Fakultät für Ingenieurwissenschaften, Informatik und Psychologie
Datenbanken und Informationssysteme

Enhancing BPMNGen with Prompting Strategies for Automated BPMN 2.0 Process Model Generation

Abschlussarbeit an der Universität Ulm

Vorgelegt von:
Philipp Letschka
philipp.letschka@uni-ulm.de
1050994

Gutachter:
Prof. Dr. Manfred Reichert

Betreuer:
Luca F. Hörner

2025

Fassung 23. Dezember 2025

Satz: PDF-LaTeX 2_ε

Zusammenfassung

Diese Arbeit erweitert das System BPMNGen von Weidel [20] und Shi [16] mit neuen Prompting-Strategien zur automatisierten Generierung von BPMN 2.0 Prozessmodellen mittels Large Language Models. Zunächst wird der ursprüngliche, auf der OpenAI-Assistants-API basierende Ansatz vollständig neu strukturiert und in ein objektorientiertes Framework überführt, das unterschiedliche Anbieter wie ChatGPT, Gemini, Grok und Claude einheitlich integrieren kann. Durch optimierte Instructions, reduzierte Tokennutzunge sowie die Unterstützung mehrerer Formate wird die Qualität und Konsistenz der erzeugten Diagramme verbessert. Darüber hinaus führt die Arbeit einen Konversationsmodus auf Basis von Chain-of-Thought ein, der interaktive Gespräche, Rückfragen und iterative Diagrammbearbeitung ermöglicht. Ergänzend werden Funktionen für Datei-Uploads via Base64-Data-URLs, Streaming über Server-Sent Events sowie neue Techniken wie Diagramm-Sampling und Reflective Prompting implementiert. Eine Performanzanalyse zeigt, dass die vorgeschlagenen Anpassungen die Qualität der Diagramme, die Geschwindigkeit der Generierung und die Kosten optimieren. Insgesamt entsteht ein flexibles BPMN Modellierungssystem, das den Einsatz von LLMs zur Prozessmodellierung deutlich verbessert.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Business Process Model and Notation (BPMN) ist ein leistungsfähiges Werkzeug zur anschaulichen Darstellung von Geschäftsprozessen. Es bietet eine standardisierte Methode zur Visualisierung von Abläufen, erleichtert deren Analyse und Optimierung. Ein zentrales Problem ist jedoch die Umsetzung von einem Prozess in ein Diagramm. Deshalb werden Prozesse häufig nur in Textform beschrieben, was schnell einen Überblick bietet, aber nur schwer zu verstehen und nicht visuell darstellbar ist.

Mit modernen Large Language Models (LLMs) eröffnet sich hier eine neue Möglichkeit. Diese Modelle können Texte verstehen und in geeignete Formate für Diagramme umwandeln, wodurch sowohl Experten als auch Fachfremde bei der Prozessmodellierung unterstützt werden.

Wie dies umgesetzt werden kann zeigte bereits Weidl [20] in seiner Arbeit zur Erstellung von BPMN Diagrammen mit Hilfe von ChatGPT Assistants. Hierbei wurde ein eigenes JSON Schema entworfen, in dem ChatGPT das Diagramm beschreiben soll, welches dann zu offiziellem BPMN XML geparst wird um es dann mit einem BPMN Viewer [1] anzuzeigen. Dieses Projekt wurde dann von Shi [16] weiter verbessert. Dabei wurde eine vollständige Client-Server Architektur implementiert, sowie die Instructions des Assistants grundlegend verbessert.

Im bisherigen Projekt zeigen sich jedoch auch einige Möglichkeiten zur Verbesserung. Die Diagramme werden nur einmalig erzeugt und können nicht wirklich interaktiv angepasst werden. Außerdem ist das Prompting sehr statisch aufgebaut, sodass die KI nicht nachfragen oder auf vorherige Nachrichten eingehen kann. Das

---

[1] `https://github.com/bpmn-io/bpmn-js`

eigene JSON-Format ist zwar funktional, aber fehleranfällig und für die KI schwer zu lernen. Auch die Unterstützung weiterer KI-Modelle war nicht vorgesehen, und Funktionen wie Datei-Uploads, Streaming oder das gleichzeitige Erzeugen mehrerer Diagramme können noch hinzugefügt werden. Diese Arbeit setzt genau hier an und soll das System in diesen Punkten verbessern.

1.2 Problemstellung und Zielsetzung

Das Ziel dieser Arbeit ist es, die Art und Weise zu verbessern, wie Nutzende mit einem KI-Modell zusammenarbeiten, um BPMN-Diagramme zu erstellen. Frühere Ansätze haben meist nur ein einziges Diagramm erzeugt, ohne die Möglichkeit, Rückfragen zu stellen oder gemeinsam Schritt für Schritt an einem Prozess zu arbeiten. In der Realität entsteht ein gutes Prozessmodell jedoch selten auf den errsten Versuch. Es muss in der Regel angepasst, erweitert und überarbeitet werden.

Damit die KI bessere Ergebnisse liefern kann, braucht sie jedoch klare Anleitungen und wirksame Prompting-Strategien. Einfache Anweisungen zur Diagrammerstellung reichen meistens nicht aus, denn das Modell muss verstehen, wie sich der Nutzer das Diagramm wirklich vorstellt und dabei Unklarheiten erkennen und gegebenenfalls nachfragen können. Techniken wie Chain of Thought oder Reflective Prompting helfen dabei, indem sie der KI dabei helfen, das gewünschte Ergebins zu erzielen. Auch das Erzeugen mehrerer Diagramm-Varianten (Sampling) kann sinnvoll sein, weil so vom Nutzenden das beste Ergebnis ausgewählt werden kann.

Hinzu kommt, dass es inzwischen viele verschiedene KI-Modelle gibt, die alle unterschiedliche Stärken haben. Ein System wie das BPMNGen sollte deshalb mehrere Anbieter unterstützen, damit je nach Nutzervorstellung und nach aktuellen Stand der LLMs das passende Modell ausgewählt werden kann.

Außerdem spielt die Benutzerfreundlichkeit eine wichtige Rolle. Funktionen wie das Hochladen von Dateien, das schrittweise Streamen von Antworten oder ein dialogartiges Verhalten der KI verbessern die Nutzerefahrung.

Diese Arbeit untersucht daher, wie solche Prompting Methoden, wie man sie von modernen Chatbots kennt, in BPMNGen integriert werden können, um die KI unterstützte Erstellung und Bearbeitung von BPMN-Diagrammen besser zu machen.

Hierfür werden die genannten Verbesserungsideen als Ziel dieser Arbeit gesetzt.

1.3 Struktur der Arbeit

Die Arbeit gliedert sich in mehrere Kapitel, die systematisch aufeinander aufbauen. Kapitel 2 behandelt die theoretischen Grundlagen, insbesondere BPMN Grundlagen sowie die verwendeten Prompting- und Interaktionstechniken.

Kapitel 3 bildet den praktischen Teil der Arbeit. Dort wird die bestehende BPMNGen Architektur analysiert und anschließend alle genannten Verbesserungsideen implementiert. Dazu gehört die Umstrukturierung zu einem objektorientierten System, das mehrere KI-Anbieter unterstützt und generell besser wartbar und erweiterbar ist. Außerdem werden Verbesserungen am Prompting gezeigt, darunter optimierte Instructions, neue Modi, das Einbinden von Konversationsverlauf und Diagrammzustand, Datei-Uploads mit Base64 Data URLs, Streaming über SSE, die Kombination von Text- und Diagrammausgaben sowie Techniken wie Reflective Prompting und Diagramm-Sampling.

Kapitel 4 untersucht die Performanz des erweiterten Systems. Dazu werden die erzeugten Diagramme auf ihre Qualität bewertet, etwa im Hinblick auf Vollständigkeit, Konsistenz und Struktur. Zusätzlich wird die Geschwindigkeit, sowie die Kosten der Generierung analysiert, die durch verschiedene Prompting-Strategien, Modelle und Formate entstehen.

2 Theoretische Grundlagen

2.1 Business Process Model and Notation 2.0

Business Process Model and Notation (BPMN 2.0) [5] ist eine standardisierte grafische Sprache, die verwendet wird, um Geschäftsprozesse so zu dokumentieren, dass sie sowohl für Fachanwender als auch für technische Entwickler leicht verständlich sind. Ähnlich wie ein Architekt Baupläne nutzt, um Gebäude darzustellen, verwenden Business-Analysten BPMN 2.0, um visuelle Darstellungen organisatorischer Arbeitsabläufe und Abläufe zu erstellen.

BPMN 2.0 Diagramme[5] zeigen die Abfolge von Tätigkeiten von Anfang bis Ende, einschließlich was passiert, wann es passiert und wer jede Aufgabe ausführt. Die Hauptkomponenten von BPMN 2.0 sind:

1. **Pools and Lanes:**
    - **Pools** repräsentieren Teilnehmer eines Geschäftsprozesses, z. B. eine gesamte Organisation, eine Abteilung oder eine Rolle.
    - **Lanes** sind Unterteilungen innerhalb von Pools, die spezifische Rollen oder Abteilungen darstellen. Sie helfen, Verantwortlichkeiten zu klären und den Prozess übersichtlich zu strukturieren.
2. **Activities:**
    - **Tasks**: Abgrenzbare Arbeitsschritte, die nicht weiter unterteilt werden können.
    - **Sub-Processes**: Komplexe Aktivitäten, die in mehrere Tasks unterteilt werden können und eigene detaillierte Abläufe enthalten.
    - **Call Activities**: Verweise auf wiederverwendbare Prozesse oder Sub-Prozesse, die an anderer Stelle definiert sind.

3. **Events:**
    - **Start Events**: Beginn eines Prozesses.
    - **Intermediate Events**: Auftretende Ereignisse während des Prozesses, die den Ablauf beeinflussen können.
    - **End Events**: Beenden eines Prozesses.
    - **Message Events, Timer Events, Error Events, Conditional Events**: Spezialisierte Ereignisse, die Nachrichten, Zeitpläne, Fehler oder Bedingungen repräsentieren.
4. **Gateways:**
    - **Exclusive Gateway (XOR)**: Nur ein Pfad wird gewählt.
    - **Parallel Gateway (AND)**: Alle Pfade werden gleichzeitig durchlaufen.
    - **Inclusive Gateway (OR)**: Einer oder mehrere Pfade werden durchlaufen.
    - **Event-based Gateway**: Entscheidung basierend auf einem Ereignis.
5. **Flows:**
    - **Sequence Flows**: Abfolge von Aktivitäten innerhalb eines Pools.
    - **Message Flows**: Kommunikation zwischen Pools oder Prozessbeteiligten.
    - **Association Flows**: Verknüpfung von Artefakten oder Datenobjekten mit Aktivitäten.
6. **Data Objects:** Repräsentieren Informationen, die in einem Prozess verwendet oder erzeugt werden, z. B. Dokumente, Datenbanken oder Formulare.
7. **Artifacts:** Zusätzliche Elemente zur Prozessdokumentation, wie *Groups* (zur visuellen Gruppierung) oder *Text Annotations* (Kommentare oder Beschreibungen).

Für die Chatbot-Implementierung ist das Verständnis dieses Systems entscheidend, da alle Elemente diesen Konventionen entsprechen müssen, damit sie in dem Tool bpmn.js [1] korrekt dargestellt werden.

---

[1] `https://github.com/bpmn-io/bpmn-js`

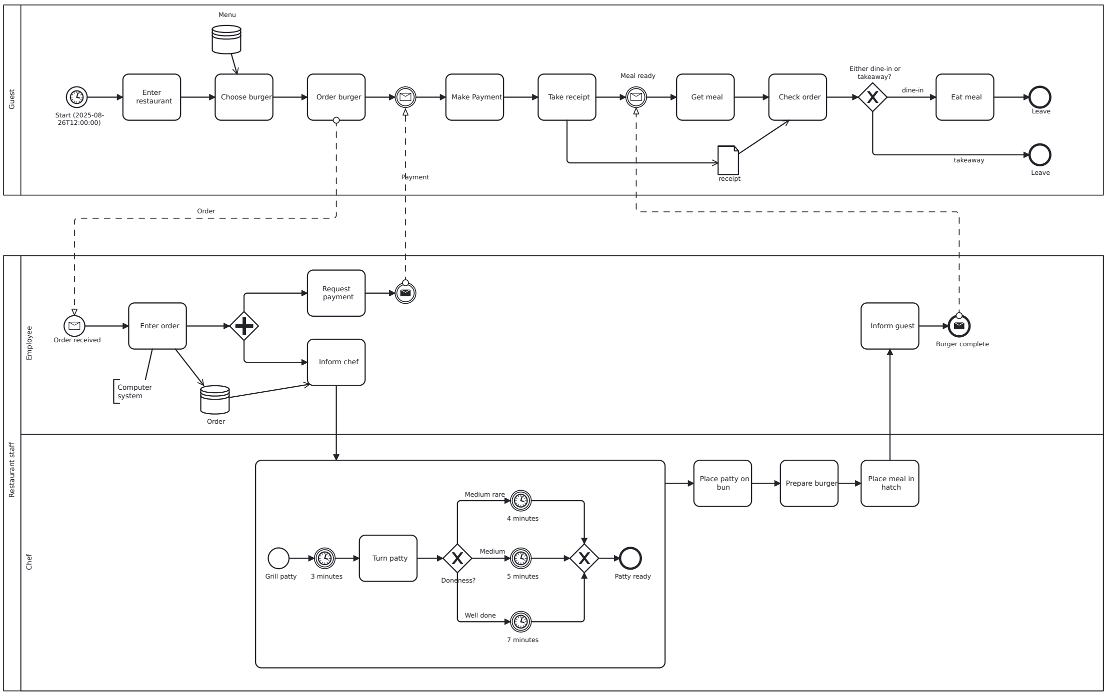

Abbildung 2.1: BPMN 2.0 [5] Diagramm mit allen relevanten Elementen

2.2 Large Language Models

Large Language Models (LLMs) sind Systeme, die auf der Verarbeitung und Generierung natürlicher Sprache spezialisiert sind. Das Konzept wurde 2017 von Vaswani et al. [18] eigeführt. Sie basieren auf Deep-neural Networks, insbesondere auf Transformer-Architekturen, die große Mengen an Textdaten analysieren, um Muster, Strukturen und Zusammenhänge in der Sprache zu erkennen. Durch dieses Training können LLMs sowohl Texte verstehen als auch neue Texte generieren.

Ein LLM wird typischerweise durch überwachtes Lernen trainiert, wobei das Ziel darin besteht, das nächste Wort oder den nächsten Satz in einem gegebenen Kontext korrekt vorherzusagen. Moderne LLMs, wie GPT-Modelle, verfügen über Milliarden von Parametern, die ein Sprachverständnis und die Fähigkeit zur Textgenerierung besitzen.

Die Anwendungsbereiche von LLMs sind vielfältig. Sie reichen von Textgenerierung, Übersetzungen und Zusammenfassungen über Frage-Antwort-Systeme bis hin zu Chatbots und automatisierten Prozessunterstützungen.

LLMs können genutzt werden, um Geschäftsprozesse in BPMN-Diagramme zu überführen. Dazu analysiert das Modell natürliche Sprache, wie z. B. Prozessbeschreibungen, Anweisungen oder Anforderungen, und wandelt diese in strukturierte BPMN-Elemente wie Tasks, Events, Gateways, Pools und Lanes um. LLMs können dabei sowohl die logische Abfolge der Prozessschritte erkennen als auch Verzweigungen sowie Kommunikationsflüsse zwischen Beteiligten identifizieren.

In der Praxis erfolgt dies oft in Form einer Text-zu-BPMN-Übersetzung, bei der das LLM die Prozessinformationen in eine JSON- oder XML-Repräsentation überführt, die anschließend von Tools wie bpmn.js visualisiert werden kann. Auf diese Weise können LLMs die Erstellung von Prozessmodellen erheblich beschleunigen, Standardisierung fördern und auch komplexe Abläufe automatisch konsistent darstellen.

2.3 Chain of Thought

Chain of Thought (CoT) beschreibt eine Methode, bei der ein Large Language Model (LLM) seine Gedankenschritte offenlegt und erklärt, wie es zu einer bestimmten Antwort kommt. Statt nur ein Ergebnis zu liefern, zeigt das Modell also den Weg dorthin, ähnlich wie ein Mensch, der seine Überlegungen laut ausspricht. Diese Vorgehensweise ist besonders hilfreich bei Aufgaben, die mehrere Denkschritte erfordern, etwa beim Lösen von Problemen, beim Strukturieren von Informationen oder beim Verstehen komplexer Anweisungen. Diese Vorgehensweise wurde von Wei et al. grundlegend untersucht [19].

Im Kontext der Unterhaltung zwischen Mensch und KI führt Chain of Thought dazu, dass das Modell transparenter und nachvollziehbarer reagiert. Die KI kann Gedankengänge ausformulieren, Entscheidungen begründen und schwierige Themen Schritt für Schritt erklären. Dadurch entsteht ein natürlicherer Dialog, da der Benutzer nicht nur das Ergebnis sieht, sondern auch versteht, wie die KI dorthin gelangt ist. Gleichzeitig hilft diese Technik dem Modell selbst, bessere Antworten zu geben, weil es Zwischenschritte bewusster berücksichtigt und Fehler eher vermeiden kann.

Für die Erstellung von BPMN-Diagrammen ist diese Vorgehensweise besonders wertvoll. BPMN erfordert eine klare Struktur von Ereignissen, Aufgaben, Gateways und Abläufen. CoT ermöglicht es ChatGPT, Prozessbeschreibungen in einzelne

Handlungsschritte zu zerlegen, diese sinnvoll anzuordnen und anschließend die passenden BPMN-Elemente daraus abzuleiten. So lässt sich Schritt für Schritt erarbeiten, welche Tasks benötigt werden, wo Entscheidungen auftreten und wie die Kommunikation zwischen Akteuren oder Abteilungen abgebildet werden muss. Chain of Thought macht die Diagrammerstellung dadurch deutlich präziser, transparenter und konsistenter.

2.4 Streaming

Streaming bezeichnet die kontinuierliche Übertragung von Daten zwischen einem Sender und einem Empfänger, wobei die Daten in aufeinanderfolgenden Teilen statt als komplette Datei oder Nachricht übertragen werden. Dieses Verfahren ermöglicht es, dass Informationen bereits während der Generierung oder Übertragung verarbeitet und angezeigt werden können, wodurch die Latenz für den Nutzer deutlich reduziert wird. Streaming findet Anwendung in vielfältigen Bereichen, etwa bei Audio- und Videoinhalten, Live-Datenübertragungen oder auch der Ausgabe von Ergebnissen von KI. Wie dieses Streaming effizient umgesetzt werden kann, wurde von Xiao et al. bei der International Conference on Representation Learning 2024 (ICLR 2025) [21] gezeigt. Technisch basiert Streaming häufig auf asynchronen Datenströmen, bei denen die empfangenen Daten in Echtzeit analysiert und weiterverarbeitet werden. Zu den verbreiteten Übertragungsprotokollen zählen HTTP-basierte Verfahren wie Server-Sent Events (SSE) [4] oder WebSockets, die eine kontinuierliche, bidirektionale Kommunikation ermöglichen. Insgesamt steigert Streaming die Geschwindigkeit der Datenverarbeitung und verbessert die Nutzererfahrung durch eine Anzeige des aktuellen Zustands in Echtzeit.

Im Kontext von KI-Systemen ermöglicht Streaming die Übersendung von Modellantworten an den Nutzer noch während die Generierung der Daten läuft. Dies ist besonders relevant bei Modellen, die unter anderem auch Text erzeugen. Technisch erfolgt dies meist über asynchrone Streams, die Datenpakete kontinuierlich übertragen. Die Textfragmente werden hier in der Regel auch `Delta` genannt. Auf diese Weise lässt sich die Benutzererfahrung deutlich verbessern, da Rückmeldungen sofort sichtbar sind und Wartezeiten minimiert werden.

2.5 Base64

Base64 ist ein Kodierungsverfahren, das Binärdaten in eine reine Textdarstellung überführt, um sie über textbasierte Protokolle wie HTTP mit unter anderem JSON zuverlässig übertragen zu können. Da viele APIs und Webschnittstellen ausschließlich Text unterstützen oder die Übermittlung binärer Inhalte erschweren, bietet Base64 eine Möglichkeit, Dateien plattformunabhängig und ohne zusätzliche Infrastruktur weiterzugeben. Die Kodierung funktioniert, indem jeweils 24 Bit (drei Byte) in vier 6-Bit-Gruppen umgewandelt und anschließend als ASCII-Zeichen dargestellt werden. Dadurch vergrößert sich die Datenmenge zwar um etwa ein Drittel, sie wird jedoch vollständig textkompatibel. [17]

Im Kontext von Webanwendungen werden Base64-kodierte Inhalte häufig in Form sogenannter `Data URL`s [3] übertragen. Eine Data URL bettet die kodierten Daten direkt in einen einzigen Zeichenstring ein, der aus vier Komponenten besteht:

1. einem Präfix zur Typangabe (data:)
2. dem MIME-Typ der Datei (z. B. image/png;)
3. der Kodierungsart (base64)
4. dem eigentlichen Datenbereich

Eine Data URL mit Base64 Encoding entspricht somit typischerweise der Form `data:<mime-type>;base64,<kodierte-daten>`. Browser und APIs können diese Darstellung unmittelbar wieder in eine gültige Datei zurückkonvertieren, ohne dass separate Dateipfade, temporäre Speicherorte oder Upload-Endpunkte notwendig sind.

Für KI-Systeme stellt dieses Verfahren einen effizienten Weg dar, Dateien wie Bilder oder Dokumente direkt in die Prompt-Struktur zu integrieren. Die Base64 Data URL kann als gewöhnlicher String an das Backend übermittelt und ohne weiteren Zwischenschritt an die API des KI-Anbieters weitergereicht werden. Dies reduziert die Komplexität der Implementierung und ermöglicht eine Unterstützung verschiedener Dateitypen. Durch diese Eigenschaften eignet sich Base64 in Kombination mit Data URLs optimal für eine unkomplizierte Übertragung von Dateien innerhalb von KI basierten Anwendungen.

2.6 Reflective Prompting

Reflective Prompting ist eine Strategie, bei der ein KI-Modell dazu aufgefordert wird, seine eigenen Ergebnisse zu überprüfen. Im Gegensatz zum klassischen Prompting, das direkt eine fertige Ausgabe erzeugt, arbeitet Reflective Prompting in zwei voneinander getrennten Phasen: Zunächst erstellt das Modell einen ersten Entwurf, eine interne Einschätzung oder einen vorläufigen Vorschlag. Anschließend reflektiert das Modell diesen Entwurf, prüft auf mögliche Fehler, Unklarheiten und logische Inkonsistenzen und verbessert daraufhin die Antwort. [22]

Durch die Einbeziehung einer Reflexionsphase steigt die Wahrscheinlichkeit, dass das Modell eigene Ungenauigkeiten erkennt, fehlende Teile ergänzt und konsistentere und qualitativ bessere Ergebnisse liefert. Reflective Prompting hat sich insbesondere bei Aufgaben bewährt, die mehrstufiges Denken, komplexe Entscheidungsfindung oder strukturiertes Argumentieren erfordern.

Gleichzeitig bringt diese Technik auch Herausforderungen mit sich. Da das Modell zusätzliche Denk- und Analyseprozesse durchläuft, erhöht sich sowohl die Rechenzeit als auch der Tokenverbrauch. Dennoch überwiegt in vielen Anwendungsszenarien der Qualitätsgewinn. In der Forschung zu Large Language Models wird Reflective Prompting daher zunehmend als zentrale Methode betrachtet, um Fehlerraten zu senken und die Robustheit der Antworten zu verbessern.

2.7 BPMNGen

BPMNGen ist ein Projekt der Universität Ulm, Institut für Datenbanken und Informationssystem. Das Projekt wurde von Weidl [20] gestartet, von Shi [16] fortgeführt und von weiteren Personen und Gruppen weiterentwickelt. Es geht in dem Projekt darum einen LLM basierten Chatbot zu entwickeln, welcher möglichst interaktiv Prozessbeschreibungen in BPMN 2.0 Diagramme umwandeln kann. Dieser Chatbot ist als Webanwendung [2] und als Rest-API implementiert.

---

[2] Die Anwendung ist verfügbar unter `https://bpmngen.de`

2.7.1 Frontend

Das Frontend ermöglicht es den Nutzerinnen und Nutzern, mit dem BPMNGen System zu interagieren. Die Grundidee besteht darin, BPMN Diagramme über Chat Interaktionen mit verschiedenen KI Modellen zu generieren. Nutzer können sich über eine Login- und Registrierungsseite anmelden. Nach dem Login erhalten sie Zugriff auf ein Dashboard, in dem sie neue Chats erstellen, bestehende Chats anzeigen, verwalten und fortsetzen können. Dabei haben sie die Möglichkeit, zwischen verschiedenen KI Modellen, Ausgabeformaten und Modi für die BPMN Diagrammgenerierung zu wählen. Zusätzlich können Dateien als Eingabe für die KI Modelle bereitgestellt werden, beispielsweise bereits vorhandene BPMN Diagramme. Die Chat Interaktion kann sowohl über Texteingabe als auch über Spracheingabe erfolgen. Das Frontend ist mit Angular umgesetzt und nutzt die Bibliothek bpmn-js zur Darstellung und Bearbeitung von BPMN Diagrammen.

In dieser Arbeit wird nicht weiter auf das Frontend eingegangen, sondern fokusiert ausschließlich auf dem Backend.

2.7.2 Backend

Das Backend stellt den Zugriff auf mehrere KI-APIs, sowie eine Nutzer- und Threadverwaltung bereit. Die Implemtierung ist darauf ausgelegt, BPMN-Diagramme aus Benutzereingaben zu generieren. Diese Eingaben können in Form von Text, Bildern, Dokumenten oder natürlicher Sprache erfolgen. Der Server kann Create- und Update-Anfragen für Diagramme entgegennehmen und leitet diese an die KI-API weiter. Die Antwort des Servers hängt dabei vom gewählten KI-Modell und dem angeforderten Ausgabeformat ab. Alle generierten Diagramme werden in einer durch Prisma erzeugten Postgres Datenbank gespeichert. Das Backend basiert auf Express.js, welches als Webserver Framework für die Verarbeitung von HTTP-Anfragen, Routing und Middleware eingesetzt wird.

2.8 Thread

Ein Thread ist ein fortlaufender Gesprächsverlauf zwischen einem Nutzer und einem KI-Modell. In einem Thread werden alle Nachrichten, also Fragen, Antworten und Kontext, gespeichert, sodass das Modell den Zusammenhang früherer Beiträge berücksichtigen kann. Dadurch kann die KI besser reagieren, sich auf bereits Gesagtes beziehen und ein Gespräch weiterführen, anstatt jede Anfrage isoliert zu behandeln.

3 Umstrukturierung und Innovation

Als erstes gilt es herauszufinden, wie der Teil des Code, der zuständig für das Prompting ist, verbessert werden kann. Das Projekt von Weidl [20] und Shi [16] benutzt zur Erstellung von BPMN Diagrammen die OpenAI API und verwendet hier die Technologie der Assistants. [1]

3.1 Generelle Umstrukturierungen

Da der Code nur für seinen (bisherigen) speziellen Anwendungsfall konzipiert ist, können hier einige Verbesserungen gemacht werden.

3.1.1 Objektorientierter Ansatz

Das Ziel ist es, den Code einfach erweiterbar und wartbar zu machen. Hierfür ist es wichtig, den Code möglichst schnell an Änderungen der OpenAI API anpassen zu können. Um das zu erreichen, wird ein objektorientierter Ansatz gewählt. Die objektorientierte Programmierung bietet für den Aufbau des Prompting-Codes viele Vorteile und macht die Entwicklung langfristig übersichtlicher und besser wartbar. Es wird eine abstrakte Klasse `Ai` erstellt, welche die gesamte Logik des Prompting beinhält, und eine Klasse `ChatGPT` welche von der AI Klasse erbt. Die ChatGPT Klasse muss nun nur noch Methoden implementieren, welche konkret auf die aktuelle Version der API angepasst sind. Durch die Verwendung von abstrakten Methoden wie `generateContent()`, `createTitle()` oder processResponse() wird sichergestellt, dass jede konkrete Implementierung dieselbe Schnittstelle einhält, aber ihre eigenen internen Abläufe definieren kann. Dies erleichtert den Austausch

---

[1] `https://platform.openai.com/docs/assistants/overview`

und die Erweiterung von Modellen, ohne den restlichen Code verändern zu müssen. Darüber hinaus werden wiederkehrende Prozesse, etwa das Speichern von Verläufen, das Verarbeiten von Antworten oder die Konvertierung zwischen Formaten, zentral in der Basisklasse gekapselt. Falls sich die API ändert, kann dies nun einfach in der erbenden Klasse angepasst werden, ohne die dahinterliegenden Logik verändern zu müssen.

So wie in Codeausschnitt 3.1.1 sieht nun in vereinfachter Variante die Klasse für die OpenAI API aus. Es gibt eine Methode `mapPromptInput();` um den Prompt in das richtige Format der API zu bringen, `generateContent();` um den eigentlichen API Aufruf durchzuführen und `processResponse();` um die Antwort der API auszulesen.

```
export class ChatGPT extends Ai {
  openai = new OpenAI({
    apiKey: OPENAI_API_KEY,
  });
  assist = await openai.beta.assistants.retrieve("asst_...");

  protected mapPromptInput(input) {
    return {
      input: input.prompt,
      model: this.model,
    };
  }

  protected async generateContent(input) {
    return await openai.beta.threads.runs.createAndPoll(
      thread_id,
      { assistant_id: assist.id },
      { role: "user", content: input }
    );
  }

  protected processResponse(response) {
    return response.output_text.toString();
  }
}
```

Codeauschnitt 3.1.1: ChatGPT Klasse

3.1.2 Von Assistants zu Responses

Da OpenAI angekündigt hat, die Assistants API einzustellen, erweisen sich die im vorherigen Kapitel beschriebenen strukturellen Änderungen als besonders sinnvoll. OpenAI empfiehlt einen Umzug zu ihrer neueren Responses API [2] [3]. Dies ist nun recht einfach umzusetzten, da nur die elementaren Methoden der API angepasst werden müssen. Wie nun zum Beispiel die neue Methode `generateContent();` für die Responses API aussieht, kann man in Abbildung 3.1.2 sehen.

```
protected async generateContent(input) {
  return this.openai.responses.create(input);
}
```

Codeauschnitt 3.1.2: generateContent()

Einer der zentralen Unterschiede zwischen der Assistants- und der Responses-API besteht darin, dass die System Instructions bei der Verwendung der Responses API manuell übergeben werden müssen. Dadurch ist es notwendig, die entsprechenden Anweisungen bei jeder Anfrage erneut mitzusenden. Dies bringt jedoch nicht nur zusätzlichen Aufwand mit sich, sondern eröffnet auch neue Möglichkeiten: Die Instructions können flexibel und situationsabhängig angepasst werden, wodurch sich das Verhalten des Modells dynamisch steuern lässt. Im folgenden Abschnitt wird gezeigt, wie dieser Ansatz weiter verbessert und effizienter gestaltet werden kann.

3.1.3 Verbesserung der Instructions

Da die Instructions nun manuell mit jeder Anfrage übergeben werden, bietet sich die Möglichkeit, deren Aufbau gezielt zu optimieren. Ziel dieser Optimierung ist es, die Anzahl der benötigten Input-Tokens zu reduzieren, ohne dabei Qualität einzubüßen. Im besten Fall wird die Ausgabequalität sogar verbessert. Der Assistant erhält als Grundlage zwei PDF-Dateien, die BPMN-Diagramme und den BPMN 2.0 Standard im Detail beschreiben, sowie zwei Textdateien: Eine mit der Definition des verwendeten JSON-Formats und eine mit allgemeinen Regeln zum Aufbau der

---

[2] `https://platform.openai.com/docs/api-reference/responses`

[3] `https://platform.openai.com/docs/assistants/migration`

Diagramme. Die beiden PDF-Dokumente umfassen zusammen mehr als 10 MB und über 100 Seiten Text. Da ChatGPT bereits ein solides Grundverständnis von BPMN-Diagrammen besitzt, werden diese umfangreichen Dateien aus den Instructions entfernt. Diese Reduktion wirkt sich nicht negativ auf die Ergebnisqualität aus. Dies wird auf Seite 19 gezeigt. Dadurch lassen sich eine große Zahl an Tokens sowie Rechenzeit und Kosten einsparen.

Die beiden verbliebenen Textdokumente werden anschließend zusammengeführt und in ein einheitliches, strukturiertes Format gebracht. Alle Regeln sind in einer geordneten Liste zusammengefasst und durch `Structured-Prompting` klar und maschinenlesbar gestaltet. Ergänzend werden den Instructions zwei illustrative Beispiele hinzugefügt: Zum einen ein minimales Beispiel, das den grundsätzlichen Aufbau des JSON-Formats verdeutlicht und die obligatorischen Elemente zeigt. Zum anderen ein umfangreicheres, praxisnahes Beispiel, das ein vollständiges BPMN-Diagramm mit allen relevanten und unterstützten Komponenten abbildet. Dieses `Few-Shot-Prompting`, oder in diesem Fall `Two-Shot-Prompting`, sorgt dafür, dass der Assistant sowohl einfache als auch komplexe Diagramme interpretieren und reproduzieren kann. Diese Technik ist auch bereits bei anderen BPMN Chatbots getestet worden und hat sich als sinnvoll erwiesen. [14, 10, 23]

3.2 Formatauswahl

Bisher wird die KI angewiesen, das Diagramm in einem eigens definierten JSON-Format zu erzeugen. Dieses Format wurde jedoch speziell für diesen Anwendungsfall entworfen und existiert in dieser Form nicht offiziell. Entsprechend konnte das Modell während des Trainings kein Vorwissen darüber erwerben, sondern muss das Format ausschließlich auf Grundlage der bereitgestellten Instructions erlernen. Dadurch besteht die Möglichkeit, dass Fehler auftreten, etwa dann, wenn die Anweisungen unvollständig sind oder dem Modell bestimmte Kontextinformationen fehlen.

Um dieses Problem zu vermeiden, wird künftig die Option ergänzt, dass die KI ihre Ausgabe auch direkt im offiziellen Standardformat erzeugen kann. Das standardisierte Format für BPMN 2.0 Diagramme ist XML, zu dem umfangreiche Dokumentationen und etablierte Werkzeuge existieren. [5] Dennoch bietet das eigens entwickelte JSON-Format einen entscheidenden Vorteil: Es besitzt eine deutlich höhere

Informationsdichte und lässt sich dadurch schneller und effizienter verarbeiten. Aus diesem Grund haben sich einige anderen Implementierungen eines Text zu BPMN durch LLM dafür entschieden, ein Zwischenformat wie das hier genannte JSON zu verwenden. Dazu gehört AutoBPMN.AI [6], welche auf das Format Mermaid gesetzt haben oder NaLa2BPMN [12], welche auch ein JSON implementieren. Beide Varianten, sowohl das direkte vollständige Generieren als auch die Generierung mit Zwischenmodell, haben ihre jeweiligen Stärken und Schwächen. Diese sind in Tabelle 3.1 gegenübergestellt.

	JSON-Format	**XML-Format**
Vorteile	hohe Informationsdichte, geringer Tokenverbrauch, schnelle Generierung.	standardisiert, gut dokumentiert, weit verbreitet, einfachere Instructions
Nachteile	kein Standard, hoher Lernaufwand, komplizierter Konvertierungsalgorithmus.	hoher Tokenverbrauch, umfangreiche Syntax, unübersichtlicher, höhere Kosten, längere Generierung.

Tabelle 3.1: Vergleich der Vor- und Nachteile der unterstützten Ausgabeformate

Für die Weiterentwicklung ist das Ziel, ein flexibles System, das eine dynamische Auswahl des Ausgabeformats besitzt. Dadurch kann der Nutzer selbst entscheiden, welches Format im jeweiligen Anwendungsfall die besseren Ergebnisse liefert. Da sich die Formatwahl ähnlich wie die Wahl des verwendeten Modells direkt auf die Qualität der Ergebnisse auswirkt, wird die Auswahlmöglichkeit direkt in die Modellkonfiguration integriert. So kann beispielsweise zwischen Varianten wie ‘gpt-4.1-mini (xml)’ und ‘gpt-4.1-mini (json)’ gewählt werden. Alternativ kann das Format auch im separaten `format` Parameter angegeben werden. Wird kein Format angegeben, erfolgt die Ausgabe standardmäßig im XML-Format.

Eine Anfrage sieht damit z. B. aus wie in Codeausschnitt 3.2.1.

Bei einer Anfrage kann dann die jeweilige AI über eine Map

```
const availableGPTs: Map<string, Ai>
```

```
// POST /threads
{
  "inputString": "Bitte generiere mir ein BPMN Diagram,
      welches den Ablauf in einem Restaurant zeigt",
  "model": "gpt-5 (xml)",
  "format": "xml"
}
```

Codeauschnitt 3.2.1: Post Request an /threads mit Format

zugeordnet werden, welche die Anfrage bearbeitet.

Diagrammupdates Die erstellten Diagramme bearbeiten zu lassen, ist ein wesentlicher Bestandteil der Software. Weidel [20] hat das Update des Diagramms so implementiert, dass ein JSON Block an das generierte Diagramm anghängt wird. In diesem Block sind die Änderungen definiert, während der Text der eigentlichen Diagrammgenerierung nicht verändert wird. Obwohl Veränderungen so sehr schnell generierbar sind, hat es leider auch manche Nachteile. Zum einen wird somit die Datei, welche auch immer an die KI mitgesendet werden muss, immer länger, wodurch auch die Anzahl an Input Tokens immer weiter steigt. Zum anderen ist es nicht möglich, dass der Nutzer manuell Änderungen an dem Diagramm vornehmen kann und die KI dann mit diesen Änderungen fortfährt.

