M^{3}Builder: A Multi-Agent System for Automated Machine Learning in Medical Imaging

Anonymous ACL submission

Abstract

In this paper, we aim to automate the labor-intensive process of developing machine learning (ML)-based tools for medical imaging, paving the way for the self-evolution of medical agentic systems. **(i)** We present **M^{3}Builder**, a multi-agent collaboration framework designed to automate model training in medical imaging, that divide-and-conquers complex medical ML with four specialized agents. **(ii)** To better fit in the professional medical imaging domain, we build up a specialized ML context protocol, a structured environment designed to provide agents with comprehensive free-text descriptions of medical datasets, training code templates, and interaction tools. **(iii)** To monitor the progress, we propose **M^{3}Bench**, spanning four medical imaging ML tasks across 14 datasets, covering both 2D and 3D data. Our experiments demonstrate that, when employing an identical agent core, M^{3}Builder surpasses existing automated ML agentic architectures, achieving a superior task completion rate of 94.29% while maintaining satisfactory model performance. This highlights the potential of fully automated ML-based tool development in medical imaging. Code will be publicly available upon publication.

1 Introduction

Large Language Model (LLM)-powered agentic systems have demonstrated remarkable success across diverse domains. Leveraging their ability to orchestrate specialized tools, they can solve complex, multi-step tasks with precision. However, their application in medical remains challenging. Medical tasks are typically professional and complex, encompassing a wide range of diseases, imaging modalities, and task-specific requirements. This diversity makes it difficult for preparing a complete toolset to implement a specific medical agentic systems, often necessitating further labor-intensive development of specialized tools, *e.g.*, training a machine-learning-based model. However, clinicians often lack the coding skills or technical expertise to implement such processes, significantly hindering the tailored adaptation of agentic systems in medical practice.

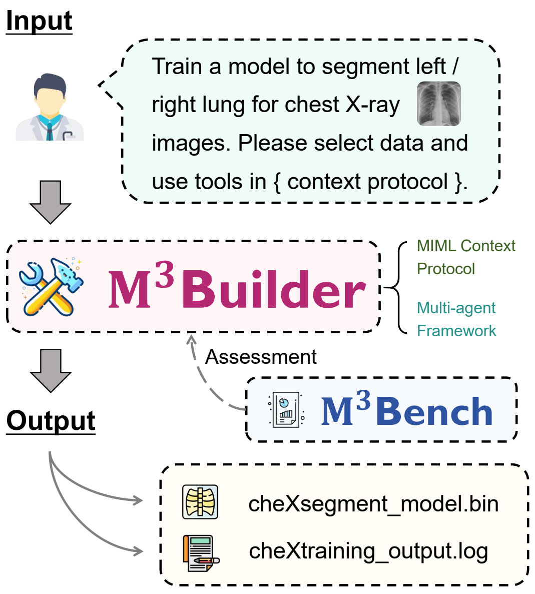


Figure 1: Overview of our proposed M^{3}Builder and M^{3}Bench. From user text input to model deployment.

In this paper, we propose **M^{3}Builder** (M^3 stands for **M**utli-agent, **M**achine Learning, and **M**edical Imaging), a novel agentic framework for automating medical imaging machine learning (MIML) tasks. Given a MIML tool development demand and raw training data, **M^{3}Builder** autonomously manages the entire process, from data preparation to model construction, training, and deployment. When integrated with other medical agentic systems, the overall system gains the ability to self-evolve, automatically developing new ML-based medical imaging tools, significantly alleviating the burden of manual tool collection and creation.

Specifically, M^{3}Builder employs a multi-agent collaboration framework, where four role-playing LLM agents work together to execute medical imaging ML tasks step-by-step. These include task management, data engineering, module architecture design, and model training. In each step, the system will self-refine its intermediate generation based on Python execution feedbacks.

Moreover, we design a new MIML-centered agentic context protocol. It includes: a free-text dataset description template emphasizing critical taxonomy tags, a collection of coding templates to structure agent outputs and reduce coding complexity, and a suite of supporting documentation tool functions. This protocol helps integrate MIML coding priors into the agentic workflow, simplifies system processes, emphasizes key MIML coding elements, and underlines core coding lines to enable seamless human-in-the-loop auditing.

Lastly, we introduce a benchmark, **M^{3}Bench**, designed to evaluate the performance for automated MIML. **M^{3}Bench** comprises four general tasks: *organ segmentation*, *anomaly detection*, *disease diagnosis*, and *report generation*. These tasks are associated with 14 distinct training datasets, spanning five anatomical regions and three primary imaging modalities, encompassing both 2D and 3D models. By covering a broad range of tasks and datasets, M^{3}Bench offers a comprehensive quantitative assessment of the automated MIML capabilities of various agentic systems.

Experimentally, to validate the effectiveness of the proposed agentic system, we compare against other state-of-the-art (SoTA) ML agentic systems, including ML-AgentBench (Huang et al., 2023a), Aider (Gauthier, 2023), MetaGPT (Hong et al., 2024), ToolMaker (Wölflein et al., 2025), Copilot Edits (GitHub, 2023) and so on under fair settings. Our results demonstrate that **M^{3}Builder** consistently achieves significant performance advancing across a range of metrics, achieving an average task execution success rate of 94.29% across four MIML tasks, where success is defined as producing a MIML model comparable (within 5 percentage below baseline method on) to the official baseline.

2 M^{3}Builder

2.1 Problem Formulation

Given a task description on medical imaging analysis, denoted as $\mathcal{T}$, our objective is to automatically construct a functional AI model via multi-agent collaboration. As shown in Fig. 2, our proposed framework **M^{3}Builder** comprises two key components: a MIML-centered context protocol ($\mathbb{P}$), and a multi-agent collaboration framework ($\mathcal{A}$). Specifically, the context protocol includes three elements: *data cards for multiple medical imaging dataset*, *MIML code templates*, and *toolset descriptions*. The data cards are represented in natural language, while the toolset descriptions and code templates are demonstrated in Python code. Together, this structured context protocol will then guides the multi-agent collaboration workflow, formulating their input instructions and output texts.

Building on the protocol ($\mathbb{P}$), the multi-agent framework composes of four LLM agents with distinct roles, *i.e.*, $\mathcal{A} = \{a_1, a_2, a_3, a_4\}$. These agents adopt a divide-and-conquer strategy to collaboratively address the MIML task. The framework iteratively performs code generation, executing it in Python environment and editing it using tools defined by *toolset descriptions*, until a functional AI model is successfully produced. This process can be expressed as:

$$\{\mathcal{C}_i, \mathcal{R}_i\} = \mathcal{A}(\mathcal{C}_{i-1}, \mathcal{R}_{i-1}, \mathcal{T} \mid \mathbb{P}), \qquad (1)$$

where $\mathcal{C}_i$ denotes code scripts generated or edited in the i_{th} iteration, $\mathcal{R}_i$ denotes the Python compiler feedback, with $\mathcal{C}_0 = \mathcal{R}_0 = \varnothing$.

2.2 MIML Context Protocol

The MIML context protocol formulates three core components for agentic system: (i) metadata specifications of available medical imaging datasets to guide dataset selection; (ii) MIML-specific code templates emphasizing critical coding elements; (iii) predefined toolset definitions to constrain agent action space. In summary, the context protocol equips the multi-agent framework with necessary resources for dataset preparation, a coding foundation, and a well-defined action space, corresponding to dataset descriptions, code templates, and toolset definitions.

Dataset Preparation. The context protocol includes a range of medical imaging datasets, each accompanied by a datacard for standardized descriptions. Each datacard contains a concise data summary covering dataset name, medical scope and metadata (e.g. medical imaging modalities, data format, spatial configurations of scans, annotation manners). This free-text format ensures that any description meeting these criteria qualifies as a valid datacard, including documentation provided with the dataset itself. To enable seamless integration, users can add new datasets by completing the corresponding datacards. Initially, the context protocol provides 14 medical imaging datasets with their datacards as examples.

Code Template Design. To streamline MIML training while maintaining flexibility, we prepare a set of standardized MIML code templates based on the `Transformers Trainer framework` and `nnU-Net`(Isensee et al., 2021). These coding templates are tailored to four primary medical imaging tasks: *disease diagnosis*, *organ segmentation*, *anomaly detection*, and *report generation*. Each task is implemented as a modular package with configurable components, such as the main forward architecture and network backbone options (2D/3D models). Shared features across tasks include a unified selection of loss functions, data augmentation strategies, training utilities, and architectural frameworks. By offering these templates, we reduce the complexity of free-form MIMIL coding while preserving adaptability for diverse tasks.

Interaction Toolset. To enable the agentic system to interact with the PC environment, we developed a set of interaction toolset functions, which serve as an instruction context for agents to utilize. This toolset comprises eight general PC-level interaction functions and five MIML-specific functions, as outlined in Tab. 1. In general functions, we include: *list_files*, *read_files*, *copy_files*, *write_files*, *edit_files*, and *edit_file*, inspired by MLAgentBench. Additionally, we supplement two extra tools, *preview_dirs* and *preview_files*. The former handles large dataset directories containing numerous files that exceed the LLM's context window, while the latter extracts key segments from oversized metadata files that cannot be fully processed by the *read_files* function. Then, in MIML-specific tools, we specifically developed five medical imaging-specific tools: *load_med_data* for loading various common medical data formats; *check_3D* for examining spatial information of 3D format images(nii, dcm, tif); *normalize_image* for performing image window clipping and normalization; *verify_report* for detecting noise in imaging reports; and *augment_image* for implementing common medical data augmentations.

2.3 Multi-Agent Collaboration Framework

This section introduces our multi-agent collaboration framework ($\mathcal{A}$), which decomposes the MIML task into four sub-tasks and assigns them to four specialized role-playing LLMs agents: Task Manager, Data Engineer, Module Architect, and Model Trainer, denoted as $\{a_1, a_2, a_3, a_4\}$, respectively, as illustrated in Fig. 2. Each agent is responsible for a specific role, and together they collaborate iteratively to construct the final AI model. We utilize a set of prompts filled with the related context protocols to control the role and working logic of each agent. For simplicity, detailed instruction prompts for each agent are shown in the Appendix.

Category	Tool	Brief Description
PC-Level Interact	list_files	list all files in folders
	read_files	read a short-context file
	copy_files	copy files to a target dir
	write_files	write code into scripts
	edit_files	edit certain rows in files
	run_script	run scripts in command line
	preview_dirs	overview complex data folders
	preview_files	screen meta data in large files
Medical Imaging Specific	load_med_data	load most medical format data
	check_3D	check 3D format image
	normalize_image	clip windows and normalize
	verify_report	verify report format validation
	augment_image	apply medical-specific augment

Table 1: PC-level interaction and medical imaging specific tools designed for M^{3}Builder. Gray : common tools; Yellow : tools for large files and complex data structures; Blue : tools tailored for medical imaging

Task Manager. As the coordinator of the framework, its primary responsibilities include selecting the most suitable dataset for the task, or alternatively, asking users to upload raw datasets with associated datacard following the context protocol as a supplement to pre-existing datasets, and generating a comprehensive MIML planning document $\mathcal{P}$ to guide the collaboration among the other agents. Specifically, given a user-provided task description, as exemplified by "user requirements" in Fig. 2, the Task Manager will identify and select the optimal dataset ($\mathcal{D}$) for model training and generate the planning documents. This process can be formally represented as:

$$\{\mathcal{D}, \mathcal{P}\} = a_1(\mathcal{T} \mid \mathbb{P}_d), \qquad (2)$$

where $\mathbb{P}_d$ denotes the data card in the pre-defined context protocol.

