



universität
uulm

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 19. November 2025

Danksagung

Das ist der Text der Danksagung

Zusammenfassung

Das ist der Text der Zusammenfassung

Inhaltsverzeichnis

1 Einleitung	1
1.1 Motivation	1
1.2 Problemstellung und Zielsetzung	1
1.3 Struktur der Arbeit	1
2 Theoretische Grundlagen	2
3 Umstrukturierung und Innovation	3
3.1 Generelle Umstrukturierungen	3
3.1.1 Objektorientierter Ansatz	3
3.1.2 Von Assistants zu Responses	5
3.1.3 Verbesserung der instructions	5
3.2 Formatauswahl	6
3.3 Dateien	8
3.4 Chain of Thought	10
3.4.1 Implementierung eines neuen Modus	10
3.4.2 Konversationskontext	11
3.4.3 Konversationen	14
3.5 Weitere Anbieter	20
3.5.1 Grok	21
3.5.2 Gemini	22
3.5.3 Claude	24
3.6 Streaming	25
3.6.1 SSE	28
3.7 Schema-Constraining	31
3.8 Evtl. Sampling	31
3.9 Evtl. Reflective Prompting	31

Inhaltsverzeichnis

4 Performanzanalyse	32
4.1 Qualität	32
4.2 Geschwindigkeit	32
4.3 Kosten	37
5 Verwandte Arbeiten	38
6 Fazit	39
6.1 Zusammenfassung	39
6.2 Ausblick	39
A Quelltexte	48
Literatur	49

1 Einleitung

1.1 Motivation

1.2 Problemstellung und Zielsetzung

1.3 Struktur der Arbeit

2 Theoretische Grundlagen

3 Umstrukturierung und Innovation

Als erstes gilt es herauszufinden welcher Teil des Code, der zuständig für das prompting ist, wie verbessert werden kann. Das Projekt von TBA benutzt zur Erstellung von BPMN Diagrammen die OpenAI API und verwendet hier die bereitgestellte Technologie der Assistants.

3.1 Generelle Umstrukturierungen

Während der Code gut für seinen (bisherigen) speziellen Anwendungsfall 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 wartbarer. Wir erstellen eine abstrakte Klasse `Ai` welche die gesamte Logik des Prompting beinhaltet 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 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 sieht nun in abgespeckter 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 `preprocessResponse()`, um die Antwort der API auszulesen.

```
1  export class ChatGPT extends Ai {
2      openai = new OpenAI({
3          apiKey: OPENAI_API_KEY,
4      });
5      assist = await openai.beta.assistants.retrieve("asst_...");
6
7      protected mapPromptInput(input) {
8          return {
9              input: input.prompt,
10             model: this.model,
11         };
12     }
13
14     protected async generateContent(input) {
15         return await openai.beta.threads.runs.createAndPoll(
16             thread_id,
17             { assistant_id: assist.id },
18             { role: "user", content: input }
19         );
20     }
21
22     protected processResponse(response) {
23         return response.output_text.toString();
24     }
25 }
```

Codeausschnitt 3.1.1: ChatGPT Klasse

3.1.2 Von Assistants zu Responses

Die Änderungen welche im vorherigen Kapitel beschrieben wurden, zeigen gleich ihren Effekt, da OpenAI ankündigt ihre Assistants API einstellen zu wollen. Sie empfehlen einen Umzug zu ihrer neuen Responses API. Dies ist nun recht einfach umzusetzen, da wir nur die elementaren Methoden der API ändern müssen. So sieht nun die neue Methode `generateContent()` aus:

```
1 protected async generateContent(input) {  
2     return this.openai.responses.create(input);  
3 }
```

Codeausschnitt 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. Dieser Umstand 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 im Detail beschreiben, sowie zwei Textdateien: eine mit der Definition des verwendeten JSON-Formats und eine mit allgemeinen Regeln zum Aufbau der 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. Mehrere Tests zeigen, dass sich diese Reduktion nicht negativ auf die Ergebnisqualität auswirkt. Dadurch lassen sich eine große Zahl an Tokens sowie Rechenzeit und Kosten einsparen.

Die beiden verbliebenen Textdokumente werden anschließend zusammengeführt, überarbeitet und in ein einheitliches, strukturiertes Format gebracht. Alle Regeln sind in einer geordneten Liste zusammengefasst und durch sogenanntes structured prompting klarer und maschinenlesbarer 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 Komponenten abbildet. Diese Kombination sorgt dafür, dass der Assistant sowohl einfache als auch komplexe Diagramme präzise interpretieren und reproduzieren kann.

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 weltweit am häufigsten verwendete Austauschformat für BPMN-Diagramme ist XML, zu dem umfangreiche Dokumentation und etablierte Werkzeuge existieren. Dennoch bietet das eigens entwickelte JSON-Format einen entscheidenden Vorteil: Es besitzt eine deutlich höhere Informationsdichte und lässt sich dadurch kompakter und effizienter verarbeiten. Beide Formate haben somit ihre jeweiligen Stärken und Einsatzgebiete.

Kriterium	JSON-Format	XML-Format
Vorteile	hohe Informationsdichte, geringer Tokenverbrauch, schnelle Generierung.	Standardisiert, gut dokumentiert, weit verbreitet, einfache Instructions
Nachteile	Kein Standard, Lernaufwand, komplizierter Konvertierungsalgorithmus.	hoher Tokenverbrauch, umfangreiche Syntax, unübersichtlicher, mehr Kosten, längere Generierung.

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

Für die Weiterentwicklung ist ein flexibles System das Ziel, 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. folgendermaßen aus:

```

1 // POST /threads
2 {
3     "inputString": "Bitte generiere mir ein BPMN Diagramm,
4                 welches den Ablauf in einem Restaurant zeigt",
5     "model": "gpt-5 (xml)",
6     "format": "xml"
7 }
```

Codeauschnitt 3.2.1: Post Request an /threads mit Format

Bei einer Anfrage kann dann die jeweilige AI über eine Map

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

zugeordnet werden, welche die Anfrage bearbeitet.