Außerdem ist ein solcher Update Block nicht im offiziellen XML Standard definiert. Daher kann diese Technik nicht auf direkte XML-Diagramm Erstellungen angewendet werden. Für den weiteren Verlauf wird daher das Verfahren der Update Blöcke entfernt und stattdessen wird bei einer Diagrammänderung das gesamte Diagramm neu generiert. [4] Einer KI die Möglichkeit zu geben, ihre vorherigen Fehler aus dem Diagramm vollständig zu entfernen, kann sinnvoll sein, da dadurch jeglicher Fehler korrigiert werden kann und die KI zum Beispiel auch wieder Teile entfernen kann, ohne dass die Diagrammdefinition länger wird. Dies haben auch andere ähnliche Projekte so entschieden. [6, 14]

---

[4] Mehr dazu im Abschnitt 3.4.2

Qualitätskontrolle Im Folgenden soll gezeigt werden, dass die Qualität durch die gemachten Veränderungen nicht nachgelassen hat. Dafür wird ein Diagramm sowohl für den alten Assistant als auch für die gemachten Änderungen erstellt. Für alle wird die einheitliche Prozessebschreibung, Dokument 1.3 aus dem PET Dataset [1], sowie GPT-4.1 verwendet (3.2.1)

> The Evanstonian is an upscale independent hotel. When a guest calls room service at The Evanstonian, the room-service manager takes down the order. She then submits an order ticket to the kitchen to begin preparing the food. She also gives an order to the sommelier (i.e., the wine waiter) to fetch wine from the cellar and to prepare any other alcoholic beverages. Eighty percent of room-service orders include wine or some other alcoholic beverage. Finally, she assigns the order to the waiter. While the kitchen and the sommelier are doing their tasks, the waiter readies a cart (i.e., puts a tablecloth on the cart and gathers silverware). The waiter is also responsible for nonalcoholic drinks. Once the food, wine, and cart are ready, the waiter delivers it to the guest‘s room. After returning to the room-service station, the waiter debits the guest‘s account. The waiter may wait to do the billing if he has another order to prepare or deliver.

Prompt 3.2.1: Dokument 1.3 des PET Datasets [1]

Die generierten Diagramme sind in den Abbildungen B.1, B.2 und B.3 zu sehen. Wie man erkennen kann, wurde die Qualität durch die Änderungen sogar verbessert.

3.3 Dateien

Um die Qualität des Promptings weiter zu verbessern, soll eine Funktionalität implementiert werden, die es ermöglicht, auch Dateien direkt an die KI zu übermitteln. Dabei steht im Vordergrund, dass das Verfahren sowohl im Frontend als auch im Backend möglichst unkompliziert umgesetzt werden kann. Es soll keine aufwendigen oder zeitraubenden Konvertierungen erfordern und eine breite Auswahl an Dateitypen unterstützen, um die Nutzung so flexibel wie möglich zu gestalten. Nach einer Analyse der OpenAI-Dokumentation [5] zeigt sich, dass die einfachste und zu-

---

[5] `https://platform.openai.com/docs/guides/images-vision?api-mode=responses&format=base64-encoded`

gleich effizienteste Methode zur Dateiübertragung die Verwendung einer Base64 Data URL ist. Diese Variante bietet eine einfache Möglichkeit, Binärdaten wie Bilder oder Dokumente direkt in Textform zu kodieren und zu übermitteln, ohne zusätzliche Infrastruktur oder spezielle Upload-Mechanismen zu benötigen.

Eine Base64 Data URL ist eine Textdarstellung einer Datei, die direkt in eine URL eingebettet wird. [3, 17] Dabei werden die ursprünglichen Binärdaten in ein spezielles Textformat namens Base64 umgewandelt. Diese kodierten Daten beginnen typischerweise mit einer Kennzeichnung wie `data:image/png;base64,...` und enthalten danach die eigentlichen kodierten Inhalte. Der große Vorteil liegt darin, dass der Browser oder die API diesen Text einfach wieder in die ursprüngliche Datei zurückwandeln kann. Auf diese Weise lassen sich Dateien direkt in JSON API-Anfragen integrieren, ohne dass zusätzliche Anfragen oder externe Speicherorte erforderlich sind. Dieses Verfahren ist daher besonders gut geeignet, um eine einfache, schnelle und universell kompatible Dateiübertragung zu ermöglichen. Außerdem wird diese Formatierung von allen relevanten LLM Anbietern unterstützt. Dadurch, dass die Datei nun einfach als String übergeben werden kann, können wir die Datei dem Body der Anfrage hinzufügen.

Eine Anfrage sieht damit z. B. wie in Codeausschnitt 3.3.1 aus. Der String wird dann in Codeausschnitt 3.3.2 auf Validität geprüft und dann wie in Codeausschnitt 3.3.3 im richtigen Format an die OpenAI API gesendet:

```
// POST /threads
{
  "inputString": "Bitte generiere mir ein BPMN Diagram,
      welches den Ablauf in einem Restaurant zeigt",
  "model": "gpt-5 (xml)",
  "file": "data:image/png;base64,A35ekZ...",
}
```

Codeauschnitt 3.3.1: Post Request an /threads mit Datei

```
private checkBase64DataUrl(dataUrl: string): boolean {
  regex = /^data:([\w.+-]+\/[\w.+-]+)?;base64,[\w+\/]+=*$/;
  return regex.test(dataUrl);
}
```

Codeauschnitt 3.3.2: checkBase64DataUrl()

```
const imageInstructions = {
  role: "user",
  content: [
    {
        type: "input_file",
        file_url: input.getFileDataUrl() as string,
    } as ResponseInputFile,
  ],
} as ResponseInputItem
```

Codeauschnitt 3.3.3: Darstellung des Formats für die ChatGPT API

3.4 Chain of Thought

Um die Erzeugung und Interaktion rund um BPMN-Diagramme weiter zu verbessern, wird eine Technik implementiert, die als `Chain of Thought` [19] bekannt ist. Dieses Verfahren ermöglicht es dem Modell, komplexere Denkprozesse intern nachzuvollziehen und schrittweise zu argumentieren, bevor eine Antwort erzeugt wird. Dadurch kann der Dialog natürlich und strukturiert verlaufen, da der Chatbot in der Lage ist, Zusammenhänge besser zu verstehen und über mehrere Gesprächsschritte hinweg Ergebnisse zu liefern.

Diese Erweiterung erlaubt es dem Nutzer, auf vielfältige Weise mit dem Chatbot zu interagieren: Es können Fragen gestellt werden, Ideen vorgeschlagen, bestehende Diagramme erläutert oder Verbesserungsvorschläge erfragt werden. Der BPMN-Bot soll damit nicht nur ein Werkzeug für Diagrammerstellung bleiben, sondern sich zu einem vollwertigen Assistenten weiterentwickeln, der beim gesamten Modellierungsprozess unterstützt.

Darüber hinaus ist vorgesehen, dass der Chatbot selbstständig Rückfragen stellt, wenn bestimmte Angaben unvollständig, mehrdeutig oder widersprüchlich sind. So kann ein interaktiver Dialog entstehen, in dem beide Seiten zum Verständnis und zur Qualität des Diagrammes beitragen. Um dies zu ermöglichen, benötigt der Chatbot Zugriff auf den bisherigen Gesprächsverlauf sowie auf den aktuellen Zustand des jeweiligen Diagrammes. Nur durch diese Kontextkenntnis kann die KI sinnvolle Antworten generieren.

3.4.1 Implementierung eines neuen Modus

Um die Erstellung von Diagrammen für den Nutzer nicht unnötig zu verkomplizieren, bleibt die direkte Generierung eines BPMN-Diagramms weiterhin bestehen. Gleichzeitig soll jedoch mehr Flexibilität bei der Art der Interaktion geboten werden. Zu diesem Zweck wird der Anfrage ein zusätzlicher Parameter hinzugefügt, der das Antwortverhalten der KI steuert.

Über den Parameter `mode` kann festgelegt werden, in welchem Modus die KI reagieren soll. Der bisherige Modus, bei dem ausschließlich das Diagramm erzeugt wird, trägt nun die Bezeichnung `quick`. In diesem Modus erfolgt die Ausgabe direkt und ohne weiteren Dialog.

Der neu eingeführte Modus `detail` aktiviert den Chain of Thought-Ansatz. In diesem Modus verhält sich die KI dialogorientiert. Sie kann Rückfragen stellen, Überlegungen anstellen oder alternative Vorschläge anbieten, bevor das endgültige Diagramm erstellt wird. Dadurch entsteht eine interaktive Konversation, die vor allem bei komplexeren Prozessen oder unvollständigen Eingaben von Vorteil ist.

Eine Anfrage, die eine solche erweiterte Unterhaltung ermöglicht, könnte beispielsweise wie in Codeausschnitt 3.4.1 aussehen:

3.4.2 Konversationskontext

Da es nun darum geht einen Chat zu implementieren, bei dem die KI möglichst gut auf Nachrichten reagieren kann, ist es wichtig, dass die KI Zugriff auf vorherige Nachrichten sowie auf das aktuellste Diagramm hat. Wäre das nicht der Fall, könnte die KI keine Fragen über das Diagramm beantworten und auch keine richtige Unterhaltung führen, da immer nur die aktuellste Nachricht vorhanden ist. Bei LLM

```
// POST /threads
{
  "inputString": "Bitte schlag drei Prozessbeschreibungen
    vor, aus denen dann eine ausgewählt wird um ein Diagramm
    zu erstellen.",
  "model": "gpt-5 (xml)",
  "mode": "detail",
}
```

Codeauschnitt 3.4.1: Post Request an /threads mit mode

Anbietern wie OpenAI ist es notwendig, dass die Nachrichtenhistorie in der Anfrage mitgeschickt wird.

Hierfür müssen alle Nachrichten eines Threads aus der Datenbank geladen werden. Die Nachrichten werden dann auf das Wesentliche gefiltert und auf ein kompaktes Format gebracht. Die OpenAI Klasse muss dann nur noch die Nachrichten auf das von der API geforderte Format bringen und in der Anfrage mitschicken.

Da die Anfragen an die KI immer komplexer werden, wird nun eine Klasse `PromptInput` angelegt. Diese Klasse beinhaltet alle Informationen, welche der KI mitgesendet werden sollen. Bei einer Erstellung dieses Objekts können alle Rohdaten wie Instructions, Dateien oder der Chatverlauf übergeben werden, welche die Klasse dann automatisch formatiert. Ein entscheidender Schritt hierbei ist es, die Teile der Nachrichten zu entfernen, in denen die KI mit einem Diagramm geantwortet hat. Das ist wichtig, da die Diagramme viele Tokens beinhalten und eigentlich nur das aktuellste Diagramm wichtig ist. Hierbei kann es aber auch sein, dass das Diagramm noch vom Nutzer bearbeitet wurde. Um dies zu berücksichtigen, sind die Diagramme nicht Teil der Nachrichten, welche mitgesendet werden. Die Implementierung der KI-Schnittstelle kann anschließend ein Objekt der Klasse `PromptInput` entgegennehmen. Dieses Objekt dient als zentrale Datenstruktur, über die alle für die Anfrage relevanten Informationen an das Modell übergeben werden. Mithilfe vordefinierter Hilfsmethoden lässt sich der Inhalt komfortabel in das gewünschte Eingabeformat konvertieren, sodass keine manuelle Aufbereitung mehr erforderlich ist.

Die Klasse `PromptInput` verfügt über die in Abbildung 3.4.2 zu sehenden Attribute.

```
export class PromptInput {
  instructions: string[];
  history: {role: "user" | "assistant", content: string}[];
  prompt: string;
  file?: string;
}
```

Codeauschnitt 3.4.2: PromptInput Klasse

Damit ist es möglich, bestehende Nachrichtenverläufe aus der Datenbank abzurufen und automatisch in das benötigte Format zu überführen. Die Klasse übernimmt hierbei die vollständige Strukturierung der Daten, sodass diese für die Kommunikation mit der KI genutzt werden können.

Das so erzeugte `PromptInput`-Objekt kann anschließend durch interne Methoden in das finale Format umgewandelt werden, das von der KI-Schnittstelle erwartet wird. 3.4.3 Dadurch entsteht ein einheitlicher, wiederverwendbarer Datenfluss zwischen Anwendung, Datenbank und Modell, der die Wartung sowie zukünftige Erweiterungen vereinfacht.

```
protected mapPromptInput(input: PromptInput) {
  [...]
  const historyInstructions = input.history.map((item) => {
    return {role: item.role, content: item.content};
  });
  return {
    input: [systemInstructions, historyInstructions,
            userInstructions, fileInstructions],
    model: this.model,
  };
}
```

Codeauschnitt 3.4.3: mapPromptInput()

Darüber hinaus soll künftig auch die jeweils aktuellste Version des Diagramms an die Anfrage angehängt werden. Auf diese Weise erhält die KI den vollständigen Kontext und kann das bestehende Diagramm nicht nur bearbeiten, sondern auch inhaltliche Fragen dazu beantworten.

Da das aktuelle Diagramm nicht Bestandteil des eigentlichen Nachrichtenverlaufs ist, wird es in den System Instructions hinterlegt. Dieser Block wird nun als ‘Update Instruction’ bezeichnet und enthält stets die zuletzt gespeicherte Version des Diagramms. Die entsprechenden Daten werden automatisch aus der Datenbank geladen und vor dem Absenden der Anfrage in die System Instructions eingefügt. Dies ist in Codeausschnitt 3.4.4 zu sehen.

Durch dieses Verfahren ist sichergestellt, dass die KI bei jeder Interaktion auf dem neuesten Stand bleibt und Änderungen im Diagramm jederzeit konsistent nachvollziehen kann.

```
protected updateInstructions(threadID: string, format: format){
  const diagram = getLatestDiagramFromDB(threadID);
  return [`The The following diagram has already been created:
    ${format == "xml" ? diagram?.xml : diagram?.json}`]
}
```

Codeauschnitt 3.4.4: updateInstructions()

Damit hat die KI nun alle Informationen, die sie benötigt, um eine Konversation führen zu können.

3.4.3 Konversationen

Für eine vollständige Konversation benötigt die KI nun aber noch Anweisungen. Dafür wird der KI noch ein weiterer Block Instructions hinzugefügt. Diese werden als ‘Update Instructions’ bezeichnet. Diese sehen zunächst so aus:

```
[Output Language depends on prompt language]

Instead of creating a bpmn diagram, you can also chat with the
user.
Any descriptions, questions, summaries or any other type of
text the user may want, you must answer.
You may ask any questions you feel like you need to
answer to create the best bpmn diagram.
If you do not have any questions anymore, you may write a
quick summary of the agreed diagram before creating the
```

```
bpmn.
If the initial prompt is really basic, provide the user with a
creative scenario summary.

If you want to provide questions, summary or any other type of
text, that is not a diagram. Do not use JSON or XML.
Only plain Text!
You may answer with both a plaintext section and a diagram.
Do not put plain text in a JSON or XML section.
If you want to provide a description, summary or any other type
of text, that is not a diagram, send it outside the diagram.
```

Diese Instructions werden nur mitgesendet, falls der `detail` modus genutzt wird.

Wenn man den Bot nun fragt: “Bitte schlag mir ein Diagramm vor”, hat dieser geantwortet mit:

Gerne, ich schlage Ihnen ein klassisches und leicht verständliches Szenario vor: einen **Online-Bestellprozess**. Dieses Diagramm würde die Interaktion zwischen einem Kunden und einem Online-Shop darstellen und eignet sich gut, um grundlegende BPMN-Konzepte zu zeigen. **Zusammenfassung des vorgeschlagenen Diagramms:**

- **Beteiligte (Pools):**
    1. Kunde
    2. Online-Shop
- **Ablauf beim Kunden:**
    1. Der Prozess beginnt damit, dass der Kunde ein Produkt auswählt.
    2. Er gibt eine Bestellung auf, was eine Nachricht an den Online-Shop sendet.
    3. Danach wartet er auf die Bestellbestätigung.
    4. Anschließend wartet er auf die Versandbestätigung.
    5. Der Prozess endet, wenn der Kunde die Ware erhalten hat.
- **Ablauf im Online-Shop:**
    1. Der Prozess startet, sobald die Bestellung des Kunden eingeht.
    2. Die Bestellung wird geprüft.
    3. Eine Entscheidung (Gateway): Ist der Artikel verfügbar?
        - **Ja:** Die Bestellung wird bestätigt (Nachricht an den Kunden), die Ware wird verpackt und versendet. Eine Versandbestätigung wird an den Kunden gesendet.
        - **Nein:** Die Bestellung wird abgelehnt (Nachricht an den Kunden) und der Prozess endet.
    4. Nach dem Versand der Ware ist der Prozess für den Shop abgeschlossen.

Dieses Szenario zeigt die Zusammenarbeit zwischen zwei Teilnehmern, den Austausch von Nachrichten und eine einfache Geschäftsentscheidung. Sind Sie mit diesem Vorschlag einverstanden? Falls ja, erstelle ich das entsprechende BPMN 2.0 XML für Sie.

Man kann diese Konversation jetzt fortführen und beispielsweise antworten mit “Ja, bitte generier mir das Diagramm genau so wie beschrieben”. Der Bot versteht, dass er nun nicht antworten, sondern ein Diagramm erstellen soll. Daher sendet er kein Klartext, sondern direkt ein BPMN XML. Dieses sieht in diesem Beispiel wie in Abbildung 3.1 aus:

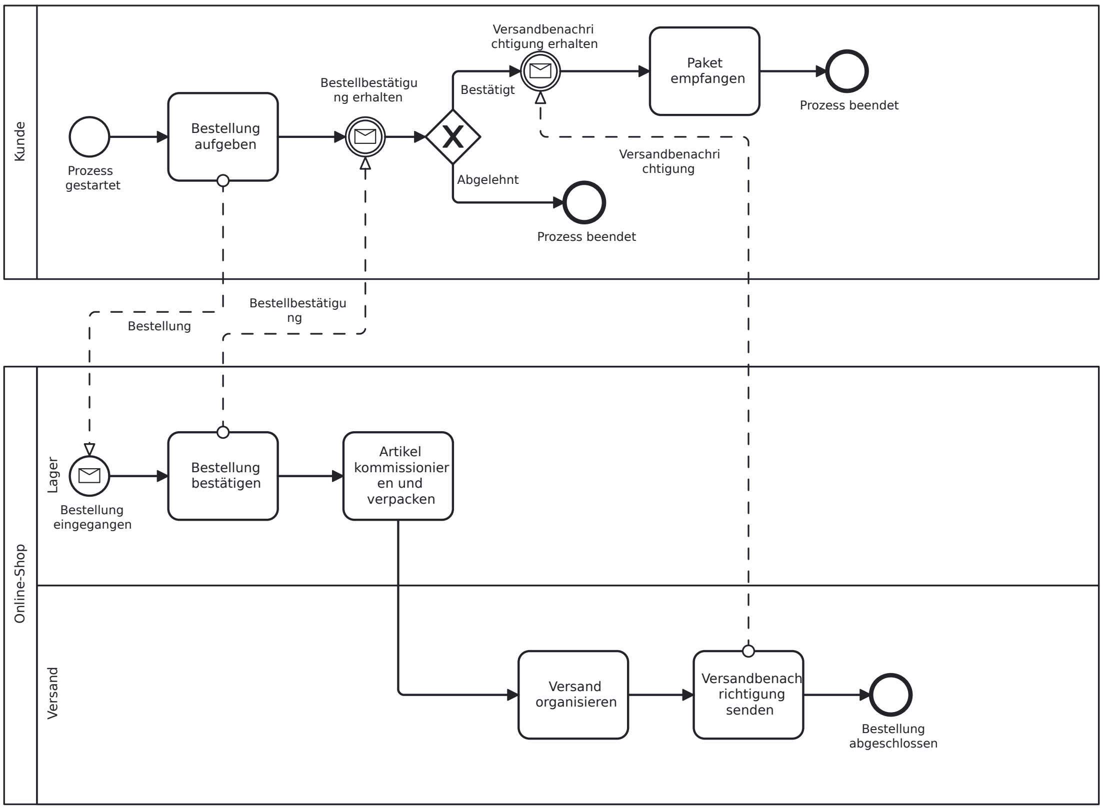

Abbildung 3.1: Generierung eines Diagrammes

Es fällt auf, dass der Bot das Gateway ‘Ist der Artikel verfügbar’ vergessen hat. Dies ist allerdings kein Problem, da das Diagramm nun weiter durch Prompts verbessern kann. “Bitte bearbeite das Diagramm, indem du das vergessene Gateway ‘Ist der Artikel verfügbar’ hinzufügst”. Der Bot erkennt nun wieder, dass keine Textantwort gewünscht ist und beginnt mit der Übersendung des neuen Diagramms (3.2).

Das Verhalten des ChatBots entspricht damit exakt den Anforderungen: Die Erstellung und Bearbeitung von Diagrammen kann vollständig interaktiv erfolgen und der Nutzer erhält abhängig vom Kontext entweder Klartext oder direkt ein BPMN-Diagramm. Im gezeigten Beispiel war die Einordnung der Antwort relativ unkompliziert, da jeweils eindeutig erkennbar war, ob die KI ausschließlich Text oder ausschließlich ein Diagramm liefern sollte.

Für die weitere Entwicklung soll der Funktionsumfang jedoch erweitert werden, sodass der ChatBot künftig auch Antworten erzeugen kann, die Klartext und Diagramm gleichzeitig enthalten. Dadurch wird es möglich, dass der Bot zunächst ei-

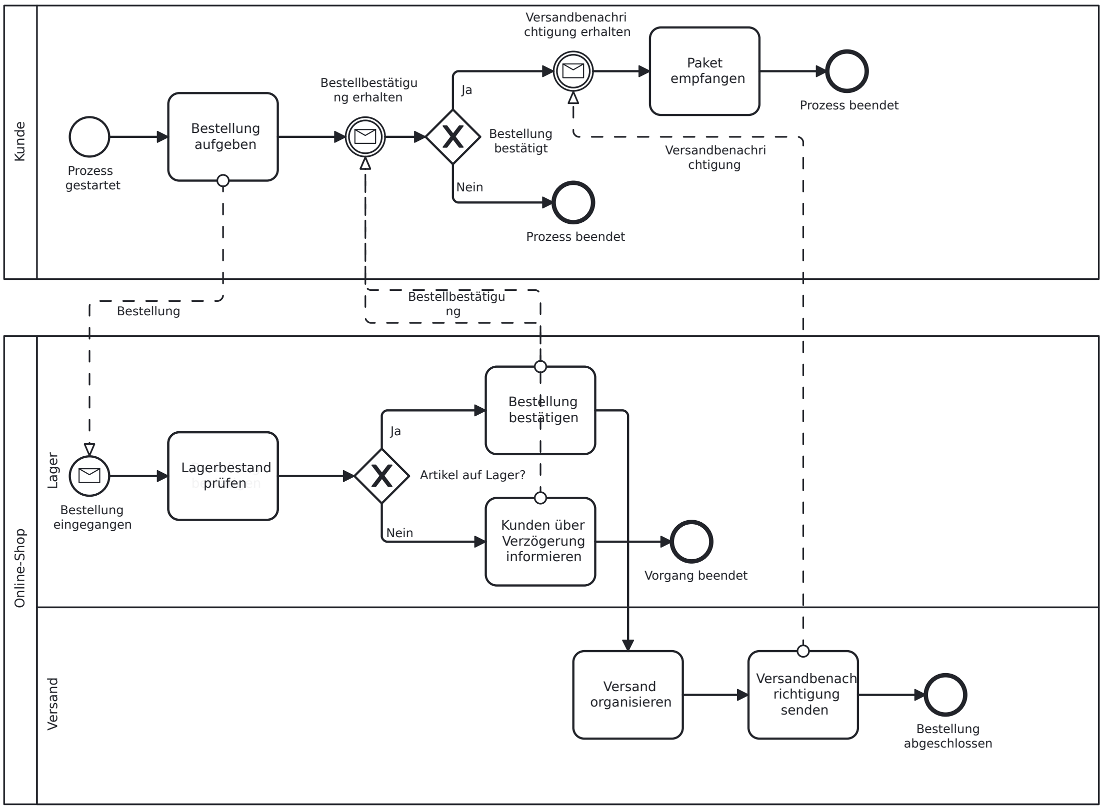

Abbildung 3.2: Überarbeitung eines Diagrammes mit dem detail mode

ne Beschreibung, Analyse oder Erklärung liefert und anschließend unmittelbar ein dazugehöriges BPMN-Diagramm generiert. Ein solcher Anwendungsfall lässt sich beispielsweise mit einer Anfrage wie „Zeig mir, was du so kannst, indem du eine Prozessbeschreibung erstellst und diese direkt in ein Diagramm umsetzt." simulieren.

In diesem Szenario entsteht eine neue Herausforderung: Die Antwort der KI besteht nicht mehr aus einer einzigen, klaren Kategorie, sondern aus zwei unterschiedlichen Inhaltstypen, die voneinander getrennt verarbeitet werden müssen. Der Klartextteil soll wie gewohnt im Chat ausgegeben werden, während der Diagrammteil in das entsprechende Ausgabeformat überführt und anschließend angezeigt wird.

Zu diesem Zweck wird ein zusätzlicher Erkennungsschritt eingeführt, der die Antwort der KI analysiert und die jeweiligen Segmente eindeutig kategorisiert. Der ChatBot muss erkennen, welche Abschnitte in natürlicher Sprache formuliert sind und welche Bestandteile ein Diagramm darstellen, das weiterverarbeitet werden soll. Dieses Verhalten ermöglicht eine deutlich flexiblere Interaktion und ermöglicht neue Möglichkeiten, besonders dann, wenn der Nutzer sowohl inhaltliche Erläuterungen als auch die direkte Umsetzung in ein BPMN-Diagramm erwartet.

Für dieses Kategorisierung wird eine Beispiel-Antwort betrachtet, welche hier zum Verständnis nicht per Markdown formatiert ist:

> Gerne, hier ist ein Vorschlag für einen einfachen, aber vollständigen Prozess:
> **Prozess:** Urlaubsantrag **Beteiligte:** Mitarbeiter, Vorgesetzter
> **Ablauf:** 1. Ein Mitarbeiter füllt einen Urlaubsantrag aus und reicht ihn ein.
> 2. Der Vorgesetzte erhält den Antrag und prüft ihn.
> 3. Der Vorgesetzte entscheidet, ob der Antrag genehmigt oder abgelehnt wird.
> * **Bei Genehmigung:** Der Mitarbeiter wird über die Genehmigung informiert.
> * **Bei Ablehnung:** Der Mitarbeiter wird über die Ablehnung informiert.
> 4. Der Prozess ist in beiden Fällen abgeschlossen.
> ```xml <?xml version=1.0 encoding=UTF-8?><bpmn:definitions
>
> [. . .]
>
> </bpmn:definitions> ```
> In diesem Beispiel gibt es ein
> <bpmn:startEvent name="Urlaubsantrag ausgefüllt"> , ein
>
> ` <bpmn:exclusiveGateway name="Antrag genehmigt"> ` und weitere tasks welche den Ablauf eines Urlaubsantrags zeigen.

Der Algorithmus soll nun zunächst alle Diagramme finden. Dies wird hier durch den Aufruf eines Regex ermöglicht. Das Regex ist in Abbildung 3.4.5 zu sehen.

```
/(?:``?`?\s*(?:xml)?\s*)?(<[^<>]*\/[^<>]*>\s*|<[^\/<>]*>[^`]*<\/[^<>]*>)+\s*(?:``?`?|(?=[^<>`\s]))\n?/g
```

Codeauschnitt 3.4.5: Regex zur Diagrammerkennung

Über dieses werden automatisch alle validen XML Teile (Im folgenden Beispiel bunt markiert) erkannt. Durch die Benutzung einer Non-capturing-group werden Wrapper des XML (Im folgenden Beispiel dunkelblau markiert) wie zum Beispiel das '```xml', automatisch entfernt. Alle gefundenen Übereinstimmungen werden danach auf ihre Länge geprüft, um herauszufinden, ob diese ein vollständiges Diagramm repräsentieren oder nur eine Referenz bzw. Erklärung als Teil des Klartextes sind. (In der folgenden Antwort gelb markiert)

Gerne, hier ist ein Vorschlag für einen einfachen, aber vollständigen Prozess:
Prozess: Urlaubsantrag **Beteiligte:** Mitarbeiter, Vorgesetzter
Ablauf: 1. Ein Mitarbeiter füllt einen Urlaubsantrag aus und reicht ihn ein.
2. Der Vorgesetzte erhält den Antrag und prüft ihn.
3. Der Vorgesetzte entscheidet, ob der Antrag genehmigt oder abgelehnt wird.
* **Bei Genehmigung:** Der Mitarbeiter wird über die Genehmigung informiert.
* **Bei Ablehnung:** Der Mitarbeiter wird über die Ablehnung informiert.
4. Der Prozess ist in beiden Fällen abgeschlossen.
```xml <?xml version=1.0 encoding=UTF-8?><bpmn:definitions
[. . . ]
</bpmn:definitions> ``` In diesem Beispiel gibt es ein
<bpmn:startEvent name="Urlaubsantrag ausgefüllt"> , ein
` <bpmn:exclusiveGateway name="Antrag genehmigt"> ` und weitere tasks welche den Ablauf eines Urlaubsantrags zeigen.

Somit kann die gesamte Nachricht kategorisiert werden. Hellblaue Textstellen sind Diagramme, Nicht markierte Teile und gelbe Textstellen sind Teil der Klartextnachricht.

So wie hier für eine Nachricht mit XML-Diagrammen kann auch eine Nachricht konzeptgleich mit JSON-Diagrammen kategorisiert werden.

## 3.5 Weitere Anbieter

Es ist sinnvoll, dass der Chatbot neben ChatGPT auch andere Chatbot-Anbieter nutzen kann, um Flexibilität, Ausfallsicherheit und Vielfalt in den Antwortmöglichkeiten sicherzustellen. Unterschiedliche Anbieter bieten verschiedene Stärken, wie etwa spezialisierte Natural Language Processing-Modelle, schnellere Antwortzeiten oder kosteneffizientere Lösungen. Durch die Integration mehrerer Anbieter kann je nach Bedarf die beste Leistung ausgewählt werden und Ausfälle eines einzelnen
```

Dienstes werden abgefedert. Dies erhöht die Zuverlässigkeit und Qualität der generierten BPMN-Diagramme.

Durch die objektorientierte Konfiguration der KI Schnittstelle ist es zudem sehr einfach, weitere Anbieter hinzuzufügen.

3.5.1 Grok

Die Einbindung von Grok ergänzt ChatGPT, weil Grok schneller auf aktuelle Daten zugreift, oft direkter formuliert und technische Zusammenhänge sehr präzise erkennt. Dadurch kann der Bot bei bestimmten Aufgaben, etwa beim Interpretieren knapper Anweisungen oder beim Erzeugen alternativer BPMN-Varianten, Ergebnisse liefern, die ChatGPT allein nicht immer erreicht.

Grok kann einfach über die xAI SDK [6] und die Typescript AI SDK [7] zu BPMNGen hinzugefügt werden.

Hierfür wird eine Klasse Grok erstellt, welche die Ai Klasse vererbt bekommt und damit nur noch die Schnittstelle der Grok API implementieren muss. Der entscheidende Schritt ist hierbei, die zu versendende Nachricht an die Grok API vom Prompt-Input Objekt auf das API Format zu bringen. Diese Konvertierung kann man in Codeauschnitt 3.5.1 sehen.

Folgende Modelle sind damit zum aktuellen Stand für BPMNGen verfügbar:

---

[6] `https://ai-sdk.dev/providers/ai-sdk-providers/xai`

[7] `https://www.npmjs.com/package/ai`

Model	Input-Kosten	Output-Kosten
grok-4-1-fast-reasoning	0.20 $	0.50 $
grok-4-fast-reasoning	0.20 $	0.50 $
grok-4-1-fast-non-reasoning	0.20 $	0.50 $
grok-4-fast-non-reasoning	0.20 $	0.50 $
grok-code-fast-1	0.20 $	1.50 $
grok-4	3.00 $	15.00 $
grok-3-mini	0.30 $	0.50 $
grok-3	3.00 $	15.00 $

Tabelle 3.2: Modelle von xAI
(Stand: 20.11.2025)

```
protected mapPromptInput(input: PromptInput) {
  const historyInst = input.history.map((item) => {
    return {role: item.role, content: item.content} ;
  })
  const fileInst = input.file ? {
    role: "user",
    content: [{
      type: "image",
      image: input.getFileDataUrl() ,
    }],
  } : [];
  const userInst = {role: "user", content: input.prompt}
  return {
    model: this.xai(this.model),
    system: input.instructions.join("\n"),
    messages: [historyInst, userInst, fileInst],
  };
}
```

Codeauschnitt 3.5.1: mapPromptInput() für Grok

Die Nachricht wird dann mit Hilfe der AI SDK an Grok gesendet und die Antwort empfangen.

3.5.2 Gemini

Die Einbindung von Gemini ergänzt ChatGPT, weil Gemini bei komplexen Analyseaufgaben, strukturiertem Denken und dem Umgang mit großen Informationsmengen besonders stark ist. Dadurch kann der Bot bei der Modellierung und Optimierung von BPMN-Prozessen zusätzliche Präzision und alternative Lösungswege bieten. Gemini erhöht so die fachliche Tiefe, Robustheit und Variantenvielfalt der Ergebnisse. Gemini hat zum aktuellen Zeitpunkt auch einen entscheidenden Vorteil gegenüber den anderen LLM Anbietern. Die API hat auch eine kostenlose Stufe, wodurch es möglich ist, bis zu einer gewissen Menge an Anfragen, kostenlose Diagramme zu erstellen.

Ähnlich wie bei Grok wird nun die Klasse Gemini erstellt, welche die Schnittstelle zur API implementiert. Dies passiert über die Gemini SDK [8], für welche die Anfrage wieder auf das gewünschte Format gebracht werden muss.

Durch Gemini sind dann diese Modelle alle benutzbar:

Model	Input-Kosten	Output-Kosten
gemini-2.5-pro	0.00 $	0.00 $
gemini-2.5-flash	0.00 $	0.00 $
gemini-2.5-flash-lite	0.00 $	0.00 $
gemini-2.0-flash	0.00 $	0.00 $
Nicht auf der kostelosen Stufe verfügbar:		
gemini-3-pro-preview	2.00 $	12.00 $

Tabelle 3.3: Modelle von Google
(Stand: 20.11.2025)

---

[8] `https://ai.google.dev/gemini-api/docs?hl=de`

```
protected mapPromptInput(input: PromptInput) {
  const systemInst = input.instructions.map((instruction) => {
    return {
      role: "model",
      parts: [{text: instruction, thought: false}]
    };
  });
  const historyInst = input.history.map((item) => {
    return {
      role: item.role == "user" ? "user" : "model",
      parts: [{
        text: item.content,
        thought: item.role == "assistant"
      }],
    };
  });
  const imageInst = input.file ? {
    role: "user", parts: [{
      type: "input_file",
      inlineData: {
        data: input.getFileBase64Data(),
        mimeType: input.getFileMimeType(),
      },
    }]} : [];
  const userInst = {
    role: "user",
    parts: [{text: input.prompt}]
  };
  return {
    model: this.model,
    contents: [systemInst, historyInst, userInst, imageInst],
  };
}
```

Codeauschnitt 3.5.2: mapPromptInput() für Gemini

Die Gemini API erwartet bei jedem Textteil der Anfrage noch das Feld 'thought', welches angibt, ob dieser Teil bereits von einer KI gedacht wurde.

3.5.3 Claude

Claude ergänzt BPMNGen besonders gut, weil er stark auf programmierbezogene Aufgaben spezialisiert ist. Sein Modell ist darauf ausgelegt, Code sehr zuverlässig zu verstehen, zu strukturieren und zu korrigieren. Dadurch liefert Claude bei der Umsetzung von BPMN-Diagrammen in JSON und XML und bei der Fehleranalyse oft besonders saubere Ergebnisse, da diese Formate besonders gut verstanden und umgesetzt werden. Die Einbindung von Claude erhöht somit die Präzision und Qualität von BPMNGen.

Claude wird nun wie schon bei den anderen Anbietern über seine eigene Klasse zu BPMNGen hinzugefügt. Hierbei wird die Schnittstelle, wie in Codeausschnitt 3.5.3, über die Anthropic SDK implemenitert. [9]

Mit Claude sind dann auch noch diese Modelle verfügbar:

Model	Input-Kosten	Output-Kosten
claude-opus-4-5	5.00 $	25.00 $
claude-opus-4-1	15.00 $	75.00 $
claude-sonnet-4-5	3.00 $	15.00 $
claude-haiku-4-5	1.00 $	5.00 $
claude-sonnet-4	3.00 $	15.00 $
claude-opus-4	15.00 $	75.00 $
claude-sonnet-3-7	3.00 $	15.00 $
claude-haiku-3-5	0.80 $	4.00 $
claude-opus-3	15.00 $	75.00 $

Tabelle 3.4: Modelle von Anthropic
(Stand: 20.11.2025)

---

[9] `https://platform.claude.com/docs/en/api/client-sdks`

```
protected mapPromptInput(input: PromptInput) {
  const systemInst = input.instructions.map((instruction) => {
    return {type: "text", text: instruction};
  });
  const historyInst = input.history.map((item) => {
    return {role: item.role, content: item.content};
  });
  const imageInst = input.file ? {
    role: "user",
    content: [{
      type: "file",
      source: {
        type: 'base64',
        data: input.getFileBase64Data(),
        media_type: input.getFileMimeType(),
      },
    }],
  } : [];
  const userInst = {role: "user", content: input.prompt};
  return {
    model: this.model,
    max_tokens: 15000,
    system: systemInst,
    messages: [historyInst, userInst, imageInst],
  };
}
```

Codeauschnitt 3.5.3: mapPromptInput() für Claude

3.6 Streaming

Bisher ergab es nur begrenzt Sinn, die Antworten der KI zu streamen, da ein Diagramm erst dann nutzbar ist, wenn es vollständig erzeugt wurde. Einzelne, unvollständige Fragmente eines BPMN-Diagramms bieten keinen Mehrwert und können vom Client nicht sinnvoll verarbeitet oder angezeigt werden. Mit der Einführung gemischter Antworten, die sowohl Klartext als auch Diagramme enthalten können, ändert sich dies jedoch.

