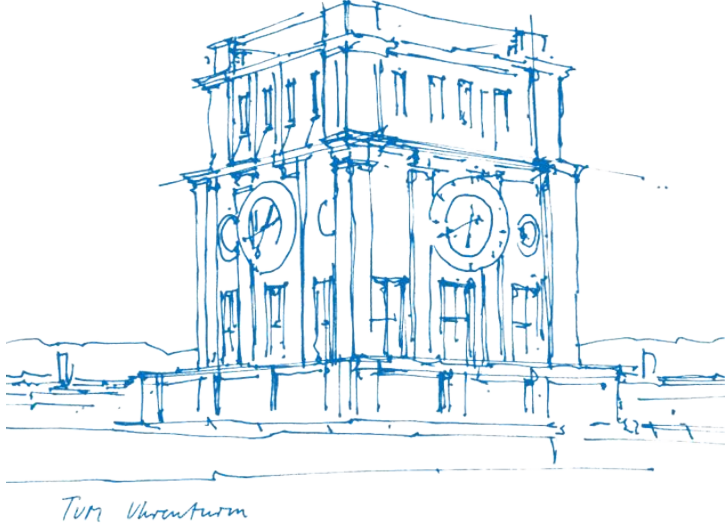
Computer Aided Medical Procedures & Augmented Reality
TUM School of Computation, Information and Technology
Technical University of Munich

Exploring Large Language Models for Scene Understanding in the Multimodal Operating Room

Elias Wohlgemuth



Exploring Large Language Models for Scene Understanding in the Multimodal Operating Room

Elias Wohlgemuth

Computer Aided Medical Procedures & Augmented Reality
TUM School of Computation, Information and Technology
Technical University of Munich

Exploring Large Language Models for Scene Understanding in the Multimodal Operating Room

Elias Wohlgemuth

Thesis for the attainment of the academic degree

Master of Science (M.Sc.)

at the TUM School of Computation, Information and Technology of the Technical University of Munich.

Examiner:
Prof. Dr. Nassir Navab

Supervisor:
Ege Özsoy

Submitted:
Munich, 15.08.2024

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

Munich, 15.08.2024 Elias Wohlgemuth

Abstract

Scene understanding is an advanced task in computer vision that has seen great progress in the past. However, current works address the task in a general domain. Deploying a system with scene understanding capabilities in the operating room (OR) and making it interactable through natural language can provide several benefits to the surgical staff, for example, in the form of decision support or context-aware assistance. This work implements a framework for scene understanding in the OR, utilizing multi-modal data from an in-house dataset. The framework employs an LLM that processes an input query and instructs multiple models for processing visual- and audio data from the multi-modal dataset, applying the paradigm of tool usage. To this end, the LLM generates code that uses tool interfaces to call the appropriate tools upon execution. This approach leverages the strong language understanding and problem-solving abilities of LLMs, bypassing their limitation to textual data by instructing specialized models for processing non-textual data of different modalities. Evaluated on general-domain benchmarks, this leads to improved performance over single-model approaches for vision-language tasks, such as BLIP-2. The multi-model approach also enables multi-modality. Additionally, the framework does not require any training but only combines pre-trained and open-source models. It is modular, allowing for the integration of new models or the replacement of existing ones. Performance with respect to scene understanding in the OR is evaluated on a range of devised benchmarks. Results show a robust performance of the LLM in generating correct code when specific examples are provided in its prompt. However, the utilized general-domain vision models do not perform adequately in the OR domain, motivating the need for specialized models to be integrated into the framework.

Contents

1 Introduction

The operating room (OR) of the present has seen many technological innovations over the past few decades, and as a result, patient safety has continuously improved by assisting surgeons and expanding their use of new tools. Advanced imaging systems, assistive surgical robots, and real-time data analytics are a few examples of how technology has enhanced the precision and efficiency of surgical procedures. Among the advancements, surgical data science has emerged as a discipline that focuses on the capture, analysis, and modeling of diverse data produced during surgeries in the operating room [31]. Fully utilizing the data holds the potential to further improve the performance of the surgery team, increasing patient safety and quality of surgical intervention [20].

A potential use of the data is for scene understanding, capturing processes, and various interactions between different entities in the information-dense OR. Providing holistic and automatic scene understanding is a crucial component of a new generation of computer-assisted surgery, as it can improve surgical efficiency and quality of care [22, 20]. It goes further than current approaches that only address specific tasks like surgical phase recognition or tool recognition [12, 38]. Extracting this information can be utilized to update the function of tools or data visualizations aiding the surgical team. A generalization over multiple specific tasks combines the individual benefits and introduces new capabilities, unlocking the next generation of computer-aided surgeries.

1.1 Surgical Data Science

Surgical data science is an emerging discipline coined by Maier-Hein et al. that utilizes large-scale data science for surgical interventions [31]. It postulates that future advances in the field of surgical interventions will come from the introduction of data science into the field, driven by advances in machine-learning techniques. This will improve the quality of interventional healthcare "through the capture, organization, analysis and modelling of data" [31]. This data, from various sources such as intraoperative imaging and video recordings, sensors from instruments or assistive devices, and medical records, has, in part, not been widely available for a long time due to the lack of standardized patient data digitalization, as well as differing clinical practices.

The improvements brought by surgical data science can be attributed to various applications. Context-aware assistance aids physicians by predicting the remaining time of procedures or the requirement of resources. Surgical training can be transformed by computer-aided skill evaluation to improve surgical skills. Another application is decision support, where consequential decisions are traditionally made based on surgeons' experience. Surgical data science offers to condense high-volume data into relevant processable information to be used by the surgeon.

1.2 Scene Understanding

Scene understanding is a subfield of computer vision, where algorithms are devised to equip systems with the ability to understand a scene, which they are presented or are part of, at different granularities. It strongly relies on image understanding but goes beyond it by also allowing the integration of further data modalities, such as a 3D point cloud or audio data. Scene understanding includes the ability of systems to recognize and interpret objects, relations, and actions in a scene. Varying degrees of it might be required for systems deployed in the field of autonomous driving, domestic robotics, or healthcare systems [36]. For example, a domestic robotic vacuum cleaner possesses scene understanding capabilities limited

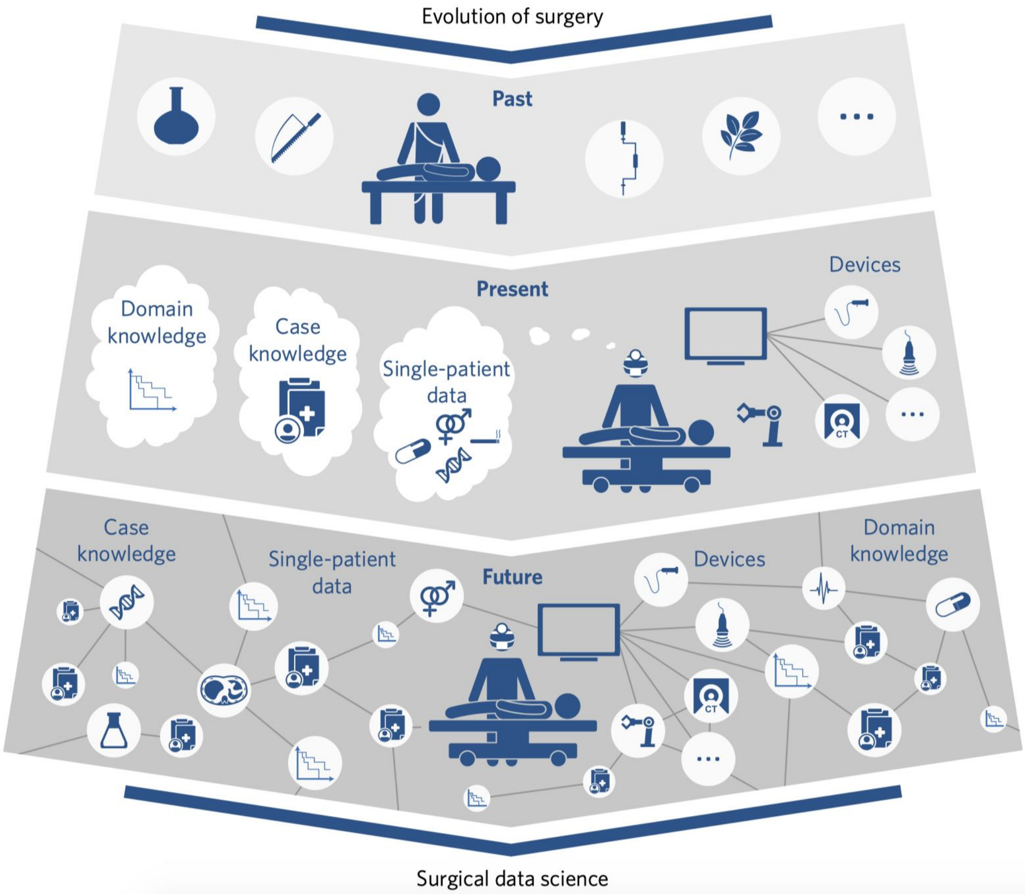


Figure 1.1 Surgical data science brings improvements to patient care through utilizing advances in large-scale data science. Image from Mayer et al. [31].

to geometric scene mapping enabling navigation, while an autonomous vehicle might additionally require contextual awareness and semantic scene understanding to navigate safely. As such, scene understanding describes an umbrella task containing different subtasks like object detection, semantic segmentation, relation detection, action detection, scene classification, and contextual awareness. These are standalone tasks that have also been addressed individually, for example, object detection by the work of He et al. [13], or semantic segmentation by the work of Long et al. [29]. The goal of current research is to devise an approach that combines these tasks to perform holistic scene understanding.

A key innovation in the journey towards holistic scene understanding is the vision transformer proposed by Dosovitskiy et al. [9]. It processes an image using the transformer architecture [57], representing the image as a multidimensional vector, which can then be used for various downstream tasks. This architecture has been integrated into many other influential and advanced approaches addressing image understanding [39, 34, 21]. Other approaches further integrate strong image understanding with language understanding, which allows tasks like visual question answering, displaying strong scene understanding capabilities as well [24, 27]. Despite recent progress, holistic scene understanding remains an open research topic.

Another approach to scene understanding is through the generation of semantic scene graphs. Utilizing scene graphs as an abstract representation of an image depicting a scene has been proposed by Johnson et al. [17], providing a structured and semantically rich representation of a scene. A graph node represents objects in the scene, and the edges represent relations between them. Further information like the geometric position or attributes of an object can be stored in its node. The approach of utilizing scene graphs for scene understanding in the OR is used in the work of Özsoy et al. [40], contributing to the research towards holistic scene understanding. It is further discussed in Section 2.1.

An important application of scene understanding in the OR is for surgical process modeling (SPM), which represents surgical activities by modeling the surgical workflow [37]. Scene understanding is required in this context for the recognition of tools, surgical actions, and relations between the surgical staff. Integrating this information with SPM enables benefits such as context-aware assistance or decision support, as described in Section 1.1.

1.3 Problem Statement

This thesis aims to contribute to the domain of surgical data science in the category of data analysis and processing, with data captured in the operating room. A focus is on bringing holistic scene understanding capabilities to the OR and integrating specific tasks like surgical phase detection and tool recognition. To this end, the goal is to devise a framework that can leverage the information from multiple data modalities and provide scene understanding. This framework should be freely interactable by posing questions about a scene of the operating room in natural language. To allow this flexibility, integrating a Large Language Model (LLM) into the framework as an interface to the user is of high priority due to LLMs' advanced natural language understanding capabilities [57, 2]. It is also explored how the LLM can be utilized for scene understanding. Another goal is the ability of the framework to utilize multiple data modalities and be trivially extendable to new data modalities available in the future. As the section on related works shows, this is not a given for current approaches. Section 2.3 introduces Vision Language Models (VLM) as a model category with current state-of-the-art performance in scene understanding. These models can become quite parameter-intensive. Together with the utilization of LLMs, where the trend of increasing the parameter count is ongoing, pushing it into the trillions, training of used models with this amount of parameters can soon become infeasible due to computing resource constraints. Therefore, the last goal is to employ an approach that allows the use of open-source pre-trained models without the need for fine-tuning. This requires the use of models with good generalization and, thereby, good zero-shot abilities, i.e., the performance on data or tasks not explicitly trained on. This has special importance for the vision models deployed for scene understanding, as data from the operating room is still very limited, which also results in the lack of vision models optimized for that domain. Utilizing models that can generalize well maximizes the chance of good performance of those models in the specialized OR domain.

In summary, this thesis aims to contribute to the field of surgical data science by developing a framework for scene understanding in the operating room, leveraging data from multiple modalities. The framework features the integration of an LLM for natural language interaction and is designed to flexibly incorporate new data modalities in the future. Emphasis is also placed on utilizing open-source, pre-trained models with strong generalization and zero-shot capabilities, thereby avoiding the computational constraints associated with training large parameter-intensive models.

2 Related Works

This thesis focuses on analyzing data captured during the surgical intervention with a framework with scene understanding capabilities. To this end, this section explores works that address this task in the operating room. However, there are still very few works in this specific domain. Subsequently, Large Language Models (LLM) are explored to understand how they can be leveraged for visual data-based problems despite being models that work only with textual data. Finally, since most of the information lies in the visual data domain for the task of scene understanding, vision-language models (VLM) are discussed as the driving force in approaches to scene understanding in the general field of computer vision.

2.1 Scene Understanding in the Operating Room

Scene graphs have been proposed as an abstract representation of images in the field of computer vision by Johnson et al. [17]. 4D-OR is the work of Özsoy et al., where the authors use semantic scene graphs (SSG) to capture the surgical scene holistically [40]. The SSGs model humans and medical equipment as nodes and the interaction between these as edges, forming a representation of the operating room for each timestep. An overview is shown in Figure 2.1. As a first step, multi-view RGB frames and fused 3D point cloud data are used to predict object- and human poses using convolutional neural networks and the transformer architecture [57]. Based on the poses, the scene graph is generated by predicting relations for object pairs using a cross-entropy loss. The obtained SSG, serving as a representation of the operating room, can then be used for further downstream tasks, e.g., clinical role prediction. As a first of its kind, 4D-OR publishes an OR dataset, which was also used for training the scene graph prediction model. It consists of data from ten simulated total knee replacement surgeries captured by multi-view RGB-D sensors. It provides frame-wise annotations of object- and human poses together with the resulting SSG and the clinical roles.

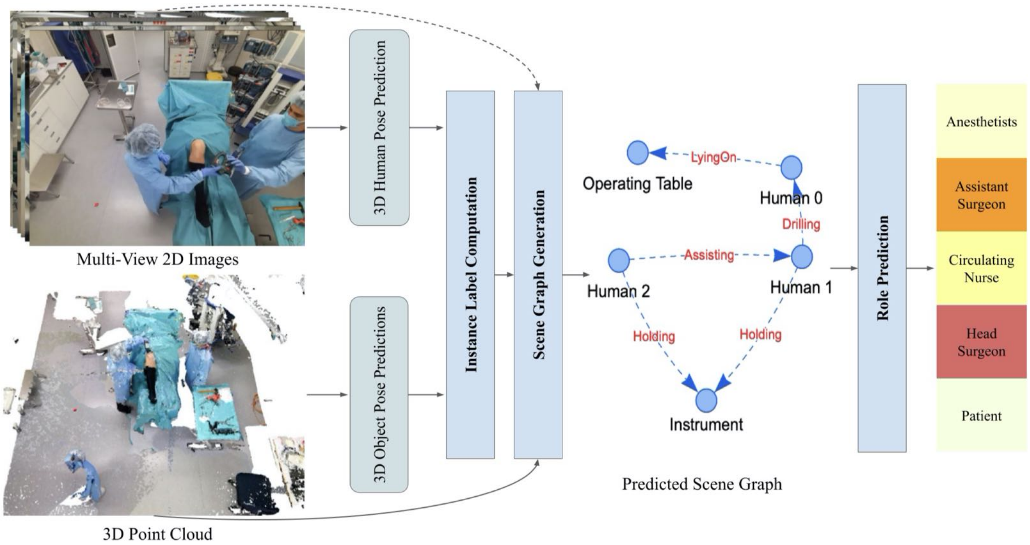


Figure 2.1 Method overview of Özsoy et al. [40]: Scene graph prediction for a single timestep used for the downstream task of clinical role prediction.

The authors can use the method to successfully generate correct scene graphs that capture the complex environment of the operating room. The metric used for quantification is the macro F1-score, which averages all individual F1-scores of the relation classes, counting a relation as correct if both relation objects and their relation were predicted correctly. The obtained macro F1-score is 0.75.

Özsoy et al. extend their research by integrating temporal information through the use of scene graphs predicted at previous timesteps, in addition to the multi-view RBG-D data, to predict the SSG of the current timestep [41]. Training is also performed on the 4D-OR dataset [40]. This approach improves the macro F1-score to 0.88.

While both approaches can capture the essential objects and relations of the operating room, they are not trivially extendable to additional data modalities but are limited in their architecture to use 2D and 3D image data only. Furthermore, the data used for training is exclusively from knee replacement surgeries at one medical facility, raising the question of generalization capabilities to heterogeneous data, such as other surgeries or other operating rooms.

2.2 Large Language Models

Models for language understanding have been a core interest of the research community for decades and have been explored within the field of natural language processing (NLP). With language being the main form of communication between humans, models possessing language skills hold the potential for seamless communication between humans and machines. Furthermore, language in the form of textual data has historically been the medium to store and convey information, representing the knowledge that humanity has accumulated. A model possessing language skills, therefore, also holds the potential ability to reason as humans do, making it applicable to any domain requiring problem-solving. The Internet, as a social and economic platform, continues to create huge amounts of text data, of which automated processing holds immense potential value, furthermore increasing the relevance of language models.

Large Language Models (LLM) are currently at the center of advancements in the field of artificial intelligence as they display continuously improving capabilities of language understanding and reasoning [5, 58]. They have shown state-of-the-art performance on tasks such as machine translation, reading comprehension, question answering, and general natural language understanding [8, 2]. These models are deep-learning algorithms that utilize a very large amount of parameters and data they are trained on, where the PaLM model uses 540 billion parameters [5] and OpenAI's GPT4 parameter count, while not officially confirmed, is speculated to be in the trillions.

At the core of LLMs architecture is the transformer network, which was developed by Vaswani et al. [57] and utilizes the attention mechanism popularized by Bahdanau et al. [1]. An input text sequence is first tokenized, which maps the sentence to tokens that can represent syllables, characters, or symbols. The tokens form the vocabulary of the LLM. Each token is then represented as a vector of a specified fixed dimensionality, creating token embeddings. Embeddings are then processed by transformer blocks that use self-attention and fully connected layers to update each embedding, repeating this process in multiple layers. This update process leads to context-aware token representations incorporating information from other relevant tokens from the sequence. The maximum number of input tokens a model can effectively process is called the context length. Finally, the updated embeddings are mapped to a probability distribution over the LLMs vocabulary, where different strategies can be used to sample the next generated token using that distribution, e.g., greedy sampling or beam search. This process is repeated iteratively until the generation process is stopped after a number of produced tokens. Generated tokens are decoded to form human-interpretable natural language.

Current research efforts focus on increasing the models' context length [49, 33], improving the parameter efficiency, shown by the improvement of Llama 3 over Llama 2 for the same parameter count [33], and continuing the scaling up of parameters, as showcased by the progression from GPT3 to GPT4 [2]. Another novel research topic is on making LLMs multi-modal, enabling them to operate beyond the textual domain [49, 54]. In this context, numerous works have explored the paradigm of tool use [51, 52, 59, 30, 46]. Two of these approaches are discussed in Section 2.2.1 and Section 2.2.2.

2.2.1 Toolformer

Toolformer by Schick et al. is a model that extends the capabilities of an LLM with external tools [51]. The LLM learns to use the tools in a self-supervised way, which minimizes the need for human data annotation. The access to tools is given by an API, and the call to an API is encoded in a special token added to the LLMs token set, i.e., its vocabulary. This requires training the LLM on the new token set in order to learn how to use the tools in a beneficial way. The included tools are a calculator, a search engine, a Q&A system, a translator, and a calendar.

An API call specifies the tool it calls but requires additional parameters to pass to the tool. To this end, a special token initializing an API call is followed by tokens representing additional parameters that are followed by another special token finalizing the API call. The training process is performed by fine-tuning a pre-trained LLM on a dataset augmented with API calls with a standard language modeling objective. This dataset is produced in a self-supervised way, where different API calls are sampled and placed among an input sequence. API calls are then executed, and the output is inserted back into the generation process of the LLM. In a filtering step it is evaluated if this input helped the LLM in predicting the correct next tokens. If so, the sample containing the API call is kept such that after this filtering, only samples containing useful API calls are obtained. By fine-tuning the LLM on this dataset, it learns when and how to do API calls in a beneficial manner.

The fine-tuned LLM, enabled with tools usage, achieves large performance improvements on tasks such as language modeling, question answering, and mathematical reasoning over the identical LLM fine-tuned on the dataset containing no API-call augmentations. The used tools specifically address some limitations of LLMs in general. The calculator addresses lacking robust skills in numerical calculations [43], and a search engine addresses hallucinating facts and limited or not up-to-date knowledge [32, 16]. As a result, the work by Schick et al. impressively demonstrates the improvement of LLM performance by integration of external tools. As a downside, this approach requires the LLM to learn the use of tools through training. Furthermore, the tools used are constrained to textual data as input and output, limiting the approach to the textual domain.

2.2.2 ViperGPT

ViperGPT by Surís et al. is a framework for visual inference built around an LLM that uses modules processing visual data [52]. A textual query is processed by the LLM, which generates code to solve it. The code includes calls to modules that can process the visual input accompanying the query. Modules are vision- or vision language models with perception and knowledge abilities, like GLIP for object detection [25], GPT-3 for integrating external knowledge [2], and BLIP-2 as a multi-purpose model, including visual question answering [24], discussed separately in Section 2.3.2.

The LLM is prompted by providing an application programming interface (API) specification followed by the textual query for a sample. The API specification is a code file defining two classes for image and video processing. The methods of the classes are the module implementations that call pre-trained models. Docstrings and in- and output-type specifications provide information on the use of the methods. In addition, query-code pairs are provided as examples. Including these examples in the prompt of the LLM is known as in-context few-shot learning, a method by Brown et al. that leverages prompt-based learning [2]. Utilizing this strategy shows that LLMs can perform tasks without being explicitly trained on them, reducing the need for fine-tuning. The generated code is executed and returns an answer to the text query. The output format is not restricted to strings but can be an arbitrary type, such as an image region.

The framework is benchmarked on various vision-centric tasks, including visual grounding and compositional image question answering, on which ViperGPT achieves state-of-the-art performance. It outperforms GLIP on visual grounding and BLIP-2 on compositional image question answering while utilizing both models as modules, showcasing the benefit of the framework over individual model use. Additionally, it enables the framework to be used for multiple tasks. Finally, the approach is training-free and utilizes an LLM and modules out-of-the-box. This makes the framework flexible and easily extendable to new modules.

2.3 Vision Language Models

Vision-and-Language is a very popular research area that combines natural language processing with computer vision [11]. The underlying idea is that the joint use of the two data modalities, language and vision, enables more robust learning, similar to how we humans obtain an understanding of the world through multiple channels. Vision language models (VLM) implementing this multi-modal approach are used for a wide variety of tasks under the umbrellas of image-text-, vision-, and video-text tasks, such as image captioning, video captioning, object detection, or visual question answering (VQA). They have progressively pushed and dominated the state-of-the-art on these tasks over the last few years, recently driven by the advent of large-scale pre-training, where noisy image-text pairs crawled from the web in the magnitude of up to billions are used for training [11]. This development has also resulted in a model being usable for a wide variety of tasks by adapting a base model to specific tasks, while earlier models were designed for a specific task from the beginning. The typical architecture of VLMs includes a vision encoder, a text encoder, and a multimodal fusion module [11]. The vision encoder processes only image data and can be a convolutional neural network but is increasingly a vision transformer [9], while the text encoder processes only textual data and is some form of transformer [57]. Both encoders produce feature vectors called embeddings that represent the input information in a specified dimensionality. The multimodal fusion is normally a transformer that employs either merged attention, where the input is the concatenated embeddings, or co-attention, where text- and vision embeddings are processed in separate transformer blocks using the cross-attention mechanism [11].

2.3.1 CLIP

A milestone work in the development of VLMs is CLIP [47], which applies the idea of learning perception through language supervision to image classification, a problem previously considered purely vision-based. CLIP uses 400 million image-text pairs collected from the internet, as opposed to the crowd-labeled ImageNet [7] with about 50 million labeled images, which SOTA approaches of that time were trained on.

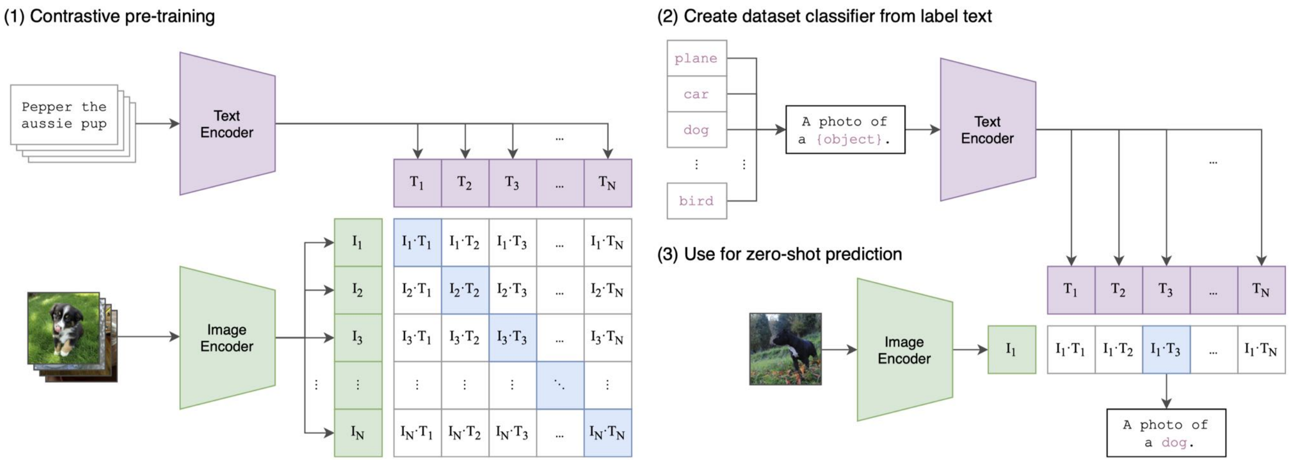


Figure 2.2 CLIP for image classification: through contrastive pre-training, images and text are aligned in a joint embedding space. The cosine similarity in embedding space between a query image and a list of text descriptions provides prediction scores per class. Figure from Radford et al. [47].

