Large Language Models as Software Components: A Taxonomy for LLM-Integrated Applications

Irene Weber

Kempten University of Applied Sciences, Germany
irene.weber@hs-kempten.de

Abstract

Large Language Models (LLMs) have become widely adopted recently. Research explores their use both as autonomous agents and as tools for software engineering. LLM-integrated applications, on the other hand, are software systems that leverage an LLM to perform tasks that would otherwise be impossible or require significant coding effort. While LLM-integrated application engineering is emerging as new discipline, its terminology, concepts and methods need to be established. This study provides a taxonomy for LLM-integrated applications, offering a framework for analyzing and describing these systems. It also demonstrates various ways to utilize LLMs in applications, as well as options for implementing such integrations.

Following established methods, we analyze a sample of recent LLM-integrated applications to identify relevant dimensions. We evaluate the taxonomy by applying it to additional cases. This review shows that applications integrate LLMs in numerous ways for various purposes. Frequently, they comprise multiple LLM integrations, which we term "LLM components". To gain a clear understanding of an application's architecture, we examine each LLM component separately. We identify thirteen dimensions along which to characterize an LLM component, including the LLM skills leveraged, the format of the output, and more. LLM-integrated applications are described as combinations of their LLM components. We suggest a concise representation using feature vectors for visualization.

The taxonomy is effective for describing LLM-integrated applications. It can contribute to theory building in the nascent field of LLM-integrated application engineering and aid in developing such systems. Researchers and practitioners explore numerous creative ways to leverage LLMs in applications. Though challenges persist, integrating LLMs may revolutionize the way software systems are built.

Keywords: large language model, LLM-integrated, taxonomy, copilot, architecture, AI agent, LLM component

1. Introduction

Large Language Models (LLMs) have significantly impacted various sectors of economy and society [47]. Due to their proficiency in text understanding, creative work, communication, knowledge work, and code writing, they have been adopted in numerous fields, such as medicine, law, marketing, education, human resources, etc.

Public discussions often focus on the ethical aspects and societal consequences of these systems [36, 39]. Meanwhile, research investigates Artificial General Intelligences and autonomous AI agents that can use services, data sources, and other tools, and collaborate to solve complex tasks [11, 62, 57, 21]. In addition, LLMs offer many opportunities to enhance software systems. They enable natural language interaction [59], automate complex tasks [19], and provide supportive collaboration, as seen with recent LLM-based assistant products often branded as "copilots"[1].

This paper addresses the potential of LLMs for software development by integrating their capabilities as components into software systems. This contrasts with current software engineering research, which views LLMs as tools for software development rather than as software components [14, 22], and with the considerable body of research examining LLMs as autonomous agents within multiagent systems [21].

Software systems that invoke an LLM and process its output are referred to as "LLM-integrated applications", "LLM-integrated systems", "LLM-based applications", etc. [32, 13, 57]. LLMs are versatile, multipurpose tools capable of providing functionalities that would otherwise be unfeasible or require substantial development efforts [15, 24]. By significantly expediting system development, they have the potential to revolutionize not only the way users interact with technology, but also the fundamental processes of software development.

LLM-integrated applications engineering is emerging as a research field. E.g., [10] proposes LLM Systems Engineering (LLM-SE) as a novel discipline, and [44, 8, 7] discuss experiences and challenges that developers of such systems encounter in practice.

This study develops a taxonomy that provides a structured framework for categorizing and analyzing LLM-integrated applications across various domains. To develop and evaluate the taxonomy, we collected a sample of LLM-integrated applications, concentrating on technical and industrial domains. These applications showcase a broad range of opportunities to leverage LLMs, often integrating LLMs in multiple ways for distinct purposes. In developing the taxonomy, we found that examining each of these integrations, termed "LLM components", separately is crucial for a clear understanding of an application's architecture.

The taxonomy adopts an original architectural perspective, focusing on how the application interacts with the LLM while abstracting from the specifics of application domains. For researchers, the taxonomy contributes to shape a common understanding and terminology, thus aiding theory building in this emerging domain [29, 50, 18]. For practitioners, the taxonomy provides inspiration for potential uses of LLMs in applications, presents design options, and helps identify challenges and approaches to address them.

Objectives. In this study, a taxonomy is understood as a set of dimensions divided into characteristics. The objective is to identify dimensions that are useful for categorizing the integration of LLMs in applications from an architectural perspective. To be most effective, the taxonomy should be easy to understand and apply, yet distinctive enough to uncover the essential aspects. Additionally, we aim to develop a visual representation tailored to the taxonomy's intended purposes.

Overview. The following section 2 provides background on LLMs and introduces relevant concepts. Section 3 presents an overview of related work. The study design adheres to a *Design Science Research* approach [46]. We apply established methods for taxonomy design [42, 48] as described in Section 4. This section also presents the sample of LLM-integrated applications used for this study. The developed taxonomy is presented, demonstrated and formally evaluated in section 5. In section 6, we discuss its usability and usefulness. Section 7 summarizes the contributions, addresses limitations, and concludes.

2. Large Language Models

2.1. Background

State-of-the-art LLMs such as GPT-3.5, GPT-4, Llama, PALM2, etc., are artificial neural networks consisting of neurons, i.e., very simple processing

[1]E.g., `https://docs.github.com/en/copilot`, `https://copilot.cloud.microsoft/en-us/copilot-excel`, `https://www.salesforce.com/einsteincopilot`

units, that are organized in layers and connected by weighted links. Training a neural network means adapting these weights such that the neural network shows a certain desired behavior. Specifically, an LLM is trained to predict the likelihoods of pieces of text termed, *tokens*, to occur as continuations of a given text presented as input to the LLM. This input is referred to as *prompt*. The prompt combined with the produced output constitutes the *context* of an LLM. It may comprise more than 100k tokens in state-of-the-art LLMs[2]. Still, its length is limited and determines the maximum size of prompts and outputs that an LLM is capable of processing and generating at a time.

Training of an LLM optimizes its parameters such that its computed likelihoods align with real text examples. The training data is a vast body of text snippets extracted, processed, and curated from sources such as Wikipedia, Github code repositories, common websites, books, or news archives. An LLM trained on massive examples is termed a *foundation model* or *pre-trained model*. During training, an LLM not only learns to produce correct language but also absorbs and stores information and factual knowledge. However, it is well known that LLMs frequently pick up biases, leading to ethical problems. They may also produce factually incorrect outputs that sound plausible and convincing, termed *hallucinations*.

Recent findings show that LLMs can be applied to a wide range of tasks by appropriately formulating prompts. Different prompt patterns succeed in different tasks. Basic approaches rely on instructing the LLM to solve a task described or explained in the prompt. In *few-shot* prompting (also known as *few-shot* learning), the prompt is augmented with example input-output pairs illustrating how to solve the task, e.g., the requested output format. The number of examples can vary. Prompting with one example is called *one-shot* prompting, while prompting without any examples is called *zero-shot* prompting. *One-shot* and *few-shot* prompting fall under the broader category of *in-context learning*. Prompt patterns such as *chain-of-thought* and *thinking-aloud* aim to elicit advanced reasoning capabilities from LLMs.

[2]`https://platform.openai.com/docs/models`

As effective prompts are crucial for unlocking the diverse capabilities of an LLM, the discipline of prompt engineering is evolving, focusing on the systematic design and management of prompts [66, 9, 53, 31].

2.2. Definitions

Invoking an LLM results in an input-processing-output sequence: Upon receiving a prompt, the LLM processes it and generates an output. We refer to an individual sequence of input-processing-output performed by the LLM as *LLM invocation*, and define an *LLM-integrated application* as a system in which the software generates the prompt for the LLM and processes its output. The concept of an application is broad, encompassing service-oriented architectures and systems with components loosely coupled via API calls.

Given an LLM's versatility, an application can utilize it for different tasks, each demanding a specific approach to create the prompt and handle the result. This paper defines a particular software component that accomplishes this as an *LLM-based software component* or, simply, *LLM component*. An LLM-integrated application can comprise several LLM components. The study develops a taxonomy for LLM components. LLM-integrated applications are described as combinations of their LLM components.

3. Related Work

With the recent progress in generative AI and LLMs, the interest in these techniques has increased, and numerous surveys have been published, providing an extensive overview of technical aspects of LLMs [72], reviewing LLMs as tools for software engineering [22], and discussing the technical challenges of applying LLMs across various fields [25]. Further studies address the regulatory and ethical aspects of Generative AI and ChatGPT, with a particular focus on AI-human collaboration [41], and Augmented Language Models (ALMs), which are LLMs that enhance their capabilities by querying tools such as APIs, databases, and web search engines [38].

Taxomonies related to LLMs include a taxonomy for prompts designed to solve complex tasks [49] and a taxonomy of methods for cost-effectively invoking a remote LLM [60]. A comparative analysis of studies on applications of ChatGPT is provided by [27], whereas LLMs are compared based on their application domains and the tasks they solve in [20]. Most closely related to the taxonomy developed here is a taxonomy for LLM-powered multiagent architectures [21] which focuses on autonomous agents with less technical detail. Taxonomies of applications of AI in enterprises [48] and applications of generative AI, including but not limited to LLMs [52], are developed using methods similar to those in our study.

Several taxonomies in the field of conversational agents and task-oriented dialog (TOD) systems address system architecture [1, 40, 12, 3]. However, they omit detailed coverage of the integration of generative language models.

4. Methods

We constructed the taxonomy following established guidelines [42, 48, 29], drawing from a sample of LLM-integrated applications. These applications are detailed in section 4.1.

4.1. Development

Taxonomy. We derived an initial taxonomy from the standard architecture of conversational assistants described in [3], guided by the idea that conversational assistants are essentially “chatbots with tools”, i.e., language-operated user interfaces that interact with external systems. This approach proved unsuccessful. The second version was based on the classical three-tier software architecture, and then extended over several development cycles. By repeatedly applying the evolving taxonomy to the example instances, we identified dimensions and characteristics using an “empirical-to-conceptual” approach. When new dimensions emerged, additional characteristics were derived in a “conceptual-to-empirical” manner. After five major refinement cycles, the set of dimensions and characteristics solidified. In the subsequent evaluation phase, we applied the taxonomy to a new set of example instances that were not considered while constructing the taxonomy. As the dimensions and characteristics remained stable, the taxonomy was considered complete. In the final phase, we refined the wording and visual format of the taxonomy.