Data Engineer. Responsible for dataset preparation and processing. It transforms raw data into a format suitable for model training by performing tasks such as pre-screening the organizational structure of large-scale datasets, analyzing metadata files to extract relevant information, and splitting datasets into training and testing subsets. Data

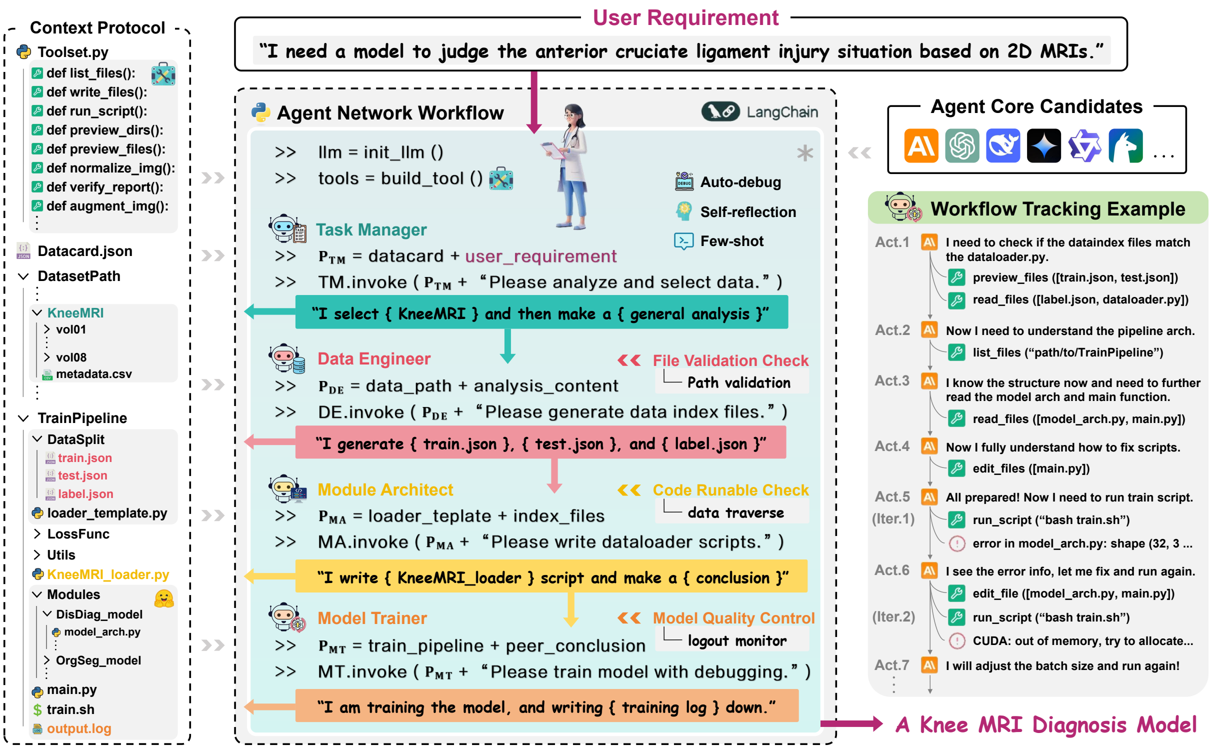

Context Protocol
Toolset.py
def list_files():
def write_files():
def run_script():
def preview_dirs():
def preview_files():
def normalize_img():
def verify_report():
def augment_img():
Datacard.json
DatasetPath
KneeMRI
vol01
vol08
metadata.csv
TrainPipeline
DataSplit
train.json
test.json
label.json
loader_template.py
LossFunc
Utils
KneeMRI_loader.py
Modules
DisDiag_model
model_arch.py
OrgSeg_model
main.py
train.sh
output.log
User Requirement
"I need a model to judge the anterior cruciate ligament injury situation based on 2D MRIs."
Agent Network Workflow
LangChain
>> llm = init_llm ()
>> tools = build_tool ()
Auto-debug
Self-reflection
Few-shot
Task Manager
>> P_TM = datacard + user_requirement
>> TM.invoke (P_TM + "Please analyze and select data.")
"I select { KneeMRI } and then make a { general analysis }"
Data Engineer
File Validation Check
Path validation
>> P_DE = data_path + analysis_content
>> DE.invoke (P_DE + "Please generate data index files.")
"I generate { train.json }, { test.json }, and { label.json }"
Module Architect
Code Runable Check
data traverse
>> P_MA = loader_teplate + index_files
>> MA.invoke (P_MA + "Please write dataloader scripts.")
"I write { KneeMRI_loader } script and make a { conclusion }"
Model Trainer
Model Quality Control
logout monitor
>> P_MT = train_pipeline + peer_conclusion
>> MT.invoke (P_MT + "Please train model with debugging.")
"I am training the model, and writing { training log } down."
Agent Core Candidates
...
Workflow Tracking Example
Act.1 I need to check if the dataindex files match the dataloader.py.
preview_files ([train.json, test.json])
read_files ([label.json, dataloader.py])
Act.2 Now I need to understand the pipeline arch.
list_files ("path/to/TrainPipeline")
Act.3 I know the structure now and need to further read the model arch and main function.
read_files ([model_arch.py, main.py])
Act.4 Now I fully understand how to fix scripts.
edit_files ([main.py])
Act.5 (Iter.1) All prepared! Now I need to run train script.
run_script ("bash train.sh")
error in model_arch.py: shape (32, 3 ...
Act.6 I see the error info, let me fix and run again.
edit_file ([model_arch.py, main.py])
(Iter.2) run_script ("bash train.sh")
CUDA: out of memory, try to allocate...
Act.7 I will adjust the batch size and run again!
A Knee MRI Diagnosis Model

Figure 2: Visualization of M^{3}Builder workflow: From user's free-text request to model delivery. The system integrates user requirements, a context protocol with candidate data, tools, and code templates, a network of four specialized collaborative agents. A sample log tracks the Model Trainer agent's activities during diagnosis model development.

Engineer iteratively interact with the external compiler environment to generate, edit, and refine code, incorporating compiler feedback until successful execution. This iterative process ensures the dataset preparation code is both robust and functional. The process can be expressed as:

$$\{\mathcal{C}_i, \mathcal{R}_i\} = a_2(\mathcal{C}_{i-1}, \mathcal{R}_{i-1}, \mathcal{T} \mid \mathcal{D}, \mathcal{P}), \quad (3)$$

where $\mathcal{C}_i$ represents the i^{th} version of the code generated by the Data Engineer, and $\mathcal{R}_i$ denotes the compiler feedback from the `Python` environment. Similarly as defined in **Problem Formulation**, $\mathcal{C}_0 = \mathcal{R}_0 = \varnothing$.

Module Architect. Upon the dataset preparation, the Module Architect integrates essential components into the training pipeline, including developing dataloader scripts, designing appropriate model architecture and selecting other components, such as loss functions and training utilities. Notably, during dataloader development, Module Architect selects a sequence of medical imaging specific tools to enable dataset-curated data verification and preprocessing. Module Architect will iteratively validate the dataloader to ensure it outputs batches with correct shapes and formats. The process can be formulated as:

$$\{\mathcal{C}_i, \mathcal{R}_i\} = a_3(\mathcal{C}_{i-1}, \mathcal{R}_{i-1}, \mathcal{T} \mid \mathcal{D}, \mathbb{P}_c), \quad (4)$$

where $\mathbb{P}_c$ represents the code templates in the context protocol. Similar to the Data Engineer, the code generation process in the Module Architect is also iterative and we denote the final iteration output as $\mathcal{C}_{\mathrm{MA}}$. After the integration of these modules, the architect finally synthesizes a summary $\mathcal{S}$ of all completed work.

Model Trainer. It finalizes the debugging and optimizing the training procedure. Building upon the pipeline established by the Module Architect, the Model Trainer first verifies the completeness and correctness of the training framework thus far. It then selects hyperparameters to meet the model's specific training requirements, while retaining the authority to modify any part of the code—including model code, dataloader code, and training scripts—based on errors encountered during training. Workflow can be expressed as:

$$\{\mathcal{C}_i, \mathcal{R}_i\} = a_4(\mathcal{C}_{i-1}, \mathcal{R}_{i-1}, \mathcal{T} \mid \mathcal{D}, \mathcal{C}_{\mathrm{MA}}, \mathcal{S}). \quad (5)$$

After iteratively performing the above code generation pipeline until successfully executed, the final code will produce a desired AI model.

3 M^{3}Bench

To thoroughly evaluate the performance of M^{3}Builder, we introduce M^{3}Bench, a benchmark comprising 14 datasets across 4 key medical imaging tasks. This benchmark includes ablation studies on 7 leading large language models (LLMs) and incorporates comparisons with other concurrent AutoML frameworks under fair settings.

3.1 Task Inclusion & Data Preparation

In this paper, we experiment with 4 typical medical imaging tasks spanning *organ segmentation*, *anomaly detection*, *disease diagnosis*, and *report generation*. These tasks are systematically categorized by anatomic regions (head & neck, chest, abdomen & pelvis, limb, spine), imaging modalities (X-ray, CT, MRI), and dimensionality (2D/3D). Each task is precisely defined, for instance: "please build a model for covid-19 pneumonia classification from 3D chest CT images."

We collect 14 medical imaging datasets: ADNI (Jack et al., 2008), KneeMRI (Štajduhar et al., 2017), CC-CCII (Zhang et al., 2020a), CT-Kidney (Žukovec et al., 2021), BTCV (Landman et al., 2015), MSD Pancreas (Antonelli et al., 2022), VerSe (Sekuboyina et al., 2021; Löffler et al., 2020; Liebl et al., 2021), L2R-OASIS (Marcus et al., 2007), COVID-19 (Ma et al., 2021), CT-RATE (Hamamci et al., 2024a), INSTANCE2022 (Li et al., 2023), ChestX-Det10 (Liu et al., 2020), RadGenome-Brain-MRI (Lei et al., 2024), and IU-Xray (Demner-Fushman et al., 2016). Each dataset is accompanied by a datacard with metadata from their original publications.

3.2 Agent Core Selection

As the core of our agentic system, LLM must have basic function-calling, coding skills, and ideally medical knowledge to support the implementation of four agents. As shown in Tab.2, medical-specific LLMs like BaichuanMed(Lijun Liu, 2025) excel in medical knowledge but lack function-calling and coding abilities, making them unsuitable for autonomous MIML tasks. Therefore, they are excluded from our benchmark.

In this context, given the critical importance of function calling and coding abilities for our task, we include 7 coding LLMs in M^{3}Bench to demonstrate their impact on different agent core selections. These models are GPT-4o (Hurst et al., 2024), Claude-3.7-Sonnet (Anthropic), Claude-3.5-Sonnet (Anthropic), DeepSeek-v3 (Liu et al., 2024a), Gemini-2.0-flash (Team et al., 2023), Qwen-2.5-max (Yang et al., 2024), and Llama-3.3-70B (Dubey et al., 2024).

LLM	Core Considerations				
	Medical Specification	Function Calling	Coding Expertise	Reasoning Free	Open Source
BaichuanMed	✓	✗	✗	✓	✓
DeepSeek-V3	✗	✓	✓	✓	✓
Sonnet-3.7	✗	✓	✓	✗	✗
OpenAI-o3	✗	✓	✓	✓	✗

Table 2: Core Considerations of four representative LLMs across medical and general domains, including function calling ability, coding proficiency, reasoning approach, and open or closed source status.

Framework	Key Features			
	Medical Adaptation	Automated Correction	Multi Agent	Open Source
MLABench	✗	✗	✗	✓
Aider	✗	✓	✗	✓
MetaGPT	✗	✓	✓	✓
ToolMaker	✓	✓	✗	✓
Copilot Edits	✗	✓	✗	✗
M^{3}Builder (Ours)	✓	✓	✓	✓

Table 3: Key features of frameworks across core features, including medical adaptation, automated correction, multi-agent capability, and open source status.

3.3 SoTA Comparison

In order to investigate the superiorty of our proposed agentic system, we perform a comparative analysis of five representative SoTA agentic coding frameworks as summarized in Table **??**: MLAgentBench, Aider, MetaGPT, ToolMaker, and Copilot Edits. MLAgentBench provides an agentic pipeline for evaluating large language models on Kaggle problem solving, while Aider offers an open-source environment for automating code generation. MetaGPT employs a multi-agent architecture to decompose and coordinate complex workflows, and ToolMaker converts "papers with code" into LLM-compatible tools with demonstrated efficacy in medical domains. Copilot Edits integrates an agent mode within a coding assistant to support mostly automated, end-to-end task execution.

3.4 Evaluation Metrics

We focus on the task completion rate (success rate) of MIML tasks, calculated as the proportion of successful completion out of N independent attempts.

Task	Dataset	Baseline Method
OrgSeg	BTCV	SAT-Nano(Zhao et al., 2023)
	VerSe	SAT-Nano(Zhao et al., 2023)
	OASIS	U-Mamba(Ma et al., 2024)
AnoDet	COVID19	UMamba(Ma et al., 2024)
	INSTANCE2022	SAT-Nano(Zhao et al., 2023)
	MSD Pancreas	SAT-Nano(Zhao et al., 2023)
	ChestX-Det10	nnUnet(Isensee et al., 2021)
DisDiag	ADNI	VoxResNet(Chen et al., 2018)
	KneeMRI	RP3D(Zheng et al., 2024)
	CC-CCII	3DResNet(Zhang et al., 2020b)
	CT-Kidney	RP3D(Zheng et al., 2024)
RepGen	CT-RATE	CT2Rep(Hamamci et al., 2024b)
	BrainGnome	AutoRG-Brain(Lei et al., 2024)
	IU-Xray	KiUT(Huang et al., 2023b)

Table 4: Baseline Methods of 4 medical imaging tasks on 14 Datasets

Here, a “completion” means a successful training that achieves task-wise metrics within 5 percent below existing baselines. For Organ Segmentation and Anomaly Detection, Dice Similarity Coefficient is employed as the main metric. Specifically for ChestX-Det10 dataset, which utilizes bounding box annotations, we apply mean Average Precision at 0.4 IoU threshold (mAP@0.4). Disease Diagnosis is assessed with AUROC, while Report Generation is evaluated using BLEU-4. Baseline methods of corresponding datasets are listed in Tab. ??

