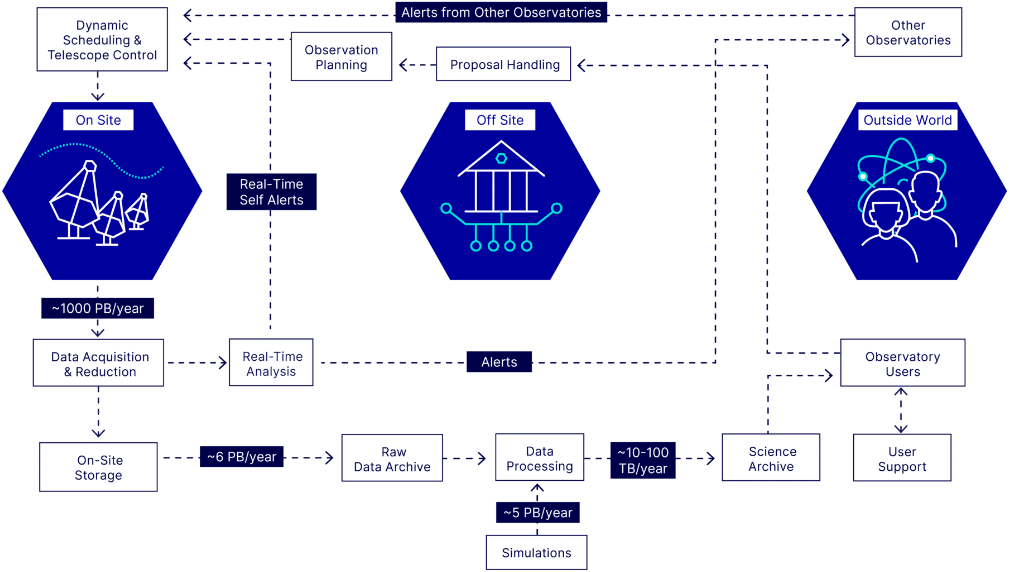
AI Agents for Ground-Based Gamma Astronomy

Dmitriy Kostunin[1], Vladimir Sotnikov[2], Sergo Golovachev[2], Alexandre Strube[3]

[1]DESY, [2]JetBrains, [3]Forschungszentrum Jülich

January 30, 2025

Data flow of Cherenkov Telescope Array Observatory (CTAO)



Points of AI application in astronomy

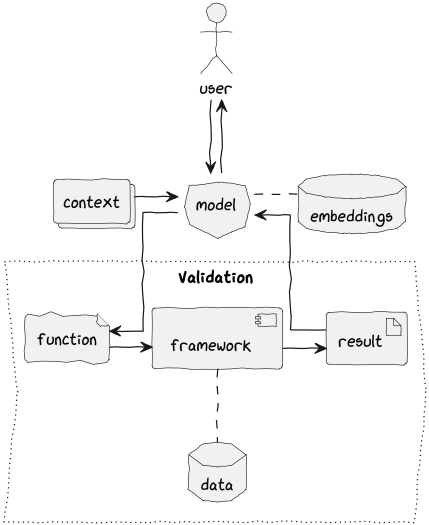
*An **AI Agent** is an autonomous entity that perceives its environment through sensors, processes the information, and takes actions to achieve specific goals or objectives. Designed to exhibit intelligent behavior, AI agents can learn from experiences, adapt to new situations, and make decisions based on their observations and programmed knowledge. They can exist as software programs or physical robots, and they operate using algorithms and models that enable reasoning, problem-solving, and interaction with their environment.*