Visualization. Developing a taxonomy involves creating a representation that effectively supports its intended purpose [29]. Taxonomies can be represented in various formats, with morphological boxes [54, 55] or radar charts [21] being well-established approaches. We evaluated morphological boxes, because they effectively position categorized instances within the design space. However, we found that they make it difficult to perceive a group of categorized instances as a whole since they occupy a large display area. This drawback is significant for our purposes, as LLM-integrated applications often comprise multiple LLM components. Therefore, we developed a more condensed visualization of the taxonomy based on feature vectors.

Example instances. We searched for instances of LLM-integrated applications for taxonomy development that should meet the following criteria:

- The application aims for real-world use rather than focusing on research only (such as testbeds for experiments or proofs-of-concept). It demonstrates efforts towards practical usability and addresses challenges encountered in real-world scenarios.
- The application's architecture, particularly its LLM components, is described in sufficient detail for analysis.
- The sample of instances covers a diverse range of architectures.
- The example instances are situated within industrial or technical domains, as we aim to focus on LLM-integrated applications beyond well-known fields like law, medicine, marketing, human resources, and education.

The search revealed a predominance of theoretical research on LLM-integrated applications while papers focusing on practically applied systems were scarce. Searching non-scientific websites uncovered commercially advertised AI-powered applications, but their internal workings were typically undisclosed, and reliable evaluations were lacking. Furthermore, the heterogeneous terminology and concepts in this emerging field make a comprehensive formal literature search unfeasible. Instead, by repeatedly searching Google Scholar and non-scientific websites using terms "LLM-integrated applications", "LLM-powered applications", "LLM-enhanced system", "LLM" and "tools", along similar variants, we selected six suitable instances. Some of them integrate LLMs in multiple ways, totaling eleven distinct LLM components.

For a thorough evaluation, we selected new instances using relaxed criteria, including those intended for research. Additionally, we included a real-world example lacking explicit documentation to broaden the diversity of our sample and assess the taxonomy's coverage. Within the five selected instances, we identified ten LLM components.

4.2. Sample of LLM-integrated applications

Table 1 gives an overview of the sample. Names of applications and LLM components are uniformly written as one CamelCase word and typeset in small caps, deviating from the format chosen by the respective authors.

Honeycomb. Honeycomb is an observability platform collecting data from software applications in distributed environments for monitoring. Users define queries to retrieve information about the observed software systems through Honeycomb's Query Builder UI. The recently added LLM-based QueryAssistant allows users to articulate inquiries in plain English, such as "slow endpoints by status code" or "which service has the highest latency?" The QueryAssistant converts these into queries in Honeycomb's format, which users can execute and manually refine [7, 8].

LowCode. LowCode is a web-based application consisting of a prompt-definition section and a dialogue section. The prompt-definition section supports the design of prompts for complex tasks, such as composing extensive essays, writing resumes for job applications or acting as a hotel service chatbot [5]. In the dialogue section, users converse with an LLM to complete the complex task based on the defined prompt.

LowCode comprises two LLM components termed Planning and Executing. Planning operates in the prompt-definition section, where a user roughly describes a complex task, and Planning designs a workflow for solving it. The prompt-definition section offers a low-code development environment where the LLM-generated workflow is visualized as a graphical flowchart, allowing a user to edit and adjust the logic of the flow and the contents of its steps. For instance, in essay-writing scenarios, this involves inserting additional sections, rearranging sections, and refining the contents of sections. Once approved by the user, LowCode translates the modified workflow back into natural language and incorporates it into a prompt for Executing. In the dialogue section, users converse in interactive, multi-turn dialogues with Executing. As defined in the prompt, it acts as an assistant for tasks such as writing an essay or resume, or as a hotel service chatbot. While the idea of the LLM planning a workflow might suggest using the LLM for application control, LowCode Planning actually serves as a prompt generator that supports developing prompts for complex tasks.

MyCrunchGpt. MyCrunchGpt acts as an expert system within the engineering domain, specifically for airfoil design and calculations in fluid mechanics. These tasks require complex workflows comprising several steps such as preparing data, parameterizing tools, and evaluating results, using various software systems and tools. The aim of MyCrunchGpt is to facilitate the definition of these workflows and automate their execution [28].

MyCrunchGpt offers a web interface featuring a dialogue window for inputting commands in plain English, along with separate windows displaying the

Table 1: Example instances selected for development (top 6) and evaluation (bottom 5)

Application	References	LLM components
Honeycomb	[7, 8]	QueryAssistant
LowCode	[5],[35]	Planning, Executing
MyCrunchGpt	[28]	DesignAssistant, SettingsEditor, DomainExpert
MatrixProduction	[69]	Manager, Operator
WorkplaceRobot	[37]	TaskPlanning
AutoDroid	[64]	TaskExecutor, MemoryGenerator
ProgPrompt	[51]	ActionPlanning, ScenarioFeedback
FactoryAssistants	[26]	QuestionAnswering
SgpTod	[71]	DstPrompter, PolicyPrompter
TruckPlatoon	[70]	Reporting
ExcelCopilot	[16, 44]	ActionExecutor, Advisor, IntentDetector, Explainer

output and results of software tools invoked by MyCrunchGpt in the backend. MyCrunchGpt relies on predefined workflows, not supporting deviations or cycles. By appending a specific instruction to the dialogue history in the prompt for each step of the workflow, it uses the LLM as a smart parser to extract parameters for APIs and backend tools from user input. APIs and tools are called in the predefined order [28, p. 56].

MyCrunchGpt is still in development. The paper [28] explains the domain as well as the integration of the LLM, but does not fully detail the implementation of the latter. Still, MyCrunchGpt illustrates innovative applications of an LLM in a technical domain. We categorize three LLM components solving tasks within MyCrunchGpt: a DesignAssistant guiding users through workflows and requesting parameters for function and API calls; a SettingsEditor updating a JSON file with settings for a backend software tool; and a DomainExpert which helps evaluating results by comparing them to related results, e.g., existing airfoil designs, which it derives from its trained knowledge.

MatrixProduction. MatrixProduction employs an LLM for controlling a matrix production system [69]. While in a classical line production setup, workstations are arranged linearly and the manufacturing steps follow a fixed sequence, matrix production is oriented towards greater flexibility. Autonomous transport vehicles carry materials and intermediate products to workstations, termed automation modules, each offering a spectrum of manufacturing skills that it can contribute to the production process. Compared to line production, matrix production is highly adaptable and can manufacture a variety of personalized products with full automation. This requires intelligent production management to (a) create workplans that orchestrate and schedule the automation modules' skills, and (b) program the involved automation modules such that they execute the required processing steps.

MatrixProduction incorporates two LLM components: Manager creates workplans as sequences of skills (a), while Operator generates programs for the involved automation modules (b).

MatrixProduction prompts Manager and Operator to provide textual explanations in addition to the required sequences of skills or automation module programs. The LLM output is processed by a parser before being used to control the physical systems. Manager relies on built-in production-specific knowledge of the LLM such as "a hole is produced by drilling".

Noteworthy in this approach is its tight integration into the system landscape of Industry 4.0. The *few-shot* Manager and Operator prompts are generated automatically using *Asset Administration Shells*, which are standardized, technologyindependent data repositories storing digital twins of manufacturing assets for use in Industry 4.0 [2].

WorkplaceRobot. An experimental robot system is enhanced with LLM-based task planning in [37]. The robot operates in a workplace environment featuring a desk and several objects. It has previously been trained to execute basic operations expressed in natural language such as "open the drawer" or "take the pink object and place it in the drawer". LLM-based task planning enables the robot to perform more complex orders like "tidy up the work area and turn off all the lights". To this end, an LLM is prompted to generate a sequence of basic operations that accomplish the complex order.

Although the robot expects operations phrased in natural language, the LLM is prompted with a Python coding task. For instance, the basic operation "turn on the green light" corresponds to a Python command `push_button('green')`. The prompt for the LLM includes several examples each consisting of a description of an environment state, a complex order formatted as a comment, and a sequence of Python robot commands that accomplish the complex order. When invoking the LLM to generate the Python program for a new order, the prompt is augmented with a description of the environment's current state and the new order as a comment.

The Python code produced by the LLM is translated back to a sequence of basic operations in natural language. When the robot executes these operations, there is no feedback about successful completion. Rather, the system assumes that all basic operations require a fixed number of timesteps to complete.

AutoDroid. The goal of mobile task automation is hands-free user interaction for smartphones through voice commands. AutoDroid is a voice control system for smartphones that can automatically execute complex orders such as "remind me to do laundry on May 11th" or "delete the last photo I took" [64, 65].

Such complex orders are fulfilled by performing sequences of basic operations in an Android app, such as "scroll down, then press button x" in the calendar app. AutoDroid employs an LLM component TaskExecutor to plan these sequences of operations. The challenge is that the next operation to execute depends on the current state of the Android app which continuously changes as the app is operated. AutoDroid solves this by invoking the TaskExecutor repeatedly after each app operation with the prompt comprising the updated state of the Graphical User Interface (GUI) along with the user's complex order.

Before executing irrevocable operations, such as permanently deleting data or calling a contact, AutoDroid prompts the user to confirm or adjust the operation. TaskExecutor is instructed to include a "confirmation needed" hint in its output for such operations.

The prompt for TaskExecutor comprises an extract from a knowledge base which is built automatically in an offline learning phase as follows: In a first step, a "UI Automator" (which is not an LLM component) automatically and randomly operates the GUI elements of an Android app to generate a UI Transition Graph (UTG). The UTG has GUI states as nodes and the possible transitions between GUI states as edges. As next steps, AutoDroid invokes two LLM components referred to as MemoryGenerators to analyze the UTG.

The first MemoryGenerator is prompted repeatedly for each GUI state in the UTG. Its task is to explain the functionality of the GUI elements. Besides instructions and examples of the table format desired as output, its prompt includes an HTML representation of the GUI state, the GUI actions preceding this state, and the GUI element operated next. Its output consists of tuples explaining the functionality of a GUI element by naming the derived functionality (e.g., "delete all the events in the calendar app") and the GUI states and GUI element actions involved. Similarly, the second MemoryGenerator is prompted to output a table listing GUI states and explanations of their functions. These tables constitute AutoDroid's knowledge base.

ProgPrompt. ProgPrompt [51] is an approach to LLM-based robot task planning similar to

WorkplaceRobot. Its robot is controlled by Python code and works in a real and a simulated household environment.