// TODO: update neuerungen des alten json formats noch machen

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 unter <https://platform.openai.com/docs/guides/images-vision?api-mode=responses&format=base64-encode-analyze-images> zeigt sich, dass die einfachste und zugleich 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 im eine Textdarstellung einer Datei, die direkt in eine URL eingebettet wird. 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 automatisch wieder in die ursprüngliche Datei zurückwandeln kann. Auf diese Weise lassen sich Dateien direkt in JSON API-Anfragen integrieren, ohne dass zusätzliche Dateipfade, Server oder externe Speicherorte erforderlich sind. Dieses Verfahren ist daher besonders gut geeignet, um eine einfache, schnelle und universell kompatible Dateiübertragung zu ermöglichen. 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. folgendermaßen aus:

```
1 // POST /threads
2 {
3     "inputString": "Bitte generiere mir ein BPMN Diagramm,
4         welches den Ablauf in einem Restaurant zeigt",
5     "model": "gpt-5 (xml)",
6     "file": "data:image/png;base64,A35ekZ...",
7 }
```

Codeausschnitt 3.3.1: Post Request an /threads mit Datei

Der string wird auf Validität geprüft

```
1 private checkBase64DataUrl(dataUrl: string): boolean {
2     regex = /^data:([\w.+-]+\/[\w.+-]+)?;base64,[\w+\//]+==$/;
3     return regex.test(dataUrl);
4 }
```

Codeausschnitt 3.3.2: checkBase64DataUrl()

und dann im richtigen Format an die OpenAI API gesendet:

```
1 const imageInstructions = {
2     role: "user",
3     content: [
4         {
5             type: "input_file",
6             file_url: input.getFileDataUrl() as string,
7         } as ResponseInputFile,
8     ],
9 } as ResponseInputItem
```

Codeausschnitt 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’ bezeichnet wird. 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, Ideen entwickelt, bestehende Diagramme erläutert oder Verbesserungsvorschläge erfragt werden. Der BPMN-Bot soll damit nicht nur ein Werkzeug für Diagrammerstellung bleiben, sondern sich zu einem vollwertigen, kontextbewussten 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 aktiv zum Verständnis und zur Qualität der Diagrammerstellung beitragen. Um dies zu ermöglichen, benötigt der Chatbot Zugriff auf den bisherigen Gesprächsverlauf sowie auf den aktuellen Zustand des jeweiligen Diagramms. Nur durch diese Kontextkenntnis kann die KI präzise, nachvollziehbare und qualitativ hochwertige 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 sogenannten ‘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 folgendermaßen aussehen:

```
1 // POST /threads
2 {
3     "inputString": "Bitte schlag drei Prozessbeschreibungen
4         vor, aus denen dann eine ausgewählt wird um ein Diagramm
5         zu erstellen.",
6     "model": "gpt-5 (xml)",
7     "mode": "detail",
8 }
```

Codeausschnitt 3.4.1: Post Request an /threads mit mode

3.4.2 Konversationskontext

Da es nun darum geht ein 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 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 aktuelle Nachricht bereitgestellt wird. Bei LLM Anbietern wie OpenAI ist es notwendig, dass die Nachrichten Historie in der Anfrage mitgeschickt wird.

Hierfür müssen alle Nachrichten eines Threads aus der Datenbank geladen werden. Die Nachrichten werden dann auf wesentliche gefiltert und auf ein kompaktes Format gebracht. Die OpenAI Klasse muss dann nur noch die Nachrichten auf das

gefoderte Format bringen und in der Anfrage mitschicken. Da die Anfagen 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 diese Objekt können alle Rohdaten wie Instructions, Dateien oder der Chatverlauf übergebe 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 Diagram wichtig ist. Hier bei 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 folgende Attribute:

```
1 export class PromptInput {  
2     instructions: string[];  
3     history: {role: "user" | "assistant", content: string}[];  
4     prompt: string;  
5     file?: ;  
6 }
```

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. Dadurch entsteht ein einheitlicher, wiederverwendbarer Datenfluss zwischen

```
1 const instructions = [formatInstructions, modeInstructions];
2 const chatsFromDB = getAllChatsFromDB(threadID);
3 new PromptInput(instructions, prompt, chatsFromDB, file);
```

Codeauschnitt 3.4.3: PromptInput Erstellung

Anwendung, Datenbank und Modell, der die Wartung sowie zukünftige Erweiterungen vereinfacht.

```
1 protected mapPromptInput(input: PromptInput) {
2     [...]
3     const historyInstructions = input.history.map((item) => {
4         return {role: item.role, content: item.content};
5     });
6     return {
7         input: [systemInstructions, historyInstructions,
8                 userInstructions, fileInstructions],
9         model: this.model,
10    };
11 }
```

Codeauschnitt 3.4.4: 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 sogenannten ‘System Instructions’ hinterlegt. Diesen Instruction Block wird nun als ‘Update Instructions’ 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.

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

```
1 protected updateInstructions(threadID: string, format: format) {  
2     const diagram = getLatestDiagramFromDB(threadID);  
3     return [`The following diagram has already been created:  
4         ${format == "xml" ? diagram?.xml : diagram?.json}`]  
5 }
```

Codeauschnitt 3.4.5: 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 werden der KI noch die sogenannten *modeInstructions* bereitgestellt. Diese sehen zunächst so aus:

```
1 [Output Language depends on prompt language]  
2  
3 Instead of creating a bpmn diagram, you can also chat with the  
4 user.  
5 Any descriptions, questions, summaries or any other type of  
6 text the user may want, you must answer.  
7 You may ask any questions you feel like you need to  
8 answer to create the best bpmn diagram.  
9 If you do not have any questions anymore, you may write a  
10 quick summary of the agreed diagram before creating the  
11 bpmn.  
12 If the initial prompt is really basic, provide the user with a  
13 creative scenario summary.  
14  
15 If you want to provide questions, summary or any other type of  
16 text, that is not a diagram. Do not use JSON or XML.  
17 Only plain Text!
```

3 Umstrukturierung und Innovation

18 You may answer with both a plaintext section and a diagram.
19 Do not put plain text in a JSON or XML section.
20 If you want to provide a description, summary or any other type
21 of text, that is not a diagram, send it outside the diagram.