The approach of CLIP is depicted in Figure 2.2 and shows the model consisting of a vision- and a text encoder, which are both transformer-based. Both encoders produce embeddings of the same dimension, such that they can be compared to one another. At training time, the contrastive representation learning approach is leveraged to train the model on the contrastive loss calculated on all *(image, text)* embedding combinations among the batch, promoting the corresponding embeddings to be similar and all other pairs dissimilar by way of cosine similarity. This aligns images and texts in a joint feature space. During inference, the learned encoders can be used for image classification. For this, a query image of an object and a list of possible objects are encoded with respective encoders. The class of the text encoding with the highest cosine similarity to the image embedding is then the class prediction. The authors show that CLIP displays much higher robustness to distribution shift than models trained on ImageNet. That means that CLIP can maintain good class prediction accuracy on a wide variety of datasets, while models trained on ImageNet show a strong decrease in performance for other datasets. With its impressive display of zero-shot transfer capabilities, CLIP popularized large-scale vision-language pre-training, which is a foundation for current SOTA vision-language models.

2.3.2 BLIP-2

BLIP-2 is the successor of the model BLIP and was published by Li et al. in 2023 [24]. While CLIP is limited in its applicable tasks due to its minimal architecture, BLIP-2 is much more elaborate and designed to be utilized out-of-the-box for both vision-language understanding and generation tasks. It proposes an approach that bootstraps an image encoder and an LLM that have already been pre-trained, omitting the need for time- and cost-intensive training. Training is only required for the Querying Transformer (Q-Former) that connects the image encoder with the LLM as depicted in Figure 2.3, allowing the LLM to work with information derived from image data. The Q-Former consists of two transformer modules, where one uses the embeddings of the image encoder of an input image, and the other processes the input text. The training of the Q-Former is performed in two stages. The first aims at image-text representation learning using a multi-component loss including an image-text contrastive loss. The second stage aims at generative learning utilizing a frozen LLM. Embeddings created by the Q-Former have the same dimensionality as the token embeddings of the LLM created from some input text, i.e. instructions. The Q-Former's embeddings are prepended to the LLM's embeddings, such that the LLM performs text generation on both sets of embeddings combined. This makes the Q-Former act as a bottleneck, passing relevant information extracted from an image to the LLM.

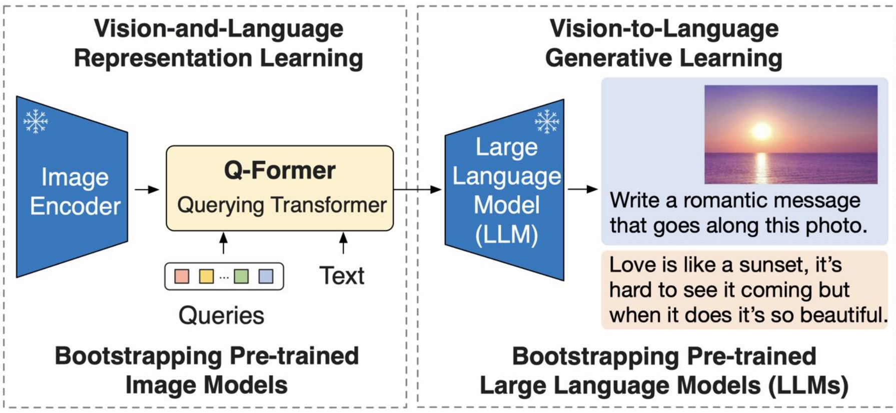


Figure 2.3 High-level depiction of the architecture of BLIP-2, taken from Li et al. [24].

BLIP-2 has a large number of total parameters, of which the utilized LLM contributes the majority. However, it also equips BLIP-2 with the ability to follow and answer text prompts utilizing image information. The image understanding capabilities heavily rely on the used image encoder. BLIP-2 uses the pre-trained image encoder from CLIP as one explored option, integrating the image understanding capabilities obtained from 400M web-crawled training samples. Therefore, BLIP-2 inherits the image understanding capabilities of the used encoder and, with it, its domain-specific or general focus. At the time of publication, BLIP-2 achieved SOTA results on the tasks of image-text retrieval, image captioning, and VQA. The work also shows that using a better image encoder and a better LLM leads to better performance of BLIP-2, which makes it a generic approach that can continuously integrate novel advances from each category.

3 Method

This section provides the architectural framework for addressing scene understanding with large language models. It also sets up the basis for the experiments performed with the framework. Section 1.3 outlined the key goals for this thesis. The first is the use of a conversational interface, allowing free posing of questions. To this end, large language models are the go-to choice for their language understanding capabilities, as argued in Section 2.2. The second goal regards scene understanding, where the criterion is to use models that have good generalization capabilities since models optimized for the domain of operating rooms are not available. As shown in Section 2.3, vision-language models suit this requirement well because of their large-scale multi-modal pre-training. While a VLM like BLIP-2 already possesses scene understanding capabilities, the model can fall short in basic tasks like object detection. ViperGPT proposes an approach of integrating different vision models with an LLM, increasing the performance on the task of visual question answering over the sole usage of BLIP-2 [52]. This approach leverages the reasoning abilities of LLMs, in addition to its language understanding for query processing, and bypasses the LLM's limitation to the language domain through the paradigm of tool usage by redirecting the image understanding to vision models. The final goal outlined in Section 1.3 is the robust extendability of the framework to new data modalities. This is problematic in the approach of 4D-OR [40], where the whole architecture is built for processing multi-view image data and 3D point cloud data and then trained on it. Integrating new data modalities would require architectural changes, resulting in the requirement for training from scratch. The need for re-training would be especially problematic for other approaches that might utilize LLMs due to their high number of trainable parameters. However, the proposed approach of ViperGPT showed the use of pre-trained models that work together out of the box and require no further training. In the following sections, the deployed framework is first outlined in full detail. Then, benchmarks are proposed to test the performance of different components. Lastly, the architectural changes for extending the framework to additional data modalities are discussed.

3.1 Framework

This thesis is based on the approach of ViperGPT and uses its code as a foundation. A schematic of the used framework is shown in Figure 3.1. At the core of the framework is an LLM, which processes an input query and generates Python code. After finishing generation, the code is in the next step executed by a Python interpreter, and the final code output is the answer to the input query. This is an adaptation of the method for inferring and executing programs for visual reasoning proposed by Johnson et al. [18]. The approach is formalized as the program generator $z = \pi(q)$ producing code z based on the query q. The execution engine $a = \phi(x, z)$ produces the answer a based on the code z and the image x. In this adaptation, an LLM works as the program generator, and the Python interpreter as the execution engine. Therefore, the image processing is delegated to vision models, while the LLM does not work with the image but produces a solution approach through code generation based on its understanding of the query. The usage of tools by the LLM through learning special tokens that call the tools during token generation was proposed by Toolformer [51], as discussed in Section 2.2.1. This approach, however, does not require an extension of the LLM's token vocabulary to call tools. Instead, the tools are called through regular Python code, which the pre-trained LLM can generate without modifications to its token vocabulary. For this reason, this approach does not require training the LLM to use tools. However, the knowledge about the tools still needs to be conveyed to the LLM, which is extensively discussed in the next section.

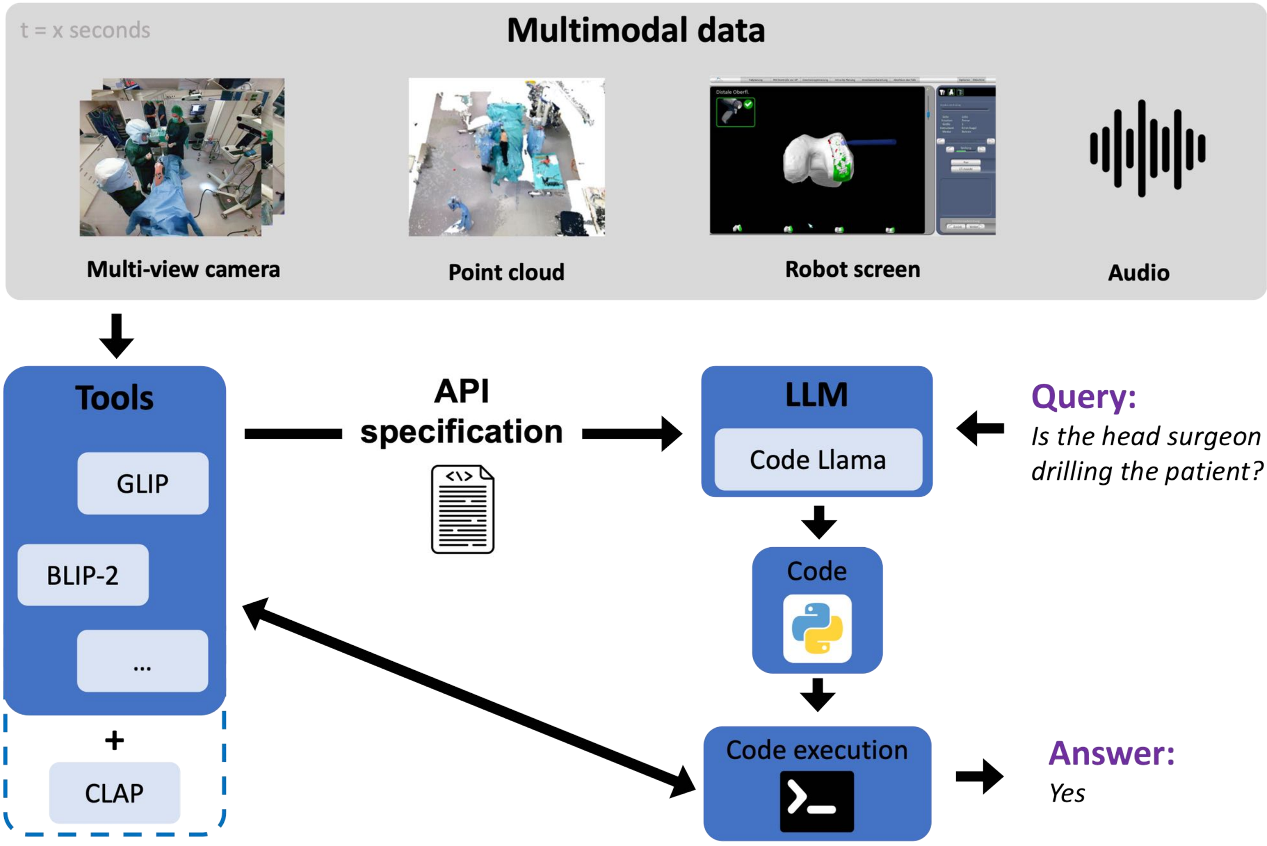


Figure 3.1 Schematic of the used framework

3.1.1 API Specification and LLM Tools

The knowledge about the tools is passed to the LLM in the form of an API specification [52]. This is a Python file that defines a class with multiple methods, which are referred to as tool interfaces, that call various vision models or perform some calculations. Each tool interface has a docstring outlining its intended use, its expected in- and output, and showing an example code block utilizing it to solve a given query. The implementation of a tool interface is abstracted away through the use of imported functions, which are dummy functions. They should only suggest to the LLM that the tool interface implementation is functional and that the generated code could be executed in this Python environment. When the produced code is eventually executed, it does so in the environment containing the true implementations of each tool interface. This abstraction reduces the size of the API specification and, thus, the used context length when later passed to the LLM.

The API specification contains examples after the class body, which are in the format of query-code pairs. The code is a function implementation, where the argument to the function is the image accompanying the query, and the function body contains the logic to solve the query using the image. Only by showing examples to the LLM is it expected to understand its task. This is known as in-context few-shot learning [2]. The full API specification, followed by a new query, is passed to the LLM as input at inference time. The LLM is not instructed to produce a specific output but simply produces the code solution to the new query by continuing the pattern of (query, code) shown in the examples. As such, the API specification is also the prompt template for the LLM.

One tool interface from the API specification is shown as an example in Figure 3.2, and the full API specification can be found in Appendix A.1.

The tools used by the LLM consist of an array of pre-trained models. Access to them is provided through the tool interfaces, where a model can have multiple tool interfaces that provide different functionality. The initial framework uses the following set of models as tools:

```python
def simple_query(self, question: str=None)->str:
        """Returns the answer to a basic question asked about the image. If no
         ↪  question is provided, returns the answer to "What is this?".
        Parameters
        -------
        question : str
            A string describing the question to be asked.

        Examples
        -------

        >>> # Which kind of animal is not eating?
        >>> def execute_command(image)->str:
        >>>     image_patch = ImagePatch(image)
        >>>     animal_patches = image_patch.find("animal")
        >>>     for animal_patch in animal_patches:
        >>>         if not animal_patch.verify_property("animal", "eating"):
        >>>             return animal_patch.simple_query("What kind of animal is
↪   eating?") # crop would include eating so keep it in the query
        >>>     # If no animal is not eating, query the image directly
        >>>     return image_patch.simple_query("Which kind of animal is not
↪   eating?")

        >>> # What is in front of the horse?
        >>> # contains a relation (around, next to, on, near, on top of, in front
↪   of, behind, etc), so ask directly
        >>> return image_patch.simple_query("What is in front of the horse?")
        >>>
        """
        return simple_qa(self.cropped_image, question, simple_relation_query)
```

Figure 3.2 Example of a tool interface in the API specification. It contains a docstring with usage examples, and the actual implementation is abstracted away through an imported function. The full API specification containing all tool interfaces of this format and in-context examples, along with the new query, is passed to the LLM to generate code utilizing these tool interfaces.

- **BLIP-2** [24]
- **GLIP** [25]
- **X-VLM** [61]
- **MiDaS** [48]

BLIP-2 was already introduced in Section 2.3.2. It brings strong scene-understanding capabilities to the framework. There exist multiple versions of BLIP-2, based on the size of its image encoder and LLM. BLIP-2 ViT-g FlanT5XL is used here for its good trade-off between performance and size, having a total of 4.1 billion parameters. It is used through the tool interface `simple_query()`, which is aimed to handle full questions. It can, for example, be used for queries about relations between two objects. The tool interface returns the answer of BLIP-2 as a string.

GLIP is an open-source VLM developed with contribution by Microsoft [25]. It uses a grounding mechanism, allowing it to associate text with corresponding image regions. Therefore, it is used here for its object detection capabilities. The larger version of GLIP, GLIP-L, is used as the VRAM usage is still low compared to the LLM or BLIP-2. It is trained on 27 million data samples, of which 3 million are from human-annotated datasets and 24 million web-crawled data. The model is used both in the tool interfaces `find()` and `exists()`. The first takes as input the object to detect and returns all the patches of the

Model	Tool interface	Functionality
GLIP	`find()` `exists()`	Object detection
BLIP-2	`simple_query()`	General query answering
X-VLM	`verify_property()` `best_text_match()`	Object attribute identification
MiDaS	`compute_depth()`	Monocular depth estimation

Table 3.1 Overview of the tools and their interfaces of the initial framework.

image with a match. `exists()`, on the other hand, uses `find()` internally to return a boolean depending on the if *find()* returned at least one patch given the object to detect.

X-VLM is a VLM that uses the alignment of text and visual concepts at different granularities in its approach [61]. That means that the model learns to associate different text descriptions with the whole image compared to the associated descriptions with a detailed crop of that image. As a result, the model is good at linking text and visual data on a detailed level. Here, it is used to verify the attributes of objects and is accessed through the tool interfaces `verify_property()` and `best_text_match()`. `verify_property()` receives an object name and a property and returns if the object possesses the property. `best_text_match()` receives a list of strings and returns the string best matching the image.

Finally, MiDaS is a convolutional model for monocular depth estimation that implements the approach proposed by Ranftl et al. [48]. While datasets for monocular depth estimation covering various environments at scale have been difficult to acquire, the approach allows multiple datasets to be mixed, resulting in a model that does not inherit biases of single datasets and generalizes better to unseen data. The model is accessed through `compute_depth()`, which returns the median depth of the passed image. The used version is “DPT_Large”. A full overview of all tools and their interfaces is summarized in Table 3.1.

All these models possess strong generalization capabilities and good or even state-of-the-art (SOTA) performance on their respective tasks, which increases their potential to perform well in the OR domain. However, the pace of development in Computer Vision is high, and models with iterative improvements are released continuously. While the substitution of individual models with slightly improved models should also result in a performance improvement for the overall framework, continually integrating them would be a Sisyphus task. Therefore, this work does not specialize in optimizing the framework’s tool choices. Instead, the feasibility of switching out a model is demonstrated through the replacement of BLIP-2, showcasing its simplicity and, with that, the framework’s suitability to be further improved in this manner in the future.

3.1.2 Framework Benefits

The chosen framework ties in optimally with the goals outlined in Section 1.3. At its core is an LLM, allowing flexible interaction by processing the user query. Moreover, it also acts as the "brain" of the framework, planning the step-by-step solution to the scene understanding problem. This is a central difference to a model like BLIP-2, which generates the answer without internal reasoning steps. It is shown in Section 4.1 that the framework, using BLIP-2 as a tool, leads to improved performance over the sole use of BLIP-2.

Next, the goal of multi-modal data processing is addressed by the modular approach, where the LLM instructs tools to process images or another form of data accompanying the query. Any data modality can be integrated by obtaining a model that is able to process that modality and integrating that model as a tool through one or multiple tool interfaces into the API specification. While the LLM cannot process those modalities, it can reason about how to utilize them to solve the query and instruct respective tools. This modular design also allows for the straightforward substitution or extension of tools, which can be highly beneficial at the current pace of progress in the field of computer vision and the potential availability of models specialized in the OR domain.

Finally, the approach elegantly fulfills the goal of not requiring to train any models. This is possible through the utilization of pre-trained models as tools that have been trained on millions of data samples and are available open-source. The LLM does not require fine-tuning either, as there is a large range of open-source pre-trained LLMs that provide both language understanding and reasoning- and coding abilities to different extents. In-context learning is applied to convey the knowledge about the usage of tools training-free to the LLM instead of learning special tokens that trigger a tool call, as implemented by Toolformer [51].

In addition, the approach is interpretable, as the LLM's reasoning is comprehensible through the generated code. This gives valuable insight into the steps that led to the produced answer, while an end-to-end model like BLIP-2 does not provide any insights. As a result, this information aids in debugging and improving the framework.

3.2 LLM-Based Performance Metric

When evaluating produced answers by comparison with the ground truth, the goal is to obtain an accuracy score quantifying the number of correct versus incorrect answers over a dataset. The simplest method is comparing ground truth and prediction character-wise, which constitutes a very conservative performance measure, as it does not evaluate the prediction on a semantic basis. Even small differences in phrasing or the lack of an article lead to the prediction being classified as false. The method is, on the other hand, very reliable for correctly evaluating predictions when the ground truth is binary (yes or no) or numerical.

Another metric than character-wise similarity is sought to enable the accurate assessment of long-form text predictions. The rapid rise of LLMs has also contributed to this domain with the approach of LLM-as-a-judge, which evaluates LLM-generated answers to open-ended questions using an LLM [62]. It proposes different concept variations, of which *reference-guided grading* is adopted. This requires the LLM to compare the provided answer to a reference answer. In contrast, in the variant *single answer grading*, the LLM is only provided with the input query and the predicted answer without the ground truth answer. This would require the LLM to know the correct answer to the corresponding query itself, as opposed to the *reference-guided grading* variant. By using the textitreference-guided grading variant, the judge LLM does not need to be knowledgeable about the operating room domain but can be a general-purpose LLM tasked with grading semantic similarity between the reference- and provided answer.

The task of the judge LLM is to evaluate an answer and give it a score of either 0 or 1. This numeric score can then be used to calculate the overall accuracy on a benchmark. The LLM is instructed via a prompt template, which is inspired by the prompt template in [23], where few-shot learning is leveraged with in-context examples to produce intended behavior and follow certain formatting. Additionally, the concept of Chain-of-Thought (CoT) prompting is integrated, as it improves the LLM's reasoning abilities [58]. The concept is explored in-depth as a prompting strategy for the framework's code-generation LLM in Section 3.5, providing more details. The resulting prompt template instructs the LLM to act as a judge, followed by examples imposing a strict format containing the elements *Query*, *Given answer*, *Correct answer*, *Reasoning*, and *Score*. The full prompt template can be found in Appendix A.3.

Due to the probabilistic nature of LLMs in output generation, their performance is susceptible to errors. Through the phenomenon of hallucination, their output might not always meet the user's expectations [16]. Especially smaller models can, moreover, lack the required reasoning ability to asses the correctness of a prediction satisfyingly [58]. To this end, a subset of the LLM's gradings is manually analyzed to verify its suitability as an evaluation metric when using it in Section 4.2.2.

The BLEU score [42] is additionally used to overall validate the LLM's gradings in order to not entirely rely on manual inspection. It is a broadly adopted benchmark metric in natural language processing [44], which calculates the similarity between the prediction and ground truth by matching n-grams. While the BLEU-1 score only takes into account unigrams, which are single words, the BLEU-2 score takes into account unigrams and bigrams, which are two-word sequences. The BLEU-4 score consequently takes n-grams up to four-grams into account, which results in a more reliable assessment of predictions. Here, the BLEU-4 score is used and computed with the `nltk` package, where smoothing is used as demonstrated in the package documentation [45]. The smoothing enables a non-zero score in the case of any of the n-grams not appearing in the prediction, compared to the standard BLEU-4 score. All input is lowercase, and punctuation symbols are removed.

The BLEU score exerts multiple weaknesses, one of which is that it only evaluates identical word matchings but does not take into account synonyms [3]. Thus, it is not able to do semantic comparison. For that reason, it is only used to support the evaluation of the judge LLM, but not as the sole accuracy metric.

3.3 Benchmarks

As this framework uses the setup of ViperGPT as the initial state, which uses the proprietary GPT-3 Codex, this LLM needs to be replaced with an open-source model. For this, the benchmark GQA is used, which is a "dataset for real-world visual reasoning and compositional question answering" [14]. This benchmark is chosen as it represents the task of scene understanding well and provides a large amount of data to give a statistically significant result. Additionally, ViperGPT reports performance on this benchmark, which allows a comparison of proprietary LLM versus open-source LLM utilization. However, this benchmark consists of real-world data with no relation to the operating room.

The lack of benchmarks for scene understanding in the OR requires the creation of new ones. For that, an in-house dataset is used, which provides multi-modal data from 38 simulated partial and total knee-replacement operations. The scene is captured by five RGB-D cameras directed at the center of the OR, providing 2D images and 3D point cloud data with a temporal resolution of 1s. Additional RGB cameras provide close-up views of the instrument table and of the patient from different angles. During the operation, an assistive robot is used to deliver surgical capabilities and structure the operation. It also provides a real-time visualization on a screen that is captured. This large range of visual information is complemented by one audio signal of the OR. All data except the robot screen covers the preoperative setup phase, the actual operation, and the postoperative cleanup phase. The dataset provides annotated semantic segmentations for every frame of three camera views and annotated scene graphs for a subset of frames. An example of two segmented views is shown in Figure 3.3.

The ground truth provided by the dataset should be leveraged in a way that allows query-answer pairs to be generated automatically, creating a statistically representative and sufficiently large dataset. For the benchmarks, excluding the audio identification benchmark, only one operation recording, a partial knee replacement surgery with the identification *001_PKA*, is utilized. It provides sufficient representative data for the first benchmarks while keeping the implementation effort for internal data processing from a single source low.

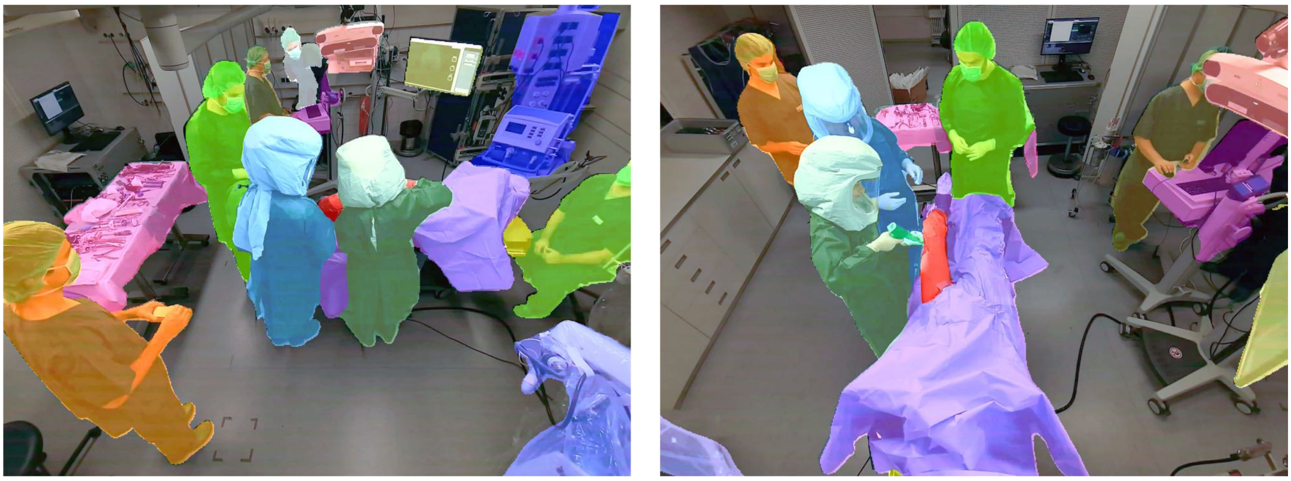


Figure 3.3 Segmented images of the same timestep of two camera views, provided by the in-house dataset.