ProgPrompt comprises two LLM components. ActionPlanning generates Python scripts for tasks such as "microwave salmon" using basic operations like `grab('salmon')`, `open('microwave')`, and `putin('salmon', 'microwave')`, notably without considering the current state of the environment. To establish a feedback loop with the environment, ActionPlanning adds `assert` statements. These statements verify the preconditions of basic operations and trigger remedial actions when preconditions are not met. For instance, a script for "microwave salmon" comprises the following code fragment:

```
if assert('microwave' is 'opened')
   else:  open('microwave')
putin('salmon', 'microwave')
```

When operating in the simulated environment, ProgPrompt can verify an `assert` statement through its second LLM component, ScenarioFeedback. Prompted with the current state of the environment and the `assert` statement, ScenarioFeedback evaluates it and outputs `True` or `False`.

FactoryAssistants. FactoryAssistants advise workers on troubleshooting production line issues in two manufacturing domains: detergent production and textile production [26]. The assistants leverage domain knowledge from FAQs and documented problem cases to answer user queries. The required domain knowledge is provided as a part of the prompt.

SgpTod. SgpTod employs an LLM to implement a chatbot, specifically, a task-oriented dialogue (TOD) system [71]. TOD systems are also known as conversational assistants. In contrast to open-domain dialogue (ODD) systems, which engage users in goalless conversations, they are designed for assisting users in specific tasks.

In general, TOD systems require the following components [3]: Natural Language Understanding (NLU), analyzing the user's input to classify intents and extract entities; Dialogue Management (DM) for deciding on a system action that is appropriate in a given dialogue state (e.g., ask for more information or invoke a hotel booking service); and Natural Language Generation (NLG) for producing a response that the TOD system can present to the user. Intent classification, also known as intent detection, matches free-text user input to one of several tasks a TOD system can perform (e.g., book a hotel). Entity extraction isolates situational values, called entities, from the user input (e.g., the town and the date of the hotel booking). The TOD system may require several dialogue turns to elicit all necessary entities from the user. In TOD research, the system's internal representation of the user's intentions and the entity values is commonly referred to as its "belief state". For example, in the restaurant search domain, the belief state may include attribute-value pairs like `cuisine:Indian` and `pricerange:medium`.

SgpTod is a multi-domain TOD system, concurrently handling multiple task domains found in standard TOD evaluation datasets, such as recommending restaurants or finding taxis. Similar to other experimental TOD systems [23], SgpTod accesses a database that stores information from the task domains, such as available hotels and restaurants.

SgpTod comprises two LLM components, called DstPrompter and PolicyPrompter, that are both invoked in every dialogue turn between SgpTod and the user. The DstPrompter handles the NLU aspect, analyzing the user's input and populating the system's belief state. It outputs is an SQL query suited to extract the database entries that match the current belief state. Upon retrieving the database entries, SgpTod invokes its PolicyPrompter which covers both DM and NLG. Prompted with the dialogue history and the database entries retrieved, it produces a two-part output: a natural language response for NLG and a system action for DM.

TruckPlatoon. The concept of truck platooning means that trucks travel closely together for better fuel efficiency and traffic flow. TruckPlatoon comprises an algorithmic control loop which autonomously maintains a consistent distance between trucks. It invokes an LLM to generate natural-language reports on the platoon's performance and stability from measurements tracked by the control algorithm, providing easily understandable information for engineers involved in monitoring and optimizing the truck platooning system.

ExcelCopilot. ExcelCopilot is an example of a recent trend where software companies integrate LLM-based assistants, often termed "copilots", into their products [44]. These copilots not only provide textual guidance but also perform actions within the software environment, constituting a distinctive type of LLM-integrated application. We chose ExcelCopilot as an example for evaluating our taxonomy. Since its implementation is undisclosed, we infer its architecture from indirect sources, including a screencast and a report on insights and experiences from copilot developers [16, 44]. This inferred architecture may deviate from the actual implementation.

ExcelCopilot is accessible in a task bar alongside the Excel worksheet. It features buttons with context-dependent suggestions of actions and a text box for users to type in commands in natural language. ExcelCopilot only works with data tables, so its initial suggestion is to convert the active worksheet's data into a data table. Copilot functions activate when a data table or part of it is selected. It then presents buttons for four top-level tasks: "add formula columns", "highlight", "sort and filter", and "analyze". The "analyze" button triggers the copilot to display more buttons, e.g., one that generates a pivot chart from the selected data. ExcelCopilot can also add a formula column to the data table and explain the formula in plain language.

When a user inputs a free-text command, ExcelCopilot may communicate its inability to fulfill it. This constantly occurs with commands requiring multiple steps, indicating that ExcelCopilot lacks a planning LLM component as seen in, for example, MatrixProduction. This observation, along with its mention in [44], suggests that ExcelCopilot employs an intent detection-skill routing architecture. This architecture includes an LLM component that maps free-text user commands to potential intents and then delegates to other LLM components tasked with generating actions to fulfill those intents. Accordingly, ExcelCopilot comprises several types of LLM components:

- Several distinct Action Executors generate code for specific application actions, such as creating a pivot table, designing a worksheet formula, inserting a diagram, and so on.
- An Advisor suggests meaningful next actions. Its outputs serve to derive button captions and prompts for ActionExecutors.
- When a user inputs a free-text command, the IntentDetector is invoked to determine and trigger a suitable ActionExecutor. The IntentDetector communicates its actions to users and informs them when it cannot devise a suitable action.
- The Explainer generates natural language explanations of formulae designed by ExcelCopilot. It is unclear whether under the hood, the ActionExecutor is generating both the formula and the explanation, or if two separate LLM components are being invoked. We assume the latter, i.e., that a separate Explainer LLM component exists.

While users interact repeatedly with ExcelCopilot, each interaction adheres to a single-turn pattern, with the user providing a command and ExcelCopilot executing it [44].

5. A Taxonomy for LLM Components and LLM-Integrated Applications

When developing the taxonomy, it emerged that analyzing an LLM-integrated application should begin with identifying and describing its distinct LLM components. Analyzing each LLM component separately helps capture details and provides a clear understanding of how the application utilizes LLM capabilities. The LLM-integrated application can then be described as a combination of the LLM components it employs.

Table 2: Dimensions and characteristics of the taxonomy. Codes of characteristics are printed in uppercase. "Meta" means "metadimension". "MuEx" means "mutual exclusiveness".

Meta	Dimension	Characteristics	MuEx
Invocation	*Interaction*	***A**pp*, ***C**ommand*, ***D**ialog*	enforced
	Frequency	***S**ingle*, ***I**terative*	yes
Function	*Logic*	*c**A**lculate*, ***C**ontrol*	yes
	UI	*none*, ***I**nput*, ***O**utput*, ***B**oth*	yes
	Data	*none*, ***R**ead*, ***W**rite*, ***B**oth*	yes
Prompt	*Instruction*	*none*, ***U**ser*, ***LLM***, ***P**rogram*	enforced
	State	*none*, ***U**ser*, ***LLM***, ***P**rogram*	enforced
	Task	*none*, ***U**ser*, ***LLM***, ***P**rogram*	yes
	Check	*none*, ***U**ser*, ***LLM***, ***P**rogram*	enforced
	Skills	*re**W**rite*, ***C**reate*, *con**V**erse*, ***I**nform*, ***R**eason*, ***P**lan*	no
Output	*Format*	***F**ree**T**ext*, ***I**tem*, ***C**ode*, ***S**tructure*	no
	Revision	*none*, ***U**ser*, ***LLM***, ***P**rogram*	enforced
	Consumer	***U**ser*, ***LLM***, ***P**rogram*, ***E**ngine*	enforced

5.1. Overview and demonstration

The taxonomy identifies 13 dimensions for LLM components, grouped into five metadimensions as shown in table 2. It comprises both dimensions with genuinely mutually exclusive characteristics and those with non-exclusive characteristics. For dimensions related to the technical integration of LLMs within applications, mutual exclusiveness is enforced. Given the open nature of software architecture, the integration of LLMs allows for significant diversity. In practice, LLM components may show multiple characteristics within these dimensions. Nonetheless, the taxonomy requires categorizing each component with a predominant characteristic, enforcing a necessary level of abstraction to effectively organize and structure the domain.

We applied the taxonomy to categorize each of the example instances described in section 4.2. The results are depicted in figure 1. The dimensions and their characteristics are detailed and illustrated with examples in section 5.2.

The taxonomy visualizes an LLM component by a feature vector comprising binary as well as multi-valued features. Non-mutually exclusive dimensions are represented by a set of binary features. The remaining dimensions are encoded as n-valued features where n denotes the number of characteristics. For compactness, we use one-letter codes of the characteristics as feature values in the visualizations. In table 2, these codes are printed in upper case in the respective characteristic's name.

A feature vector representing an LLM component is visualized in one line. For dimensions with non-mutually exclusive characteristics, all possible codes are listed, with the applicable ones marked. The remaining dimensions are represented by the code of the applicable characteristic, with the characteristic *none* shown as an empty cell. We shade feature values with different tones to support visual perception. LLM components within the same application are grouped together, visualizing an LLM-integrating application in a tabular format.

5.2. Dimensions and characteristics

5.2.1. Invocation dimensions

Two *Invocation* dimensions address the way the LLM is invoked within the application.