— ChatGPT o1-preview

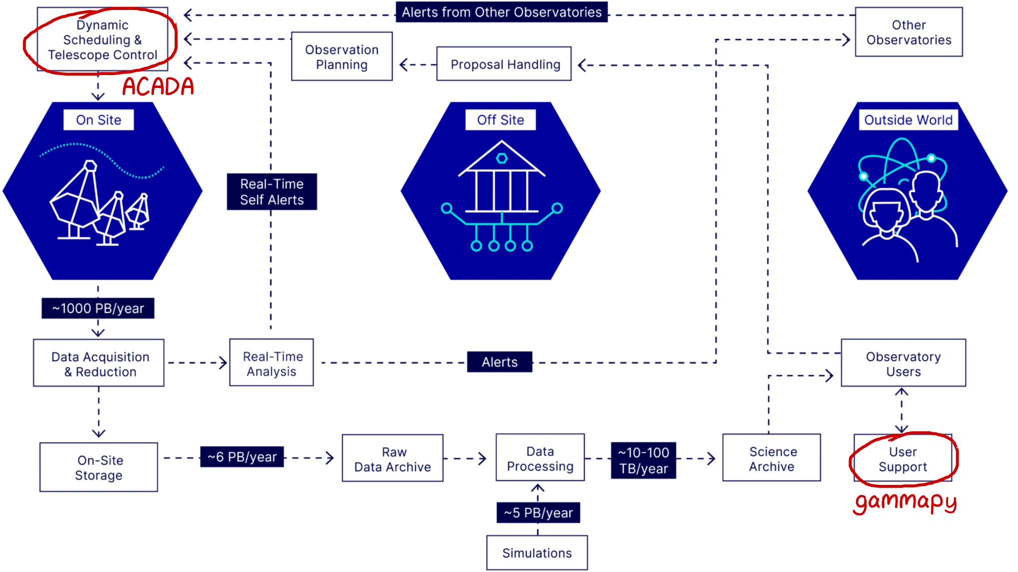
- Telescope control (i.e., "copilots")
- Monitoring / Alarms / Reporting
- Data Quality Assurance
- Data Analysis and observation proposal processing

Do we fall under “agent” definition?

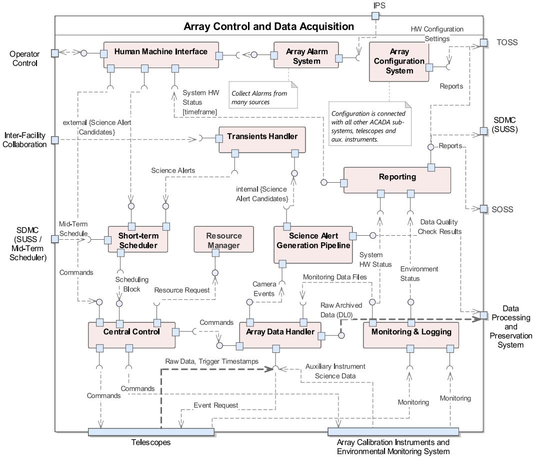
- *exhibit intelligent behavior*
  - reasoning, chain-of-thoughts?
- *learn from experiences*
  - **validation** using our software and (synthetic) data
- *make decisions*
  - triggered by user (direct prompting? gitlab bot?)



Let's start at opposite ends



Array Control and Data Acquisition (ACADA) of CTAO



A single package consisting of eleven subsystems with the following stack:

- ACS[a] + OPC UA as middleware
- C++, Python, Java for backend
- Web-based frontend (HMI)
- MySQL, MongoDB, Cassandra, Redis
- Apache Kafka, ØMQ, Google Protobuf
- Slurm
- AlmaLinux + Docker

[a] Alma Common Software

Configuration Database (CDB) of ACADA

- Lightweight Python component
- SQLAlchemy + Pydantic + FastAPI
- Connected with ACADA subsystems, telescopes and auxiliary instruments

Main challenges for integration

- Definition of JSON/Pydantic schemas
- Configuration data maintenance

Generating telescope structure configuration


MST-STR-ICD-36141000-00008
Version 1.3 10.07.2024

cta

MST Structure Configuration

Prepared by: Philipp Wagner, Ullrich Schwanke, Florian Leitgeb, Tim Holch, Thomas Murach

