie das Ergebnis der Ausführung und die Ergebnisse der Prüfungen werden in auditierbarer Form persistiert.

1.2. Verwandte Systeme

Ähnliche Anwendungen gibt es z.B. bei JetBrainsAcademy, Codewars, Hackerrank. Dort kann der Anwender ebenfalls Code hochladen (oder auch — anders als hier geplant — im Browser schreiben), welcher dann ebenfalls ausgeführt, geprüft und bewertet wird. Typische Roundtrip-Zeiten (Quellcode hochladen → Laufen lassen → Ergebnis einsehen) liegen in diesen Anwendungen bei ca. 20-60 Sekunden.

1.3. Qualitätsziele

Tabelle 1. Die wichtigsten Ziele

Nr.	Ziel	Erläuterung
1	Persistenz	Der hochgeladene Quellcode samt Ausführungsergebnis und Benotung soll persistiert werden
2	Auditierbarkeit	Die Daten müssen unveränderbar hinterlegt werden
3	Sicherheit	Es muss verhindert werden, dass der hochgeladene Quellcode in ungesicherten Umgebungen läuft und diese korrumpiert oder Zugriff auf die gespeicherten Daten erhält

HINWEIS 1 und 2 ergeben sich direkt aus der Anforderung. Nr. 3 und die Reihenfolge ist meine Annahme.

1.4. Stakeholder

Tabelle 2. Stakeholder

Stakeholder	Erwartung
Universität	Zeit- und Kostenersparnis bei der Bewertung der Aufgaben, Budgetfreundliche Gesamtlösung
Professoren	Benutzerkomfort, Zeitersparnis
Auditor	Zugang zu Export der Audit-Daten, Vertrauen auf Unveränderbarkeit der Daten

Stakeholder	Erwartung
Studenten	Verfügbarkeit der Anwendung, zeitnahe Bewertung der hochgeladenen Arbeit
Administrator	stösst den Export der Audit-Daten an, pflegt BasisImage und Treibercode

2. Randbedingungen

- Das bestehende mainframe-basierte System 'Leaning Management System' `LMS` soll integriert werden.
- Der Quellcode wird gegen einen externen, web-basierten Service `TurnItIn` geprüft.
- Aus Kostengründen werden verstärkt kostenfreie bzw. günstige Komponenten/Frameworks verwendet und auf teure Kaufsoftware oder Komponenten verzichtet.

3. Fachlicher Context

Folgendes Diagram zeigt den fachlichen Context mit allen Aktoren und allen externen Schnittstellensystemen.

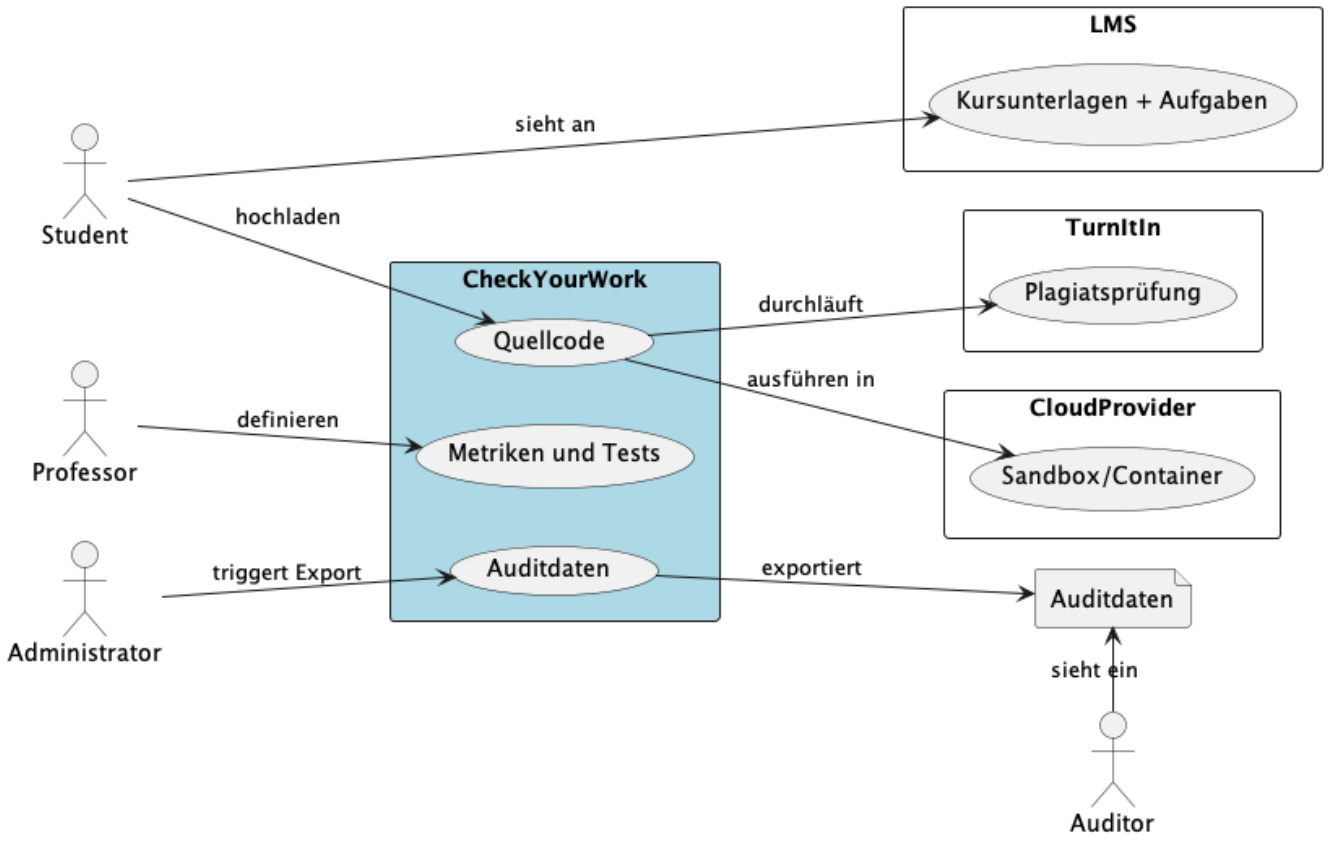

Abbildung 1. Fachlicher Context

- Aktoren: Student, Professor, Auditor, Administrator
- externe Systeme: LMS, TurnItIn, CloudProvider

3.1. Funktionale Anforderungen

HINWEIS	Ich bin mir nicht sicher, ob detailliertere UseCases hier hingehören. Für das Gesamtverständnis finde ich sie aber förderlich. Daher möchte ich den Benutzungsablauf aus Sicht der beteiligten Aktoren hier etwas detaillierter darstellen. In einem echten Projekt ist dies Teil der Requirements-Analyse (bzw. Lastenheft o.ä.)