Sobald ein Teil der Antwort aus natürlicher Sprache besteht, entsteht ein klarer Vorteil beim Streaming: Textinhalte können bereits angezeigt werden, während der restliche Output noch generiert wird. Dadurch erhält der Nutzer deutlich schneller Rückmeldung, was insbesondere bei komplexeren oder längeren Antworten zu einem spürbar verbesserten Nutzungserlebnis führt.

Technisch bedeutet dies, dass die Ausgabe der KI zunächst an den BPMNGen-Server gestreamt werden muss, der die eingehenden Daten analysiert und korrekt kategorisiert. Anschließend wird der Klartext Teil der Antwort weiter an den Client gestreamt. Entscheidend ist dabei, dass ausschließlich der textuelle Anteil der Antwort übertragen wird. Diagrammfragmente oder Codeblöcke würden beim Client zu Fehlern führen, da diese erst nach vollständiger Generierung sinnvoll weiterverarbeitet oder angezeigt werden können.

Durch diese Trennung entsteht ein effizientes zweistufiges Streaming-Verfahren: Der BPMNGen-Server verarbeitet die gesamte Antwort, extrahiert den Klartext in Echtzeit und leitet ihn unmittelbar weiter, während das Diagramm erst nach seiner vollständigen Fertigstellung bereitgestellt wird.

Die gestreamte Ausgabe eines LLM-Anbieters zu starten, ist in den meisten Fällen unkompliziert. Viele Modelle bieten dafür einen expliziten Parameter an, der direkt in der Anfrage gesetzt werden kann. Sowohl ChatGPT als auch Claude unterstützen dieses Vorgehen nativ, sodass sich das gewünschte Verhalten bereits beim Abschicken des Requests konfigurieren lässt. Ein entsprechendes Beispiel sieht wie in Codeausschnitt 3.6.1 aus.

Andere Anbieter wie Gemini oder Grok verfolgen dagegen einen leicht unterschiedlichen Ansatz und stellen das Streaming über eine separate API-Funktion bereit. Je nach verwendetem Endpunkt wird entweder ein normaler Text generiert oder ein Stream zurückgegeben. Der Wechsel zwischen beiden Varianten lässt sich dadurch einfach implementieren, siehe 3.6.2.

Sobald die Antwort der KI eintrifft, wird zunächst überprüft, ob es sich tatsächlich um einen Stream handelt. Da JavaScript-basierte Streams typischerweise das

```
protected mapPromptInput(input: PromptInput, stream: boolean) {
  [...]
  return {
    model: this.model,
    stream: stream,
    system: systemInst,
    input: [historyInst, userInst, imageInst]
  };
}
```

Codeauschnitt 3.6.1: mapPromptInput() für einen Stream

```
async generateContent(input: any, stream: boolean) {
  if (stream) return streamText(input)
  return generateText(input);
}
```

Codeauschnitt 3.6.2: generateContent() für einen Stream

Symbol `asyncIterator` implementieren, lässt sich dies zuverlässig über eine einfache Typprüfung feststellen:

```
protected isStream(obj: any): boolean {
  return obj && typeof obj[Symbol.asyncIterator] == "function";
}
```

Codeauschnitt 3.6.3: isStream()

Wird ein Stream erkannt, kann dieser anschließend mithilfe eines asynchronen Iterations-Loops ausgelesen werden. Die empfangenen Delta-Fragmente lassen sich so in Echtzeit weiterverarbeiten, um Klartext sofort an den Client zu streamen, bevor das vollständige Diagramm erzeugt wird. Wie diese Deltas verarbeitet werden, ist in Codeausschnitt 3.6.4 dargestellt.

Während der Stream verarbeitet wird, wird die Antwort der KI schrittweise in einem internen Buffer aufgebaut. Nach jeder Erweiterung des Buffers wird der Klartextanteil, wie in Abschnitt 3.4.3 beschrieben, extrahiert und an den Client weitergeleitet. Auf diese Weise erhält der Nutzer bereits während der Generierung der vollständi-

```
protected async processStream(stream: any){
  for await (const chunk of stream) {
    switch (chunk.type) {
      case "response.output_text.delta":
        // Text oder andere Änderungen
        processDelta(chunk.delta);
        break;

      case "response.completed":
        // Fertig gelesen
        return;

      case "response.error":
      case "response.failed":
        // Fehler
        error(chunk.response.error.message);
        return;

      default:
        break;
    }
  }
}
```

Codeauschnitt 3.6.4: processStream()

gen Antwort fortlaufend Informationen, ohne auf das Endergebnis warten zu müssen.

Erkennt das System jedoch, dass der Stream aktuell ein Diagramm enthält, wird dieser zunächst zurückgehalten und nicht an den Client übertragen. Diagrammfragmente sind während der laufenden Generierung weder syntaktisch vollständig noch anzeigbar. Es macht daher keinen Sinn diese zu streamen.

Sobald das Diagramm allerdings vollständig empfangen wurde, kann es analysiert und eindeutig einer Kategorie zugeordnet werden. Dabei wird, wie in Abschnitt 3.4.3 beschrieben, entschieden, ob es sich um ein finales BPMN-Diagramm handelt, das angezeigt werden soll, oder ob der betreffende Abschnitt lediglich Bestandteil eines beschreibenden Klartextes ist und somit kein eigenständiges Diagramm ist. Dieses Vorgehen stellt sicher, dass sowohl Text- als auch Diagrammausgaben sauber voneinander getrennt und jeweils korrekt verarbeitet werden, ohne dass unvollständige oder fehlerhafte Diagrammfragmente beim Client ankommen.

3.6.1 SSE

Für die Übertragung der erzeugten Informationen an den Client wird die Technik der Server-Sent Events (SSE) eingesetzt. SSE ermöglicht es dem Server, Daten in Echtzeit an den Client zu senden, ohne dass dieser wiederholt aktiv Anfragen stellen muss. Jede gesendete Nachricht folgt dabei einem strukturierten Aufbau und besteht aus zwei wesentlichen Komponenten: einem `event`-Feld und einem `data`-Feld.

Der `event`-Teil enthält in der Regel ein einzelnes Wort, das den Typ oder die Bedeutung der übertragenen Daten beschreibt, zum Beispiel `delta`, `diagram`, `error` oder `end`. Dadurch kann der Client umgehend erkennen, wie der empfangene Inhalt weiterzuverarbeiten ist.

Der eigentliche Inhalt befindet sich im `data`-Teil. Hier werden die Daten hinterlegt, beispielsweise ein Textdelta oder ein fertig generiertes Diagramm.

Zusammengefügt und korrekt formatiert wird die gesamte SSE-Nachricht schließlich in folgender Form an den Client übermittelt:

event: *event-name*
data: *Erste Zeile der Daten*
data: *Weitere Zeile mit Daten*

event: *Nächstes Event*
...

Im Folgenden wird nun gezeigt, welche Events für die Übertragung einer Antwort an den Client implementiert wurden.

Der Stream beginnt mit einem `start` Event. In diesem wird dem Client die Thread-ID mitgeteilt und indirekt erkenntlich gemacht, dass nun ein Stream gestartet wird.

Da manche KI Anbieter Modelle entwickelt haben, welche zunächst Websuchen durchführen oder interne Prozesse durchführen, kann es sein, dass zwischen dem Start des Streams und dem ersten Delta einiges an Zeit vergeht. Um zu verhindern, dass der Browser des Clients deshalb durch einen Timeout die Verbindung schließt, wird das `alive` Event implementiert. Dieses wird jede Sekunde gesendet und beinhaltet als data lediglich die aktuelle Uhrzeit.

Das erste eigentliche Daten Event ist nun das `delta` Event bei dem die tatsächlichen Deltas des Klartextteils versendet werden. Die Deltas haben keine fixe Größe und können je nach LLM Anbieter und Antwort variieren.

Jegliche Fehler werden dem Client als `error` Event mitgeteilt, wobei die Error Nachricht als Data mitgesendet wird. Nach einem `error` Event wird die Verbindung automatisch vom Server beendet.

Wenn der Server erkennt, dass gerade ein vollständiges Diagramm generiert wird, sendet er ein `diagram-start` Event welches dem Client mitteilt, welches Modell verwendet wird. Sobald das Diagramm fertig generiert wurde, wird auch ein `diagram-end` Event gesendet, welches dem Client mitteilt, dass das Diagramm fertig erzeugt wurde, sowie ein `diagram` Event, welches das fertige und formatierte Diagramm enthält.

Sobald der Stream des LLMs fertig ist, wird noch ein `end` Event versendet, welches die gesamte Antwort als JSON versendet. Dieses Event sieht genau so aus, wie die Antwort gewesen wäre, wenn nicht an den Client gestreamt worden wäre. Dadurch ist es möglich, Fehler bei der Übersendung des Streams ausbessern zu können.

Final wird noch ein `save` Event versendet, welches dem Client mitteilt, dass die generierte Antwort nun auch erfolgreich in der Datenbank abgespeichert wurde und nun für weitere Anfragen bereit steht. Danach wird der Stream geschlossen und die Übertragung ist abgeschlossen.

Ein vollständiger Stream mit SSE sieht dann beispielsweise so aus:

```
event: start
data: faf8ad85-5546-4da2-98d9-8784844f1ea9

event: delta
data: Ich bin

event: delta
data: ChatGPT 4.1 mini

event: diagram-start
data: gpt-4.1-mini

event: alive
data: 01.01.2025 00:02

event: diagram-end
data: gpt-4.1-mini

event: diagram
data: <?xml version=\“1.0\” encoding=\“UTF-8\”?>
data: <bpmn:definitions ...
data: </bpmn:definitions>

event: end
data: {
data: “text”: “Ich bin ChatGPT 4.1 mini”
data: “xml”: “<?xml?><bpmn:definitions>...</bpmn:definitions>”
data: }

event: save
data: success
```

3.7 Schema-Constraining

Schema Constraining bezeichnet die Technik, bei der das Ausgabeformat eines KI-Modells durch ein vorgegebenes Schema eingeschränkt wird. Statt den Text frei formulieren zu können, muss das Modell seine Antwort exakt in der festgelegten Struktur ausgeben. Dies kann beispielsweise ein JSON-Schema, ein XML-Schema oder eine andere Form haben. Schema Constraing wurde bereits auch bei vergleichbaren BPMN Bots eingesetzt und hat erfolgreich alle syntaktischen Fehler eliminieren können. [14]

Der große Vorteil besteht darin, dass die Antworten vorhersehbar sind. Fehler wie fehlende Felder, falsche Datentypen oder ungültige Strukturen werden verhindert, da das Modell gezwungen ist, jede Ausgabe formal korrekt zu gestalten. Dies ist besonders wichtig in Anwendungen, bei denen die KI-Ausgabe weiterverarbeitet wird, wie etwa bei der Erstellung von BPMN-Diagrammen.

Dadurch ist sichergestellt, dass alle erzeugten Diagramme syntaktisch korrekt sind. Allerdings kann das Schema-Constraining nicht auf den `detail` Modus angewendet werden, da dieser nach dem Design auch frei mit Klartext, Beschreibungen, Beispielen, Fragen und allem, was gewünscht wird, antworten können soll. Daher wird das Schema Constraining nur bei dem `quick` Modus verwendet.

Die Nutzung des Schema Constraining wird von allen Anbietern bereitgestellt, allerdings hat auch jeder Anbieter seine eigene Umsetzung dieser Constraints. Alle Anbieter erlauben aber die Nutzung des Schema-Validators `zod`. [10] Hierbei wird das Schema über Objects, Arrays und Enums abgebildet, wobei diese jeweils primitive Attribute besitzen wie z. B. numbers, strings, booleans, uuids, chars...

Mit zod lässt sich sehr gut das definierte JSON Schema darstellen. Wie diese Schema Constraints für JSON aussehen, wird in Codeausschnitt 3.7.1 gezeigt. Eine Unterstützung für XML Schemas ist leider mit Zod nur begrenzt möglich.

---

[10] `https://zod.dev/`

3.8 Diagramm-Sampling

Während die Bezeichnung `Sampling` in Bezug auf LLMs bereits eine Bedeutung hat, wird hier im Bezug auf BPMN Generierung nicht von Token-Sampling, sondern von Diagramm-Sampling gesprochen.

Der Begriff „Sampling“ bedeutet im Kontext von KI-Modellen allgemein: Aus einer Menge möglicher Modellantworten mehrere Alternativen erzeugen.

Dies bedeutet konkret, dass beim Token-Sampling mehrere Token erstellt werden und daraus das beste gewählt wird. Beim Sampling für Diagramme werden nun auch mehrere Diagramme erstellt. Allerdings ist es schwierig zu beurteilen, welches Diagramm nun das ‘beste’ ist. Darum wird bei der Erstellung der Diagramme klar festgelegt, welche Anbieter ein Diagramm erzeugen und alle werden dem Nutzer präsentiert, da dieser selber am besten die Qualität beurteilen kann.

Die Erstellung mehrerer Diagramme erfolgt, indem dieselbe textuelle Prozessbeschreibung parallel an verschiedene Sprachmodelle bzw. KI-Anbieter gesendet wird. Jedes Modell generiert daraufhin ein vollständiges BPMN-Diagramm, basierend auf seiner internen Architektur und seinen Trainingsdaten. Durch den gleichen Prompt wird sichergestellt, dass alle Modelle unter vergleichbaren Bedingungen arbeiten und so miteinander vergleichbare Ergebnisse liefern. Dieser parallele Erstellungsprozess ermöglicht es, innerhalb eines einzigen Ausführungsvorgangs mehrere unabhängige Modellierungen desselben Prozesses zu erhalten, ohne zusätzliche Laufzeit oder iterative Interaktion mit einem einzelnen Modell zu benötigen.

Da keine zusätzlichen textuellen Ausgaben benötigt werden, erfolgt die Generierung der zusätzlichen Diagramme ausschließlich im `quick` Modus. Die Auswahl der Modelle, die für die parallele Diagrammerzeugung genutzt werden, ist flexibel und kann vom Nutzer oder vom Client als Parameter der Anfrage frei bestimmt werden. Dadurch lässt sich die Zahl der beteiligten Anbieter dynamisch variieren, was insbesondere für Vergleiche oder Qualitätssicherung von Vorteil ist.

Alle erzeugten Diagramme, unabhängig davon, welches Modell sie generiert hat, werden anschließend gesammelt und gebündelt an den Client übermittelt. Dies ermöglicht es, die Ergebnisse unmittelbar nebeneinander anzuzeigen, wodurch der Nutzer einen direkten Vergleich der unterschiedlichen Modelle erhält.

Eine Anfrage mit Diagramm Sampling kann in etwa so aussehen wie in Anfrage 3.8.1.

Der Code in Codeausschnitt 3.8.2 führt die parallele Generierung aller BPMN-Diagramme aus. Zunächst wird anhand des ausgewählten Modells die primäre Generierung vorbereitet. Die Sampling-Modelle, hier die sekundären genannt, werden zunächst gefiltert, um ungültige Einträge sowie Duplikate zu entfernen, und anschließend auf maximal fünf zusätzliche Modelle begrenzt. Für jedes dieser Modelle wird ebenfalls die Diagrammgenerierung vorbereitet, jedoch im ressourcenschonenden `quick`-Modus.

Alle Diagramm-Generierungen, das primäre sowie die sekundären, werden schließlich über `Promise.all([])` parallel ausgeführt. Dadurch entstehen mehrere unabhängige Modellantworten in einem einzigen Ausführungsschritt, die anschließend gemeinsam an den Client geschickt werden.

3.9 Reflective Prompting

Reflective Prompting ist eine Technik, bei der ein KI-Modell bewusst dazu angeleitet wird, über seine eigenen Antworten nachzudenken, bevor es ein endgültiges Ergebnis liefert. Anders als beim klassischen Prompting, bei dem das Modell direkt versucht, die bestmögliche Antwort zu erzeugen, besteht Reflective Prompting aus zwei Stufen: Zuerst erstellt das Modell einen Entwurf oder eine Einschätzung, anschließend überprüft es diesen Entwurf selbstständig, reflektiert mögliche Fehler oder Unklarheiten und verbessert die Antwort basierend auf dieser Selbstkritik.

Dieses Vorgehen hat mehrere Vorteile: Das Modell erkennt häufiger eigene Ungenauigkeiten, identifiziert logische Fehler oder fehlende Details und kann dadurch qualitativ hochwertigere Ergebnisse liefern. Reflective Prompting eignet sich besonders für Aufgaben, die komplexes Denken, Fehlererkennung oder mehrstufiges Argumentieren erfordern, etwa bei der Analyse und Erstellung von Diagrammen.

Da das Modell aktiv „darüber nachdenkt“, wie gut seine Antwort ist, nähert es sich stärker menschlichem Problemlösungsverhalten an. Gleichzeitig kann diese Technik aber auch zu längeren Antwortzeiten oder höheren Tokenkosten führen, da das Modell intern mehrere Schritte durchläuft. In vielen Fällen lohnt sich Reflective

Prompting jedoch, weil die resultierenden Antworten deutlich präziser und verlässlicher sind.

In dem Anwendungsfall des BPMNGen Bots wird das `Reflective Prompting` nicht als ein Schritt der Generierung implementiert, bei dem das Diagramm bereits überarbeitet wird, bevor es der Nutzer zu sehen bekommt. Da eine Generierung des Diagramms viel Zeit beansprucht, würde dies die Generierungszeit verdoppeln. Stattdessen wird die erste Diagrammerstellung dem Nutzer normal angezeigt. Wenn der Nutzer nun Änderungswünsche hat, werden diese parallel mit den Diagrammfehlern zum Überarbeiten an die KI gesendet. Damit können nun gleichzeitig Änderungswünsche des Nutzers umgesetzt werden sowie Probleme des Diagramms intern behoben werden. Die Implementierung wird in Codeauschnitt 3.9.1 auf Seite 51 gezeigt.

Ein solches Verfahren, bei dem die Fehler eines Diagrammes deterministisch bestimmt werden und dann in einem zweiten Schritt an die KI gesendet werden, wird auch von den meisten alternativen LLM-basierten BPMN-Anwendungen genutzt.[10, 11, 14] Dies ist ein optimaler Mittelweg zwischen schneller Generierung und guter Qualität. Die Implementierung des Reflective Prompting in BPMNGen ermöglicht es nun, dass Fehler automatisch ohne ein aktives Fordern des Nutzers korrigiert werden, parallel zu jeglichen anderen Wünschen, welche der Nutzer an den Chatbot stellt. Es wird zunächst nur auf syntaktische Fehler untersucht, der Code ermöglicht aber ein einfaches Hinzufügen weiterer Validierungsalgorithmen. Es ist außerdem ein reines Beheben der Fehler möglich, indem eine Update Anfrage ohne Nutzertext an den BPMNGen Chatbot gesendet wird, falls der Nutzer nur Fehler beheben möchte. Bei BPMNGen wird dieses Verfahren besonders effizient angewendet, da der Nutzer nicht explizit Fehler korrigieren lassen muss und alle erkannten Fehler auch nicht dauerhaft im Chat gespeichert werden, sondern nur für das aktuellste Diagramm an die KI gesendet werden.

```
const CoordinateSchema = z.array(
  z.number().int().nonnegative()
).length(2);
const ComponentTypeEnum = z.enum([
  'startEvent','messageStartEvent','timerStartEvent',
  'intermediateCatchEvent','intermediateThrowEvent',
  'messageCatchEvent','messageThrowEvent','timerIntermediateEvent',
  'endEvent','messageEndEvent','task','subProcess',
  'exclusiveGateway','parallelGateway','inclusiveGateway'
]);
const FlowTypeEnum = z.enum([
  'sequenceFlow','messageFlow','association',
  'dataInputAssociation','dataOutputAssociation',
]);
const FlowSchema = z.object({
  ID: z.string().min(1),
  Start: z.string(),
  Target: z.string(),
  Type: FlowTypeEnum,
  StartXY: CoordinateSchema,
  TargetXY: CoordinateSchema,
  Descriptor: z.string(),
});
const ComponentSchema: z.ZodType<any> = z.lazy(() => z.object({
    ID: z.string().min(1),
    Name: z.string(),
    Type: ComponentTypeEnum,
    x: z.number().int().nonnegative(),
    y: z.number().int().nonnegative(),
    Incoming: z.array(z.string()),
    Outgoing: z.array(z.string()),
    Components: z.array(ComponentSchema).optional().nullable(),
    Flows: z.array(FlowSchema).optional().nullable(),
}));
const LaneSchema = z.object({
  ID: z.string().min(1),
  Name: z.string().min(1),
  XY: CoordinateSchema,
  width: z.number().int().positive(),
  height: z.number().int().positive(),
  Components: z.array(ComponentSchema),
  Flows: z.array(FlowSchema),
});
```

Codeauschnitt 3.7.1: Schema Constraining with ZOD

```
// POST /threads
{
  "inputString": "Bitte generiere mir ein BPMN Diagram,
      welches den Ablauf in einem Restaurant zeigt",
  "model": "gpt-5 (xml)",
  "mode": "detail",
  "samples": [
    "gemini-2.5-pro (xml)",
    "grok-4 (json)"
  ]
}
```

Codeauschnitt 3.8.1: Post Request an /threads mit Sampling

```
const gpt = getGPT(model, format)!;
const sampling = !!samples;
const primary = gpt.createBPMN([...], mode, "primary");
const secondaries = samples
  .map(str => getGPT(str))
  .filter(gpt => !!gpt)                     // remove invalid
  .filter((v, i, s) => s.indexOf(v) == i) // remove duplicates
  .slice(0, 5)                              // limit to 5 samples
  .map(gpt => gpt.createBPMN([...], "quick", "secondary"));
const outputs = await Promise.all([primary, ...secondaries]);
```

Codeauschnitt 3.8.2: Ausführung der Samples

```
protectes updateInstructions(threadID: string, format: format){
  const diagram = await this.getLatestDiagramFromDB(threadID);
  if (!diagram || !diagram.xmlContent)
    return [];
  const xmlModdle = await moddle.fromXML(diagram.xmlContent);
  const warnings = xmlModdle.warnings;
  return [`The The following diagram has already been created:
    ${diagram.xmlContent}\n
    The following warnings were found in the diagram:
    ${warnings.join("\n")}\n
    Fix the warnings while updating the diagram.
    Update the diagram, if asked for, for the given prompt.`]
}
```

Codeauschnitt 3.9.1: updateInstructions() mit Reflektion

4 Performanzanalyse

4.1 Qualität

Im Folgenden wird die Qualität der erstellten Diagramme untersucht.

Zunächst soll analysiert werden, welche KI-Modelle besonders hochwertige Diagramme erzeugen können. Aspekte wie Zeitaufwand und Kosten werden in diesem Abschnitt bewusst nicht berücksichtigt, da der Fokus ausschließlich auf der Ergebnisqualität liegt.

Für die Untersuchung werden mit verschiedenen KI-Modellen Diagramme generiert. Dabei kommt eine einheitliche Prozessbeschreibung (siehe Prompt 4.1.1) zum Einsatz. Diese wurde bewusst nicht einfach oder kurz gehalten, sondern möglichst komplex formuliert, um zu evaluieren, welche Modelle in der Lage sind, anspruchsvolle und detaillierte Beschreibungen korrekt zu verarbeiten und umzusetzen.

In einem weiteren Schritt wird auch ein vereinfachter Prompt verwendet. Ziel ist es zu überprüfen, ob einzelne Modelle selbst bei weniger komplexen Prozessbeschreibungen Schwierigkeiten aufweisen und somit grundlegende Anforderungen nicht zuverlässig erfüllen können.

4.1.1 Modellunterschiede komplexer Diagramme

Welches KI Modell kann besonders gut aus komplexen Prozessbeschreinungen BPMN Diagramme erstellen?

Hierfür werden einige der aktuellsten Modelle der implentierten Anbieter getestet und die Ergebnisse verglichen. Gleichzeigt wird zudem auch zwischen den verwendeten Formaten XML und JSON, welche auf Seite 16 beschrieben sind, unterschieden. Die Modelle, welche getestet werden sind:

- **Gemini 2.5 Pro** erzeugt Diagramm B.4 in JSON und Diagramm B.5 in XML
- **Gemini 2.5 Flash** erzeugt Diagramm B.6 in JSON und Diagramm B.7 in XML
- **ChatGPT 5.2** erzeugt Diagramm B.8 in JSON und Diagramm B.9 in XML
- **ChatGPT 5.1** erzeugt Diagramm B.10 in JSON und Diagramm B.11 in XML
- **ChatGPT 4.1** erzeugt Diagramm B.12 in JSON und Diagramm B.13 in XML
- **Grok 4** erzeugt Diagramm B.14 in JSON und Diagramm B.15 in XML
- **Grok 4.1 Fast** erzeugt Diagramm B.16 in JSON und Diagramm B.17 in XML
- **Claude Opus 4.5** erzeugt Diagramm B.18 in JSON
- **Claude Sonnet 4.5** erzeugt Diagramm B.19 in JSON

Es wird für alle die einheitliche Prozessbeschreibung 4.1.1 verwendet. Um auch zu testen, ob die Modelle in der Lage sind die gewünschte Ausgabesprache zu erkennen und zu verwenden, wird diese Prozessbeschreibung in deutsch geschrieben.

> Der Kunde sendet online seine Bestellung an die E-Commerce-Plattform. Dort wird parallel in der Finanzbuchhaltung die Zahlungsautorisierung angefragt, wobei eine Kreditprüfung (automatisch, manuell nur über 200€) erfolgt. Die Finanzbuchhaltung meldet dann „Zahlung OK“ oder „abgelehnt“ zurück, wobei nach einer Stunde ohne Antwort eine Erinnerung folgt. Gleichzeitig verzweigt der Prozess: Es wird für jeden Artikel der Bestand beim Lager & Logistik angefragt, und wenn ein Artikel eine Sonderanfertigung ist, geht zusätzlich eine Anfrage an die Fertigung. Im Lager wird bei Verfügbarkeit reserviert, bei Nichtverfügbarkeit der Kunde informiert und eine Nachbestellung ausgelöst. Die Fertigung beginnt den Subprozess der Sonderanfertigung und sendet nach Abschluss eine Fertigstellungsnachricht an die Plattform und eine Abholbereitmeldung ans Lager. Die E-Commerce Plattform wartet auf alle Rückmeldungen. Bei Zahlungsablehnung wird alles storniert und der Kunde benachrichtigt. Bei Erfüllung geht der Kommissionierungsauftrag ans Lager (mit Eskalation an den Manager nach 48 Stunden). Das Lager kommissioniert, verpackt und sendet den Lieferschein an die Finanzbuchhaltung sowie eine Abholanforderung an den Versanddienstleister.

Prompt 4.1.1: Komplexe Prozessbeschreibung für einen Qualitätstest

Vorab ist es nun wichtig zu erwähnen, dass mit diesem Prompt kein Diagramm von `Claude Opus 4.5` mit XML erstellt werden konnte, da dies die Maximaltokenanzahl dieses Modells übersteigt.

Formale Richtigkeit Damit wird geprüft, ob das Diagramm regelkonform nach BPMN 2.0 ist. Hierbei ist entscheidend, wie viele formatbedingte und conventionelle Fehler in dem Diagramm erzeugt werden. Dazu zählen unter anderem:

- Jeder Flow beginnt mit einem Start Event und endet mit einem End Event.
- Gateways haben korrekte Ein- und Ausgänge (z. B. XOR, AND, Event-Based).
- Kein Element hat ungültige Sequenzen (z. B. Aktivitäten direkt nach Nachrichtenflüssen).
- Es gibt keine ID Duplikate.
- Keine grundlegenden Formatfehler.
- Alle Referenzen sind richtig.
- ...

Bei dem Test wurden alle Fehler gesammelt und zusammengefasst. Die Fehler wurden mit Hilfe von bpmn-moddle [1] gesammelt. Hierbei ist ein Fehler, durch den das gesamte Diagramm nicht angezeigt werden kann, ein `kritischer Fehler`. Tritt ein kritischer Fehler auf, wird die Generierung wiederholt, bis keine kritischen Fehler erstellt werden. Ein Fehler, durch den ein einzelnes Element, nicht angezeigt werden kann, ist ein `elementarer Fehler`. Bei diesem Test wurde auf das vorgestellte Schema Constraining auf Seite 45 verzichtet um den Modellen möglichst viel Freiraum zu geben und dadurch die Richtigkeit der Diagramme besser beurteilen zu können. Das jeweils, ohne kritische Fehler, erstellte Diagramm der Modelle wird in den Abbildungen: B.4, B.5, B.6, B.7, B.10, B.11, B.8, B.9, B.12, B.13, B.14, B.15, B.16, B.17, B.19 B.18 dargestellt. Die Anzahl an Fehlern ist in Tabelle 4.1 zu sehen.

Die Auswertung der formalen Richtigkeit zeigt, dass sich deutliche Leistungsunterschiede erkennen lassen.

Claude Opus 4.5, Gemini 2.5 Pro und ChatGPT 5.1 stechen insbesondere im JSON Format hervor, da hier keine formalen Fehler festgestellt wurden. Die erzeugten Diagramme enthalten keine formalen Fehler und halten die grundlegenden BPMN-2.0-Konventionen ein. Damit zeigen diese Modelle eine hohe Eignung für die Modellierung komplexer Prozesse, sofern JSON als Zielformat verwendet wird.

---

[1] `https://github.com/bpmn-io/bpmn-moddle`

Modell		**kritische Fehler**	**elementare Fehler**
Gemini 2.5 Pro	JSON	0	0
	XML	1	0
Gemini 2.5 Flash	JSON	0	33
	XML	0	24
ChatGPT 5.1	JSON	0	0
	XML	0	2
ChatGPT 5.2	JSON	0	4
	XML	2	0
ChatGPT 4.1	JSON	0	13
	XML	0	5
Grok 4	JSON	0	16
	XML	0	0
Grok 4.1 Fast	JSON	0	20
	XML	0	3
Claude Opus 4.5	JSON	0	0
Claude Sonnet 4.5	JSON	0	26

Tabelle 4.1: Formale Richtigkeit

Im XML-Format offenbaren sich auch stärkere Unterschiede. Während einige Modelle, darunter Gemini 2.5 Pro und ChatGPT 5.2, kritische Fehler erzeugen, die eine erneute Generierung erforderlich machen, liefert Grok 4 in diesem Format formal fehlerfreie Ergebnisse. Dies deutet darauf hin, dass Grok 4 besonders gut mit der strengen Struktur und den Referenzabhängigkeiten von BPMN-XML umgehen kann und sich daher besonders für Diamgrammerstellungen ohne Zwischenformat eignet. Interessant ist auch, dass, obwohl Gemini 2.5 Pro und ChatGPT 5.2 kritische Fehler erstellen, sie bei einer nicht-kritischen Erstellung eines Diagramms auch keine elementaren Fehler erstellen.

Semantische Richtigkeit Hier geht es darum, ob das Modell inhaltlich korrekt ist.

- Bildet das Diagramm den beschriebenen Prozess korrekt ab?
- Sind alle nötigen Komponenten Vorhanden?
- Wurden unnötige Komponenten hinzugefügt?

- Stimmen Reihenfolgen und Abhängigkeiten überein?

In der folgenden Tabelle 4.2 wurde untersucht, wie viele Komponenten in den einzelnen Diagrammen jeweils fehlen bzw. ungewünscht hinzugefügt wurden.

Modell		**fehlend**	**unnötig**
Gemini 2.5 Pro	JSON	0	4
	XML	1	0
Gemini 2.5 Flash	JSON	10	6
	XML	0	4
ChatGPT 5.1	JSON	9	2
	XML	7	8
ChatGPT 5.2	JSON	1	8
	XML	1	2
ChatGPT 4.1	JSON	15	2
	XML	40	1
Grok 4	JSON	8	0
	XML	3	0
Grok 4.1 Fast	JSON	5	3
	XML	2	1
Claude Opus 4.5	JSON	0	4
Claude Sonnet 4.5	JSON	11	1

Tabelle 4.2: Semantische Richtigkeit

Die Ergebnisse in Tabelle 4.2 zeigen, dass die Modelle deutliche Unterschiede hinsichtlich der Vollständigkeit der erzeugten Diagramme aufweisen.

Besonders positiv fällt erneut Claude Opus 4.5 auf, welcher im JSON-Format keine fehlenden Komponenten erzeugt. Zwar wurden einige zusätzliche, nicht explizit geforderte Elemente ergänzt, diese beeinflussen jedoch die grundlegende Prozesslogik nicht wesentlich. Insgesamt deutet dieses Ergebnis auf ein sehr gutes semantisches Prozessverständnis hin.

Gemini 2.5 Pro zeigt ebenfalls eine hohe semantische Qualität. Im JSON-Format fehlen keine Prozessbestandteile, allerdings wurden mehrere unnötige Komponenten ergänzt. Im XML-Format fehlt nur ein einziges Element, dafür wurde kein einziges unnötiges hinzugefügt. Dies legt nahe, dass das Modell den Prozess grundsätzlich korrekt interpretiert.

Die Modelle der ChatGPT-Reihe liefern insgesamt gemischte Ergebnisse. Während ChatGPT 5.2 im Vergleich zu älteren Versionen deutlich weniger fehlende Komponenten aufweist, insbesondere im XML-Format, tendiert es dazu, zusätzliche Tasks zu ergänzen, die nicht explizit in der Prozessbeschreibung stehen. ChatGPT 5.1 und insbesondere ChatGPT 4.1 zeigen hingegen eine deutlich höhere Anzahl fehlender Komponenten.

Grok 4 und Grok 4.1 Fast positionieren sich im Mittelfeld. Beide Modelle erfassen den Kernprozess weitgehend korrekt und verzichten nahezu vollständig auf unnötige Ergänzungen, insbesondere im XML-Format.

Abgesehen von den ausgearbeiteten Komponentenfehler, lassen sich noch einige weitere Eigenheiten erkennen. Bei Claude Sonnet 4.5 und Gemini 2.5 Flash JSON wurden alle Message Flows fehlerhaft erzeugt, wodurch sie nicht vorhanden sind. ChatGPT 4.1 XML hat ein ähnliches Problem, hier sind allerdings nicht die Message Flows, sondern fast alle Sequence Flows betroffen. Es gab ausserdem auch Probleme die Elemente in die jeweiligen Lanes zu platzieren bei ChatGPT 5.2 mit XML sowie bei ChatGPT 5.1 mit JSON.

Was auch auffällt, ist, dass bei XML die Komponeten weitaus unübersichtlicher angeordnet werden. Dies kann auch daran liegen, dass die Positionierung der Komponenten im JSON Format um einiges einfacher definiert ist. JSON liefert in der Regel ein weitaus ordnetlicheres Ergebnis.

Generell fällt auf, dass auch bei ‘ordentlichen’ Diagrammen immernoch Raum zur Verbesserung besteht. Es werden leider sehr häufig Flows genau auf Kanten von Lanes, Pools oder anderen Flows gelegt, wes es sehr schwer macht diese nachzuverfolgen. Auch mit expliziten Anweisungen dies nicht zu machen, kann dieses Problem nicht vollständig mit Prompting behoben werden. Es würde sich anbieten einen Algorithmus zu implementieren, welcher die Positionierung aller Komponenten nachverbessert. Dies übersteigt allerdings den Rahmen dieser Arbeit.

Zusammenfassend lässt sich festhalten, dass Claude Opus 4.5 und Gemini 2.5 Pro das beste inhaltliche Verständnis der komplexen Prozessbeschreibung zeigen. Modelle wie ChatGPT 4.1 oder Gemini 2.5 Flash mit hoher Anzahl falscher, fehlender oder unnötiger Komponenten sind hingegen für die automatisierte Erstellung semantisch korrekter BPMN-Diagramme aus komplexen Textbeschreibungen nur eingeschränkt geeignet.

4.1.2 Modellunterschiede mittelschwerer Diagramme

Nachdem nun getestet wurde welche Modelle sich besonders gut für komplexe Prozessbeschreibungen eignen, soll nun noch die Frage beantwortet werden.

Welche Modelle kommen selbst mit einer mittelschweren Prozessbeschreibung nicht zurecht?

Um dies herauszufinden wird die einheitliche Prozessbeschreibung 4.1.2, Dokument 1.2 des PET Datasets [1], für alle Modelle verwendet.

> A customer brings in a defective computer and the CRS checks the defect and hands out a repair cost calculation back. If the customer decides that the costs are acceptable, the process continues, otherwise she takes her computer home unrepaired. The ongoing repair consists of two activities, which are executed, in an arbitrary order. The first activity is to check and repair the hardware, whereas the second activity checks and configures the software. After each of these activities, the proper system functionality is tested. If an error is detected another arbitrary repair activity is executed, otherwise the repair is finished.

Prompt 4.1.2: Dokument 1.2 des PET Datasets [1]

Es wird auch wieder zwischen den verwendeten Formaten XML und JSON, welche auf Seite 16 beschrieben sind, unterschieden. Die Modelle, welche getestet werden sind:

- **Gemini 2.5 Pro** erzeugt Diagramm B.20 in JSON und Diagramm B.21 in XML
- **Gemini 2.5 Flash** erzeugt Diagramm B.22 in JSON und Diagramm B.23 in XML
- **ChatGPT 5.2** erzeugt Diagramm B.24 in JSON und Diagramm B.25 in XML
- **ChatGPT 5.1** erzeugt Diagramm B.26 in JSON und Diagramm B.27 in XML
- **ChatGPT 4.1** erzeugt Diagramm B.28 in JSON und Diagramm B.29 in XML
- **Grok 4** erzeugt Diagramm B.30 in JSON und Diagramm B.31 in XML
- **Grok 4.1 Fast** erzeugt Diagramm B.32 in JSON und Diagramm B.33 in XML

- **Claude Opus 4.5** erzeugt Diagramm B.34 in JSON und Diagramm B.35 in XML
- **Claude Sonnet 4.5** erzeugt Diagramm B.36 in JSON und Diagramm B.37 in XML

Formale Richtigkeit Wie bereits bei den komplexeren Prozessbeschreibungen wird nun die formale Richtigkeit untersucht. [2] Die Anzahl an erstellten kritischen und elementaren Fehlern ist in Tabelle 4.3 zu sehen.