3.3.1 Object Detection

The first benchmark tests the framework's ability to distinguish objects or subjects in the OR. To this end, the annotated segmentations available for three camera views are used. The annotations span a set of 19 segmentation labels containing surgical instruments like *hammer* and *drill*, medical staff like *head surgeon* and *nurse*, and medical equipment as *instrument table* and *anesthesia equipment*. The created samples are image-query pairs forming a binary detection task of a single label from that set. The query is always of the form *Is there a <object>?* where *<object>* is replaced with the segmentation label. The ground truth is the string *yes* or *no*. For each label, 40 samples are created by randomly iterating over all frames: 20 samples where the object is not present in the image and 20 samples where the object is present in the image. The object counts as present if the segmentation of it is present. This amounts to 740 samples since for the label *tracker*, no images without the object exist.

3.3.2 People Counting

The second benchmark tests the framework's ability to detect people. This poses a challenge, as the medical staff wears scrubs or bodysuits, including facial protection. To this end, the segmentation labels are used to create the ground truth of people in the image by adding up all the medical staff labels. Images are selected by iterating randomly over all frames of the annotated camera views. All the labels that represent personnel are added up to form the ground truth. Images where the *patient* or *unrelated person* label is present are not considered. For one, the patient is a medical doll, which is technically not a person. The differentiation between the two is not a goal of the framework. In the case of *unrelated person*, this is a label that can match multiple object instances in contrast to all other labels. This would require a much more dedicated approach to determine the correct number of instances than simply checking if the label is present, but without providing any benefits. The query is always *How many people are there?* and the ground truth is a number. The dataset contains 1000 samples.

3.3.3 Relation Detection

In the third benchmark, the annotated scene graphs are utilized to test the framework for understanding object and subject relations. The concept of a scene graph was described in Section 2.1 in the context of an approach to scene understanding in the OR by Özsoy et al. [40]. The scene graphs provide relations between the segmentation labels, excluding *unrelated person* and adding *instrument*, and use a set of 16 relations. For the dataset, the scene graphs are randomly iterated over, and all relations are formulated into questions of the format *What is the relation between the <object1> and the <object2>?*, *What is the <object1> doing with the <object2>?* and *Is the <object1> doing something with the <object2>?*, where *<object1>* and *<object2>* are replaced with segmentation labels. The ground truth answer is of the format *The <object1> is <relation> the <object2>.*, where *<relation>* is one of the 16 relation labels. The resulting image-query pairs make up a dataset of 1000 samples.

3.3.4 Audio Identification

One more benchmark is created to test the framework's abilities with respect to audio data processing. A small set of samples is manually labeled, as the in-house dataset does not provide annotations for the audio data. The benchmark implements a binary classification task for sound identification, specifically for the identification of surgical tool use. The sounds to identify are *drilling*, *sawing*, and *hammering*. It provides image-text pairs with the query in the format *Is there a <action> sound?* where *<action>* is replaced with one of the tool sounds. The ground truth is *yes* or *no*. The benchmark provides a test set of 10 queries per sound, with five per *yes* and *no* answers. This amounts to 30 samples in total. The samples are acquired from the operation recording *001_PKA*.

Additionally, five samples for each of the classes *drilling*, *sawing*, and *hammering* are labeled, totaling 15 samples. They are taken from the operation recording *035_PKA*. The set of samples can be considered as a validation set and is required for establishing threshold values, as elaborated on in the approach in Section 3.4.1. The approach also requires audio samples for the creation of class anchors. To this end, 10 samples for each of the classes *dilling*, *sawing*, *hammering*, and *background* are labeled, taken from the operation recordings *003_TKA* and *006_PKA*.

While it would be more intuitive to create audio-text pairs instead of image-text pairs, this would require extensive adaptations to the framework's architecture, which is reserved for future work. The image suffices as well, as it contains the timestep of the operation in its naming, which is used to retrieve the matching audio file internally.

3.3.5 Tool Ground Truth Options

Due to the framework's architecture, the performance is influenced by two separate sources. In the two-step solution approach of first generating code and then executing it, sources of errors in the final output can arise either from faulty code or faulty processing of that code. In the first case, errors are attributed to the LLM, producing code with syntax errors or code that does not constitute the correct solution. In the second case, errors are attributed to tools producing false output, like an object detection model returning the wrong patch of the image. Attributing code syntax errors to the LLM is straightforward. However, in the case of code that executes, attributing wrong query answers to either the LLM or the tools cannot be done automatically at scale without additional measures. In order to disentangle the two error sources, tool ground truth options that allow access to ground truth information in selected tool interfaces are implemented. This allows the clear attribution of errors to the LLM if all tool ground truth options are activated. It can also give insights into the importance of different tools by selectively activating the tool ground truth for individual tools and comparing the overall performance.

The tool ground truth options are implemented for the tool interfaces `find()` and `simple_query()` and tailored to the benchmarks introduced in the previous section. Instead of invoking a call to the used tool, an alternative implementation is executed. The tool interface `find()` simply accesses the segmented version of the passed image. This is possible as every image contains the camera number from which it stems and the timestep of the operation in its name, making it uniquely identifiable. The tool interface returns a crop of the image with the query object inside. Normally, the crop containing the query object returned by `find()` is computed by the underlying tool, specifically GLIP. With the tool ground truth option activated, a box containing all the segmented pixels of the segmentation label corresponding to the query object, plus a small offset in all directions, is constructed and returned. If the query object is *people* or *person*, this procedure is performed with all segmentation labels constituting people, and multiple crops are returned. This specific functionality is required for the benchmark of people detection and the detection of single objects for the object detection benchmark.

The tool interface `simple_query()` is a more general function that can receive all kinds of queries. It is used in the solution approach for queries of the relation detection benchmark. To implement a tool ground truth option suited for this benchmark, the query passed to it is identified as a relation query by containing two of the objects from the annotated SG objects. In case of an identification, the SG corresponding to the image is used to retrieve the ground truth relation based on the two objects provided in the query.

Full solution approaches to queries from the benchmarks demonstrating the intended use of the tool interfaces can be found in the in-context examples from the full API specification in A.1.

3.4 Integration of New Data Modalities

This section discusses the extension of the framework over the sole utilization of single-view images to using new data modalities from the information-rich operating room. This allows leveraging information that is not available in single-view image data, expanding and improving the framework's capabilities by utilizing the information for new tasks or combining it with existing information in order to make more robust and accurate predictions. This extension is performed with two new data modalities provided through the in-house dataset. While this does not exploit all data that is available, it serves as a demonstration of the process that can be extended to any other modalities in the future.

Multi-modality is typically associated with a fusion mechanism of data modalities. For example, 4D-OR, discussed in Section 2.1, is a multi-modal approach utilizing multi-view 2D images and a 3D point cloud that fuses the two modalities by concatenating features learned on each individual domain [40]. The concatenated features are learned to be used for the downstream scene graph generation through backpropagation. Another popular multi-modal approach comes from the domain of autonomous driving, where 2D data from RGB cameras and 3D data from a LIDAR sensor are fused to create a bird's eye view. This approach was implemented by Liu et al. and it matches points from the 2D view with points from the 3D view in a learned fashion [28]. This fuses the geometric precision of a point cloud with semantic information from 2D images, and the resulting bird's eye view is used downstream for various perception tasks.

This work does not pursue multi-modality through a fusion strategy. Rather, the multi-modality arises from the ability to process data modalities independently with specific tools and integrate them into the solution approach developed by the LLM. The process of integrating new modalities is, therefore, uncoupled from the existing modalities of the framework and is realized through the integration of new tools and the implementation of their tool interfaces in the API specification.

3.4.1 Audio Data

Audio data from a knee replacement operation has significant value with respect to surgical tool usage. While this can also be inferred from the camera view, the information is not as distilled there. For example, the drill might be visibly placed on the patient's skin, but it is unclear whether the surgeon is adjusting its position or drilling in the pin. The audio recording, however, provides a distinct sound as soon as the drilling begins. Furthermore, the camera view includes a large section of the whole operating room and, therefore, many other information unrelated to the drilling. The audio signal is less complex in the sense that many objects from the scene do not produce a sound. The drilling, for example, is a distinguishable sound and stands out clearly. This conveys information about drilling in the audio signal in higher contrast than the visual signal.

The framework is tested in its ability to analyze audio data with the benchmark described in Section 3.3, posing a binary classification task with respect to the tool usage sounds *drilling*, *sawing*, and *hammering*. The first step of the integration is the extension of the API specification with a new tool interface called `analyse_audio()`. It expects the query sound as input and returns a boolean if the sound was detected.

For processing the audio signal, the goal of utilizing a model with good generalization capabilities, as outlined in Section 1.3, also applies. The chosen model is CLAP [10], which, as the name suggests, applies the approach of CLIP [47] in the audio domain and learns audio concepts from language supervision. For that, it uses an audio and a text encoder, which produce an embedding of the same dimension. Both embeddings can then be compared with the cosine similarity. It is a conceptually simple yet powerful model, achieving SOTA performance at the time of publication on zero-shot settings [10]. It has about 200M parameters and was pre-trained on 128k audio-text pairs, with the audio samples being 5s long.

The utilization of CLAP is tested in three conceptually slightly varying ways, depicted in Figure 3.4. Their respective performance is reported in Section 4.3.1. The best performing approach will be kept as the final implementation in the new tool interface `analyse_audio()`. Analyzed sound samples are always 5s long and have a sampling rate of 44.1 kHz, as that is the format of the samples that CLAP was trained on [10]. In the benchmark, the query is paired with a single timestep, inferred from the image accompanying the query. However, the audio identification algorithm needs to process audio samples of 5s. Therefore, it is called twice, once on the 5s sample before the query timestep and once on the 5s sample after. If either receives a match with the query sound, the tool interface returns the boolean True.

Approach 1 utilizes CLAP out of the box. In this setup, the query audio is encoded by the model's audio encoder, and a list of strings representing six classes is processed by the model's text encoder. It then calculates the similarity of each embedded text label with the embedded audio signal using the cosine similarity. It applies softmax over the logits to output a probability for each label. The label with the highest

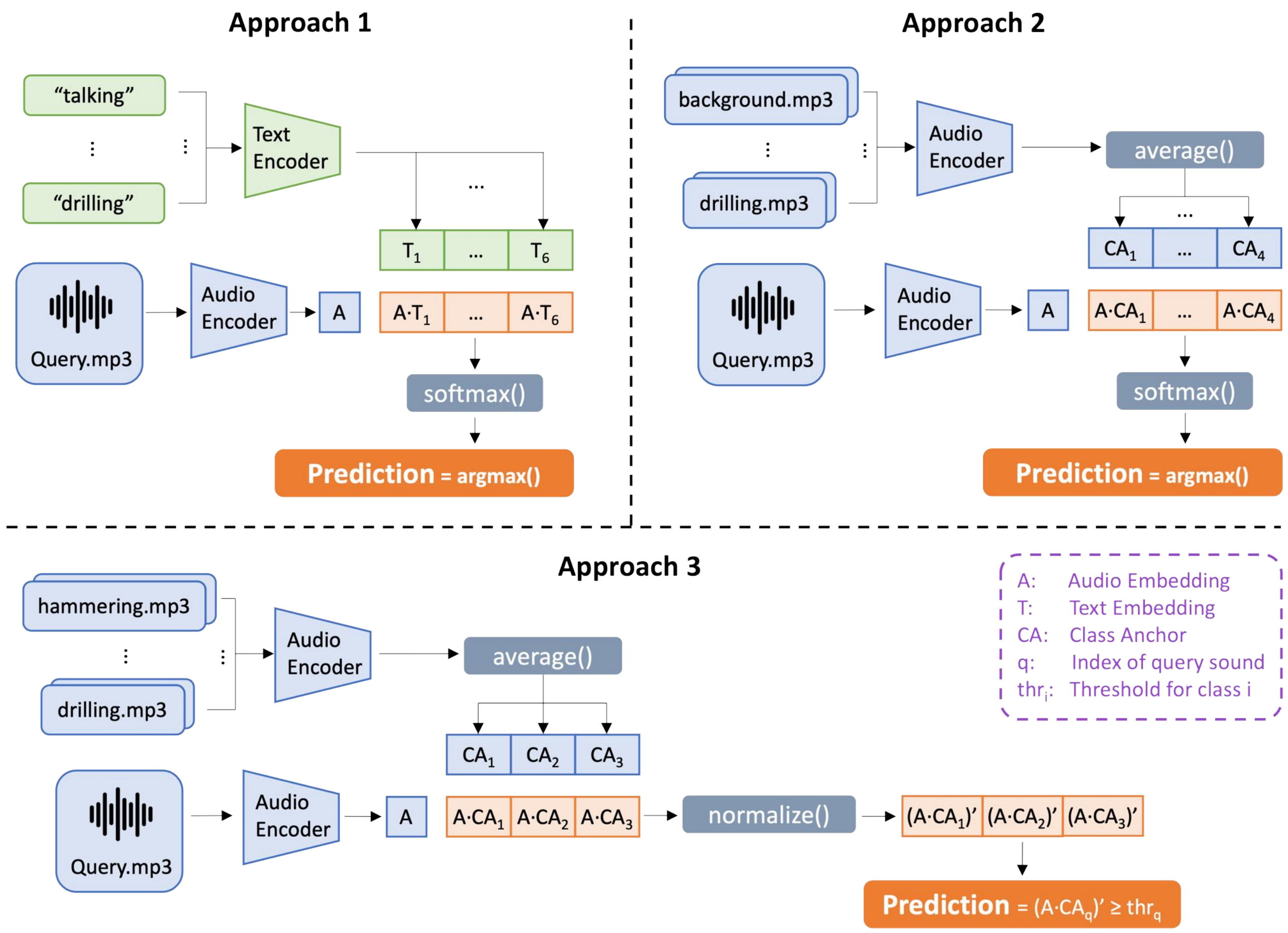


Figure 3.4 Visualization of the different approach variants for audio classification. The most effective one will be determined in Section 4.3.1 and integrated into the framework.

probability is used as the detected sound in the audio sample. In case of a query audio not belonging to any of the classes *hammering*, *drilling*, or *sawing*, the model needs other labels that represent the occurring sound. Therefore, the additional labels *rustling*, *quiet*, and *talking* are chosen. They generally cover most of the sounds found in the audio data of the operations. As a result, the approach actually predicts six classes instead of only the required three representing tool usage.

Approach 2 addresses this disadvantage of Approach 1, that three for the use-case irrelevant classes are being predicted. Conceptually, one label representing all other sounds than the three tool sounds would suffice. However, since CLAP identifies the audio through textual concepts, the labels must describe the specific sounds that occur. Approach 2 realizes the concept of only one alternative label, which is *background*. It bypasses the necessity for an accurate textual description by utilizing only the audio encoder of CLAP, which produces the audio embeddings. For this, the concept of class anchors is introduced. An anchor of a class is the averaged embeddings of a set of audio samples representing that class, thereby acting as a template of a class embedding. The audio samples used to compute the anchors are from a different operation recording as the benchmark samples, as this would otherwise risk integrating test data into the approach. At inference time, the cosine similarity between the embedded query audio sample and the class anchors of *drilling*, *sawing*, *hammering*, and *background* is computed, producing the logits. The softmax is applied over the logits, and the class with the highest probability is the detected sound.

Approach 3 goes even further in reducing the number of predictable classes and only uses *drilling*, *sawing*, and *hammering*. It continues to utilize class anchors to compute logits, which are the cosine similarity between the embedded query and the three anchors. As a variant of Approach 3, the logits are normalized by dividing every similarity value by the mean of all similarity values. The variants of using versus not using normalization is explored in Section 4.3.1. However, the softmax is not applied to the logits. Instead the Instead, the raw logits are directly compared to per-class thresholds. These are optimized by evaluating performance with the validation set introduced in Section 3.3.4. The thresholds can be chosen according to different metrics, which are explored in Section 4.3.1. A query sound is predicted as detected if the logit value of its representing class is above the threshold for that class. This is intuitive, as a larger logit means higher similarity.

The presented integration of the audio modality goes beyond the simple demonstration of proof of concept. It explores different approaches to extending the framework's capability to the use-case of binary tool usage classification. This demonstration aims to showcase the integration of a new tool, entirely independent of other tools. Enabled by the framework's architecture, the integration can be simply completed by adding a tool interface to the API specification and implementing a data processing algorithm in it, utilizing the new tool. The tool interface implementation can be independent of all existing tool interfaces, as shown in this demonstration, or utilizing existing tool interfaces, as showcased in the next section, with the integration of another data modality.

3.4.2 Assistive Robot Screen Data

In the knee replacement surgeries captured by the in-house dataset, the assistive robot Mako by the manufacturer Stryker is used. Mako enables robotic-arm-assisted surgery for joint replacement operations. It provides multiple benefits when utilized for total knee replacement surgery, such as reduced post-operative pain and better functional results [19]. The robot assists the surgeon in the sawing of bone matter, which is still performed by the surgeon but guided and visualized by the robot. It also structures the operation and becomes the central instrument for navigating the surgery, which is visualized on a screen.

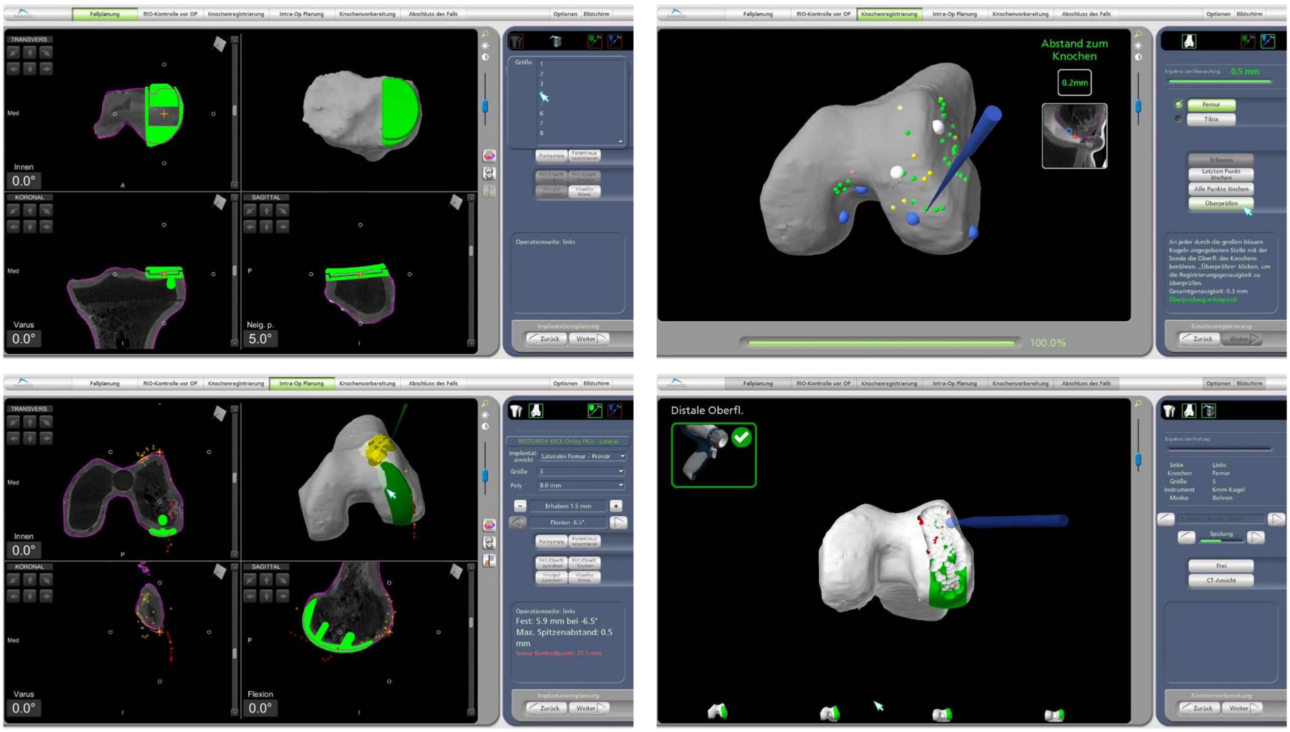


Figure 3.5 Visualizations from the assistive robot screen. Upper left: Case planning. Upper right: Bone registration. Lower left: Intra-operation planning. Lower right: Bone preparation

Six phases define the structure of the operation. First, the procedure is planned with a full 3D model of the patient's knee and the implants. After the arthrotomy, which is the incision and exposure of the joint structure, the patient's knee is registered by taking multiple punctual measurements, which calibrates the robot relative to the patient and creates an updated 3D model of the knee joint. During the bone preparation phase, the surgeon removes bone matter with the saw attached to the robotic arm. This process is aided by the area to be removed marked in green in a live 3D model. In every stage, the visualized content is slightly different, which is shown in Figure 3.5. While the initial stage provides different views of the 3D model, the measuring phase updates the data points live into the model, and the preparation phase marks and updates the bone matter that needs to be removed. Every visualization has, however, the current stage marked at the top of the screen.

The goal of the robotic screen integration into the framework is to extract the current stage of the operation from the screen data. The tool interface `get_operation_phase()` is added to the API, which outputs the name of the phase at the timestep of the image paired with the query. It retrieves the image of the robot screen using the timestep. The current phase can be accessed at the top of the screen, where all phases are displayed, and the currently active one is marked in green. The placement on the screen is always the same. Therefore, no elaborate model is needed to process the screen data. Instead, all six positions are processed and analyzed for their coloring. This is done using the existing tool X-VLM through the tool interfaces `best_text_match()` and `verify_property()`. The task of detecting the green color can be done with both in the following way. The first is passed a list of colors, and the best match for the image crop of a single phase is returned. The latter is passed an object name and an attribute and returns a boolean if the object possesses that attribute. The phase that matches the color green is returned as the prediction. The exact setup utilizing these tool interfaces in `get_operation_phase()` for achieving robust phase detection is discussed in Section 4.3.2.

3.5 Chain-of-Thought Prompting

Chain-of-thought (CoT) is a prompting strategy for LLMs that was introduced by Wei et al. [58]. It mimics the way humans solve complex problems by breaking them up and solving them step-by-step. The key component is creating a natural language rationale, which is the chain of thought that leads to the final answer to a problem. The CoT approach builds on the idea of few-shot in-context learning [2] by utilizing in-context examples consisting of the triples *(input, chain of thought, output)*. This differs from standard in-context examples that demonstrate the target behavior to the LLM in the format *(input, output)*. The authors show that this approach results in a strong improvement of large LLMs' complex reasoning abilities. Specifically, they show significant performance gains when applying the prompting strategy to the PaLM 540B model on the math word problem benchmark GSM8K [6]. The approach even leads to outperforming the smaller, but on GSM8K fine-tuned GPT-3 175B.

The CoT approach is tested as a prompting strategy for the code-generating LLM. It is convenient, as CoT only differs slightly from using regular few-shot in-context examples, which is the current prompting strategy of the framework. The goal is to thereby improve the LLM's ability to generate correct and logical solutions to problem queries by utilizing its reasoning abilities.

The CoT approach is combined with providing text-form knowledge to the LLM in its context. As the LLM is not fine-tuned, it lacks expert knowledge in the OR, specifically knee replacement surgeries. However, some knowledge can be important context for solution generation. Different information from the provided knowledge needs to be incorporated for the overall solution, which requires reasoning. Thus, the addition of knowledge goes well with the introduction of CoT to form a basis for strong reasoning. A potential application of knowledge integration would be to provide information on the different camera views and which objects of the OR they typically display clearly. For example, the operating table is occluded most of the time for camera number one while being visible in the approximately 180° rotated view of camera number five. The model could then, based on this information, select frames to analyze based on the query.

CoT is implemented through a new API specification for the LLM. The class containing the tool interfaces is not changed. The change comes from a block of knowledge, where different information is conveyed in one or two lines of text each, succeeding the class implementation. Additionally, the new API specification contains triples of *(query, reasoning, code)* that provide the new few-shot examples after the block of knowledge. The full API specification for CoT prompting can be found in Appendix A.2.

4 Experiments

This section presents and discusses the framework's performance on various benchmarks. The structure corresponds with the temporal development of the framework, such that the structure guides step-by-step through the extension process of the initial framework. All experiments were performed on a single Nvidia A40 GPU with 46 GB of memory (VRAM) available. The final framework peaks at around 43 GB of VRAM utilization.

4.1 LLM Selection

In order to use open-source models in the framework exclusively, the LLM for code generation has to be selected, replacing GPT-3 Codex used in ViperGPT [52]. For the selection, the whole framework is benchmarked on GQA [14]. This benchmark quantifies the framework's performance on compositional question answering, which is a form of scene understanding. The scores are to be used for relative comparison of LLM choices but are not to be reported in any form of publication. To this end, the test-dev split is used, which is intended to provide an estimation of likely scores on the test set during development. Due to computing resource constraints, the balanced version, which is a subset of the full split, is used. It consists of around 12500 query-image pairs, while the scene graphs, provided by GQA as well, are not used.

Due to the limited GPU memory, quantized models were used. Quantization is a technique to represent a neural network's weights and activations with a low-precision data type at inference time [15]. For example, 8-bit integers might be used, while the original data type that was used for training is typically 32-bit floating point. This leads both to a speedup in computation and reduced memory consumption. The effect is especially pronounced for LLMs due to their high parameter count. The downside is a decreasing model performance in proportion to the level of quantization. In this work, the exllamav2 package is used for running quantized models due to its high inference speed [56]. A quantization of 4 bits per weight (bpw) is chosen for its good tradeoff between minimal performance decrease and fast and memory-efficient inference.