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

Es ist nun eine Konversation möglich.

Wenn man den Bot nun fragt: "Bitte schlag mir ein Diagramm vor" antwortet dieser 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 darauf 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 nun kein Klartext sondern direkt ein BPMN XML. Dieses sieht folgendermaßen aus:

3 Umstrukturierung und Innovation

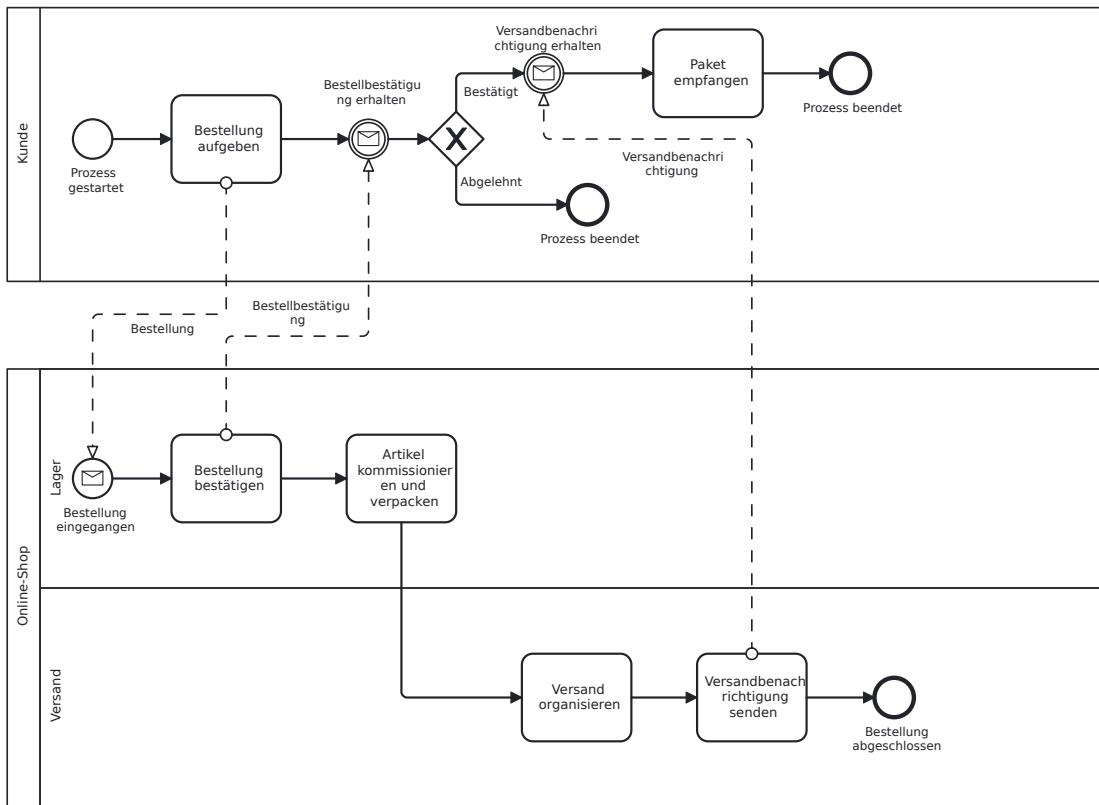


Abbildung 3.1: Generierung eines Diagrammes

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.

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 Dia-

3 Umstrukturierung und Innovation

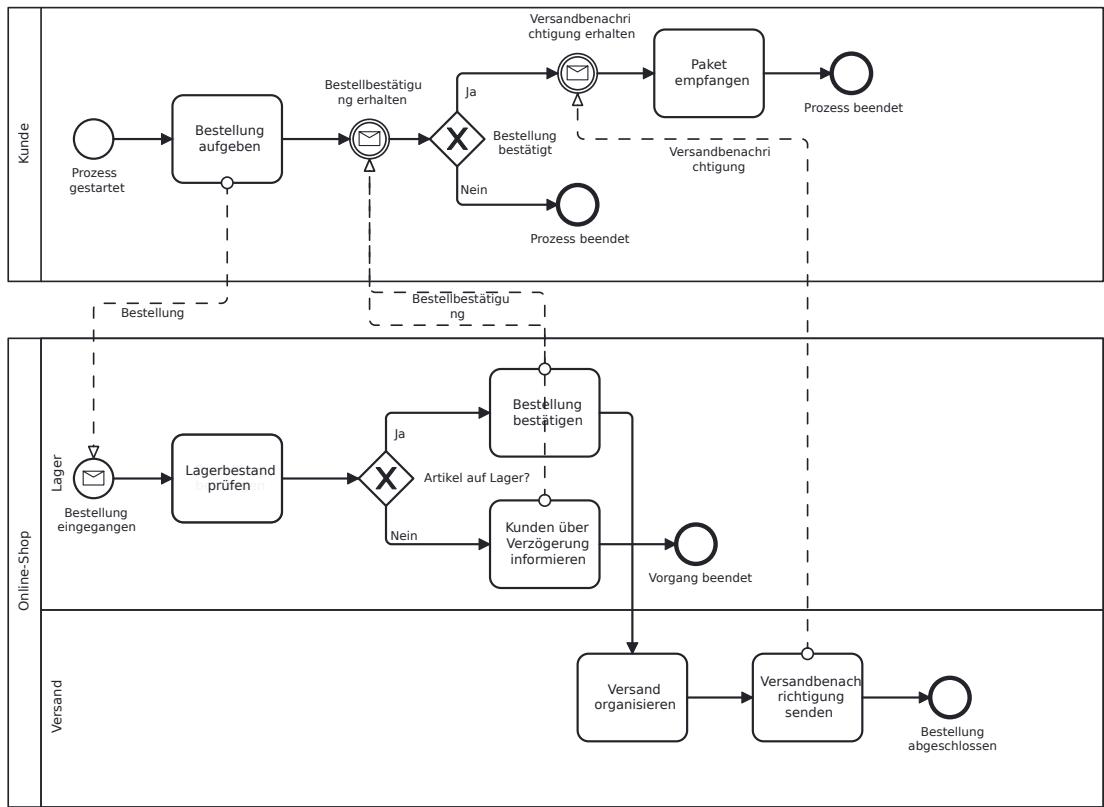


Abbildung 3.2: Überarbeitung eines Diagrammes mit dem detail mode

gramm gleichzeitig enthalten. Dadurch wird es möglich, dass der Bot zunächst eine 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, klar abgrenzbaren 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 Ant-