Interaction describes how the user interacts with the LLM with three characteristics:

App: Users never converse with the LLM directly in natural language, rather the application invokes the LLM automatically. E.g., users do not interact

	Invocation		*Function*			*Prompt*				*Skills*						*Out. Format*				*Output*	
	Interaction	*Frequency*	*Logic*	*UI*	*Data*	*Instruction*	*State*	*Task*	*Check*	*reWrite*	*Create*	*conVerse*	*Inform*	*Reason*	*Plan*	*FreeText*	*Item*	*Code*	*Structure*	*Revision*	*Consumer*
HONEYCOMB QUERYASSISTANT	C	S	A		R	P	P	U	P						P			C		P	E
LOWCODE PLANNING	C	S	A			P		U					I		P				S	U	L
LOWCODE EXECUTING	D	I	A	B		P	L	U			C	V	I			F					U
MYCRUNCHGPT DESIGNASSISTANT	D	I	A	B		P	P	U				V							S		E
MYCRUNCHGPT SETTINGSEDITOR	C	S	A			P	P	P		W								C			E
MYCRUNCHGPT DOMAINEXPERT	C	S	A			P	P	P					I			F					U
MATRIXPRODUCTION MANAGER	C	S	C	I		P	P	U					I		P	F			S		L
MATRIXPRODUCTION OPERATOR	A	S	C			P	P	L							P	F			S		E
WORKPLACEROBOT	C	S	C	I		P	P	U							P			C			E
AUTODROID EXECUTOR	C	I	C	I		P	L	U	P						P		I		S		E
AUTODROID MEMORYGENERATOR[2]	A	I	A			P	P	P						R					S		L
PROGPROMPT ACTIONPLANNING	C	S	C	I		P		U							P			C			E
PROGPROMPT SCENARIOFEEDBACK	A	I	C			P	P	L						R			I				E
FACTORYASSISTANT	D	S	A			P	P	U		W		V				F					U
SGPTOD DSTPROMPTER	D	S	A	I	R	P	P	U				V		R				C			E
SGPTOD POLICYPROMPTER	A	S	C	O		P	P	P						R		F	I				P
TRUCKPLATOON	A	S	A	O		P	P	P		W						F					U
EXCELCOPILOT ACTIONEXECUTOR*	A	S	A			P	P	L							P	F		C			E
EXCELCOPILOT ADVISOR	A	S	A			P	P	P						R		F			S		P
EXCELCOPILOT INTENTDETECTOR	C	S	C			P	P	U						R			I		S		P
EXCELCOPILOT EXPLAINER	A	S	A			P	P	P						R		F					U

Figure 1: Categorized example instances. See table 2 for a legend. $*, 2$: multiple LLM components.

directly with EXCELCOPILOT ACTIONEXECUTOR or with MATRIXPRODUCTION OPERATOR.

Command: Users input single natural language commands. E.g., users interact with AUTODROID TASKEXECUTOR through single natural language commands.

Dialog: Users engage in multi-turn dialogues with the LLM component to achieve a use goal. E.g., users repeatedly prompt LOWCODE EXECUTING or MYCRUNCHGPT DESIGNASSISTANT in multi-turn dialogues to obtain an essay or an airfoil design, respectively.

Frequency addresses how often the application invokes a specific LLM component to fulfill a goal:

Single: A single invocation of an LLM component is sufficient to produce the result. E.g., in MYCRUNCHGPT, the application internally invokes distinct LLM components once for each user input by injecting varying prompt instructions.

Iterative: The LLM component is invoked repeatedly to produce the result. E.g., AUTODROID TASKExecutor is invoked multiple times to fulfill a command with an updated environment description in the *State* prompt; LowCode Executing is repeatedly prompted by the user to achieve the use goal while the application updates the dialogue history.

5.2.2. Function dimensions

The *Function* dimensions are derived from the classical three-tier software architecture model which segregates an application into three distinct layers: presentation, logic and data [17]. The presentation layer implements the UI. On the input side, it allows users to enter data and commands that control the application. On the output side, it presents information and provides feedback on the execution of commands. The logic layer holds the code that directly realizes the core objectives and processes of an application such as processing data, performing calculations, and making decisions. The data layer of an application manages the reading and writing of data from and to persistent data storage. Due to its versatility, an LLM component can simultaneously implement functionality for all three layers. The taxonomy addresses this with three *Function* dimensions.

UI indicates whether an LLM component contributes significantly to the user interface of an application, avoiding the need to implement graphical UI controls or display elements:

none: No UI functionality is realized by the LLM. E.g., in ExcelCopilot, the LLM does not replace any UI elements.

Input: Input UI is (partially) implemented by the LLM. E.g., in MatrixProduction Manager, users input their order in natural language, obviating a product configuration GUI.

Output: Output UI is (partially) implemented by the LLM. E.g., in TruckPlatoon, the output generated by the LLM component can replace a data cockpit with gauges and other visuals displaying numerical data.

Both: Input and output UI are (partially) implemented by the LLM. E.g., in MyCrunchGpt, the DesignAssistant provides a convenient conversational interface for parameterization of APIs and tools and feedback on missing values, which otherwise might require a complex GUI.

Logic indicates whether the LLM component determines the control flow of the application. It discerns two characteristics:

cAlculate: The output does not significantly impact the control flow of the application, i.e., the output is processed like data. E.g., MyCrunchGpt SettingsEditor modifies a JSON file, replacing a programmed function; MyCrunchGpt DesignAssistant asks the user for parameters, but the sequence of calling APIs and tools follows a predefined workflow; the workflow computed by LowCode Planning is displayed without influencing the application's control flow.

Control: The output of the LLM is used for controlling the application. E.g., the plans generated by MatrixProduction Manager serve to schedule and activate production modules; the actions proposed by AutoDroid TaskExecutor are actually executed and determine how the control flow of the app proceeds.

Since an LLM invocation always computes a result, *cAlculate* is interpreted as "calculate only", making *cAlculate* and *Control* mutually exclusive.

Data addresses whether the LLM contributes to reading or writing persistent data:

none: The LLM does not contribute to reading or writing persistent data. This characteristic applies to most sample instances.

Read: The LLM is applied for reading from persistent data store. E.g., SgpTod DstPrompter generates SQL queries which the application executes; Honeycomb QueryAssistant devises analytical database queries.

Write and *Both*: No LLM component among the samples generates database queries for creating or updating persistent data.

5.2.3. Prompt-related dimensions

Integrating an LLM into an application poses specific requirements for prompts, such as the need for prompts to reliably elicit output in the requested form [68]. While a broad range of prompt patterns have been identified and investigated [66], there is still a lack of research on successful prompt patterns specifically for LLM-integrated applications, on which this taxonomy could build. Developing prompt taxonomies is a challenging research endeavor in itself [49] and is beyond the scope of this research. Therefore, the taxonomy does not define a dimension with specific prompt patterns as characteristics, but rather focuses on how the application generates the prompt for an LLM component from a technical perspective.

Prompts generally consist of several parts with distinct purposes, generated by different mechanisms. Although many authors explore the concepts, a common terminology has yet to be established. This is illustrated in table 3, showing terms from an ad-hoc selection of recent papers addressing prompt generation in applications. In the table, italics indicate that the authors refrain from introducing an abstract term and instead use a domain-specific description. The term "examples" indicates a *one-shot* or *few-shot* prompt pattern. The terms that are adopted for the taxonomy are underlined.

The taxonomy distinguishes three prompt parts referred to as *Prompt Instruction*, *Prompt State*, and *Prompt Task*. These parts can occur in any order, potentially interleaved, and some parts may be absent.

- *Instruction* is the part of a prompt that outlines how to solve the task. Defined during LLM component development, it remains static throughout an application's lifespan.
- *State* is the situation-dependent part of the prompt that is created dynamically every time the LLM is invoked. The taxonomy opts for the term *State* instead of "context" in order to avoid confusion with the "LLM context" as explained in section 2. The *State* may include the current dialogue history, an extract of a knowledge base needed specifically for the current LLM invocation, or a state or scene description, etc.
- *Task* is the part of the prompt conveying the task to solve in a specific invocation.

<u>*Prompt Instruction, State* and *Task*</u> describe the origins of the prompt parts by uniform characteristics:

none: The prompt part is not present. E.g., ProgPrompt ActionPlanning has no *State* prompt, nor does LowCode Planning (except the dialogue history when planning a subprocess). *Instruction* and *Task* prompt parts are present in all sample instances.

User: The user phrases the prompt part. E.g., the *Task* for ExcelCopilot IntentDetector or for LowCode Planning is phrased by the user. There are no sample instances where the user provides the *Instruction* or *State* prompt parts.

LLM: The prompt part is generated by an LLM. E.g., LowCode Planning generates the *State* for LowCode Executing and ExcelCopilot IntentDetector generates the *Task* for ExcelCopilot ActionExecutors.

Program: Application code generates the prompt part. E.g., AutoDroid programmatically generates the *State* and the *Task* parts for its MemoryGenerators in the knowledge base building phase.

The *Prompt Instruction* dimension is always generated by *Program*. While a user and possibly an LLM have defined this prompt part during application development, this falls outside the scope of this taxonomy. Therefore, the *Prompt Instruction* dimension is not discriminating and categorizes all cases as *Program*. It is retained in the taxonomy for completeness and better understandability.

<u>*Prompt Check*</u> describes whether the application employs a review mechanism to control and modify the prompt before invoking the LLM. The same characteristics as for the prompt parts are applicable:

none: The prompt is used without check.

User: The user checks and revises the prompt.

LLM: Another LLM component checks or revises the prompt.

Program: The application comprises code to check or revise the prompt. E.g., AutoDroid removes personal data, such as names, to ensure privacy before invoking the TaskExecutor; Honeycomb QueryAssistant incorporates a coded mechanism against prompt injection attacks.

Table 3: Terms used for prompt parts. Expressions specific to a domain are printed in italics, "examples" indicates a *one-shot* or *few-shot* prompt pattern. Terms adopted for the taxonomy are underlined.

Source	Instruction	State	Task
[72]	task description + examples		test instance
[34]	<u>instruction</u> prompt		data prompt
[32]	predefined prompt		user prompt
[45]	prompt template + examples	*DB schema*	user input question
[45]	examples		*SQL query result*
[37]	prompt context, i.e., examples	*environment* <u>*state*</u>*, scene description*	input <u>task</u> commands
[5]	education prompt	dialogue history	user input <u>task</u> prompt
[5]	education prompt	dialogue history + *provided workflow*	*(circumscribed)*
[69]	role and goal + <u>instruction</u> + examples	context	current <u>task</u>
[26]	predefined system <u>instruction</u> + *domain-specific information*	*query results from knowledge graph*	the user's request

Most example instances omit prompt checks. There are no examples where a *Check* is performed by a *User* or an *LLM*.

5.2.4. Skills dimensions

The *Skills* dimension captures the types of LLM capabilities that an application utilizes. It is designed as a dimension with six non-mutually exclusive characteristics.

<u>*Skills*</u> is decomposed into six specific capabilities:

reWrite: The LLM edits or transforms data or text, such as rephrasing, summarizing, reformatting, correcting, or replacing values. E.g., MyCrunchGpt SettingsEditor replaces values in JSON files; TruckPlatoon converts measurements into textual explanations.

Create: The LLM generates novel output. E.g., LowCode Executing generates substantial bodies of text for tasks like essay writing.

conVerse: The application relies on the LLM's capability to engage in purposeful dialogues with humans. E.g., MyCrunchGpt DesignAssistant asks users for missing parameters; SgpTod PolicyPrompter decides how to react to user inputs and formulates chatbot responses.