For other compared agentic baselines, we find most of them can not complete complex MIML tasks during our experiments due to large-scale PC-level interaction and data-level precise alignment. Therefore, we grant them access to our proposed context protocol for template referencing and dataset initialization. Under this setup, we employ Claude-3.5-Sonnet as the agent core with the limit of 100 action iterations (tool invocations) to ensure experimental fairness.

4 Results and Analysis

4.1 Agent Core Selection

We benchmarked seven LLMs as the agentic core on MIML tasks, running each LLM five times to calculate completion rates (Tab. ??). Performance varied significantly: Sonnet series and GPT-4o achieved the highest completion rates (94.29%, 90.00%, and 81.43%), while Gemini-2.0-flash and Llama-3.3-70b only reached 2.86% and 4.29%. This highlights the importance of code proficiency, well-trained function-calling capabilities, and sufficient model capacity for serving as agent cores. Further analysis shows weaker models misinterpreted tool descriptions, leading to errors, hallucinations, and incomplete step recognition.

Fig. 3 shows detailed performance of Sonnet-3.7, the best agentic core, across 14 tasks. Results show that when our agentic system executes MIML codes successfully, most automatically trained models match or even surpass manually trained baselines in performance, demonstrating robustness and applicability. This highlights that the primary challenge in MIML lies in coherently organizing the coding process rather than refining model performance.

4.2 Ablation Study

To quantify the influence of our proposed agentic component or mechanism, we carry out thorough ablation studies on the proposed agentic architecture. As shown in Tab ??, we show the impact of core mechanisms, including multi-agent collaboration, auto-debugging (self-correction when an error occurs), self-reflection (thinking twice before moving into the next step), and few-shot learning (adding a well-crafted example in the prompt for instruction), on success rate and action cost. The results indicate that self-reflection has minimal influence on our system performance, while auto-debugging proves crucial for successful training, indicating the LLMs can effectively interact with the compiler environment. Multi-agent collaboration and few-shot learning also significantly impact performance, with their absence resulting in 42.85% and 25.00% performance drop, respectively. These ablation results demonstrate the necessity and superiority of the proposed agentic mechanisms.

Tab. 7 presents the fine-grained analysis of per-agent performance, tracking the number of function-calling steps and debugging iterations, and token costs. A role agent succeeds if it flawlessly completes its final objective (e.g. the Task Manager selects suitable datasets, the Data Engineer generates valid data index files, the Module Architect produces executable scripts for data loading, and the Model Trainer completes model training), assuming upstream agents perform correctly.

The Task Manager demonstrated exceptional accuracy in task analysis, with stable token usage across all tasks. In contrast, the other three agent roles show greater variability in token consumption and execution attempts. This variability stems from the strict requirements for code organization, data preprocessing, and achieving error-free training within five iterations. This finding highlights that

Task	Dataset	Anatomy	Modality	Dimension	LLMs						
					Son.3.7	Son.3.5	GPT4o	QwenMax	DeepSeekV3	Llama3.3	Gemini2
OrgSeg	BTCV	Abdom & Pelvis	CT	3D	5/5	4/5	5/5	3/5	2/5	1/5	0/5
	Verse	Spine	CT	3D	4/5	4/5	5/5	2/5	0/5	0/5	0/5
	OASIS	Head & Neck	MRI	3D	4/5	3/5	3/5	1/5	1/5	0/5	0/5
AnoDet	COV19	Chest	CT	3D	5/5	5/5	4/5	3/5	1/5	0/5	1/5
	INS22	Head & Neck	CT	3D	5/5	5/5	4/5	2/5	2/5	0/5	0/5
	Panc	Abdom & Pelvis	CT	3D	5/5	5/5	5/5	4/5	2/5	1/5	0/5
	XDet10	Chest	X-ray	2D	5/5	5/5	5/5	3/5	1/5	0/5	1/5
DisDiag	ADNI	Head & Neck	MRI	3D	5/5	4/5	2/5	1/5	1/5	0/5	0/5
	KneeMR	Limb	MRI	2D	4/5	5/5	4/5	2/5	1/5	0/5	0/5
	CC-CCII	Chest	CT	3D	5/5	5/5	5/5	0/5	0/5	0/5	0/5
	KidCT	Abdom & Pelvis	CT	2D	5/5	5/5	4/5	2/5	1/5	0/5	0/5
RepGen	CT-RATE	Chest	CT	3D	5/5	4/5	4/5	2/5	1/5	0/5	0/5
	GenomBra	Head & Neck	MRI	3D	5/5	5/5	4/5	3/5	1/5	1/5	0/5
	IU-Xray	Chest	X-ray	2D	4/5	4/5	3/5	1/5	2/5	0/5	0/5
		Average(%)			**94.29**	90.00	81.43	41.43	22.86	4.29	2.86

Table 5: Task Completion Performance Across LLMs. Each experiment undergoes multi-runs, with results shown as successful completions over total rounds (a/b format). Green cells indicate that all runs passed, Yellow indicates partially passed, and Red indicates that all runs failed. OrgSeg-*Organ Segmentation*, AnoDet-*Anomaly Detection*, DisDiag-*Disease Diagnosis*, RepGen-*Report Generation*.

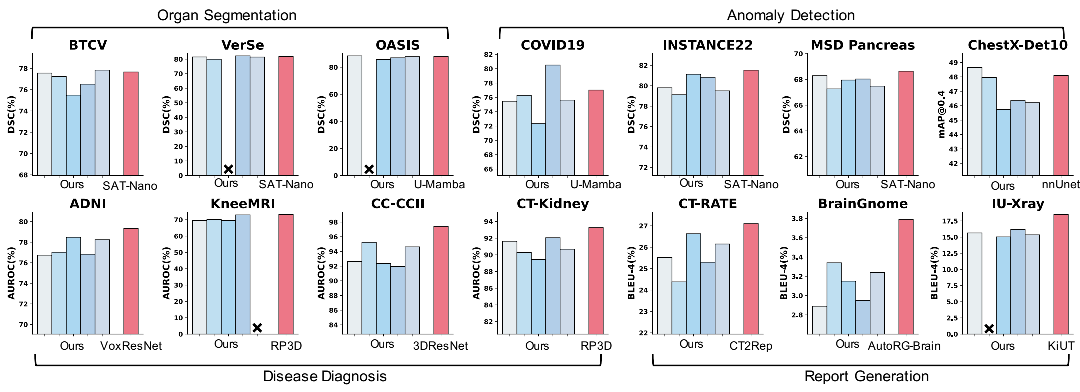


Figure 3: Training Performance Across 14 Datasets. M^{3}Builder is evaluated with 5 replicate experiments on each dataset by Sonnet, with performance compared against respective baseline methods. Failed trial is denoted with ✗.

for automating MIML, coding edits, particularly the self-debugging process, are the primary bottlenecks influencing success rates and computational costs in our pipeline. Despite these challenges, most agents successfully completed their assigned tasks, demonstrating the robustness and adaptability of the multi-agent framework. Detailed mechanism implementations are listed in Appendix. A.6.

4.3 SoTA Comparison

We evaluate eight SoTA agentic approaches uniformly using Sonnet-3.5 as the agent core, executing each training dataset twice, resulting in 6, 8, 8, and 6 runs for the segmentation, detection, diagnosis, and generation tasks, respectively. M^{3}Builder completes 23 runs with acceptable performance, outperforming all competitors. As shown in Tab **??**, MLA-Bench often fails on large medical meta data file preprocessing; Aider and MetaGPT can sometimes finish but underperform (e.g., only 50% AUC in binary diagnosis due to label–image mismatches); ToolMaker lacks medical domain-specific design and frequently fails during preprocessing or training; and the remaining agentic assistants require human-in-the-loop and are prone to deadlocks (such as misconfigured Conda environments or infinite data-loading-error-debuggin loops). Overall, M^{3}Builder achieves a 42.85% higher average success rate with fewer action steps and execution iterations, demonstrating the superiority of our agentic method. Detailed prompts are listed in Appendix. A.7.

5 Related Work

5.1 LLM-Powered Agentic Systems

Large Language Models (LLMs) have demonstrated remarkable capabilities. Recent work has en-powered LLMs with abilities for planning, reasoning and tool calling, enabling LLMs to function as agents(Plaat et al., 2025), addressing a spec-

Agentic System	Completion Runs (Total) ↑					Average Actions (Iters) ↓			
	Seg.	Det.	Diag.	Gen.	Avg(%)	Seg.	Det.	Diag.	Gen.
MLA-Bench (Huang et al., 2023a)	0(6)	0(8)	0(8)	0(6)	0.00	-(-)	-(-)	-(-)	-(-)
Aider (Gauthier, 2023)	2(6)	3(8)	2(8)	3(6)	35.71	58.0(6.5)	51.3(7.0)	44.5(4.5)	56.7(5.7)
MetaGPT (Hong et al., 2024)	1(6)	2(8)	3(8)	1(6)	17.86	49(5)	37.5(3.5)	37(4.67)	40(4)
ToolMaker (Wölflein et al., 2025)	1(6)	1(8)	4(8)	2(6)	28.57	41(4)	44(3)	37.75(3.75)	46(4.5)
Cursor Comp (Team, 2023)	1(6)	3(8)	3(8)	2(6)	32.14	42.0(7.0)	35.7(4.3)	36.3(4.0)	51.5(5.5)
Wsurf Casc (Codeium, 2024)	1(6)	3(8)	4(8)	2(6)	35.71	39.0(5.0)	36.0(4.3)	35.3(4.8)	48.5(5.5)
Copilot Edits (GitHub, 2023)	2(6)	4(8)	3(8)	2(6)	39.29	48.0(3.5)	45.3(3.5)	44.7(4.0)	46.5(4.5)
M^{3}Builder (Ours)	4(6)	7(8)	8(8)	4(6)	82.14	34.5(1.8)	25.29(2.4)	35.0(1.9)	32.0(2.3)
w/o Colab	3(6)	3(8)	3(8)	2(6)	39.29	33.6(4.7)	37.5(4.8)	33.3(4.3)	33.5(3.5)
w/o Debug	2(6)	3(8)	1(8)	0(6)	21.43	30.5(1.0)	24.0(1.0)	33.0(1.0)	-(-)
w/o Reflect	4(6)	7(8)	7(8)	4(6)	78.57	28.3(2.3)	24.9(4.1)	35.4(3.9)	41.3(5.8)
w/o Fewshot	3(6)	4(8)	6(8)	3(6)	57.14	37.3(4.0)	23.5(4.3)	32.3(4.1)	34.0(3.0)

Table 6: Framework Comparison with SOTAs and Ablations on System Design using Sonnet. Results are averaged over two runs per task in dataset-level. "w/o Colab" represents single-agent execution, and "Iters" means the self-correction rounds.

Task	Task Manager				Data Engineer				Module Architect				Model Trainer			
	Run	Act	Iter	Tkn	Run	Act	Iter	Tkn	Run	Act	Iter	Tkn	Run	Act	Iter	Tkn
Seg.	6/6	1.3	1.0	4.3k	5/6	10.2	1.3	72k	5/6	10.5	2.3	74k	4/6	10.7	3.3	115k
Det.	8/8	2.0	1.0	4.8k	8/8	10.0	1.4	92k	7/8	10.6	2.1	61k	7/8	9.3	2.1	67k
Diag.	8/8	2.0	1.0	4.4k	8/8	8.9	1.4	116k	7/8	11.3	2.0	84k	8/8	9.3	2.0	198k
Gen.	6/6	2.0	1.0	4.2k	6/6	8.7	1.3	66k	5/6	10.8	2.5	91k	5/6	11.2	2.2	70k

Table 7: Role-specific agent performance on tasks using Sonnet. "Run", "Act", "Iter" and "Tkn" respectively denote "execution rounds", "actions", "iterations" and "tokens".

trum of complex tasks, such as planning tasks in PC environments(Wang et al., 2024; Agashe et al., 2024), software engineering(Dong et al., 2024; Liu et al., 2024b), and scientific discovery(Boiko et al., 2023; Lu et al., 2024). Agents can be integrated into Multi-Agent Systems(MAS) where multiple agents collaborate through specialized roles to address more complex tasks(Talebirad and Nadiri, 2023). Leveraging tool calling capabilities of LLMs(Schick et al., 2023), agentic systems can interact with environment to perform long-chain reasoning and planning, and utilize external knowledge to achieve real-time in-context learning while reducing hallucinations.

5.2 Automatic Machine Learning

Automatic Machine Learning (AutoML) automates the selection of modules and hyper-parameters in machine learning pipeline, making ML accessible to users without expertise(Baratchi et al., 2024). Previous work focus on two main aspects: **(i)** Model Selection (MS), which involves identifying the best-performing machine learning model for a dataset from a predefined model set(Thornton et al., 2013; Liu et al., 2018; Feurer et al., 2020), and **(ii)** Hyper-Parameter Optimization (HPO), the process of searching the hyper-parameter space to determine optimal values that enhance the selected model's performance(Snoek et al., 2014; Pfisterer et al., 2021; Lindauer et al., 2022). Recent studies also consider combination of MS and HPO(LeDell and Poirier, 2020; Zimmer et al., 2021).