3.1.1. Aus Sicht des Studenten

Annahme: Folgende Punkte erfolgen **nicht** in dieser Anwendung `CheckYourWork`:

- Student schaut Aufgabe im `LMS` an
- Student schreibt Quellcode zur Aufgabe (auf seinem privaten Rechner in der IDE seiner Wahl, **nicht** im Browser in dieser Anwendung)

Ablauf im `CheckYourWork` nun:

- Student meldet sich an (Abgleich gegen `LMS`)
- der Student wählt die Aufgabe, zu der er eine Lösung hochladen möchte
- ist das Abgabedatum für diese Aufgabe bereits überschritten, ist kein Upload mehr möglich → Ende
- der Student lädt den Quellcode zu der gewählten Aufgabe hoch
- Quellcode durchläuft die dynamische CodeAnalyse, d.h. wird in passender Sandbox ausgeführt, dabei Ausgaben/Tests geprüft
- Quellcode durchläuft die statische CodeAnalyse, d.h.:
  - Quellcode wird gegen Metriken geprüft
  - Quellcode durchläuft die interne Plagiatsprüfung (Vergleich gegen andere Lösungen anderer Studenten)
  - Quellcode durchläuft die externe Plagiatsprüfung (Vergleich gegen Fremdsystem `TurnItIn`)
- Quellcode wird benotet
- Quellcode, Ausführung und Ergebnis werden auditierbar gespeichert
- das Ergebnis wird dem Studenten (nicht editierbar) angezeigt: Bewertung, Metriken, Testergebnisse
- Student kann weitere Versuche hochladen (selber Ablauf wie vorher)
- Am Ende sind alle Versuche des Studenten zu dieser Aufgabe auditierbar festgehalten
- der Student kann einmal hochgeladenen Code nicht mehr ändern oder löschen

Annahme: Die Prüfung des Quellcodes geht schnell (<60 sek) und der Student bekommt synchron Feedback. So kann er zeitnah Nacharbeiten/Verbesserungen vornehmen und einen erneuten Versuch hochladen

3.1.2. Aus Sicht des Professors

Annahmen:

- Die Aufgaben/Kursunterlagen sind im `LMS` hinterlegt
- Zuordnung Studenten zu Kurs sind im `LMS` hinterlegt
- Zugangsdaten des Professors sind im `LMS` hinterlegt

Ablauf (im neuen System `CheckYourWork`):

- Professor meldet sich an (Abgleich gegen `LMS`)
- Professor legt fest:
  - welche Aufgabe aus dem `LMS`
  - für welchen Kurs (und damit indirekt für welche Studenten)
  - bis wann (Abgabedatum/zeit)

    - mit welchen Abnahmekriterien (Metriken / Tests)

gelöst werden soll.

Dies könnte sinngemäss etwa so aussehen:

> Alle Studenten des Kurses 'Python für Fortgeschrittene - WinterSemester2022' sollen bis 24.9.2022 15:00Uhr die Aufgaben PFF-18, PFF-19 und PFF-20 (referenziert Aufgaben aus `LMS`) lösen. Dabei ist zu beachten:
>
> - Bei Aufgabe PFA-18 bitte nicht mehr als 20 Lines of code und
> - bei PFA-19 eine Lösung mit Rekursion schreiben, d.h. ohne Schleifen (while/for)

Im Grunde genommen eine einfache CRUD-Anwendung, d.h. der Professor kann Daten auch nochmal ändern.

3.1.3. Aus Sicht des Auditors bzw. des Administrators

HINWEIS	Hier ist unklar, wie der Ablauf erfolgen soll. Soll der Auditor Zugang zur Datenbank bekommen und kann dort dann die Daten prüfen? Oder erwartet er einen Export? Wenn ja in welchem Format? Excel? CSV? JSON? XML? Hier gibt es sicherlich rechtliche Vorgaben, in welcher Form ein Audit ablaufen soll.

Annahme: Ein Export als EXCEL reicht aus. Der Auditor erhält **keinen** Zugang zur Datenbank oder zur Anwendung.

- Es wird ein Audit von den Auditoren angekündigt (unter Angabe der Rahmenbedingungen, d.h. Audit für den Jahrgang X und / oder alle Studenten, die den Kurs Y besucht haben)
- Ein Administrator der Anwendung `CheckYourWork` triggert den Export der Audit-Daten
- Die Audit-Daten werden vom System im EXCEL-Format exportiert und vom Administrator heruntergeladen
- Der Adminstrator übergibt diese Datei dem Auditor
- Der Auditor nimmt diese exportierten Daten entgegen und führt das Audit durch

4. Lösungsstrategie

Aspekt	Lösung	siehe
Sicherheit	Der hochgeladene Quellcode wird bei einem externen CloudProvider in einem eigens dafür provisionierten Container ("Sandbox") compiliert und ausgeführt. So ist sichergestellt, dass kein Zugriff von diesem Quellcode auf das Produktivsystem und/oder die Datenbank erfolgen kann.	Section 8.4, “Codeausführung”
Auditierbarkeit	Die gespeicherten Daten werden beim Speichern vollumfänglich aufgelöst, d.h. sämtliche Referenzen ausgehend vom Hauptdatensatz ("Ausführung") werden aufgelöst (Snapshot aller referenzierten Daten zum Zeitpunkt der Codeausführung). Die Zugriffsrechte auf die Datenbank werden so eingeschränkt, dass einmal gespeicherte Datensätze im Nachhinein nicht verändert werden können.	Section 8.6, “Auditierbarkeit”
Plagiatsprüfung	Die Plagiatsprüfung erfolgt in zwei Stufen: intern und extern. Bei der internen Plagiatsprüfung wird die hochgeladene Lösung mit anderen Lösungen zur selben Aufgabe verglichen. Bei der externen Plagiatsprüfung wird die Lösung gegen den externen Anbieter `TurnItIn` verglichen.	Section 8.3, “interne Plagiatsprüfung” Section 8.10, “Schnittstelle TurnItIn”
Datenstruktur	Das Ergebnis der Codeausführung enthält überwiegend unstrukturierte, schema-lose, nicht relationale Textdaten. Daher wird das Ergebnis einer Codeausführung als Document in einer NoSQL-Datenbank gespeichert. (**TODO**: oder doch im PostgresQL? mit Datentyp `jsonb`?)	f
Anmeldung am System / "single sign on"	Beim Anmelden ans System `CheckYourWork` wird ein Weiterleiten an `LMS` durchgeführt, d.h. im `CheckYourWork` selber werden keinerlei Benutzerdaten der Anwender gespeichert.	Section 8.2, “Authentisierung / Autorisierung”

5. Bausteinsicht

5.1. Ebene 1

Hier die Bausteinsicht der Applikation im technischen Kontext (C4 - Ebene 1). Vergleiche hierzu Section 3, “Fachlicher Context”