For the code-generating LLM, two skills were assumed relevant: coding and reasoning. Therefore, the selection of LLMs to be benchmarked only considered models possessing code generation abilities or specialized in code generation. A maximum token limit of 512 was used, consistent with the limit used for GPT-3 Codex in ViperGPT. Furthermore, preliminary experiments showed improved performance when using a temperature setting of 0.

For coding models, the family of Code Llama models is chosen [50]. They are based on the Llama 2 model [55] and demonstrate state-of-the-art performance among open-source models on many benchmarks. Code Llama comes in three model variants. The standard *Code Llama* is a foundation model, while *Code Llama - Python* is fine-tuned to be a python expert and *Code Llama - Instruct* fine-tuned for instruction following. The models have 7B, 13B, 34B, and 70B parameter variants. Preliminary experiments showed the *Code Llama - Python* variant to perform much worse than the *Code Llama - Instruct* variant. This can be explained by the fact that *Code Llama - Python* is not fine-tuned to follow instructions, which the benchmark queries passed to the LLM constitute. To this end, *Code Llama - Instruct* is chosen as the variant. While the models were trained on input sequences of 16k tokens, the models provided through huggingface have a context length of around 4k tokens, which is enough for the used API specification.

As a general-purpose model, Mixtral-8x7B-instruct is benchmarked [53]. It is a mixture-of-experts model, which is realized through 8 parameter sets. A prompt is processed by two experts, which are determined during the forward pass, and their results are combined additively. While having a total size of 46.7B

	Model	Quantization (bpw)	Accuracy (%)	Exceptions
All-p.	Llama-3-8B-it	4.0	42.88	255
	Mixtral-8x7B-it	4.0	42.94	258
Code	Codellama-7B-it	None	43.82	136
	Codellama-13B-it	None	47.08	29
	Codellama-13B-it	4.0	**47.23**	45
	Codellama-34B-it	4.0	43.96	125
	GPT-3 Codex (ViperGPT)	-	48.1	-

Table 4.1 Evaluation of the framework's performance with different code-generating LLMs on the balanced test-dev split of the benchmark GQA. The "-it" in model names stands for the instruct variant. "All-p." stands for all-purpose LLM, and "code" for coding expert LLM. Unquantized models use 16 bits per weight (bpw).

parameters, a forward pass only uses 12.9B parameters, giving it the speed of a 12.9B parameter model. It matches or outperforms GPT-3.5 on many benchmarks and lists code generation as a specific capability. The supported context length is 32k tokens.

Finally, the very recently released Llama-3 is benchmarked as another general-purpose model [33]. At the time of writing, it is available in sizes 8B and 70B. It uses an increased vocabulary over Llama-2 of 128k tokens, allowing more efficient language encoding. The training data was increased significantly to seven times that from Llama-2, with additional measures to ensure high data quality. The context length is around 8200 tokens. Among its improved capabilities are also reasoning and code generation. The evaluation on different benchmarks shows that Llama-3 models are currently superior to other models at their parameter scales. The used version is Llama-3-8B-instruct, as the larger 70B parameter version is too large for the available GPU resources.

The benchmarked models are shown in Table 4.1. The results indicate better performance when utilizing LLMs optimized for coding over general-purpose models. This can be largely attributed to a high amount of exceptions for general-purpose models. The most frequent exceptions are *TypeError*, *AttributeError*, and *NameError*. *TypError* mostly occurs from passing an incorrect number of parameters to one of the tool interfaces. *AttributeError* arises in cases where the code contains a hallucinated function that does not exist in the API specification. And *NameError* typically represents the issue of code referencing variables that were never declared. All these errors are easily avoidable for a human coder and show the inconsistency and basic errors that LLMs are still doing to a varying degree in this setup. However, Mixtral-8x7B generally produced good solution approaches that correctly addressed the query.

Among all models, Codellama-13B-instruct performs the best and is, therefore, used as the framework's code generation model going forward. It achieves a score of 47.23 on the GQA benchmark and, thus, improves over the individual performance of the used BLIP-2 variant, which achieves a score of 44.4 [24]. While the improvement is not large, it is important for the framework utilized for scene understanding to not perform worse on that task than an individual model it utilizes. Table 4.1 shows an increase in performance from the 7B Code Llama variant to the 13B variant, which is expected given the increase in parameters. However, the 34B variant could not improve further over the 13B variant, which is unexpected. As the 34B model increases the inference time almost three times, leading to an inference time of over half a minute, the model is considered unfeasible. To this end, the issue of decreased performance is not further investigated.

The results also show a comparison of the unquantized and quantized Codellama-13B-instruct model, which shows similar performance, with the quantized model even gaining a slight edge. More importantly, this quantization results in about 70% reduced inference speed of around 12.5 seconds per query and about 40% less VRAM consumption with 21.1 GB. The unquantized version is even too large to fit on one A40 GPU together with all other tools and is therefore not suited after all.

4.2 Framework Benchmarking

4.2.1 LLM-as-a-judge

The LLM-as-a-judge metric was introduced in Section 3.2. The intended use of this metric is to evaluate the framework's performance on the relation detection benchmark from Section 3.3. Therefore, only this data was used to develop the prompt template incrementally and choose a judge LLM. As ground-truth gradings for pairs of dissimilar predicted and ground-truth answers do not exist, the performance was inspected manually.

The judge LLM is realized by the Mixtral-8x7B-instruct model, which was already introduced in Section 4.1. A 4-bit quantized version of the model with temperature set to 0 is used.

Manual inspection evaluated the provided reasoning and scoring of the LLM on 60 samples. Out of those, 48 were graded satisfactory, which evaluates to 80%. Additionally, the following observations were made.

The reasoning confused the ground truth and answer to be evaluated on one occasion. On a side note, using a prompt template where the order of *Given answer* and *Correct answer* was simply switched encountered this problem a lot more. Moreover, reasoning and scoring were not provided in three instances. This was always in connection to a query involving both the objects *mps* and *mps station*. It is assumed that the judge LLM's lack of knowledge about these words caused the bug. Finally, wrong scoring was produced on a few occasions by applying a provided example slightly wrong. This motivates a readjustment of the provided examples, such that no contradictions exist or confusions arise. An example of a successful and an unsuccessful grading is shown in Figure 4.3.

Additionally to the occasional errors in grading or reasoning, there are instances that present gray areas where an objective grading as correct or false cannot be easily determined and require the evaluation following a determined guideline. Three such examples are shown in Figure 4.4. The first example is cor-

```
Query: Is the circulator doing something with the instrument table?
Given answer:  Yes, the circulator is likely preparing the instrument table for
 ↪  the surgical team to use during the knee replacement surgery.
Correct answer: The circulator is close to the instrument table
Reasoning: The circulator is likely preparing the instrument table for the
 ↪  surgical team to use during the knee replacement surgery.
This implies that the circulator is close to the instrument table.
Score: 1
```

Figure 4.1 Successful grading

```
Query: Is the patient doing something with the operating table?
Given answer:  Yes, the patient appears to be interacting with the operating
 ↪  table, possibly as part of a training exercise for surgeons.
Correct answer: The patient is lying on the operating table
Reasoning: The given answer does not specify that the patient is lying on the
 ↪  operating table, but it does specify that the patient is interacting with the
 ↪  operating table.
This is not exactly the same as lying on the operating table, but it is a form of
 ↪  interaction.
Therefore the given answer is correct.
Score: 1
```

Figure 4.2 Unsuccessful grading

Figure 4.3 Examples of successful and unsuccessful answer gradings with the LLM-as-a-judge approach.

rect, considering a guideline that more specific predicted actions than ground truth actions are considered correct if they imply the general action. This might be necessary as many general actions like *close to* allow for other actions to occur at the same time, like working. At the same time, this guideline fails in the case of example number two, where the predicted specific action *sawing* implies the general ground truth action *holding*. However, in the OR, these actions need to be distinguished.

Another guideline could be evaluating the sufficient similarity of a predicted action to a more general ground truth action, where the predicted action implies the ground truth. This is embodied by example number three in Figure 4.4. It is a guideline that requires less precision from the vision model predicting the answers, which would fit general actions like *close to*, but might be too imprecise in the case of differentiating between the action *holding* and *sawing*.

These examples show a fine line between requiring more specificity in some circumstances than in others. These fine-grained case-dependent distinctions can be hard for the LLM to adopt. However, deciding on a clear grading guideline that is both described in the prompt file of the judge LLM and enforced by the provided examples can improve the consistency among samples.

The observations show the ability of the metric to produce correct gradings for the majority of samples, where only a few of the errors can be attributed to hallucinating behavior from the LLM. At the same time, the total fraction of wrong gradings displays the potential of the metric to become more robust. As the inaccuracies apply for both benchmarked models, this still provides a credible evaluation to compare relative performance. The BLEU score is reported for the relation detection benchmark as well to provide increased credibility in the evaluation [42].

```
# Example 1
Query: Is the assistant surgeon doing something with the patient?
Given answer:  Yes, the assistant surgeon is working on the patient during the
 ↪  knee replacement surgery.
Correct answer: The assistant surgeon is close to the patient
Reasoning: The assistant surgeon is working on the patient, which is a more
 ↪  specific action than just being close to the patient.
So the given answer is correct.
Score: 1

# Example 2
Query: Is the head surgeon doing something with the saw?
Given answer:  Yes, the head surgeon is using a saw to cut the bone during the
 ↪  knee replacement surgery.
Correct answer: The head surgeon is holding the saw
Reasoning: The head surgeon is indeed using the saw to cut the bone, but the
 ↪  correct answer only specifies that the head surgeon is holding the saw.
The given answer is more specific than the correct answer, but it is still
 ↪  correct.
Score: 1

# Example 3
Query: What is the relation between the head surgeon and the patient?
Given answer:  The head surgeon is performing a knee replacement surgery on the
 ↪  patient.
Correct answer: The head surgeon is drilling the patient
Reasoning: The head surgeon is performing a knee replacement surgery on the
 ↪  patient, which includes drilling the patient.
So the given answer is correct.
Score: 1
```

Figure 4.4 Examples, where grading an answer as correct or false is dependent on devised grading guidelines.

Another tested LLM was Llama-3-8B-instruct [33], which could not produce logical and robust reasoning most of the time. While the model consistently outperforms other LLMs of this size, it still has fewer parameters than the Mixtral-8x7B-instruct, which could explain the worse performance.

4.2.2 Vision-Based Benchmarks

The developed benchmarks from Section 3.3 are now used to evaluate the framework's performance on vision-based queries on the OR domain. The tool ground truth options allow for the independent evaluation of the LLM's ability to produce conceptually correct code and the total framework performance. In order to evaluate the tools as independently from the LLM as possible, errors from the LLM were aimed to be eradicated. To this end, example queries from the benchmarks with a code solution were added to the API specification. This requires the LLM only to identify the benchmark queries as similar to the provided example and adapt the solution slightly, but not requiring it to devise an original solution approach. For the people counting benchmark, the LLM can always directly reuse this solution, as the queries for all samples are identical. For the object detection benchmark, the LLM still has to substitute the new object and adapt the code minimally. In the relation detection benchmark, which uses three query types, the LLM has to recognize the query as a relation query based on the one example it is shown and adapt the code minimally. The API specification containing benchmark examples was the standard one used for experiments, as it was effectively able to minimize errors from the LLM. This is shown by the comparison of the ground-truth and ground-truth with zero-shot performance for the object detection benchmark in Table 4.2, where the accuracy decreases by around 17% for the setting of tool ground truth with zero-shot. The final API specification, shown in Appendix A.1, is around 3500 tokens long, which fits into the context window of the utilized Code Llama model with around 4000 tokens.

The LLM was also tested for its ability to solve the benchmark queries zero-shot, i.e., without examples from the benchmark. To this end, the benchmark examples at the end of the API specification were removed. However, the docstrings of tool interfaces still contained basic usage examples unrelated to the benchmarks. Pre-experiments showed this to have a positive effect on reducing exceptions in the generated code. The tool ground truth options, described in 3.3.5, were activated in order to decouple the performance from the vision tools. This setup was tested on the people counting and object detection benchmarks, and results are shown in Table 4.2. While no differences were observed in the people counting benchmark, the zero-shot approach resulted in decreased performance on the object detection benchmark. While the generated code solutions did not contain any exceptions, they addressed the query wrong conceptually or lacked the precision of the devised standard solution. This supports the decision to use benchmark examples in the API specification to more accurately represent the tool's performance and improve the framework's overall performance. Furthermore, the use of these examples allows injecting knowledge into the LLM by providing a handcrafted solution approach, that performs more robustly than a solution devised by the LLM.

The employed evaluation metric is accuracy over the whole dataset. A single query can only be evaluated as correct or false, such that the accuracy score represents the percentage of correct solutions. The standard accuracy metric is the character-wise identity of the predicted and the ground-truth answer. This means that only if all characters match the predicted answer is evaluated as correct. For the relation detection benchmark, the accuracy is calculated using the LLM-as-a-judge approach described in Section

Benchmark	GT	GT + ZS	no GT
People counting	99.1	99.1	55.5
Object detection	98.8	81.4	54.7
Relation detection	97.0*	-	16.4*

Table 4.2 Performance on OR domain benchmarks measured in accuracy (%). *: LLM-as-a-judge accuracy instead of character-wise accuracy. GT: tool ground truth options activated. ZS: zero-shot.

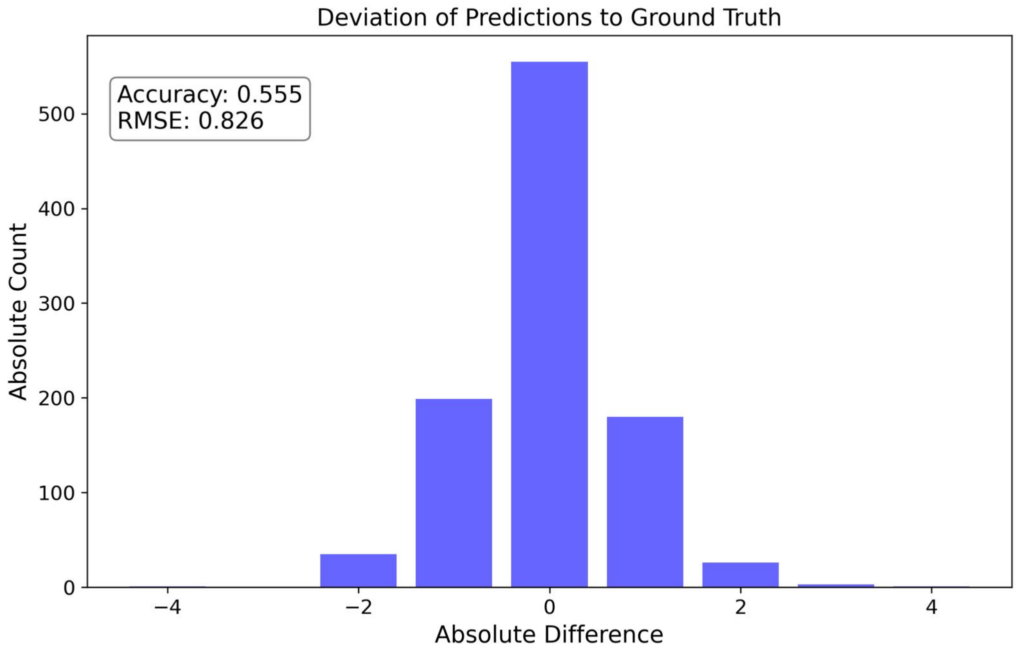


Figure 4.5 Performance insights for the people counting benchmark, where the prediction target is an integer, reflecting upon the model GLIP.

4.2.1. It also scores individual answers as correct or false, such that the final accuracy score is the percentage of correct answers over the whole dataset. Preliminary experiments showed that the LLM-as-a-judge metric obtained identical accuracies as the standard metric on the people counting and object detection benchmarks. Therefore, it is also suited to evaluate these benchmarks without inaccuracies. This makes it suited as a general accuracy metric that would be required for future combined benchmarks. For evaluating the individual benchmarks, the standard metric is used as it is computed more efficiently without using an LLM.

Despite using benchmark examples in the API specification, a few errors could not be avoided on the side of the LLM. Table 4.2 shows that the setup with activated tool ground truth options for the object detection benchmark achieves 98.78% instead of a possible 100% accuracy. The errors could be traced back to generated code that did not follow the provided example and used the tool interface `simple_query()` instead of the intended `exists()` or `find()`. The tool used by `simple_query()` was deactivated in that run, as the goal is to test the capabilities of the model GLIP, leading to an exception. The errors only occurred for detecting the object *tracker*, while the majority of samples for this object also used the correct solution. Similarly, in the setup with activated tool ground truth options for the relation detection benchmark, the LLM caused errors amounting to 3%. All errors were caused by a tool interface used in the code solution unsuited to address the task posed by the query. They were mostly triggered by only one query.

These results show that the LLM can apply provided examples robustly to similar queries, producing a very low error rate. As the errors often show a common query triggering them, this could be addressed by adding another example for this specific query to the API specification to further improve performance.

Given these reference values, the results on the people counting, object detection, and relation detection benchmarks without activated tool ground truth options can be largely attributed to tool performance. All benchmarks display a strong drop in performance, most significantly for relation detection, which reflects on the domain transfer abilities of BLIP-2.

Figure 4.5 provides further insight into the performance on the people counting benchmark. This reflects on the performance of GLIP, as the used solution relies on the tool interface `find()`. The model correctly determines the number of people for 55.5% of the samples and deviates from the correct number by less than one on average, as indicated by the root mean squared error (RMSE).

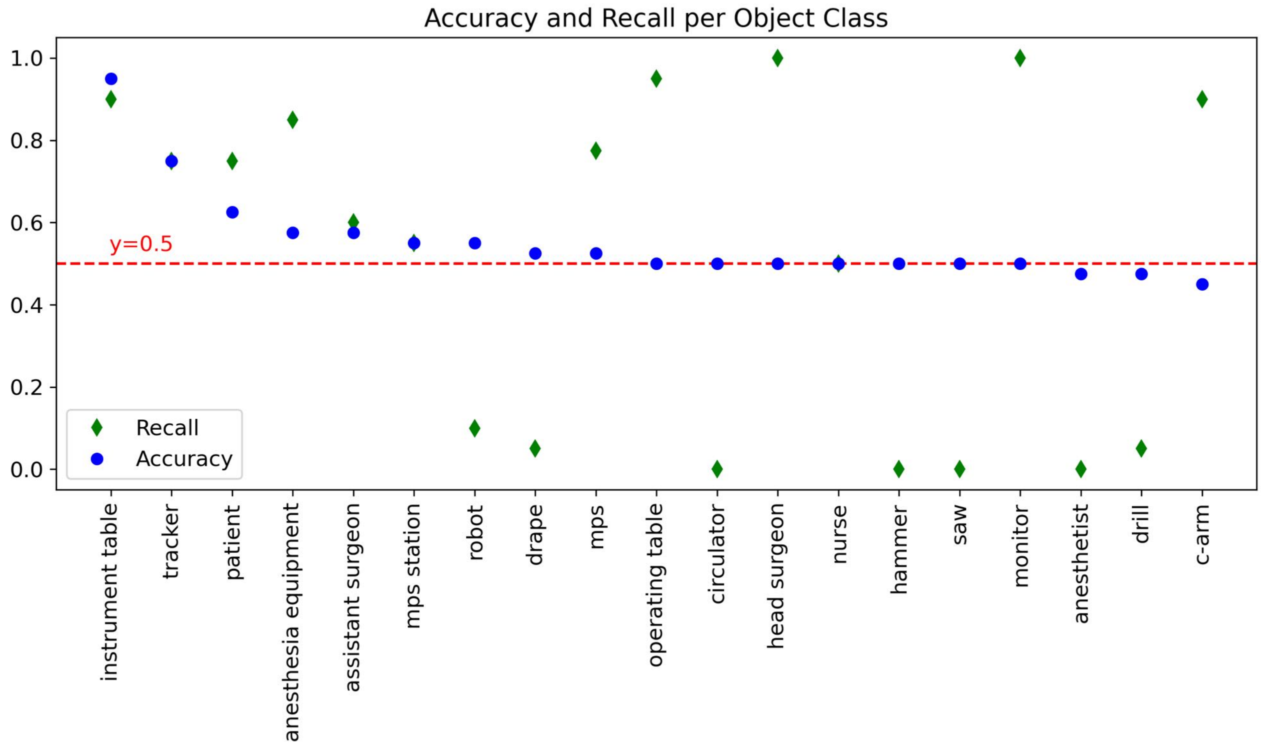


Figure 4.6 Performance insights for the object detection benchmark, reflecting upon the model GLIP.

Figure 4.6 gives closer insights on the object detection benchmark, which also reflects on GLIP through the usage of `exists()` in the correct solution. For most of the objects, the accuracy is around 50% and, therefore, just as good as chance. Only for the instrument table can the model provide a robust prediction. The recall for the low-performing objects is mostly close to 0 or 1, meaning that the model is either very biased toward predicting matches or no matches, respectively. This might indicate that the model has some wrong understanding of the objects upon which it confidently acts.

In total, the tools utilized do not perform well in the OR domain. Even on the task of object detection, posed as a binary classification task, GLIP can not outperform a coin toss for most object classes. This makes testing GLIP on a comprehensive object detection benchmark measuring intersection over union (IoU) unnecessary. However, many of the detection objects like *c-arm*, *mps* or *anesthesia equipment* are arguably very difficult to know for a model trained on data crawled from the internet, due to being especially domain-specific. GLIP displays somewhat better generalization capabilities in identifying people in the OR who wear heavy protective medical clothing, as in the case of the two surgeons. The lack of object understanding is also a problem in the case of BLIP-2, where an action between objects has to be identified. BLIP-2 was tested in preliminary experiments on the object detection benchmark but performed similarly to GLIP. Experiments to provide an image with color-marked objects and provide the name of the marked object in the query to BLIP-2 did not elicit any improvement. A more elaborate approach to injecting domain knowledge into BLIP-2 is needed.

While the bad performance might be expected due to the tools not being trained on OR data, the approach of using VLMs aimed to counter this, but the effect of good generalization did not materialize.

4.3 Data Modality Extensions

4.3.1 Audio Data

The integration of audio data and three approaches to do tool usage prediction with it were described in Section 3.4.1. The third approach utilizes threshold values to predict whether a tool sound is present or

Figure 4.7 Audio samples are encoded, and the similarity to audio class anchors is computed (left). A higher value means higher similarity. The similarity scores are normalized among all three audio class anchor similarities (right). The normalization shows a clustering, from which empirical threshold values are inferred to be used for binary sample classification during inference.

not. The selection of these thresholds, as well as the results of all approaches on the audio identification benchmark, are presented in this section.

A small set of annotated sound samples, functioning as a validation set introduced in Section 3.3.4, is analyzed to obtain the threshold values for cosine similarity. The idea is to find a threshold per sound class above which samples that represent the sound lie. For each query sound, all samples are processed to obtain the cosine similarity to the class anchor of the query sound. The resulting cosine similarity values are shown in Figure 4.7. As no clustering can be observed, this approach would not be effectively usable.

A slight adaptation is tested, where the similarity values are normalized. The cosine similarity is computed between the embedded input sound with all three anchor class embeddings, producing three similarity values. The value for the query class is simply divided by the average of all three values. This can be considered normalization as the value is scaled relative to the current average. The normalized similarities are shown in Figure 4.7 on the right, and in this case, clusters can be identified. As there is still some overlap between the samples that correspond or do not correspond to the class, an optimal threshold by visual inspection cannot be determined.

To extract optimal threshold values per audio class, a simple binary classifier per class is tested with different thresholds. If the similarity is above or equal to the threshold, the sample is classified as positive and negative otherwise. This is done for thresholds in the range from 0.9 to 1.1 with a step size of 0.001. For each threshold, the true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN) are calculated. The F1-score is an established metric to evaluate the performance of classifiers. It is defined as

$$F1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{4.1}$$

where Precision and Recall are given by Equations (4.2) and (4.3), respectively:

$$\text{Precision} = \frac{TP}{TP + FP} \tag{4.2}$$

$$\text{Recall} = \frac{TP}{TP + FN} \tag{4.3}$$

The threshold producing the best F1-score is chosen as the optimal threshold for that class.

All three approaches from Section 3.4.1 are now benchmarked, and results are shown in Table 4.3. It shows that Approach 3 achieves the best accuracy and is, therefore, finalized as the implementation in the tool interface `get_operation_phase()`. It achieves a solid 80% accuracy, but the approach is further extended by using the audio modality in tandem with the image modality. The implemented solution can be found in the full API specification in A.1, and it uses `get_operation_phase()` as a first check. If the tool interface detects the sound, `simple_query()` is used to check if any of the surgeons are performing the tooling action on the patient. Only if this confirms the action is the query answered positively; otherwise, it is answered negatively. As `get_operation_phase()` is used as a first check, it is beneficial for the tool interface to have a positive bias, i.e., the tendency to predict samples as positive, as the vision model can then be used to perform a verification to correct false predictions. A positive bias is manifested in a high recall, which is optimized instead of the F1-score from the previous. To optimize recall, true positives have to be maximized, and false negatives minimized, with an achievable optimal recall of 1 (see Equation (4.3)). A model that naively predicts every sample as positive would have an optimal recall but would, of course, not be optimal for practical purposes. Instead, recall is optimized by choosing the thresholds that produce optimal recall while still minimizing false positives. This corresponds to using the lowest similarity value of a sample that truly belongs to a class as the threshold for that class. By predicting samples below that threshold as negatives, the model minimizes false positives while still achieving an optimal recall of 1, like a naive model. Using recall-optimized thresholds in a combined approach of `get_operation_phase()` and `simple_query()` leads to an accuracy of 100%, shown in Table 4.3. The combined approach also leads to an improvement over the sole usage of `get_operation_phase()` when using the F1-score-optimized thresholds, shown by the accuracy increase from 80% to 90%.