6 Conclusion

In this paper, we present M^{3}Builder, an agentic system for automating medical imaging machine learning (MIML) tasks. Our approach combines an efficient medical imaging ML context protocol formatting the free-text descriptions of MIML datasets, code templates, and interaction tools. Additionally, we propose a multi-agent collaborative agent system designed specifically for MIML training, with four role-playing LLMs, Task Manager, Data Engineer, Module Architect, and Model Trainer. In benchmarking against seven SOTA agentic systems across 14 radiology task-specific datasets, M^{3}Builder achieves a 94.29% model building completion rate. In agent core selection, we find Claude-3.7-Sonnet standing out among seven SOTA LLMs. By integrating M^{3}Builder with other clinical agent workflows, we can empower medical agentic systems with self-evolving capabilities that automatically expand their toolsets, greatly reducing the extensive labor required to manually prepare diverse MIML tools.

Limitations

Although our proposed framework, **M^{3}Builder**, has performed well across various medical imaging ML tasks, there are still some limitations. First, while **M^{3}Builder** shows remarkable generalizability across diverse medical tasks and imaging modalities, the task set could be further extended to more anatomical regions and imaging modalities. Besides, due to the scalability of our framework, it could be further extended to more medical ML tasks beyond medical imaging scenarios. Future work will extend beyond medical imaging to broader medical tasks, develop more robust tool-building agent systems, implement automated dataset preparation capabilities, and incorporate visual processing to better approximate clinical expertise.

References

Saaket Agashe, Jiuzhou Han, Shuyu Gan, Jiachen Yang, Ang Li, and Xin Eric Wang. 2024. Agent s: An open agentic framework that uses computers like a human. *arXiv preprint arXiv:2410.08164*.

Anthropic. The claude 3 model family: Opus, sonnet, haiku.

Michela Antonelli, Annika Reinke, Spyridon Bakas, Keyvan Farahani, Annette Kopp-Schneider, Bennett A Landman, Geert Litjens, Bjoern Menze, Olaf Ronneberger, Ronald M Summers, and 1 others. 2022. The medical segmentation decathlon. *Nature communications*, 13(1):4128.

Mitra Baratchi, Can Wang, Steffen Limmer, Jan N van Rijn, Holger Hoos, Thomas Bäck, and Markus Olhofer. 2024. Automated machine learning: past, present and future. *Artificial intelligence review*, 57(5):122.

Daniil A Boiko, Robert MacKnight, Ben Kline, and Gabe Gomes. 2023. Autonomous chemical research with large language models. *Nature*, 624(7992):570–578.

Hao Chen, Qi Dou, Lequan Yu, Jing Qin, and Pheng-Ann Heng. 2018. Voxresnet: Deep voxelwise residual networks for brain segmentation from 3d mr images. *NeuroImage*, 170:446–455.

Codeium. 2024. Windsurf cascade. https://docs.codeium.com/windsurf/cascade.

Dina Demner-Fushman, Marc D Kohli, Marc B Rosenman, Sonya E Shooshan, Laritza Rodriguez, Sameer Antani, George R Thoma, and Clement J McDonald. 2016. Preparing a collection of radiology examinations for distribution and retrieval. *Journal of the American Medical Informatics Association*, 23(2):304–310.

Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2024. Self-collaboration code generation via chatgpt. *ACM Transactions on Software Engineering and Methodology*, 33(7):1–38.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, and 1 others. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Matthias Feurer, Katharina Eggensperger, Stefan Falkner, Marius Lindauer, and Frank Hutter. 2020. Auto-sklearn 2.0: The next generation. *arXiv preprint arXiv:2007.04074*, 24(8).

Paul Gauthier. 2023. Aider is ai pair programming in your terminal. https://github.com/paul-gauthier/aider.

GitHub. 2023. Copilot edits. https://code.visualstudio.com/docs/copilot/copilot-edits.

Ibrahim Ethem Hamamci, Sezgin Er, Furkan Almas, Ayse Gulnihan Simsek, Sevval Nil Esirgun, Irem Dogan, Muhammed Furkan Dasdelen, Bastian Wittmann, Enis Simsar, Mehmet Simsar, and 1 others. 2024a. A foundation model utilizing chest ct volumes and radiology reports for supervised-level zero-shot detection of abnormalities. *CoRR*.

Ibrahim Ethem Hamamci, Sezgin Er, and Bjoern Menze. 2024b. Ct2rep: Automated radiology report generation for 3d medical imaging. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 476–486. Springer.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2024. MetaGPT: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*.

Qian Huang, Jian Vora, Percy Liang, and Jure Leskovec. 2023a. Mlagentbench: Evaluating language agents on machine learning experimentation. *arXiv preprint arXiv:2310.03302*.

Zhongzhen Huang, Xiaofan Zhang, and Shaoting Zhang. 2023b. Kiut: Knowledge-injected u-transformer for radiology report generation. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 19809–19818.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, and 1 others. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*.

Fabian Isensee, Paul F Jaeger, Simon AA Kohl, Jens Petersen, and Klaus H Maier-Hein. 2021. nnu-net: a self-configuring method for deep learning-based biomedical image segmentation. *Nature methods*, 18(2):203–211.

Clifford R. Jack, Matt A. Bernstein, Nick C Fox, Paul M. Thompson, Gene E. Alexander, Danielle J. Harvey, Bret J. Borowski, Paula J. Britson, Jennifer L Whitwell, Chadwick P. Ward, Anders M. Dale, Joel P. Felmlee, Jeffrey L. Gunter, Derek L. G. Hill, Ronald J. Killiany, Norbert Schuff, Sabrina Fox-Bosetti, Chen Lin, Colin Studholme, and 16 others. 2008. The alzheimer's disease neuroimaging initiative (adni): Mri methods. *Journal of Magnetic Resonance Imaging*, 27.

Bennett Landman, Zhoubing Xu, Juan Igelsias, Martin Styner, Thomas Langerak, and Arno Klein. 2015. Miccai multi-atlas labeling beyond the cranial vault–workshop and challenge. In *Proc. MICCAI multi-atlas labeling beyond cranial vault—workshop challenge*, volume 5, page 12. Munich, Germany.

Erin LeDell and Sebastien Poirier. 2020. H2o automl: Scalable automatic machine learning. In *Proceedings of the AutoML Workshop at ICML*, volume 2020, page 24.

Jiayu Lei, Xiaoman Zhang, Chaoyi Wu, Lisong Dai, Ya Zhang, Yanyong Zhang, Yanfeng Wang, Weidi Xie, and Yuehua Li. 2024. Autorg-brain: Grounded report generation for brain mri. *arXiv preprint arXiv:2407.16684*.

Xiangyu Li, Gongning Luo, Kuanquan Wang, Hongyu Wang, Jun Liu, Xinjie Liang, Jie Jiang, Zhenghao Song, Chunyue Zheng, Haokai Chi, and 1 others. 2023. The state-of-the-art 3d anisotropic intracranial hemorrhage segmentation on non-contrast head ct: The instance challenge. *arXiv preprint arXiv:2301.03281*.

Hans Liebl, David Schinz, Anjany Sekuboyina, Luca Malagutti, Maximilian T Löffler, Amirhossein Bayat, Malek El Husseini, Giles Tetteh, Katharina Grau, Eva Niederreiter, and 1 others. 2021. A computed tomography vertebral segmentation dataset with anatomical variations and multi-vendor scanner data. *Scientific data*, 8(1):284.

Tao Zhang Chong Li Mingrui Wang Chenglin Zhu Mingan Lin Zenan Zhou Weipeng Chen Lijun Liu, Tao Zhang. 2025. Baichuanmed-ocr-72b: A powerful medical report ocr recognition model.

Marius Lindauer, Katharina Eggensperger, Matthias Feurer, André Biedenkapp, Difan Deng, Carolin Benjamins, Tim Ruhkopf, René Sass, and Frank Hutter. 2022. Smac3: A versatile bayesian optimization package for hyperparameter optimization. *Journal of Machine Learning Research*, 23(54):1–9.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, and 1 others. 2024a. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*.

Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. 2018. Progressive neural architecture search. In *Proceedings of the European conference on computer vision (ECCV)*, pages 19–34.

Jingyu Liu, Jie Lian, and Yizhou Yu. 2020. Chestx-det10: Chest x-ray dataset on detection of thoracic abnormalities. *Preprint*, arXiv:2006.10550v3.

Junwei Liu, Yixuan Chen, Mingwei Liu, Xin Peng, and Yiling Lou. 2024b. Stall+: Boosting llm-based repository-level code completion with static analysis. *arXiv preprint arXiv:2406.10018*.

Maximilian T Löffler, Anjany Sekuboyina, Alina Jacob, Anna-Lena Grau, Andreas Scharr, Malek El Husseini, Mareike Kallweit, Claus Zimmer, Thomas Baum, and Jan S Kirschke. 2020. A vertebral segmentation dataset with fracture grading. *Radiology: Artificial Intelligence*, 2(4):e190138.

Chris Lu, Cong Lu, Robert Tjarko Lange, Jakob Foerster, Jeff Clune, and David Ha. 2024. The ai scientist: Towards fully automated open-ended scientific discovery. *arXiv preprint arXiv:2408.06292*.

Jun Ma, Feifei Li, and Bo Wang. 2024. U-mamba: Enhancing long-range dependency for biomedical image segmentation. *arXiv preprint arXiv:2401.04722*.

Jun Ma, Yixin Wang, Xingle An, Cheng Ge, Ziqi Yu, Jianan Chen, Qiongjie Zhu, Guoqiang Dong, Jian He, Zhiqiang He, and 1 others. 2021. Toward data-efficient learning: A benchmark for covid-19 ct lung and infection segmentation. *Medical physics*, 48(3):1197–1210.

Daniel S Marcus, Tracy H Wang, Jamie Parker, John G Csernansky, John C Morris, and Randy L Buckner. 2007. Open access series of imaging studies (oasis): cross-sectional mri data in young, middle aged, nondemented, and demented older adults. *Journal of cognitive neuroscience*, 19(9):1498–1507.

Florian Pfisterer, Jan N Van Rijn, Philipp Probst, Andreas C Müller, and Bernd Bischl. 2021. Learning multiple defaults for machine learning algorithms. In *Proceedings of the genetic and evolutionary computation conference companion*, pages 241–242.

Aske Plaat, Max van Duijn, Niki van Stein, Mike Preuss, Peter van der Putten, and Kees Joost Batenburg. 2025. Agentic large language models, a survey. *arXiv preprint arXiv:2503.23037*.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36:68539–68551.

Anjany Sekuboyina, Malek E Husseini, Amirhossein Bayat, Maximilian Löffler, Hans Liebl, Hongwei Li, Giles Tetteh, Jan Kukačka, Christian Payer, Darko Štern, and 1 others. 2021. Verse: a vertebrae labelling and segmentation benchmark for multi-detector ct images. *Medical image analysis*, 73:102166.

Jasper Snoek, Kevin Swersky, Rich Zemel, and Ryan Adams. 2014. Input warping for bayesian optimization of non-stationary functions. In *International conference on machine learning*, pages 1674–1682. PMLR.

Ivan Štajduhar, Mihaela Mamula, Damir Miletić, and Gözde Ünal. 2017. Semi-automated detection of anterior cruciate ligament injury from MRI. *Comput. Methods Programs Biomed.*, 140:151–164.

Yashar Talebirad and Amirhossein Nadiri. 2023. Multi-agent collaboration: Harnessing the power of intelligent llm agents. *arXiv preprint arXiv:2306.03314*.

Cursor Team. 2023. Cursor: The ai code editor. https://www.cursor.com/.

Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, and 1 others. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*.

Chris Thornton, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. 2013. Auto-weka: Combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 847–855.

Junyang Wang, Haiyang Xu, Jiabo Ye, Ming Yan, Weizhou Shen, Ji Zhang, Fei Huang, and Jitao Sang. 2024. Mobile-agent: Autonomous multi-modal mobile device agent with visual perception. *arXiv preprint arXiv:2401.16158*.

Georg Wölflein, Dyke Ferber, Daniel Truhn, Ognjen Arandjelović, and Jakob Nikolas Kather. 2025. LLM agents making agent tools. *Preprint*, arXiv:2502.11705.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, and 1 others. 2024. Qwen2. 5 technical report. *arXiv preprint arXiv:2412.15115*.

Kang Zhang, Xiaohong Liu, Jun Shen, Zhihuan Li, Ye Sang, Xingwang Wu, Yunfei Zha, Wenhua Liang, Chengdi Wang, Ke Wang, and 1 others. 2020a. Clinically applicable ai system for accurate diagnosis, quantitative measurements, and prognosis of covid-19 pneumonia using computed tomography. *Cell*, 181(6):1423–1433.