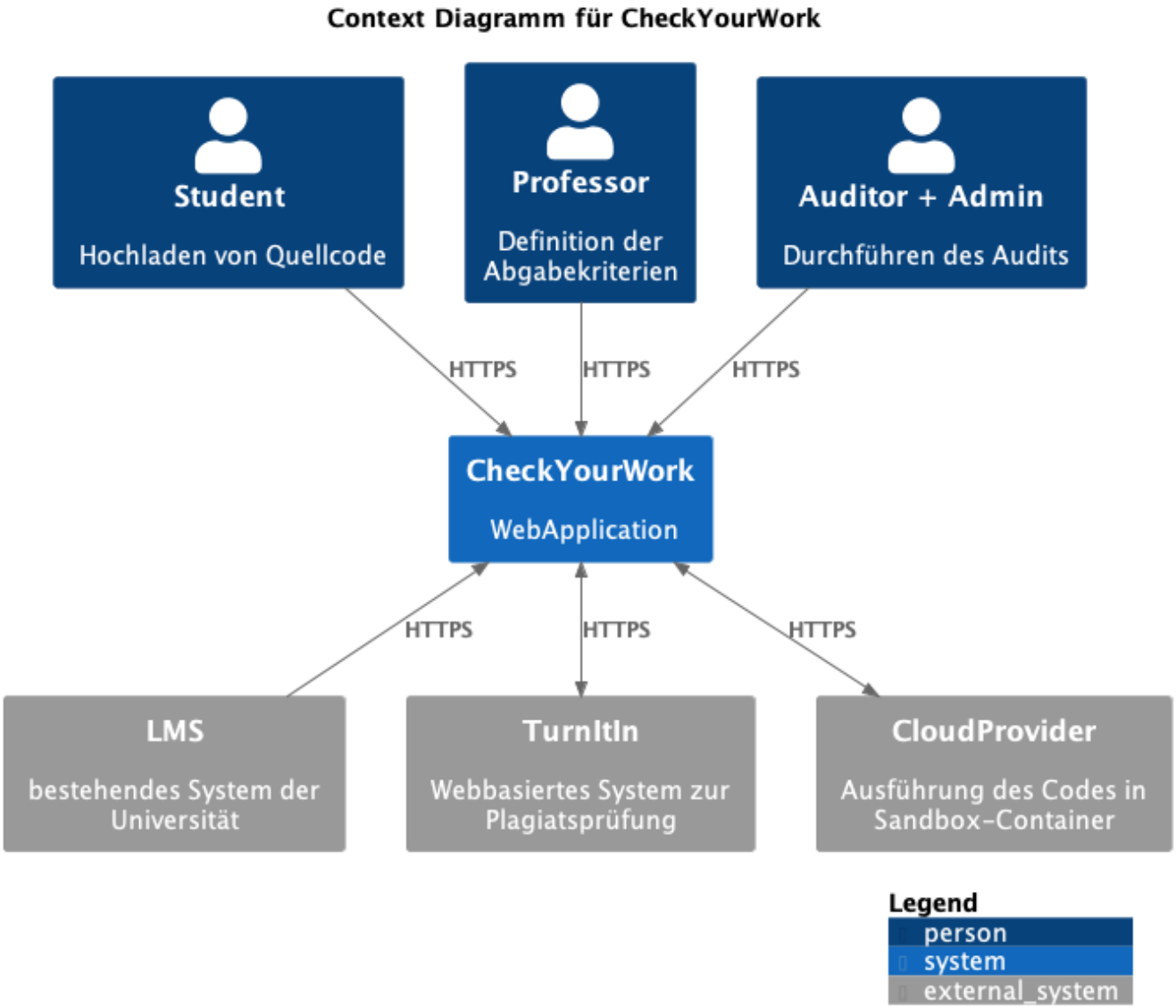

Abbildung 2. Bausteinsicht - Ebene 1

5.2. Ebene 2

Hier ein Überblick über die Teilkomponenten des Systems (C4 - Ebene 2).

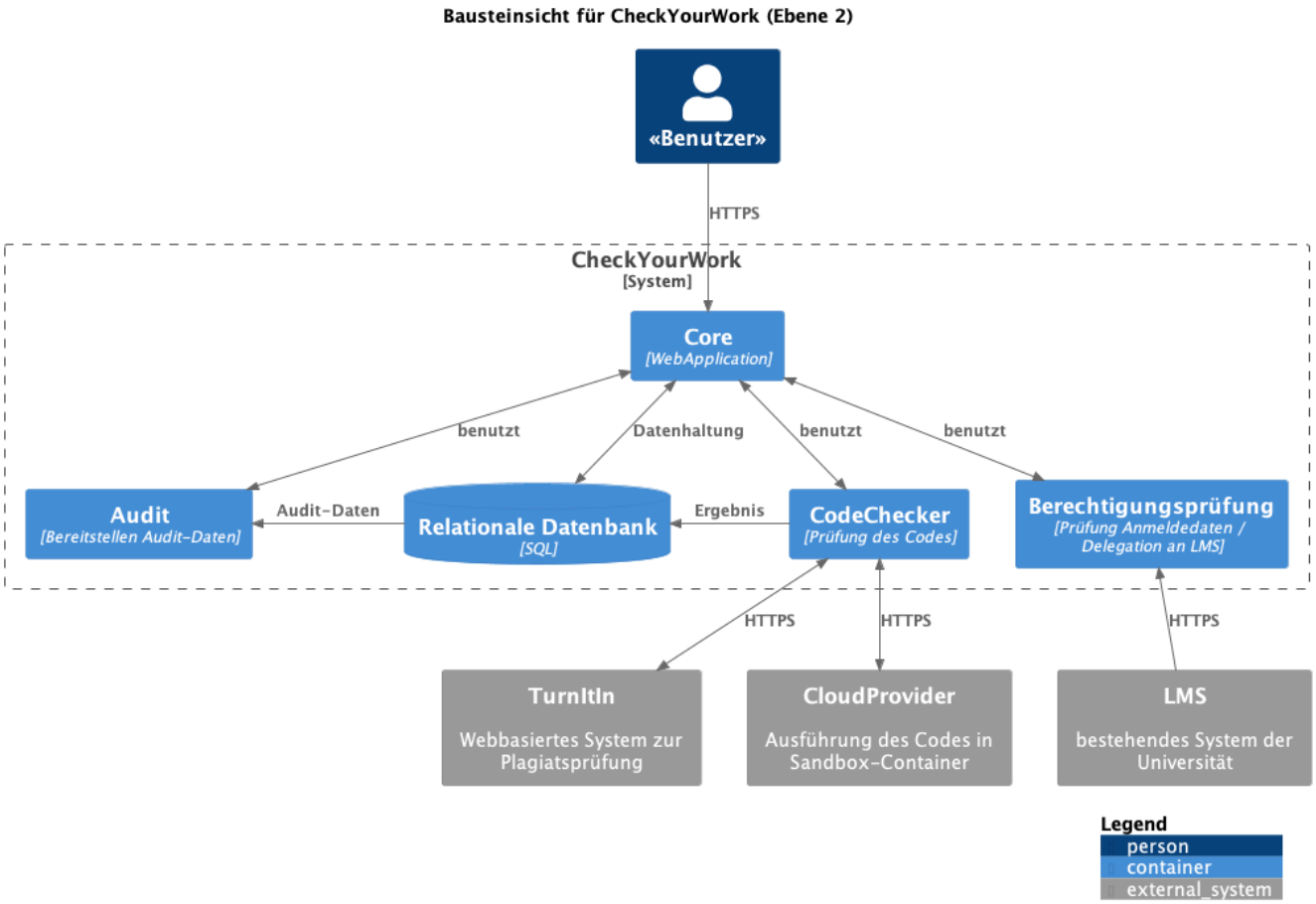

Abbildung 3. Bausteinsicht - Ebene 2

Tabelle 3. Komponenten im System

Komponente	**Zweck**
Core	Kernkomponente, übernimmt die Datenhaltung zur DB, orchestriert die anderen Komponenten, stellt Oberfläche dar
Berechtigungsprüfung	prüft Berechtigungen gegen LMS
CodeChecker	kümmert sich um die Prüfung und Bewertung des hochgeladenen Quellcodes
Audit	Komponente, welche die Audit-Daten erzeugt und ausleitet
Datenbank	hält die Daten

5.3. Ebene 3 - Komponente CodeChecker

Bei diesem Entwurf wird jeder Teilaspekt der Codeprüfung (Metriken, Plagiatsprüfung intern und extern, statische Codeprüfung, etc.) in seiner eigenen Teilkomponente abgebildet.

Das Laufzeitverhalten wird hier erläutert: Section 6.1, “Laufzeitsicht des Modul CodeChecker”