4.3.2 Robot Screen Data

The integration of the robot screen data through the new tool interface `get_operation_phase()` was introduced in Section 3.4.2. The problem of phase detection was simplified to the problem of detecting the area highlighted in green among all areas. For this, the framework already presents the two tool interfaces `best_text_match()` and `verify_property()`. The first fits the problem better, as it is passed a list of texts and returns the one that matches the image best. It can be, therefore, passed a selection of colors that the areas display at different stages. The latter needs an object and a property to verify. This is more unintuitive to use as the area does not contain any objects other than text. However, "text" cannot be passed as the object as it does not change color, but the background does.

To this end, `best_text_match()` is used to process the six image crops of the phases. It is passed the options *["gray", "green", "blue"]*, as the area is gray if not active, and during the transition between phases, the screen can turn blue as part of a loading process. Preliminary experiments showed that

	Approach 1	**Approach 2**	**Approach 3**		
	Audio	Audio	Audio	Audio + Vision	Audio + Vision
Accuracy (%)	56.7	73.3	80	90*	**100**†

Table 4.3 Performance of different approaches on the audio benchmark. Audio: uses audio modality only. Audio + Vision: uses audio and image modality for final prediction. *: thresholds optimizing F1-score. †: thresholds optimizing Recall.

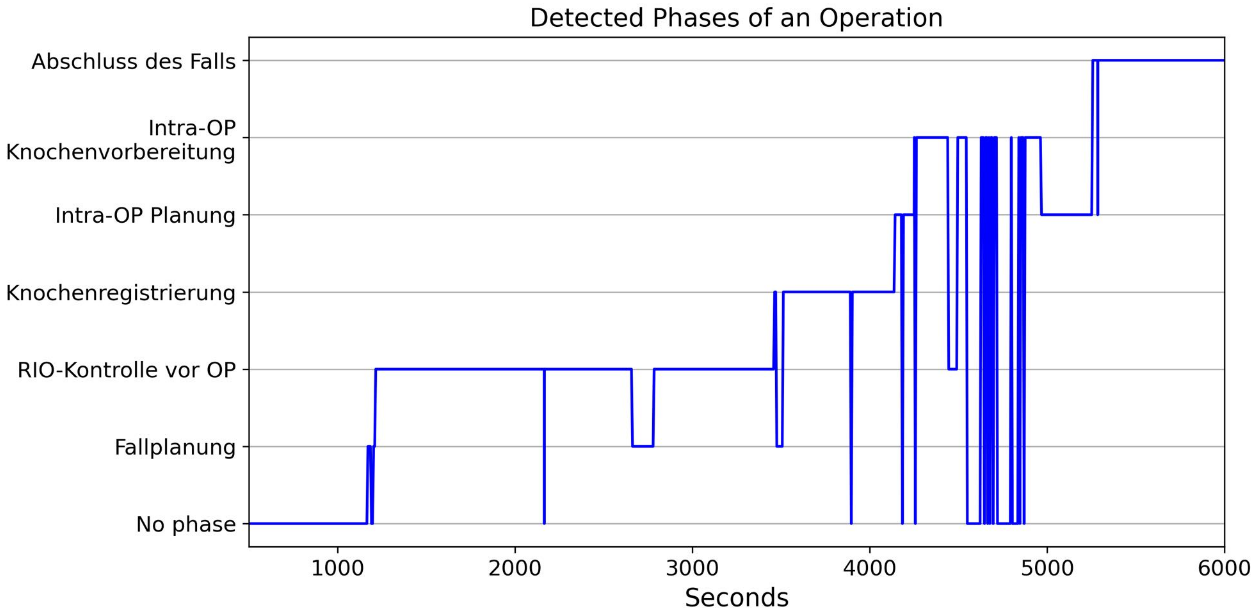


Figure 4.8 Phases of an operation extracted from the screen of the MAKO assistive robot. This information is made accessible to the framework by integrating the robot screen as a new data modality.

this resulted in some instances where multiple phases were falsely detected as green. To correct this, `verify_property()` is added as a second check in case of multiple matches. It is passed the "area" as object and "green" as property. It successfully catches all the false positives.

The algorithm was tested by running the phase detection on every frame of an operation. As there does not exist any ground truth information and labeling would have been too time-consuming, the testing was done through manual inspection. Not one instance of a false detection could be identified, leading to the assumption of a very robust algorithm. The *No phase* label represents frames where the robot screen was booting or transitioning between phases. The extracted phases over the duration of a whole operation are visualized in Figure 4.8. It shows a continual progression through the phases, with occasional going back to previous phases initiated by the operating surgeons.

4.4 Tool Substitution

Modularity as a benefit of the framework is demonstrated through the substitution of a tool. The performed modification is discussed step-by-step to provide an intuitive understanding of the switching process uncoupled from other tools. A core tool for scene understanding is BLIP-2, used through its tool interface `simple_query()`, which is often utilized when a problem cannot be broken down further. Since the publication of BLIP-2, new and improved models have come along, most notably the model LLaVA [27]. It combines a vision encoder and an LLM into one model and proposes the approach of visual instruction tuning, which helps the model interpret images based on natural language instructions in a prompt and respond to it accordingly. Instruction tuning is an effective practice in the language domain, which was seen already in Section 4.1. An iteration of the initial model, LLaVA-1.5, achieves SOTA results on many benchmarks, including GQA, and outperforms BLIP-2 by large margins [26]. This work integrates the model llava-v1.6-mistral-7b through huggingface. As the name suggests, it uses Mistral-7B-instruct as LLM.

The replacement is realized by simply processing the passed query in `simple_query()` with LLaVA instead of BLIP-2. This requires implementing a new model class, following the implementation structure for all other models, which uses LLaVA through the huggingface API and implements a `forward()` method that can be used in `simple_query()`.

	Accuracy (%)		BLEU-4	
	no GT	GT	no GT	GT
BLIP-2	16.4	18	0.082	0.079
LLaVA	55.9	57.4	0.132	0.135

Table 4.4 Framework performance for different vision models used as tools in `simple_query()`. GT represents the usage of the tool ground truth option for object detection.

Preliminary experiments showed that the prompt passed to LLaVA needs to be augmented. For example, to the prompt "What is the nurse doing with the patient?" the model argued that this is a trick question, as the patient is, in fact, a doll. This showed very advanced capabilities compared to BLIP-2 straight away. It was also noticed that the answers are very long. The prompt was, therefore, extended with the instructions to provide a concise answer. It was further stated that the image stems from surgery for training purposes and that the doll is to be referred to as the patient. This significantly improved the responses of the model.

The resulting framework with LLaVA integrated was tested on the relation detection benchmark, and results are shown in Table 4.4. The BLEU score, described in Section 3.2, is reported in addition to the accuracy computed with the LLM-as-a-judge approach. Both metrics show a significant performance increase when utilizing LLaVA over BLIP-2. Many failure cases of LLaVA arise when the model processes the query of the format "What is the relation between the object1 and the object2?". The model interprets these as questions about a functional relation and answers them by providing general information on the relation. For example, a query about the anesthetist and the patient is answered with the statement that the anesthetist is responsible for administering anesthesia to the patient. This is correct, given the interpretation of the query. However, the query asks about the spatial relation. This misunderstanding can be addressed in the LLaVA prompt in the future, likely leading to a further significant increase in performance on the relation detection benchmark.

The framework was further tested by activating the tool ground truth option for object detection, which is realized in `find()`. This tool interface is used in the reference solution for relation queries, where it is used to create a crop including the two relevant objects, such that `simple_query()` can process a focused image. This should help with the fact that the vision model behind `simple_query()` might lack domain knowledge, especially with respect to medical personnel. The focused view can remove some personnel unrelated to the query. Results in Table 4.4 show only a small improvement, suggesting that the solution approach producing the cropped view might not be necessary.

4.5 Chain-of-Thought Prompting

The LLM is a core part of the framework, and intuition on its behavior was gained in Section 4.2.2. To achieve this, its performance was tested by either providing benchmark-specific examples or using the model zero-shot. The zero-shot setup for the benchmark of object detection showed that the LLM still has some potential to improve its devised solutions. This weakness was addressed by providing it with benchmark-specific examples, resulting in consistently strong performance on all benchmarks. This also allows the presentation of specific, handcrafted solution approaches to the LLM, which it applies to queries of the same type. However, this is a setup that does not allow good generalization.

The Chain-of-Thought (CoT) approach offers the potential to increase the LLM's reasoning abilities and, thereby, its problem-solving skills in a more generalizable manner [58]. As written in Section 3.5, it also serves as a good format to integrate text-form domain knowledge into the code solution. The knowledge is simply passed to the LLM as part of the API specification.

As the automated evaluation of generated code with respect to quality is not possible, manual evaluation is performed. While code was evaluated automatically during the benchmarks on GQA in Section 4.1, it only evaluated syntax errors leading to compilation errors. In this case, code quality is sought to be evaluated in a more nuanced manner on the following criteria:

- **code correctness** - is the solution approach logically sound, and would it likely get the correct result given perfect vision model performance
- **knowledge integration** - does the solution integrate relevant information, which was given to the LLM in the prompt

A setup is devised that aims to test the LLM "in the wild". To this end, a new set of 13 distinct queries is devised, where most of the queries are novel compared to the curated benchmarks. The majority of queries aim at the integration of a piece of knowledge to achieve an optimal solution but can also be solved successfully, according to the *code correctness* criterion, without the knowledge integration. For example, the query *Is there sawing?* can be correctly answered by simply using the tool interface `analyse_audio()`. However, integrating the information that sawing only occurs in the operation phase *Intra-OP Knochenvorbereitung* and using the operation phase extracted through `get_operation_phase()` as a condition to perform sound detection is intended as the optimal solution. The class containing the tool interfaces is unchanged in the API specification. After the class body, the knowledge is inserted with the note "Important information to consider". Some information is factually incorrect but serves to test if the LLM incorporates it. The API specification then provides three examples, which are fewer compared to the base setup, in order to allow for more queries in the test set. The resulting API specification for the CoT experiments can be found in Appendix A.2.

The setup is tested in two variants: using CoT, such that in-context examples have the format (query, reasoning, code), or standard in-context examples of the format (query, code). It is also tested with the framework's current model Codellama-13B-instruct and with Llama-3-8B-instruct. As the CoT approach requires generating a logical rationale in addition to code, a general capabilities model is considered along the specialized coding model.

The generated code with and without CoT is shown for the query "Is the hammer being used?" for Codellama in 4.9 and for Llama-3 in 4.10. The information from the knowledge relevant to this query is "The audio model behind analyse_audio() produces high false positives and high recall, i.e. has a positive bias." and "Operating tools are only used by the head surgeon and assistant surgeon.". All answers on the full test set are in Appendix A.4.

Evaluating the code shows that Codellama produces a simple solution that is considered correct, where `verify_property()` is utilized. In the CoT setup, knowledge about who is using the hammer is used. The approach to use `simple_query()` to verify the usage of tools is also shown in one in-context example. The solution is considered correct, and knowledge is integrated successfully.

Analyzing the code produced by Llama-3 shows that the code of the standard setup is incorrect, as it evaluates if a hammer is found in the image instead of being used. There is also no knowledge being integrated. The CoT setup, however, produces a correct solution that, in contrast to Codellama in the CoT setup, uses all the relevant information from the knowledge as well. First, the use of the audio model is correctly identified, and the information about the bias of the model is correctly integrated. The following code uses the information that Codellama also used, leading to the correct solution.

The generated reasoning is sound for both models. However, Codellama phrases its reasoning very close to the examples, almost building the reasoning from snippets of example reasoning. Llama-3, on the

	Codellama		**Llama-3**	
	non-CoT	CoT	non-CoT	CoT
Code correctness	7	7	5	**9**
Knowledge integration	3	**7**	5	**8**

Table 4.5 Evaluating the Chain-of-Thought (CoT) approach on the criteria code correctness and knowledge integration. The number represents the sample count fulfilling the criteria out of a set of 13 samples.

```python
# Is the hammer being used?
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    hammer_patches = image_patch.find("hammer")
    if len(hammer_patches) == 0:
        return image_patch.simple_query("Is the hammer being used?")
    return bool_to_yesno(hammer_patches[0].verify_property("hammer", "being used"))
```

Listing 1 Non-CoT - code correctness: yes, knowledge integration: no

```python
# Query:
# Is the hammer being used?
# Thoughts:
# The object detection model has to be queried with the relevant object "hammer". If not found, ask the general purpose vision model directly.
# If it is found, it has to be checked if the head surgeon or assistant surgeon is using it. If this check is also positive, the action can be confirmed.
# Code:
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    hammer_patches = image_patch.find("hammer")
    if len(hammer_patches) == 0:
        return image_patch.simple_query("Is the hammer being used?")
    hammer_patch = hammer_patches[0]
    # either head surgeon or assistant surgeon can be holding the corresponding tool
    head_surgeon_hammering = image_patch.simple_query("Is the head surgeon hammering the patient?")
    assistant_surgeon_hammering = image_patch.simple_query("Is the assistant surgeon hammering the patient?")
    if head_surgeon_hammering == "yes" or assistant_surgeon_hammering == "yes":
        return "yes"
    return "no"
```

Listing 2 CoT - code correctness: yes, knowledge integration: yes

Figure 4.9 Code generated by Codellama-13B-instruct to the query "Is the hammer being used?" in two setups. CoT: using Chain-of-Thought examples in API specification. Non-CoT: using standard examples in API specification.

other hand, produces a very original and naturally sounding reasoning, rephrasing the information from the knowledge instead of matching it to the specific phrasing of the examples. The improvement of Llama-3 in the CoT setup suggests the effectiveness of its coherent reasoning for improving code correctness, even surpassing Codellama3, which is a coding expert model.

This example provides some intuition on using the CoT approach and visualizes its effects on the code. For both models, leveraging CoT leads to a solution that integrates relevant information from the knowledge provided in the API specification over the standard approach. In the case of Llama-3, using CoT even leads to a correct solution in contrast to the standard setup. While this observation can not be made for all of the 13 queries, it is the general observation among the samples. All samples are evaluated on code correctness and knowledge integration. The results in Table 4.5 indicate that utilizing CoT can improve the correctness of solutions and enhance the integration of provided knowledge in the solution compared to using standard in-context examples, even for models of relatively small size. This comes, however, at the cost of requiring the generation of additional tokens, leading to a slower framework.

```python
# Is the hammer being used?
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    hammer_patches = image_patch.find("hammer")
    if len(hammer_patches) == 0:
        return image_patch.simple_query("Is the hammer being used?")
    return bool_to_yesno(len(hammer_patches) > 0)
```

Listing 3 Non-CoT - code correctness: no, knowledge integration: no

```python
# Query:
# Is the hammer being used?
# Thoughts:
# The audio model can be used to detect the sound of the hammer. However, due to
↪  the high false positives, a second check is necessary.
# The vision model can be used to check if the head surgeon or assistant surgeon
↪  is holding the hammer.
# Code:
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    sound_detected = image_patch.analyse_audio("hammering")
    if sound_detected:
        # either head surgeon or assistant surgeon can be holding the hammer
        head_surgeon_hammering = image_patch.simple_query("Is the head surgeon
        ↪  holding the hammer?")
        assistant_surgeon_hammering = image_patch.simple_query("Is the assistant
        ↪  surgeon holding the hammer?")
        if head_surgeon_hammering == "yes" or assistant_surgeon_hammering ==
        ↪  "yes":
            return "yes"
    return "no"
```

Listing 4 CoT - code correctness: yes, knowledge integration: yes

Figure 4.10 Code generated by Llama-3-8B-instruct to the query "Is the hammer being used?" in two setups. CoT: using Chain-of-Thought examples in API specification. Non-CoT: using standard examples in API specification.

While this evaluation is performed on a small test set, this limitation is grounded in the framework's architecture. The approach of creating a small test set and manually inspecting the code is the only option due to the code only being an intermediate solution step. Executing the code leads to the final output using the vision models. The evaluation of this output is possible on OR benchmarks that were curated. However, when only evaluating the final output, the attribution of performance to the LLM or the tools is impossible, as the tool ground truth option has to be devised anticipating certain behavior. That is impossible for a query set with high variety, where the solution to a query can take many forms, as shown in 4.9 and 4.10. Moreover, the integration of knowledge could not be verified, assuming a perfect ground truth option for all tools.

Despite the limited test set size, the evaluation displays promising tendencies and can serve as an impulse for a more rigorous evaluation of the approach on more samples. Ideas for use-cases of integrating CoT into the framework and potential variations of it are discussed in Section 5.4.

5 Summary and Discussion

5.1 Key Findings

To explore the capabilities of an LLM-based framework for scene understanding in the OR, this work adapted the approach of ViperGPT [52]. It uses an LLM that processes queries about an image and produces Python code. The code contains calls to tools, which are vision models with different expertise, processing the image. The Python Interpreter executes the code, and the final output of the execution is the answer to the query initially processed by the LLM. The problem of scene understanding in the OR was initially represented by three vision-based benchmarks.

Evaluating the framework on the published benchmark GQA showed that open-source LLMs specialized for code generation can compete with the proprietary GPT-3 Codex. The experiments also displayed the superiority of code-generation LLMs over general-purpose LLMs.

Evaluating the framework on devised benchmarks using OR data was done by decoupling LLM and vision model performance as far as possible. Experiments showed that the LLM can produce correct code robustly when examples from the benchmark are provided. This does not require the LLM to generalize well and devise good solutions on its own. Instead, it can be understood as a classifier, where the LLM, given a query, needs to classify the relevant example and adapt the code minimally. The addition of examples is only limited by the LLM's context length, where different approaches are devised to increase it [4][35]. By utilizing them, newer models are constantly increasing their context lengths [33][49]. Providing benchmark-specific examples also allows the injection of human knowledge, e.g., by providing the most robust solution to a problem that the LLM might not develop.

Evaluating the performance of VLMs as framework tools showed insufficient transfer abilities, resulting in inadequate performance on the OR benchmarks. The tool ground truth option allowed the independent evaluation of the LLM and tools, where the tool ground truth option activated showcased an accuracy close to 100% on all benchmarks. This highlights the tools for visual processing as the current bottlenecks of the framework. The result suggests that despite their strong generalization capabilities, VLMs do not display the necessary knowledge for the highly specialized OR domain, which involves specific medical personnel and equipment. As they are trained on data from the internet, the OR domain seems to pose a strong domain shift. This motivates the necessity for specialized vision models.

The evaluation of free-form textual answers showed the necessity for an evaluation metric that takes semantic similarity with the ground truth into account. This is not the case for the established BLEU score, so it was only used with a supporting function. A metric following the LLM-as-a-judge principle was implemented and used successfully to evaluate the framework's performance on the relation detection benchmark. Manual inspection of the metrics gradings showed potential for further improvement, making this metric more robust.

The framework was then extended to new modalities available in the in-house dataset. Audio data was integrated by utilizing a new tool, CLAP, and exploring different approaches to utilizing it. The domain shift aspect was minimized through the concept of audio anchors. It disregards CLAP's text encoder and only utilizes its audio encoder. A number of audio samples from the annotated dataset are used as sound templates, encoded with CLAPs audio encoder, and compared to the encoding of query audio. The evaluation on a small, manually annotated test set showed robust tool sound detection capabilities infused into the framework. The newly implemented tool interface was combined with an existing tool interface to devise a coded solution, utilizing audio and vision data, that achieved perfect results on the task of tool sound detection.

As another new data modality, screen data of an assistive robot provided by the in-house dataset was integrated with the goal of detecting the operation phase. The developed approach extracts very specific information from the screen that contains the task-relevant information. This was achieved by utilizing existing tools in the newly implemented tool interface. Due to the lack of ground truth data, the approach was manually verified. It showed the successful extension of the framework to the robot screen data modality, unlocking the capability of operation phase detection.

Integrating the new data modalities extends the core structure of the framework in a new direction, with the new components following a different paradigm than the original ones. Initially, the framework employed basic tool interfaces such as `find()` for object detection and `simple_query()` for general image understanding, which is broadly applicable to most scene understanding tasks. However, the audio modality was integrated through the `analyse_audio()` tool interface, which provides a binary prediction for the specific tool usage sounds *drilling*, *sawing*, and *hammering*. Therefore, it requires specific queries and lacks general audio understanding capabilities. Similarly, the robot screen was integrated using the `get_operation_phase()` tool interface. It retrieves specific information from the screen instead of providing the framework access to all of the available information there. The new paradigm is, therefore, the extension of the framework with specific new capabilities in mind. This adaptation reflects the framework's evolution from a general-purpose tool to being transferred to the OR domain, requiring specialized components to extract specific information.

Finally, Chain-of-Thought (CoT) was explored as an approach to enable reasoning in the code generation LLM proposed by Wei et al. [58]. It builds on the concept of in-context prompting, where examples passed to the LLM demonstrate the intended behavior. CoT requires generating a sequence of reasonings about the problem, which breaks down the problem into smaller problems and leads to the final answer, which is a code solution in this framework. Using in-context prompting as a core component of this framework allowed the straightforward adaption of CoT for experiments. In addition to unlocking the code-generation LLMs reasoning abilities for better code solutions, CoT was pursued as a format to integrate external knowledge into the LLMs generation process. Code quality and the integration of provided knowledge were manually inspected on a small set of queries, testing the LLM on new query types not contained in the benchmarks used so far. The experiments showed promising results in terms of improving the rate of knowledge integration. In addition, utilizing CoT had positive effects on code quality. While the general-purpose Llama-3 model showed worse performance compared to the specialized coding model Code Llama when not using CoT, it was able to produce code of better quality than Code Llama in the CoT setting. However, this comes at the cost of requiring the LLM to generate additional tokens for reasoning, increasing inference time.

5.2 Discussion

The benefits of the initial framework were presented in Section 3.1.2. Experimenting with and extending the framework allows for evaluating these benefits from a more informed perspective.

A limitation of the initial framework was identified in the LLM-based code generation, as there is a small possibility of the code being malicious. This could lead to file alterations on the machine that executes the code due to Python's read, write, and execute rights. As the project is run on a high-performance cluster used by the chair, malicious code execution could have far-reaching detrimental consequences. To mitigate this risk, a safety mechanism is implemented using sandboxing. This is realized by using the built-in Python function `exec()`, which executes dynamically produced code. It allows executing the code in a different global environment than `exec()` is executed in by providing a dictionary containing the accessible functions. In this dictionary, only a limited number of functions required for the successful execution of generated code, such as the built-in `min()` or `len()`, are included. It does not include, however, packages used for manipulating the machine's file system, for example, `shutil` and `os`. By restricting the available Python functions, even if the code is malicious, it will not execute.

The framework's modular approach of using different tools in the form of vision- and audio models independently from each other is a key benefit of the framework. It allows for the straightforward integration of new tools and new tool interfaces that provide access to them. This was demonstrated by the integration of the CLAP model through `analyse_audio()` and the robot screen through the `get_operation_phase()`. This modularity enables multi-modality, as the extensions are not restricted with respect to the data domain. In addition, the modularity also provides the option to easily replace tools with a more updated model version or a new model altogether, which was demonstrated by substituting the model BLIP-2 [24] with the superior LLaVA [27].

The second main benefit is utilizing the approach in a train-free manner. This, in addition to the modular architecture, enabled the time-efficient extension and replacement of modules. It further allowed allocating the constrained computing resources solely to inference experiments. A framework fine-tuned for the integrated functioning of its tools would not need to be trained once initially but upon every change in one of the tools.

Finally, the architecture of solving a problem by executing an intermediate code solution to the problem provides great interpretability. The code solution was inspected constantly during development to gain an explanation of the final answer to a query. Both the code as a whole, reflecting on the LLM, and tool interface calls in the code, reflecting on the tools, were analyzed. Additionally, the tool ground truth options were implemented, allowing for the isolated evaluation of the LLM performance. This produced the insight that the LLM can generate correct code more robustly when provided with an example from the benchmark, as it can apply the same solution to queries of the same type, as shown in Table 4.2. Attributing this specifically to the LLM without an intermediate code solution would not have been as straightforward. This insight was integrated into the final framework through the API specification, improving overall performance and allowing a more independent evaluation of tool performance.

However, the approach of solving a problem through an intermediate code solution also poses a disadvantage due to increased computing requirements. While a general-purpose vision model like BLIP-2 directly outputs the answer to a query, this approach requires the generation of code, which, upon execution, calls the vision model. This computation overhead materializes in requiring more computing resources for running an LLM in addition to the vision model and increased inference times. While this approach managed this overhead by adhering to the VRAM limitations of one Nvidia A40 GPU, access to this amount of computing resources might not always be given. Apart from the resource requirements, the inference times with this approach were around 13s, which is objectively a long time to obtain an answer to a question from a user's perspective.

Another disadvantage is that code is generated statically for the problem instead of dynamically integrating intermediate code execution results. The LLM generates a code solution based only on the description of tool interfaces but not on the intermediate output of those interfaces, requiring the anticipation of their behavior. This is especially problematic for tool interfaces returning a string of no constrained format instead of a boolean or a list, where the length of the list contains information usable in subsequent steps. An example of the latter is the solution to the problem of people counting, where the tool interface `find()` returns a list containing the detected objects. By applying the `len()` function on the list, subsequent code can unambiguously utilize the intermediate result provided by `find()`. However, utilizing a string is ambiguous and not that robust. The code provided in 4.9 shows the usage of `simple_query()` as an intermediate step, where the output string is assigned to the variable `head_surgeon_hammering` and compared in an if-statement to the string *yes* to arrive at the final solution. If the tool interface returns a slight variation of this, such as *yes, that is true*, the if-statement does not produce the intended result. This results in the difficulty of using `simple_query()` for intermediate steps, and it is instead rather used to produce the final answer.