wort der KI analysiert und die jeweiligen Segmente eindeutig identifiziert. Der Chat-Bot muss erkennen, welche Abschnitte in natürlicher Sprache formuliert sind und welche Bestandteile tatsächlich ein Diagramm darstellen, das weiterverarbeitet oder visualisiert werden soll. Dieses Verhalten ermöglicht eine deutlich flexiblere Interaktion und eröffnet neue Einsatzmöglichkeiten, besonders dann, wenn Nutzer sowohl inhaltliche Erläuterungen als auch die direkte Umsetzung in einem BPMN-Diagramm erwarten.

Mit dieser Erweiterung entwickelt sich der BPMN-Bot zunehmend zu einem umfassenden Assistenzsystem, das nicht nur Modellierungsartefakte erzeugt, sondern auch begleitend erklärt, begründet und interaktiv mit dem Nutzer zusammenarbeitet.

Für dieses Kategorisierung wird eine Beispiel-Antwort betrachtet:

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.

Über dieses werden automatisch alle validen XML Teile (In TBA label bunt markiert) erkannt. Durch die Benutzung einer Non-capturing group werden Wrapper des XML (In TBA label blau markiert) wie zum Beispiel das '```xml', automatisch entfernt.

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

#### Codeauschnitt 3.4.6: Regex zur Diagrammerkennung

Alle matches 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 TBA label rot 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. Grüne Teile sind Diagramme, Nicht markierte Teile und rote Teile sind Teil der Klartextnachricht.

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 zu gewährleisten. Unterschiedliche Anbieter bieten verschiedene Stärken, wie etwa spezialisierte Natural Language Processing-Modelle, schnellere Antwort-

zeiten oder kosteneffizientere Lösungen. Durch die Integration mehrerer Anbieter kann der Bot je nach Bedarf die beste Leistung auswählen, alternative Perspektiven liefern und Ausfälle eines einzelnen Dienstes abfedern. Dies erhöht die Zuverlässigkeit und Qualität der generierten BPMN-Diagramme.

Durch die Objektorientierte Konfiguration der KI Schnittstelle ist es nun sehr einfach weitere Anbieter hinzuzufügen. Im weiteren wird gezeigt wie weitere Anbieter nun hinzugefügt werden.

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 erhöht damit die Vielfalt, Genauigkeit und Zuverlässigkeit des Systems.

Grok kann einfach über die XAI SDK hinzugefügt werden. Die Dokumentation findet man hier: <https://ai-sdk.dev/providers/ai-sdk-providers/xai>

Die zu versendende Nachricht an die Grok API muss nun nur noch vom PromptInput Objekt auf das API Format gebracht werden.

```
1  protected mapPromptInput(input: PromptInput) {
2    const historyInst = input.history.map((item) => {
3      return {role: item.role, content: item.content} ;
4    })
5    const fileInst = input.file ? {
6      role: "user",
7      content: [
8        type: "image",
9        image: input.getFileDataUrl() ,
10       ],
11     } : [];
12    const userInst = {role: "user", content: input.prompt}
13    return {
14      model: this.xai(this.model),
15      system: input.instructions.join("\n"),
16      messages: [historyInst, userInst, fileInst],
17    };
18  }
```

Codeausschnitt 3.5.1: mapPromptInput() für Grok

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.

```
1 protected mapPromptInput(input: PromptInput) {
2     const systemInst = input.instructions.map((instruction) => {
3         return {
4             role: "model",
5             parts: [{text: instruction, thought: false}]
6         };
7     });
8     const historyInst = input.history.map((item) => {
9         return {
10            role: item.role == "user" ? "user" : "model",
11            parts: [
12                {
13                    text: item.content,
14                    thought: item.role == "assistant"
15                },
16            ];
17        };
18    });
19    const imageInst = input.file ? {
20        role: "user",
21        parts: [
22            {
23                type: "input_file",
24                inlineData: {
25                    data: input.getFileBase64Data(),
26                    mimeType: input.getFileMimeType(),
27                },
28            },
29        ],
30    } : [];
31    const userInst = {
32        role: "user",
33        parts: [{text: input.prompt}]
34    };
35    return {
36        model: this.model,
37        contents: [systemInst, historyInst, userInst, imageInst],
38    };
39}
```

Codeausschnitt 3.5.2: mapPromptInput() für Gemini

Die Gemini API erwartet bei Jedem Text Teil der Anfrage noch das Feld ‘thought’, welches angibt ob dieser Teil bereit von einer KI gedacht wurde.