Kang Zhang, Xiaohong Liu, Jun Shen, Zhihuan Li, Ye Sang, Xingwang Wu, Yunfei Zha, Wenhua Liang, Chengdi Wang, Ke Wang, and 1 others. 2020b. Clinically applicable ai system for accurate diagnosis, quantitative measurements, and prognosis of covid-19 pneumonia using computed tomography. *Cell*, 181(6):1423–1433.

Ziheng Zhao, Yao Zhang, Chaoyi Wu, Xiaoman Zhang, Ya Zhang, Yanfeng Wang, and Weidi Xie. 2023. One model to rule them all: Towards universal segmentation for medical images with text prompts. *arXiv preprint arXiv:2312.17183*.

Qiaoyu Zheng, Weike Zhao, Chaoyi Wu, Xiaoman Zhang, Lisong Dai, Hengyu Guan, Yuehua Li, Ya Zhang, Yanfeng Wang, and Weidi Xie. 2024. Large-scale long-tailed disease diagnosis on radiology images. *Nature Communications*, 15(1):10147.

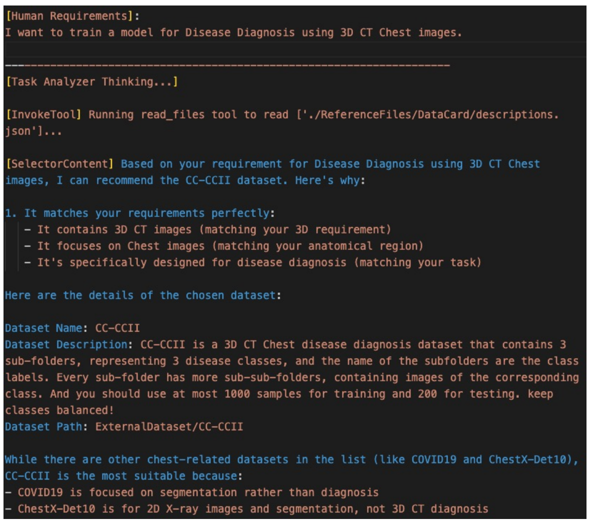
Lucas Zimmer, Marius Lindauer, and Frank Hutter. 2021. Auto-pytorch: Multi-fidelity metalearning for efficient and robust autodl. *IEEE transactions on pattern analysis and machine intelligence*, 43(9):3079–3090.

Martin Žukovec, Lara Dular, and Žiga Špiclin. 2021. Modeling multi-annotator uncertainty as multi-class segmentation problem. In *International MICCAI Brainlesion Workshop*, pages 112–123. Springer.

A Appendix

A.1 Case Study.

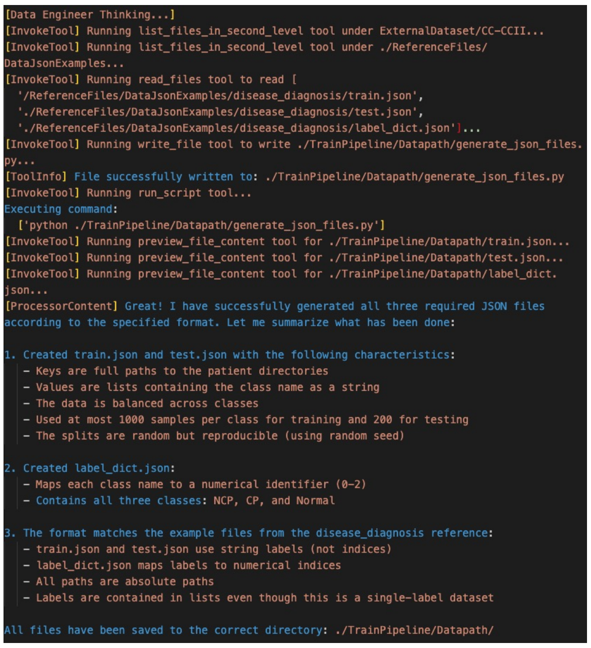
Here we present terminal logs of a successful running case. In this case, we require the system to train a model for Disease Diagnosis using 3D CT Chest image data. As shown in Fig. 4, Task Analyzer first read thoroughly through all the dataset descriptions, then chooses the CC-CCII dataset and list the reasons to justify the choice. Task Analyzer also returns the detail information of CC-CCII dataset for down-stream role agents.



```
[Human Requirements]:
I want to train a model for Disease Diagnosis using 3D CT Chest images.

---------------------------------------------------------------------
[Task Analyzer Thinking...]

[InvokeTool] Running read_files tool to read ['./ReferenceFiles/DataCard/descriptions.
json']...

[SelectorContent] Based on your requirement for Disease Diagnosis using 3D CT Chest
images, I can recommend the CC-CCII dataset. Here's why:

1. It matches your requirements perfectly:
   - It contains 3D CT images (matching your 3D requirement)
   - It focuses on Chest images (matching your anatomical region)
   - It's specifically designed for disease diagnosis (matching your task)

Here are the details of the chosen dataset:

Dataset Name: CC-CCII
Dataset Description: CC-CCII is a 3D CT Chest disease diagnosis dataset that contains 3
sub-folders, representing 3 disease classes, and the name of the subfolders are the class
labels. Every sub-folder has more sub-sub-folders, containing images of the corresponding
class. And you should use at most 1000 samples for training and 200 for testing. keep
classes balanced!
Dataset Path: ExternalDataset/CC-CCII

While there are other chest-related datasets in the list (like COVID19 and ChestX-Det10),
CC-CCII is the most suitable because:
- COVID19 is focused on segmentation rather than diagnosis
- ChestX-Det10 is for 2D X-ray images and segmentation, not 3D CT diagnosis
```

Figure 4: Case Study: Terminal log of Task Analyzer.

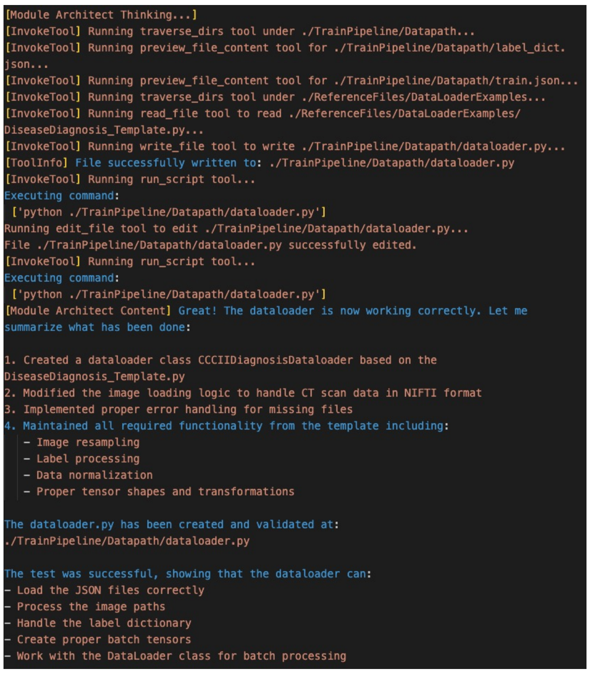
Following Task Analyzer, Data Engineer traverse the dataset files, splits the dataset into train/test sets. It refers to the example data index files, and generates a python script to generate the json files. The process is shown in Fig. 5. Finally, Data Engineer concludes its work, and shows a briefing to the user.



```
[Data Engineer Thinking...]
[InvokeTool] Running list_files_in_second_level tool under ExternalDataset/CC-CCII...
[InvokeTool] Running list_files_in_second_level tool under ./ReferenceFiles/
DataJsonExamples...
[InvokeTool] Running read_files tool to read [
  '/ReferenceFiles/DataJsonExamples/disease_diagnosis/train.json',
  './ReferenceFiles/DataJsonExamples/disease_diagnosis/test.json',
  './ReferenceFiles/DataJsonExamples/disease_diagnosis/label_dict.json']...
[InvokeTool] Running write_file tool to write ./TrainPipeline/Datapath/generate_json_files.
py...
[ToolInfo] File successfully written to: ./TrainPipeline/Datapath/generate_json_files.py
[InvokeTool] Running run_script tool...
Executing command:
  ['python ./TrainPipeline/Datapath/generate_json_files.py']
[InvokeTool] Running preview_file_content tool for ./TrainPipeline/Datapath/train.json...
[InvokeTool] Running preview_file_content tool for ./TrainPipeline/Datapath/test.json...
[InvokeTool] Running preview_file_content tool for ./TrainPipeline/Datapath/label_dict.
json...
[ProcessorContent] Great! I have successfully generated all three required JSON files
according to the specified format. Let me summarize what has been done:

1. Created train.json and test.json with the following characteristics:
   - Keys are full paths to the patient directories
   - Values are lists containing the class name as a string
   - The data is balanced across classes
   - Used at most 1000 samples per class for training and 200 for testing
   - The splits are random but reproducible (using random seed)

2. Created label_dict.json:
   - Maps each class name to a numerical identifier (0-2)
   - Contains all three classes: NCP, CP, and Normal

3. The format matches the example files from the disease_diagnosis reference:
   - train.json and test.json use string labels (not indices)
   - label_dict.json maps labels to numerical indices
   - All paths are absolute paths
   - Labels are contained in lists even though this is a single-label dataset

All files have been saved to the correct directory: ./TrainPipeline/Datapath/
```

Figure 5: Case Study: Terminal log of Data Engineer.

As shown in Fig. 6, Module Architect first checks that Data Engineer has generated all required files, then refers to the example dataloader scripts, and generates its own dataloader file, which guarantees standardized data processing and formatting. Before ending, it executes the generated dataloader file to check its correctness, by traversing and loading all the training and testing samples. Finally, it prints a briefing to conclude its work.



```
[Module Architect Thinking...]
[InvokeTool] Running traverse_dirs tool under ./TrainPipeline/Datapath...
[InvokeTool] Running preview_file_content tool for ./TrainPipeline/Datapath/label_dict.
json...
[InvokeTool] Running preview_file_content tool for ./TrainPipeline/Datapath/train.json...
[InvokeTool] Running traverse_dirs tool under ./ReferenceFiles/DataLoaderExamples...
[InvokeTool] Running read_file tool to read ./ReferenceFiles/DataLoaderExamples/
DiseaseDiagnosis_Template.py...
[InvokeTool] Running write_file tool to write ./TrainPipeline/Datapath/dataloader.py...
[ToolInfo] File successfully written to: ./TrainPipeline/Datapath/dataloader.py
[InvokeTool] Running run_script tool...
Executing command:
 ['python ./TrainPipeline/Datapath/dataloader.py']
Running edit_file tool to edit ./TrainPipeline/Datapath/dataloader.py...
File ./TrainPipeline/Datapath/dataloader.py successfully edited.
[InvokeTool] Running run_script tool...
Executing command:
 ['python ./TrainPipeline/Datapath/dataloader.py']
[Module Architect Content] Great! The dataloader is now working correctly. Let me
summarize what has been done:

1. Created a dataloader class CCCIIDiagnosisDataloader based on the
DiseaseDiagnosis_Template.py
2. Modified the image loading logic to handle CT scan data in NIFTI format
3. Implemented proper error handling for missing files
4. Maintained all required functionality from the template including:
   - Image resampling
   - Label processing
   - Data normalization
   - Proper tensor shapes and transformations

The dataloader.py has been created and validated at:
./TrainPipeline/Datapath/dataloader.py

The test was successful, showing that the dataloader can:
- Load the JSON files correctly
- Process the image paths
- Handle the label dictionary
- Create proper batch tensors
- Work with the DataLoader class for batch processing
```

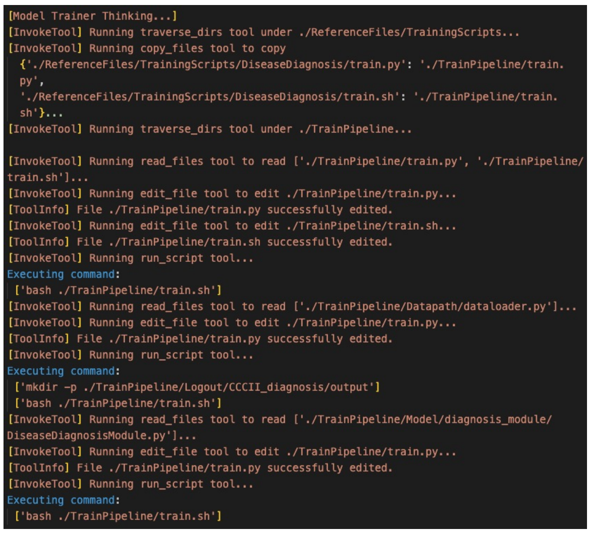
Figure 6: Case Study: Terminal log of Module Architect.

Finally, Fig. 7 shows the workflow of Model Trainer. It first choose a training code framework, and a ML module according to the medical task, then assemble the dataloader, the ML module and the training codes to form an executable training script. As we can see in Fig. 7, the first training attempts fails. After checking the trace-back information(not shown in the figure), it modifies the training scripts accordingly, and finally performs a successful run.