Additionally, code has increased length due to being generated statically, as all potential cases have to be anticipated and implemented. An example is the code in 4.9, where two code lines are implemented for the case that `find()` did not find the specified object. This represents using the general-purpose model through `simple_query()` as a backup solution, which has been adopted as best practice for many solution approaches. However, generating the code dynamically could check the condition and, based on its result, generate the backup solution or the intended implementation, as in lines 10-16. This would result in only generating the needed code, speeding up the overall inference.

Furthermore, utilizing an LLM training-free limits the ability to give feedback to it. This results in the LLM generating the same faulty solution approach every time when provided with the query type triggering that code solution instead of learning from it. Thereby, integrating feedback is limited to providing in-context examples, that provide a solution to the LLM for specific failure cases.

Finally, there is no ground truth data for the intermediate code solutions to verify the generated code. Additionally, due to the numerous possible code solutions for a query, verifying the code automatically against a ground truth solution would not be feasible. Consequently, the code is usable for selective inspection during debugging but not for generally attributing errors to the LLM or the vision models in case of an incorrect final answer. The implementation of the ground truth option for selected vision models was devised to address this. This workaround allows for the isolated evaluation of LLM performance. However, it requires anticipating specific tool interface usages to provide the correct ground truth in the case of `simple_query()`. Implementing a ground truth option for arbitrary usage of all tool interfaces is not feasible, and extending it to a large set of anticipated usages is time-consuming.

5.3 Implications

This work constitutes the first implementation of an approach providing scene understanding in the multimodal operating room. To our knowledge, this is currently the first framework targeted at this specialized domain that allows the processing of posed questions. The work flattens the path toward holistic scene understanding by enabling the integration of arbitrary data modalities, which is a core ability of the framework. This gives it a strong justification for its existence. In contrast, ViperGPT operates exclusively in the image domain. If optimizing exclusively performance, its justification relies on outperforming the general-purpose vision model it uses, specifically BLIP-2. Even then, this comes at the cost of increased inference time, which has to be weighed against the performance gain. While ViperGPT outperforms BLIP-2 by some margin [52], this can change for a more capable general-purpose vision model like LLaVA.

The framework demonstrated its capability to match posed queries robustly to provided example queries, adapting the reference code solution to the specific query. This delegates the responsibility of solving the query correctly to the tools used. Here, the framework showed insufficient capabilities, as the tools, specifically BLIP-2 and GLIP [25], could not robustly generalize to the OR domain. The integration of LLaVA instead of BLIP-2 showed the potential to improve the framework's performance by substituting individual models. This direction could be pursued further by integrating tools specialized in the OR domain. A currently developed in-house segmentation model could be used as a first integration and serve as a first step in exploring its benefits. The replacement of tools is considered a promising source for optimizing the framework. As another source, the approach of CoT showed great potential. Ideas for future work are discussed in Section 5.4.

While the framework currently lacks the capabilities to be used reliably in the operating room, it has strong potential for improvement from different sources, i.e., the integration of more specialized tools or utilizing CoT for the LLM. Improving robustness and scene understanding capabilities would lead to a framework that can be beneficially deployed in the OR. For instance, it could be used for communication purposes, where the framework can serve as an information access point, providing up-to-date information upon external request on the current progress of an ongoing operation. It could also be used for quality assurance, where an operation is processed after recording with the goal of assessing compliance with sterility standards or the correct use of tools.

5.4 Future Work

This section provides various ideas for future work, including the extension of benchmarks and further exploration of the framework's different components. The modular architecture of an LLM instructing multiple models of different modalities provides many opportunities for future work.

As discussed in Section 1.2, scene understanding is a multi-faceted task. Evaluating the framework on multiple benchmarks that test selective performance aspects allows the isolated inspection of tools with respect to different tasks that are part of scene understanding. For example, the effects of changing the audio model can be observed best through the benchmark solely for audio detection. Further benchmarks testing the specific abilities of object detection and relation detection can be devised. The object detection problem was posed as a binary classification task, which works as a first evaluation. However, the task of object detection is typically formulated as predicting a bounding box containing the query object. This poses a harder problem, which reflects on a model's capabilities more precisely. For example, the model for object detection, GLIP, might associate wrong objects with the query objects, which does not specifically show up in the used benchmark other than in low accuracy. However, obtaining the bounding box predictions provides a richer insight into the model's capabilities and can be used to improve the model utilization in the scope of the framework.

The understanding of object relations, on the other hand, was posed as an open-set recognition problem, where the model is required to freely predict the occurring relation. This can lead to the model predicting a correct relation, which is, however, not the ground truth relation. The LLM-as-a-judge accuracy metric as a measure of the model's ability to predict relations that are sufficiently similar to the ground truth aimed at addressing this issue. As shown in Section 4.2.1, this might require choosing evaluation guidelines, complicating the metric and reducing transparency. The problem could, alternatively, be posed as a classification problem, where the model is provided with the set of relations from the annotated scene graphs. This makes evaluation straightforward, and a model predicting more than those relations might not even be required, depending on the use case.

In addition to selective benchmarks, evaluating the capabilities of the whole framework with a combined benchmark is also needed. To this end, a new benchmark could be devised that integrates the individual benchmarks and contains new query types, testing the generalization capabilities of the framework. However, the distribution of query types, representing respective scene understanding abilities, needs to be balanced, which raises the issue of scaling the benchmark up to a high sample count, as the currently available ground truth data from annotations in the in-house dataset was already used in the created benchmarks. This suggests the requirement for more annotated data, enabling a representative evaluation of the framework's intended performance.

A more general evaluation of the framework was pursued with the query set used for testing chain-of-thought performance. This included queries requiring the evaluation of sterility or making operation phase-dependent associations. More ideas can be included in such a query set to test the framework's abilities in a broader and more general manner. However, this again requires annotating queries by hand, as ground truth data is currently not available. Additionally, these queries often pose a compositional problem that is more complex than simple object detection, requiring an advanced framework that can, for example, integrate external knowledge, as pursued in the CoT approach.

A data source of the in-house dataset that has not been used yet is 3D point cloud data, which is provided by the depth sensors of all five RGB-D cameras. Point cloud data provides precise spatial information while lacking semantic information [28]. To this end, point cloud data is often utilized in a fusion strategy with 2D image data, resulting in semantically augmented point cloud data [40, 28]. As such, the integration into the framework is likely not as straightforward as the audio data integration, which can be directly processed by a dedicated model. Instead, the integration could be performed through the addition of a model that uses a fusion strategy, effectively leveraging both the 2D image modality and the 3D point cloud modality. The model implemented by Özsoy et al. [40] is a potential candidate, as it is a specialized model on the OR domain. The model predicts the scene graph for a given timestep, which could serve as input for another model, possibly another LLM, that utilizes it to answer the natural language query.

Furthermore, the framework currently only uses single-view image data per query. An interesting extension would be to allow the LLM to select the most fitting camera view or even multiple given a query, i.e., dynamic view selection. This would need to be coupled with providing the LLM information about what each view displays and what it typically displays without occlusions. For example, the LLM might choose the image from camera number five when given a query about the instrument table, as this view shows the instrument table in the foreground, such that usually nobody from the medical staff is occluding it. A convenient cosmetic adaption in this context would also be to provide query-timestamp pairs in the benchmarks instead of query-image pairs, while the images also contain the timestamp.

A source for new experiments and adaptations is the framework's LLM used for code generation. Codellama-13B-instruct, as the chosen model, provides a tradeoff between speed and performance. It could be further investigated both how smaller models perform, improving the framework's inference time, and how larger models perform, improving the framework's ability to solve compositional queries requiring advanced reasoning capabilities. As the LLM selection was performed on the GQA benchmark without benchmark-specific examples in the API specification, a smaller model than Codellama-13B-instruct might be sufficient for a use case based on the available in-house dataset, where the model only needs to handle a few different query types. If one or multiple examples exist for every query type the model needs to handle, this would somewhat rephrase the LLM task as a classification problem, where it needs to identify the relevant example given a new query and apply the solution approach, requiring only small adaptations. This behavior was successfully displayed by Codellama-13-instruct separately for all benchmarks, shown in Section 4.2.2 by the high accuracy for the setup of activated tool ground truth options in Table 4.2.

Section 4.5 demonstrated CoT as a promising approach to apply to the LLM, resulting in improved problem-solving abilities. Especially for the case of the tested Llama-3-8B-instruct, the produced reasoning was coherent and logical, with the only challenges arising for temporal queries like *Has there been sawing so far?*. The work of Wei et al. showed CoT-enabled reasoning to be emerging increasingly with model scale [58]. To this end, exploring LLMs with more parameters than the tested 8B and 13B models poses an interesting exploration in the context of CoT as well.

CoT also demonstrated promising results for the integration of external knowledge into a code solution generated by the LLM. External knowledge is required when extending the framework to dynamic view selection, as it provides the LLM with essential information about the camera views. It might also be required to pass information to the LLM needed for devising coded solutions. While this can also be demonstrated specifically for each problem through an in-context example, the integration through general knowledge allows the LLM to apply it to multiple problem solutions, allowing more diversity in the generated solutions.

A further extension in the context of CoT is the sequential use of two LLMs, one for generating thoughts and the other for generating code based on the provided thoughts. As experiments in Section 4.1 showed, coding expert LLMs produced fewer exceptions than all-purpose LLMs, outperforming them overall on the benchmark GQA. However, the experiments in Section 4.5 showed that Codellama-13B-instruct, being a coding expert model, displays inferior reasoning when generating thoughts in the context of CoT compared to Llama-3-8B, largely copying thoughts provided in the in-context examples of the API specification. The sequential use of two LLMs would combine the reasoning abilities of a general-purpose model that generates thoughts with the coding abilities of a coding expert model that generates solution code based on the thoughts.

The framework's tools are the second architectural component offering further experiments and extensions. As results have shown in Section 4.2.2, using the LLM with the tool ground truth option results in performance close to 100%. However, processing visual OR data with the actual tools does not perform satisfactorily, showcasing that the tools for visual inference, namely GLIP and BLIP-2, are currently the main bottleneck in the framework. While the use of LLaVA over BLIP-2 strongly improves performance on the relation detection benchmark, an accuracy of around 60% is still not robust enough. Including a new tool utilizing fused point cloud and image information was already mentioned earlier. The exchange of existing tools should, however, mainly focus on tools that have the potential to perform well in the OR domain by being specialized. This might include the segmentation model that is currently being trained on the in-house OR dataset, which can be used in an object detection capacity for objects for which a segmentation label exists. Approaches to fine-tuning existing pre-trained models might offer another alternative. Furthermore, LLaVA showed the potential to integrate domain knowledge through its prompt, as the internal LLM can process it.

The static code generation of the LLM was discussed as a weakness of the approach underlying the framework in Section 5.2. A solution to this problem would be to generate the code step-by-step. The LLM produces code in chunks, where intermediate output from tool interface calls is shown to the LLM before generating the next chunk. This would be similar to the approach of ReAct by Yao et al., which combines reasoning abilities with action plan generation in an LLM [60]. To this end, the LLM generates triples of (thought, action, observation) iteratively. A thought is a text that arrives at the decision to perform a subsequent action. This is carried out through an API call, such as a database search of a given keyword. The output of the API call is called an observation, and it is shown to the LLM. The next step is again the generation of a thought that incorporates this observation to plan the next action, repeating this pattern until a termination condition is reached. The approach utilizes in-context learning and no training. An issue in adopting the approach of ReAct into this framework is its limitation to the language domain. The output of tool interfaces in this framework is, however, not always in text form. The output of the tool interface `find()` is a list of varying length containing image crops. On the other hand, many tool interfaces do produce output represented as or convertible to text, such as `simple_query()`, `exists()` or `best_text_match()`. Future work might explore ways to integrate an approach similar to ReAct into this framework, which would utilize more of the LLM's capabilities through dynamic code generation, addressing a significant current limitation.

6 Conclusion

This work implements a training-free, multi-modal, extendable, and interpretable framework for scene understanding in the operating room. This is achieved by utilizing an LLM with the paradigm of tool usage. The LLM processes queries about the scene and generates code that, when executed, calls tools that process different data modalities and answers the query. The framework utilizes single-view image data, robot screen data, and audio data and is evaluated on multiple benchmarks using an in-house multi-modal dataset. By providing the LLM with in-context examples that solve the problem posed by the query type found in each benchmark, it can generate correct code robustly. However, the framework shows insufficient performance in the tasks of people counting, object detection, and relation detection due to the lack of domain-specific performance of the employed tools for visual processing, which are vision language models trained on general-domain data. This motivates the need to integrate domain knowledge into the existing tools or use specialized tools in future work. Further extensions include utilizing multi-view image data and 3D point cloud data from the in-house dataset and extending the benchmarks. The LLM choice can further be optimized for different use cases, and the approach of Chain-of-Thought evaluated more extensively [58].

Bibliography

[1] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. “Neural machine translation by jointly learning to align and translate”. In: *arXiv preprint arXiv:1409.0473* (2014).

[2] Tom Brown et al. “Language models are few-shot learners”. In: *Advances in neural information processing systems* 33 (2020), pp. 1877–1901.

[3] Chris Callison-Burch, Miles Osborne, and Philipp Koehn. “Re-evaluating the role of BLEU in machine translation research”. In: *11th conference of the european chapter of the association for computational linguistics*. 2006, pp. 249–256.

[4] Shouyuan Chen et al. “Extending context window of large language models via positional interpolation”. In: *arXiv preprint arXiv:2306.15595* (2023).

[5] Aakanksha Chowdhery et al. “Palm: Scaling language modeling with pathways”. In: *Journal of Machine Learning Research* 24.240 (2023), pp. 1–113.

[6] Karl Cobbe et al. “Training verifiers to solve math word problems”. In: *arXiv preprint arXiv:2110.14168* (2021).

[7] Jia Deng et al. “Imagenet: A large-scale hierarchical image database”. In: *2009 IEEE conference on computer vision and pattern recognition*. Ieee. 2009, pp. 248–255.

[8] Jacob Devlin et al. “Bert: Pre-training of deep bidirectional transformers for language understanding”. In: *arXiv preprint arXiv:1810.04805* (2018).

[9] Alexey Dosovitskiy et al. “An image is worth 16x16 words: Transformers for image recognition at scale”. In: *arXiv preprint arXiv:2010.11929* (2020).

[10] Benjamin Elizalde et al. “Clap learning audio concepts from natural language supervision”. In: *ICASSP 2023-2023 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2023, pp. 1–5.

[11] Zhe Gan et al. “Vision-language pre-training: Basics, recent advances, and future trends”. In: *Foundations and Trends® in Computer Graphics and Vision* 14.3–4 (2022), pp. 163–352.

[12] Carly R Garrow et al. “Machine learning for surgical phase recognition: a systematic review”. In: *Annals of surgery* 273.4 (2021), pp. 684–693.

[13] Kaiming He et al. “Mask r-cnn”. In: *Proceedings of the IEEE international conference on computer vision*. 2017, pp. 2961–2969.

[14] Drew A Hudson and Christopher D Manning. “Gqa: A new dataset for real-world visual reasoning and compositional question answering”. In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2019, pp. 6700–6709.

[15] Benoit Jacob et al. “Quantization and training of neural networks for efficient integer-arithmetic-only inference”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 2704–2713.

[16] Ziwei Ji et al. “Survey of hallucination in natural language generation”. In: *ACM Computing Surveys* 55.12 (2023), pp. 1–38.

[17] Justin Johnson et al. “Image retrieval using scene graphs”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 3668–3678.

[18] Justin Johnson et al. “Inferring and executing programs for visual reasoning”. In: *Proceedings of the IEEE international conference on computer vision*. 2017, pp. 2989–2998.

[19] B Kayani et al. “Robotic-arm assisted total knee arthroplasty is associated with improved early functional recovery and reduced time to hospital discharge compared with conventional jig-based total knee arthroplasty: a prospective cohort study”. In: *The bone & joint journal* 100.7 (2018), pp. 930–937.

[20] Lauren R Kennedy-Metz et al. “Computer vision in the operating room: Opportunities and caveats”. In: *IEEE transactions on medical robotics and bionics* 3.1 (2020), pp. 2–10.

[21] Alexander Kirillov et al. “Segment anything”. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2023, pp. 4015–4026.

[22] Florent Lalys and Pierre Jannin. “Surgical process modelling: a review”. In: *International journal of computer assisted radiology and surgery* 9 (2014), pp. 495–511.

[23] Langchain. *Aligning LLM-as-a-Judge with Human Preferences*. Accessed: 2024-07-14. 2024. URL: `https://blog.langchain.dev/aligning-llm-as-a-judge-with-human-preferences/#:~:text=Using%20an%20%E2%80%9C%20LLM-as-a-Judge%20%E2%80%9D%20is%20a%20popular,LLM%20and%20asking%20it%20to%20judge%20the%20output`.

[24] Junnan Li et al. “Blip-2: Bootstrapping language-image pre-training with frozen image encoders and large language models”. In: *International conference on machine learning*. PMLR. 2023, pp. 19730–19742.

[25] Liunian Harold Li et al. “Grounded language-image pre-training”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2022, pp. 10965–10975.

[26] Haotian Liu et al. “Improved baselines with visual instruction tuning”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2024, pp. 26296–26306.

[27] Haotian Liu et al. “Visual instruction tuning”. In: *Advances in neural information processing systems* 36 (2024).

[28] Zhijian Liu et al. “Bevfusion: Multi-task multi-sensor fusion with unified bird’s-eye view representation”. In: *2023 IEEE international conference on robotics and automation (ICRA)*. IEEE. 2023, pp. 2774–2781.

[29] Jonathan Long, Evan Shelhamer, and Trevor Darrell. “Fully convolutional networks for semantic segmentation”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 3431–3440.

[30] Pan Lu et al. “Chameleon: Plug-and-play compositional reasoning with large language models”. In: *Advances in Neural Information Processing Systems* 36 (2024).

[31] Lena Maier-Hein et al. “Surgical data science for next-generation interventions”. In: *Nature Biomedical Engineering* 1.9 (2017), pp. 691–696.

[32] Joshua Maynez et al. “On faithfulness and factuality in abstractive summarization”. In: *arXiv preprint arXiv:2005.00661* (2020).

[33] Meta. *Introducing Meta Llama 3: The most capable openly available LLM to date*. Accessed: 2024-07-20. 2024. URL: `https://ai.meta.com/blog/meta-llama-3/`.

[34] Matthias Minderer et al. “Simple open-vocabulary object detection”. In: *European Conference on Computer Vision*. Springer. 2022, pp. 728–755.

[35] Tsendsuren Munkhdalai, Manaal Faruqui, and Siddharth Gopal. “Leave no context behind: Efficient infinite context transformers with infini-attention”. In: *arXiv preprint arXiv:2404.07143* (2024).

[36] Muzammal Naseer, Salman Khan, and Fatih Porikli. “Indoor scene understanding in 2.5/3d for autonomous agents: A survey”. In: *IEEE access* 7 (2018), pp. 1859–1887.

[37] Thomas Neumuth. “Surgical process modeling”. In: *Innovative surgical sciences* 2.3 (2017), pp. 123–137.

[38] Chinedu Innocent Nwoye et al. "Weakly supervised convolutional LSTM approach for tool tracking in laparoscopic videos". In: *International journal of computer assisted radiology and surgery* 14 (2019), pp. 1059–1067.

[39] Maxime Oquab et al. "Dinov2: Learning robust visual features without supervision". In: *arXiv preprint arXiv:2304.07193* (2023).

[40] Ege Özsoy et al. "4d-or: Semantic scene graphs for or domain modeling". In: *International Conference on Medical Image Computing and Computer-Assisted Intervention*. Springer. 2022, pp. 475–485.

[41] Ege Özsoy et al. "Labrad-or: lightweight memory scene graphs for accurate bimodal reasoning in dynamic operating rooms". In: *International Conference on Medical Image Computing and Computer-Assisted Intervention*. Springer. 2023, pp. 302–311.

[42] Kishore Papineni et al. "Bleu: a method for automatic evaluation of machine translation". In: *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 2002, pp. 311–318.

[43] Arkil Patel, Satwik Bhattamishra, and Navin Goyal. "Are NLP models really able to solve simple math word problems?" In: *arXiv preprint arXiv:2103.07191* (2021).

[44] Matt Post. "A call for clarity in reporting BLEU scores". In: *arXiv preprint arXiv:1804.08771* (2018).

[45] NLTK Project. *Sample usage for bleu*. Accessed: 2024-08-14. 2023. URL: `https://www.nltk.org/howto/bleu.html` (visited on 08/09/2024).

[46] Yujia Qin et al. "Toolllm: Facilitating large language models to master 16000+ real-world apis". In: *arXiv preprint arXiv:2307.16789* (2023).

[47] Alec Radford et al. "Learning transferable visual models from natural language supervision". In: *International conference on machine learning*. PMLR. 2021, pp. 8748–8763.

[48] René Ranftl et al. "Towards robust monocular depth estimation: Mixing datasets for zero-shot cross-dataset transfer". In: *IEEE transactions on pattern analysis and machine intelligence* 44.3 (2020), pp. 1623–1637.

[49] Machel Reid et al. "Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context". In: *arXiv preprint arXiv:2403.05530* (2024).

[50] Baptiste Roziere et al. "Code llama: Open foundation models for code". In: *arXiv preprint arXiv:2308.12950* (2023).

[51] Timo Schick et al. "Toolformer: Language models can teach themselves to use tools". In: *Advances in Neural Information Processing Systems* 36 (2024).

[52] Dídac Surís, Sachit Menon, and Carl Vondrick. "Vipergpt: Visual inference via python execution for reasoning". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2023, pp. 11888–11898.

[53] Mistral AI Team. *Mixtral of experts*. Accessed: 2024-07-20. 2023. URL: `https://mistral.ai/news/mixtral-of-experts/`.

[54] OpenAI Team. *Hello GPT-4o*. Accessed: 2024-08-04. 2024. URL: `https://openai.com/index/hello-gpt-4o/`.

[55] Hugo Touvron et al. "Llama 2: Open foundation and fine-tuned chat models". In: *arXiv preprint arXiv:2307.09288* (2023).

[56] turboderp. *exllamav2*. `https://github.com/turboderp/exllamav2`. Accessed: 2024-07-19. 2023.

[57] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems* 30 (2017).

[58] Jason Wei et al. "Chain-of-thought prompting elicits reasoning in large language models". In: *Advances in neural information processing systems* 35 (2022), pp. 24824–24837.

[59] Zhengyuan Yang et al. “Mm-react: Prompting chatgpt for multimodal reasoning and action”. In: *arXiv preprint arXiv:2303.11381* (2023).

[60] Shunyu Yao et al. “React: Synergizing reasoning and acting in language models”. In: *arXiv preprint arXiv:2210.03629* (2022).

[61] Yan Zeng, Xinsong Zhang, and Hang Li. “Multi-grained vision language pre-training: Aligning texts with visual concepts”. In: *arXiv preprint arXiv:2111.08276* (2021).

[62] Lianmin Zheng et al. “Judging llm-as-a-judge with mt-bench and chatbot arena”. In: *Advances in Neural Information Processing Systems* 36 (2024).