Inform: The application depends on knowledge that the LLM has acquired during its training, unlike applications that provide all necessary information within the prompt. E.g., MyCrunchGpt DomainExpert provides expert knowledge on airfoil designs; MatrixProduction relies on built-in knowledge of production processes, such as "a hole is produced by drilling"; LowCode Executing uses its learned knowledge for tasks like essay writing.

Reason: The LLM draws conclusions or makes logical inferences. E.g., FormulaExplainer in ExcelCopilot explains the effects of Excel functions in formulas; AutoDroid MemoryGenerators explain the effects of GUI elements in Android apps.

Plan: The LLM designs a detailed method or course of action to achieve a specific goal. E.g., AutoDroid TaskExecutor and WorkplaceRobot TaskPlanning devise action plans to achieve goals.

The *Plan* and *Reason* characteristics are interrelated, as planning also requires reasoning. The intended handling of these characteristics is to categorize an LLM component as *Plan* only and understand *Plan* as implicitly subsuming *Reason*.

The effectiveness of LLMs as components of software applications relies on their commonsense knowledge and their ability to correctly interpret and handle a broad variety of text inputs, including instructions, examples, and code. It is reasonable to assume that a fundamental capability, which might be termed *Unterstand*, is leveraged by every LLM component. As it is not distinctive, the taxonomy does not list it explicitly in the *Skills* dimension.

Applying this taxonomy dimension requires users to determine which skills are most relevant and worth highlighting in an LLM component. Given the versatility of LLMs, reducing the focus to few predominant skills is necessary to make categorizations distinctive and expressive.

5.2.5. Output-related dimensions

Output Format characterizes the format of the LLM's output. As an output may consist of several parts in diverse formats, this dimension is designed as non-mutually exclusive, same as the *Skills* dimension. It distinguishes four characteristics that are distinctive and well discernible:

FreeText: unstructured natural language text output. E.g., TruckPlatoon and MyCrunchGpt DomainExpert generate text output in natural language; MatrixProduction Manager and MatrixProduction Operator produce *FreeText* explanations complementing output in custom formats to be parsed by the application.

Item: a single text item from a predefined set of items, such as a class in a classification task. E.g., ProgPrompt ScenarioFeedback outputs either `True` or `False`.

Code: source code or other highly formalized output that the LLM has learned during its training, such as a programming language, XML, or JSON. E.g., AutoDroid TaskExecutor produces code to steer an Android app; MyCrunchGpt SettingsEditor outputs JSON.

Structure: structured, formalized output adhering to a custom format. E.g., LowCode Planning outputs text in a format that can be displayed as a flow chart; MatrixProduction Manager and Operator produce output in custom formats combined with *FreeText* explanations.

Output Revision indicates whether the application checks or revises the LLM-generated output before utilization. These characteristics and their interpretations mirror those in the *Prompt Check* dimension:

none: There is no revision of the LLM output.

User: The user revises the LLM output. E.g., the user improves the plan generated by LowCode Planning.

LLM: A further LLM component checks or revises the output of the LLM component under consideration.

Program: Programmed code checks or revises the LLM output. E.g., Honeycomb QueryAssistant corrects the query produced by the LLM before executing it [7].

There are no instances in the sample set where another LLM revises or checks the output of the LLM. Most sample applications do not check or revise the LLM's output, though several of them parse and transform it. The purpose of the *Output Revision* dimension is to indicate whether the application includes control or correction mechanisms, rather than just parsing it.

Output Consumer addresses the way of utilizing the LLM output:

User signifies that the LLM output is presented to a human user. E.g., the text output of TruckPlatoon is intended for humans, as well as the output of MyCrunchGPT DomainExpert.

LLM indicates that the output serves as a prompt part in a further LLM invocation. E.g., the knowledge base entries generated by an AutoDroid MemoryGenerator become part of the prompt for AutoDroid TaskExecutor; the plan output by LowCode Planning serves as a part of the prompt for LowCode Executing.

Program describes instances where the LLM output is consumed and processed further by a software component of the application. E.g., the output of MatrixProduction Manager is handled by software systems (including a Manufacturing Execution System) which use it to compute prompts for other LLM components.

Engine covers scenarios where the LLM output is intended for execution on a runtime engine. E.g., the SQL query generated by SgpTod DstPrompter is processed by a SQL interpreter; a part of the output of MatrixProduction Operator is executed by automation modules.

Although applications may parse and transform the LLM output before use, the *Output Consumer* dimension is meant to identify the ultimate consumer, such as an execution engine, rather than an intermediary parser or transformation code. When applications divide the LLM output into parts for different consumers, users applying the taxonomy need to determine which consumer is most relevant, since this dimension is designed to be mutually exclusive.

5.3. Evaluation

Figure 2 displays the number of occurrences of characteristics within the example instances. It must be noted, however, that these do not reflect actual frequencies, as similar LLM components within the same application are aggregated together, indicated by symbols $*$ and 2 in figure 1. Furthermore, ExcelCopilot likely includes occurrences of *Prompt Check* and *Output Revision* which are not counted due to insufficient system documentation.

We evaluate the taxonomy against commonly accepted quality criteria: comprehensiveness, robustness, conciseness, mutual exclusiveness, explanatory power, and extensibility [58, 42]. The taxonomy encompasses all example instances including those that were not considered during its development. This demonstrates **comprehensiveness**. As figure 1 shows, all example instances have unique categorizations, supporting the taxonomy's **robustness**. This not only indicates that the dimensions and characteristics are distinctive for the domain, but also highlights the wide variety possible in this field. **Conciseness** demands that the taxonomy uses the minimum number of dimensions and characteristics. The taxonomy gains conciseness by identifying relatively few and abstract characteristics within each dimension. However, it does not adhere to the related subcriterion that each characteristic must be present in at least one investigated instance [54]. Unoccupied characteristics are retained for dimensions whose characteristics were derived conceptually, specifically, for the *Prompt* dimensions, the *Output Revision* dimension, and the *Data Function* dimension, enhancing the taxonomy's ability to illustrate design options and inspire novel uses for LLM integrations in applications. Some dimensions are constructed in parallel, sharing common sets of characteristics. While this affects conciseness, it makes the taxonomy easier to understand and apply. As is often seen in taxonomy development [54], we deliberately waived the requirement for **mutual exclusiveness** for some dimensions, specifically the *Output Format* and *Skills* dimensions. In the context of this taxonomy, these can equivalently be understood as a set of of six and four binary dimensions respectively, each divided into characteristics "yes" and "no". However, framing them as a single dimension with non-mutually exclusive characteristics seems more intuitive.

Metadimensions structure the taxonomy, and most of the characteristics are illustrated through examples. These measures are recognized for enhancing the **explanatory power** of a taxonomy [58]. The taxonomy's flat structure allows for the easy addition of dimensions and characteristics, indicating that its **extensibility** is good. Potential extensions and further aspects of the taxonomy, including its usefulness and ease of use, are discussed in section 6.

We visualize the taxonomy (or, strictly speaking, categorized instances) in a compact form using feature vectors with characteristics abbreviated to single-letter codes. This approach has a drawback, as it requires referencing a legend. Additionally, non-applicable characteristics in mutually exclusive dimensions are not visible, which means the design space is not completely shown. However, the compactness of the representation allows LLM components within a common application to be grouped closely, so that an LLM-integrated application can be perceived as a unit without appearing convoluted. This is a significant advantage for our purposes.

6. Discussion

The discussion first focuses on the taxonomy's applicability and ease of use before considering its overall usefulness.

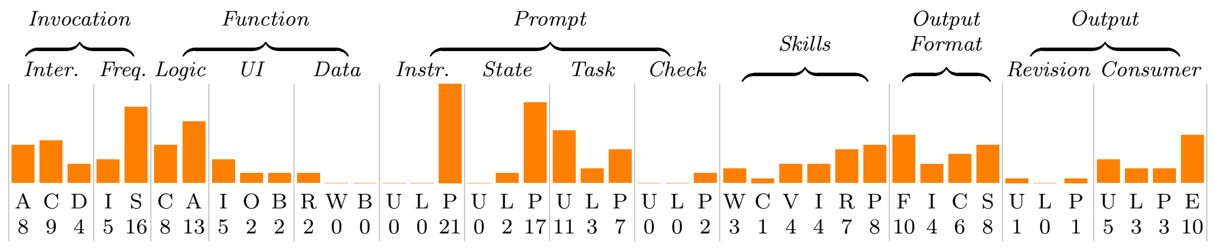

Figure 2: Occurrences of characteristics in the sample set of LLM-integrated applications.

6.1. *Applicability and ease of use*

The taxonomy was effectively applied to LLM-integrated applications based on research papers, source code blog posts, recorded software demonstrations, and developer experiences. The analysis of LowCode revealed it to be a prompt definition tool combined with an LLM-based chatbot, which deviates from the strict definition of an LLM-integrated application. Still, the taxonomy provided an effective categorization and led to a clear understanding of the system's architecture.

Obviously, the ease of categorization depends on the clarity and comprehensiveness of the available information, which varies across analyzed systems. Analyzing applications of LLMs in novel and uncommon domains can be challenging. While these papers present inspiring and innovative ideas for LLM integration, such as MyCrunchGpt and TruckPlatoon, they may prioritize explaining the application area and struggle to detail the technical aspects of the LLM integration. A taxonomy for LLM-integrated applications can guide and facilitate the writing process and lead to more standardized and comparable descriptions.

Applying the taxonomy is often more straightforward for research-focused systems. Omitting the complexities required for real-world applications, such as prompt checks and output revisions, their architectures are simpler and easier to describe. A taxonomy can point out such omissions.

A fundamental challenge in applying the taxonomy arises from the inherent versatility of LLMs, which allows to define LLM components serving multiple purposes. This is exemplified by SgpTod PolicyPrompter, where the prompt is designed to produce a structure with two distinct outcomes (a class label and a chatbot response), and similarly by MatrixProduction, as detailed section 4.2. Drawing an analogy to "function overloading" in classical programming, such LLM components can be termed "overloaded LLM components".

A taxonomy can handle overloaded LLM components in several ways: (1) define more dimensions as non-mutually exclusive, (2) label overloaded LLM components as "overloaded" without a more detailed categorization, or (3) categorize them by their predominant purpose or output. While the first approach allows for the most precise categorization, it complicates the taxonomy. Moreover, it will likely result in nearly all characteristics being marked for some LLM components, which is ultimately not helpful. The second approach simplifies categorization but sacrifices much detail. Our taxonomy adopts the third approach, enforcing simplification and abstraction in descriptions of overloaded LLM components while retaining essential detail. The taxonomy can easily be extended to include approach (2) as an additional binary dimension.