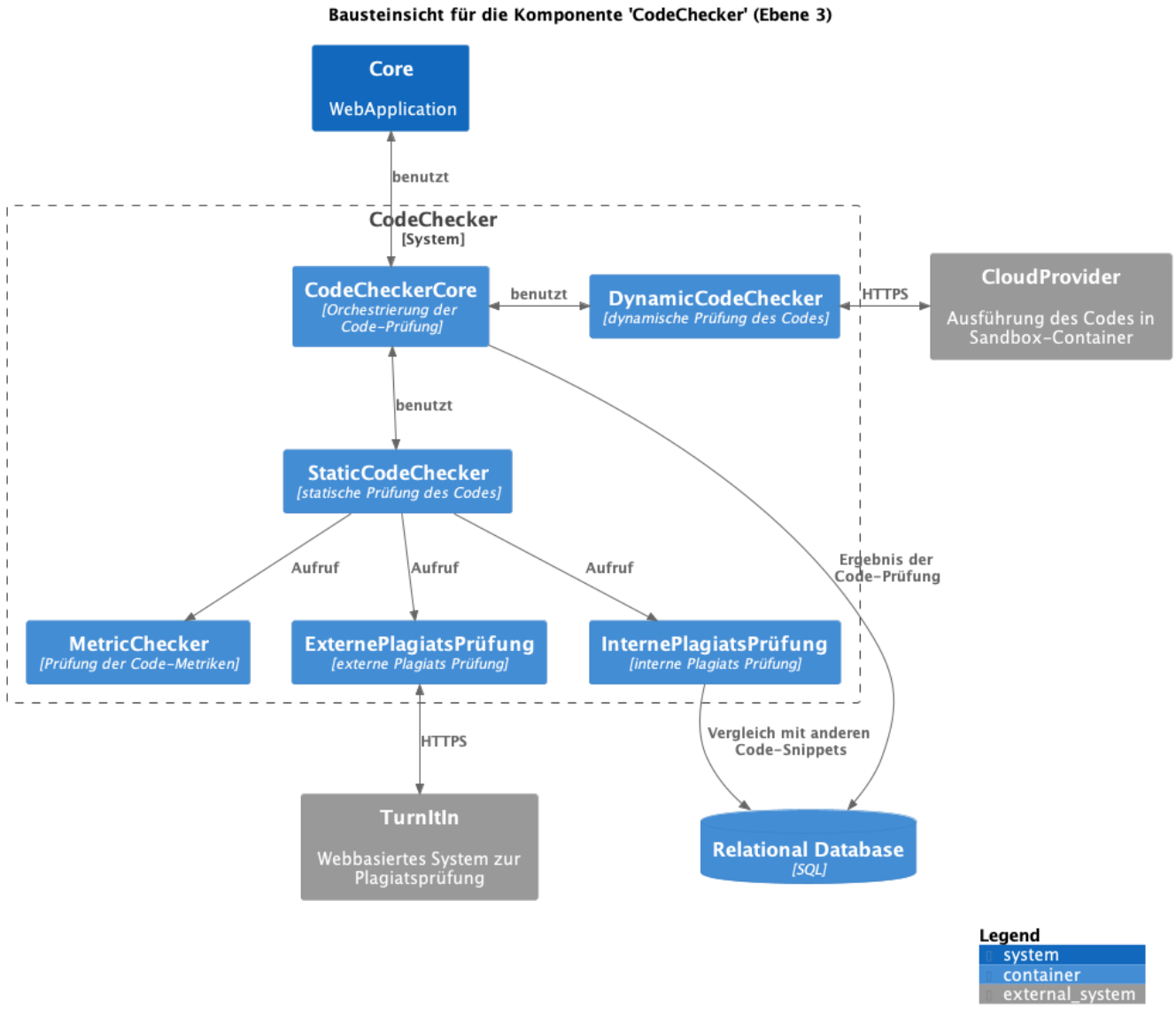

Abbildung 4. Bausteinsicht - Ebene 3

6. Laufzeitsicht

Hier beschränke ich mich auf den nicht-trivialen Teil der Code-Prüfung Section 5.3, “Ebene 3 - Komponente CodeChecker”.

6.1. Laufzeitsicht des Modul CodeChecker

Folgende Diagramme zeigen das Laufzeitverhalten innerhalb der Komponente CodeChecker. Zuerst erfolgt die dynamische Codeprüfung, anschliessend die statische. Diagram 1 zeigt den gesamten Ablauf, Diagram 2 die dynamische Prüfung, Diagram 3 die statische Prüfung.

Die Prüfungen laufen wie folgt ab:

- dynamische Prüfung mit Ausführung des Quellcodes in der Sandbox beim CloudProvider
- statische Prüfung mit
  - Metrikcheck
  - interner Plagiatsprüfung (Vergleich gegen Quellcode anderer Studenten)
  - externer Plagiatsprüfung mit Aufruf des externen Systems `TurnItIn`
- Benotung der Arbeit
- Aggregation und Speichern der Ergebnisse in Datenbank

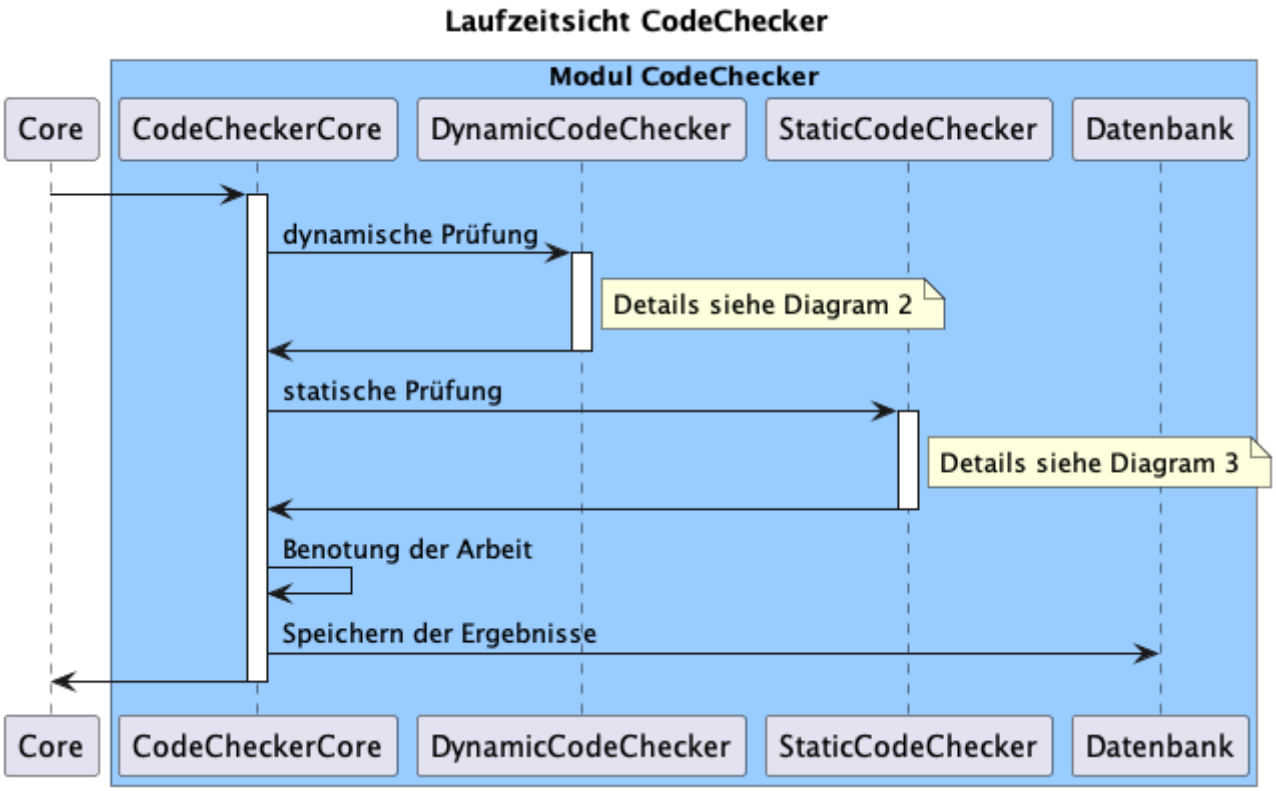

Abbildung 5. Laufzeitsicht-CodeChecker - Übersicht (ohne CloudProvider und TurnItIn)