A Appendix

A.1 LLM Prompt Template

```
from PIL import Image
from vision_functions import find_in_image, simple_qa, verify_property,
↪  best_text_match, get_operation_phase, analyse_audio

def bool_to_yesno(bool_answer: bool)->str:
    return "yes" if bool_answer else "no"

class ImagePatch:
    """A Python class containing a crop of an image centered around a particular
     ↪  object, as well as relevant information.
    Attributes
    ----------
    cropped_image : array_like
        An array-like of the cropped image taken from the original image.
    left : int
        An int describing the position of the left border of the crop's bounding
↪   box in the original image.
    lower : int
        An int describing the position of the bottom border of the crop's bounding
↪   box in the original image.
    right : int
        An int describing the position of the right border of the crop's bounding
↪   box in the original image.
    upper : int
        An int describing the position of the top border of the crop's bounding
↪   box in the original image.

    Methods
    -------
    find(object_name: str)->List[ImagePatch]
        Returns a list of new ImagePatch objects containing crops of the image
↪   centered around any objects found in the image matching the object_name.
    simple_query(question: str=None)->str
        Returns the answer to a basic question asked about the image. If no
↪   question is provided, returns the answer to "What is this?".
    exists(object_name: str)->bool
        Returns True if the object specified by object_name is found in the image,
↪   and False otherwise.
    verify_property(property: str)->bool
        Returns True if the property is met, and False otherwise.
    best_text_match(string1: str, string2: str)->str
        Returns the string that best matches the image.
    crop(left: int, lower: int, right: int, upper: int)->ImagePatch
        Returns a new ImagePatch object containing a crop of the image at the
↪   given coordinates.
    get_operation_phase()->str
        Returns the phase of the operation at the time of the current image.
```

```
    analyse_audio(query_sound)->bool
        Returns if the queried sound is detected at the timestamp of the image.
    """

    def __init__(self, image, left: int=None, lower: int=None, right: int=None,
    ↪   upper: int=None):
        """Initializes an ImagePatch object by cropping the image at the given
        ↪   coordinates and stores the coordinates as attributes.
        If no coordinates are provided, the image is left unmodified, and the
↪   coordinates are set to the dimensions of the image.
        Parameters
        -------
        image : array_like
            An array-like of the original image.
        left : int
            An int describing the position of the left border of the crop's
↪   bounding box in the original image.
        lower : int
            An int describing the position of the bottom border of the crop's
↪   bounding box in the original image.
        right : int
            An int describing the position of the right border of the crop's
↪   bounding box in the original image.
        upper : int
            An int describing the position of the top border of the crop's
↪   bounding box in the original image.

        """
        if left is None and right is None and upper is None and lower is None:
            self.cropped_image = image
            self.left = 0
            self.lower = 0
            self.right = image.shape[2]  # width
            self.upper = image.shape[1]  # height
        else:
            self.cropped_image = image[:, lower:upper, left:right]
            self.left = left
            self.upper = upper
            self.right = right
            self.lower = lower

        self.width = self.cropped_image.shape[2]
        self.height = self.cropped_image.shape[1]

        self.horizontal_center = (self.left + self.right) / 2
        self.vertical_center = (self.lower + self.upper) / 2

    def find(self, object_name: str)->List["ImagePatch"]:
        """Returns a new ImagePatch object containing the crop of the image
        ↪   centered around the object specified by object_name.
        Parameters
        -------
        object_name : str
            A string describing the name of the object to be found in the image.
        """
        return find_in_image(self.cropped_image, object_name)
```

```
    def simple_query(self, question: str=None)->str:
        """Returns the answer to a basic question asked about the image. If no
        ↪  question is provided, returns the answer to "What is this?".
        Parameters
        -------
        question : str
            A string describing the question to be asked.

        Examples
        -------

        >>> # Which kind of animal is not eating?
        >>> def execute_command(image)->str:
        >>>     image_patch = ImagePatch(image)
        >>>     animal_patches = image_patch.find("animal")
        >>>     for animal_patch in animal_patches:
        >>>         if not animal_patch.verify_property("animal", "eating"):
        >>>             return animal_patch.simple_query("What kind of animal is
↪   eating?") # crop would include eating so keep it in the query
        >>>     # If no animal is not eating, query the image directly
        >>>     return image_patch.simple_query("Which kind of animal is not
↪   eating?")

        >>> # What is in front of the horse?
        >>> # contains a relation (around, next to, on, near, on top of, in front
↪   of, behind, etc), so ask directly
        >>> return image_patch.simple_query("What is in front of the horse?")
        >>>
        """
        return simple_qa(self.cropped_image, question, simple_relation_query)

    def exists(self, object_name: str)->bool:
        """Returns True if the object specified by object_name is found in the
        ↪  image, and False otherwise.
        Parameters
        -------
        object_name : str
            A string describing the name of the object to be found in the image.

        Examples
        -------
        >>> # Are there both cakes and gummy bears in the photo?
        >>> def execute_command(image)->str:
        >>>     image_patch = ImagePatch(image)
        >>>     is_cake = image_patch.exists("cake")
        >>>     is_gummy_bear = image_patch.exists("gummy bear")
        >>>     return bool_to_yesno(is_cake and is_gummy_bear)
        """
        return len(self.find(object_name)) > 0

    def verify_property(self, object_name: str, property: str)->bool:
        """Returns True if the object possesses the property, and False otherwise.
        Differs from 'exists' in that it presupposes the existence of the object
↪   specified by object_name, instead checking whether the object possesses the
↪   property.
        Parameters
        -------
```

```
        object_name : str
            A string describing the name of the object to be found in the image.
        property : str
            A string describing the property to be checked.

        Examples
        -------
        >>> # Do the letters have blue color?
        >>> def execute_command(image)->str:
        >>>     image_patch = ImagePatch(image)
        >>>     letters_patches = image_patch.find("letters")
        >>>     # Question assumes only one letter patch
        >>>     if len(letters_patches) == 0:
        >>>         # If no letters are found, query the image directly
        >>>         return image_patch.simple_query("Do the letters have blue
↪   color?")
        >>>     return bool_to_yesno(letters_patches[0].verify_property("letters",
↪   "blue"))
        """
        return verify_property(self.cropped_image, object_name, property)

    def best_text_match(self, option_list: List[str]) -> str:
        """Returns the string that best matches the image.
        Parameters
        -------
        option_list : str
            A list with the names of the different options
        prefix : str
            A string with the prefixes to append to the options

        Examples
        -------
        >>> # Is the cap gold or white?
        >>> def execute_command(image)->str:
        >>>     image_patch = ImagePatch(image)
        >>>     cap_patches = image_patch.find("cap")
        >>>     # Question assumes one cap patch
        >>>     if len(cap_patches) == 0:
        >>>         # If no cap is found, query the image directly
        >>>         return image_patch.simple_query("Is the cap gold or white?")
        >>>     return cap_patches[0].best_text_match(["gold", "white"])
        """
        return best_text_match(self.cropped_image, option_list)

    def crop(self, left: int, lower: int, right: int, upper: int)->"ImagePatch":
        """Returns a new ImagePatch cropped from the current ImagePatch. Only
        ↪  useful to create a focused image version enclosing multiple objects.
        Parameters
        -------
        left : int
            The leftmost pixel of the cropped image.
        lower : int
            The lowest pixel of the cropped image.
        right : int
            The rightmost pixel of the cropped image.
        upper : int
            The uppermost pixel of the cropped image.
```

```
        -------
        """
        return ImagePatch(self.cropped_image, left, lower, right, upper)

    def get_operation_phase(self)->str:
        """Returns the operation phase of the timestamp calculated from the
        ↪  current image index. Will resolve non-singular detections internally
        ↪  and
        directly return the final prediction.
        Possible answers are "", "Fallplanung", "RIO-Kontrolle vor OP",
↪   "Knochenregistrierung", "Intra-OP Planung", "Intra-OP Knochenvorbereitung",
↪   "Abschluss des Falls".

        Examples
        -------
        >>> # In what phase of the operation are we?
        >>> def execute_command(image)->str:
        >>>     image_patch = ImagePatch(image)
        >>>     return image_patch.get_operation_phase()
        """
        return get_operation_phase(self.cropped_image)

    def analyse_audio(self, query_sound: int)->bool:
        """Returns if the query sound is present at the time. Can only handle
        ↪  specified set of query sounds.
        Parameters
        -------
        query_sound : str
            One of the following:
                -sawing
                -drilling
                -hammering
                -rustling
                -tool clanking
                -talking

        Examples
        -------
        >>> # Is there drilling?
        >>> def execute_command(image)->str:
        >>>     image_patch = ImagePatch(image)
        >>>     sound_detected = image_patch.analyse_audio("drilling")
        >>>     if sound_detected:
        >>>         # either head surgeon or assistant surgeon can be holding the
↪   corresponding tool
        >>>         head_surgeon_drilling = image_patch.simple_query("Is the head
↪   surgeon drilling the patient?")
        >>>         assistant_surgeon_drilling = image_patch.simple_query("Is the
↪   assistant surgeon drilling the patient?")
        >>>         if head_surgeon_drilling == "yes" or
↪   assistant_surgeon_drilling == "yes":
        >>>             return "yes"
        >>>     return "no"
        """
        return analyse_audio(query_sound)

# Examples of using ImagePatch
```

```python
# Is there a person?
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    person_exists = image_patch.exists("person")
    return bool_to_yesno(person_exists)

# Is every person wearing scrubs?
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    person_patches = image_patch.find("person")
    # ask about every person
    is_wearing = []
    for person_patch in person_patches:
        is_wearing.append(person_patch.simple_query("Is the person wearing
        ↪  scrubs?"))
    return "-".join(is_wearing)

# How many people are there?
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    person_patches = image_patch.find("person")
    return len(person_patches)

# What is the relation between the anesthetist and the operating table?
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    anesthetist_patches = image_patch.find("anesthetist")
    operating_table_patches = image_patch.find("operating table")
    if len(anesthetist_patches) == 0 or len(operating_table_patches) == 0:
        # If no anesthetist or operating table is found, query the image directly
        return image_patch.simple_query("What is the relation between the
        ↪  anesthetist and the operating table?")
    # Question assumes only one anesthetist and one operating table
    anesthetist_patch = anesthetist_patches[0]
    operating_table_patch = operating_table_patches[0]
    union_patch = image_patch.crop(min(anesthetist_patch.left,
    ↪  operating_table_patch.left), min(anesthetist_patch.lower,
    ↪  operating_table_patch.lower),\
                                   max(anesthetist_patch.right,
                                   ↪  operating_table_patch.right),
                                   ↪  max(anesthetist_patch.upper,
                                   ↪  operating_table_patch.upper))
    return union_patch.simple_query("What is the relation between the anesthetist
    ↪  and the operating table?")

# INSERT_QUERY_HERE
def execute_command(image)->str:
```

A.2 LLM Prompt Template for Chain-of-Thought

```python
from PIL import Image
from vision_functions import find_in_image, simple_qa, verify_property,
↪  best_text_match, get_operation_phase, analyse_audio
```

```python
def bool_to_yesno(bool_answer: bool)->str:
    return "yes" if bool_answer else "no"

class ImagePatch:
    """A Python class containing a crop of an image centered around a particular
    ↪ object, as well as relevant information.
    Attributes
    ----------
    cropped_image : array_like
        An array-like of the cropped image taken from the original image.
    left : int
        An int describing the position of the left border of the crop's bounding
↪   box in the original image.
    lower : int
        An int describing the position of the bottom border of the crop's bounding
↪   box in the original image.
    right : int
        An int describing the position of the right border of the crop's bounding
↪   box in the original image.
    upper : int
        An int describing the position of the top border of the crop's bounding
↪   box in the original image.

    Methods
    -------
    find(object_name: str)->List[ImagePatch]
        Returns a list of new ImagePatch objects containing crops of the image
↪   centered around any objects found in the image matching the object_name.
    simple_query(question: str=None)->str
        Returns the answer to a basic question asked about the image. If no
↪   question is provided, returns the answer to "What is this?".
    exists(object_name: str)->bool
        Returns True if the object specified by object_name is found in the image,
↪   and False otherwise.
    verify_property(property: str)->bool
        Returns True if the property is met, and False otherwise.
    best_text_match(string1: str, string2: str)->str
        Returns the string that best matches the image.
    crop(left: int, lower: int, right: int, upper: int)->ImagePatch
        Returns a new ImagePatch object containing a crop of the image at the
↪   given coordinates.
    get_operation_phase()->str
        Returns the phase of the operation at the time of the current image.
    analyse_audio(query_sound)->bool
        Returns if the queried sound is detected at the timestamp of the image.
    """

    def __init__(self, image, left: int=None, lower: int=None, right: int=None,
    ↪   upper: int=None):
        """Initializes an ImagePatch object by cropping the image at the given
        ↪ coordinates and stores the coordinates as attributes.
        If no coordinates are provided, the image is left unmodified, and the
↪   coordinates are set to the dimensions of the image.
        Parameters
        -------
        image : array_like
            An array-like of the original image.
```

```
        left : int
            An int describing the position of the left border of the crop's
↪   bounding box in the original image.
        lower : int
            An int describing the position of the bottom border of the crop's
↪   bounding box in the original image.
        right : int
            An int describing the position of the right border of the crop's
↪   bounding box in the original image.
        upper : int
            An int describing the position of the top border of the crop's
↪   bounding box in the original image.

        """
        if left is None and right is None and upper is None and lower is None:
            self.cropped_image = image
            self.left = 0
            self.lower = 0
            self.right = image.shape[2]  # width
            self.upper = image.shape[1]  # height
        else:
            self.cropped_image = image[:, lower:upper, left:right]
            self.left = left
            self.upper = upper
            self.right = right
            self.lower = lower

        self.width = self.cropped_image.shape[2]
        self.height = self.cropped_image.shape[1]

        self.horizontal_center = (self.left + self.right) / 2
        self.vertical_center = (self.lower + self.upper) / 2

    def find(self, object_name: str)->List["ImagePatch"]:
        """Returns a new ImagePatch object containing the crop of the image
        ↪  centered around the object specified by object_name.
        Parameters
        -------
        object_name : str
            A string describing the name of the object to be found in the image.
        """
        return find_in_image(self.cropped_image, object_name)

    def simple_query(self, question: str=None)->str:
        """Returns the answer to a basic question asked about the image based on a
        ↪  general purpose vision model. If no question is provided, returns the
        ↪  answer to "What is this?".
        Parameters
        -------
        question : str
            A string describing the question to be asked.

        Examples
        -------
        >>> # What is in front of the horse?
        >>> return image_patch.simple_query("What is in front of the horse?")
        >>>
```

```
        """
        return simple_qa(self.cropped_image, question, simple_relation_query)

    def exists(self, object_name: str)->bool:
        """Returns True if the object specified by object_name is found in the
        ↪  image, and False otherwise.
        Parameters
        -------
        object_name : str
            A string describing the name of the object to be found in the image.

        Examples
        -------
        >>> # Are there both cakes and gummy bears in the photo?
        >>> def execute_command(image)->str:
        >>>     image_patch = ImagePatch(image)
        >>>     is_cake = image_patch.exists("cake")
        >>>     is_gummy_bear = image_patch.exists("gummy bear")
        >>>     return bool_to_yesno(is_cake and is_gummy_bear)
        """
        return len(self.find(object_name)) > 0

    def verify_property(self, object_name: str, property: str)->bool:
        """Returns True if the object possesses the property, and False otherwise.
        Differs from 'exists' in that it presupposes the existence of the object
↪   specified by object_name, instead checking whether the object possesses the
↪   property.
        Parameters
        -------
        object_name : str
            A string describing the name of the object to be found in the image.
        property : str
            A string describing the property to be checked.

        Examples
        -------
        >>> # Do the letters have blue color?
        >>> def execute_command(image)->str:
        >>>     image_patch = ImagePatch(image)
        >>>     letters_patches = image_patch.find("letters")
        >>>     # Question assumes only one letter patch
        >>>     if len(letters_patches) == 0:
        >>>         # If no letters are found, query the image directly
        >>>         return image_patch.simple_query("Do the letters have blue
↪   color?")
        >>>     return bool_to_yesno(letters_patches[0].verify_property("letters",
↪   "blue"))
        """
        return verify_property(self.cropped_image, object_name, property)

    def best_text_match(self, option_list: List[str]) -> str:
        """Returns the string that best matches the image.
        Parameters
        -------
        option_list : str
            A list with the names of the different options
        prefix : str
```

```
            A string with the prefixes to append to the options

        Examples
        -------
        >>> # Is the cap gold or white?
        >>> def execute_command(image)->str:
        >>>     image_patch = ImagePatch(image)
        >>>     cap_patches = image_patch.find("cap")
        >>>     # Question assumes one cap patch
        >>>     if len(cap_patches) == 0:
        >>>         # If no cap is found, query the image directly
        >>>         return image_patch.simple_query("Is the cap gold or white?")
        >>>     return cap_patches[0].best_text_match(["gold", "white"])
        """
        return best_text_match(self.cropped_image, option_list)

    def crop(self, left: int, lower: int, right: int, upper: int)->"ImagePatch":
        """Returns a new ImagePatch cropped from the current ImagePatch. Only
        ↪ useful to create a focused image version enclosing multiple objects.
        Parameters
        -------
        left : int
            The leftmost pixel of the cropped image.
        lower : int
            The lowest pixel of the cropped image.
        right : int
            The rightmost pixel of the cropped image.
        upper : int
            The uppermost pixel of the cropped image.
        -------
        """
        return ImagePatch(self.cropped_image, left, lower, right, upper)

    def get_operation_phase(self)->str:
        """Returns the operation phase of the timestamp calculated from the
        ↪ current image index. Will resolve non-singular detections internally
        ↪ and
        directly return the final prediction.
        Possible answers are "", "Fallplanung", "RIO-Kontrolle vor OP",
↪   "Knochenregistrierung", "Intra-OP Planung", "Intra-OP Knochenvorbereitung",
↪   "Abschluss des Falls".

        Examples
        -------
        >>> # In what phase of the operation are we?
        >>> def execute_command(image)->str:
        >>>     image_patch = ImagePatch(image)
        >>>     return image_patch.get_operation_phase()
        """
        return get_operation_phase(self.cropped_image)

    def analyse_audio(self, query_sound: int)->bool:
        """Returns if the query sound is present at the time. Can only handle
        ↪ specified set of query sounds.
        Parameters
        -------
        query_sound : str
```

```
            One of the following:
                -sawing
                -drilling
                -hammering
                -rustling
                -tool clanking
                -talking

        Examples
        -------
        >>> # Is there rustling?
        >>> def execute_command(image)->str:
        >>>     image_patch = ImagePatch(image)
        >>>     return bool_to_yesno(image_patch.analyse_audio("rustling"))
        """
        return analyse_audio(query_sound)

"""
########### Important information to consider ###########

You are answering queries about knee replacement surgeries based on the following
↪  information.
The queries are about an image from some timepoint of the complete surgery.

The audio model behind analyse_audio() produces high false positives and high
↪  recall, i.e. has a positive bias.

Operating tools are only used by the head surgeon and assistant surgeon.

The vision model used by the methods find() and exists() should be queried with
↪  the specific instruments drill, saw, and hammer,
instead of "instruments", as it can detect them better than the general term
↪  "instruments".

The temporal order of the phases the operation goes through is: "Fallplanung",
↪  "RIO-Kontrolle vor OP", "Knochenregistrierung", "Intra-OP Planung",
"Intra-OP Knochenvorbereitung", "Abschluss des Falls".

Sawing is only performed in the operation phase "Intra-OP Knochenvorbereitung". It
↪  is always performed in that phase.

The patient is lying on the operating table in all phases except "Fallplanung".

The robot is in use during the phase "Intra-OP Knochenvorbereitung".

The object "monitor" refers to the monitor which is next to the tracker. There are
↪  other irrelevant monitors in the
operating room. In any query about the monitor, it should be checked that it is
↪  located next to the tracker.

Sterility is fulfilled by a person, if they are wearing gloves.

For the phase "Knochenregistrierung" to start the robot must stand next to the
↪  operating table
and the nurse must stand next to the patient.
"""
```

```python
# Examples of using ImagePatch

# Query:
# Is there drilling?
# Thoughts:
# Drilling produces a distinct sound that the audio analysis model is designed to
↪  detect.
# Since the audio model has a positive bias it will sometimes detect a sound when
↪  there is none.
# It is therefore necessary to add a second check if the audio model predicts a
↪  sound. This can be done using the general purpose vision model.
# As the operating tools can only be used by head surgeon or assistant surgeon, it
↪  has to be checked if either of them is performing the action, in this case
↪  drilling,
# on the patient. If this check is also positive, the action can be confirmed.
# Code:
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    sound_detected = image_patch.analyse_audio("drilling")
    if sound_detected:
        # either head surgeon or assistant surgeon can be holding the
         ↪  corresponding tool
        head_surgeon_drilling = image_patch.simple_query("Is the head surgeon
         ↪  drilling the patient?")
        assistant_surgeon_drilling = image_patch.simple_query("Is the assistant
         ↪  surgeon drilling the patient?")
        if head_surgeon_drilling == "yes" or assistant_surgeon_drilling == "yes":
            return "yes"
    return "no"

# Query:
# What is the relation between the anesthetist and the operating table?
# Thoughts:
# First, the object detection model has to be queried with the two relevant
↪  objects "anesthetist" and "operating table". If not found, ask the general
↪  purpose vision model directly.
# If they are both found, an image crop enclosing both of them should be created.
↪  Based on this focused view the general purpose vision model is queried.
# Code:
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    anesthetist_patches = image_patch.find("anesthetist")
    operating_table_patches = image_patch.find("operating table")
    if len(anesthetist_patches) == 0 or len(operating_table_patches) == 0:
        return image_patch.simple_query("What is the relation between the
         ↪  anesthetist and the operating table?")
    anesthetist_patch = anesthetist_patches[0]
    operating_table_patch = operating_table_patches[0]
    union_patch = image_patch.crop(min(anesthetist_patch.left,
     ↪  operating_table_patch.left), min(anesthetist_patch.lower,
     ↪  operating_table_patch.lower),\
                                   max(anesthetist_patch.right,
                                    ↪  operating_table_patch.right),
                                    ↪  max(anesthetist_patch.upper,
                                    ↪  operating_table_patch.upper))
    return union_patch.simple_query("What is the relation between the anesthetist
     ↪  and the operating table?")
```

```
# Query:
# How many people are there?
# Thoughts:
# For this query, the functionality of the method find() suffices, as it returns
↪  all detections of the query object in a list. So find() has to be queried with
↪  "people"
# and the length of the list is returned as the final answer.
# Code:
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    person_patches = image_patch.find("person")
    return len(person_patches)

# Query:
# INSERT_QUERY_HERE
# Thoughts:
```

A.3 LLM-as-a-judge Accuracy Metric Prompt Template

```
You are a tough but fair judge asessing if answers to a query match the correct
↪  answer to that query sufficiently.
You are provided with the query, the given answer, that is to be graded, and the
↪  correct answer to the query. You have to provide your reasoning on if the
↪  given answer matches the correct answer semantically.
It does not matter if the given answer is correct in itself, but only if it
↪  matches the correct answer semantically.
Only then you give a score of 1, otherwise 0. Other scores are not possible.

Examples for your reference:

# Example 1
Query: Is the patient doing something with the operating table?
Given answer: Yes, the patient is lying on the operating table, which is a part of
↪  the training setup for the knee replacement surgery.
Correct answer: The patient is lying on the operating table
Reasoning: It is correctly specified that the patient is lying on the operating
↪  table.
Score: 1

# Example 2
Query: What is the relation between the nurse and the instrument table?
Given answer: the nurse is on the instrument table
Correct answer: The nurse is close to the instrument table
Reasoning: If a person "is close to" some object this typically refers to the
↪  person standing in the proximity of that object, but not "being on top of"
↪  that object.
So this is not correct.
Score: 0

# Example 3
Query: What is the relation between the nurse and the instrument table?
Given answer: The nurse is standing next to the instrument table, which is likely
↪  being used to hold surgical instruments during the knee replacement surgery.
Correct answer: The nurse is close to the instrument table
```

```
Reasoning: The nurse "being close to" the instrument table implies standing.
 ↪  "Close to" and "next to" describe the same thing.
So while the phrasing of the answer does not perfectly match the ground truth it
 ↪  refers to the same action and is therefore correct.
Score: 1

# Example 4
Query: Is the nurse doing something with the instrument table?
Given answer: Yes
Correct answer: The nurse is close to the instrument table
Reasoning: Just saying "yes" or "no" is not a sufficient answer when asked about
 ↪  the specific action or relation between two objects.
So this is wrong because the action is not further specified.
Score: 0

# Example 5
Query: What is the mps doing with the patient?
Given answer: performing surgery
Correct answer: The mps is preparing the patient
Reasoning: Preparing a patient and performing surgery on a patient are not the
 ↪  same thing, so the given answer is wrong.
Score: 0

# Example 6
Query: What is the circulator doing with the instrument table?
Given answer: he is moving it
Correct answer: The circulator is close to the instrument table
Reasoning: It is claimed that the table is being moved, but the correct answer
 ↪  does not say anything about that but only, that the circulator is close to the
 ↪  table.
Therefore this is incorrect.
Score: 0

Example 7
Query: What is the anesthetist doing with the patient?
Given answer: giving anesthesia
Correct answer: The anesthetist is close to the patient
Reasoning: While giving anesthesia is typically something that is performed by an
 ↪  anesthetist on a patient, the correct answer does not say anything about that
 ↪  in this case.
It only specifies that the anesthetist is close to the patient, which does not
 ↪  imply any further actions by the anesthetist.
Because the given answer describes an action that is not provided in the correct
 ↪  answer, the given answer is wrong.
Score: 0

Now judge this one:

Query: INSERT QUERY
Given answer: INSERT ANSWER
Correct answer: INSERT GROUND TRUTH
Reasoning:
```