Approved by: Ullrich Schwanke, Thomas Murach

Released by: Markus Garczarczyk

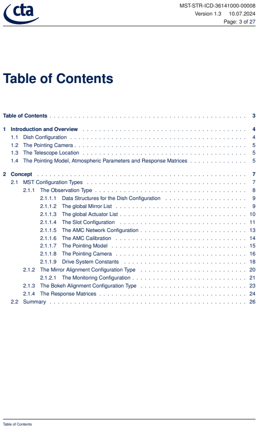
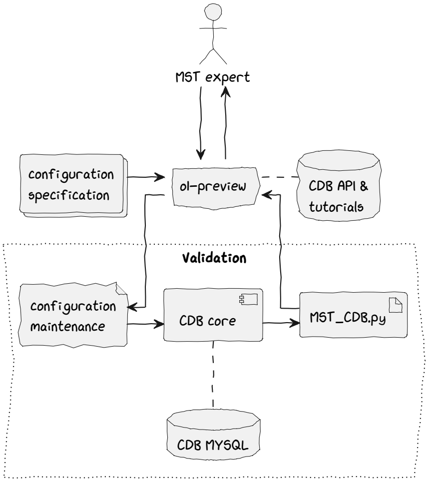

MST-STR-ICD-36141000-00008
Version 1.3 10.07.2024
Page: 3 of 27

Table of Contents

Table of Contents



We used specifications' draft provided by telescope team

Generation and cross-check of the data models

- Configuration note is well drafted in LaTeX format JSON examples are included as separate files, i.e. not visible for LLM

P: *Analyze the text of this book in latex. It describes the configuration model of a telescope. You need to provide Pydantic classes that describe the configuration model. Use as many classes as needed, they can (and probably should) be nested. Here's the text: ...*

P: *Compare your models with these JSON schemas and provide a detailed analysis: ...*

⇒ The most of the human-generated JSON schemas are in agreement with LLM-generated Pydantic models, however, few discrepancies were found correctly.

12. `pcam_sw.json`

JSON Schema:

```
{
   "$id":"https://acadacdb.cta-observatory.org/schemas/mst.struct/pcam_sw.json",
   "type":"object",
   "properties":{
      "exposures":{
         "type":"array",
         "items":{"type":"number"}
      },
      "flags":{
         "type":"string",
         "enum":["SINGLE_SHOT", "CONTINUOUS"]
      },
      "rate":{"type":"number"}
   },
   "required":["exposures", "flags", "rate"]
}
```

Pydantic Model:

```
class PCAMFlags(str, Enum):
    SINGLE_SHOT = 'SINGLE_SHOT'
    CONTINUOUS = 'CONTINUOUS'

class PointingCameraSoftwareConfiguration(BaseModel):
    exposures: List[float]
    flags: PCAMFlags
    rate: float
```

Analysis:

- **Fields and Types:** All fields match, with `flags` using an `Enum`.
- **Conclusion:** The `PointingCameraSoftwareConfiguration` model matches the `pcam_sw.json` schema.

Implementing ACADA CDB wrapper

- We provided necessary context from CDB (API and tutorials) and models generated in previous step.

P: *Here's a tutorial that demonstrates how you can upload Pydantic models to the telescope's configuration database: ...*
Here's the 'cdb.py' from 'acadacdb.core': ...
Here are Pydantic models you will need to use: ...
Write the code that adds all top-level configurations from the script above to the database (backend).

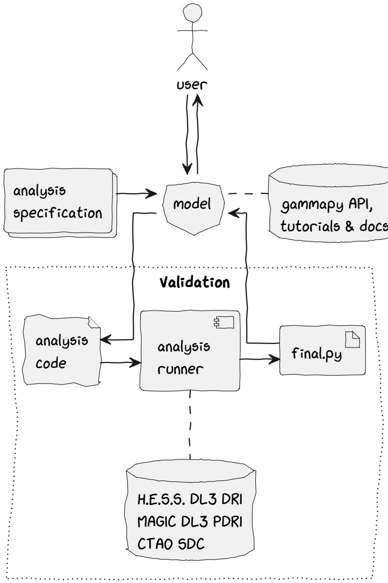
⇒ The model generated a skeleton for CDB population:

  - Naming conventions are understood from tutorial
  - Generator for dummy values is provided
  - Data were successfully uploaded out-of-box

```python
# Define configurations to add
configurations = {
    'ObservationConfiguration': ObservationConfiguration,
    'BokehAlignmentConfiguration': BokehAlignmentConfiguration,
    'AMCDeviceServer': AMCDeviceServer,
    'Mirror': Mirror,
    'Actuator': Actuator
}

for name, model in configurations.items():
    print(f"Uploading {name} configuration schema to the database")
    # Create Confschema
    confschema = Confschema(data=model.schema()).add(backend)

    print(f"Creating corresponding element and conftype for {name}")
    element_name = f'tutorial.{name.lower()}'
    conftype_name = f'{name}Type'

    # Create Conftype and Element if not exists
    if not Conftype(name=conftype_name).get(backend):
        Conftype(name=conftype_name).add(backend)
    if not Element(name=element_name).get(backend):
        Element(name=element_name).add(backend)

    print(f"Linking default schema for {name}")
    DefaultSchema(element=element_name, conftype=conftype_name,
                  confschema=confschema).add(backend)

    print(f"Adding default configuration for {name}")
    # Create an instance of the configuration with example data
    data = create_example_data(model)
    Configuration(element=element_name, conftype=conftype_name, item_id=1,
                  data=data).add(backend)

print("All configurations have been uploaded to the database.")


def create_example_data(model):
    if model is ObservationConfiguration:
        # Create an instance of ObservationConfiguration with example data
        observation_config = ObservationConfiguration(
            title='Default Observation Configuration',
            description='This is a default observation configuration',
            creation_date='2023-10-04',
```

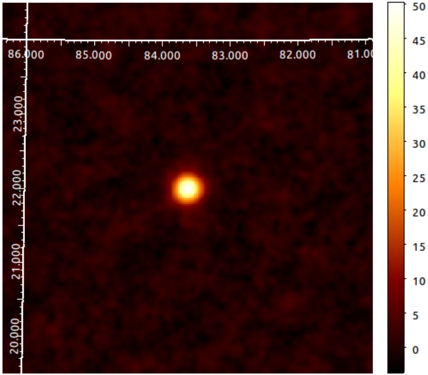
First steps towards `gammapy` agent

- Generic AI solutions (e.g., ChatGPT) feature shallow knowledge and outdated datasets
- **Gammapy is challenging for LLMs**
  - Different (incompatible) versions in training dataset
  - Namings, data structures, and logic are *unnatural* for machine
- **AstroAgent is addressing the challenge**
  - Field-specific prompts and Retrieval-Augmented Generation (RAG)
  - Iterative function calling and validation against data



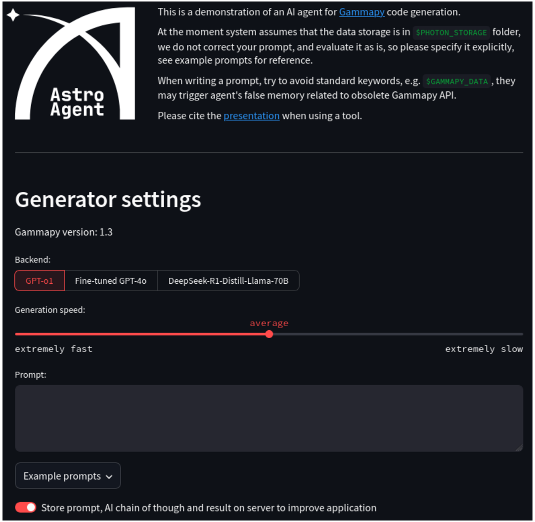
First light (Crab Nebula detection from H.E.S.S. DL3 DR1)