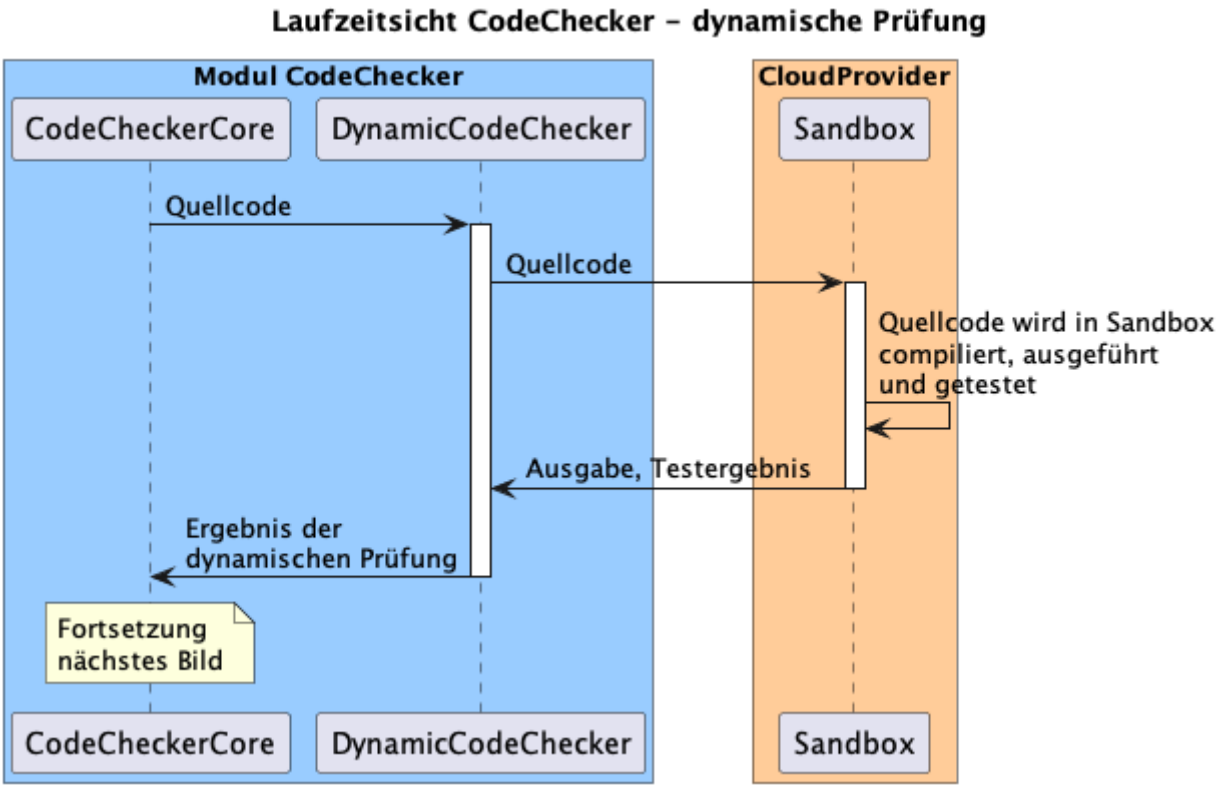

Abbildung 6. Laufzeitsicht-CodeChecker - dynamische Prüfung

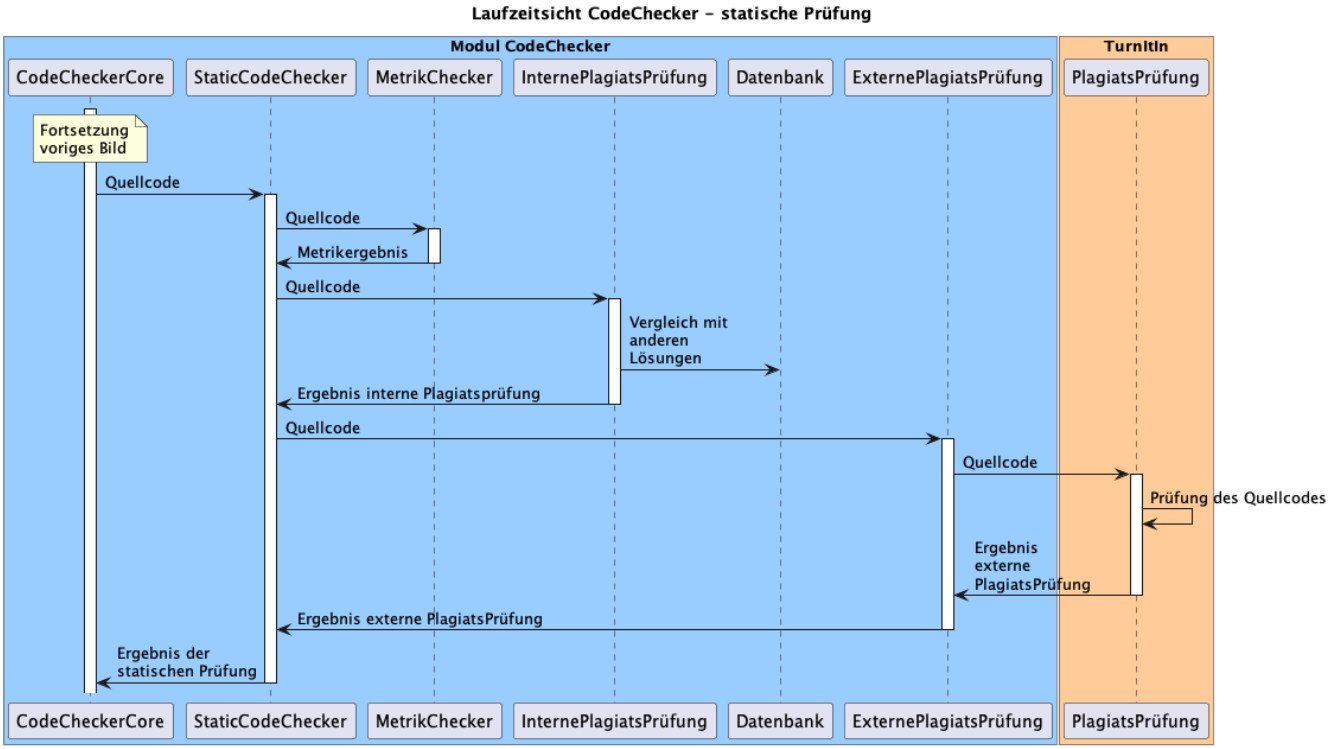

Abbildung 7. Laufzeitsicht-CodeChecker - statische Prüfung

7. Verteilungssicht

HINWEIS

Die Infrastruktur ist komplett unklar. Im echten Projekt würde man hier weitere Informationen beim Kunden einholen müssen:

- Gibt es Infrastruktur, die mitbenutzt werden kann oder soll?
- Gibt es Vorgaben bzgl CloudProvidern?
- Ist externes Hosting erlaubt oder verboten?

Ebenfalls zu klären ist, ob es für den externen Dienst `TurnItIn` eine Testumgebung gibt, welche in der Integrationsumgebung angebunden werden kann.

Annahme: Es gibt interne Infrastruktur bei der Universität. Dort werden Umgebungen bereitgestellt. Das Hosting der externen Sandboxes erfolgt bei AWS.

7.1. Umgebungen

Es werden folgende Umgebungen in der Infrastruktur der Universität eingerichtet:

- Test-Umgebung
- Integrationsumgebung
- Produktivumgebung

Desweiteren werden beim CloudProvider dynamisch Container-Instanzen bereitgestellt.

Die BasisImages werden beim CloudProvider in der Image Registry hochgeladen.

7.2. Deployment

Alle Komponenten der Anwendung `CheckYourWork` sind in jeder Umgebung jeweils auf einem Server deployt. Der Code wird in einem Container beim externen CloudProvider compiliert und gestartet. Diese Container werden dort "on demand" gestartet, so dass eine gewisse Elastizität gewährleistet ist. Bei jedem Code-Upload eines Studenten wird beim CloudProvider eine eigene Container-Instanz hochgefahren.

HINWEIS

Hier ist mir auch unklar, wie genau so ein Bild aussehen soll. Die bei PlantUML sehen alle komisch aus…

8. Querschnittliche Konzepte

8.1. Anbindung LMS

Hier handelt es sich um das bestehende, mainframe-basierte Learning Management System (siehe https://de.wikipedia.org/wiki/Lernplattform)

HINWEIS	Hier fehlt sehr viel Information, z.B. **warum** soll LMS überhaupt angebunden werden? Welche Daten werden ausgetauscht?