A.4 Chain-of-Though Results

A.4.1 Code Llama

```
# Query: Is there a person?

######## Standard ########
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    person_patches = image_patch.find("person")
    return bool_to_yesno(len(person_patches) > 0)

######## CoT ########
# Thoughts:
# For this query, the functionality of the method exists() suffices, as it returns
↪  True if the query object is found in the image and False otherwise.
# So exists() has to be queried with "person" and the answer is returned directly.
# Code:
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    return bool_to_yesno(image_patch.exists("person"))

#####################################################
# Query: Is every person wearing scrubs?

######## Standard ########
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    person_patches = image_patch.find("person")
    for person_patch in person_patches:
        if not person_patch.verify_property("person", "scrubs"):
            return "no"
    return "yes"

######## CoT ########
# Thoughts:
# For this query, the functionality of the method exists() suffices, as it returns
↪  True if the query object is found in the image and False otherwise.
# So exists() has to be queried with "scrubs" and the answer is returned as the
↪  final answer.
# Code:
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    scrubs_found = image_patch.exists("scrubs")
    return bool_to_yesno(scrubs_found)

#####################################################
# Query: Is the head surgeon holding an instrument?

######## Standard ########
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    head_surgeon_patches = image_patch.find("head surgeon")
    if len(head_surgeon_patches) == 0:
        return image_patch.simple_query("Is the head surgeon holding an
         ↪  instrument?")
    head_surgeon_patch = head_surgeon_patches[0]
```

```
    return bool_to_yesno(head_surgeon_patch.exists("instrument"))

######## CoT ########
# Thoughts:
# The general purpose vision model has to be queried with the object "head
↪  surgeon" and the property "holding instrument".
# Code:
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    return image_patch.verify_property("head surgeon", "holding instrument")

#####################################################
# Query: Is the hammer being used?

######## Standard ########
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    hammer_patches = image_patch.find("hammer")
    if len(hammer_patches) == 0:
        return image_patch.simple_query("Is the hammer being used?")
    return bool_to_yesno(hammer_patches[0].verify_property("hammer", "being
    ↪  used"))

######## CoT ########
# Thoughts:
# The object detection model has to be queried with the relevant object "hammer".
↪  If not found, ask the general purpose vision model directly.
# If it is found, it has to be checked if the head surgeon or assistant surgeon is
↪  using it. If this check is also positive, the action can be confirmed.
# Code:
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    hammer_patches = image_patch.find("hammer")
    if len(hammer_patches) == 0:
        return image_patch.simple_query("Is the hammer being used?")
    hammer_patch = hammer_patches[0]
    # either head surgeon or assistant surgeon can be holding the corresponding
    ↪  tool
    head_surgeon_hammering = image_patch.simple_query("Is the head surgeon
    ↪  hammering the patient?")
    assistant_surgeon_hammering = image_patch.simple_query("Is the assistant
    ↪  surgeon hammering the patient?")
    if head_surgeon_hammering == "yes" or assistant_surgeon_hammering == "yes":
        return "yes"
    return "no"

#####################################################
# Query: Is there sawing?

######## Standard ########
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    sound_detected = image_patch.analyse_audio("sawing")
    if sound_detected:
        # either head surgeon or assistant surgeon can be holding the
        ↪  corresponding tool
        head_surgeon_sawing = image_patch.simple_query("Is the head surgeon sawing
        ↪  the patient?")
```

```
        assistant_surgeon_sawing = image_patch.simple_query("Is the assistant
         ↪  surgeon sawing the patient?")
        if head_surgeon_sawing == "yes" or assistant_surgeon_sawing == "yes":
            return "yes"
    return "no"

######## CoT ########
# Thoughts:
# Sawing is only performed in the operation phase "Intra-OP Knochenvorbereitung".
↪  It is always performed in that phase.
# The audio model has a positive bias, so it will sometimes detect a sound when
↪  there is none.
# It is therefore necessary to add a second check if the audio model predicts a
↪  sound. This can be done using the general purpose vision model.
# As the operating tools can only be used by head surgeon or assistant surgeon, it
↪  has to be checked if either of them is performing the action, in this case
↪  sawing,
# on the patient. If this check is also positive, the action can be confirmed.
# Code:
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    operation_phase = image_patch.get_operation_phase()
    if operation_phase == "Intra-OP Knochenvorbereitung":
        sound_detected = image_patch.analyse_audio("sawing")
        if sound_detected:
            # either head surgeon or assistant surgeon can be holding the
            ↪  corresponding tool
            head_surgeon_sawing = image_patch.simple_query("Is the head surgeon
            ↪  sawing the patient?")
            assistant_surgeon_sawing = image_patch.simple_query("Is the assistant
            ↪  surgeon sawing the patient?")
            if head_surgeon_sawing == "yes" or assistant_surgeon_sawing == "yes":
                return "yes"
    return "no"

################################################
Query: Has there been sawing so far?

######## Standard ########
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    sawing_patches = image_patch.find("sawing")
    if len(sawing_patches) == 0:
        return "no"
    return "yes"

######## CoT ########
# Thoughts:
# The audio model has a positive bias, so it will sometimes detect a sound when
↪  there is none.
# The vision model has to be queried with the specific instrument "saw" to detect
↪  the sawing.
# Code:
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    sound_detected = image_patch.analyse_audio("sawing")
    if sound_detected:
```

```
        sawing_detected = image_patch.exists("saw")
        if sawing_detected:
            return "yes"
    return "no"

#####################################################
# Query: Which instruments are there?

######## Standard ########
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    instrument_patches = image_patch.find("instrument")
    instrument_names = []
    for instrument_patch in instrument_patches:
        instrument_names.append(instrument_patch.simple_query("What is this?"))
    return instrument_names

######## CoT ########
# Thoughts:
# For this query, the functionality of the method find() suffices, as it returns
↪   all detections of the query object in a list. So find() has to be queried with
↪   "instruments"
# and the names of the objects in the list are returned as the final answer.
# Code:
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    instrument_patches = image_patch.find("instrument")
    instrument_names = []
    for instrument_patch in instrument_patches:
        instrument_names.append(instrument_patch.simple_query("What is this?"))
    return ", ".join(instrument_names)

#####################################################
# Query: Is the head surgeon adhering to sterility?

######## Standard ########
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    head_surgeon_patches = image_patch.find("head surgeon")
    if len(head_surgeon_patches) == 0:
        return image_patch.simple_query("Is the head surgeon adhering to
         ↪   sterility?")
    head_surgeon_patch = head_surgeon_patches[0]
    if head_surgeon_patch.verify_property("head surgeon", "sterility"):
        return "yes"
    return "no"

######## CoT ########
# Thoughts:
# The head surgeon is the only person who can perform the action of drilling.
# Sterility is fulfilled by a person, if they are wearing gloves.
# The head surgeon is the only person who can be wearing gloves.
# Therefore, if the head surgeon is drilling and wearing gloves, the answer is
↪   yes.
# Code:
def execute_command(image)->str:
    image_patch = ImagePatch(image)
```

```
    head_surgeon_drilling = image_patch.simple_query("Is the head surgeon drilling
     ↪  the patient?")
    head_surgeon_wearing_gloves = image_patch.simple_query("Is the head surgeon
     ↪  wearing gloves?")
    if head_surgeon_drilling == "yes" and head_surgeon_wearing_gloves == "yes":
        return "yes"
    return "no"

#####################################################
# Query: Is the patient lying down?

######## Standard ########
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    patient_patches = image_patch.find("patient")
    if len(patient_patches) == 0:
        return image_patch.simple_query("Is the patient lying down?")
    patient_patch = patient_patches[0]
    return bool_to_yesno(patient_patch.lower > patient_patch.upper)

######## CoT ########
# Thoughts:
# The patient is lying down if the object "patient" is found in the image.
# Code:
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    return bool_to_yesno(image_patch.exists("patient"))

#####################################################
# Query: Is the monitor on?

######## Standard ########
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    monitor_patches = image_patch.find("monitor")
    if len(monitor_patches) == 0:
        return image_patch.simple_query("Is the monitor on?")
    monitor_patch = monitor_patches[0]
    if monitor_patch.verify_property("monitor", "on"):
        return "yes"
    return "no"

######## CoT ########
# Thoughts:
# The monitor is on if it is turned on and the patient is lying on the operating
 ↪  table.
# The monitor is turned on if the light is on.
# The light is on if the lamp is on.
# The lamp is on if the switch is on.
# The switch is on if the electricity is on.
# The electricity is on if the socket is on.
# The socket is on if the plug is on.
# The plug is on if the outlet is on.
# The outlet is on if the socket is on.
# The socket is on if the electricity is on.
# The electricity is on if the plug is on.
# The plug is on if the outlet is on.
```

```
# The outlet is on if the socket is on.
# The socket is on if the lamp is on.
# The lamp is on if the light is on.
# The light is on if the monitor is on.
# The monitor is on if the patient is lying on the operating table.
# Code:
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    monitor_patches = image_patch.find("monitor")
    if len(monitor_patches) == 0:
        return "no"
    monitor_patch = monitor_patches[0]
    if monitor_patch.verify_property("monitor", "on"):
        return "yes"
    return "no"

#####################################################
# Query: What is the next operation phase after the current one?

######## Standard ########
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    current_phase = image_patch.get_operation_phase()
    if current_phase == "Fallplanung":
        return "RIO-Kontrolle vor OP"
    elif current_phase == "RIO-Kontrolle vor OP":
        return "Knochenregistrierung"
    elif current_phase == "Knochenregistrierung":
        return "Intra-OP Planung"
    elif current_phase == "Intra-OP Planung":
        return "Intra-OP Knochenvorbereitung"
    elif current_phase == "Intra-OP Knochenvorbereitung":
        return "Abschluss des Falls"
    else:
        return ""

######## CoT ########
# Thoughts:
# The current operation phase can be determined by the method
↪   get_operation_phase().
# The next operation phase can be determined by a dictionary that maps the current
↪   operation phase to the next one.
# Code:
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    current_phase = image_patch.get_operation_phase()
    next_phase_dict = {"Fallplanung": "RIO-Kontrolle vor OP", "RIO-Kontrolle vor
    ↪   OP": "Knochenregistrierung", "Knochenregistrierung": "Intra-OP Planung",
    ↪   "Intra-OP Planung": "Intra-OP Knochenvorbereitung", "Intra-OP
    ↪   Knochenvorbereitung": "Abschluss des Falls"}
    return next_phase_dict[current_phase]

#####################################################
# Query: When is the robot being used?

######## Standard ########
def execute_command(image)->str:
```

```
    image_patch = ImagePatch(image)
    robot_patches = image_patch.find("robot")
    if len(robot_patches) == 0:
        return image_patch.simple_query("When is the robot being used?")
    robot_patch = robot_patches[0]
    return robot_patch.get_operation_phase()

######## CoT ########
# Thoughts:
# The robot is only used during the phase "Intra-OP Knochenvorbereitung".
# The general purpose vision model can be queried with the question "What is the
↪  operation phase of the timestamp calculated from the current image index?".
# If the answer is "Intra-OP Knochenvorbereitung", the robot is being used.
# Code:
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    operation_phase = image_patch.get_operation_phase()
    if operation_phase == "Intra-OP Knochenvorbereitung":
        return "yes"
    return "no"

#####################################################
# Query: Can the operation proceed with ""Knochenregistrierung""?

######## Standard ########
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    robot_patches = image_patch.find("robot")
    operating_table_patches = image_patch.find("operating table")
    nurse_patches = image_patch.find("nurse")
    patient_patches = image_patch.find("patient")
    if len(robot_patches) == 0 or len(operating_table_patches) == 0 or
     ↪  len(nurse_patches) == 0 or len(patient_patches) == 0:
        return image_patch.simple_query("Can the operation proceed with
         ↪  \"Knochenregistrierung\"?")
    robot_patch = robot_patches[0]
    operating_table_patch = operating_table_patches[0]
    nurse_patch = nurse_patches[0]
    patient_patch = patient_patches[0]
    union_patch = image_patch.crop(min(robot_patch.left,
     ↪  operating_table_patch.left, nurse_patch.left, patient_patch.left),
     ↪  min(robot_patch.lower, operating_table_patch.lower, nurse_patch.lower,
     ↪  patient_patch.lower),\
                                   max(robot_patch.right,
                                    ↪  operating_table_patch.right,
                                    ↪  nurse_patch.right, patient_patch.right),
                                    ↪  max(robot_patch.upper,
                                    ↪  operating_table_patch.upper,
                                    ↪  nurse_patch.upper, patient_patch.upper))
    return union_patch.simple_query("Can the operation proceed with
     ↪  \"Knochenregistrierung\"?")

######## CoT ########
# Thoughts:
# The operation can proceed with "Knochenregistrierung" if the robot is in use and
↪  the nurse is standing next to the patient.
# The robot is in use if it is standing next to the operating table.
```

```
# The nurse is standing next to the patient if she is standing next to the patient
↪  and the patient is lying on the operating table.
# Code:
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    robot_patches = image_patch.find("robot")
    nurse_patches = image_patch.find("nurse")
    patient_patches = image_patch.find("patient")
    operating_table_patches = image_patch.find("operating table")
    if len(robot_patches) == 0 or len(nurse_patches) == 0 or len(patient_patches)
    ↪  == 0 or len(operating_table_patches) == 0:
        return "no"
    robot_patch = robot_patches[0]
    nurse_patch = nurse_patches[0]
    patient_patch = patient_patches[0]
    operating_table_patch = operating_table_patches[0]
    robot_next_to_operating_table = robot_patch.left < operating_table_patch.right
    ↪  and robot_patch.right > operating_table_patch.left and robot_patch.lower <
    ↪  operating_table_patch.upper and robot_patch.upper >
    ↪  operating_table_patch.lower
    nurse_next_to_patient = nurse_patch.left < patient_patch.right and
    ↪  nurse_patch.right > patient_patch.left and nurse_patch.lower <
    ↪  patient_patch.upper and nurse_patch.upper > patient_patch.lower
    patient_lying_on_operating_table = patient_patch.lower <
    ↪  operating_table_patch.upper and patient_patch.upper >
    ↪  operating_table_patch.lower
    if robot_next_to_operating_table and nurse_next_to_patient and
    ↪  patient_lying_on_operating_table:
        return "yes"
    return "no"
```

A.4.2 Llama-3

```
# Query: Is there a person?

######## Standard ########
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    return bool_to_yesno(image_patch.exists("person"))

######## CoT ########
# Thoughts:
# This query is similar to the previous one, but it only needs to know if there is
↪  at least one person in the image. So the length of the list returned by find()
↪  is checked.
# Code:
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    person_patches = image_patch.find("person")
    return bool_to_yesno(len(person_patches) > 0)

#####################################################
# Query: Is every person wearing scrubs?

######## Standard ########
```

```python
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    person_patches = image_patch.find("person")
    if len(person_patches) == 0:
        return image_patch.simple_query("Is every person wearing scrubs?")
    for person_patch in person_patches:
        if not person_patch.exists("scrubs"):
            return "no"
    return "yes"

######## CoT ########
# Thoughts:
# First, the object detection model has to be queried with the object "person". If
↪  not found, ask the general purpose vision model directly.
# If found, the general purpose vision model is queried for each person if they
↪  are wearing scrubs.
# Code:
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    person_patches = image_patch.find("person")
    if len(person_patches) == 0:
        return image_patch.simple_query("Is every person wearing scrubs?")
    for person_patch in person_patches:
        scrubs_worn = person_patch.simple_query("Are scrubs worn?")
        if scrubs_worn != "yes":
            return "no"
    return "yes"

#####################################################
# Query: Is the head surgeon holding an instrument?

######## Standard ########
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    head_surgeon_patches = image_patch.find("head surgeon")
    instrument_patches = image_patch.find("instrument")
    if len(head_surgeon_patches) == 0 or len(instrument_patches) == 0:
        return image_patch.simple_query("Is the head surgeon holding an
         ↪  instrument?")
    head_surgeon_patch = head_surgeon_patches[0]
    instrument_patch = instrument_patches[0]
    union_patch = image_patch.crop(min(head_surgeon_patch.left,
     ↪  instrument_patch.left), min(head_surgeon_patch.lower,
     ↪  instrument_patch.lower),\
                                   max(head_surgeon_patch.right,
                                    ↪  instrument_patch.right),
                                    ↪  max(head_surgeon_patch.upper,
                                    ↪  instrument_patch.upper))
    return union_patch.simple_query("Is the head surgeon holding an instrument?")

######## CoT ########
# Thoughts:
# First, the object detection model has to be queried with the two relevant
↪  objects "head surgeon" and "instrument". If not found, ask the general purpose
↪  vision model directly.
# If they are both found, an image crop enclosing both of them should be created.
↪  Based on this focused view the general purpose vision model is queried.
```

```
# Code:
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    head_surgeon_patches = image_patch.find("head surgeon")
    instrument_patches = image_patch.find("instrument")
    if len(head_surgeon_patches) == 0 or len(instrument_patches) == 0:
        return image_patch.simple_query("Is the head surgeon holding an
         ↪  instrument?")
    head_surgeon_patch = head_surgeon_patches[0]
    instrument_patch = instrument_patches[0]
    union_patch = image_patch.crop(min(head_surgeon_patch.left,
     ↪  instrument_patch.left), min(head_surgeon_patch.lower,
     ↪  instrument_patch.lower),\
                                   max(head_surgeon_patch.right,
                                    ↪  instrument_patch.right),
                                    ↪  max(head_surgeon_patch.upper,
                                    ↪  instrument_patch.upper))
    return union_patch.simple_query("Is the head surgeon holding an instrument?")

#####################################################
# Query: Is the hammer being used?

######## Standard ########
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    hammer_patches = image_patch.find("hammer")
    if len(hammer_patches) == 0:
        return image_patch.simple_query("Is the hammer being used?")
    return bool_to_yesno(len(hammer_patches) > 0)

######## CoT ########
# Thoughts:
# The audio model can be used to detect the sound of the hammer. However, due to
↪  the high false positives, a second check is necessary.
# The vision model can be used to check if the head surgeon or assistant surgeon
↪  is holding the hammer.
# Code:
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    sound_detected = image_patch.analyse_audio("hammering")
    if sound_detected:
        # either head surgeon or assistant surgeon can be holding the hammer
        head_surgeon_hammering = image_patch.simple_query("Is the head surgeon
         ↪  holding the hammer?")
        assistant_surgeon_hammering = image_patch.simple_query("Is the assistant
         ↪  surgeon holding the hammer?")
        if head_surgeon_hammering == "yes" or assistant_surgeon_hammering ==
         ↪  "yes":
            return "yes"
    return "no"

#####################################################
# Query: Is there sawing?

######## Standard ########
def execute_command(image)->str:
    image_patch = ImagePatch(image)
```

```
    sound_detected = image_patch.analyse_audio("sawing")
    if sound_detected:
        return "yes"
    return "no"

######## CoT ########
# Thoughts:
# Sawing is only performed in the operation phase "Intra-OP Knochenvorbereitung".
↪   It is always performed in that phase.
# Therefore, the phase of the operation has to be determined first. If the phase
↪   is "Intra-OP Knochenvorbereitung", the audio model can be queried.
# If the phase is not "Intra-OP Knochenvorbereitung", the query can be answered
↪   directly with "no".
# Code:
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    operation_phase = image_patch.get_operation_phase()
    if operation_phase == "Intra-OP Knochenvorbereitung":
        sound_detected = image_patch.analyse_audio("sawing")
        return bool_to_yesno(sound_detected)
    return "no"

#####################################################
# Query: Has there been sawing so far?

######## Standard ########
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    return bool_to_yesno(image_patch.analyse_audio("sawing"))

######## CoT ########
# Thoughts:
# Sawing is only performed in the phase "Intra-OP Knochenvorbereitung". It is
↪   always performed in that phase.
# Therefore, the phase of the operation at the time of the image can be checked.
↪   If it is "Intra-OP Knochenvorbereitung", then sawing has been performed.
# Code:
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    operation_phase = image_patch.get_operation_phase()
    return bool_to_yesno(operation_phase == "Intra-OP Knochenvorbereitung")

#####################################################
# Query: Which instruments are there?

######## Standard ########
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    drill_patches = image_patch.find("drill")
    saw_patches = image_patch.find("saw")
    hammer_patches = image_patch.find("hammer")
    if len(drill_patches) == 0 and len(saw_patches) == 0 and len(hammer_patches)
    ↪   == 0:
        return image_patch.simple_query("Which instruments are there?")
    return "drill" if len(drill_patches) > 0 else "saw" if len(saw_patches) > 0
    ↪   else "hammer" if len(hammer_patches) > 0 else "none"
```

```python
######## CoT ########
# Thoughts:
# The method find() is used to find all instruments in the image. The list of
↪  patches is then iterated over and the names of the instruments are returned.
# Code:
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    instrument_patches = image_patch.find("instruments")
    instrument_names = []
    for patch in instrument_patches:
        instrument_name = patch.best_text_match(["drill", "saw", "hammer"])
        instrument_names.append(instrument_name)
    return ", ".join(instrument_names)

#####################################################
# Query: Is the head surgeon adhering to sterility?

######## Standard ########
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    head_surgeon_patches = image_patch.find("head surgeon")
    if len(head_surgeon_patches) == 0:
        return image_patch.simple_query("Is the head surgeon adhering to
        ↪  sterility?")
    head_surgeon_patch = head_surgeon_patches[0]
    gloves_patches = image_patch.find("gloves")
    if len(gloves_patches) == 0:
        return image_patch.simple_query("Is the head surgeon adhering to
        ↪  sterility?")
    gloves_patch = gloves_patches[0]
    union_patch = image_patch.crop(min(head_surgeon_patch.left,
    ↪  gloves_patch.left), min(head_surgeon_patch.lower, gloves_patch.lower),\
                                 max(head_surgeon_patch.right,
                                 ↪  gloves_patch.right),
                                 ↪  max(head_surgeon_patch.upper,
                                 ↪  gloves_patch.upper))
    return union_patch.simple_query("Is the head surgeon adhering to sterility?")

######## CoT ########
# Thoughts:
# The head surgeon is adhering to sterility if they are wearing gloves. The vision
↪  model can be queried directly for this information.
# Code:
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    head_surgeon_patches = image_patch.find("head surgeon")
    if len(head_surgeon_patches) == 0:
        return image_patch.simple_query("Is the head surgeon adhering to
        ↪  sterility?")
    head_surgeon_patch = head_surgeon_patches[0]
    return bool_to_yesno(head_surgeon_patch.verify_property("head surgeon",
    ↪  "wearing gloves"))

#####################################################
# Query: Is the patient lying down?

######## Standard ########
```

```python
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    patient_patches = image_patch.find("patient")
    if len(patient_patches) == 0:
        return image_patch.simple_query("Is the patient lying down?")
    patient_patch = patient_patches[0]
    return bool_to_yesno(patient_patch.lower > 0)

######## CoT ########
# Thoughts:
# The patient is lying on the operating table in all phases except "Fallplanung".
↪   So the vision model has to be queried with the object "patient" and the object
↪   "operating table".
# If the patient is found on the operating table, the answer is "yes", otherwise
↪   "no".
# Code:
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    patient_patches = image_patch.find("patient")
    operating_table_patches = image_patch.find("operating table")
    if len(patient_patches) == 0 or len(operating_table_patches) == 0:
        return image_patch.simple_query("Is the patient lying down?")
    patient_patch = patient_patches[0]
    operating_table_patch = operating_table_patches[0]
    if patient_patch.lower > operating_table_patch.lower:
        return "no"
    return "yes"

#####################################################
# Query: Is the monitor on?

######## Standard ########
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    monitor_patches = image_patch.find("monitor")
    if len(monitor_patches) == 0:
        return image_patch.simple_query("Is the monitor on?")
    monitor_patch = monitor_patches[0]
    tracker_patches = image_patch.find("tracker")
    if len(tracker_patches) == 0:
        return "no"
    tracker_patch = tracker_patches[0]
    union_patch = image_patch.crop(min(monitor_patch.left, tracker_patch.left),
    ↪   min(monitor_patch.lower, tracker_patch.lower),\
                                 max(monitor_patch.right, tracker_patch.right),
                                 ↪   max(monitor_patch.upper,
                                 ↪   tracker_patch.upper))
    return bool_to_yesno(union_patch.exists("tracker")) and
    ↪   bool_to_yesno(union_patch.exists("monitor"))

######## CoT ########
# Thoughts:
# The monitor is only on if it is located next to the tracker. So first, the
↪   object detection model has to be queried with the two relevant objects
↪   "monitor" and "tracker".
# If not found, ask the general purpose vision model directly. If they are both
↪   found, an image crop enclosing both of them should be created.
```

```
# Based on this focused view the general purpose vision model is queried.
# Code:
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    monitor_patches = image_patch.find("monitor")
    tracker_patches = image_patch.find("tracker")
    if len(monitor_patches) == 0 or len(tracker_patches) == 0:
        return image_patch.simple_query("Is the monitor on?")
    monitor_patch = monitor_patches[0]
    tracker_patch = tracker_patches[0]
    union_patch = image_patch.crop(min(monitor_patch.left, tracker_patch.left),
     ↪  min(monitor_patch.lower, tracker_patch.lower),\
                                   max(monitor_patch.right, tracker_patch.right),
                                    ↪  max(monitor_patch.upper,
                                    ↪  tracker_patch.upper))
    return union_patch.simple_query("Is the monitor on?")

#####################################################
# Query: What is the next operation phase after the current one?

######## Standard ########
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    current_phase = image_patch.get_operation_phase()
    if current_phase == "Fallplanung":
        return "RIO-Kontrolle vor OP"
    elif current_phase == "RIO-Kontrolle vor OP":
        return "Knochenregistrierung"
    elif current_phase == "Knochenregistrierung":
        return "Intra-OP Planung"
    elif current_phase == "Intra-OP Planung":
        return "Intra-OP Knochenvorbereitung"
    elif current_phase == "Intra-OP Knochenvorbereitung":
        return "Abschluss des Falls"
    else:
        return "unknown"

######## CoT ########
# Thoughts:
# The current operation phase is determined by the general purpose vision model.
↪  The next phase is determined by the temporal order of the phases.
# Code:
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    current_phase = image_patch.get_operation_phase()
    phases = ["Fallplanung", "RIO-Kontrolle vor OP", "Knochenregistrierung",
     ↪  "Intra-OP Planung", "Intra-OP Knochenvorbereitung", "Abschluss des Falls"]
    phase_index = phases.index(current_phase)
    if phase_index == len(phases) - 1:
        return "Abschluss des Falls"
    else:
        return phases[phase_index + 1]

#####################################################
# Query: When is the robot being used?

######## Standard ########
```

```python
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    robot_patches = image_patch.find("robot")
    if len(robot_patches) == 0:
        return image_patch.simple_query("When is the robot being used?")
    robot_patch = robot_patches[0]
    return robot_patch.simple_query("When is the robot being used?")

######## CoT ########
# Thoughts:
# The robot is only used during the phase "Intra-OP Knochenvorbereitung". So the
↪   method get_operation_phase() has to be queried with the current image.
# If the result is "Intra-OP Knochenvorbereitung", the robot is being used.
# Code:
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    operation_phase = image_patch.get_operation_phase()
    return bool_to_yesno(operation_phase == "Intra-OP Knochenvorbereitung")

#####################################################
# Query: Can the operation proceed with "Knochenregistrierung"?

######## Standard ########
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    robot_patches = image_patch.find("robot")
    nurse_patches = image_patch.find("nurse")
    patient_patches = image_patch.find("patient")
    if len(robot_patches) == 0 or len(nurse_patches) == 0 or len(patient_patches)
    ↪   == 0:
        return image_patch.simple_query("Can the operation proceed with
        ↪   'Knochenregistrierung'?")
    robot_patch = robot_patches[0]
    nurse_patch = nurse_patches[0]
    patient_patch = patient_patches[0]
    if robot_patch.left > patient_patch.right or robot_patch.right <
    ↪   patient_patch.left or nurse_patch.lower > patient_patch.upper or
    ↪   nurse_patch.upper < patient_patch.lower:
        return "no"
    return "yes"

######## CoT ########
# Thoughts:
# The operation can proceed with "Knochenregistrierung" if the robot is standing
↪   next to the operating table and the nurse is standing next to the patient.
# The robot and the nurse are detected by the object detection model. If they are
↪   both found, the operation can proceed.
# Code:
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    robot_patches = image_patch.find("robot")
    nurse_patches = image_patch.find("nurse")
    if len(robot_patches) > 0 and len(nurse_patches) > 0:
        return "yes"
    return "no"
```