3.5.3 Claude

```
1  protected mapPromptInput(input: PromptInput) {
2    const systemInst = input.instructions.map((instruction) => {
3      return {type: "text", text: instruction};
4    });
5    const historyInst = input.history.map((item) => {
6      return {role: item.role, content: item.content};
7    });
8    const imageInst = input.file ? {
9      role: "user",
10     content: [
11       type: "file",
12       source: {
13         type: 'base64',
14         data: input.getFileBase64Data(),
15         media_type: input.getFileMimeType(),
16       },
17     ],
18   } : [];
19   const userInst = {role: "user", content: input.prompt};
20   return {
21     model: this.model,
22     max_tokens: 15000,
23     system: systemInst,
24     messages: [historyInst, userInst, imageInst],
25   };
26 }
```

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 folgt aus:

Andere Anbieter wie Gemini oder Grok verfolgen dagegen einen leicht unterschiedlichen Ansatz und stellen das Streaming über eine separate API-Funktion bereit.

```
1 protected mapPromptInput(input: PromptInput, stream: boolean) {  
2     [...]  
3     return {  
4         model: this.model,  
5         stream: stream,  
6         system: systemInst,  
7         input: [historyInst, userInst, imageInst]  
8     };  
9 }
```

Codeauschnitt 3.6.1: mapPromptInput() für einen Stream

Je nach verwendetem Endpunkt wird entweder ein normaler Text generiert oder ein Stream zurückgegeben. Der Wechsel zwischen beiden Varianten lässt sich dadurch einfach implementieren:

```
1 async generateContent(input: any, stream: boolean) {  
2     if (stream) return streamText(input)  
3     return generateText(input);  
4 }
```

Codeauschnitt 3.6.2: generateContent() für einen Stream

Sobald die Antwort der KI eintrifft, wird zunächst überprüft, ob es sich tatsächlich um einen Stream handelt. Da JavaScript-basierte Streams typischerweise das Symbol `asyncIterator` implementieren, lässt sich dies zuverlässig über eine einfache Typprüfung feststellen:

```
1 protected isStream(obj: any): boolean {  
2     return obj && typeof obj[Symbol.asyncIterator] === "function";  
3 }
```

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.

```
1  protected async processStream(stream: any) {
2    for await (const chunk of stream) {
3      switch (chunk.type) {
4        case "response.output_text.delta":
5          // Text oder andere Änderungen
6          processDelta(chunk.delta);
7          break;
8
9        case "response.completed":
10         // Fertig gelesen
11         return;
12
13       case "response.error":
14       case "response.failed":
15         // Fehler
16         error(chunk.response.error.message);
17         return;
18
19       default:
20         break;
21     }
22   }
23 }
```

Codeausschnitt 3.6.4: processStream()

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

Erkennt das System jedoch, dass der Stream gerade einen Diagrammteil 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 von einander 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` jedeglich die aktuelle Uhrzeit.

Das erste eigentliche Daten Event ist nun das `delta` Event bis 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 mitgleteil, 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 erzeugt wird, sendet er ein `diagram-start` Event welches dem Client mitteilt welche modell verwendet wird. Sobald das Diagramm fertig erstellt wurde, wird auch ein `diagram-end` Event gesendet, welches dem Client mitteilt, dass das Diagramm fertig erzeugt wurde, sowie ein `diagram` Event gesendet, 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, genau so, wie die Antwort gewesen wäre, falls nicht an den Client gestreamt worden wäre. Dies ist da um mögliche 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.

```
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 version=\“1.0\” encoding=\“UTF-8\”?>. . . </bpmn:definitions>”
data: }
```

```
event: save
data: success
```

3.7 Schema-Constraining

3.8 Evtl. Sampling

3.9 Evtl. Reflective Prompting

4 Performanzanalyse

4.1 Qualität

4.2 Geschwindigkeit

Um die Geschwindigkeit verschiedener 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, ob sich die Generierungszeiten eines LLM Anbieters unterscheiden, wenn man stream 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

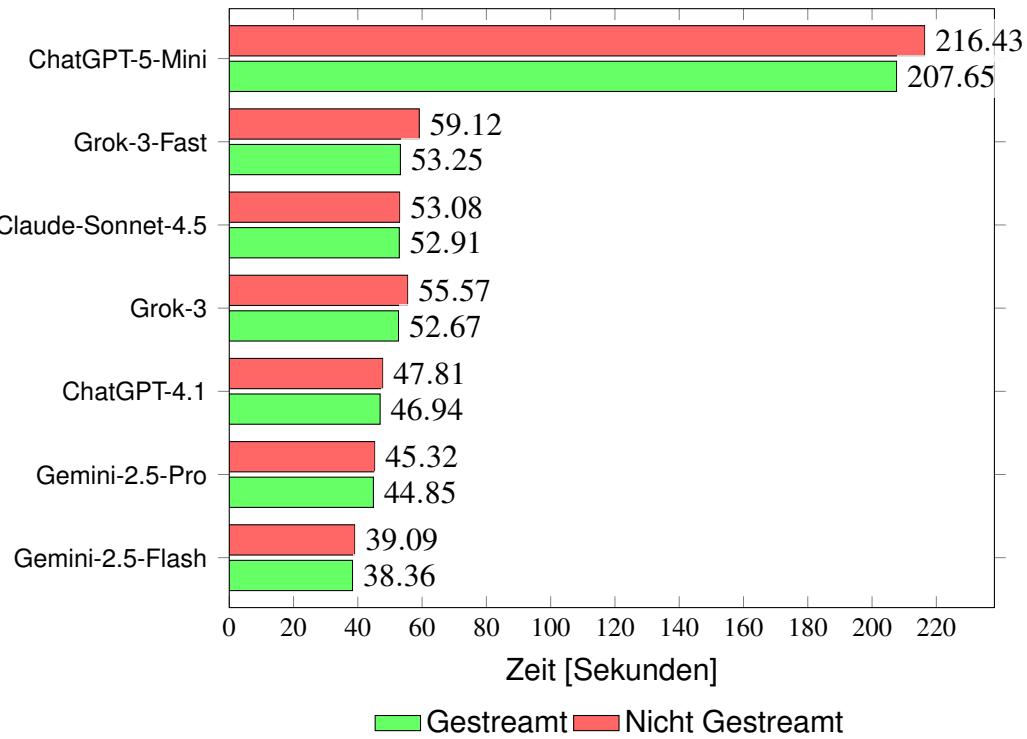


Abbildung 4.1: Zeitperformanzvergleich Gestreamt vs Nicht Gestreamt

Aus diesen Daten 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 Streamings nicht der Variante Ohne Streamings unterliegt und für weitere Tests kann die Variante des Streamings verwendet werden während die Nicht Streaming Variante vernachlässigt wird.