Modell		**kritische Fehler**	**elementare Fehler**
Gemini 2.5 Pro	JSON	0	0
	XML	0	0
Gemini 2.5 Flash	JSON	0	0
	XML	0	8
ChatGPT 5.1	JSON	0	0
	XML	0	0
ChatGPT 5.2	JSON	0	0
	XML	0	0
ChatGPT 4.1	JSON	0	0
	XML	0	6
Grok 4	JSON	0	0
	XML	0	0
Grok 4.1 Fast	JSON	0	0
	XML	0	6
Claude Opus 4.5	JSON	0	0
	XML	0	0
Claude Sonnet 4.5	JSON	0	0
	XML	0	0

Tabelle 4.3: Formale Richtigkeit

Die Untersuchung der formalen Richtigkeit bei der mittelschweren Prozessbeschreibung zeigt insgesamt ein deutlich besseres Ergebnis als bei den zuvor betrachteten komplexen Diagrammen. Wie man in Tabelle 4.3 sehen kann, sind bei keinem

---

[2] Wie die formale Richtigkeit festgestellt wird, wir auf Seite 4.1.1 gezeigt.

der getesteten Modelle kritische Fehler aufgetreten, unabhängig vom verwendeten Ausgabeformat und Modell. Dies bedeutet, dass alle erzeugten BPMN-Diagramme grundsätzlich darstellbar waren und keine strukturellen Fehler enthielten.

Ein Großteil der Modelle, darunter Gemini 2.5 Pro, ChatGPT 5.1, ChatGPT 5.2, Grok 4, Claude Opus 4.5, erzeugte sowohl im JSON- als auch im XML-Format vollständig formal korrekte Diagramme ohne elementare Fehler. Eine geringe Menge an elementaren Fehlern sind bei Gemini 2.5 Flash, ChatGPT 4.1 und Grok 4.1 Fast, jeweils im XML-Format aufgetreten. Diese Fehler betreffen zwar lediglich einzelne Diagrammbestandteile und beeinträchtigen nicht das ganze Diagramm, bedeuten aber, dass diese Modelle selbst bei dieser einfacheren Aufgabe Probleme haben.

Semantische Richtigkeit Hier soll nun auch wieder die semenatische Richtigkeit der erstellten Diagramme und damit konkret auf fehlende und unnötige Elemente getestet werden. [3]

Die Ergebnisse in Tabelle 4.4 zeigen, dass die meisten Modelle die mittelschwere Prozessbeschreibung inhaltlich weitgehend korrekt umsetzen können. Fehlende Komponenten treten nur vereinzelt auf und betreffen vor allem ältere oder schnellere Modellvarianten. Unnötige Elemente werden hingegen häufiger ergänzt, beeinflussen jedoch meist nicht den grundlegenden Prozessablauf.

Besonders überzeugend sind ChatGPT 5.2, Gemini 2.5 Pro und Claude Opus 4.5 die besonders wenige Abweichungen zu der Prozessbeschreibung haben. Schwächen zeigen sich bei ChatGPT 4.1, Grok 4.1 Fast und Claude Sonnet 4.5, bei denen einzelne Elemente fehlen oder viele unötige hinzugefügt werden.

Wichtig ist hier noch zu erwähnen, dass Gemini 2.5 Pro die mit Abstand ordentlichsten Diagramme erstellt hat. Alle Elemte sind sinnvoll platziert, es gibt weder zu viel noch zu wenig Abstand zwischen Elementen und auch die Flows sind gut nachzuvollziehen. Dadurch zeigt Gemini 2.5 Pro einen großen Vorsprung gegenüber allen anderen getesteten Modellen.

Zusammnefassend zeigen die Ergebnisse, dass mittelschwere Prozessbeschreibungen zwar von allen getesteten Modellen grundsätzlich verarbeitet werden können, die Qualität der Ergebnisse aber teilweise noch verbessungspotenzial hat.

---

[3] Wie die semantische Richtigkeit festgestellt wird, wir auf Seite 4.1.1 gezeigt.

Modell		**fehlend**	**unnötig**
Gemini 2.5 Pro	JSON	0	0
	XML	0	3
Gemini 2.5 Flash	JSON	0	2
	XML	0	2
ChatGPT 5.1	JSON	2	0
	XML	0	3
ChatGPT 5.2	JSON	0	1
	XML	0	0
ChatGPT 4.1	JSON	3	1
	XML	0	3
Grok 4	JSON	0	1
	XML	0	2
Grok 4.1 Fast	JSON	3	0
	XML	2	0
Claude Opus 4.5	JSON	0	2
	XML	0	1
Claude Sonnet 4.5	JSON	0	4
	XML	6	1

Tabelle 4.4: Semantische Richtigkeit

Während formale Fehler nahezu vollständig vermieden werden, treten weiterhin einige semantische Fehler auf. Modelle wie ChatGPT 4.1, Grok 4.1 Fast und Claude Sonnet 4.5 zeigen hier deutliche Defizite, was ihre Eignung für eine LLM-basierte BPMN Diagramm Erstellung einschränkt. ChatGPT 5.2 und Claude Opus 4.5 erziehlen gute Ergebnisse, mit wenigen Fehlern, wodurch der gewünschte Prozess gut abgebildet werden kann. Alleinig Gemini 2.5 Pro erziehlt im Vergleich zu den anderen Modellen sehr gute Diagramme, die kaum Nachbearbeitung benötigen und zusätzlich noch optisch sehr ansprechend sind.

4.2 Geschwindigkeit

Um die Geschwindigkeit verscheidener Prozessabschnitte und modellen zu testen, bietet es sich an, die Antwort der KI als Stream zu betrachten, da dieser in Echtzeit ausgewertet werden kann. So kann zum Beispiel auch untersucht werden bei welchen Modellen die Textgenerierung und bei welchen die Diagrammgenerierung schneller ist. Um sich aber nur auf gestreamte Daten zu begrenzen, muss zunächst festgestellt werden: Unterscheiden sich die Generierungszeiten eines LLM Anbieters, wenn man streamt bzw. nicht streamt?

Dafür werden nun in Abbildung 4.1 einige Modelle getestet, ob sich die Zeiten jeweils stark unterscheiden. Hierfür wird folgender Prompt verwendet:

> Erstelle eine Prozessbeschreibung eines Beliebigen Prozesses mit 95 bis 105 Wörtern welche 5 tasks, 1 gateway und 2 message flows beinhaltet. Setzte diese dann direkt in ein Diagramm um. Das Diagramm soll nur genau die 5 tasks, 1 gateway und 2 message flows beinhalten, mehr nicht.

Prompt 4.2.1: Prompt für einen Geschwindigkeitstest

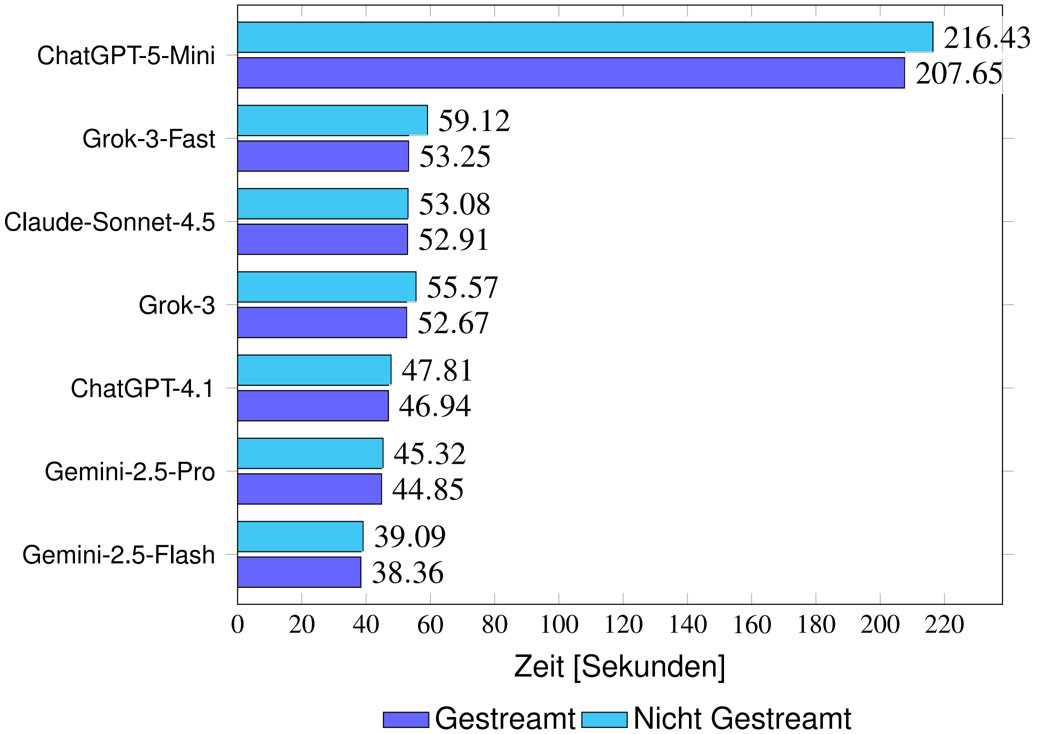

Abbildung 4.1: Zeitperformanzvergleich Gestreamt vs Nicht Gestreamt

Aus den Daten ind Abbildung 4.1 geht hervor, dass bei jedem Modell die Variante des Streamings die Variante ohne Streamings in Bezug auf Geschwindigkeit überbietet. Der Unterschied beträgt jeweils unter 10%. Damit ist nun klar, dass die Variante des Stremings nicht der Variante Ohne Streamings unterliegt und für weitere Tests kann die Variante des Streamings verwendet werden während die Nicht Streaming Variante vernachlässigt wird.

4.2.1 Formatunterschiede

Als nächstes soll nun untersucht werden welches der zwei implementierten Formate `JSON` und `XML` sich Zeitlich besser verhält. In Abbildung 4.2 wird für das Promptbeispiel auf Seite 62 dieses Verhalten getestet. Da die Laufzeit für die Schritte Textgenerierung, Formatierung und Datenbankaufruf, sowie bei Gemini auch Streamstart im Vergleich zu der Diagrammgenereiung sowie der API Antwort sehr wenig Zeit beanspruchen, Sind die exakten Daten auch noch in Tabelle 4.5 zu sehen.

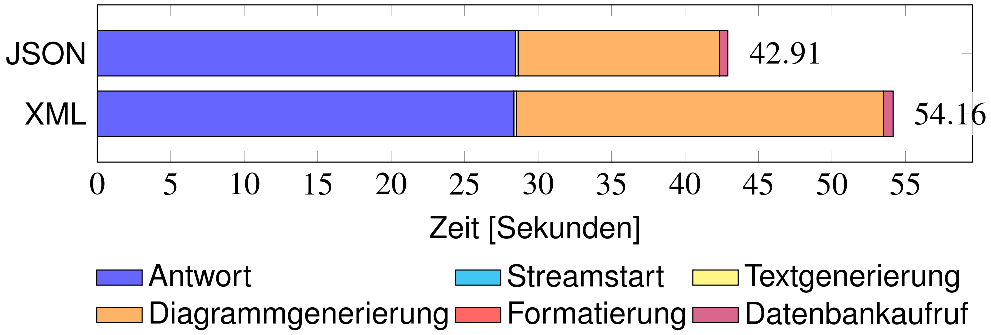

Abbildung 4.2: Zeitperformanzvergleich JSON vs XML bei Gemini 2.5 Pro

Format	Antwort	Stream.	Text.	Diagr.	Form.	Datenb.
XML	28.345	0.004	0.194	24.949	0.000	0.671
JSON	28.453	0.001	0.189	13.707	0.009	0.554

Tabelle 4.5: Zeitperformanzvergleich JSON vs XML bei Gemini 2.5 Pro
Zeit in Sekunden

Man erkannt einfach, dass die Diagrammgenerierung bei JSON um einiges schneller ist als bei XML. Der Konvertierungsprozess von JSON zu XML beträgt in diesem Beispiel nur 9 ms und ist damit um einiges effizienter als die um 11242 ms längere Diagrammgenerierung bei XML.

Interessant ist nun noch zu sehen wie diese Aufwandsdifferenz von der Größe des Diagramms abhängt. Hierfür wird nun in Abbildung 4.3 Gemini 2.5 Pro im Quick modus benutzt um verschieden große Diagramme zu erzeugen. Hierfür wird Prompt 4.2.2 verwendet.

> Erstelle ein BPMN Diagramm für ein Prozess deiner Wahl. Sei kreativ. Benutze insgesamt genau [anzahl-elemente] Elemente wie z.B. Tasks, Gates, End-Events, Message-Flows, Pools, Lanes, etc.

Prompt 4.2.2: Prompt für einen Diagramm-Geschwindigkeitstest nach Größe

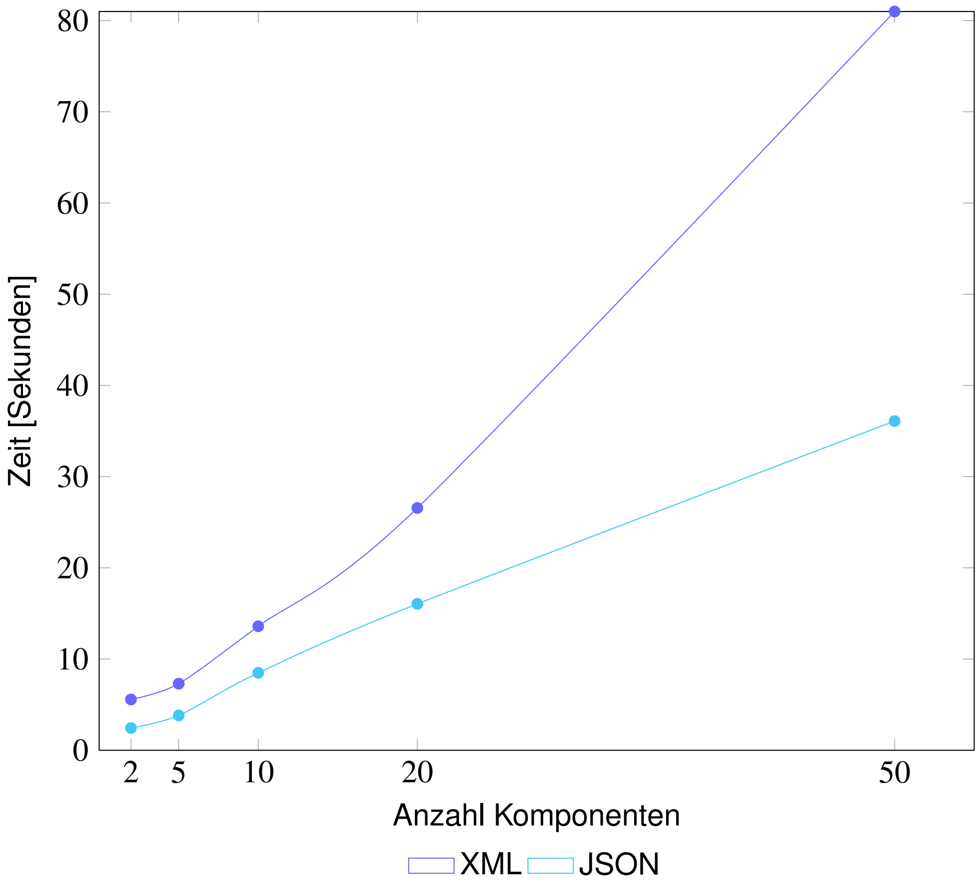

Abbildung 4.3: Zeitperformanzvergleich JSON vs XML bei Gemini 2.5 Pro nach Anzahl der Komponenten (Nur Diagramm)

Die Auswertung der gemessenen Laufzeiten in Abbildung 4.3 zeigt deutlich, dass die Diagrammgenerierung im JSON-Format gegenüber dem XML-Format einen spürbaren Geschwindigkeitsvorteil bietet.

Insbesondere bei zunehmender Diagrammgröße wächst der Unterschied merklich. In den meisten Testfällen liegt die Generierungsdauer des JSON-Modells bei ungefähr der Hälfte der Zeit, die für die entsprechende XML-Ausgabe erforderlich ist. Dieser Geschwindigkeitsvorteil lässt sich vor allem auf die kompaktere Syntax und die geringere Redundanz zurückführen.

Insgesamt wird dadurch klar, dass JSON für zeitperformanzkritische Anwendungsfälle, insbesondere bei großen oder komplexen Diagrammen, erhebliche Vorteile bietet.

4.2.2 Modellunterschiede

Weitergehend soll nun untersucht werden welche Modelle sich für eine Zeiteffiziente Generierung eignen. Dafür werden in Abbildung 4.4 einige gänige Modelle getestet. Die exakten Zeiten stehen in Tabelle 4.6.

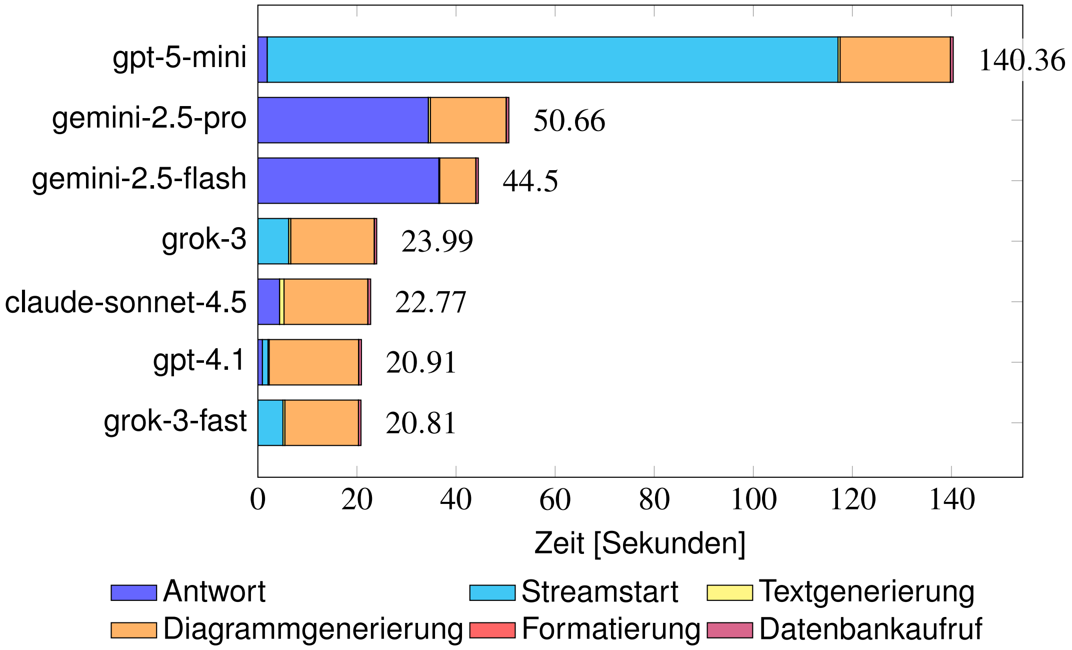

Abbildung 4.4: Zeitperformanzvergleich verschiedener Modelle

Format	Antwort	Stream.	Text.	Diagr.	Form.	Datenb.
gemini-2.5-pro	34.396	0.001	0.433	15.274	0.005	0.554
gemini-2.5-flash	36.529	0.002	0.162	7.292	0.005	0.512
grok-3	0.003	6.203	0.418	16.850	0.005	0.510
grok-3-fast	0.004	5.056	0.386	14.853	0.007	0.508
gpt-4.1	0.904	1.138	0.232	18.051	0.003	0.580
gpt-5-mini	1.874	115.226	0.414	22.258	0.007	0.580
claude-sonnet-4.5	4.370	0.001	0.906	16.903	0.006	0.582

Tabelle 4.6: Zeitperformanzvergleich verschiedener Modelle
Zeit in Sekunden

Auffällig in Abbildung 4.4 ist, dass ChatGPT-5-Mini mit großem Abstand die längste Gesamtzeit benötigt Besonders der Streamstart dauert extrem lange, was darauf hindeutet, dass dieses Modell trotz möglicher inhaltlicher Stärke für eine schnelle Generierung ungeeignet ist. Die beiden Gemini-2.5-Modelle liegen im mittleren Bereich und zeigen ihre Stärken vor allem in der schnellen eigentlichen Antwortphase und soliden Textgenerierung, verlieren jedoch viel Zeit dabei die Anfrage anzuhmen und die Antwort zu übersenden. Grok-3, Grok-3-Fast, Claude-Sonnet-4.5 und ChatGPT-4.1 zeigen die insgesamt ausgewogenste Performance, da keine der Einzeldisziplinen überproportional viel Zeit beansprucht. Besonders Grok-3-Fast und ChatGPT-4.1 sind nahezu gleich schnell und deutlich effizienter als die größeren Modelle. Sie zeichnen sich durch kurze Streamstart-Phasen und schnelle Diagramm- und Texterstellung aus. Insgesamt zeigen die kompakten oder speziell optimierten Modelle eine hohe Reaktionsgeschwindigkeit über alle Teilschritte hinweg, während größere Modelle wie ChatGPT-5-Mini und die Gemini-Reihe durch längere Initialisierungen ausgebremst werden.

4.3 Kosten

Die Tabelle 4.7 zeigt einen Überblick über die Tokenpreise der implementierten KI-Modellanbieter, im Stand von November 2025. Jeder Modellanbieter hat eine breite Abdeckung an Modellen mit unterschiedlichen Preisen. Hierbei gibt es oftmals ein billiges Modell, welches möglicherweise qualitativ schlechtere Ergebnisse erziehlt und ein teureres Modell, welches qualitativ besser ist. Über alle Modelle hinweg wird aber klar, dass Tokens, welche für den Output verwendet werden, mit Abstand am teuersten sind, während Tokens für den Input generell eher billiger sind.

Die Gemini kostenlose Stufe bietet den Vorteil, dass man manche Gemini Modelle, darunter Gemini 2.5, völlig kostenlos nutzen kann. Allerdings gibt es hier auch Nachteile: Laut offizieller Rate-Limits sind beispielsweise bei Gemini 2.5 Flash nur 10 Anfragen pro Minute und 250 Anfragen pro Tag erlaubt.

4.3.1 Input Caching

Betrachtet man nun eine Diagrammerstellung mit dem Prompt von Seite 62, kann man sehen wie sich die Menge der Tokens Für Input und Output auf den Preis auswirken. In Diagramm 4.5 wurde Beispielsweise GPT-4.1 verwendet um ein Diagramm zu erstellen.

Provider	Model	Input	Cached Input	Output
OpenAI	gpt-5.1	1.25 $	0.13 $	10.00 $
OpenAI	gpt-5	1.25 $	0.13 $	10.00 $
OpenAI	gpt-5-mini	0.25 $	0.03 $	2.00 $
OpenAI	gpt-5-nano	0.05 $	0.01 $	0.40 $
OpenAI	gpt-5-pro	15.00 $	0.00 $	120.00 $
OpenAI	gpt-4.1	2.00 $	0.50 $	8.00 $
OpenAI	gpt-4.1-mini	0.40 $	0.10 $	1.60 $
OpenAI	gpt-4.1-nano	0.10 $	0.03 $	0.40 $
OpenAI	gpt-4o	2.50 $	1.25 $	10.00 $
OpenAI	gpt-4o-mini	0.15 $	0.08 $	0.60 $
Anthropic	claude-opus-4-1	15.00 $	1.50 $	75.00 $
Anthropic	claude-sonnet-4-5	3.00 $	0.30 $	15.00 $
Anthropic	claude-haiku-4-5	1.00 $	0.10 $	5.00 $
Anthropic	claude-sonnet-4	3.00 $	0.30 $	15.00 $
Anthropic	claude-opus-4	15.00 $	1.50 $	75.00 $
Anthropic	claude-sonnet-3-7	3.00 $	0.30 $	15.00 $
Anthropic	claude-haiku-3-5	0.80 $	0.08 $	4.00 $
Anthropic	claude-opus-3	15.00 $	1.50 $	75.00 $
xAI	grok-4-1-fast-reasoning	0.20 $	0.05 $	0.50 $
xAI	grok-4-1-fast-non-reasoning	0.20 $	0.05 $	0.50 $
xAI	grok-4-fast-reasoning	0.20 $	0.05 $	0.50 $
xAI	grok-4-fast-non-reasoning	0.20 $	0.05 $	0.50 $
xAI	grok-4	3.00 $	0.05 $	15.00 $
xAI	grok-3-mini	0.30 $	0.08 $	0.50 $
xAI	grok-3	3.00 $	0.75 $	15.00 $
Google	gemini-2.5-pro	0.00 $	0.00 $	0.00 $
Google	gemini-2.5-flash	0.00 $	0.00 $	0.00 $
Google	gemini-2.5-flash-lite	0.00 $	0.00 $	0.00 $
Google	gemini-2.0-flash	0.00 $	0.00 $	0.00 $

Tabelle 4.7: Tokenpreise pro 1M Token
(Stand: 20.11.2025)

Wie man sieht sind, obwohl die input token preise um einiges billiger sind als die output token preise, die Kosten des Inputs für eine Diagrammerstellung in der ersten Nachricht mehr als 50%. In der fünften Nachricht sind die Input Token Kosten bereits bei über 75%. Während die Kosten für das erste Diagramm noch 0.0466 $

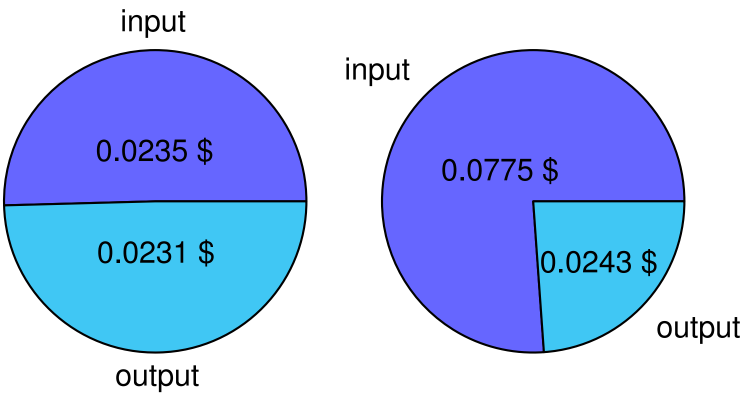

Abbildung 4.5: Kosten für die erste und fünfte Diagrammerstellung im Thread

betragen, sind es bei der fünften Nachricht schon 0.1018 $. Da der Input bei Folgenachrichten zum Großteil der gleiche ist, wie der bei Nachrichten davor, Bieten viele LLM Anbieter eine Funktionalität des `Input Caching` an.

Input zu cachen kann viel bringen, wenn sich Teile einer Anfrage immer wiederholen. Viele Inhalte der Anfragen ist dem LLM Anbieter durch vorherige ANchrichten in einem Thread bereits bekannt. Diese Inhalte jedes Mal neu an das Modell zu schicken, kostet viele Tokens und damit Geld. Wenn der Input aber gecacht wird, kann das Modell auf eine bereits gespeicherte interne Darstellung zurückgreifen. Dadurch muss es die Daten nicht noch einmal vollständig verarbeiten.

Die Tabelle 4.7 zeigt, dass gecachter Input bei vielen Modellen deutlich günstiger ist als normaler Input, bei manchen Modellen sogar gar nichts. Das bedeutet: Je mehr wiederverwendete Daten eine Anfrage hat, desto stärker sinken die Gesamtkosten. Gleichzeitig antwortet das Modell schneller, weil weniger berechnet werden muss.

Man sieht in Abbildung 4.6 gut, dass das Caching des Inputs viel Geld sparen kann. Während in diesem Beispiel das fünfte Diagramm ohne Caching noch 0.1018 $ gekostet hat, hat das fünfte Diagramm mit Caching nur noch 0.0578 $ gekostet. Für dieses Beispiel hat sich Caching sehr gelohnt.

Das Caching hat aber für den BPMN Bot nur begrenzt einen Effekt. Es ist Teil der Software, dass der Nutzer das KI Modell komplett frei wählen kann. Dies kann er auch innerhalb eines Threads bei jeder neuen Nachricht entscheiden und ist dabei nicht an einen Anbieter gebunden. Dadurch muss bei einem Modellwechsel trotzdem der gesamte Threadkontext an das neue Modell gesendet werden.

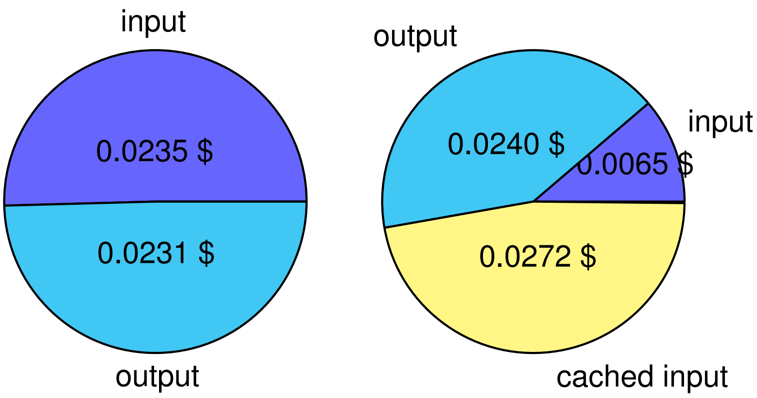

Abbildung 4.6: Kosten für die erste und fünfte Diagrammerstellung im Thread mit Input-Caching

4.3.2 Formatunterschiede

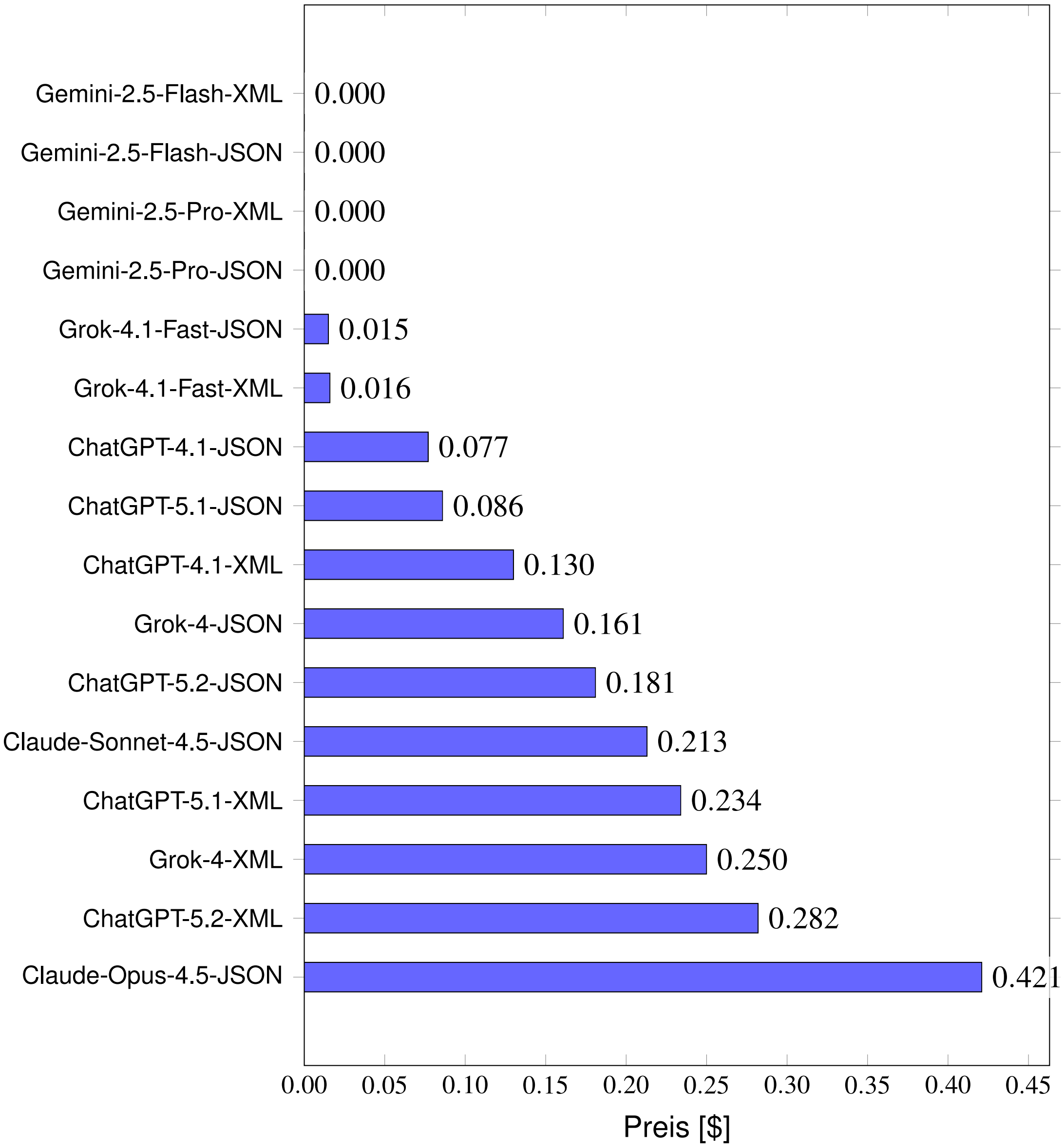

Abbildung 4.7: Kosten verschieder Modelle bei Prozessbeschreibung 4.1.1

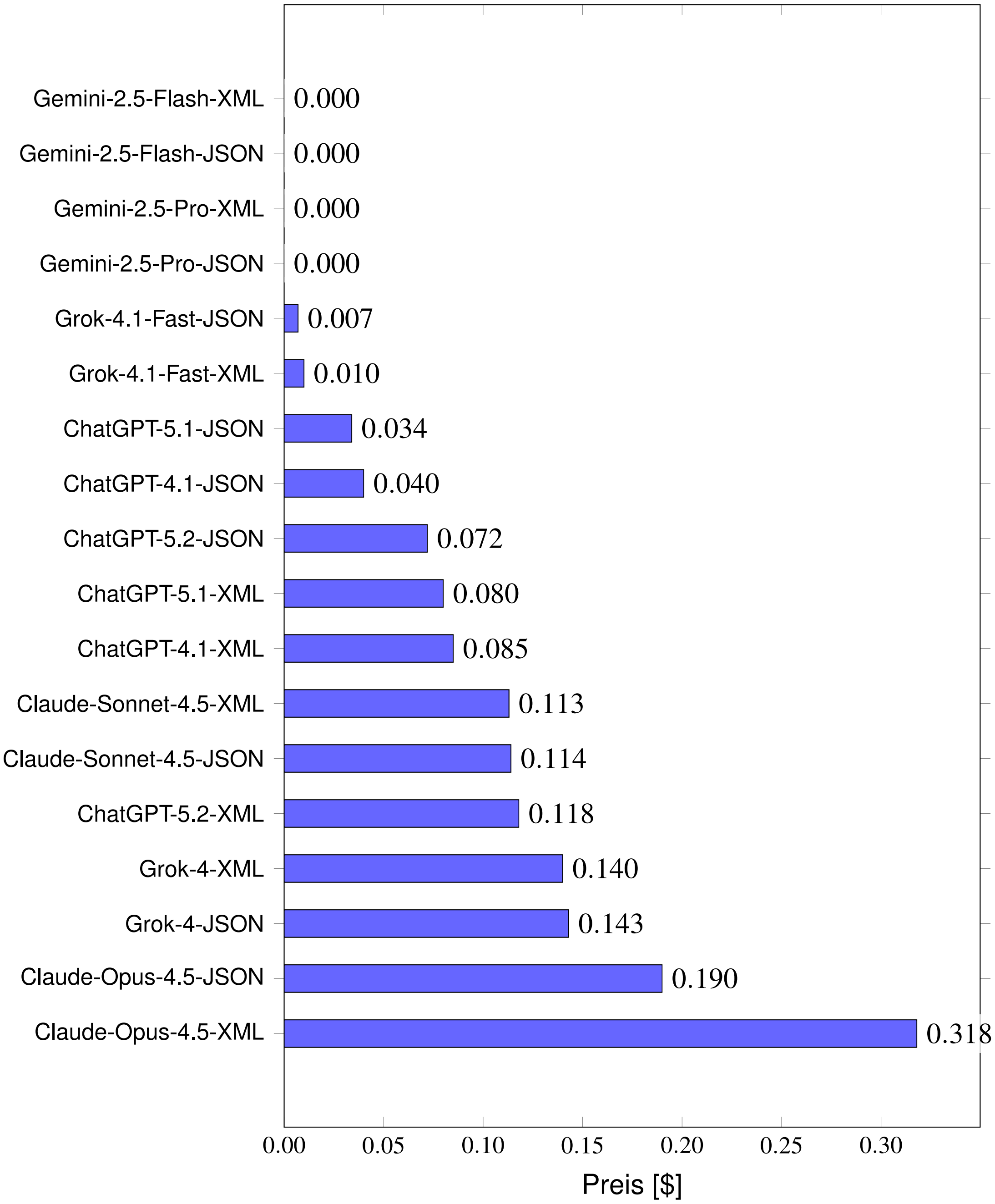

Abbildung 4.8: Kosten verschieder Modelle bei Prozessbeschreibung 4.1.2

Wie auch schon im Kapitel 4.2 besprochen, hat auch die Wahl des Diagrammformats eine entscheidenden Rolle. Die Nutzung von JSON gegenüber XML kann eine Einsparung von bis zu etwa 50% der Output Token bewirken, wodurch weiter einiges an Kosten gespart werden kann.