6.2. *Usefulness*

The search for instances of LLM-integrated applications uncovered activities across various domains. Substantial research involving LLM integrations, often driven by theoretical interests, is notable in robot task planning [37, 51, 61, 33, 63] and in the TOD field [23, 71, 4, 6, 56]. Research exploring LLM potentials from a more practical perspective can be found in novel domains, such as industrial production [69, 26] and other technical areas [28, 70]. Furthermore, developers of commercial LLM-based applications are beginning to communicate their efforts and challenges [44, 7]. The taxonomy has been applied to example instances from these and additional areas. This demonstrates its potential as a common, unified framework for describing LLM-integrated applications, facilitating the comparison and sharing of development knowledge between researchers and practitioners across various domains.

When applying the taxonomy to the example instances, it proved to be effective and useful as an analytical lens. Descriptions of LLM-integrated applications commonly explain background information and details of the application domain in addition to its LLM integration. When used as an analytical lens, the taxonomy quickly directs the analysis towards the aspects of LLM integration, abstracting from the specificities of the domain.

The taxonomy describes how LLM capabilities can be leveraged in software systems, offers inspiration for LLM-based functions, and outlines options for their implementation as follows. The *Skills* dimension outlines the range of capabilities an LLM can contribute to an application through a concise set of characteristics, while the *Function* dimension suggests potential uses, further supported by the *Interaction* dimension. The *Output Type* dimension indicates options for encoding the output of an LLM in formats beyond plain text, making it processable by software. The *Output Consumer* dimension illustrates the diverse ways to utilize or act upon LLM output. Thus, the taxonomy, as intended, spans a design space for LLM integrations.

The sampled LLM-integrated applications showcase the creativity of researchers and developers in applying and exploiting the potentials of LLMs, ranging from straightforward solutions (e.g., TruckPlatoon) to highly sophisticated and technically complex ones (e.g., AutoDroid). When using the taxonomy to inspire innovative uses of LLMs, we recommend supplementing it with descriptions of example applications to enhance its illustrativeness. The characteristics of the *Skills* dimension are derived pragmatically from the investigated example instances. While they do not claim to be exhaustive or deeply rooted in LLM theory or cognitive science, they add relevant details to the categorizations and illustrate design options and potentials for using LLMs as software components.

It emerged as a key insight of this research that, rather than analyzing an LLM-integrated application in whole, analysis should start with the identification and description of its distinct LLM components. This is essential for gaining a clear understanding of how the application utilizes the capabilities of LLMs. The LLM-integrated application then manifests as a combination of its LLM components. As shown in figure 1, the visualization effectively displays both the quantity and the variety of LLM components in an LLM-integrated application.

LLM components interact through prompt chaining, where one LLM component's output feeds into another's input [67]. When an LLM-integrated application involves such an interaction, the taxonomy represents it as an *LLM* characteristic within a *Prompt* dimension. The taxonomy can capture the variance in these interactions. For instance, in AutoDroid TaskExecutor and LowCode Executing, the *LLM* characteristic appears in the *Prompt State* dimension, because their prompt components (knowledge base excerpts and prompt definition, respectively) are generated by other LLM components in a preparatory stage. In contrast, the *LLM* characteristic appears in the *Prompt Task* dimension for MatrixProduction Operator, because its prompt part is generated individually by the MatrixProduction Manager almost immediately before use.

Taxonomy dimensions that cover entire LLM-integrated applications may be useful. Given their complexity, these dimensions should be designed based on a broader range of examples, which will only become available as more LLM-integrated applications are developed and their architectures disclosed in the future. Extensions to the taxonomy could also include dimensions for describing the structure of prompts in more detail, as well as dimensions addressing characteristics of the language models used.

Table 4: LLM usage in the sample instances. "Evals" indicates evaluations of various LLMs.

Application	Used or best LLM	Evals	Comments
Honeycomb	GPT-3.5	yes	GPT-4 far too slow
LowCode	GPT-3.5-turbo		
MyCrunchGpt	GPT-3.5		then awaiting the publication of GPT-4
MatrixProduction	text-davinci-003		
WorkplaceRobot	GPT-3		
AutoDroid	GPT-4	yes	GPT-4 best for tasks requiring many steps
ProgPrompt	GPT-3		CODEX better, but access limits prohibitive
FactoryAssistants	GPT-3.5		
SgpTod	GPT-3.5	yes	GPT-3.5 best more often than others combined
TruckPlatoon	GPT-3.5-turbo		
ExcelCopilot	N/A		combined LLMs in Copilot for Microsoft 365 [43]

7. Conclusion

This paper investigates the use of LLMs as software components. Its perspective differs from current software engineering research, which investigates LLMs as tools for software development [14, 22] and from research examining LLMs as autonomous agents [11, 62, 57, 21]. This paper defines the concept of an LLM component as a software component that realizes its functionality by invoking an LLM. While LLM components implicitly appear in various works, termed, for example, "prompters", "prompted LLM", "prompt module", or "module" [30, 71, 6, 7], to our knowledge, this concept has not yet been formalized or systematically investigated.

The main contribution of this study is a taxonomy for the analysis and description of LLM components, extending to LLM-integrated applications by characterizing them as combinations of LLM components. In addition to the dimensions and characteristics of the taxonomy, the study contributes a taxonomy visualization based on feature vectors, which is more compact than the established visualizations such as morphological boxes [55] or radar charts. It represents an LLM-integrated application as one visual entity in a tabular format, with its LLM components displayed as rows.

The taxonomy was constructed using established methods, based on a set of example instances, and evaluated with a new set of example instances. The combined samples exhibit broad variation along the identified dimensions. For some instances, information was not available, necessitating speculative interpretation. However, since the sample is used for identifying options rather than quantitative analysis, this issue and the representativeness of the sample are not primary concerns. The evaluation was conducted by the developer of the taxonomy, consistent with recent related work [21, 52, 48]. Using a new sample for evaluation strengthens the validity of the results.

A further significant contribution of the paper is a systematic overview of a sample of LLM-integrated applications across various industrial and technical domains, illustrating a spectrum of conceptual ideas and implementation options.

As the examples show, LLM components can replace traditionally coded functions in software systems and enable novel use cases. However, practical challenges persist. Developers report that new software engineering methods are required, e.g., for managing prompts as software assets and for testing and monitoring applications. For instance, the costs of LLM invocations prohibit the extensive automated testing that is standard in software development practice [44, 7]. Challenges also arise from the inherent indeterminism and uncontrollability of LLMs. Small variations in prompts can lead to differences in outputs, while automated output processing in LLM-integrated applications requires the output to adhere to a specified format.

Furthermore, the deployment mode of LLMs, whether local (on the same hardware as the application) or remote, managed privately or offered as Language-Models-as-a-Service (LMaaS), has impact on performance and usability. Table 4 gives an overview of the LLMs used in our sample of applications. Where papers report evaluations of multiple LLMs, the table displays the chosen or best-performing LLM. Although not representative, the table provides some insights. LMaaS dominates, likely due to its convenience, but more importantly, due to the superior performance of the provided LLMs.

Concerns regarding LMaaS include privacy, as sensitive data might be transmitted to the LLM through the prompt [64], and service quality, i.e., reliability, availability, and costs. Costs typically depend on the quantity of processed tokens. This quantity also affects latency, which denotes the processing time of an LLM invocation. A further important factor for latency is the size of the LLM, with larger models being slower [7].

When building LLM-based applications for real-world use, the reliability and availability of an LMaaS are crucial. Availability depends not only on the technical stability of the service, but also on factors such as increased latency during high usage periods or usage restrictions imposed by the provider of an LMaaS, as reported for ProgPrompt [51]. Beyond technical aspects, the reliability of an LMaaS also encompasses its behavior. For instance, providers might modify a model to enhance its security, potentially impacting applications that rely on it.

Despite practical challenges, integrating LLMs into systems has the potential to alter the way software is constructed and the types of systems that can be realized. Prompts are central to the functioning of LLM components which pose specific requirements such as strict format adherence. Therefore, an important direction for future research will be prompt engineering specifically tailored for LLM-integrated applications.

In future work, the taxonomy will be extended to distinguish finer-grained parts of prompts, allowing a more detailed description and comparison of prompts and related experimental results. Initial studies share results on the format-following behavior of LLMs [68] as a subtopic of instruction-following [73], derived with synthetic benchmark data. It is necessary to complement their results with experiments using data and tasks from real application development projects because, in the early stages of this field, synthetic benchmarks may fail to cover relevant aspects within the wide range of possible options. Another crucial research direction involves exploring how LLM characteristics correspond to specific tasks, such as determining the optimal LLM size for intent detection tasks. The taxonomy developed in this study can systematize such experiments and their outcomes. Additionally, it provides a structured framework for delineating design choices in LLM components, making it a valuable addition to future training materials.

Acknowledgements

Special thanks to Antonia Weber and Constantin Weber for proofreading and providing insightful and constructive comments.

References

[1] Eleni Adamopoulou and Lefteris Moussiades. An Overview of Chatbot Technology. In Ilias Maglogiannis, Lazaros Iliadis, and Elias Pimenidis, editors, *Artificial Intelligence Applications and Innovations*, IFIP Advances in Information and Communication Technology, pages 373–383, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-49186-4_31.

[2] Sebastian Bader, Erich Barnstedt, Heinz Bedenbender, Bernd Berres, Meik Billmann, and Marko Ristin. Details of the asset administration shell-part 1: The exchange of information between partners in the value chain of industrie 4.0 (version 3.0 rc02). Working Paper, Berlin: Federal Ministry for Economic Affairs

and Climate Action (BMWK), 2022. `doi.org/10.21256/zhaw-27075`.

[3] Marcos Baez, Florian Daniel, Fabio Casati, and Boualem Benatallah. Chatbot integration in few patterns. *IEEE Internet Computing*, pages 1–1, 2020. doi:10.1109/MIC.2020.3024605.

[4] Tom Bocklisch, Thomas Werkmeister, Daksh Varshneya, and Alan Nichol. Task-Oriented Dialogue with In-Context Learning. (arXiv:2402.12234), February 2024. doi:10.48550/arXiv.2402.12234.

[5] Yuzhe Cai, Shaoguang Mao, Wenshan Wu, Zehua Wang, Yaobo Liang, Tao Ge, Chenfei Wu, Wang You, Ting Song, Yan Xia, Jonathan Tien, and Nan Duan. Low-code LLM: Visual Programming over LLMs. (arXiv:2304.08103), April 2023. doi:10.48550/arXiv.2304.08103.