Als nächstes soll nun untersucht werden welcher der zwei implementierten Formate JSON und XML sich Zeitlich besser verhält. In Abbildung 4.2 wird für das Promptbeispiel auf Seite 32 dieses Verhalten getestet.

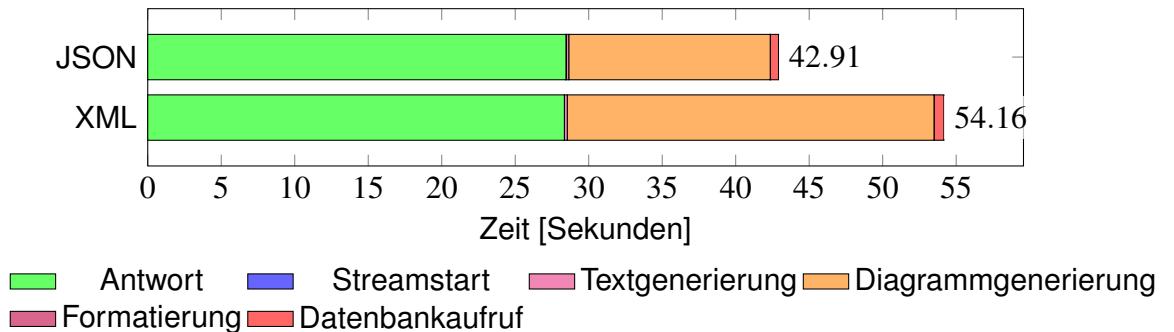


Abbildung 4.2: Zeitperformanzvergleich JSON vs XML bei Gemini 2.5 Pro

Man erkennt 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 11.242 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 folgender Prompt 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

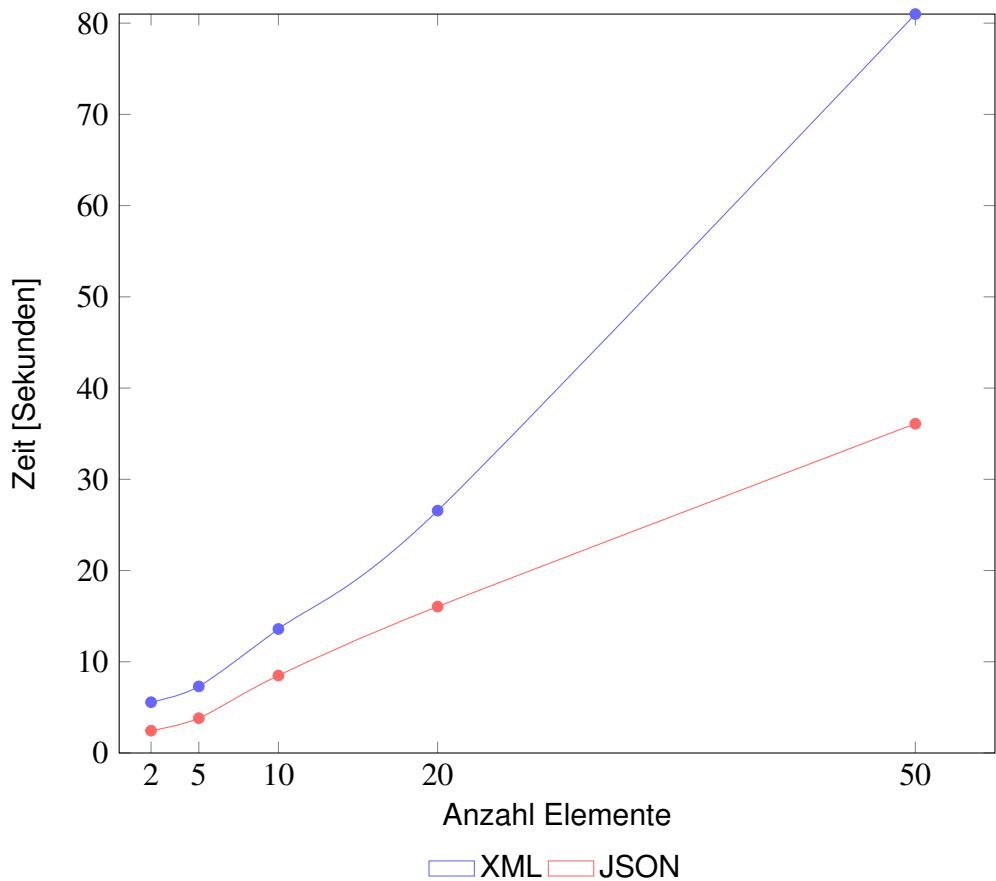


Abbildung 4.3: Zeitperformanzvergleich JSON vs XML bei Gemini 2.5 Pro nach Anzahl der Elemente (Nur Diagramm)

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 performanzkritische Anwendungsfälle, insbesondere bei großen oder komplexen Diagrammen, erhebliche Vorteile bietet.

Weitergehend soll nun untersucht werden welche Modelle sich für eine Zeiteffiziente Generierung eignen. Dafür werden in Abbildung 4.4 einige gängige Modelle getestet.