The data for gammapy analysis are stored in `$PHOTON_STORAGE`. Generate a code which selects available observations of Crab Nebula. Using these observations, save a plot with a significance map (`sqrt_ts`) based on the `RingBackground` method. Use an exclusion mask for background maker.




```python
# ... imports and comments are removed, some parts are shortened
crab_position = SkyCoord(ra=83.63308, dec=22.01450, unit="deg", frame="icrs")
selection = data_store.obs_table.select_sky_circle(
    center=crab_position, radius=2.5 * u.deg
)
observation_ids = selection["OBS_ID"].data
observations = data_store.get_observations(observation_ids)
energy_axis = MapAxis.from_energy_bounds(0.1, 10.0, nbin=10, unit="TeV", name="en
geom = WcsGeom.create(
    skydir=crab_position, axes=[energy_axis]
    width=(5, 5), binsz=0.02, frame="icrs",
)
exclusion_mask = WcsNDMap.from_geom(geom)
crab_exclusion_region = CircleSkyRegion(center=crab_position, radius=0.3 * u.deg)
exclusion_mask.data += geom.region_mask(regions=[crab_exclusion_region], inside=F
dataset_empty = MapDataset.create(geom=geom)
dataset_maker = MapDatasetMaker()
bkg_maker = RingBackgroundMaker(r_in="0.35 deg", width="0.3 deg", exclusion_mask=
safe_mask_maker = SafeMaskMaker(methods=["offset-max"], offset_max=4.0 * u.deg)
for obs in observations:
    dataset = dataset_maker.run(dataset_empty, obs)
    dataset = bkg_maker.run(dataset)
    dataset = safe_mask_maker.run(dataset, obs)
    dataset_empty.stack(dataset)
estimator = TSMapEstimator()
ts_map = estimator.run(dataset_empty)
significance_map = ts_map["sqrt_ts"]
print(f"Max significance value: {np.max(significance_map.data):.2f}")
# prints: Max significance value: 50.32
significance_map.write("significance_map.fits", overwrite=True)
```

AstroAgent prototype



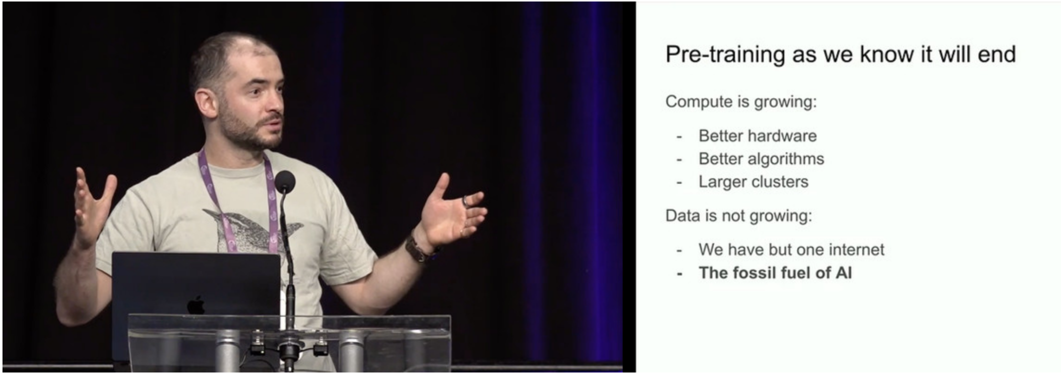
majestix-vm8.zeuthen.desy.de

- Agent demonstrator application
- Several LLMs are included
  - OpenAI LLMs as state-of-art
  - Open-source LLMs from Helmholtz Blablador
  - DeepSeek is in progress
- Sophisticated prompting is still needed (see examples)
- H.E.S.S. DL3 DR1 is used for validation testing
- Successful generation is used for GPT-4o fine-tuning
- Collecting feedback from community

Bonus: What is about the data?

Interesting talk by Ilya Sutskever:

https://www.youtube.com/watch?v=1yvBqasHLZs



- Synthetic data?
- Inference-time scaling?
- Agents?

Conclusion

- LLMs show clear enhancement in data modeling and code generation
- Very successful test for CTAO data models generation
- Human field-specific expertise is still crucial for the fine-tuning
- The comparison between different backends in progress
- Aim for switching to the open source models

Acknowledgements

gammapy | CTAO ACADA | H.E.S.S.

Blablador | JetBrains