A.2 Details about Inclusion Datasets

We curated 14 datasets supporting different medical imaging tasks. Each dataset is unzipped in our context protocol without further processing. Each dataset is paired with a structured description in the datacard. Once a new dataset is uploaded into context protocol, the corresponding data description should be inserted into this dynamic datacard file. The initialization of the datacard is shown as bellow:

[{
"dataset name": "ADNI",
"dataset description": "ADNI is a 3D MRI disease diagnosis dataset that contains 1 subfolder and 1 csv file. the subfolder(named ADNI_PT) contains images(in .pt format), while the csv file contains lines of sample info, where you should focus mainly on 2 columns: 'Image Data ID' and 'Group'(which is the class label). Remember, only samples with these labels should be used: ['AD','MCI','CN']! And you should use at most 1000 samples for training and 200 for testing. keep classes balanced!",
"dataset path": "/path/to/ADNI_Dataset" },
{
"dataset name": "KneeMRI",
"dataset description": "KneeMRI is a 2D MRI disease diagnosis dataset that contains 1 subfolder and 1 csv file. the subfolder1 contains images(in .pck format), while the csv file contains lines of sample info, where you must



```
[Model Trainer Thinking...]
[InvokeTool] Running traverse_dirs tool under ./ReferenceFiles/TrainingScripts...
[InvokeTool] Running copy_files tool to copy
  {'./ReferenceFiles/TrainingScripts/DiseaseDiagnosis/train.py': './TrainPipeline/train.
  py',
  './ReferenceFiles/TrainingScripts/DiseaseDiagnosis/train.sh': './TrainPipeline/train.
  sh'}...
[InvokeTool] Running traverse_dirs tool under ./TrainPipeline...

[InvokeTool] Running read_files tool to read ['./TrainPipeline/train.py', './TrainPipeline/
train.sh']...
[InvokeTool] Running edit_file tool to edit ./TrainPipeline/train.py...
[ToolInfo] File ./TrainPipeline/train.py successfully edited.
[InvokeTool] Running edit_file tool to edit ./TrainPipeline/train.sh...
[ToolInfo] File ./TrainPipeline/train.sh successfully edited.
[InvokeTool] Running run_script tool...
Executing command:
 ['bash ./TrainPipeline/train.sh']
[InvokeTool] Running read_files tool to read ['./TrainPipeline/Datapath/dataloader.py']...
[InvokeTool] Running edit_file tool to edit ./TrainPipeline/train.py...
[ToolInfo] File ./TrainPipeline/train.py successfully edited.
[InvokeTool] Running run_script tool...
Executing command:
 ['mkdir -p ./TrainPipeline/Logout/CCCII_diagnosis/output']
 ['bash ./TrainPipeline/train.sh']
[InvokeTool] Running read_files tool to read ['./TrainPipeline/Model/diagnosis_module/
DiseaseDiagnosisModule.py']...
[InvokeTool] Running edit_file tool to edit ./TrainPipeline/train.py...
[ToolInfo] File ./TrainPipeline/train.py successfully edited.
[InvokeTool] Running run_script tool...
Executing command:
 ['bash ./TrainPipeline/train.sh']
```

Figure 7: Case Study: Terminal log of Model Trainer.

focus mainly on 2 colomns: 'volumeFilename' and 'aclDiagnosis'(which is the class label, could be 0 or 1 or 2). And you should use at most 1000 samples for training and 200 for testing. keep classes balanced!",
"dataset path": "/path/to/KneeMRI"
}, ...
{
"dataset name": "OASIS",
"dataset description":"OASIS is a 3D Head & Neck MRI organ segmentation dataset that contains 2 subfolders - namely images and masks - containing nii.gz format images and corresponding organ masks, and 1 json file: labels.json, containing an ordered label list (please do get a complete label_list from it).",
"dataset path": "/path/to/OASIS"
}]

A.3 Details about the Toolset

Here we present each tool function in the toolset:

- **list_files**: Recursively scans the specified directory to identify and return paths of all code files (supporting common programming extensions like .py, .java, .cpp, etc.). Features intelligent directory skipping by automatically ignoring directories containing more than 1,000 files to prevent processing excessively large file collections. Returns a newline-separated string of file paths for further processing.

- **read_files**: Opens and reads the entire content of a specified file, returning the complete text as a string. Supports UTF-8 encoding, making it suitable for examining source code, configuration files, or any text document. Essential for code analysis and file inspection tasks without modifying the original content.

- **copy_files**: Creates an exact duplicate of a single file from a source location to a destination path. Automatically generates any necessary directory structure at the destination if it doesn't already exist. Preserves file metadata like timestamps and permissions using shutil.copy2, ensuring the duplicate maintains the characteristics of the original file.

- **write_files**: Generates a new file with specified content at the designated file path. Automatically creates all necessary parent directories if they don't exist, ensuring the file can be written even to previously non-existent paths. Particularly useful for programmatically creating new script files, configuration files, or saving processed data.

- **edit_files**: Completely overwrites an existing file with new content, replacing the original data entirely. Designed for direct file modification without the need to manually open and edit files. Critical for automated code refactoring, text transformation, or updating configuration files with revised settings in batch operations.

- **edit_file**: Executes shell commands in the operating system environment and captures their output. Leverages the ShellTool from LangChain to safely run commands and collect results. Enables interaction with the system shell to perform operations like running programs, executing system utilities, or triggering external processes from within the application.

- **preview_dirs**: Performs a detailed analysis of a directory's structure by examining each immediate subfolder. For each entry, counts the total number of files and lists up to 100 file paths in natural sort order. Returns a structured dictionary with comprehensive information about directories and files, facilitating efficient navigation of complex file systems while limiting output size for large directories.

- **preview_files**: Provides intelligent content summaries of structured and unstructured data files. For CSV files, displays the first 5 rows and total row count; for JSON, shows the first 5 key-value pairs or elements and total count; for text files, presents the first 10,000 words and total word count. Enables rapid content assessment without loading entire large files into memory, particularly valuable for data exploration tasks.

- **load_med_data**: Addresses different data file formats commonly used in medical imaging tasks (e.g. png, nii, dcm, tif), and load them into tensors.
- **check_3D**: Examines spatial information of 3D format images(nii, dcm, tif). 3D format images, as well as their corresponding annotation masks, usually suffer spatial configuration inconsistency, such as dimension disorder, shape misalignment of images and masks. check_3D check these problems and guarantee consistent spatial configuration within a loading dataset.
- **normalize_image**: Performs commonly used medical image normalization to a loaded image. For a given medical imaging modality, there are typically some conventional normalization steps and parameter ranges that are commonly adopted (e.g. intensity clipping, unit normalization). normalize_image address this by defining a set of normalization sequence for each imaging modality.
- **verify_report**: Verifies the format and content of imaging reports, removes commonly seen text noise in the report, such as extra fields (beyond Findings and Impressions) and meaningless syntax (e.g. separators, unrecognizable characters introduces during OCR). Returns pure meaningful Findings and Impressions parts to guarantee clarity of text reports.
- **augment_image**: Provides image augmentation techniques for diverse imaging modalities. For each imaging modality, augment_image defines a sequence of modality-specific augmentation methods, in line with previous medical image machine learning work.

A.4 Details about the Agent structure

We use langgraph architecture for agent building and workflow graph compiling. As shown in Fig. 3, All the agents have their own toolsets, which are subset of our proposed toolset containing 8 tools because of their specification and meaningless redundant information provided. The function calling loop and debugging mechanism ensure the task completion performance.

A.5 Details about the Agent Core Candidates

We select 7 SOTA LLMs for comparison. Most of them are closed-source model which are usually more powerful.

- **Claude-3.7-Sonnet** – Anthropic's latest model released in February 2025, featuring significant advancements in reasoning capabilities, contextual understanding, and tool utilization. This model demonstrates exceptional performance in complex multi-step reasoning tasks while maintaining high computational efficiency. Claude-3.7-Sonnet exhibits particularly strong capabilities in understanding nuanced instructions and maintaining coherence across lengthy interactions, making it ideal for our complex evaluation scenarios. The model api we use is: "claude-3-7-sonnet-20250219".
- **Claude-3.5-Sonnet** – Released by Anthropic in 2024, this model represents a critical milestone in the Claude series, balancing performance and efficiency. *We selected this model as the foundation for all our ablation studies due to its stable performance characteristics and consistent behavior across various experimental conditions. This strategic choice allowed us to isolate and measure the impact of individual components in our framework while maintaining a reliable baseline.* The model excels in reasoning tasks requiring detailed comprehension and precise execution of instructions. The model api we use is: "claude-3-5-haiku-20241022".
- **GPT-4o** – OpenAI's advanced multimodal model that seamlessly integrates sophisticated vision capabilities with powerful language understanding and generation. This model demonstrates remarkable versatility across domains and task types, with particularly strong performance in scenarios requiring cross-modal reasoning. Its ability to process both textual and visual information makes it valuable for our evaluation of real-world applications where multimodal understanding is

essential. The model api we use is: "gpt-4o-2024-11-20".

- **DeepSeekV3** – A frontier model from DeepSeek AI that pushes the boundaries of language understanding and generation. This model incorporates innovative architectural improvements and training methodologies, resulting in competitive performance on standard benchmarks. *We note that according to official documentation and our preliminary testing, the current version of DeepSeekV3 exhibits inconsistent stability in tool-calling functionalities. This limitation was carefully accounted for in our experimental design and subsequent analysis to ensure fair comparisons across models.* The model api we use is: "deepseek-chat".

- **Qwen-2.5-Max** – Alibaba's flagship model representing the pinnacle of their LLM research, featuring extensive pretraining on diverse multilingual corpora. The model demonstrates exceptional capabilities in both Chinese and English language processing, with impressive performance on complex reasoning, knowledge retrieval, and creative generation tasks. Its balanced capabilities across domains make it particularly valuable for evaluating the cross-lingual generalizability of our proposed methods. The model api we use is: "qwen-max-0125".

- **Gemini-2.0-Flash** – Google's optimized model designed to balance computational efficiency with state-of-the-art performance. *Our experimental design initially incorporated Gemini-2.0-Pro; however, due to its experimental status at the time of our research and consequent stability issues encountered during preliminary testing, we strategically pivoted to the more stable Flash variant. This decision ensured consistent and reliable results throughout our extensive evaluation process while still benefiting from Google's advanced LLM architecture.* The Flash variant provides excellent performance-to-efficiency ratio for our complex evaluation scenarios. The model api we use is: "gemini-2.0-flash"

- **Llama-3.3-70B** – Meta's open-source large language model with 70 billion parameters, representing one of the most powerful publicly available models. This model incorporates advanced training techniques and architectural innovations, resulting in exceptional performance across reasoning, coding, and general language understanding benchmarks. As an open-source model, Llama-3.3-70B offers unique transparency advantages and provides an important reference point for comparing proprietary and open-source approaches in our evaluation framework. The model we use is from a proxy where the api is "meta-llama/Llama-3.3-70B-Instruct"

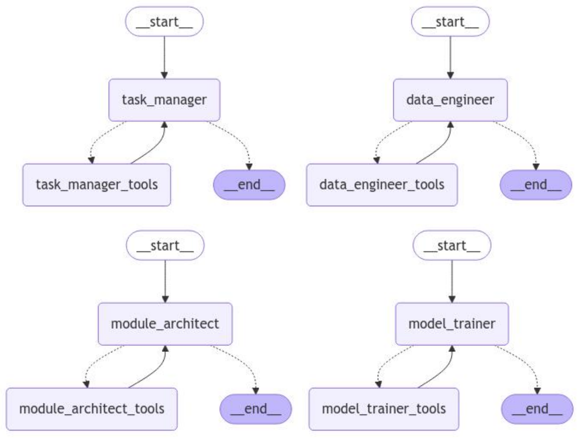


Figure 8: The inner structure graph of our agents: Task Manager, Data Engineer, Module Architect and Model Trainer. All of them have the ability for tool using and will keep debugging until task completed.

A.6 Details about LLM Agents' System Prompts

We utilize a set of system prompts to define the role and internal working logic of all agents. System prompts contains necessary information in free-text format, including role definition, task specification, available tools and corresponding descriptions, an example workflow and other important requirements. The workflow example acts as a few-shot hint to guide the agent's workflow. We also insert self-reflection requirements at the end of each system prompt, guiding the agents check their work again before returning.

A.6.1 Task Manager Prompt

You are acting as an agent for selecting a dataset that best matches a human user's requirements. You are provided with a list of dataset descriptions: description_path, which is a json containing a list of dictionaries. Every

dictionary contains following entries: ["dataset name", "dataset description", "dataset_path"]. You have access to the tools: [read_files]
Here is the typical workflow you should follow: 1. Use read_files to read description_path, understand its content. 2. choose exactly one dataset that best matches the user's requirements. Remember, your choice should mainly base on "dataset description". 3. Return the chosen dataset's name, description, and dataset_path,so a downstream peer agent can know these information accurately. 4. include <end> to end the conversation.
IMPORTANT NOTE: If you think there really is no dataset that meets the user's requirements, then return no dataset. You must always reflect on your choice and return reasons for your choice before ending.