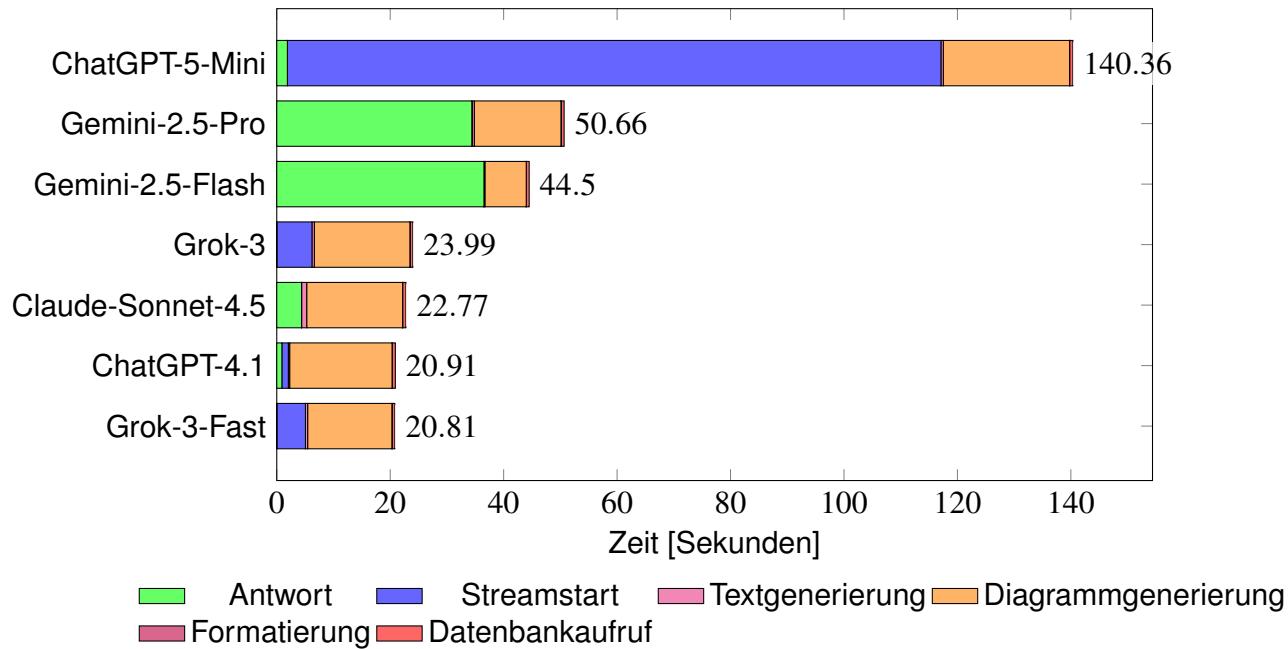


Abbildung 4.4: Zeitperformanzvergleich verschiedener Modelle

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 anzuhören 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

5 Verwandte Arbeiten

Hier sind verwandte Arbeiten

6 Fazit

6.1 Zusammenfassung

6.2 Ausblick

Notizen

Gefundene Paper

A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT

- **Autoren:** Jules White et al. (2023)
- **Inhalt:** Katalogisierung von Prompting-Mustern zur systematischen Wiederverwendung.
- **Link:** arxiv.org/abs/2302.11382

Chain-of-Thought Prompting Elicits Reasoning in Large Language Models

- **Autoren:** Jason Wei et al. (2022)
- **Inhalt:** Einführung der „Chain-of-Thought“-Technik zur Verbesserung logischer Schlussfolgerungen.
- **Link:** arxiv.org/abs/2201.11903

The Prompt Report: A Systematic Survey of Prompting Techniques

- **Autoren:** Schulhoff et al. (2024)
- **Inhalt:** Systematische Übersicht mit 58 Techniken, Kategorisierung und Beispiele.
- **Link:** arxiv.org/abs/2406.06608

A Systematic Survey of Prompt Engineering in Large Language Models

- **Autoren:** Sahoo et al. (2024)
- **Inhalt:** Überblick über Methoden, Anwendungen, Modelle und Herausforderungen.
- **Link:** arxiv.org/abs/2402.07927

Training Language Models to Follow Instructions with Human Feedback (InstructGPT)

- **Autoren:** Ouyang et al. (OpenAI, 2022)
- **Inhalt:** Einsatz von RLHF zur Verbesserung der Instruktionsbefolgung durch Sprachmodelle.
- **Link:** arxiv.org/abs/2203.02155

Vorlagenbeispiele

Diese kleine Einleitung soll dem Nutzer helfen selbst die eigene Arbeit mit L^AT_EX zu schreiben. Sie enthält Beispiele zu den wichtigsten Themen .

 Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Dokumentgliederung

Für diese Arbeit verwendet man folgende LaTeX-Kommados zur Strukturierung:

```
\chapter{Einleitung}
\section{Dokumentgliederung}
\subsection{}
\subsubsection{}
```

Allerdings sollte man sich überlegen, ob man wirklich bis zur Stufe subsubsection Überschriften benötigt.

Illustrationen

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Bilder und Abbildungen

Auch in einer wissenschaftlichen Arbeit können Bilder und Abbildungen zur Veranschaulichung und zur Illustration sachlicher Inhalte integriert und eingefügt werden. Für Fotografien und Bilder unterstützt PDF-L^AT_EX direkt jpg und png. Ansonsten empfiehlt es sich, Vektorgrafiken zu verwenden und diese als pdf zu speichern. Sollte ein Bild einmal von zu viel weißem Raum umgeben sein, kann man mit dem Werkzeug pdfcrop das Bild automatisch zuschneiden.

Mit Hilfe eines Labels `\label{fig:bild1}` kann man sich dann im fortlaufenden Text mittels eines Querverweises auf diese Grafik beziehen: `\ref{fig:bild1}`.

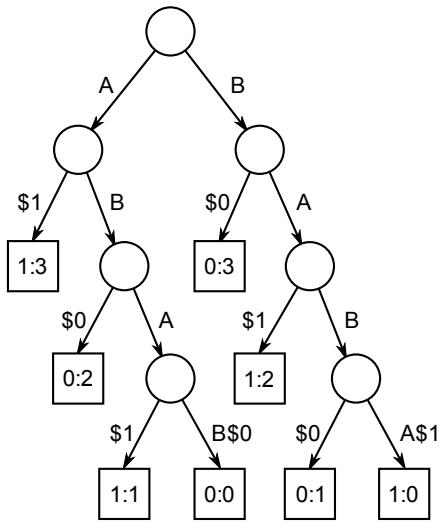


Abbildung 6.1: Beschreibung/Beschriftung des Bilds

An der Stelle des ref-Kommandos platziert LaTeX die Nummer der Abbildung: „siehe Abbildung 6.1“.