8.1.1. fachliche Details

Annahmen: Im `LMS`

- sind Kursunterlagen und Aufgaben hinterlegt
- sind Studenten und Professoren als Benutzer hinterlegt
- sind die Studenten ihren Kursen zugeordnet
- sind die Professoren ihren Kursen zugeordnet
- sind die Aufgaben den Kursen zugeordnet
- sind die Aufgaben einer Programmiersprache zugeordnet (oder sollte sich aus dem Kurs ergeben)

Annahme: Es wird eine Webschnittstelle (REST) zum `LMS` eingerichtet, welche die o.g. Informationen bereitstellt.

Abgerufene Informationen:

1. Wenn ein Student sich im `CheckYourWork` anmeldet, wird gegen `LMS` abgefragt, welche Kurse (und die dazugehörigen Aufgaben) dieser Student sehen darf. Diese Aufgaben werden dem Studenten aufgelistet. Er kann dann die Aufgabe auswählen, zu der er Quellcode hochladen möchte.
2. Wenn ein Professor sich im `CheckYourWork` anmeldet, sieht er die Kurse, die er gerade unterrichtet. Er kann einen dieser Kurse auswählen und sieht die zu diesem Kurs gehörenden Aufgaben. Er kann dann zu einer dieser Aufgabe die Abgabekriterien eingeben (Deadline, Metriken, etc.).

8.1.2. technische Details

HINWEIS	In einem echten Projekt müsste man hier die technischen Details der Schnittstelle zum `LMS` weiter ausarbeiten, d.h. • exakte Informationen was ausgetauscht wird • unter welchen URLs ist `LMS` aufzurufen • Absicherung der Verbindung (technischer User? API-Key?) • SLA (Service level agreement)

- etc.

8.2. Authentisierung / Autorisierung

Beim Anmelden an das System `CheckYourWork` wird die Authentisierung gegen `LMS` durchgeführt ("single sign on"). So wird vermieden, dass im `CheckYourWork` Informationen redundant zum LMS gehalten werden müssen. D.h. Studenten und Professoren können sich am `CheckYourWork` mit derselben Kennung (Benutzername und Passwort) anmelden wie am `LMS`.

Jeder Anwender, der `CheckYourWork` benutzen möchte oder darf, muss vorher im `LMS` registriert sein. Dies erscheint plausibel, da sowohl Professoren als auch Studenten sicherlich im `LMS` registriert sind (Immatrikulation, Einschreiben in Kurse, etc.), bevor sie im `CheckYourWork` Aufgaben aufgeben oder lösen werden.

HINWEIS

Falls diese Daten **nicht** im LMS hinterlegt sind, so sind sie sicherlich in einem anderen System der Universität hinterlegt. Dann wäre dringend zu empfehlen eine Schnittstelle zu diesem System zu erstellen.

8.3. interne Plagiatsprüfung

HINWEIS

komplett unklar:

- Es soll gegen andere Lösungen anderer Studenten verglichen werden. Gegen welche Studenten? Nur die des gleichen Kurses? Oder auch gegen die Studenten des Vorjahres, die diesen Kurs besucht haben? Oder gegen den kompletten Lösungspool einer Aufgabe aller Studenten aller Jahre, die jemals diese eine Aufgabe gelöst haben? Je grösser man diesen Pool macht, desto höher die Wahrscheinlichkeit ähnliche Lösungen zu finden, die dann als vermeintliches Plagiat markiert werden.
- Wie soll aber ein Plagiat erkannt werden? Je kleiner die Aufgabe ("simple programming assignments"), desto ähnlicher sind alle Lösungen. Die Aufgabe "Schreibe eine Funktion, die prüft, ob eine Zahl eine Primzahl ist" wird bei 30 Studenten eines Kurses sicherlich sehr oft sehr ähnlich gelöst. Geht man weitere Jahre zurück, steigt die Wahrscheinlichkeit weiter, eine ähnliche Lösung zu finden.
- Wie kann man "false positives" vermeiden? Also Code, der fälschlicherweise als Plagiat erkannt wird.
- Wie kann man "false negatives" vermeiden? z.B. haben zwei Studenten zusammengearbeitet und jeweils die selbe Lösung nur mit anderen Variablennamen im Code. Wie kann das erkannt werden?
- Wie sieht überhaupt so ein Ergebnis einer internen Plagiatsprüfung aus? ("Code 100% identisch zu Codeausführung <id> von Student <student_id> vom <Datum>" ?)
- evtl optional Kauflösungen evaluieren (das widerspricht aber etwas dem schmalen IT Budget)

- evtl gibt es auch Frameworks, die sowas können? (auch für verschiedene Programmiersprachen?)
- evtl https://github.com/manuel-freire/ac2 evaluieren

→ **Risiko**: interne Plagiatsprüfung noch sehr unklar/offen, weitere Klärung nötig

Vorschlag einer denkbaren/kostengünstigen Minimallösung:

- Der hochgeladene Quellcode wird mittels Textvergleich gegen alle in diesem Jahr zu dieser Aufgabe hochgeladenen Quellcodes anderer Studenten verglichen. Bei exaktem Match gilt der Quellcode als Plagiat.
- Um zu viele "false positives" gerade bei kurzen Lösungen zu vermeiden, wird die interne Prüfung erst ab einer Zeilenanzahl von 10 Zeilen durchgeführt. (Alternativ: Könnte man auch pro Aufgabe einstellbar machen. D.h. bei manchen Aufgaben, bei denen man eine kurze Lösung erwartet, kann diese Prüfung komplett ausgeschaltet werden)

8.4. Codeausführung

8.4.1. Motivation

Der Quellcode des Studenten soll ausgeführt, geprüft und benotet werden.

8.4.2. Hintergründe / Konsequenzen

- Abhängig vom Kurs und der Aufgabe kann der hochgeladene Quellcode in unterschiedlichen Programmiersprachen geschrieben sein.
- Abhängig von der Programmiersprache muss der Quellcode evtl. vorher compiliert werden.
- Zur Ausführung des Quellcodes ist eine Art "Treibercode" nötig. Dies kann Testcode sein (z.B. JUnit im Java) oder ein Startskript (z.B. `java <MainClass>`).
- Es muss vermieden werden, dass vom Studenten hochgeladener Quellcode direkt auf den Rechnerinstanzen der Anwendung ausgeführt wird und diese - beabsichtigt oder unbeabsichtigt - korrumpiert. Daher läuft der hochgeladene Quellcode in einem eigenen, isolierten Container, der keinen Zugriff auf andere Container, Server, Netzwerke oder Datenbanken hat.
- Aus Sicherheitsgründen wird dieser Container mit einem Zeitlimit von 30 Sekunden und einem Speicherlimit von 256 MB gestartet.
- Dieser BasisContainer wird aus einem BasisImage gestartet.
- Für jede Programmiersprache wird vom Administrator der Anwendung `CheckYourWork` initial ein eigenes Basisimage (mit passendem Compiler, passender Laufzeitumgebung, etc.) erzeugt und in der ImageRegistry beim CloudProvider hochgeladen.

HINWEIS

Diesen Punkt müsste man in einem "echten" Projekt vorher nochmal ausprobieren und verifizieren. Wie kann man von einer Webanwendung aus einen anderen Container starten, dort Code einbinden, compilieren, laufen

> lassen und Ergebnisse/Ausgaben abgreifen?