Zusammenfassend lässt sich festhalten, dass es verschiedene Strategien gibt, die Kosten für die Nutzung von KI-Modellen zu reduzieren. Ein naheliegender Ansatz ist die Auswahl eines Modells mit niedrigen Tokenpreisen. Dabei besteht jedoch immer das Risiko, dass günstigere Modelle auch qualitativ schwächere Ergebnisse liefern. Für bestimmte Anwendungsfälle kann das ausreichend sein, bei komplexeren Diagrammen kann es jedoch zu deutlichen Qualitätseinbußen kommen. Eine weitere Möglichkeit bietet das kostenfreie Gemini-Angebot. Allerdings bringt die kostenlose Stufe klare Einschränkungen mit sich, insbesondere die strengen Limits an täglichen und minütlichen Anfragen. Dadurch wird die skalierung der Software auf eine große Nutzeranzahl verhindert. Auch das Input-Caching kann eine wirksame Methode zur Kostenreduzierung sein. Insbesondere bei langen Chats, kann das Caching den Preis pro Anfrage deutlich senken. Gleichzeitig verbessert sich die Antwortgeschwindigkeit, da bereits bekannte Inhalte nicht erneut ausgewertet werden müssen.

5 Verwandte Arbeiten

Zunächst sind hier natürlich die zwei Arbeiten zu nennen, auf denen diese Arbeit hier basiert. Dies ist einmal ‘BPMN diagram generation with ChatGPT’ von Weidl [20] und ‘Enhancing BPMNGen: Improving LLM-based BPMN 2.0 Process Model Generation through Natural Language Processing’ von Shi [16]. Die genannten zwei Arbeiten bilden die Grundlage für dieses Arbeit.

ProMoAI Es existieren auch andere ähnliche Projekte, welche Prozessmodelle mit Hilfe von LLMs erstellen. Dazu gehört das Projekt ProMoAI [1] von Kourani et al. [10] an der RWTH Aachen University. ProMoAI setzt mehrere Prompting-Strategien ein, um Prozessmodelle aus Text zu erzeugen. Dazu gehört Role Prompting, bei dem das LLM als Prozessexperte und Prozessowner eingesetzt wird, sowie Few-Shot-Prompting, bei dem Beispielmodelle vorgegeben werden, damit das LLM typische Muster lernt. Zusätzlich nutzt ProMoAI Negative Prompting, um häufige Modellierungsfehler explizit zu vermeiden. Ein wichtiger Unterschied zu dieser Arbeit ist die Fokussierung auf die von Kourani et al. selbst entwickelte Modellierungssprache POWL [9], deren Struktur und klar definierte Operatoren dazu dienen, dass das LLM Prozessmodelle in einem gut überprüfbaren Format generiert.

AutoBPMN.AI Ein anderes Projekt ist AutoBPMN.AI[2] von Klievtsova et al. [6] an der Technischen Universität München. Bei diesem Projekt steht auch, wie bei BPMNGen, die konversationelle Modellierung im Vordergrund. AutoBPMN.AI erstellt oder updatet Prozessmodelle in einem iterativen Austausch zwischen Nutzer und einem LLM, ähnlich wie bei dem Projekt dieser Arbeit. Hierbei kann an den Chatbot entweder eine Prozessbeschreibung oder eine Prozessbeschreibung inkl.

---

[1] `https://promoai.streamlit.app/`

[2] `https://autobpmn.ai/`

bereits erstelltem Prozessmodell geschickt werden, worauf mit einem Prozessmodell geantwortet wird. Es werden allerdings nicht BPMN 2.0 Prozessmodelle implementiert, sondern Graphen in der Software CPEE [3]. Für die KI wird allerdings so, wie auch bei BPMNGen, ein anderes Format für die KI benutzt. AutoBPMN.AI setzt hier auf Mermaid[4]. Dieses Format könnte auch für BPMNGen interessant sein. In den veröffentlichten Arbeiten werden die genauen Prompting-Strategien jedoch nur sehr begrenzt beschrieben, sodass sich hier nur schwer konkrete Strategien ableiten lassen.

NaLa2BPMN Ein weiteres relevantes Projekt ist NaLa2BPMN[5], von Nour Eldin et al. [12, 11] das ebenfalls LLMs zur automatischen Erzeugung von BPMN-Modellen aus Text nutzt. NaLa2BPMN verfolgt einen hybriden Ansatz, bei dem der Gesamtprozess in zwei strukturierte Schritte zerlegt wird. Zunächst wird das LLM eingesetzt, um den Nutzertext zu analysieren, zu verbessern und fehlende Details zu ergänzen. Anschließend werden Aktivitäten, Events, Abhängigkeiten und Verzweigungen von dem LLM identifiziert und als Graph in Textform dargestellt. Als zweiter Schritt baut ein algorithmischer, deterministischer Teil aus den extrahierten Informationen das BPMN-Modell. Der Algorithmus unterstütz dabei Techniken wie Loop-Filtering, Split & Join Discovery und Loop Construction, welche Fehler entfernen können und die Qualität verbessern. Laut einer Qualitätsstudie erzeugt NaLa2BPMN bessere Diagramme als ProMoAI [11].

LLM4PM Ein weiteres aktuelles Projekt ist LLM4PM von Ziche et al. [23], das in einer realen Unternehmensumgebung untersucht, wie LLMs zur Unterstützung der Prozessmodellierung eingesetzt werden können. Hierbei wurde der LLM-gestützte Chatbot PRODIGY entwickelt, um Prozessmodellierer bei der Erstellung von Prozessdiagrammen zu unterstützen. Der Ansatz nutzt dialogorientiertes Prompting, bei dem Nutzer in natürlicher Sprache Anfragen an das System stellen und das LLM schrittweise Rückfragen und Modellteile generiert, die in bestehende Modellierungswerkzeuge übernommen werden können. Hierbei ist das System sehr ähnlich wie der in dieser Arbeit verwendete Chain-of-Thought Ansatz, mit dem Unter-

---

[3] `https://cpee.org/index.php?t=cpee`

[4] `https://mermaid.js.org/`

[5] `https://nala2bpmn.bonitapps.com/`

schied, dass PRODIGY mehr darauf ausgelegt ist, einzelne Elemente des Prozessdiagramms zu optimieren, als vollständige Diagramme zu erzeugen. Ein zentrales Element bei LLM4PM ist die Integration unternehmensspezifischer Dokumentation, die über Retrieval-Augmented Generation eingebettet wird, um das LLM mit organisationsspezifischem Kontextwissen zu versorgen und die Qualität der generierten Modelle zu verbessern. Dadurch zielt LLM4PM weniger auf die direkte automatische Generierung vollständiger BPMN-Modelle ab, sondern eher auf eine assistierende, interaktive Modellierungsunterstützung in realen Unternehmensumgebungen. PRODIGY selber kann keine Diagramme erstellen, sondern erstellt formatierten Klartext, welcher von dem Tool BPMN Sketch miner[6] interpretiert und in elementare Diagramme umgesetzt werden kann.

BPMN-Chatbot++ Ein weiteres relevantes Projekt ist der BPMN-Chatbot++ von Köpke und Safan [14], ein dem hier vorgestellten BPMNGen sehr ähnliches Projekt, das erstmals auf der BPM-Konferenz 2024 [7] präsentiert und anschließend weiterentwickelt wurde. Im Rahmen dieses Projekts entstand außerdem eine React-basierte Anwendung [8] [7], die zur Generierung von BPMN-2.0-Diagrammen mit BPMN-Chatbot++ genutzt werden kann. Zwischen BPMNGen und BPMN-Chatbot++ bestehen einige Parallelen. Dazu zählt die dynamische Prompt-Generierung, durch die das Modell zum Beispiel aufgefordert werden kann, konkrete Fehler im Diagramm zu beheben. Ebenso wird ein Zwischenmodell in JSON erzeugt, in dem die KI antworten soll und das anschließend mittels Model2Model-Transformation in BPMN-XML überführt wird. Es wurde zunächst Zero-Shot-Prompting ausprobiert aber dann auf Few-Shot-Prompting umgestellt. Auch das Schema-Constraining zur Validierung des generierten JSON-Modells wird wie bei BPMNGen verwendet. Ein wesentlicher Aspekt des BPMN-Chatbot++ sind die integrierten Model checking components [14, 13]. Diese sind direkt auf dem Zwischenmodell implementiert und ermöglichen es konkrete Fehler zu erkennen. Implementiert wurden hierfür sowohl der Process Application Validator (vPAV) [15] als auch der S^3 checker [2]. Durch deren Rückmeldungen erhalten sowohl der Nutzer als auch die KI eine Übersicht, welche Probleme im aktuellen Diagramm noch bestehen. Durch diese Validatoren kann die Qualität der Diagramme Schritt für Schritt verbessert werden, da die KI

---

[6] `https://www.bpmn-sketch-miner.ai/`

[7] `https://bpmnchatbot.aau.at/pubserv/BPMN-Chatbot-Beta/`

anhand der gefundenen Fehler auch semantische Fehler korrigieren kann.

6 Fazit

6.1 Zusammenfassung

In dieser Arbeit wurde das bestehende BPMNGen-System grundlegend erweitert und modernisiert, um die automatische Erstellung und Bearbeitung von BPMN-Diagrammen mit Hilfe von Large Language Models deutlich zu verbessern. Dazu wurde zunächst die Architektur vollständig neu strukturiert und in ein objektorientiertes System überführt, das mehrere KI-Anbieter flexibel unterstützt. Durch optimierte Instructions, neue Interaktionsmodi, das Einbeziehen des Chatverlaufs und des aktuellen Diagrammzustands sowie Funktionen wie Datei-Uploads, Streaming und die Kombination von Text- und Diagrammausgaben konnte die Qualität der Diagramme und die Nutzererfahrung deutlich gesteigert werden. Ergänzende Methoden wie Reflective Prompting und Diagramm-Sampling ermöglichen zudem eine präzisere und vielfältigere Modellgenerierung. Insgesamt zeigt die Arbeit, dass moderne Prompting-Strategien und flexible Systemarchitekturen einen entscheidenden Beitrag dazu leisten können, LLMs sinnvoll und effizient für die BPMN-Modellierung einzusetzen.

6.2 Ausblick

Obwohl das BPMNGen-System nun deutliche Fortschritte erzielt hat, bieten sich zahlreiche Ansatzpunkte für zukünftige Entwicklungen. In diesem Abschnitt soll gezeigt werden, inwiefern das Prompting und die Diagrammerstellung weiter verbessert werden könnte.

Auto layouting Viele generierte Diagramme sind korrekt, aber optisch unübersichtlich. Man könnte ein KI-Layoutmodell integrieren oder einen heuristischen Ansatz wählen. Es gibt hierzu bereits Ansätze wie das BPMN-Auto-Layout. [1] Dieses kann aber zum aktuellen Stand nicht ohne Überarbeitung zum Projekt hinzugefügt werden.

Automatische Prozessoptimierung Der BPMNGen Chatbot könnte auch, ohne dass der Nutzer ihn darum bittet, Vorschläge zur Optimierung machen. Dies könnten sowohl semantische Vorschläge wie unnötige Elemente, fehlende Gates, etc. als auch formale Vorschläge wie fehlende oder duplizierte IDs oder sogar inhaltliche Vorschläge wie veränderte Prozesse sein. Der Bot könnte diese Vorschläge ganz ohne Aufforderung zum Beispiel als PopUp anzeigen.

Fine Tuning Der Chatbot könnte mit den Daten der generierten Diagramme weitertrainiert werden. Hierbei könnte zum Beispiel mit Hilfe der Änderungen, welche der Nutzer selber durchführt, dem Chatbot mitgeteilt werden, was er zukünftig besser machen könnte. Alternativ könnten auch die Änderungsanweisungen genutzt werden, um dem Chatbot Verbesserungen aufzuzeigen. Somit wäre es möglich, dass der Bot mit der Zeit immer besser wird. Es könnte hier sowohl ein generelles Fine-Tuning des BPMNGen Chatbots erstellt werden, sowie ein nutzerspezifisches Fine-Tuning.

---

[1] `https://github.com/bpmn-io/bpmn-auto-layout`

A Quelltexte

```
export type format = "xml" | "json";
export type mode = "quick" | "detail";
export type sampling = "false" | "primary" | "secondary";
export type diagramResponse = {model?: string, text?: string; xml?: string; json?: DiagramJson | string;
                               threadId?: string, samples?: {}};
export type modelPriority = Record<format, number>;

export abstract class Ai {
  private createDebugFile: boolean = PRODUCTION.toLowerCase() == "false";

  public model: string = "";
  public format: format = "xml";

  protected constructor(model?: string, format?: format) {
    this.model = model ?? this.model;
    this.format = format ?? this.format;
  }

  public async createBPMN(prompt: string, file: string, userID: number, res: Response, mode: mode = "quick",
                          stream = false, sampling: sampling = "false"): Promise<diagramResponse> {
    const threadID = await this.createThread();
    const instructions = Array.prototype.concat(Ai.formatInstructions(this.format), Ai.modeInstructions(mode))
    const promptInput = new PromptInput(instructions, prompt, await Ai.getAllChatsFromDB(threadID), null, file);
    const input = this.mapPromptInput(promptInput, mode, stream);
    if (!input) throw new Error("Unable to create input for the ai");
    Ai.getMeasurements(threadID);
    Ai.startStopwatches(['api-response', 'full-response'], threadID)
    const response = await this.generateContent(input, mode, stream);
    Ai.endStopwatch('api-response', threadID)
    if (!response) throw new Error("Ai unreachable");
    const diagramOutput = this.isStream(response)
      ? await this.startStream(response, res, threadID, mode, sampling)
      : await this.startResponse(response, res, threadID, mode, sampling);
    Ai.endStopwatch('full-response', threadID)
    if (!diagramOutput) throw new Error("The ai response is not valid");
    const titleInstructions = "[no prose] [only return title] Find a fitting title for the given scenario";
    const chatTitle = await this.createTitle(`${titleInstructions}\n\n${prompt}`);
    if (!chatTitle) throw new Error("Unable to create title for the thread");
    Ai.startStopwatch('database-save', threadID)
    if (sampling !== "secondary")
      await Ai.saveNewThreadToDB(threadID, userID, chatTitle, prompt, diagramOutput);
    Ai.endStopwatch('database-save', threadID)
    if (sampling === "false" && stream) {
      res.write(`event: save\ndata: success\n\n`);
      res.end();
    } else if (sampling === "false") {
      res.status(201).json(Ai.diagramOutputToStringVersion(diagramOutput));
    }
    return diagramOutput;
  }

  public async updateBPMN(prompt: string, file: string, threadID: string, res: Response, mode: mode = "quick",
                          stream = false, sampling: sampling = "false"): Promise<diagramResponse> {
    const instructions = Array.prototype.concat(Ai.formatInstructions(this.format), Ai.modeInstructions(mode),
                                                await Ai.updateInstructions(threadID, this.format))
    const promptInput = new PromptInput(instructions, prompt, await Ai.getAllChatsFromDB(threadID),
```

```
                                        undefined, file);
    const input = this.mapPromptInput(promptInput, mode, stream);
    if (!input) throw new Error("Unable to create update input for the ai");
    const response = await this.generateContent(input, mode, stream);
    if (!response) throw new Error("Ai unreachable");
    const diagramOutput = this.isStream(response)
      ? await this.startStream(response, res, threadID, mode, sampling)
      : await this.startResponse(response, res, threadID, mode, sampling);
    if (!diagramOutput) throw new Error("The ai response is not valid");
    if (sampling !== "secondary")
      await Ai.saveUpdatedThreadToDB(threadID, prompt, diagramOutput);
    if (sampling === "false" && stream) {
      res.write(`event: save\ndata: success\n\n`);
      res.end();
    } else if (sampling === "false") {
      res.status(201).json(Ai.diagramOutputToStringVersion(diagramOutput));
    }
    return diagramOutput;
  }

  protected async createThread(): Promise<string> {
    return crypto.randomUUID();
  }

  protected abstract generateContent(input: any, mode: mode, stream: boolean): Promise<any>;

  protected abstract createTitle(prompt: string): Promise<string>;

  public abstract getModelNamesWithPriority(): Map<string, modelPriority>;

  get modelPriority(): number {
    switch (this.format) {
      case "xml": return this.getModelNamesWithPriority().get(this.model)?.xml ?? -1;
      case "json": return this.getModelNamesWithPriority().get(this.model)?.json ?? -1;
      default: return -1;
    }
  }

  public getFormatsWithPriority(): Map<format, number> {
    return new Map([["xml", 1,], ["json", 0]]);
  }

  get formatPriority(): number {
    return this.getFormatsWithPriority().get(this.format) ?? -1;
  }

  protected mapPromptInput(input: PromptInput, mode: mode, stream: boolean): any {
    return input.join("\n\n");
  }

  private async startResponse(obj: any, res: Response, threadId: string, mode: mode,
                              sampling: sampling = "false"): Promise<diagramResponse | null> {
    Ai.startStopwatch('format-conversion', threadId)
    const stringResponse = this.processResponse(obj);
    if (!stringResponse) throw new Error("Invalid ai response format");
    const diagramOutput = Ai.convertResponseToDiagramOutput(stringResponse, this.format, mode);
    if (!diagramOutput) throw new Error("Unable to convert response to diagram output");
    if (sampling === "secondary")
      diagramOutput.model = `${this.model} (${this.format})`;
    const tokens = await this.retrieveTokens(obj, threadId);
    const prize = await this.calculatePrize(tokens);
    obj.tokens = tokens;
    obj.prize = prize;
    if (diagramOutput.xml && this.createDebugFile)
      await Ai.generateDebugFile(diagramOutput.xml, threadId, obj, `${this.model} (${this.format})`);
    if (sampling !== "secondary")
      diagramOutput.threadId = threadId;
    Ai.endStopwatch('format-conversion', threadId)
    return diagramOutput;
  }

```

```
protected abstract processResponse(response: any): string;

protected isStream(obj: any): boolean {
  return false;
}

private async startStream(obj: any, res: Response, threadId: string, mode: mode,
                          sampling: sampling = "false"): Promise<diagramResponse | null> {
  const sendSSE = (event: string, payload: any) => {
    if (!payload) return
    res.write(`event: ${event}\ndata: ${payload.replace(/\r?\n/g, () => "\ndata:")}\n\n`);
  };
  const debugDataCallback = (data: any) => {
    obj = data;
  }
  const tokenCallback = (token: string) => {
    Ai.endStopwatch('stream-initialization', threadId);
    try {
      buffer += token;
      const data = Ai.convertStringToStreamData(buffer, this.format);
      if (data && data.currentlyLargeDiagram && !inDiagramStream) {
        // diagram streaming started
        inDiagramStream = true;
        diagramStreamFinished = false;
        sendSSE("diagram-start", this.model);
        Ai.endStopwatch('text-generation', threadId);
        Ai.startStopwatch('diagram-generation', threadId);
      }
      if (data && data.currentlyText && inDiagramStream && !diagramStreamFinished) {
        // diagram streaming ended
        diagramStreamFinished = true;
        sendSSE("diagram-end", this.model);
        if (this.format == "xml")
          sendSSE("diagram", data.largeDiagrams.at(-1) ?? "");
        if (this.format == "json")
          sendSSE("diagram", convertJsonToXml(JSON.parse(data.largeDiagrams.at(-1) ?? "")));
        Ai.endStopwatch('diagram-generation', threadId);
      }
      if (data && data.text && data.text.length > textBuffer.length) {
        let textDelta = data.text.replace(textBuffer, "").replace("\u0004", "");
        textBuffer = data.text;
        sendSSE("delta", textDelta);
        if (!Ai.isStopwatchRunning('text-generation', threadId))
          Ai.startStopwatch('text-generation', threadId);
      }
    } catch (error) {
      sendSSE("error", String(error));
      res.end()
    }

  }
  const error = (error: any) => {
    sendSSE("error", error);
    res.end()
  }
  let buffer = "";
  let textBuffer = "";
  let inDiagramStream = false;
  let diagramStreamFinished = false;

  Ai.startStopwatches(['stream-initialization', 'full-stream'], threadId)
  if (sampling !== "secondary")
    res.writeHead(202, {
      "Content-Type": "text/event-stream",
      "Cache-Control": "no-cache",
      "Connection": "keep-alive",
    });
  // Heartbeat alle Sekunde
  const heartbeat = setInterval(() => {
    sendSSE("alive", new Date().toLocaleString());
  }, 1000);
```

```

    if (sampling !== "secondary")
      sendSSE("start", threadId);
    await this.processStream(obj, tokenCallback, error, debugDataCallback);
    tokenCallback('\u0004') // END OF TEXT token
    Ai.startStopwatch('format-conversion', threadId)
    const diagramOutput = Ai.convertResponseToDiagramOutput(buffer, this.format, mode);
    Ai.endStopwatch('format-conversion', threadId)
    if (!diagramOutput) throw new Error("Unable to parse response to correct output");
    if (sampling === "secondary")
      diagramOutput.model = `${this.model} (${this.format})`;
    if (sampling !== "secondary")
      diagramOutput.threadId = threadId;
    Ai.endStopwatches(['full-stream', 'text-generation'], threadId);
    const tokens = await this.retrieveTokens(obj, threadId);
    const prize = await this.calculatePrize(tokens);
    obj.tokens = tokens;
    obj.prize = prize;
    if (diagramOutput.xml && this.createDebugFile)
      await Ai.generateDebugFile(diagramOutput.xml, threadId, obj, `${this.model} (${this.format})`);
    if (sampling === "false")
      sendSSE("end", JSON.stringify(Ai.diagramOutputToStringVersion(diagramOutput)));
    clearInterval(heartbeat);
    return diagramOutput;
  }

  protected async processStream(stream: any, token: (content: string) => void, error: (content: string) => void,
                                debugData: (content: any) => void): Promise<void> {}

  protected async retrieveTokens(response: any, threadId: string): Promise<{ [key: string]: number }> {
    return {};
  }

  protected async calculatePrize(tokens:{ [key: string]: number }) : Promise<{ [key: string]: number }> {
    return {};
  }

  ////////////////////////////////////////
  /////////// HELPER FUNCTIONS ///////////
  ////////////////////////////////////////

  ////////// DATABASE FUNCTIONS //////////

  protected static async getLatestDiagramFromDB(threadID: string) {
    const allChats = await this.getAllChatsFromDB(threadID, "desc");
    if (!allChats) return null;
    for (const chatMessage of allChats) {
      const diagram = await this.getDiagramFromDB(chatMessage)
      if (diagram) {
        return diagram;
      }
    }
  }

  protected static async getLatestChatFromDB(threadID: string) {
    const currChat = await handlePrisma(() =>
      getPrisma().chatMessage.findFirst({
        where: {
          threadId: threadID,
        },
        orderBy: {
          createdAt: "desc",
        },
      })
    );
    if (isError(currChat)) {
      console.error("Error fetching current chat message");
      return null;
    }

    if (!currChat) {
```

```
    console.debug("No chat found!");
    return null;
  }
  return currChat;
}

protected static async getAllChatsFromDB(threadID: string, sortOrder: "asc" | "desc" = "asc") {
  const allChats = await handlePrisma(() =>
    getPrisma().chatMessage.findMany({
      where: {
        threadId: threadID,
      },
      orderBy: {
        createdAt: sortOrder,
      },
    })
  );
  if (isError(allChats)) {
    console.error("Error fetching all thread chat messages");
    return undefined;
  }

  if (!allChats) {
    console.debug("No chats found!");
    return undefined;
  }
  return allChats;
}

protected static async getDiagramFromDB(currChat: {chatMessageId: any; text: string} | null) {
  if (!currChat) {
    return null;
  }
  const currDiagram = await handlePrisma(() =>
    getPrisma().diagram.findFirst({
      where: {
        chatMessageId: currChat.chatMessageId,
      },
      orderBy: {
        lastEditedAt: "desc",
      },
    })
  );
  if (isError(currDiagram)) {
    console.error("Error fetching current diagram");
    return null;
  }

  if (!currDiagram) {
    console.debug("no Diagram in Chat found! (Scenario: " + currChat.text + ")");
    return null;
  }
  return currDiagram;
}

protected static async saveNewThreadToDB(threadID: string, userID: number, title: string, prompt: string,
                                         diagramOutput: diagramResponse) {
  const messages: { text: string; author: Author; diagrams?: { create:
    { xmlContent: string; jsonContent: InputJsonValue; previewImageSVG?: string | null;};
  }; }[]  = []
  if (diagramOutput.xml && !diagramOutput.text) {
    // message 1: prompt + xml (+ json)
    messages.push({
        text: prompt,
        author: Author.USER,
        diagrams: {
          create: {
            xmlContent: diagramOutput.xml ?? "",
            jsonContent: diagramOutput.json as InputJsonValue ?? "",
            previewImageSVG: diagramOutput.xml ? await xmlToSvg(diagramOutput.xml) : null,
          },
```

```
        },
      }
    )
  } else {
    // message 1: prompt
    messages.push({
        text: prompt,
        author: Author.USER,
      }
    )
  }
  if (diagramOutput.text && diagramOutput.xml) {
    // message 2: text + xml (+ json)
    messages.push({
        text: diagramOutput.text,
        author: Author.AI,
        diagrams: {
          create: {
            xmlContent: diagramOutput.xml ?? "",
            jsonContent: diagramOutput.json as InputJsonValue ?? "",
            previewImageSVG: diagramOutput.xml ? await xmlToSvg(diagramOutput.xml) : null,
          },
        },
      }
    )
  } else if (diagramOutput.text && !diagramOutput.xml) {
    // message 2: text
    messages.push({
        text: diagramOutput.text,
        author: Author.AI,
      }
    )
  } else {
    // no message 2
  }
  const success = await handlePrisma(() =>
    getPrisma().thread.create({
      data: {
        userId: userID!,
        threadId: threadID,
        title: title,
        chatMessages: {
          create: messages,
        },
      },
    })
  );
  if (isError(success)) {
    console.error("Error creating new thread");
    throw new Error("Error saving new thread");
  }
}

protected static async saveUpdatedThreadToDB(threadID: string, prompt: string, diagramOutput: diagramResponse) {
  const messages: { text: string; threadId: string; author: Author; diagrams?: { create:
    { xmlContent: string; jsonContent: InputJsonValue; previewImageSVG?: string | null;};
  }; }[] = []
  if (diagramOutput.xml && !diagramOutput.text) {
    // message 1: prompt + xml (+ json)
    messages.push({
        text: prompt,
        threadId: threadID,
        author: Author.USER,
        diagrams: {
          create: {
            xmlContent: diagramOutput.xml ?? "",
            jsonContent: diagramOutput.json as InputJsonValue ?? "",
            previewImageSVG: diagramOutput.xml ? await xmlToSvg(diagramOutput.xml) : null,
          },
        },
      }
```

```
      )
    } else {
      // message 1: prompt
      messages.push({
          text: prompt,
          threadId: threadID,
          author: Author.USER,
        }
      )
    }
    if (diagramOutput.text && diagramOutput.xml) {
      // message 2: text + xml (+ json)
      messages.push({
          text: diagramOutput.text,
          threadId: threadID,
          author: Author.AI,
          diagrams: {
            create: {
              xmlContent: diagramOutput.xml ?? "",
              jsonContent: diagramOutput.json as InputJsonValue ?? "",
              previewImageSVG: diagramOutput.xml ? await xmlToSvg(diagramOutput.xml) : null,
            },
          },
        }
      )
    } else if (diagramOutput.text && !diagramOutput.xml) {
      // message 2: text
      messages.push({
          text: diagramOutput.text,
          threadId: threadID,
          author: Author.AI,
        }
      )
    } else {
      // no message 2
    }
    const success1 = await handlePrisma(() =>
      getPrisma().chatMessage.create({
        data: messages[0],
      })
    );
    if (isError(success1)) {
      console.error("Error updating thread: new chat message could not be created");
      throw new Error("Error updating thread: new chat message could not be created");
    }
    if (messages[1]) {
      const success2 = await handlePrisma(() =>
        getPrisma().chatMessage.create({
          data: messages[1],
        })
      );
      if (isError(success2)) {
        console.error("Error updating thread: new chat message could not be created");
        throw new Error("Error updating thread: new chat message could not be created");
      }
    }
  }

  public static async saveDiagramToDB(threadID: string, chatMessageId: number, diagram: diagramResponse) {
    if (!diagram.xml) return;
    const success = await handlePrisma(async () =>
      getPrisma().diagram.create({
        data: {
          chatMessageId: chatMessageId,
          xmlContent: diagram.xml!,
          jsonContent: diagram.json as InputJsonValue ?? undefined,
          previewImageSVG: diagram.xml ? await xmlToSvg(diagram.xml) : null,
        },
      })
    );
    if (isError(success)) {
```

```
      console.error("Error saving diagram");
      throw new Error("Error saving diagram");
    }
  }

  public static async saveDiagramsToDB(threadID: string, chatMessageId: number, diagrams: diagramResponse[]) {
    const diagramDataPromise = diagrams
      .filter(diagram => diagram.xml)
      .map(async diagram => {
        return {
          chatMessageId: chatMessageId,
          xmlContent: diagram.xml!,
          jsonContent: diagram.json as InputJsonValue ?? undefined,
          previewImageSVG: diagram.xml ? await xmlToSvg(diagram.xml) : null
        }
      });
    const diagramData = await Promise.all(diagramDataPromise);
    const success = await handlePrisma(async () =>
      getPrisma().diagram.createMany({
        data: diagramData,
      })
    );
    if (isError(success)) {
      console.error("Error saving diagrams");
      throw new Error("Error saving diagrams");
    }
  }

  public static async saveSamplesToDB(threadID: string, diagrams: diagramResponse[]) {
    const allChats = await this.getAllChatsFromDB(threadID, "desc");
    if (!allChats) return null;
    let chatMessageId = 0;
    for (const chatMessage of allChats) {
      const diagram = await this.getDiagramFromDB(chatMessage)
      if (diagram) {
        chatMessageId = chatMessage.chatMessageId;
        break;
      }
    }
    if (!chatMessageId) return null;
    await this.saveDiagramsToDB(threadID, chatMessageId, diagrams);
  }

  ///////// INSTRUCTION HELPERS //////////

  protected static readInstructionsFile(location: string): string {
    try {
      const file_location = path.join(process.cwd()!, location);
      return fs.readFileSync(file_location, "utf8");
    } catch (error) {
      console.error("Error reading instructions file:", error);
      return "Create a diagram in JSON format for the following scenario: ";
    }
  }

  protected static formatInstructions(format: format) : string[] {
    if (format == "xml") {
      return [
        Ai.readInstructionsFile("data/assistant/knowledge/instructions_xml.txt"),
        Ai.readInstructionsFile("data/assistant/knowledge/example-burger-restaurant.bpmn"),
      ]
    } else {
      return [
        Ai.readInstructionsFile("data/assistant/knowledge/instructions_current_json.txt"),
        Ai.readInstructionsFile("data/assistant/knowledge/example-burger-restaurant.json"),
      ]
    }
  }

  protected static modeInstructions(mode: mode) : string[] {
    if (mode == "quick") {
```

```
      return [
        Ai.readInstructionsFile("data/assistant/knowledge/instructions_quick.txt"),
      ]
    } else if (mode == "detail") {
      return [
        Ai.readInstructionsFile("data/assistant/knowledge/instructions_detail.txt"),
      ]
    } else {
      return []
    }
  }

  protected static async updateInstructions(threadID: string, format: format): Promise<string[]> {
    const getWarnings = async (diagram: string) => {
      const moddle = new BpmnModdle();
      try {
        const xmlModdle = await moddle.fromXML(diagram);
        return xmlModdle.warnings;
      } catch (error) {
        if (error instanceof Error) {
          return [error.message];
        }
        return [];
      }
    }
    const latestDiagram = await this.getLatestDiagramFromDB(threadID);
    if (!latestDiagram || !latestDiagram.xmlContent)
      return [];
    const warnings = await getWarnings(latestDiagram.xmlContent);
    return [`The The following diagram has already been created:
      ${format == "xml" ? latestDiagram.xmlContent : latestDiagram.jsonContent ?? latestDiagram.xmlContent}\n
      ${warnings.length > 0 ? "The following warnings were found in the diagram: " : ""}
      ${warnings.join("\n")}\n
      ${warnings.length > 0 ? "Please fix the warnings while updating the diagram." : ""}
      Please update the diagram, if asked for, for the given prompt.`]
  }

  /////////////// DEBUGGING //////////////

  private static async generateDebugFile(xml: string, threadID: string, additionalDebugData?: any,
                                         model?: string): Promise<void> {
    try {
      const templatePath = path.join(process.cwd()!, "data/assistant/debug/template.html");
      const template = await fs.promises.readFile(templatePath, "utf8");
      const generatedPath = path.join(process.cwd()!, "data/assistant/debug/generated");
      const filename = `debug_${threadID}_${Date.now()}.html`;
      const outputPath = path.join(generatedPath, filename);
      const timingMeasurements = Ai.getMeasurements(threadID);
      additionalDebugData.timings = timingMeasurements;
      const json = !additionalDebugData ? "" : JSON.stringify(additionalDebugData)
        .replaceAll(/\\[nr"]/g, "")
        .replaceAll(/`/g, "");
      const content = template
        .replaceAll("%xml", xml || "")
        .replaceAll("%model", model || "")
        .replaceAll("%info", json || "");
      await fs.promises.mkdir(generatedPath, { recursive: true });
      await fs.promises.writeFile(outputPath, content);
      console.log(`Debug file generated: file:///${outputPath.replace(/\\/g, () => `/`)}`);
    } catch (error) {
      console.error("Error generating debug file:", error);
    }
  }

  private static STOPWATCHES: { [key: string]: number } = {};
  private static MEASUREMENTS : { [key: string]: number } = {};

  private static startStopwatch(label: string, threadID: string = "") {
    Ai.STOPWATCHES[`${threadID}-${label}`] = performance.now();
    // if there are a lot of stopwatches, clean up the old ones
    if (Object.keys(Ai.STOPWATCHES).length > 100) {
```

```
      Ai.cleanupStopwatches();
    }
    // if there are a lot of measurements, clean up the old ones
    if (Object.keys(Ai.MEASUREMENTS).length > 100) {
      Ai.cleanupMeasurements();
    }
  }

  private static startStopwatches(labels: string[], threadID: string = "") {
    labels.forEach(label => Ai.startStopwatch(label, threadID));
  }

  private static endStopwatch(label: string, threadID: string = "", log: boolean = false): number {
    if (!(`${threadID}-${label}` in Ai.STOPWATCHES))
      return -1;
    const duration = performance.now() - Ai.STOPWATCHES[`${threadID}-${label}`];
    delete Ai.STOPWATCHES[`${threadID}-${label}`];
    const labelNumberMatch = label.match(/[0-9]+$/);
    if (`${threadID}-${label}` in Ai.MEASUREMENTS)
      label = labelNumberMatch ? `${label}-${labelNumberMatch[0] + 1}` : `${label}-2`;
    Ai.MEASUREMENTS[`${threadID}-${label}`] = duration;
    if (log)
      console.log(`${threadID} | ${label}: ${duration} ms`);
    return duration;
  }

  private static endStopwatches(labels: string[], threadID: string = "", log?: boolean): number[] {
    return labels.map(label => Ai.endStopwatch(label, threadID, log));
  }

  private static isStopwatchRunning(label: string, threadID: string = ""): boolean {
    return label in Ai.STOPWATCHES;
  }

  private static getMeasurements(threadID: string): { [key: string]: number } {
    const measurements: { [key: string]: number } = {};
    for (const [key, value] of Object.entries(Ai.MEASUREMENTS)) {
      if (key.startsWith(threadID)) {
        measurements[key.replace(`${threadID}-`, "")] = value;
        delete Ai.STOPWATCHES[key];
      }
    }
    return measurements;
  }

  private static cleanupMeasurements(){
    const now = performance.now();
    for (const [key, value] of Object.entries(Ai.MEASUREMENTS)) {
      if (now - value > 600000) { // 10 minutes
        delete Ai.MEASUREMENTS[key];
      }
    }
  }

  private static cleanupStopwatches(){
    const now = performance.now();
    for (const [key, value] of Object.entries(Ai.STOPWATCHES)) {
      if (now - value > 600000) { // 10 minutes
        delete Ai.STOPWATCHES[key];
      }
    }
  }

  //////////// PARSING OUTPUT ////////////

  protected static convertResponseToDiagramOutput(response: string, format: format,
                                                  mode: mode): diagramResponse | null {
    if (format == "xml") {
      const xml = this.convertStringToXml(response);
      const text = this.convertStringToTextPart(response, format);
      if (mode === "quick" || text === "")
```

```
        return {xml: xml};
      return {xml: xml, text: text};
    } else if (format == "json") {
      const json = this.convertStringToJson(response);
      //console.debug(JSON.stringify(json, null, 4))
      if (!json) return {text: response};
      const xml = convertJsonToXml(json);
      const text = this.convertStringToTextPart(response, format);
      if (!xml) throw new Error("Conversion from json to xml failed");
      if (mode === "quick" || text === "")
        return {xml: xml, json: json};
      return {xml: xml, json: json, text: text};
    }
    return null;
  }

  protected static convertStringToJson(input: string, promptComplete: boolean = true): DiagramJson | undefined {
    try {
      if (promptComplete)
        return input.match(/{[^`]*}(?=\s*``?`?|\s*[^{}[\]\s,]|,?\s*\w|$)\n?/g)?.
          sort((a, b) => a.length - b.length).
          map(match => JSON.parse(match))[0];
      return input.match(/{[^`]*}(?=\s*``?`?|\s*[^{}[\]\s,]|,?\s*\w)\n?/g)?.
        sort((a, b) => a.length - b.length).
        map(match => JSON.parse(match))[0];
    } catch {
      return undefined;
    }
  }

  protected static convertStringToJsons(input: string, promptComplete: boolean = true): DiagramJson[] | undefined {
    const MIN_DIAGRAM_LENGTH = 100;
    try {
      if (promptComplete)
        return input.match(/{[^`]*}(?=\s*``?`?|\s*[^{}[\]\s,]|,?\s*\w|$)\n?/g)?.
          filter(match => match.length >= MIN_DIAGRAM_LENGTH).
          map(match => JSON.parse(match));
      return input.match(/{[^`]*}(?=\s*``?`?|\s*[^{}[\]\s,]|,?\s*\w)\n?/g)?.
        filter(match => match.length >= MIN_DIAGRAM_LENGTH).
        map(match => JSON.parse(match));
    } catch {
      return undefined;
    }
  }

  protected static convertStringToXml(input: string, promptComplete: boolean = true): string | undefined {
    try {
      if (promptComplete)
        return input.match(/(?:<[^<>]*\/[^<>]*>\s*|<[^\/<>]*>[^`]*<\/[^<>]*>)+\s*(?=``?`?|[^<>`\s]|$)\n?/g)?.
          sort((a, b) => a.length - b.length)[0];
      return input.match(/(?:<[^<>]*\/[^<>]*>\s*|<[^\/<>]*>[^`]*<\/[^<>]*>)+\s*(?=``?`?|[^<>`\s])\n?/g)?.
        sort((a, b) => a.length - b.length)[0];
    } catch {
      return undefined;
    }
  }

  protected static convertStringToXmls(input: string, promptComplete: boolean = true): string[] | undefined {
    const MIN_DIAGRAM_LENGTH = 100;
    try {
      if (promptComplete)
        return input.match(/(?:<[^<>]*\/[^<>]*>\s*|<[^\/<>]*>[^`]*<\/[^<>]*>)+\s*(?=``?`?|[^<>`\s]|$)\n?/g)?.
          filter(match => match.length >= MIN_DIAGRAM_LENGTH);
      return input.match(/(?:<[^<>]*\/[^<>]*>\s*|<[^\/<>]*>[^`]*<\/[^<>]*>)+\s*(?=``?`?|[^<>`\s])\n?/g)?.
        filter(match => match.length >= MIN_DIAGRAM_LENGTH);
    } catch {
      return undefined;
    }
  }

  protected static convertStringToTextPart(input: string, format: format): string | undefined {
```

```
    let strict = input;
    const MIN_DIAGRAM_LENGTH = 100;
    if (format == "xml") {
      try {
        // remove complete and large diagrams
        input.match(
/(?:``?`?\s*(?:xml)?\s*)?(?:<[^<>]*\/[^<>]*>\s*|<[^\/<>]*>[^`]*<\/[^<>]*>)+\s*(?:``?`?|(?=[^<>`\s]))\n?/g
        )?.forEach(match => {
          strict = input.replace(match, "");
          if (match.length < MIN_DIAGRAM_LENGTH) return;
          input = input.replace(match, "");
        });
        // removing the incomplete diagram at the end
        strict.match(
/(?:``?`?\s*(?:xm?l?)?\s*)?(?:\s*(?:<[^<>`]*>?\s*|<[^\/<>`]*?>?[^<>`]*?<?\/?[^<>`]*?>?)*?)*(?:``?`?)?$/
        )?.forEach(match => {
          if (!match || !match.trim()) return;
          input = input.replace(match, "");
        });
        return input.trim().replace("\u0004", "");
      } catch {
        return "";
      }
    } else if (format == "json") {
      try {
        let bracketCounter = 0;
        let tickCounter = 0;
        let buffer = "";
        let textBuffer = "";
        let diagram = "";
        for (let char of input) {
          if (char !== "\u0004") buffer += char;
          if (char === "{") bracketCounter++;
          if (char === "}") bracketCounter--;
          if (char === "`") tickCounter++;
          if (bracketCounter == 0 && tickCounter % 2 == 0 && char !== "`" && char !== "}") {
            if (diagram){
              // diagram finished
              if (diagram.length < MIN_DIAGRAM_LENGTH) {
                textBuffer += diagram;
              }
              diagram = "";
            }
            if (char !== "\u0004") textBuffer += char;
          } else {
            diagram += char
          }
        }
        return textBuffer;
      } catch {
        return "";
      }
    }
  }

  protected static convertStringToStreamData(input: string, format: format): {
    currentlyText: boolean;
    currentlyDiagram: boolean;
    currentlyLargeDiagram: boolean;
    currentlySmallDiagram: boolean;
    numLargeDiagrams: number;
    numSmallDiagrams: number
    largeDiagrams: string[];
    smallDiagrams: string[];
    currentDiagram: string;
    text: string;
  } | undefined {
    let strict = input;
    let currentlyText = true;
    let currentlyDiagram = false;
    let currentlyLargeDiagram = false;
```

```
    let currentlySmallDiagram = false;
    let numLargeDiagrams = 0;
    let numSmallDiagrams = 0;
    let largeDiagrams: string[] = [];
    let smallDiagrams: string[]  = [];
    let currentDiagram = "";
    let text = input;

    const MIN_DIAGRAM_LENGTH = 100;
    if (format == "xml") {
      try {
        // remove complete and large diagrams
        text.match(
/(?:``?`?\s*(?:xml)?\s*)?(?:<[^<>]*\/[^<>]*>\s*|<[^\/<>]*>[^`]*<\/[^<>]*>)+\s*(?:``?`?|(?=[^<>`\s]))\n?/g
        )?.forEach(match => {
          strict = text.replace(match, "");
          if (match.length < MIN_DIAGRAM_LENGTH) {
            numSmallDiagrams++;
            const diagram = match.match(
/(?:<[^<>]*\/[^<>]*>\s*|<[^\/<>]*>[^`]*<\/[^<>]*>)+\s*(?=``?`?|[^<>`\s]|$)\n?/
            )?.at(0);
            if (diagram) smallDiagrams.push(diagram);
          }
          else {
            numLargeDiagrams++;
            text = text.replace(match, "");
            const diagram = match.match(
/(?:<[^<>]*\/[^<>]*>\s*|<[^\/<>]*>[^`]*<\/[^<>]*>)+\s*(?=``?`?|[^<>`\s]|$)\n?/
            )?.at(0);
            if (diagram) largeDiagrams.push(diagram);
          }
        });
        // removing the incomplete diagram at the end
        strict.match(
/(?:``?`?\s*(?:xm?l?)?\s*)?(?:\s*(?:<[^<>`]*>?\s*|<[^\/<>`]*?>?[^<>`]*?<?\/?[^<>`]*?>?)*?)*(?:``?`?)?$/
        )?.forEach(match => {
          if (!match || !match.trim()) return;
          currentlyDiagram = true;
          currentlyText = false;
          if (match.length < MIN_DIAGRAM_LENGTH) currentlySmallDiagram = true;
          else currentlyLargeDiagram = true;
          currentDiagram = match;
          text = text.replace(match, "");
        });
        text = text.replace("\u0004", "")
        return {currentlyText, currentlyDiagram, currentlyLargeDiagram, currentlySmallDiagram,
                numLargeDiagrams, numSmallDiagrams, largeDiagrams, smallDiagrams, currentDiagram, text};
      } catch {
        return undefined;
      }
    } else if (format == "json") {
      try {
        // old: (?:``?`?\s*(?:json)?\s*)?{(?:[{[\]},\s\d]|"[^{[\]}]*":?)*}(?=\s*``?`?|\s*[^{}[\]\s,]|,?\s*\w)\n?
        let bracketCounter = 0;
        let tickCounter = 0;
        let buffer = "";
        let textBuffer = "";
        let diagram = "";
        let diagrams = [];
        for (let char of input) {
          buffer += char;
          if (char === "{") bracketCounter++;
          if (char === "}") bracketCounter--;
          if (char === "`") tickCounter++;
          if (bracketCounter == 0 && tickCounter % 2 == 0 && char !== "`" && char !== "}") {
            if (diagram){
              // diagram finished
              diagrams.push(diagram);
              if (diagram.length < MIN_DIAGRAM_LENGTH) {
                textBuffer += diagram;
              }
```

```
              diagram = "";
            }
            textBuffer += char;
          } else {
            diagram += char
          }

        }
        text = textBuffer.replace("\u0004", "");
        currentlyDiagram = !!diagram;
        currentlyText = !currentlyDiagram;
        if (diagram.length < MIN_DIAGRAM_LENGTH) currentlySmallDiagram = currentlyDiagram;
        else currentlyLargeDiagram = currentlyDiagram;
        currentDiagram = diagram;
        diagrams.forEach(match => {
          const diagram = match.match(
/{(?:[{[\]},\s\d]|"[^{[\]}]*":?)*}(?=\s*``?`?|\s*[^{}[\]\s,]|,?\s*\w|$)\n?/
          )?.at(0) ?? "";
          if (!diagram) return;
          if (diagram.length < MIN_DIAGRAM_LENGTH) {
            numSmallDiagrams++;
            smallDiagrams.push(diagram);
          }
          else {
            numLargeDiagrams++;
            largeDiagrams.push(diagram);
          }
        });
        return {currentlyText, currentlyDiagram, currentlyLargeDiagram, currentlySmallDiagram,
                numLargeDiagrams, numSmallDiagrams, largeDiagrams, smallDiagrams, currentDiagram, text};
      } catch {
        return undefined;
      }
    }
  }

  public static diagramOutputToStringVersion(diagramOutput: diagramResponse): diagramResponse {
    const diagramOutputStringVersion = diagramOutput;
    diagramOutputStringVersion.json = diagramOutput.json ? JSON.stringify(diagramOutput.json) : undefined;
    return diagramOutputStringVersion;
  }
}
```