Tabellen

Seite 44, Abschnitt 6.2, enthält Beispieltabelle 6.1. In vielen \LaTeX -Büchern finden sich gute Anleitungen zum Erstellen von Tabellen. Komplexere Tabellen können sinnvollerweise in Excel oder einer anderen Tabellenkalulation vorgefertigt und mit einem Umwandlungsprogramm oder -werkzeug in LaTeX-Quellcode konvertiert werden.

| A | B | C |
|---|---|---|
| x | x | x |
| x | x | x |

Tabelle 6.1: Eine kleine Beispieltabelle

Formeln

Mathematische Formeln lassen sich in der Umgebung `math` erzeugen. Die Kurz-Schreibweise lautet `\(a^2+b^2=c^2 \)`; hierbei steht die Formel dann im laufenden Text: $a^2 + b^2 = c^2$. Die kürzeste Form ist mit zwei \$ um die Formel, z.B. so: Wasser ist H₂O. `H$_2$O`

Mit der Schreibweise `\[y=x^2 \]` wird die Formel mittig in einer eigenen Zeile gesetzt, z.B.

$$y = x^2$$

Formeln in der Umgebung `equation` werden mittig in einer eigenen Zeile gesetzt und fortlaufend nummeriert:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \quad (6.1)$$

Wenn wir z.B. über die beliebte Mitternachtsformel (Gleichung 6.1) Details im umliegenden Text schreiben wollen, lässt sich diese wie ein Bild oder eine Tabelle referenzieren, sofern man ihr ein Label zugewiesen hat..

Programmier-Code

Mehrzeiliger Programmier- und Quellcode kann mit `verbatim` in einer Umgebung gesetzt werden:

Dieser Text steht in einer `verbatim`-Umgebung und wird daher in Schreibmaschinenschrift geschrieben.

LaTeX-Kommandos, z.B. `\includegraphics[width=.6\textwidth]{bild.jpg}` werden nicht interpretiert, sondern "verbatim" ausgegeben.

Schöner und professioneller lässt sich Programmier-Code mit dem `listings`-Paket, eingeben, formatieren und ausgeben. Dazu kann man in der Präambel die Sprache angeben, in der die Quellcodes geschrieben sind.

```
1 public class Hello {  
2     public static void main(String[] args) {  
3         System.out.println("Hello World");  
4     }  
5 }
```

Innerhalb einer Zeile gibt man Wörter am Besten als \verb## an, dabei erwartet LATEX zweimal das gleiche Zeichen als Begrenzer. Im Beispiel ist dies die Raute #, man kann aber auch jedes andere Zeichen nehmen, z.B. das Plus +.

Text

Textteile können bei Bedarf mit dem Befehl \emph{} hervorgehoben werden. Falls in einem Satz ein Punkt vorkommt, macht man danach kein Leerzeichen sondern eine Tilde (z.~B.~so!), denn dann fügt LATEX den korrekten Abstand ein, z. B. so!

In der Präambel der vorliegenden tex-Datei gibt es den Befehl hypenation, der zur Silbentrennung da ist. LATEX verfügt zwar über eine eingebaute Silbentrennung, die jedoch bei manchen Wörtern falsch trennt. Damit diese Wörter korrekt getrennt werden, gibt man sie dann mit dem Befehl in der Präambel an¹.

Fußnoten werden mit dem Befehl footnote mitten in den fortlaufenden Text eingefügt.²

In wissenschaftlichen Arbeiten muss man des öfteren andere Arbeiten zitieren. Dazu nutzt man die Stiloptionen und Zitierbefehle des Pakets biblatex, z. B. numeric (=Standard-Stil) oder verbose resp. \cite{name} oder \autocite{name}. In eckigen Klammern kann man noch die Seitenzahl angeben, falls notwendig. Der Name ist ein Schlüssel aus der Datei bibliography.bib. Falls einmal ein Werk nur indirekt zu einem Teil der Arbeit beigetragen hat, kann man es auch mit nocite angeben, dann landet es in der Literaturliste, ohne dass es im Text ausdrücklich zitiert wird.

¹Das Wort Silbentrennung ist hier das Beispiel

²Wie man schon im vorherigen Absatz sehen konnte.

Weiterführendes

Zum Schluss sei auf die Vielzahl an Büchern zu \LaTeX verwiesen. In jeder Bibliothek wird sich eine Einführung finden, in der dann weitere Themen wie mathematische Formeln, Aufbau von Briefen und viele nützliche Erweiterungen besprochen werden.

A Quelltexte

In diesem Anhang sind einige wichtige Quelltexte aufgeführt.

```
1 function greet(name: string) : void {  
2     console.log(`Hello, ${name}!`);  
3 }  
4 greet("World");
```

Literatur

- [1] Jörg Knappen. *Schnell ans Ziel mit LATEX* 2e. 3., überarb. Aufl. München: Oldenbourg, 2009.
- [2] Frank Mittelbach, Michel Goossens und Johannes Braams. *Der Latex-Begleiter*. 2., überarb. und erw. Aufl. ST - Scientific tools. München [u.a.]: Pearson Studium, 2005.
- [3] Joachim Schlosser. *Wissenschaftliche Arbeiten schreiben mit LATEX : Leitfäden für Einsteiger*. 5., überarb. Aufl. Frechen: mitp, 2014.
- [4] Thomas F. Sturm. *LATEX : Einführung in das Textsatzsystem*. 9., unveränd. Aufl. RRZN-Handbuch. Hannover [u.a.]: Regionales Rechenzentrum für Niedersachsen, RRZN, 2012.
- [5] Herbert Voß. *LaTeX Referenz*. 2., überarb. u. erw. Aufl. Berlin: Lehmanns Media, 2010.

Name: Philipp Letschka

Matrikelnummer: 1050994

Erklärung

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

Ulm, den

Philipp Letschka