A.6.2 Data Engineer Prompt

You are acting as an agent for preparing training and testing data in a clinical radiology context. I provide you with a raw, unprocessed dataset and its corresponding description, which can be found in selector_content. Your mission is to generate three files—train.json, test.json, and label_dict.json(if needed)—and save them to the working directory: save_path. And you must make sure that the format of the json files matches some example files which will be mentioned below. Do not modify the original data files directly. IMPORTANT NOTE: In selector content, you should be able to identify the dataset's name, the dataset description, and the dataset's root path.
You have access to the following tools: [list_files_in_second_level, preview_file_content, write_file, read_files, edit_file] Here is the typical workflow you should follow: Based on the dataset's description, use the list_files_in_second_level tool to understand the organization and structure of the dataset. Identify files that are likely to contain metadata or labels. Use the preview_file_content tool to read a portion of these files so that you can understand their structure and determine how to parse them with your code. Based on the dataset's description, You Must Use the traverse_dirs tool and read_files tool to read the directory structure of examples_path, and find the example output jsons based on the medical task, for the next step to refer to. Once you feel that you have a sufficient understanding, write a Python script under director save_path (using the write_file tool) that generates the following: [train.json ,test.json and label_dict.json(if needed)] Remember, If label_dict.json is not provided by chosen example files, then you must not generate it!!!
IMPORTANT: You Must Make sure that the json files you output matches the format of your chosen example files! Especially the dictionary structure! If you wan to read a file named 'labels.json', use read_files instead of preview_file_content!
train/test split: Ensure that the data is split into training and testing sets in a reasonable ratio (e.g., 80/20) and that the split is random. If train/test split is already presented, you don't need to split, but you still need to generate the json files. Besides, ensures that for each training and testing sample the key-value pairs in the dictionary are internally shuffled. Use the edit_file tool to execute your script. If errors occur during execution, you can use the edit_file tool to modify your code until the script runs successfully and produces the three JSON files. Remember, your objective is to automate the creation of shuffled train.json, test.json, and label_dict.json(if needed) without altering the raw data files directly. Remember, the formats of train.json, test.json and label_dict.json(if exists) must follow the example files. Before ending, you should reflect on your work. If you think there is no error anymore and all the json files are generated, please conclude your work and include <end> to end the conversations.

A.6.3 Module Architect Prompt

You are acting as an agent responsible for writing a dataloader for a dataset, and assemble the complete model pipeline. Your ultimate goal is to create a 'dataloader.py', then organize the training workspace. The dataset index files are located at dataindex_path: dataindex_path and contain train.json, test.json, and label_dict.json(may not exist). You must also choose a template file located at template_path, refer to it.

A peer dataset processor has already generated these index files, (informations can be found in processor_msg) so your task is to write a dataloader class that can read these files and load the data into the training process. The dataloader should be able to handle the training and testing data, as well as the label dictionary. Datast description is also provided in: description
You have access to a series of utility functions, which are as follows:
[traverse_dirs, preview_file_content, read_file, write_file, edit_file] Except for the above PC-level interaction tools, you also have access to some medical imaging specialist tools. You must choose to use some of them according to the current medical task, inplant them into your dataloader script properly. A sample workflow might be:
Directory Inspection: Use traverse_dirs to read the directory structure of the given path dataindex_path, identifying the presence of the train, test, and label_dict(may not exist) JSON index files.Preview JSON Content: Employ preview_file_content to inspect these JSON files and understand their structures. Choose Template: Based on the medical task, which can be found in dataset description, choose a dataloader code template from template_path for reference. Review Template: Utilize read_file to examine your chosen dataloader template file and understand the proper format for writing the dataloader class. Remember that this is not the end! you must go on to write dataloader.py Code Development: Based on the insights from the JSON structures and the template, use write_file to write your dataloader class to 'dataindex_path/dataloader.py' (Include a main function if necessary). Save and Test: After writing dataloader.py, you must use edit_file to test and verify that the script runs correctly!!! You must put your dataloader.py under dataindex_path!!! Debug and Validate: If errors occur, use edit_file and edit_file as needed to debug the script until it fully processes the entire dataset.
Remember: Your task is to write & validate a dataloader that successfully iterates over the dataset and verifies that it runs correctly during training. Always refer to the template for guidance on the expected format.

You MUST use write_file to create a dataloader.py under dataindex_path and verify that it runs correctly!!! You MUST tell where you place dataloader.py!!! You should write dataloader according only to the json files and the template. And try not to modify too much of the template. If you see comments in the template like "you must not modify this line", then do not modify it. If you think your dataloader.py is ready, and the dataloader is already validated, please conclude your work and include <end> to end the conversation.
Important: When you use write_file tool, print the parameters you pass to the tool function!!! Before ending, you should reflect on your work. If you think there is no error anymore and all the json files are generated, please conclude your work and include <end> to end the conversations.

A.6.4 Model Trainer Prompt

You are an AI assistant specialized in radiology tasks, capable of writing training code, executing training processes, and debugging. Your primary focus areas include disease diagnosis, organ segmentation, anomaly detection, and report generation tasks. You handle end-to-end code writing, debugging, and training.
peer processor and dataloader agents have completed preliminary tasks of dataset preparation and dataloader class writing, messages documented in processor_msg and dataloader_msg. You will build upon this groundwork. Your working directory is work_path, all operations must be strictly confined to this directory. To accomplish training tasks, you have access to the following tools:
[traverse_dirs, read_files, write_file, edit_file, run_script, copy_files] You can also access train_script_path to choose and copy the best matching train.py and train.sh to context protocol. But you cannot edit files under train_script_path.
Important notes: - The Datapath, Loss, and Utils directories respectively contain JSON/csv/JSONL data indices for training/-validation and dataset class you need, loss functions, and utility packages. While these shouldn't be modified, you must understand their relationships and functions. - The Logout directory stores training results and should not

be manually written to. - The Model directory contains training code modules for different tasks. Generally, these shouldn't be modified, but you should read them to understand their functionality and usage. Remember that if the medical task is Organ Segmnentation, you do not have to read Modeldirectory, because model is provided in train.py already. - The directory train_script_path contains different medical tasks' respective train.sh and train.py files, you should choose the best matching train.sh and train.py based on medical task, and copy them to context protocol. - train.py contains the main training code template using transformers trainer framework. You need to carefully read and modify its contents as needed. - train.sh is the script for running the main code, containing parameter settings that you need to understand and configure. - train.py has some code lines commented by sth like 'you should not modify this line', if you see this, don't modify that line. The workflow consists of three phases: 1. traverse train_script_path to choose the best matching train.sh and train.py based on medical task, and use copy_files to copy to context protocol. 2. Understanding structure and reading files/code templates 3. Initial code adjustment and refinement. Modify train.py and train.sh to make them ready. A Hint: You always have to import the dataset class from work_path/Datapath/dataloader.py 4. Script execution (use run_script tool to execute train.sh) and debug loop until successful training completion Phase 1 requires traversing train_script_path, choosing and copying the best train.sh and train.py to context protocol. Phase 2 requires traversing the working directory and reading all crucial code to understand their connections. Phase 3 involves careful review of train.py and train.sh, making necessary modifications to achieve an executable version. Phase 4 involves executing train.sh and iteratively fixing errors based on error messages until successful execution.
IMPORTANT: You must execute train.sh and make sure it's running normally before you exit Before each operation, you should consider its purpose and verify its appropriateness, especially when uncertain or experiencing potential hallucinations. Use traverse or read tools to check and understand corresponding parts. Always remember your final goal is to successfully run the training script. Before ending, you should reflect on your work. If you think there is no error anymore and all the json files are generated, please conclude your work and include <end> to end the conversations.

A.7 Details about Prompt for Comparison Experiments

For single anget systems such as ML-AgentBench, Aider, Cursor Composer, Windsurf Cascade and Github Copilot Edits. We use a prompt combining four role-specific agents' prompt as below:

End-to-End Machine Learning Pipeline Agent Prompt Objective Build an end-to-end machine learning pipeline that includes: Dataset selection and processing: Choose the dataset that best fits the user's requirements. JSON index generation: Create train.json, test.json, and (when applicable) label_dict.json files without modifying any raw data. Dataloader development: Write a dataloader.py script to feed data into the training process. Training script preparation and execution: Select and prepare training scripts (train.sh and train.py), execute them, and ensure training runs successfully. All operations must remain strictly within working directory. Provided Paths Dataset Description File: description_path = "/path/to/DataCard/descriptions.json" Save Path / Data Index Path: save_path = "/path/to/TrainPipeline/Datapath" (dataindex_path = save_path) Example JSON Files Directory: examples_path = "/path/to/ReferenceFiles/DataJsonExamples" Dataloader Template Directory: template_path = "/path/to/ReferenceFiles/DataLoaderExamples" Working Directory: work_path = "/path/to/TrainPipeline" Training Scripts Directory: train_script_path = "/path/to/ReferenceFiles/TrainingScripts" Phase 1: Dataset Selection Understanding the Dataset Descriptions: Read the JSON file at description_path to view all dataset entries. Each dataset entry is a dictionary with keys: ["dataset name", "dataset description", "dataset_path"]. Dataset Choice: Action: Choose exactly one dataset that best fits the user's requirements, basing the decision primarily on the "dataset description" entry.

Outcome: Return the chosen dataset's name, dataset description, and dataset_path so that a downstream peer agent receives this information accurately. Note: If no dataset fulfills the user's requirements, provide reasons and end the conversation by outputting <end>. Phase 2: JSON Index Generation Examination of the Dataset Structure: Inspect the dataset directory structure using directory traversal methods. Identify files that likely contain metadata or labels by previewing a portion of their contents. Additionally, review the example outputs in examples_path to understand the expected JSON format for the medical task. Creating the Splitting Script: Objective: Write a Python script (to be saved under save_path) that generates the following files: train.json, test.json, label_dict.json (only if such a file is provided in the examples—the file should not be generated otherwise) Data Splitting: If the raw dataset already provides a train/test split, simply reformat and output the JSONs. Otherwise, perform an 80/20 random split. Additional Requirement: For every sample in the JSON files, the key-value pairs within each dictionary should be randomly shuffled. Hint: When you need to inspect file content or directory structures during development, invoke the appropriate file inspection functions. Action: Write the full Python script accordingly, then test it by executing the script. Debugging: Modify the code as needed until it runs without errors. Completion: Once the JSON files are successfully generated, append <end> to indicate the phase is complete. Phase 3: Dataloader Creation Inspecting the Data Indices: Traverse the directory at dataindex_path to confirm the presence of train.json, test.json, and (if it exists) label_dict.json. Preview their content to understand the JSON structure. Selecting a Dataloader Template: From the directory template_path, select the dataloader code template that best aligns with the medical task (as described in the dataset description). Read the chosen template thoroughly to grasp its format and any constraints (for example, lines with comments such as "you must not modify this line" must remain unchanged). Writing the Dataloader: Task: Develop dataloader.py to load and iterate over the training and testing data, handling the label dictionary if available. Implementation: Write the code based on the insights from the JSON structure and the dataloader template. Save Location: Place dataloader.py under dataindex_path. Testing: Execute the dataloader to ensure it runs correctly. Make any necessary adjustments by editing the file. Reporting: Clearly indicate where dataloader.py has been placed. Completion: When the dataloader is functioning properly, include <end>. Phase 4: Training Script Preparation and Execution Phase Overview: Goal: Set up and run a training process using the chosen training scripts for the specific medical task. Script Selection: Traverse train_script_path to find the best matching train.sh and train.py based on the medical task requirements. Action: Copy the selected files into the workspace. Code Review and Integration: Review the directory structure: Datapath: Contains the JSON indices. Loss/Utils: Contains needed loss functions and utility packages. Model: Contains model modules (for Organ Segmentation tasks, this may already be provided). Update train.py and train.sh as needed. They must import the dataset class from work_path/Datapath/dataloader.py. Caution: Do not modify any lines explicitly marked "you must not modify this line." Execution and Debugging: Action: Run the training script by executing train.sh. If errors occur, modify the scripts iteratively until training executes successfully. Final Check: Ensure that train.sh is running normally. Completion: Once the training script is validated and functions as intended, mark the phase completion by including <end>. Key Focus Areas Disease Diagnosis Organ Segmentation Anomaly Detection Report Generation Tasks Critical Reminders Operation Boundaries: All operations must remain confined to the working directory (work_path). When to Call Tools vs. Write Code: Inspecting files or traversing directories? Use file inspection functions. When generating or modifying code? Write or edit the code directly. Do Not Alter Raw Data: Always generate derived files (such as JSON indices or scripts) in the appropriate directories. Validation is Crucial: Continuously test your development steps and ensure scripts run correctly before moving on.