B Anhänge

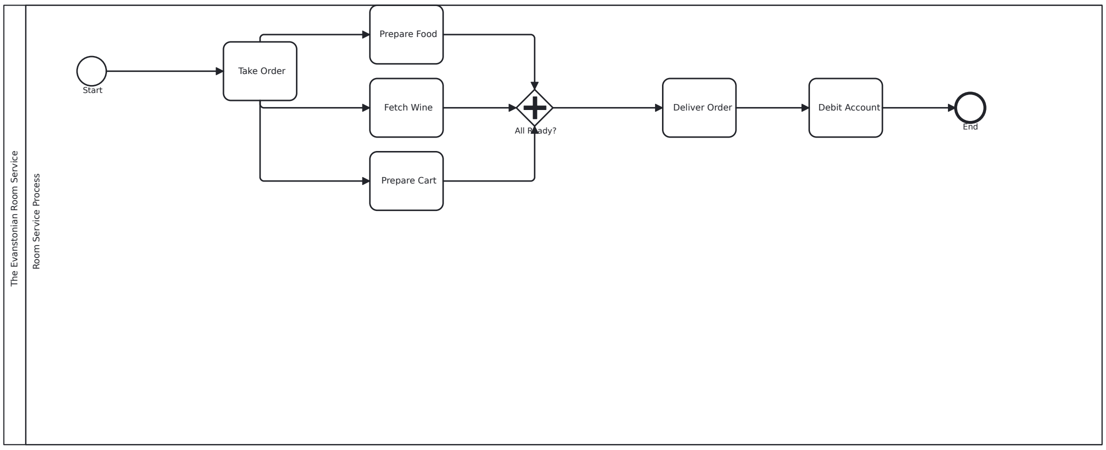

Abbildung B.1: Diagramm mit dem ChatGPT Assistant

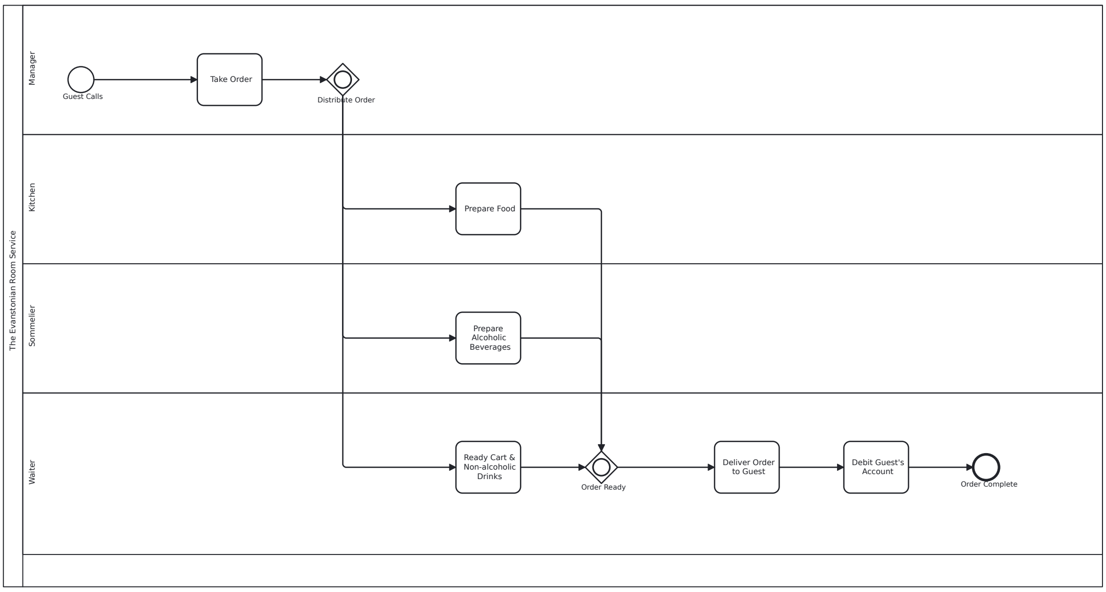

Abbildung B.2: Diagramm mit der Responses API und JSON Format

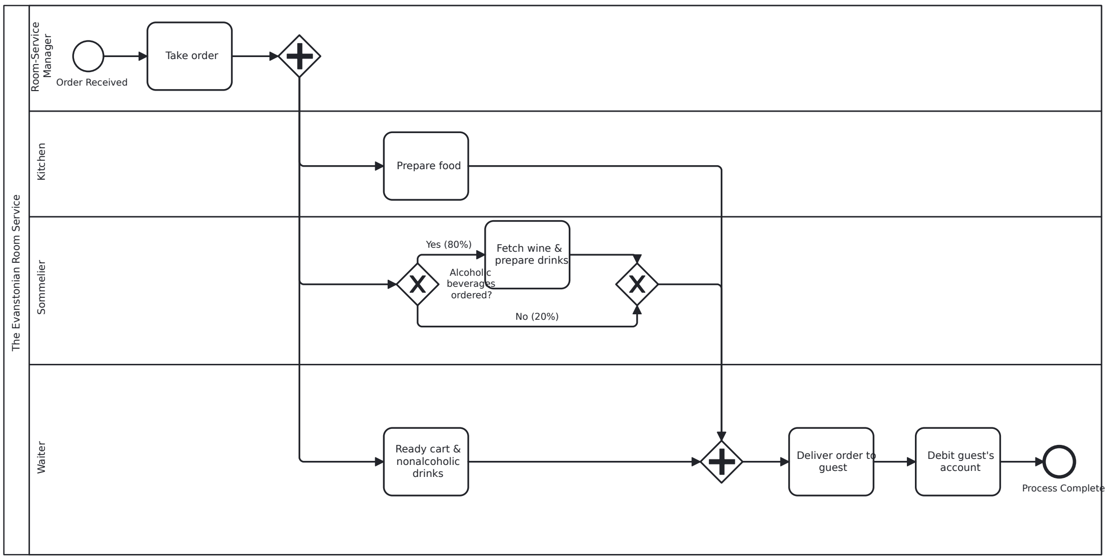

Abbildung B.3: Diagramm mit der Responses API und XML Format

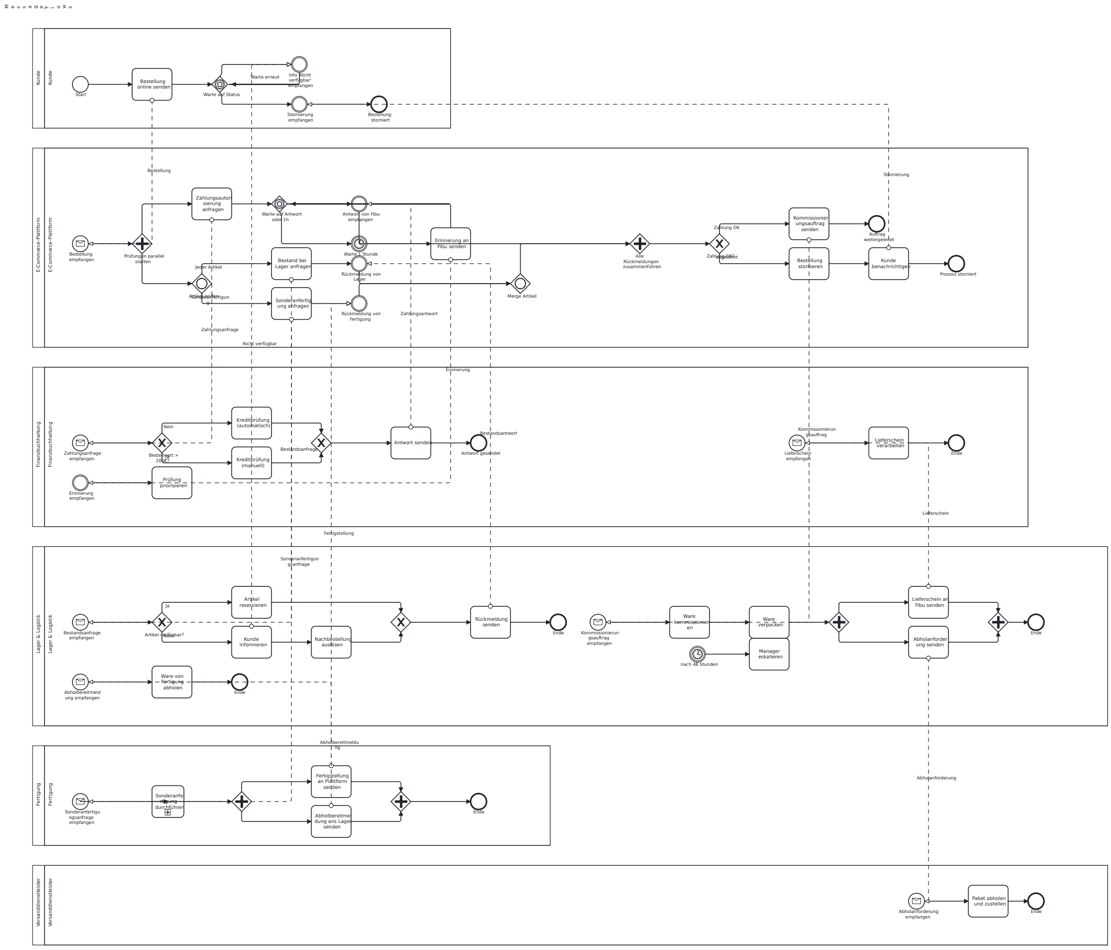

Abbildung B.4: Diagramm von Gemini 2.5 Pro mit JSON
11904 TOKEN | 0.000 $ | 134 s

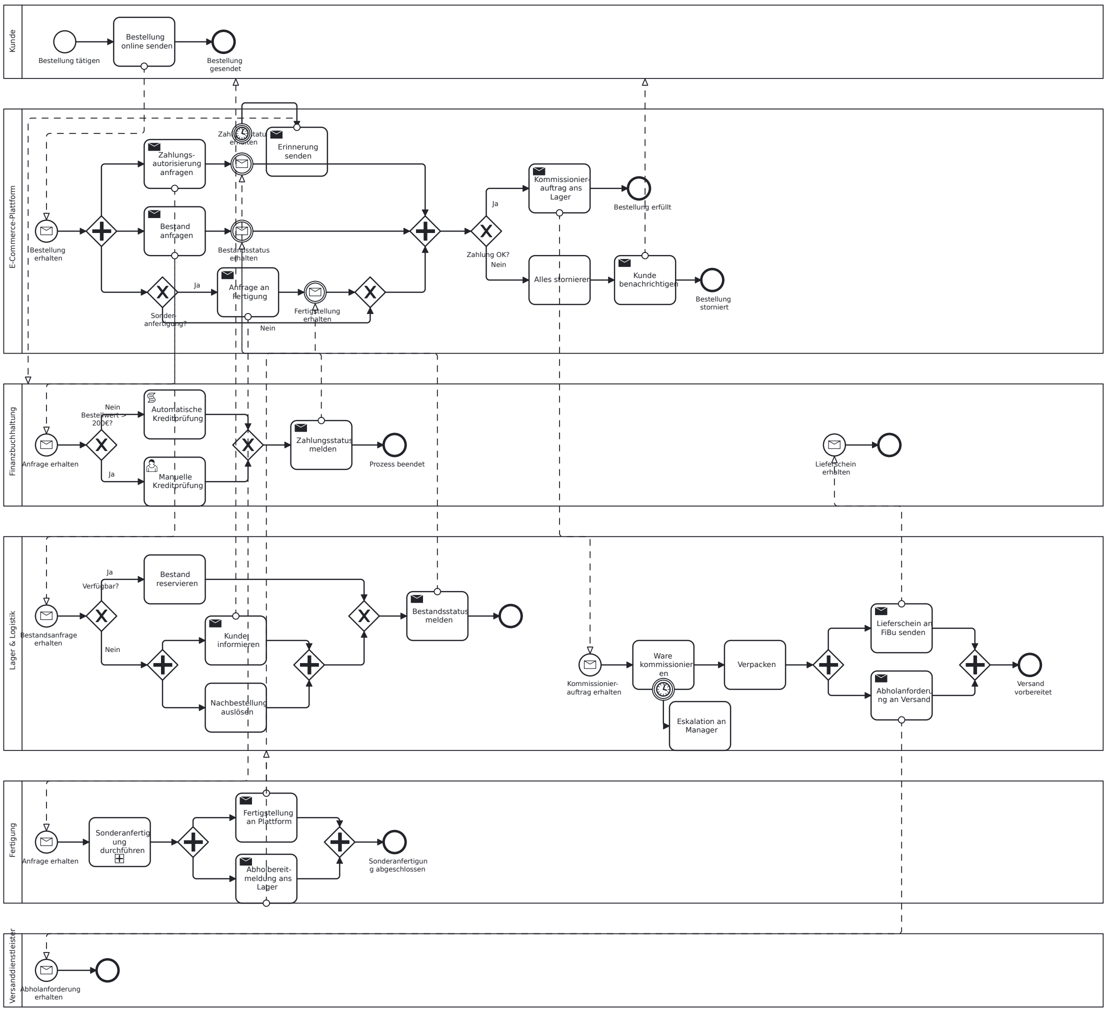

Abbildung B.5: Diagramm von Gemini 2.5 Pro mit XML
21183 TOKEN | 0.000 $ | 180 s

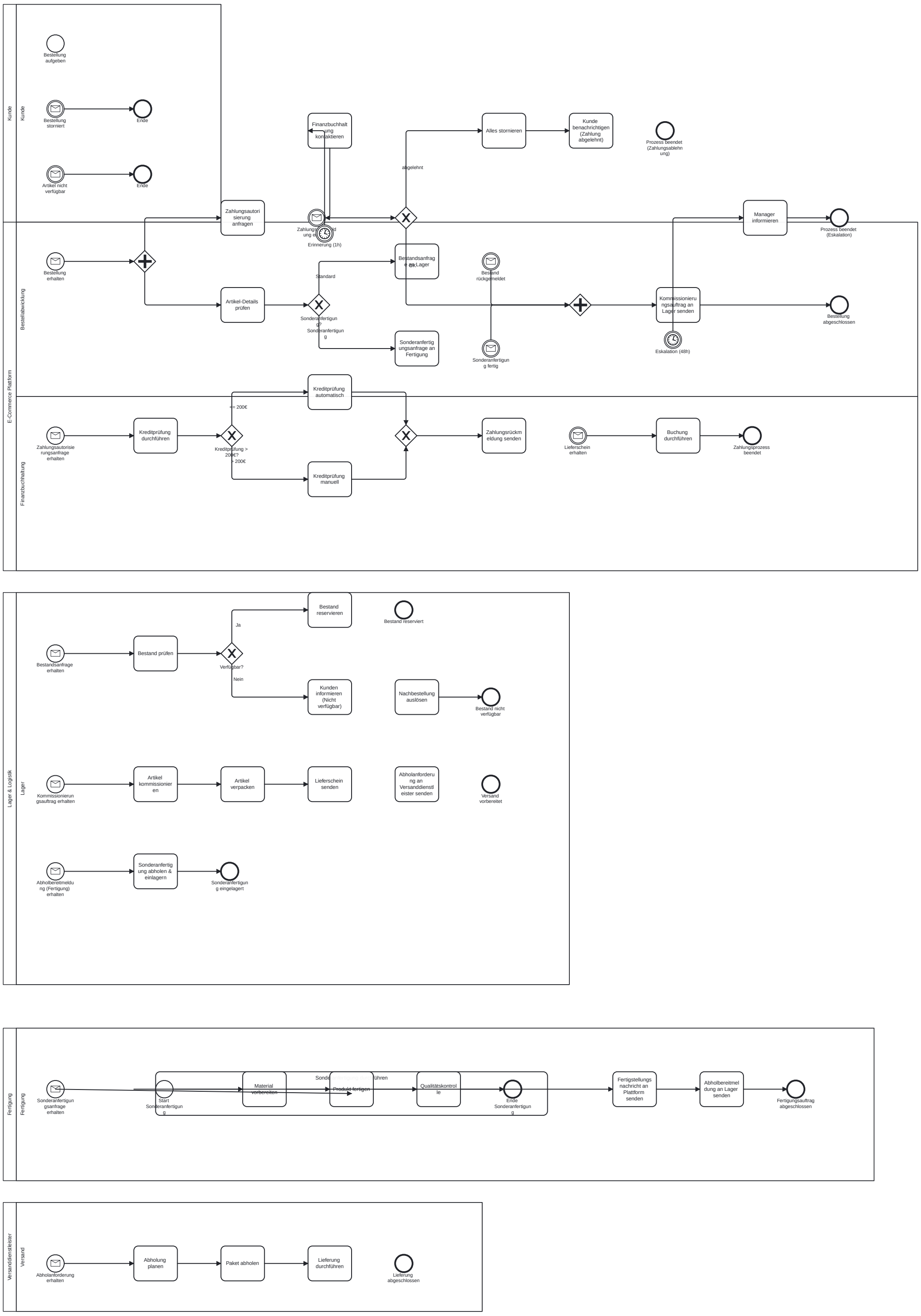

Abbildung B.6: Diagramm von Gemini 2.5 Flash mit JSON
11814 TOKEN | 0.000 $ | 104 s

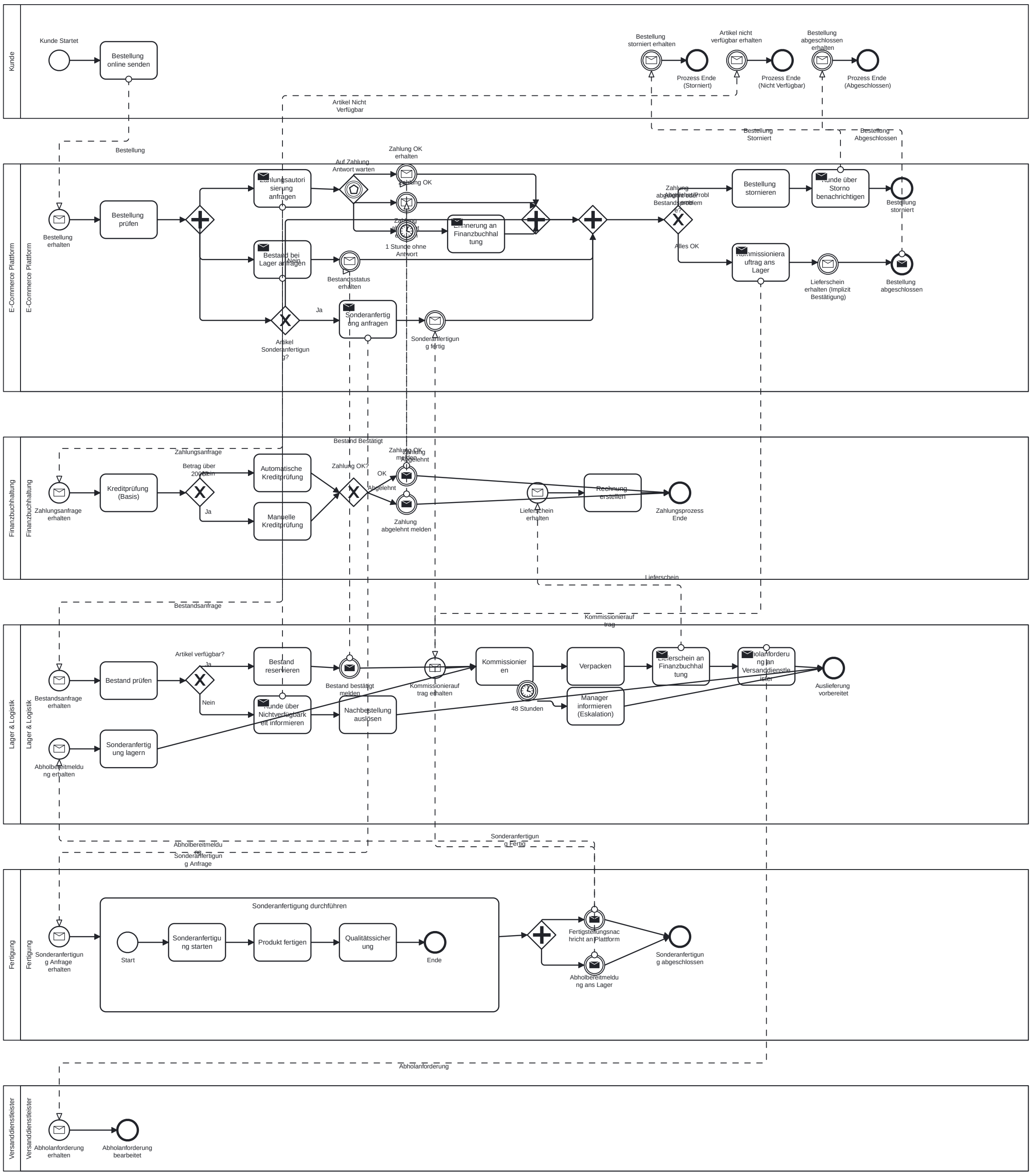

Abbildung B.7: Diagramm von Gemini 2.5 Flash mit XML
32371 TOKEN | 0.000 $ | 174 s