[6] Lang Cao. DiagGPT: An LLM-based Chatbot with Automatic Topic Management for Task-Oriented Dialogue. (arXiv:2308.08043), August 2023. doi:10.48550/arXiv.2308.08043.

[7] Phillip Carter. *All the Hard Stuff Nobody Talks About When Building Products with LLMs.* Honeycomb, May 2023. `https://www.honeycomb.io/blog/hard-stuff-nobody-talks-about-llm`.

[8] Phillip Carter. *So We Shipped an AI Product. Did It Work?* Honeycomb, October 2023. `https://www.honeycomb.io/blog/we-shipped-ai-product`.

[9] Banghao Chen, Zhaofeng Zhang, Nicolas Langrené, and Shengxin Zhu. Unleashing the potential of prompt engineering in Large Language Models: A comprehensive review. (arXiv:2310.14735), October 2023. doi:10.48550/arXiv.2310.14735.

[10] Wang Chen, Yan-yi Liu, Tie-zheng Guo, Dapeng Li, Tao He, Li Zhi, Qing-wen Yang, Hui-han Wang, and Ying-you Wen. Systems engineering issues for industry applications of large language model. *Applied Soft Computing*, 151:111165, January 2024. doi:10.1016/j.asoc.2023.111165.

[11] Yuheng Cheng, Ceyao Zhang, Zhengwen Zhang, Xiangrui Meng, Sirui Hong, Wenhao Li, Zihao Wang, Zekai Wang, Feng Yin, Junhua Zhao, and Xiuqiang He. Exploring Large Language Model based Intelligent Agents: Definitions, Methods, and Prospects. (arXiv:2401.03428), January 2024. doi:10.48550/arXiv.2401.03428.

[12] Silvia Colabianchi, Andrea Tedeschi, and Francesco Costantino. Human-technology integration with industrial conversational agents: A conceptual architecture and a taxonomy for manufacturing. *Journal of Industrial Information Integration*, 35:100510, October 2023. doi:10.1016/j.jii.2023.100510.

[13] Jonathan Evertz, Merlin Chlosta, Lea Schönherr, and Thorsten Eisenhofer. Whispers in the Machine: Confidentiality in LLM-integrated Systems. (arXiv:2402.06922), February 2024. doi:10.48550/arXiv.2402.06922.

[14] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. Large Language Models for Software Engineering: Survey and Open Problems. (arXiv:2310.03533), November 2023. doi:10.48550/arXiv.2310.03533.

[15] Wenqi Fan, Zihuai Zhao, Jiatong Li, Yunqing Liu, Xiaowei Mei, Yiqi Wang, Zhen Wen, Fei Wang, Xiangyu Zhao, Jiliang Tang, and Qing Li. Recommender Systems in the Era of Large Language Models (LLMs). (arXiv:2307.02046), August 2023. doi:10.48550/arXiv.2307.02046.

[16] David Fortin. *Microsoft Copilot in Excel: What It Can and Can't Do.* YouTube, January 2024. `https://www.youtube.com/watch?v=-fsu9IXMZvo`.

[17] Martin Fowler. *Patterns of Enterprise Application Architecture.* 2002. ISBN 978-0-321-12742-6.

[18] Shirley Gregor. The nature of theory in information systems. *MIS quarterly*, pages 611–642, 2006. doi:10.2307/25148742.

[19] Yanchu Guan, Dong Wang, Zhixuan Chu, Shiyu Wang, Feiyue Ni, Ruihua Song, Longfei Li, Jinjie Gu, and Chenyi Zhuang. Intelligent Virtual Assistants with LLM-based Process Automation. (arXiv:2312.06677), December 2023. doi:10.48550/arXiv.2312.06677.

[20] Muhammad Usman Hadi, Qasem Al Tashi, Rizwan Qureshi, Abbas Shah, Amgad Muneer, Muhammad Irfan, Anas Zafar, Muhammad Bilal Shaikh, Naveed Akhtar, Jia Wu, and Seyedali Mirjalili. Large Language Models: A Comprehensive Survey of its Applications, Challenges, Limitations, and Future Prospects, September 2023. doi:10.36227/techrxiv.23589741.v3.

[21] Thorsten Händler. A Taxonomy for Autonomous LLM-Powered Multi-Agent Architectures:. In *Proceedings of the 15th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management*, pages 85–98, Rome, Italy, 2023. SCITEPRESS - Science and Technology Publications. doi:10.5220/0012239100003598.

[22] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large Language Models for Software Engineering: A Systematic Literature Review. (arXiv:2308.10620), September 2023. doi:10.48550/arXiv.2308.10620.

[23] Vojtěch Hudeček and Ondrej Dusek. Are Large Language Models All You Need for Task-Oriented Dialogue? In Svetlana Stoyanchev, Shafiq Joty, David Schlangen, Ondrej Dusek, Casey Kennington, and Malihe Alikhani, editors, *Proceedings of the 24th Annual Meeting of the Special Interest Group on Discourse and Dialogue*, pages 216–228, Prague, Czechia, September 2023. Association for Computational Linguistics. doi:10.18653/v1/2023.sigdial-1.21.

[24] Kevin Maik Jablonka, Qianxiang Ai, Alexander Al-Feghali, Shruti Badhwar, Joshua D. Bocarsly, Andres M. Bran, Stefan Bringuier, Catherine L. Brinson, Kamal Choudhary, Defne Circi, Sam Cox, Wibe A. de Jong, Matthew L. Evans, Nicolas Gastellu, Jerome Genzling, María Victoria Gil, Ankur K. Gupta, Zhi Hong, Alishba Imran, Sabine Kruschwitz, Anne Labarre, Jakub Lála, Tao Liu, Steven Ma, Sauradeep Majumdar, Garrett W. Merz, Nicolas Moitessier, Elias Moubarak, Beatriz Mouriño, Brenden Pelkie, Michael Pieler, Mayk Caldas Ramos, Bojana Ranković, Samuel Rodriques, Jacob Sanders, Philippe Schwaller, Marcus Schwarting, Jiale Shi, Berend Smit, Ben Smith, Joren Van Herck, Christoph Völker, Logan Ward, Sean Warren, Benjamin Weiser, Sylvester Zhang, Xiaoqi Zhang, Ghezal Ahmad Zia, Aristana Scourtas, K. Schmidt, Ian Foster, Andrew White, and Ben Blaiszik. 14 examples of how LLMs can transform materials science and chemistry: A reflection on a large language model hackathon. *Digital Discovery*, 2(5):1233–1250, 2023. doi:10.1039/D3DD00113J.

[25] Jean Kaddour, Joshua Harris, Maximilian Mozes, Herbie Bradley, Roberta Raileanu, and Robert McHardy. Challenges and Applications of Large Language Models, July 2023. doi:10.48550/arXiv.2307.10169.

[26] Samuel Kernan Freire, Mina Foosherian, Chaofan Wang, and Evangelos Niforatos. Harnessing Large Language Models for Cognitive Assistants in Factories. In *Proceedings of the 5th International Conference on Conversational User Interfaces*, CUI '23, pages 1–6, New York, NY, USA, July 2023. Association for Computing Machinery. doi:10.1145/3571884.3604313.

[27] Anis Koubaa, Wadii Boulila, Lahouari Ghouti, Ayyub Alzahem, and Shahid Latif. Exploring ChatGPT Capabilities and Limitations: A Survey. *IEEE Access*, 11:118698–118721, 2023. doi:10.1109/ACCESS.2023.3326474.

[28] Varun Kumar, Leonard Gleyzer, Adar Kahana, Khemraj Shukla, and George Em Karni-

adakis. MyCrunchGPT: A LLM Assisted Framework for Scientific Machine Learning. *Journal of Machine Learning for Modeling and Computing*, 4(4), 2023. `doi.org/10.1615/JMachLearnModelComput.2023049518`.

[29] Dennis Kundisch, Jan Muntermann, Anna Maria Oberländer, Daniel Rau, Maximilian Röglinger, Thorsten Schoormann, and Daniel Szopinski. An Update for Taxonomy Designers. *Business & Information Systems Engineering*, 64(4):421–439, August 2022. doi:10.1007/s12599-021-00723-x.

[30] Gibbeum Lee, Volker Hartmann, Jongho Park, Dimitris Papailiopoulos, and Kangwook Lee. Prompted LLMs as chatbot modules for long open-domain conversation. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki, editors, *Findings of the association for computational linguistics: ACL 2023*, pages 4536–4554, Toronto, Canada, July 2023. Association for Computational Linguistics. doi:10.18653/v1/2023.findings-acl.277.

[31] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing. *ACM Computing Surveys*, 55(9):195:1–195:35, January 2023. doi:10.1145/3560815.

[32] Yi Liu, Gelei Deng, Yuekang Li, Kailong Wang, Tianwei Zhang, Yepang Liu, Haoyu Wang, Yan Zheng, and Yang Liu. Prompt Injection attack against LLM-integrated Applications, June 2023. doi:10.48550/arXiv.2306.05499.

[33] Yuchen Liu, Luigi Palmieri, Sebastian Koch, Ilche Georgievski, and Marco Aiello. DELTA: Decomposed Efficient Long-Term Robot Task Planning using Large Language Models. (arXiv:2404.03275), April 2024. doi:10.48550/arXiv.2404.03275.

[34] Yupei Liu, Yuqi Jia, Runpeng Geng, Jinyuan Jia, and Neil Zhenqiang Gong. Prompt Injection Attacks and Defenses in LLM-Integrated Applications. (arXiv:2310.12815), October 2023. doi:10.48550/arXiv.2310.12815.

[35] Shaoguang Mao, Qiufeng Yin, Yuzhe Cai, and Dan Qiao. LowCodeLLM. `https://github.com/chenfei-wu/TaskMatrix/tree/main/LowCodeLLM`, May 2023.

[36] Scott McLean, Gemma J. M. Read, Jason Thompson, Chris Baber, Neville A. Stanton, and Paul M. Salmon. The risks associated with Artificial General Intelligence: A systematic review. *Journal of Experimental & Theoretical Artificial Intelligence*, 35(5):649–663, July 2023. doi:10.1080/0952813X.2021.1964003.

[37] Oier Mees, Jessica Borja-Diaz, and Wolfram Burgard. Grounding Language with Visual Affordances over Unstructured Data. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 11576–11582, London, United Kingdom, May 2023. IEEE. doi:10.1109/ICRA48891.2023.10160396.