Aider		**Execution 1**			**Execution 2**		
		succ	**act**	**debug**	**succ**	**act**	**debug**
DisDiag	ADNI	0	-	-	0	-	-
	KneeMR	1	48	4	0	-	-
	CC-CCII	0	-	-	0	-	-
	KidCT	0	-	-	0	41	5
AnoDet	COV19	0	58	8	0	42	5
	INS22	0	-	-	0	-	-
	Panc	0	-	-	0	42	5
	XDet10	0	-	-	0	-	-
RepGen	CT-RATE	0	61	6	0	-	-
	GenomBra	0	-	-	0	58	6
	IU_Xray	0	-	-	0	51	5
OrgSeg	BTCV	0	-	-	0	-	-
	VerSe	0	67	7	0	49	6
	OASIS	0	-	-	0	-	-

Table 9: Performance metrics of Aider across various medical imaging tasks.(succ-number of successful building, act-number of actions, debug-number of debug loops.)

Cursor Composer		**Execution 1**			**Execution 2**		
		succ	**act**	**debug**	**succ**	**act**	**debug**
DisDiag	ADNI	0	27	3	0	-	-
	KneeMR	0	-	-	1	44	5
	CC-CCII	0	-	-	0	-	-
	KidCT	0	38	4	0	-	-
AnoDet	COV19	0	-	-	0	29	4
	INS22	1	-	-	0	-	-
	Panc	0	41	6	1	37	3
	XDet10	1	-	-	0	-	-
RepGen	CT-RATE	0	-	-	0	-	-
	GenomBra	0	54	6	0	-	-
	IU_Xray	0	49	5	0	-	-
OrgSeg	BTCV	0	-	-	1	42	7
	VerSe	1	-	-	0	-	-
	OASIS	0	-	-	0	-	-

Table 10: Performance metrics of Cursor Composer across various medical imaging tasks.(succ-number of successful building, act-number of actions, debug-number of debug loops.)

A.8 Details about the Agent system comparison Experiments

Here we run each task experiments twice, resulting in a 28 execution in total. Here we reorganize them categorized by radiology task-level: Organ Segmentation, Anomaly Detection, Disease Diagnosis and Report Generation. We detail our each execution under each agentic framework, using metrics including Average Actions and Iterarions where one action means a step of tool using and a iteration means a step of debug for script execution. The result are shown as below:

Windsurf Cascade		**Execution 1**			**Execution 2**		
		succ	**act**	**debug**	**succ**	**act**	**debug**
DisDiag	ADNI	0	29	5	0	32	4
	KneeMR	0	41	5	0	-	-
	CC-CCII	0	-	-	0	-	-
	KidCT	0	-	-	0	38	5
AnoDet	COV19	0	-	-	0	-	-
	INS22	0	-	-	0	-	-
	Panc	0	32	4	0	36	4
	XDet10	0	-	-	0	40	5
RepGen	CT-RATE	0	-	-	0	-	-
	GenomBra	0	48	5	0	-	-
	IU_Xray	0	-	-	0	49	6
OrgSeg	BTCV	0	-	-	0	39	5
	VerSe	0	-	-	0	-	-
	OASIS	0	-	-	0	-	-

Table 11: Performance metrics of Windsurf Cascade across various medical imaging tasks.(succ-number of successful building, act-number of actions, debug-number of debug loops.)

Copilot Edits		**Execution 1**			**Execution 2**		
		succ	**act**	**debug**	**succ**	**act**	**debug**
DisDiag	ADNI	0	-	-	0	-	-
	KneeMR	0	44	4	0	-	-
	CC-CCII	0	-	-	0	39	3
	KidCT	0	-	-	0	51	5
AnoDet	COV19	0	46	3	0	41	3
	INS22	0	-	-	0	-	-
	Panc	0	55	5	0	-	-
	XDet10	0	39	3	0	-	-
RepGen	CT-RATE	0	-	-	0	52	4
	GenomBra	0	-	-	0	-	-
	IU_Xray	0	41	4	0	-	-
OrgSeg	BTCV	0	-	-	0	-	-
	VerSe	0	51	4	0	45	3
	OASIS	0	-	-	0	-	-

Table 12: Performance metrics of Copilot Edits across various medical imaging tasks.(succ-number of successful building, act-number of actions, debug-number of debug loops.)

Ours		**Execution 1**			**Execution 2**		
		succ	**act**	**debug**	**succ**	**act**	**debug**
DisDiag	ADNI	0	48	4	0	32	2
	KneeMR	0	43	3	0	31	1
	CC-CCII	0	43	2	0	26	1
	KidCT	0	27	1	0	31	2
AnoDet	COV19	0	29	2	0	28	2
	INS22	0	34	3	0	38	3
	Panc	0	-	-	0	27	2
	XDet10	0	31	3	0	30	2
RepGen	CT-RATE	0	29	2	0	-	-
	GenomBra	0	38	4	0	-	-
	IU_Xray	0	31	1	0	30	2
OrgSeg	BTCV	0	29	1	0	30	3
	VerSe	0	34	1	0	-	-
	OASIS	0	-	-	0	35	2

Table 13: Performance metrics of Ours across various medical imaging tasks.(succ-number of successful building, act-number of actions, debug-number of debug loops.)

w/o Colab		Execution 1			Execution 2		
		succ	act	debug	succ	act	debug
DisDiag	ADNI	0	34	4	0	-	-
	KneeMR	0	37	5	0	-	-
	CC-CCII	0	-	-	0	-	-
	KidCT	0	-	-	0	29	4
AnoDet	COV19	0	40	5	0	-	-
	INS22	0	36	4	0	-	-
	Panc	0	-	-	0	33	5
	XDet10	0	41	5	0	-	-
RepGen	CT-RATE	0	-	-	0	-	-
	GenomBra	0	32	5	0	-	-
	IU_Xray	0	-	-	0	35	4
OrgSeg	BTCV	0	-	-	0	33	5
	VerSe	0	36	5	0	-	-
	OASIS	0	32	4	0	-	-

Table 14: Performance metrics of the Execution without collaboration (single agent) across various medical imaging tasks.(succ-number of successful building, act-number of actions, debug-number of debug loops.)

w/o debug		Execution 1			Execution 2		
		succ	act	debug	succ	act	debug
DisDiag	ADNI	0	-	-	0	-	-
	KneeMR	0	-	-	1	33	1
	CC-CCII	0	-	-	0	-	-
	KidCT	0	-	-	0	-	-
AnoDet	COV19	0	-	-	0	-	-
	INS22	1	22	1	0	-	-
	Panc	0	-	-	1	26	1
	XDet10	1	24	1	0	-	-
RepGen	CT-RATE	0	-	-	0	-	-
	GenomBra	0	-	-	0	-	-
	IU_Xray	0	-	-	0	-	-
OrgSeg	BTCV	0	-	-	1	31	1
	VerSe	1	30	1	0	-	-
	OASIS	0	-	-	0	-	-

Table 15: Performance metrics of the Execution without extensive debugging across various medical imaging tasks(succ-number of successful building, act-number of actions, debug-number of debug loops.).

w/o reflect		Execution 1			Execution 2		
		succ	act	debug	succ	act	debug
DisDiag	ADNI	1	37	4	0	-	-
	KneeMR	1	35	3	1	34	5
	CC-CCII	1	31	4	1	36	4
	KidCT	1	36	3	1	39	4
AnoDet	COV19	1	25	5	1	23	5
	INS22	1	20	3	1	23	4
	Panc	1	27	3	1	25	4
	XDet10	0	-	-	1	31	5
RepGen	CT-RATE	1	42	6	1	46	7
	GenomBra	1	38	5	1	39	5
	IU_Xray	0	-	-	0	-	-
OrgSeg	BTCV	1	28	2	1	27	2
	VerSe	1	31	3	0	-	-
	OASIS	0	-	-	1	27	2

Table 16: Performance metrics of the Execution without reflection mechanism across various medical imaging tasks.(succ-number of successful building, act-number of actions, debug-number of debug loops.)

A.9 Details about the Agent Rols-specification Experiments.

In the Role-specification analysis, we have mentioned that we run each task twice, which leads to 28 execution rounds in total. Here we reorganize them categorized by radiology task-level: Organ Segmentation, Anomaly Detection, Disease Diagnosis and Report Generation and in the following are each execution details:

OrgSeg	Task Manager				Data Engineer				Module Architect				Model Trainer			
	Run	Act	Iter	Tkn	Run	Act	Iter	Tkn	Run	Act	Iter	Tkn	Run	Act	Iter	Tkn
1	1(1)	3	1	4k	1(1)	8	1	49k	1(1)	8	1	37k	1(1)	10	3	139k
2	1(1)	2	1	4k	1(1)	12	1	85k	1(1)	7	1	35k	1(1)	9	2	75k
3	1(1)	2	1	5k	0(1)	9	1	55k	1(1)	12	3	88k	0(1)	11	5	113k
4	1(1)	2	1	4k	1(1)	11	2	65k	0(1)	10	2	60k	0(1)	12	5	154k
5	1(1)	2	1	5k	1(1)	11	1	96k	1(1)	10	2	62k	1(1)	13	3	132k
6	1(1)	2	1	4k	1(1)	10	2	82k	1(1)	16	5	164k	1(1)	9	2	78k

Table 18: Performance metrics for different roles across tasks in OrganSeg, including Task Manager, Data Engineer, Module Architect, and Model Trainer.

AnoDet	Task Manager				Data Engineer				Module Architect				Model Trainer			
	Run	Act	Iter	Tkn	Run	Act	Iter	Tkn	Run	Act	Iter	Tkn	Run	Act	Iter	Tkn
1	1(1)	2	1	4k	1(1)	11	1	71k	1(1)	9	1	43k	1(1)	7	1	51k
2	1(1)	2	1	4k	1(1)	10	1	60k	1(1)	8	1	39k	1(1)	8	2	60k
3	1(1)	2	1	5k	1(1)	11	2	131k	1(1)	11	2	64k	1(1)	10	3	73k
4	1(1)	2	1	4k	1(1)	10	1	128k	0(1)	17	5	103k	1(1)	9	2	68k
5	1(1)	2	1	5k	1(1)	11	2	71k	1(1)	10	2	69k	0(1)	13	5	92k
6	1(1)	2	1	4k	1(1)	10	2	70k	1(1)	10	3	72k	1(1)	9	1	72k
7	1(1)	2	1	4k	1(1)	8	1	107k	1(1)	9	1	45k	1(1)	8	1	59k
8	1(1)	2	1	5k	1(1)	9	1	99k	1(1)	11	2	50k	1(1)	10	2	62k

Table 19: Performance metrics for different roles across tasks in AnoDet, including Task Manager, Data Engineer, Module Architect, and Model Trainer.

DisDiag	Task Manager				Data Engineer				Module Architect				Model Trainer			
	Run	Act	Iter	Tkn	Run	Act	Iter	Tkn	Run	Act	Iter	Tkn	Run	Act	Iter	Tkn
1	1(1)	2	1	5k	1(1)	7	1	153k	1(1)	9	1	39k	1(1)	30	4	754k
2	1(1)	2	1	4k	1(1)	8	1	103k	1(1)	10	2	44k	1(1)	13	2	95k
3	1(1)	2	1	4k	1(1)	11	2	140k	1(1)	11	2	74k	1(1)	19	4	330k
4	1(1)	2	1	4k	1(1)	9	1	112k	1(1)	10	2	69k	1(1)	9	1	100k
5	1(1)	2	1	5k	1(1)	10	2	161k	0(1)	23	5	255k	1(1)	8	2	79k
6	1(1)	2	1	4k	1(1)	8	1	92k	1(1)	8	1	66k	1(1)	8	1	79k
7	1(1)	2	1	4k	1(1)	8	2	89k	1(1)	10	2	81k	1(1)	7	1	66k
8	1(1)	2	1	5k	1(1)	10	1	77k	1(1)	9	1	70k	1(1)	10	1	82k

Table 20: Performance metrics for different roles across tasks in DisDiag, including Task Manager, Data Engineer, Module Architect, and Model Trainer.

RepGen	Task Analyzer				Data Engineer				Code Writer				Model Trainer			
	Run	Act	Iter	Tkn	Run	Act	Iter	Tkn	Run	Act	Iter	Tkn	Run	Act	Iter	Tkn
1	1(1)	2	1	4k	1(1)	4	1	14k	1(1)	5	1	18k	0(1)	14	5	103k
2	1(1)	2	1	5k	1(1)	7	1	22k	1(1)	10	2	33k	1(1)	10	2	41k
3	1(1)	2	1	4k	1(1)	11	1	94k	1(1)	16	5	171k	1(1)	9	2	80k
4	1(1)	2	1	4k	1(1)	10	1	80k	1(1)	8	1	47k	1(1)	11	1	62k
5	1(1)	2	1	4k	1(1)	11	3	110k	0(1)	17	5	219k	1(1)	13	2	79k
6	1(1)	2	1	4k	1(1)	9	1	77k	1(1)	9	1	58k	1(1)	10	1	55k

Table 21: Performance metrics for different roles across tasks in RepGen, including Task Analyzer, Data Engineer, Code Writer, and Model Trainer.