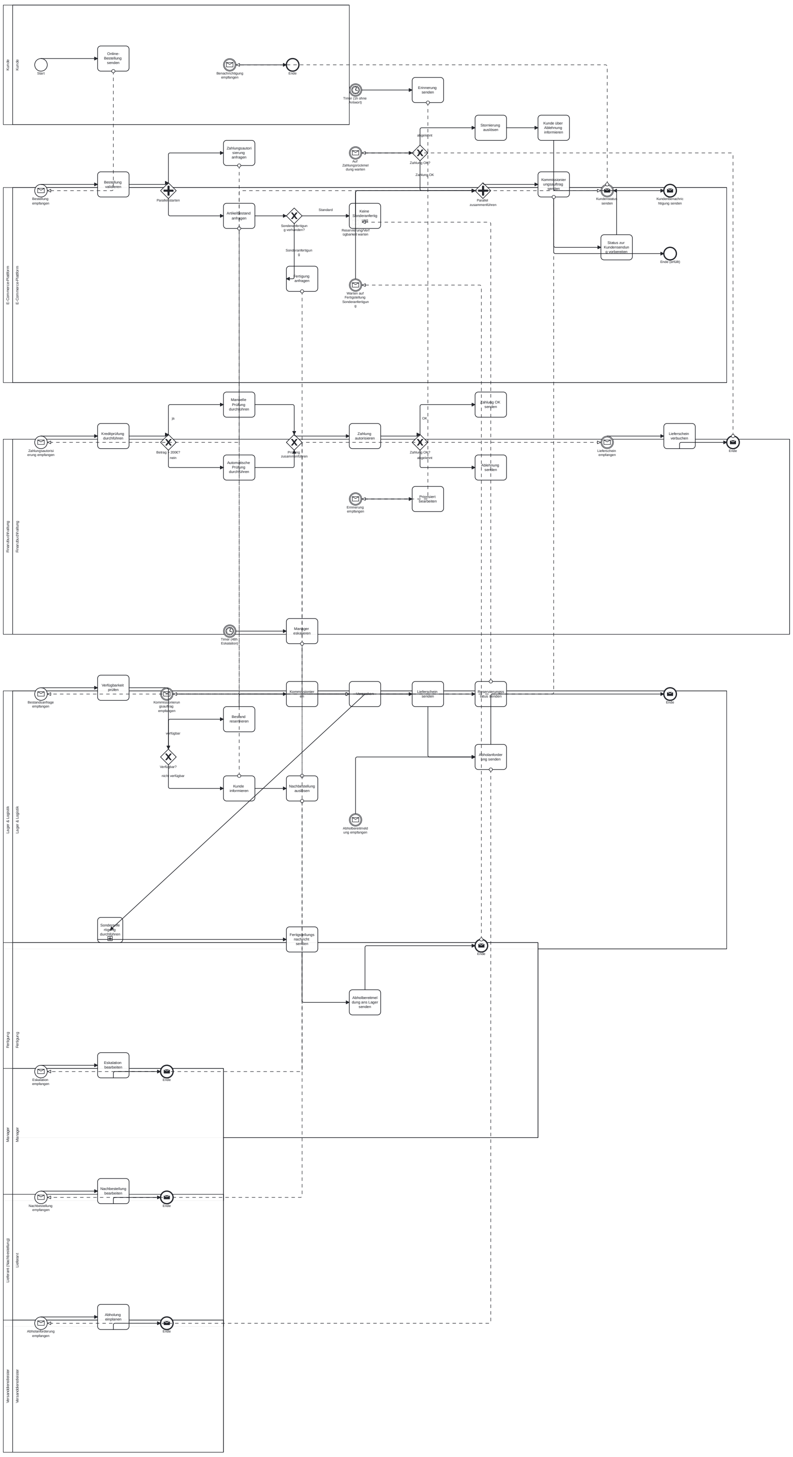

Abbildung B.8: Diagramm von ChatGPT 5.2 mit JSON
11391 TOKEN | 0.181 $ | 253 s

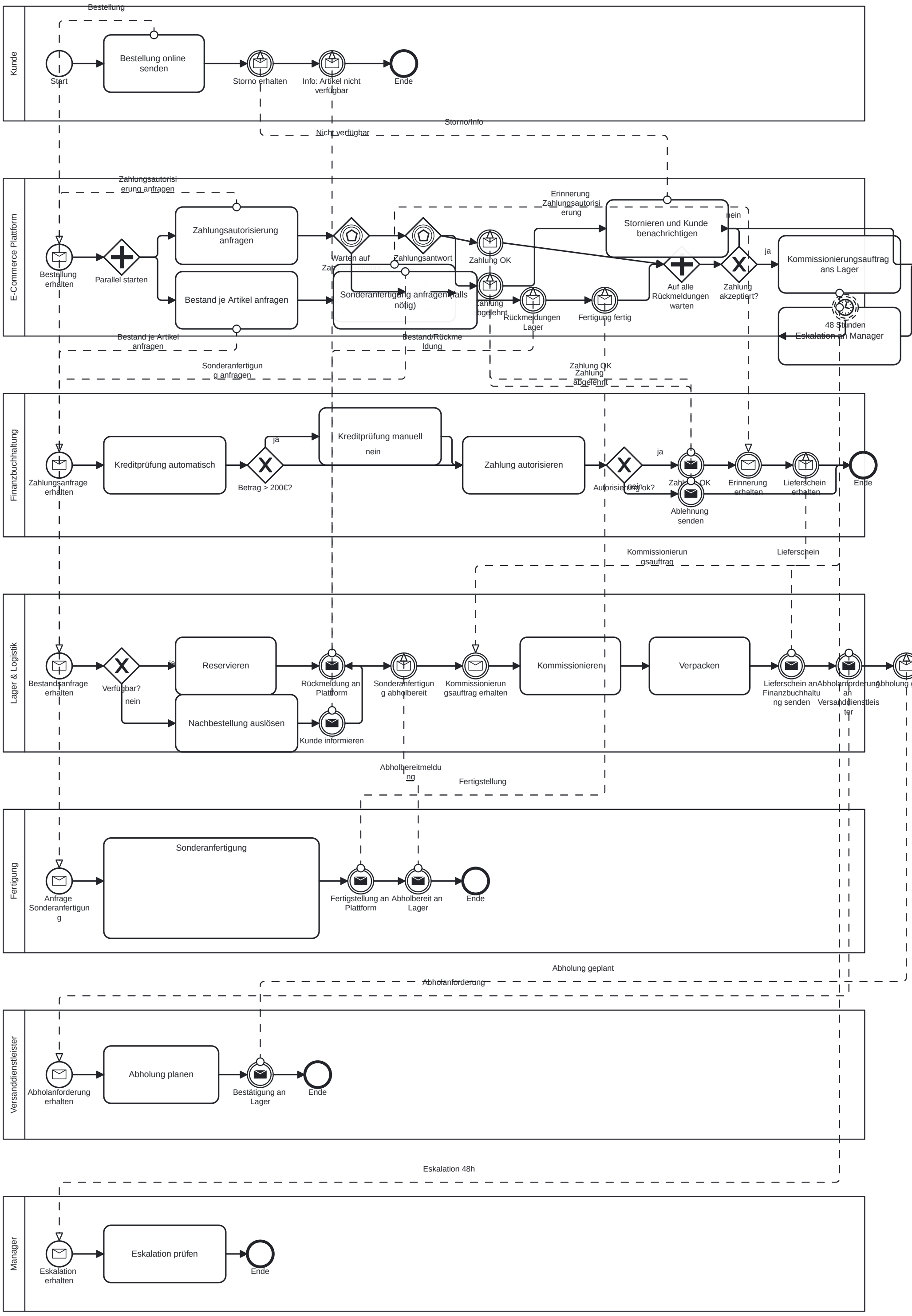

Abbildung B.9: Diagramm von ChatGPT 5.2 mit XML
18340 TOKEN | 0.282 $ | 607 s

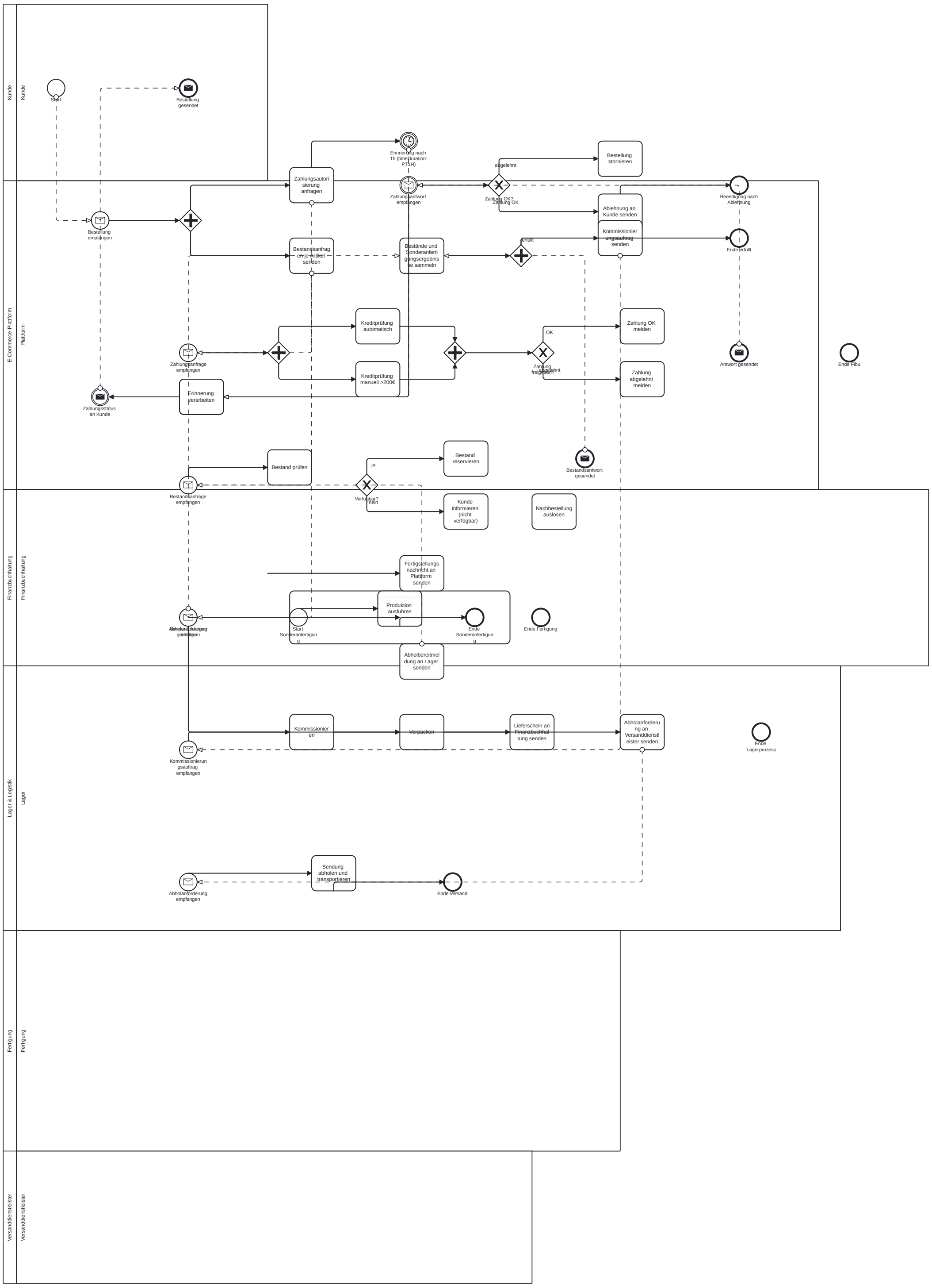

Abbildung B.10: Diagramm von ChatGPT 5.1 mit JSON
7031 TOKEN | 0.086 $ | 34 s

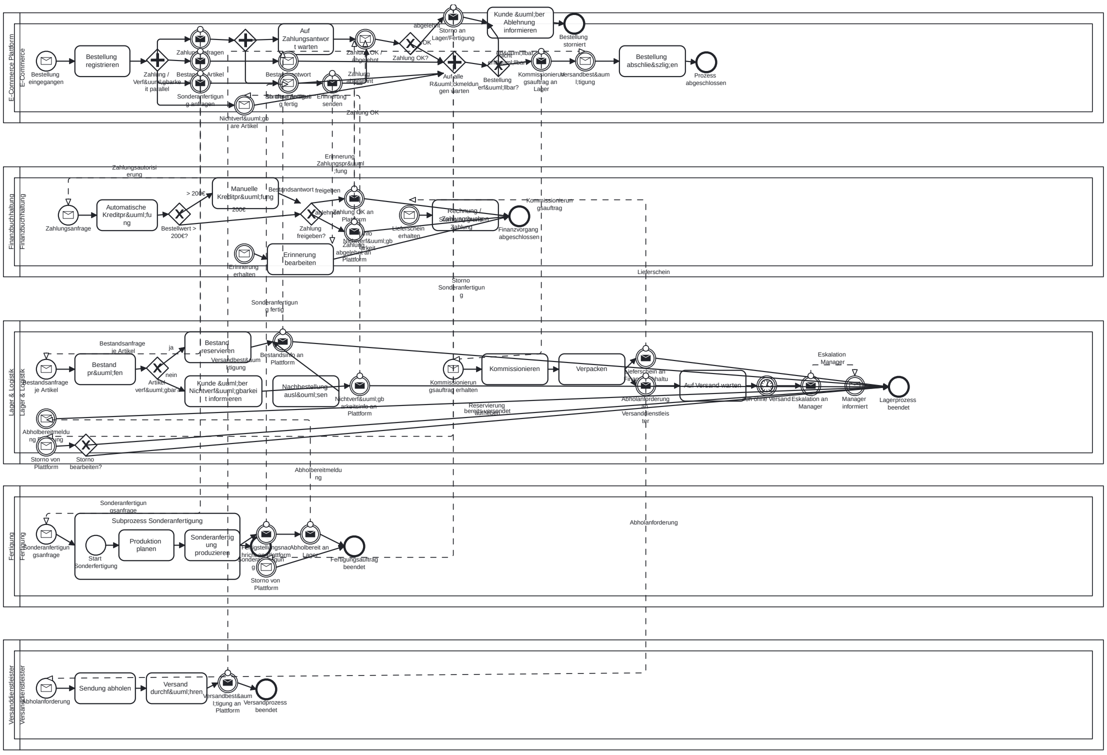

Abbildung B.11: Diagramm von ChatGPT 5.1 mit XML
21697 TOKEN | 0.234 $ | 105 s

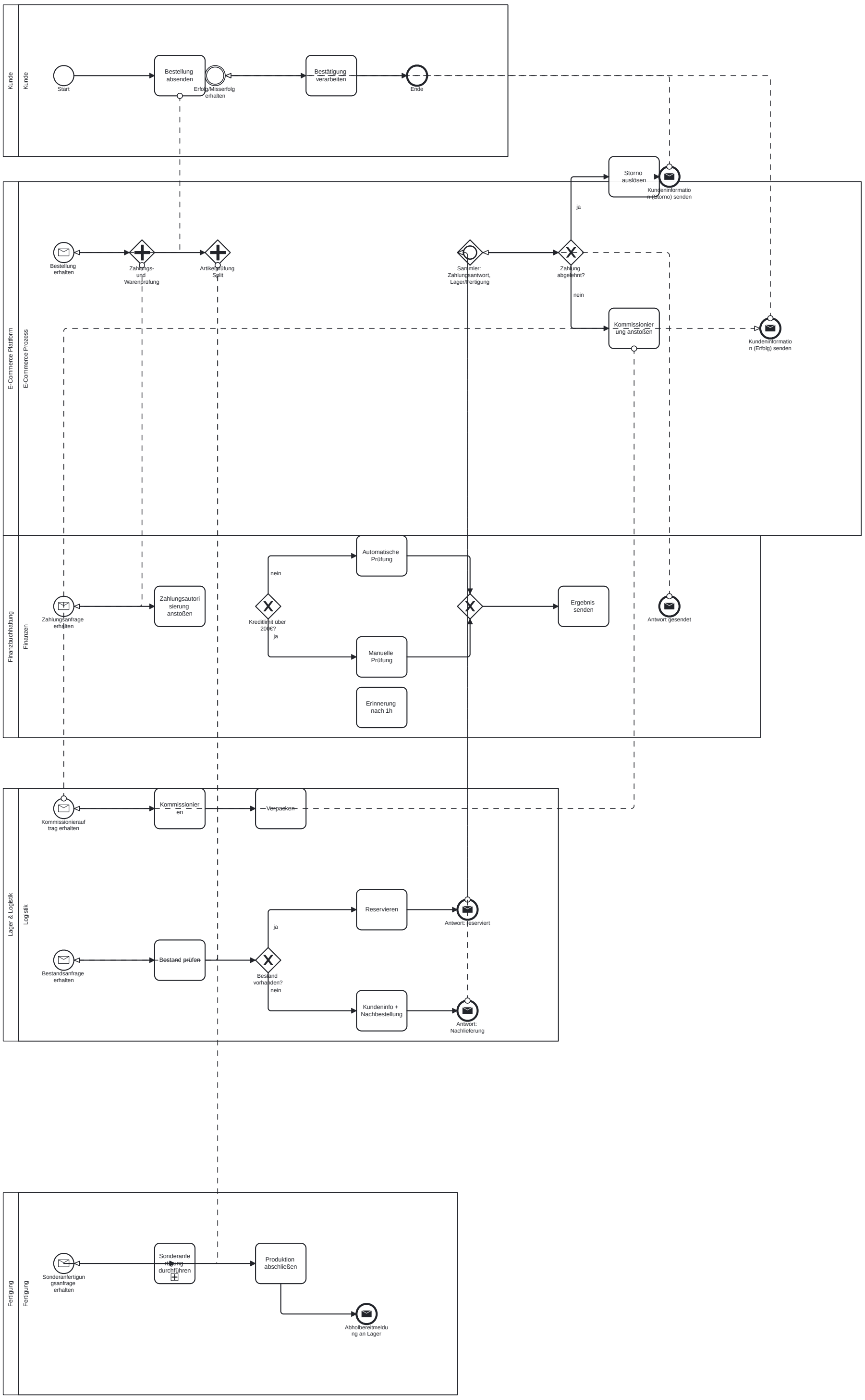

Abbildung B.12: Diagramm von ChatGPT 4.1 mit JSON
5793 TOKEN | 0.077 $ | 69 s

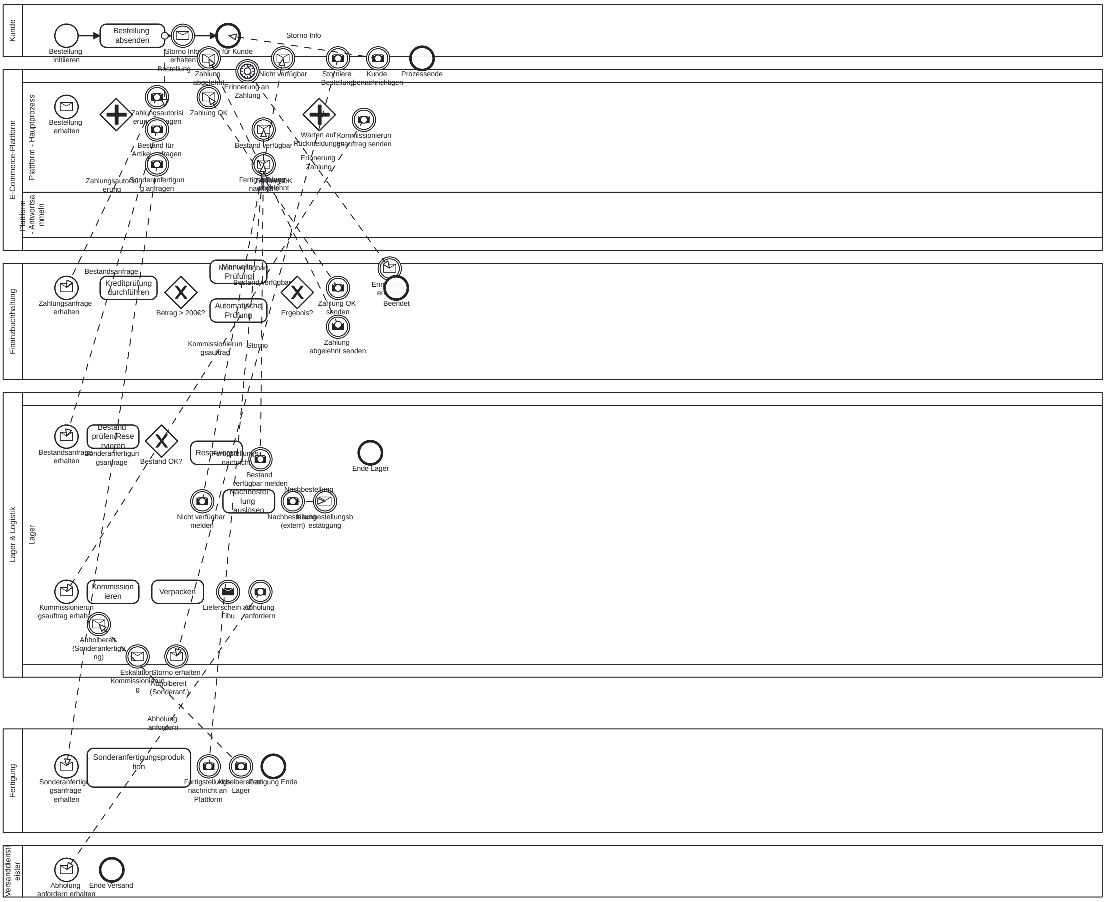

Abbildung B.13: Diagramm von ChatGPT 4.1 mit XML
12944 TOKEN | 0.130 $ | 168 s

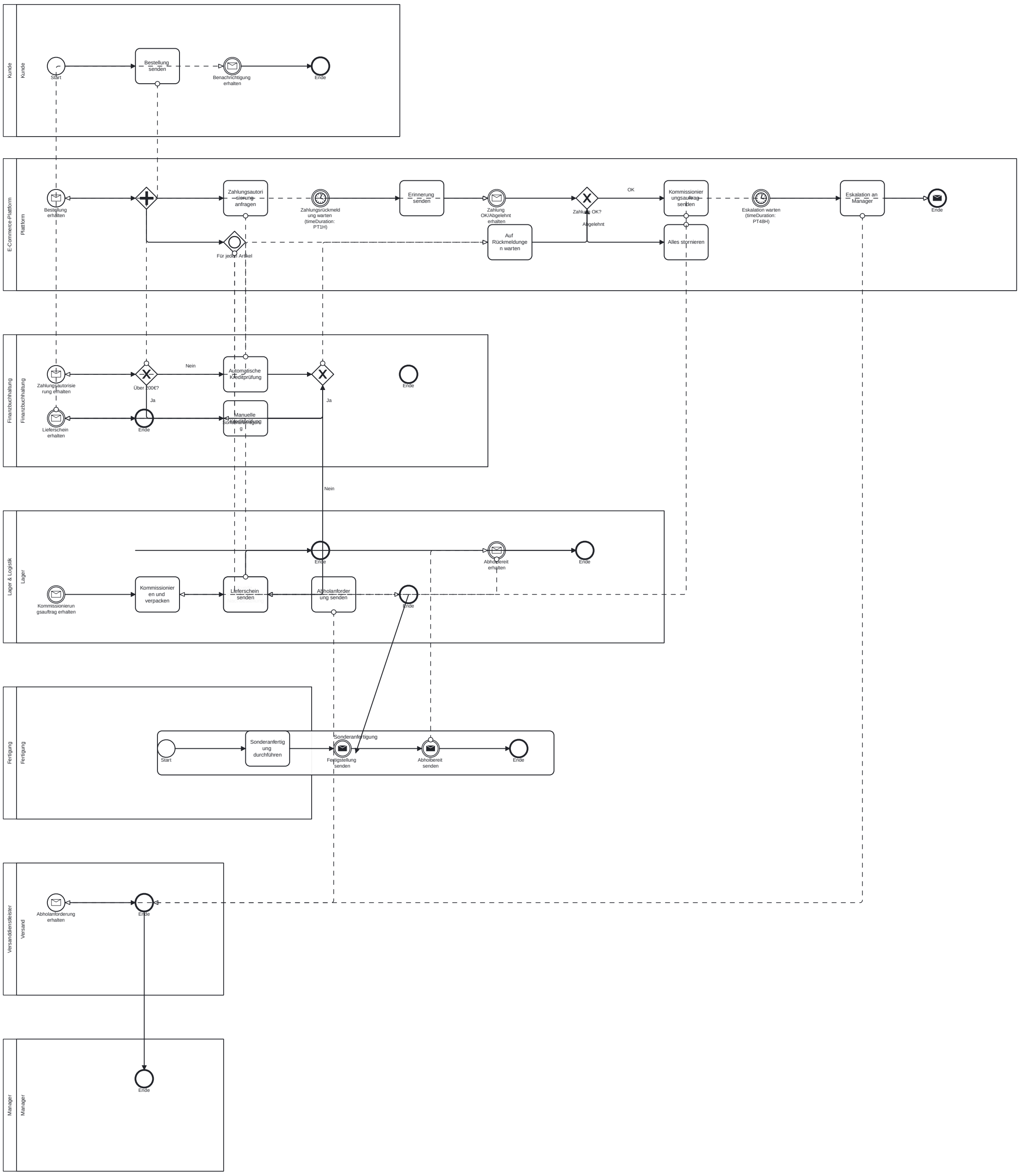

Abbildung B.14: Diagramm von Grok 4 mit JSON
7263 TOKEN | 0.161 $ | 248 s

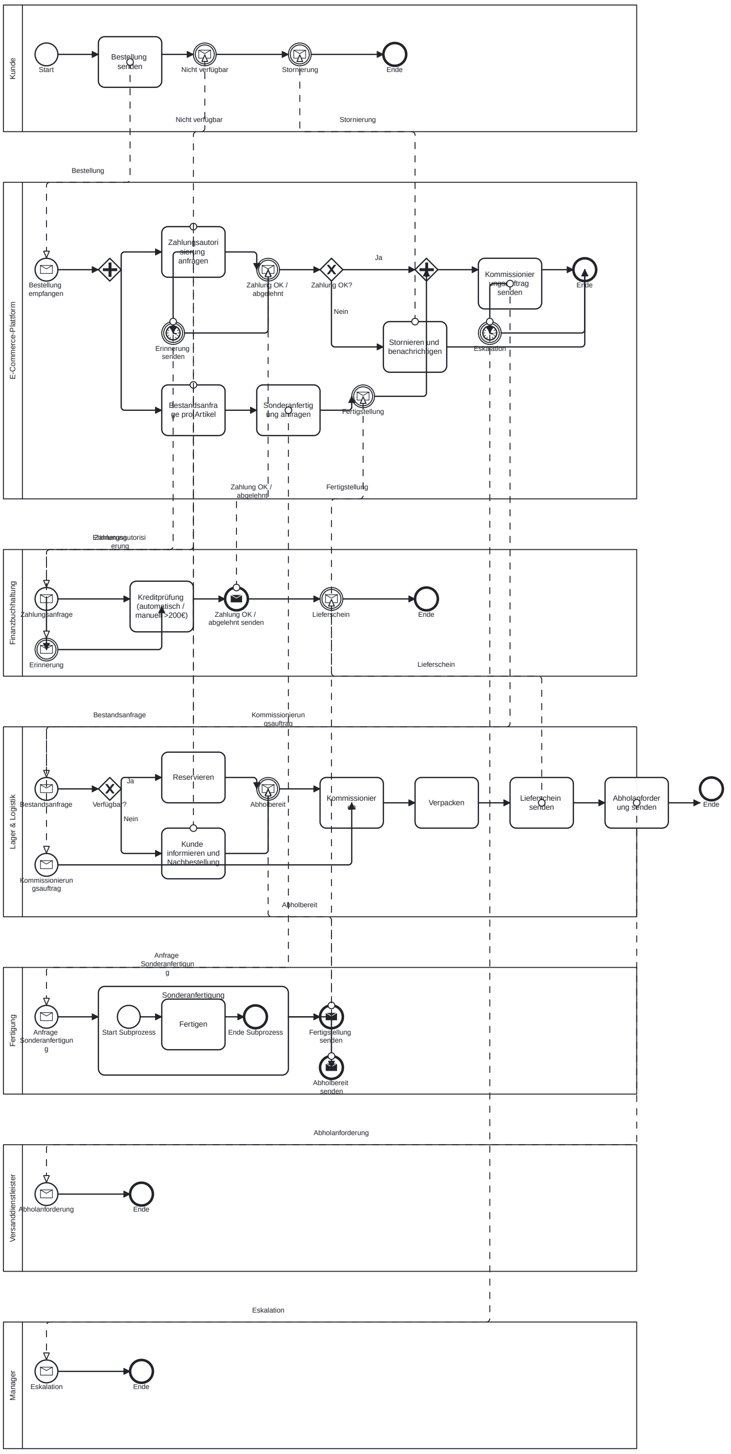

Abbildung B.15: Diagramm von Grok 4 mit XML
13268 TOKEN | 0.250 $ | 604 s

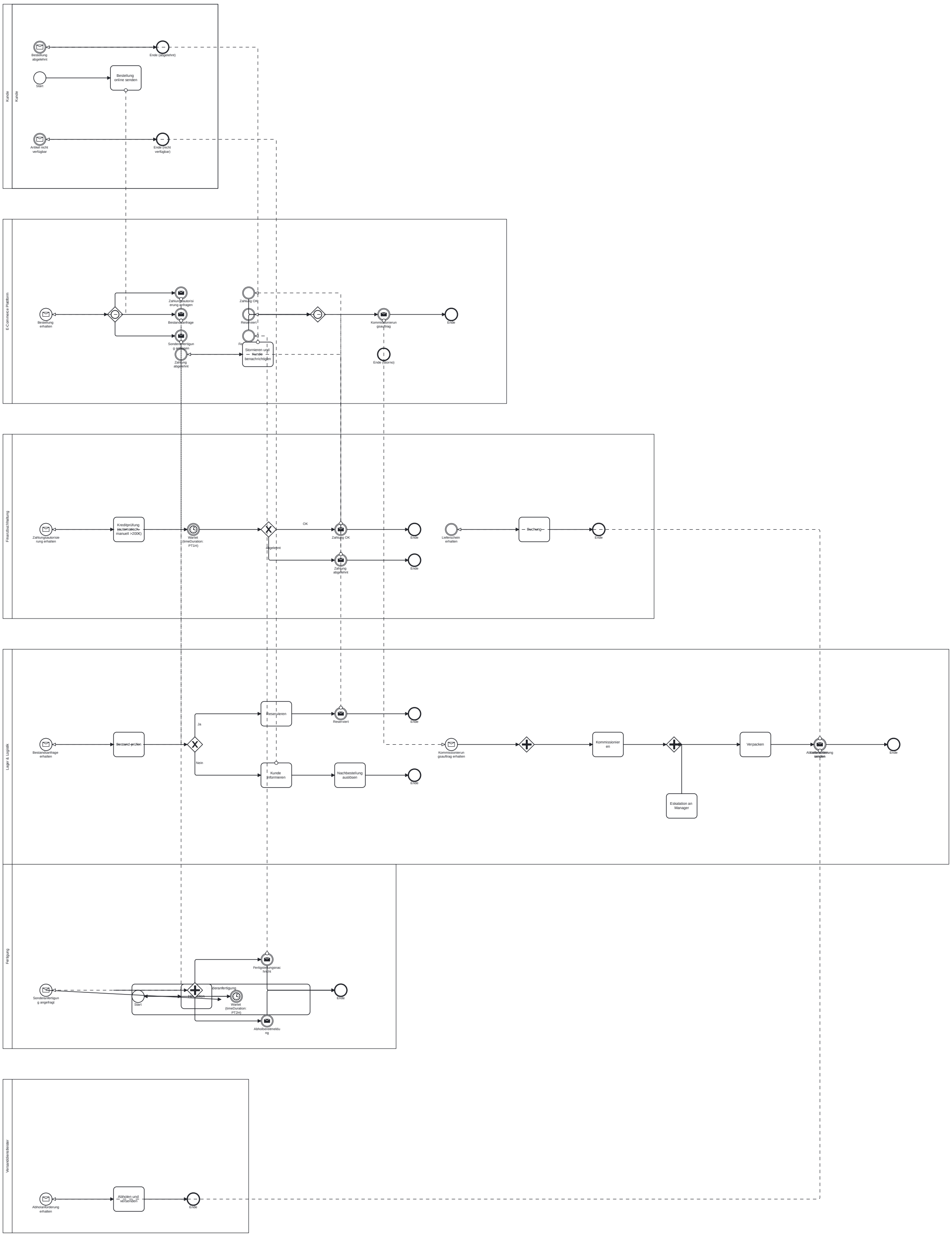

Abbildung B.16: Diagramm von Grok 4.1 Fast mit JSON
9640 TOKEN | 0.015 $ | 191 s

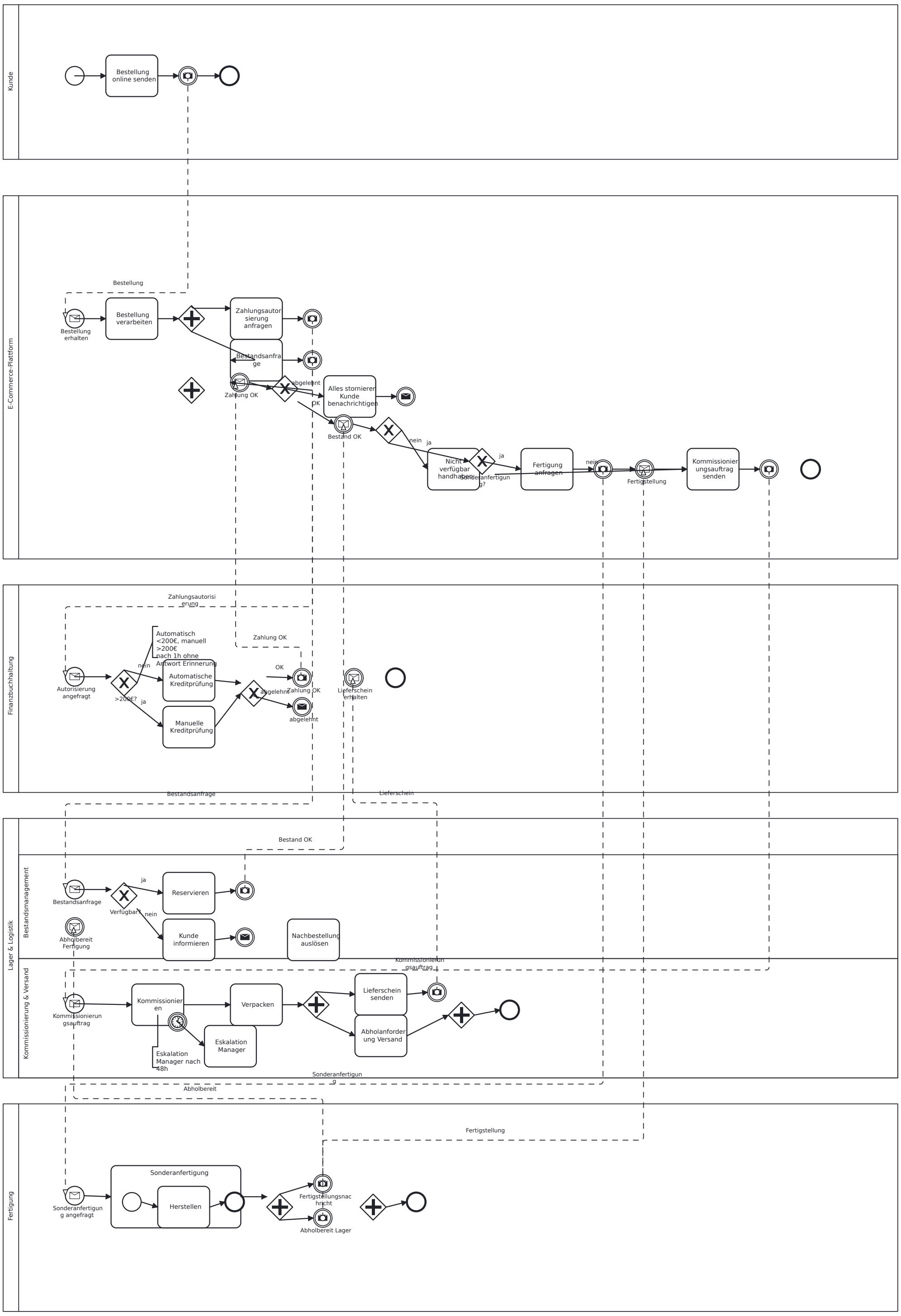

Abbildung B.17: Diagramm von Grok 4.1 Fast mit XML
16555 TOKEN | 0.016 $ | 200 s

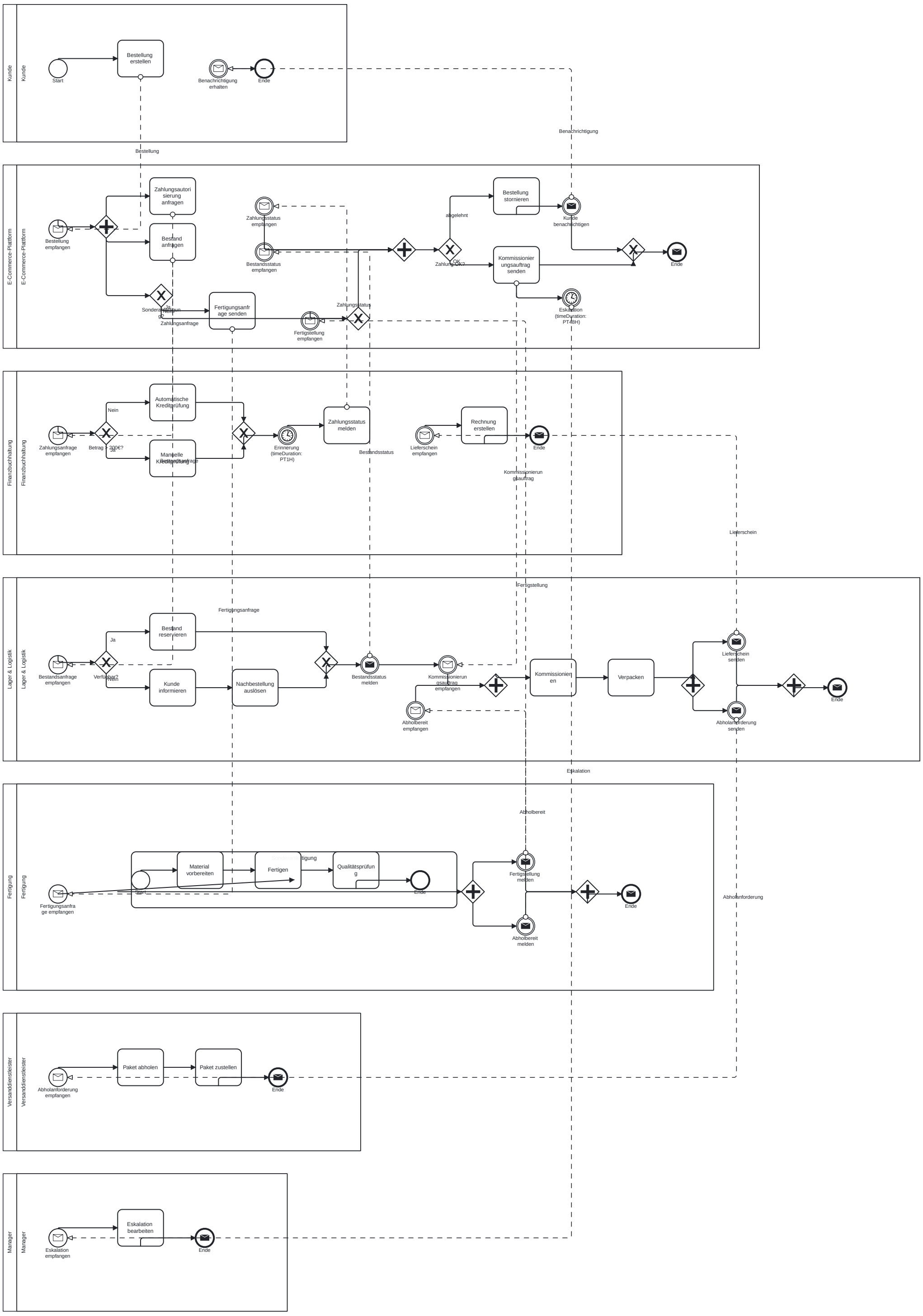

Abbildung B.18: Diagramm von Claude Opus 4.5 mit JSON
13826 TOKEN | 0.421 $ | 118 s

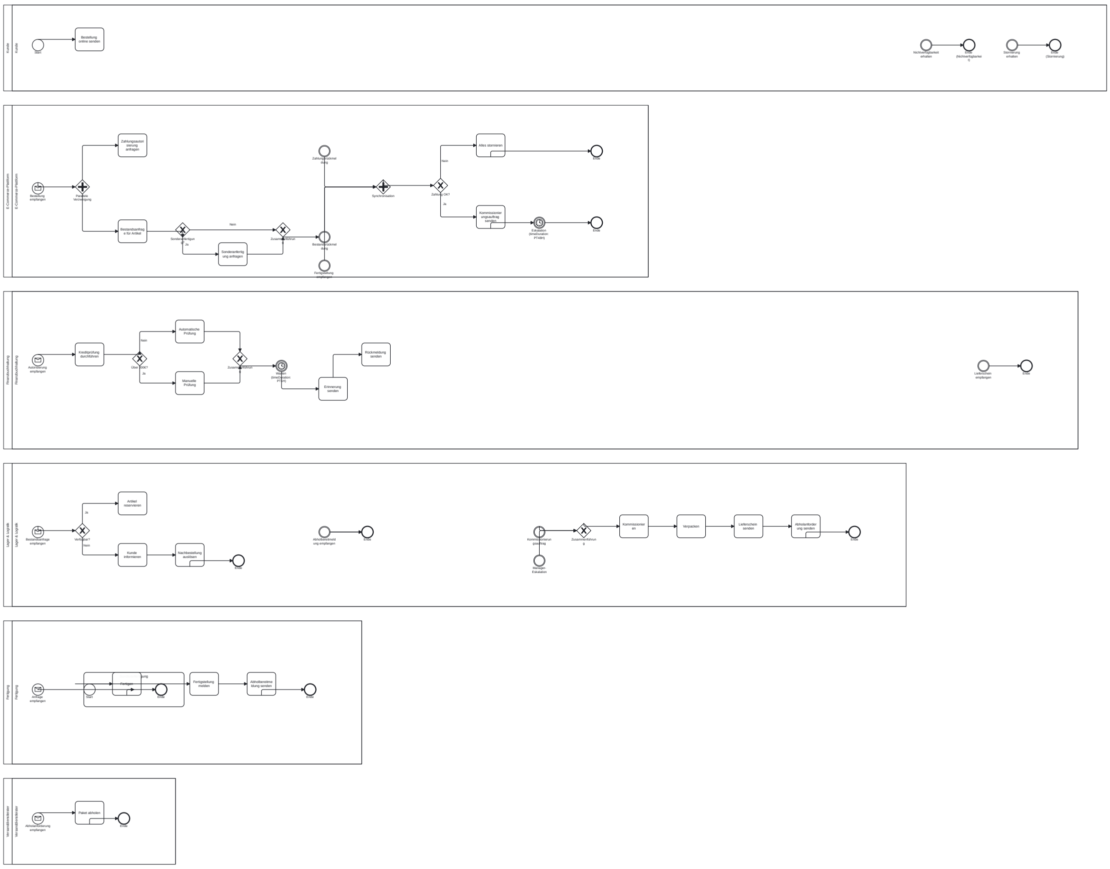

Abbildung B.19: Diagramm von Claude Sonnet 4.5 mit JSON
11172 TOKEN | 0.213 $ | 107 s