8.4.3. genauer Ablauf

Vorbereitende Arbeiten (bevor ein Student eine Aufgabe lösen kann):

- Administrator erstellt programmiersprachen-spezifisches Basisimage (z.B. Java, Go, Python, etc.) und inkludiert darin den passenden Compiler und die Laufzeitumgebung
- Administrator lädt dieses Basisimage beim CloudProvider in die dortige ImageRegistry
- Administrator registriert dieses BasisImage im System `CheckYourWork`, so dass alle Kurse für diese Programmiersprache dieses BasisImage verwenden
- Administrator erstellt für jede Programmieraufgabe Treibercode bzw. Testcode — also Code, der die Korrektheit des vom Studenten hochgeladenen Codes verifiziert **OFFEN: Wo liegt dieser Treibercode und wie testet er den Studentencode? Wie kommt der in den Container? Es braucht JEDE(!) Aufgabe auch eigenen, spezifischen Treibercode! Damit JEDE Programmieraufgabe individuell geprüft werden kann!**

Ablauf bei Lösen einer Aufgabe durch den Studenten:

- Student lädt Quellcode zu einer Aufgabe hoch
- die Programmiersprache des Quellcodes bzw. der Aufgabe bestimmt welches Basisimage verwendet wird
- beim CloudProvider wird das passende BasisImage aus der ImageRegistry gezogen und als Container gestartet
- in diesen Container wird der Treibercode eingebunden (VolumeMount)
- in diesen Container wird der hochgeladene Quellcode des Studenten eingebunden (VolumeMount)
- Quellcode und Treibercode werden compiliert, falls nötig (entfällt bei Skriptsprachen)
- Abbruch falls Quellcode nicht compiliert
- Treibercode wird gestartet und prüft den Quellcode
- die Ausgabekanäle (StdOut und StdErr) werden während der Ausführung protokolliert
- mit Beendung des Treibercodes wird der Container sauber heruntergefahren und beendet
- falls der Container nach 30 Sekunden noch läuft, wird er gekillt

8.5. Datenhaltung / Persistenz

TODO: Was wird alles persistiert? Evtl ersten groben Datenmodellentwurf hier aufnehmen? Datenstruktur NoSQL! evtl kurzes Beispiel JSON beschreiben!

8.6. Auditierbarkeit

Was genau heisst "auditierbar"?

"auditierbar" heisst hier Revisionssicher, d.h. im Wesentlichen korrekt, vollständig und unveränderbar ('immutable'). siehe z.B. auch https://de.wikipedia.org/wiki/Revisionssicherheit

8.6.1. Umfang der Daten

Bei jeder(!) Code-Ausführung wird in der Datenbank gespeichert:

Tabelle 4. Umfang der gespeicherten Daten

Dateninhalt	**Form**
der Quellcode in textueller Form, unverändert, so wie er hochgeladen wurde	**Text**
der Zeitpunkt der Ausführung	Timestamp
der Benutzer, der diesen Code hochgeladen hat	Benutzer-Id
eine Referenz zur Aufgabe, die dieser Code lösen soll	Aufgaben-Id
eine Referenz oder ID vom Basis-Image, in dem dieser Code beim ext. Provider lief	Id
der Testcode in textueller Form, der zur Prüfung verwendet wurde	**Text**
Inhalt des StdOut während der Ausführung	**Text**
Inhalt des StdErr während der Ausführung	**Text**
Ergebnis der statischen CodeAnalyse (Metrikprüfung)	**Text**
Ergebnis der internen Plagiatsprüfung	**Text**
Ergebnis der externen Plagiatsprüfung	**Text**
die automatisch ermittelte Gesamtbenotung des Code	Zahl

TODO: Das sind echt viele Daten ...

Diese Daten werden beim Speichern aufgelöst und dupliziert gespeichert, d.h. Referenzen werden aufgelöst, etc. So ist sichergestellt, dass nachträgliche Änderungen an z.B. Metriken sich nicht in die persistierten Auditdaten durchschlagen. (Snapshot zum Zeitpunkt der Ausführung)

Beispiel: Es werden **keine** Referenzen gespeichert (sinngemäß):

```
Metrik mit Metrik-ID 132 ist verletzt
```

Wenn man nur die Referenz speichert, wäre unklar, wie die Metrik mit der ID 132 zum Zeitpunkt der Codeausführung aussah. Diese Referenz wird in aufgelöster Form gespeichert:

```
Metrik mit Metrik-ID 132 (lines of code darf 40 nicht übersteigen) ist verletzt
```

So ist auditierbar festgehalten, das die Metrik **zum Zeitpunkt der Prüfung** loc < 40 beinhaltete, obwohl sie ggf. heute auf loc < 100 geändert wurde.

8.6.2. Speichern in Dokumentenform

Wie man in Section 8.6.1, “Umfang der Daten” erkennen kann, enthält ein gespeicherter Datensatz einer Ausführung sehr viele Informationen im Freitextformat unbekannter Struktur und Grösse. Daher wird das Ergebnis einer Ausführung als Dokument in einer Dokumenten-Datenbank gespeichert.

8.6.3. Einschränkung der Zugriffsrechte

Sobald die Daten in der Datenbank persistiert sind, darf kein User diese verändern oder löschen. Dies wird sichergestellt, in dem der technische Benutzer mit dem die Anwendung auf die Datenbank zugreift nur Inserts und Selects machen darf, keine Updates oder Deletes.

Also vom "CRUD" (Create - Read - Update - Delete) sind nur C und R erlaubt, U und D nicht.

8.6.4. Aufbewahrungsfristen

Um der Revisionssicherheit zu genügen werden die Daten regelmässig archiviert und aufbewahrt. Archivierungsfrequenz und Aufbewahrungsdauer sollte im echten Projekt hier aufgeführt werden.

8.7. Metriken

HINWEIS	**TODO**: externe Lösung prüfen? sonarqube? einfaches RegEx wird evtl nicht ausreichen um Metriken zu prüfen? halte ich für sehr schwierig und teuer, wenn man hier ein SonarQube nachbauen möchte. Es müssten Metriken für jede Programmiersprache (zu der es einen Kurs gibt) aufgenommen werden. Ein kurzer Blick ins https://sonarcloud.io zeigt dort 650 Regeln für Java, 580 für C++, 222 für Python, etc. Will man das alles nachbauen? Geht das überhaupt so einfach? Ich bezweifle, dass eine Code-Metrik immer nur eine einfache RegEx ist. Evtl muss man auch im compilierten Code (AST, Abstract Syntax Tree) Prüfungen machen?

→ Für mich die grösste Unsicherheit im Projekt und das grösste Risiko, siehe [section-technical-risks]

8.8. Automatische Benotung

HINWEIS	**TODO**: Wie soll das ablaufen? Wer definiert wo die Regeln? "Keine Fehler + keine Metriken verletzt + Plagiatsprüfung negativ == 100 Punkte"? Wo gibt es wann wie welchen Punktabzug? Welche Noten gibt es überhaupt?

Folgende Teilaspekte sind denkbar und müssten in die Benotung einfliessen:

- Code compiliert ja/nein

- Code läuft ja/nein
- Code liefert das richtige Ergebnis ja/nein
- Code erfüllt Metriken ja/nein
- Code erfüllt interne Plagiatsprüfung ja/nein
- Code erfüllt externe Plagiatsprüfung ja/nein

Volle Punktzahl gibt es für die Aufgabe, wenn alle Punkte positiv durchlaufen werden, d.h.:

- ☑ Code compiliert
- ☑ Code läuft
- ☑ Code liefert das richtige Ergebnis
- ☑ Code erfüllt Metriken
- ☑ Code besteht interne Plagiatsprüfung, d.h. ist **kein** Plagiat
- ☑ Code besteht externe Plagiatsprüfung, d.h. ist **kein** Plagiat

Wie sich das Nicht-Erfüllen eines oder mehrerer Teilaspekte in der Gesamtbenotung der Aufgabe bemerkbar macht, ist offen und kann hier auch nur schwer angenommen werden. (Ist für die Architektur an sich aber auch nicht weiter entscheidend. Das ist Teil der Business-Logic.)