[38] Grégoire Mialon, Roberto Dessì, Maria Lomeli, Christoforos Nalmpantis, Ram Pasunuru, Roberta Raileanu, Baptiste Rozière, Timo Schick, Jane Dwivedi-Yu, Asli Celikyilmaz, Edouard Grave, Yann LeCun, and Thomas Scialom. Augmented Language Models: A Survey, February 2023. doi:10.48550/arXiv.2302.07842.

[39] Melanie Mitchell. Debates on the nature of artificial general intelligence. *Science*, 383(6689):eado7069, March 2024. doi:10.1126/science.ado7069.

[40] Quim Motger, Xavier Franch, and Jordi Marco. Software-Based Dialogue Systems: Survey, Taxonomy, and Challenges. *ACM Computing Surveys*, 55(5):91:1–91:42, December 2022. doi:10.1145/3527450.

[41] Fiona Fui-Hoon Nah, Ruilin Zheng, Jingyuan Cai, Keng Siau, and Langtao Chen. Generative AI and ChatGPT: Applications, challenges, and AI-human collaboration. *Jour-*

nal of Information Technology Case and Application Research, 25(3):277–304, July 2023. doi:10.1080/15228053.2023.2233814.

[42] Robert C Nickerson, Upkar Varshney, and Jan Muntermann. A method for taxonomy development and its application in information systems. *European Journal of Information Systems*, 22(3):336–359, May 2013. doi:10.1057/ejis.2012.26.

[43] Camille Pack, Cern McAtee, Samantha Robertson, Dan Brown, Aditi Srivastava, and Kweku Ako-Adjei. Microsoft Copilot for Microsoft 365 overview. `https://learn.microsoft.com/en-us/copilot/microsoft-365/microsoft-365-copilot-overview`, March 2024.

[44] Chris Parnin, Gustavo Soares, Rahul Pandita, Sumit Gulwani, Jessica Rich, and Austin Z. Henley. Building Your Own Product Copilot: Challenges, Opportunities, and Needs. (arXiv:2312.14231), December 2023. doi:10.48550/arXiv.2312.14231.

[45] Rodrigo Pedro, Daniel Castro, Paulo Carreira, and Nuno Santos. From Prompt Injections to SQL Injection Attacks: How Protected is Your LLM-Integrated Web Application? (arXiv:2308.01990), August 2023. doi:10.48550/arXiv.2308.01990.

[46] Ken Peffers, Tuure Tuunanen, Marcus A. Rothenberger, and Samir Chatterjee. A Design Science Research Methodology for Information Systems Research. *Journal of Management Information Systems*, 24(3):45–77, December 2007. ISSN 0742-1222, 1557-928X. doi:10.2753/MIS0742-1222240302.

[47] Mohaimenul Azam Khan Raiaan, Md. Saddam Hossain Mukta, Kaniz Fatema, Nur Mohammad Fahad, Sadman Sakib, Most Marufatul Jannat Mim, Jubaer Ahmad, Mohammed Eunus Ali, and Sami Azam. A Review on Large Language Models: Architectures, Applications, Taxonomies, Open Issues and Challenges. *IEEE Access*, 12:26839–26874, 2024. doi:10.1109/ACCESS.2024.3365742.

[48] Jack Daniel Rittelmeyer and Kurt Sandkuhl. Morphological Box for AI Solutions: Evaluation and Refinement with a Taxonomy Development Method. In Knut Hinkelmann, Francisco J. López-Pellicer, and Andrea Polini, editors, *Perspectives in Business Informatics Research*, Lecture Notes in Business Information Processing, pages 145–157, Cham, 2023. Springer Nature Switzerland. doi:10.1007/978-3-031-43126-5_11.

[49] Shubhra Kanti Karmaker Santu and Dongji Feng. TELeR: A General Taxonomy of LLM Prompts for Benchmarking Complex Tasks. (arXiv:2305.11430), October 2023. doi:10.48550/arXiv.2305.11430.

[50] Thorsten Schoormann, Frederik Möller, and Daniel Szopinski. Exploring Purposes of Using Taxonomies. In *Proceedings of the International Conference on Wirtschaftsinformatik (WI)*, Nuernberg, Germany, February 2022.

[51] Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. ProgPrompt: Generating Situated Robot Task Plans using Large Language Models. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 11523–11530, London, United Kingdom, May 2023. IEEE. doi:10.1109/ICRA48891.2023.10161317.

[52] Gero Strobel, Leonardo Banh, Frederik Möller, and Thorsten Schoormann. Exploring Generative Artificial Intelligence: A Taxonomy and Types. In *Proceedings of the 57th Hawaii International Conference on System Sciences*, Honolulu, Hawaii, January 2024. `https://hdl.handle.net/10125/106930`.

[53] Hendrik Strobelt, Albert Webson, Victor Sanh, Benjamin Hoover, Johanna Beyer, Hanspeter Pfister, and Alexander M. Rush. Interactive and Visual Prompt Engineering for Adhoc Task Adaptation With Large Language

Models. *IEEE Transactions on Visualization and Computer Graphics*, pages 1–11, 2022. doi:10.1109/TVCG.2022.3209479.

[54] Daniel Szopinski, Thorsten Schoormann, and Dennis Kundisch. Criteria as a Prelude for Guiding Taxonomy Evaluation. In *Proceedings of the 53rd Hawaii International Conference on System Sciences*, 2020. `https://hdl.handle.net/10125/64364`.

[55] Daniel Szopinski, Thorsten Schoormann, and Dennis Kundisch. Visualize different: Towards researching the fit between taxonomy visualizations and taxonomy tasks. In *Tagungsband Der 15. Internationalen Tagung Wirtschaftsinformatik (WI 2020)*, Potsdam, 2020. doi:10.30844/wi_2020_k9-szopinski.

[56] Manisha Thakkar and Nitin Pise. Unified Approach for Scalable Task-Oriented Dialogue System. *International Journal of Advanced Computer Science and Applications*, 15(4), 2024. doi:10.14569/IJACSA.2024.01504108.

[57] Oguzhan Topsakal and Tahir Cetin Akinci. Creating Large Language Model Applications Utilizing Langchain: A Primer on Developing LLM Apps Fast. In *International Conference on Applied Engineering and Natural Sciences*, volume 1, pages 1050–1056, 2023.

[58] Michael Unterkalmsteiner and Waleed Adbeen. A compendium and evaluation of taxonomy quality attributes. *Expert Systems*, 40(1): e13098, 2023. doi:10.1111/exsy.13098.

[59] Bryan Wang, Gang Li, and Yang Li. Enabling Conversational Interaction with Mobile UI using Large Language Models. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, CHI '23, pages 1–17, New York, NY, USA, April 2023. Association for Computing Machinery. doi:10.1145/3544548.3580895.

[60] Can Wang, Bolin Zhang, Dianbo Sui, Zhiying Tu, Xiaoyu Liu, and Jiabao Kang. A Survey on Effective Invocation Methods of Massive LLM Services. (arXiv:2402.03408), February 2024. doi:10.48550/arXiv.2402.03408.

[61] Jun Wang, Guocheng He, and Yiannis Kantaros. Safe Task Planning for Language-Instructed Multi-Robot Systems using Conformal Prediction. (arXiv:2402.15368), February 2024. doi:10.48550/arXiv.2402.15368.

[62] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Jirong Wen. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):186345, March 2024. doi:10.1007/s11704-024-40231-1.

[63] Shu Wang, Muzhi Han, Ziyuan Jiao, Zeyu Zhang, Ying Nian Wu, Song-Chun Zhu, and Hangxin Liu. LLM3:Large Language Model-based Task and Motion Planning with Motion Failure Reasoning. (arXiv:2403.11552), March 2024. doi:10.48550/arXiv.2403.11552.

[64] Hao Wen, Yuanchun Li, Guohong Liu, Shanhui Zhao, Tao Yu, Toby Jia-Jun Li, Shiqi Jiang, Yunhao Liu, Yaqin Zhang, and Yunxin Liu. Empowering LLM to use Smartphone for Intelligent Task Automation. (arXiv:2308.15272), September 2023. doi:10.48550/arXiv.2308.15272.

[65] Hao Wen, Yuanchun Li, and Sean KiteFlyKid. MobileLLM/AutoDroid. Mobile LLM, January 2024. `https://github.com/MobileLLM/AutoDroid`.

[66] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C. Schmidt. A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT. (arXiv:2302.11382), February 2023. doi:10.48550/arXiv.2302.11382.

[67] Tongshuang Wu, Michael Terry, and Carrie Jun Cai. AI Chains: Transparent and

Controllable Human-AI Interaction by Chaining Large Language Model Prompts. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, CHI '22, pages 1–22, New York, NY, USA, April 2022. Association for Computing Machinery. doi:10.1145/3491102.3517582.

[68] Congying Xia, Chen Xing, Jiangshu Du, Xinyi Yang, Yihao Feng, Ran Xu, Wenpeng Yin, and Caiming Xiong. FOFO: A Benchmark to Evaluate LLMs' Format-Following Capability. (arXiv:2402.18667), February 2024. doi:10.48550/arXiv.2402.18667.

[69] Yuchen Xia, Manthan Shenoy, Nasser Jazdi, and Michael Weyrich. Towards autonomous system: Flexible modular production system enhanced with large language model agents. In *2023 IEEE 28th International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8, 2023. doi:10.1109/ETFA54631.2023.10275362.

[70] I. de Zarzà, J. de Curtò, Gemma Roig, and Carlos T. Calafate. LLM Adaptive PID Control for B5G Truck Platooning Systems. *Sensors*, 23(13):5899, January 2023. doi:10.3390/s23135899.

[71] Xiaoying Zhang, Baolin Peng, Kun Li, Jingyan Zhou, and Helen Meng. SGP-TOD: Building Task Bots Effortlessly via Schema-Guided LLM Prompting. (arXiv:2305.09067), May 2023. doi:10.48550/arXiv.2305.09067.

[72] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. A Survey of Large Language Models. (arXiv:2303.18223), May 2023. doi:10.48550/arXiv.2303.18223.

[73] Jeffrey Zhou, Tianjian Lu, Swaroop Mishra, Siddhartha Brahma, Sujoy Basu, Yi Luan, Denny Zhou, and Le Hou. Instruction-Following Evaluation for Large Language Models. (arXiv:2311.07911), November 2023. doi:10.48550/arXiv.2311.07911.