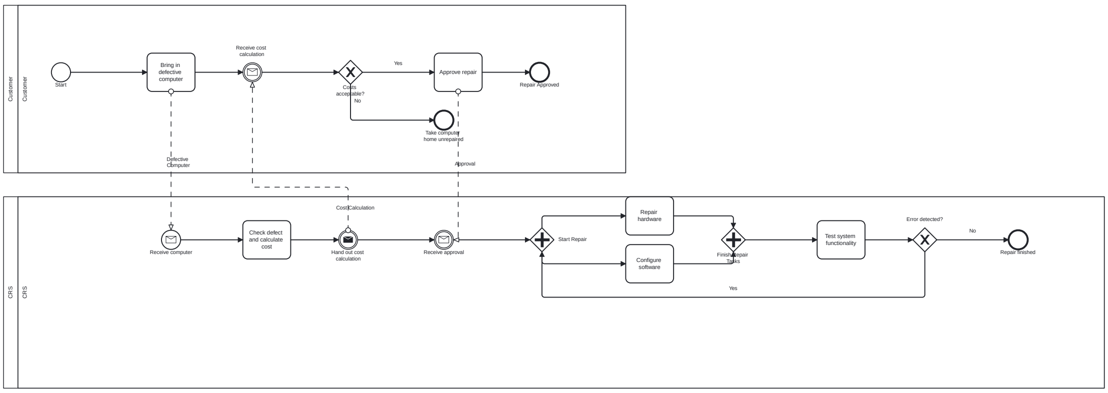

Abbildung B.20: Diagramm von Gemini 2.5 Pro mit JSON
3817 TOKEN | 0.000 $ | 63 s

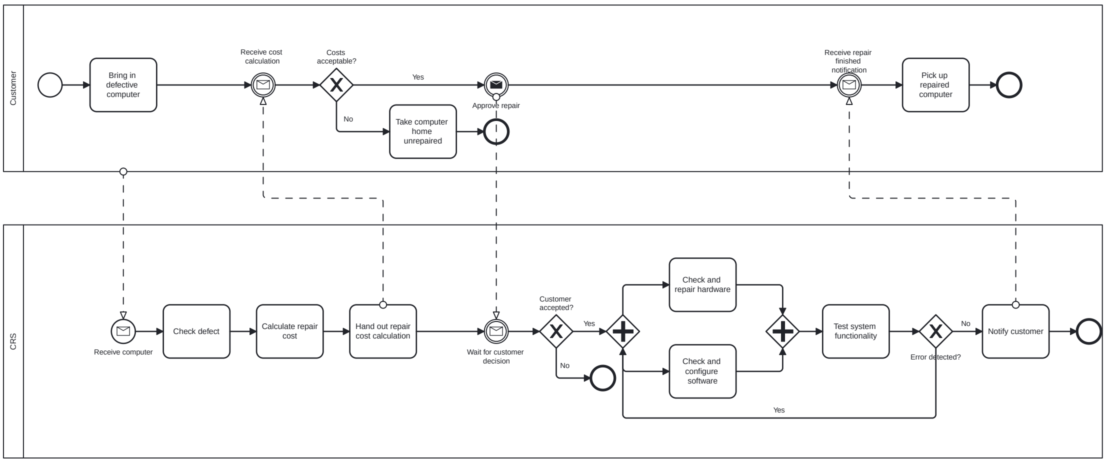

Abbildung B.21: Diagramm von Gemini 2.5 Pro mit XML
8632 TOKEN | 0.000 $ | 94 s

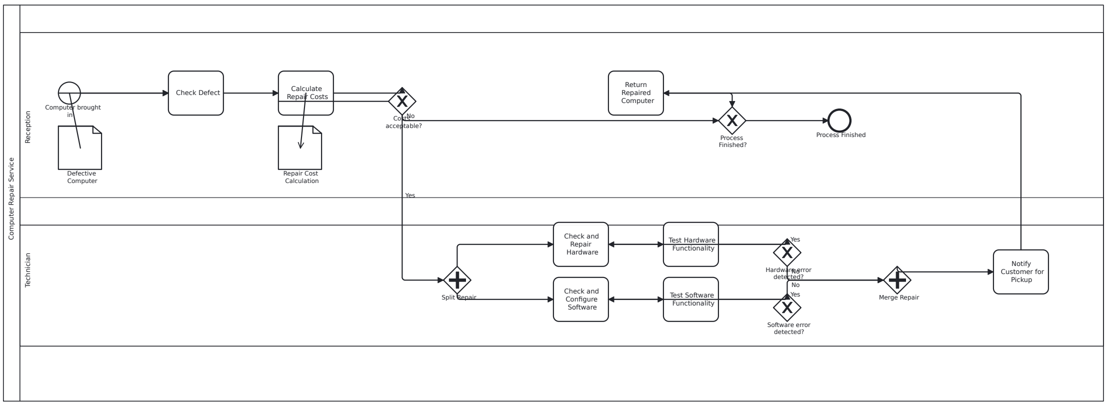

Abbildung B.22: Diagramm von Gemini 2.5 Flash mit JSON
3938 TOKEN | 0.000 $ | 95 s

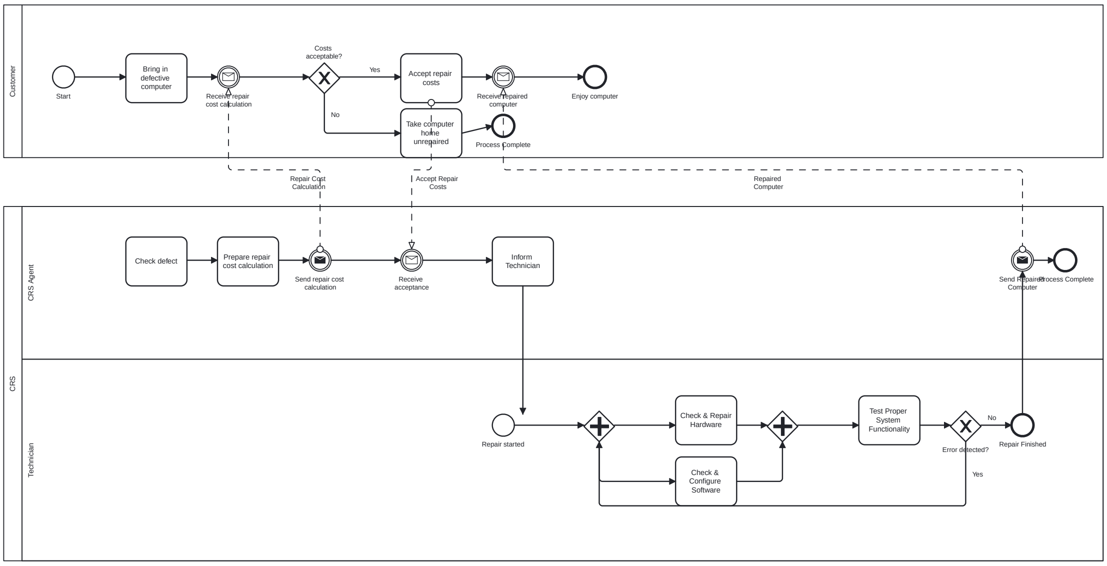

Abbildung B.23: Diagramm von Gemini 2.5 Flash mit XML
10757 TOKEN | 0.000 $ | 84 s

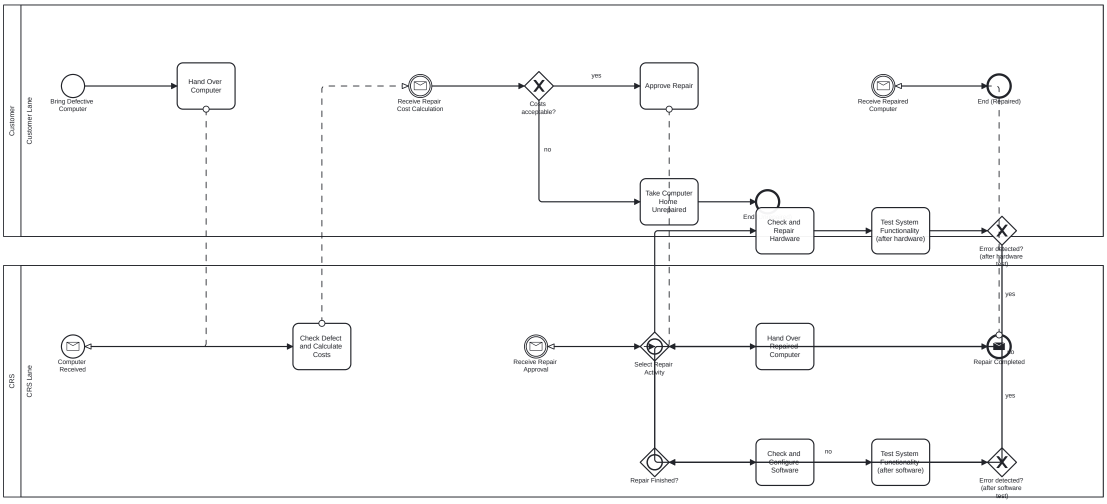

Abbildung B.24: Diagramm von ChatGPT 5.2 mit JSON
3651 TOKEN | 0.072 $ | 27 s

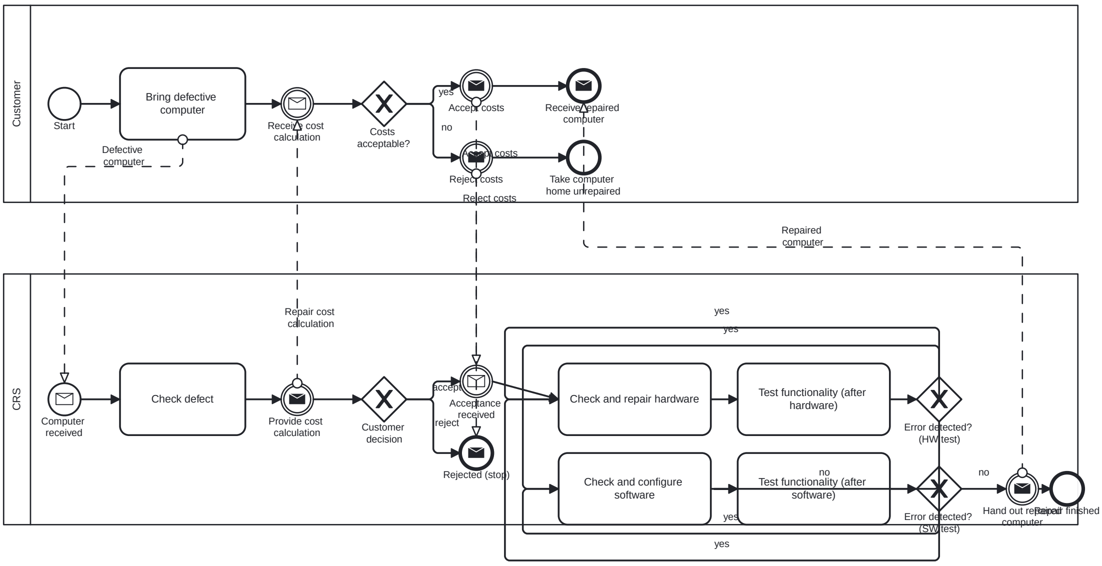

Abbildung B.25: Diagramm von ChatGPT 5.2 mit XML
6767 TOKEN | 0.118 $ | 50 s

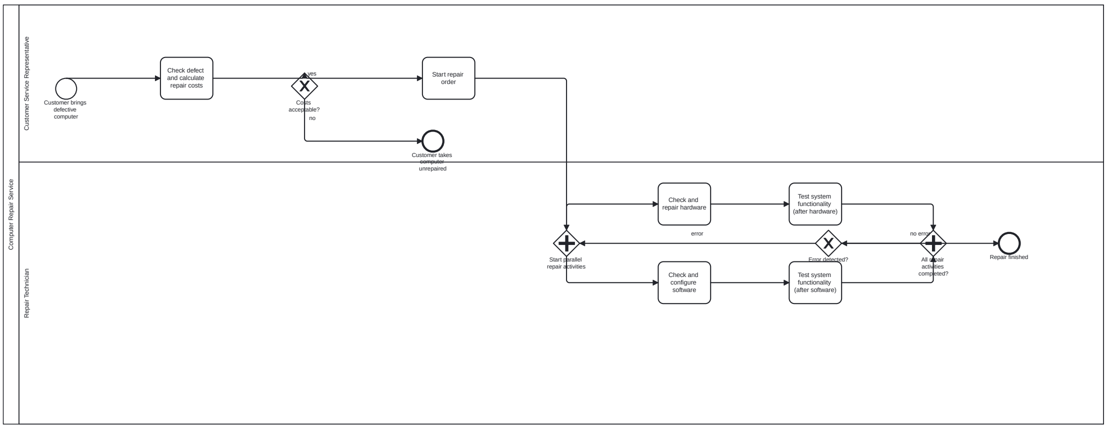

Abbildung B.26: Diagramm von ChatGPT 5.1 mit JSON
1895 TOKEN | 0.034 $ | 22 s

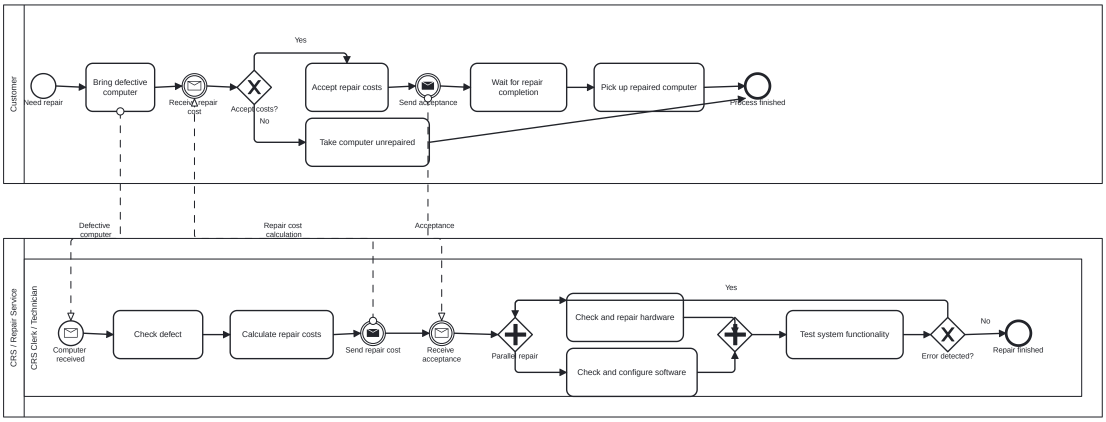

Abbildung B.27: Diagramm von ChatGPT 5.1 mit XML
6347 TOKEN | 0.080 $ | 83 s

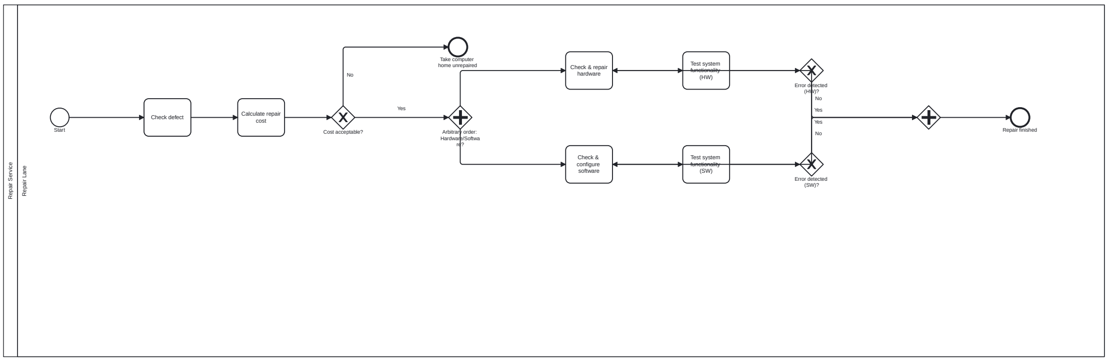

Abbildung B.28: Diagramm von ChatGPT 4.1 mit JSON
2021 TOKEN | 0.040 $ | 18 s

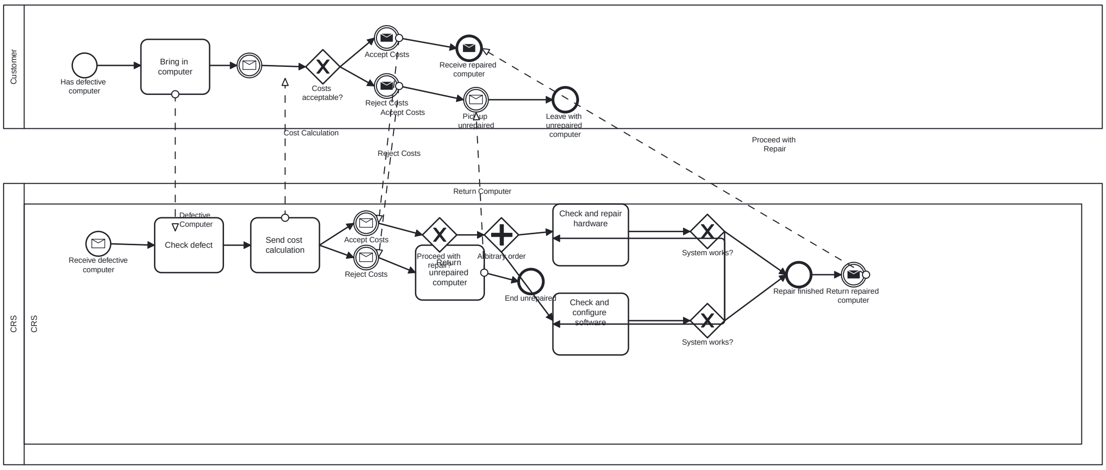

Abbildung B.29: Diagramm von ChatGPT 4.1 mit XML
7379 TOKEN | 0.085 $ | 70 s

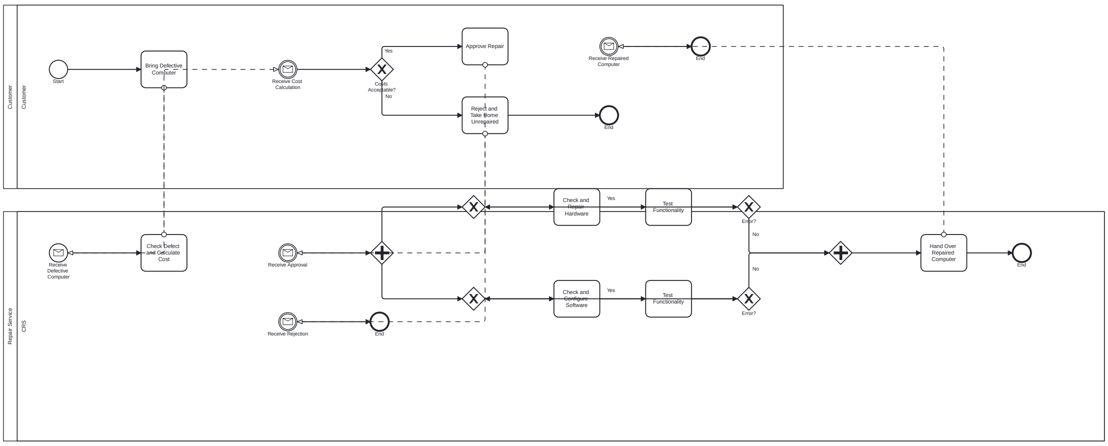

Abbildung B.30: Diagramm von Grok 4 mit JSON
3578+3432 TOKEN | 0.143 $ | 159 s

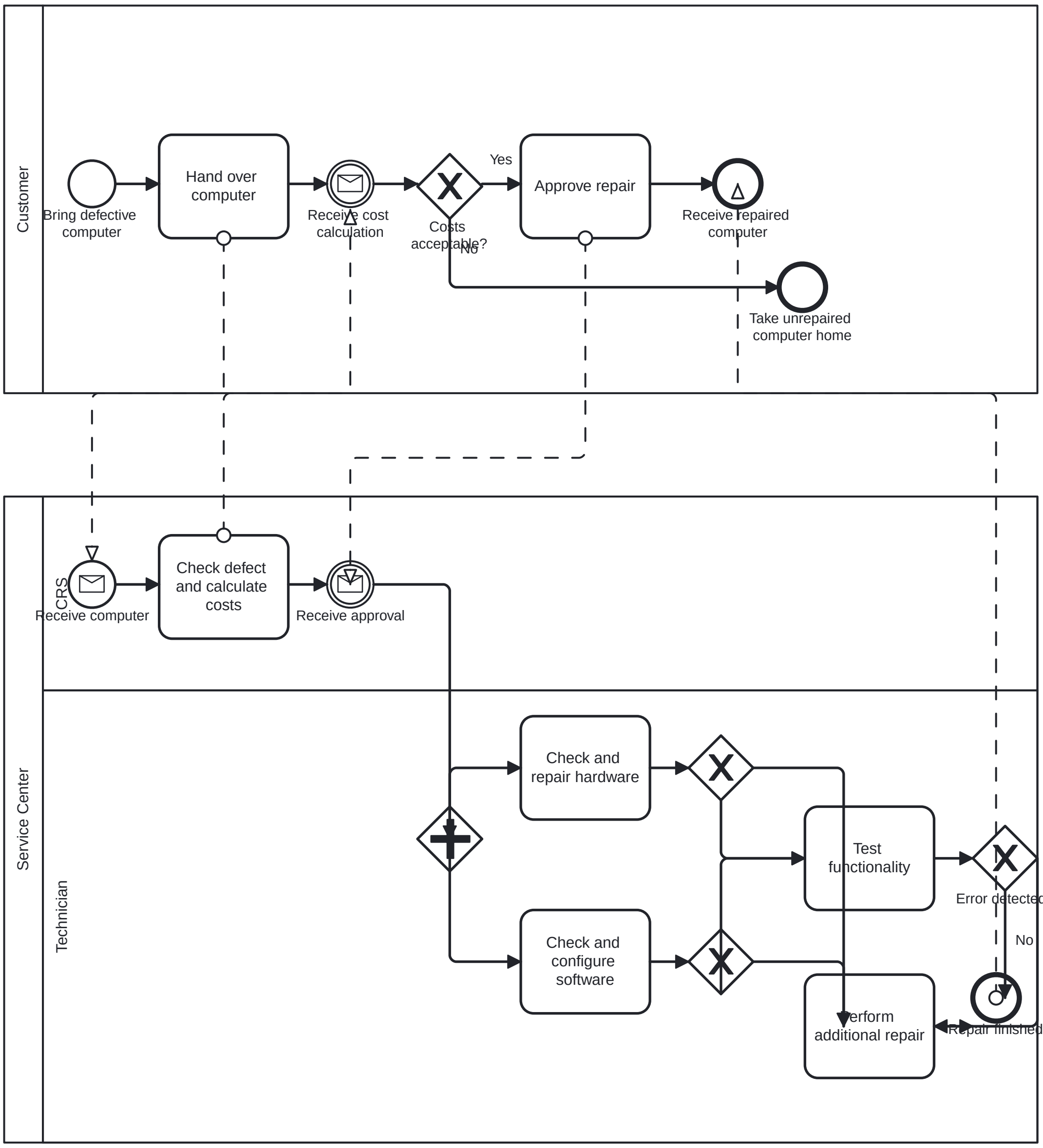

Abbildung B.31: Diagramm von Grok 4 mit XML
5885+780 TOKEN | 0.140 $ | 162 s

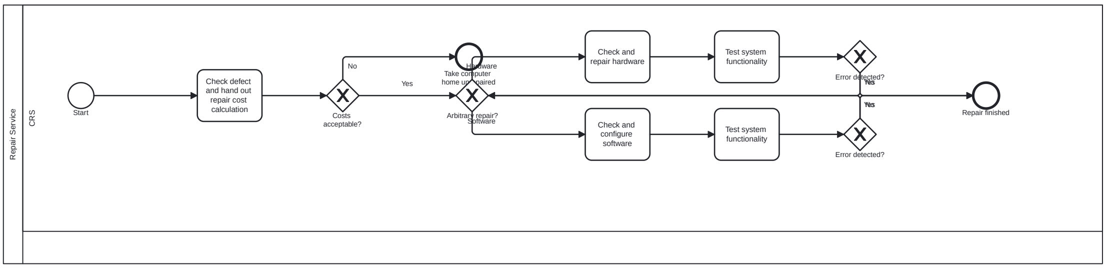

Abbildung B.32: Diagramm von Grok 4.1 Fast mit JSON
1760+8033 TOKEN | 0.007 $ | 115 s

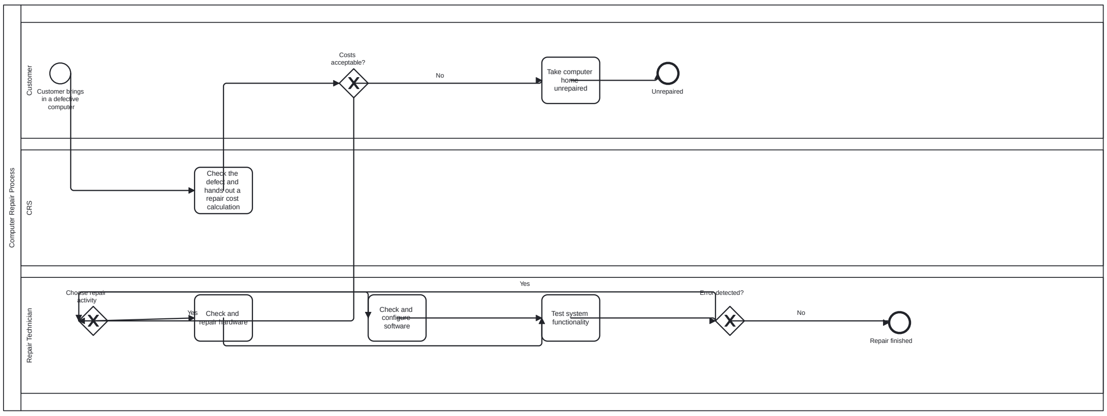

Abbildung B.33: Diagramm von Grok 4.1 Fast mit XML
4254+10244 TOKEN | 0.010 $ | 126 s

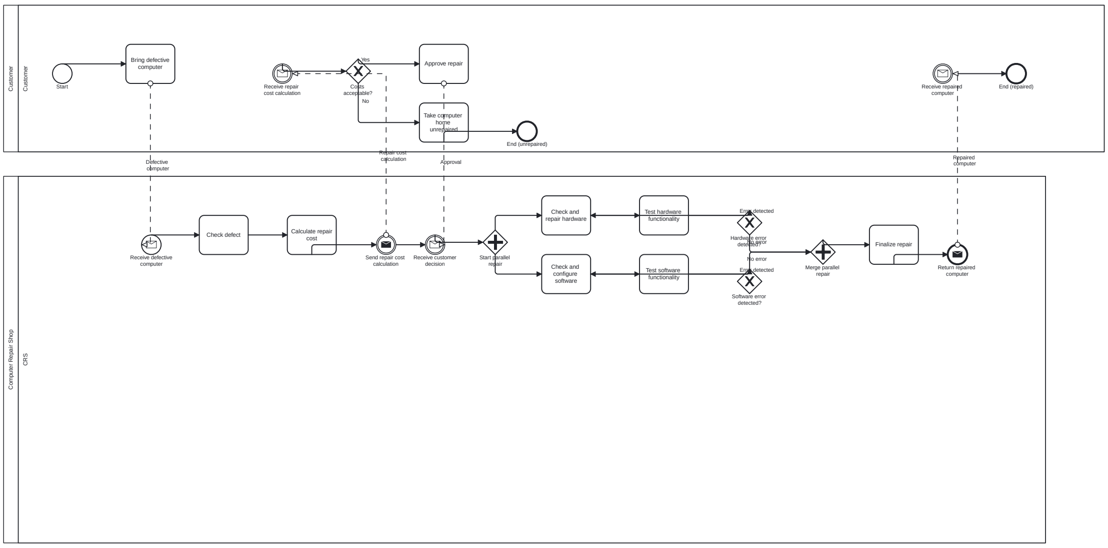

Abbildung B.34: Diagramm von Claude Opus 4.5 mit JSON
4657 TOKEN | 0.190 $ | 42 s

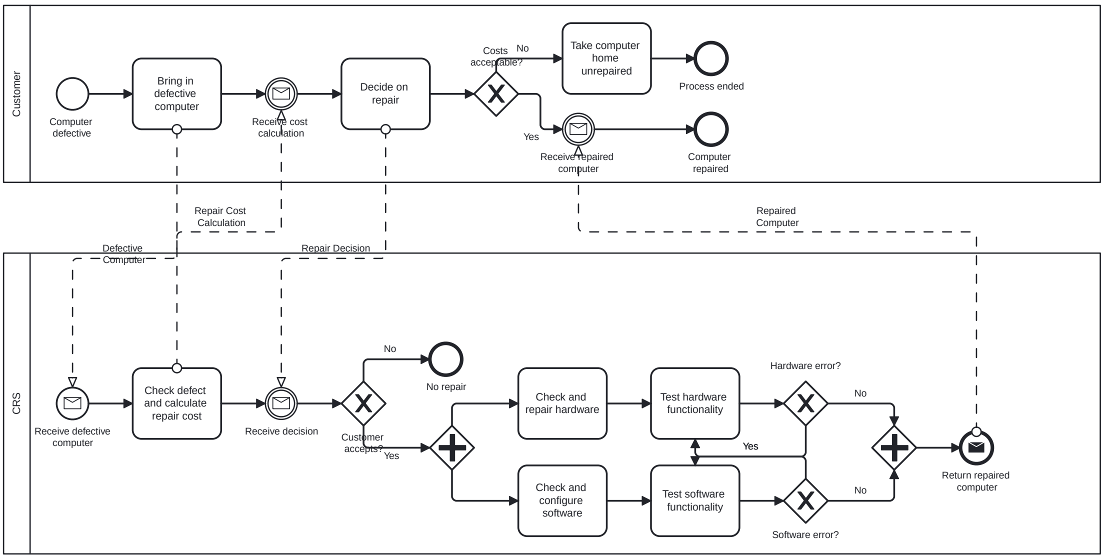

Abbildung B.35: Diagramm von Claude Opus 4.5 mit XML
9531 TOKEN | 0.318 $ | 74 s

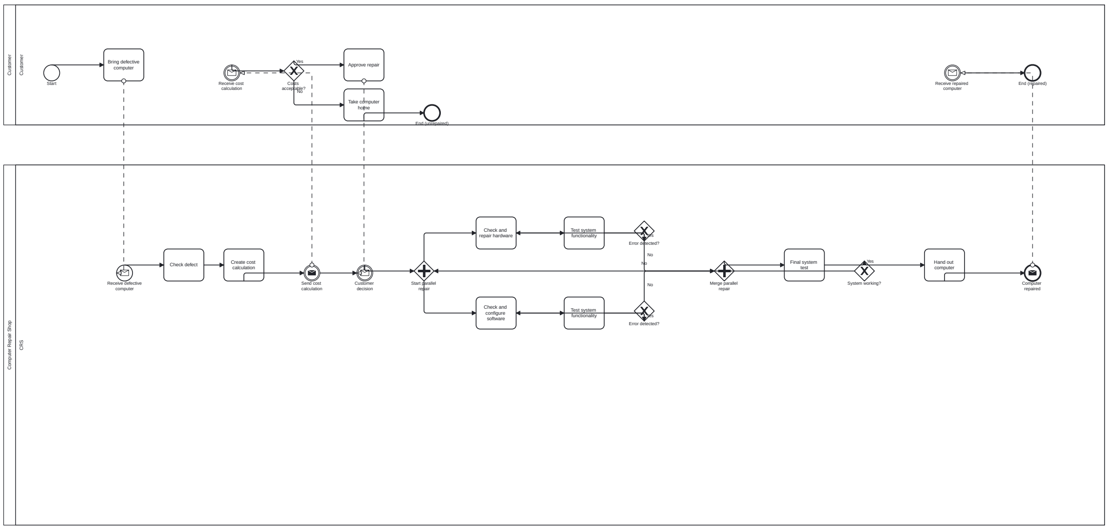

Abbildung B.36: Diagramm von Claude Sonnet 4.5 mit JSON
4623 TOKEN | 0.114 $ | 45 s

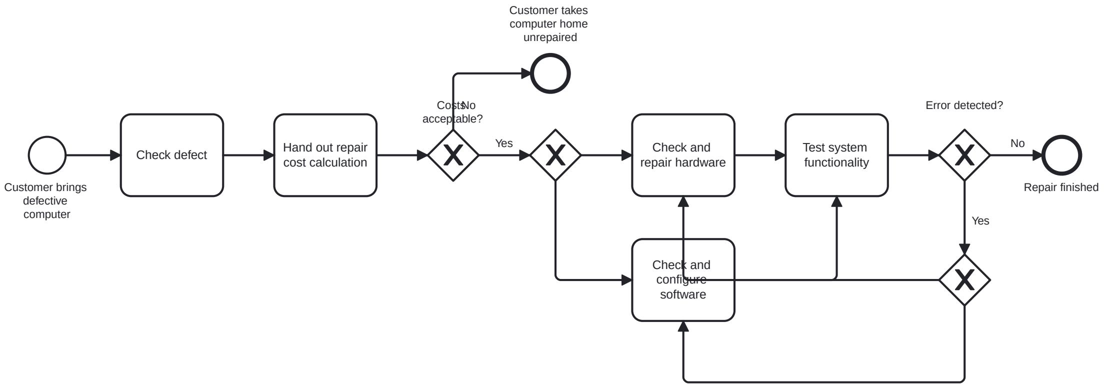

Abbildung B.37: Diagramm von Claude Sonnet 4.5 mit XML
4341 TOKEN | 0.113 $ | 39 s

Literatur

[1] Patrizio Bellan u. a. „PET: An Annotated Dataset for Process Extraction from Natural Language Text Tasks“. In: *Business Process Management Workshops - BPM 2022 International Workshops, Münster, Germany, September 11-16, 2022, Revised Selected Papers*. Hrsg. von Cristina Cabanillas, Niels Frederik Garmann-Johnsen und Agnes Koschmider. Bd. 460. Lecture Notes in Business Information Processing. Springer, 2022, S. 315–321. DOI: `10.1007/978-3-031-25383-6\_23`. URL: `https://doi.org/10.1007/978-3-031-25383-6\_23`.

[2] Flavio Corradini u. a. „Correctness checking for BPMN collaborations with sub-processes“. In: *Journal of Systems and Software* 166 (2020), S. 110594. ISSN: 0164-1212. DOI: `https://doi.org/10.1016/j.jss.2020.110594`. URL: `https://www.sciencedirect.com/science/article/pii/S0164121220300716`.

[3] D. Connolly and L. Masinter. *RFC 2397: The “data” URL Scheme*. `https://www.rfc-editor.org/rfc/rfc2397.html`. W3C / IETF Standard. Juli 1998.

[4] Ian Fette und Alexey Melnikov. *HTML5 Server-Sent Events*. `https://www.w3.org/TR/eventsource/`. W3C Recommendation. 2011.

[5] Object Management Group. *Business Process Model and Notation (BPMN), Version 2.0.2*. Normative specification. Jan. 2014. URL: `https://www.omg.org/spec/BPMN/2.0.2/`.

[6] Nataliia Klievtsova u. a. „AutoBPMN.AI: Conversational Process Modeling and Automation“. English. In: *CEUR Workshop Proceedings* 4032 (2025). Publisher Copyright: © 2025 Copyright for this paper by its authors.; Best Dissertation Award, Doctoral Consortium, and Demonstration and Resources Forum at 23rd International Conference on Business Process Management, BPM-D

2025 ; Conference date: 31-08-2025 Through 05-09-2025, S. 304–311. ISSN: 1613-0073.

[7] Julius Köpke und Aya M. A. Safan. „Efficient LLM-Based Conversational Process Modeling“. In: *NLP4BPM Workshop at BPM 2024*. 2024.

[8] Julius Köpke und Aya M. A. Safan. „Introducing the BPMN-Chatbot for Efficient LLM-Based Process Modeling“. In: *BPM 2024 Demos*. 2024.

[9] Humam Kourani und Sebastiaan van Zelst. „POWL: Partially Ordered Workflow Language“. In: *arXiv preprint* (2023). URL: `https://sebastiaanvanzelst.com/wp-content/uploads/2023/08/paper_6723.pdf`.

[10] Humam Kourani u. a. „ProMoAI: Process Modeling with Generative AI“. In: *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI-24*. Hrsg. von Kate Larson. Demo Track. International Joint Conferences on Artificial Intelligence Organization, Aug. 2024, S. 8708–8712. DOI: `10.24963/ijcai.2024/1014`. URL: `https://doi.org/10.24963/ijcai.2024/1014`.

[11] Ali Nour Eldin u. a. „A Decomposed Hybrid Approach to Business Process Modeling with LLMs“. In: Nov. 2024.

[12] Ali Nour Eldin u. a. „Nala2BPMN: Automating BPMN Model Generation with Large Language Models“. In: Nov. 2025.

[13] Aya M. A. Safan und Julius Köpke. „A Framework for LLM-Based Conceptual Modeling: Application to BPMN Collaboration Diagrams“. In: *ER Forum in ER 2025: Companion Proceedings of the 44th International Conference on Conceptual Modeling: Industrial Track, ER Forum, 8th SCME, Doctoral Consortium, Tutorials, Project Exhibitions, Posters and Demos*. Poitiers, France, 2025.

[14] Aya M. A. Safan und Julius Köpke. „BPMN-Chatbot++: LLM-Based Modeling of Collaboration Diagrams with Data“. In: *Proceedings of the Best Dissertation Award, Doctoral Consortium, and Demonstration & Resources Forum at BPM 2024 co-located with the 23rd International Conference on Business Process Management (BPM 2025)*. Seville, Spain: CEUR-WS.org, 2025.

[15] Konrad Schneid u. a. *Data-Flow Analysis of BPMN-based Process-Driven Applications: Detecting anomalies across model and code*. eng. ERCIS Working Paper 38. Münster, 2021. URL: `https://hdl.handle.net/10419/243142`.

[16] Zhe Shi. „Enhancing BPMNGen: Improving LLM-based BPMN 2.0 Process Model Generation through Natural Language Processing“. Submitted for the degree of Bachelor of Science (B.Sc) in Informatik. Advisor: Prof. Dr. Manfred Reichert; Supervisor: Luca Hörner. Bachelor thesis. Ulm, Germany: Ulm University, 2025.

[17] T. D. Hansen and P. Hoffman and A. Malhotra. *RFC 4648: The Base16, Base32, and Base64 Data Encodings*. `https://www.rfc-editor.org/rfc/rfc4648.html`. IETF Standard. Okt. 2006.

[18] Ashish Vaswani u. a. *Attention Is All You Need*. 2023. arXiv: `1706.03762 [cs.CL]`. URL: `https://arxiv.org/abs/1706.03762`.

[19] Jason Wei u. a. *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models*. 2023. arXiv: `2201.11903 [cs.CL]`. URL: `https://arxiv.org/abs/2201.11903`.

[20] Niklas Weidl. „BPMN Diagram Generation with ChatGPT“. Submitted for the degree of Bachelor of Science (BSc) in Medieninformatik. Advisors: Prof. Dr. Manfred Reichert; Supervisor: Luca Hörner. Bachelor thesis. Ulm, Germany: Ulm University, 2024.

[21] Guangxuan Xiao u. a. „Efficient Streaming Language Models with Attention Sinks“. In: *International Conference on Representation Learning*. Hrsg. von B. Kim u. a. Bd. 2024. 2024, S. 21875–21895. URL: `https://proceedings.iclr.cc/paper_files/paper/2024/file/5e5fd18f863cbe6d8ae392a93fd271c9-Paper-Conference.pdf`.

[22] Xinran Zhao u. a. *Fact-and-Reflection (FaR) Improves Confidence Calibration of Large Language Models*. 2024. arXiv: `2402.17124 [cs.CL]`. URL: `https://arxiv.org/abs/2402.17124`.

[23] Clara Ziche und Giovanni Apruzzese. „LLM4PM: A case study on using Large Language Models for Process Modeling in Enterprise Organizations“. In: (2024).

Name: Philipp Letschka Matrikelnummer: 1050994

Erklärung

Ich erkläre, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den ....

Philipp Letschka