8.9. Mengengerüste

HINWEIS	unklar wo sowas im arc42 hingehört, finde ich aber wichtig für ein neues System!

Hier einige kurze Abschätzungen:

8.9.1. Code Uploads und Ausführungen

- 300 User pro Jahr scheint nicht viel, keine globale / überregionale Lösung nötig
- in der Anforderung ist von "simple programming assignments" die Rede, d.h vermutlich <100 lines of code pro Aufgabenlösung
- Annahmen:
  - 300 Studenten pro Jahr
  - Jeder Student besucht pro Vorlesungstag 2 Kurse
  - Jeder Kurs stellt pro Vorlesungstag eine Aufgabe (oder anders: Jeder Student erhält in Summe pro Woche 10 Aufgaben)

    ```
    --> 300 x 2 x 1 = 600 Codesnippets plus Ausführungen/Plagiatsprüfungen pro
    Vorlesungstag
    ```

- Annahme:

    - Während der Vorlesungszeit (werktags 8-18 Uhr) wenig Aktivität (da sind die Studenten ja in der Vorlesung)
    - Erhöhte Aktivität in der Vorlesungszeit abends, nachts und am Wochenende (zu den Zeiten, wenn die Aufgaben gelöst werden)
    - Während der Ferien wenig Aktivität

    ```
    --> 600 Uploads pro Vorlesungstag --> verteilt auf 10 Stunden vorlesungsfreie
    Zeit am Tag macht das ca. 60 Uploads pro Stunde, d.h. ca 1 pro Minute
    ```

- Vermutung: Vor Deadlines erhöhtes Aufkommen von Anfragen, d.h. Elastizität nicht unwichtig
- Annahme:
    - 16 Wochen Vorlesungszeit pro Semester
    - 5 Tage pro Woche
    - 2 Semester pro Jahr

    ```
    --> 2 * 16 * 5 = 160 Vorlesungstage pro Kalenderjahr
    ```

    ```
    --> 600 Code-Ausführungen pro Vorlesungstag * 160 Vorlesungstage pro Jahr
    --> 96.000 Code-Ausführungen pro Kalenderjahr
    ```

TODO: Kann das stimmen? Kommt mir hoch vor. Wie soll jemand ein Audit über 100.000 Dinge machen?

8.9.2. Speicherbedarf einer Code-Ausführung

Bei jeder(!) Code-Ausführung wird ein Datensatz gemäss Section 8.6.1, “Umfang der Daten” in der Datenbank gespeichert.

Da aktuell noch fast jeder der o.g. Punkte im groben Planungsstand ist, kann hier noch keine verlässliche Grösse angegeben werden. (Dies sollte aber im weiteren Verlauf der Architekturarbeit noch erfolgen!)

8.9.3. Grösse eines Audit-Reports

Wird ein Audit pro Jahr gemacht und alle 96.000 CodeAusführungen eines Jahres im Audit exportiert, so erhält man ca. **xxx** MB pro Audit-Report

8.10. Schnittstelle TurnItIn

TODO: unklar wo sowas (Schnittstellen) im arc42 am besten aufgehoben ist

HINWEIS	Auch hier sehr dürftige Info in der Anforderung, Art der Schnittstelle unklar ("web-based"? REST? SOAP? XML?)

8.10.1. fachliche Details

- Web-basierte Schnittstelle zur Plagiatserkennung
- Annahme:
  - synchrone Webschnittstelle
  - das Codesnippet wird per HTTP POST ans `TurnItIn` geschickt
  - das Ergebnis der Plagiatsprüfung liegt binnen weniger Sekunden vor und wird synchron in der Response (JSON) zurückgeschickt

ähnliche Systeme:

- https://codequiry.com/
- https://copyleaks.com/code-plagiarism-checker/

8.10.2. technische Details

HINWEIS

In einem echten Projekt müsste man hier die technischen Details der Schnittstelle zum `TurnItIn` weiter ausarbeiten, d.h.

- exakte Informationen was ausgetauscht wird
- unter welchen URLs ist `TurnItIn` aufzurufen
- Absicherung der Verbindung (technischer User? API-Key?)
- SLA (Service level agreement)
- etc.

9. Architekturentscheidungen

Tabelle 5. Architekturentscheidungen

Thema	Begründung
Quellcode wird in isolierter Sandbox compiliert und ausgeführt	grundlegende Sicherheitsentscheidung. Es ist zu risikoreich unbekannten Code in einem produktiv genutzten System laufen zu lassen
Authentisierung/Autorisierung/' single sign on' gegen LMS	Es soll vermieden werden, dass Daten dupliziert vorliegen und somit voneinander abweichen können. Ausserdem wird dadurch der Benutzerkomfort erhöht (eine Kennung für `LMS` und `CheckYourWork`)
AWS als externe CloudProvider	Da es hier keine Vorgaben gibt, nehme ich das womit ich mich am besten auskenne.
Techstack: Kotlin/SpringBoot/Postgres	Da es hier keine Vorgaben gibt, nehme ich das womit ich mich am besten auskenne.
klassische Webanwendung, serverside rendering, keine 'single page application'	Es gibt keine Anforderung einer reaktiven Oberfläche und der Umfang der Oberfläche scheint recht gering.

10. Qualitätsanforderungen

Tabelle 6. Qualitätsanforderungen

Thema	Begründung
Verfügbarkeit/Elastizität	Ein Student kann jederzeit eine Aufgabe hochladen, d.h. es ist immer eine Sandbox verfügbar. Die Wartezeit auf eine Sandbox beträgt maximal 20 Sekunden
Performance	Die Gesamtzeit für einen Roundtrip (Quellcode hochladen → in Sandbox laufen lassen → dynamische und statische Codeanalyse → Ergebnis einsehen) darf 60 Sekunden nicht überschreiten
Sicherheit	Es ist sichergestellt, dass der hochgeladene Code keinen Schaden anrichten oder Daten manipulieren kann.
Revivionssicherheit	Es ist sichergestellt, dass einmal persistierte Daten (insbes. Ergebnisse der Code-Prüfung und Benotung) im Nachhinein nicht manipuliert werden können.
Benutzerfreundlichkeit	Ein Professor kann eine Aufgabe einem Kurs und damit allen Studenten dieses Kurses zuordnen. Es ist nicht akzeptabel, dass ein Professor eine Aufgabe jedem der ca. 30 Studenten eines Kurses einzeln zuordnet.

11. Risiken

Tabelle 7. Risiken

Thema	Begründung
Eigenentwicklung der Metriken	Dieser Punkt erscheint sehr risikoreich. Das kann ausarten, siehe Section 8.7, “Metriken”
Eigenentwicklung der internen Plagiatsprüfung	Auch dieser Punkt erscheint zum jetzigen Zeitpunkt sehr unklar und risikoreich, siehe Section 8.3, “interne Plagiatsprüfung”
Grundsätzlich	Im Grunde baut man hier eine kleine CI/CD Pipeline nach (Quellcode wird hochgeladen, es wird ein Build gestartet, dann laufen Tests und Metriken, am Ende ein Ergebnis angezeigt). Evtl würde auch eine einfachere Lösung reichen, bei der der Student den Code nicht in einer Anwendung hochlädt, sondern in einem Repository eincheckt und dann auf den Durchlauf dieser "Buildpipeline" wartet. Das komplette Ergebnis dieses Laufs mitsamt Codes könnte man dann evtl auch im Repository einchecken. Evtl würde das sogar den Audit-Regeln genügen.

12. Glossar

Begriff	Definition
BasisImage	Image aus dem der BasisContainer/Sandbox gestartet wird
BasisContainer / Sandbox	Containerinstanz beim CloudProvider, strikt von der Umgebung isoliert
Treibercode	Code, der den vom Studenten hochgeladenen Code ausruft und auf Korrektheit prüft