Dynamic Tool Generation for Autonomous Multi-Agent Systems in Space Missions

Dominik Opitz
German Aerospace Center (DLR)
Cologne, Germany
dominik.opitz@dlr.de

Carsten Hartmann
German Space Operations Center (DLR-GSOC)
Weßling, Germany
carsten.hartmann@dlr.de

Tobias Hecking
German Aerospace Center (DLR)
Cologne, Germany
tobias.hecking@dlr.de

Michael Felderer
German Aerospace Center (DLR)
Cologne, Germany
michael.felderer@dlr.de

ABSTRACT

Deep space missions are advancing increasingly further away from Earth, making continuous communication for mission maintenance progressively more difficult. To enable ongoing scientific exploration even without communication to Earth and to prevent mission abortion in critical situations, spacecrafts must be capable of operating with a high degree of autonomy. Achieving this level of autonomy requires intelligent systems that can reason about complex situations and interact effectively with their environment, often by leveraging specialized tools.

Current autonomous agents, however, typically rely on predefined tools to interact with their environment, which limits their flexibility in novel or unexpected situations. In this work, we present a framework that enables LLM-based agents to dynamically generate Python tools at runtime, allowing autonomous reasoning and task execution in previously unseen environments - a crucial capability for deep space missions. Our system connects a dedicated Tool-Agent with a planning agent to form a fully autonomous pipeline capable of planning and executing unseen tasks without predefined tools or task-specific guidelines. The system functions with standard mid-sized LLMs (up to 30B parameters), without pre-training or in-context learning. We demonstrate this approach on a space debris scenario, where a satellite detects nearby debris; our system successfully assesses collision risk by autonomously creating, executing, and analyzing tools for trajectory calculation of the two objects.

While this work is still a proof-of-concept, it highlights the potential of runtime tool generation to enhance the autonomy of agents operating in deep space, enabling more resilient and adaptive mission operations.

KEYWORDS

LLM, Tool-Generation, Multi-Agent Systems, Space Systems

Appears at the International Workshop on Autonomous Agents and Multi-Agent Systems for Space Applications (MASSpace-26). Held as part of the Workshops at the 25th International Conference on Autonomous Agents and Multiagent Systems., S. Chien, G. Picard, I. Zilberstein (Chairs), May 2026, Paphos, Cyprus. © 2026 Copyright held by the owner/author(s).

ACM Reference Format:
Dominik Opitz, Carsten Hartmann, Tobias Hecking, and Michael Felderer. 2026. Dynamic Tool Generation for Autonomous Multi-Agent Systems in Space Missions. In *Appears at the International Workshop on Autonomous Agents and Multi-Agent Systems for Space Applications (MASSpace-26). Held as part of the Workshops at the 25th International Conference on Autonomous Agents and Multiagent Systems., Paphos, Cyprus, May 2026*, IFAAMAS, 10 pages.

1 INTRODUCTION

We propose a framework that integrates a Planning-Agent with a dedicated Tool-Agent capable of dynamically generating Python scripts at runtime. The Planning-Agent maintains a high-level task plan that is continuously updated with gained knowledge and subtasks as needed. The Tool-Agent focuses on executing delegated subtasks by creating and utilizing tools when required. It is tightly coupled to the Planning-Agent and can request additional input from it at any time during task execution. Tools are stored and reusable so that they can be used multiple times and in sequence. Complex and large outputs from tools are stored in dedicated artifacts, keeping the agents working context manageable. Both agents condense their interaction history after logical steps, preserving task-relevant information and supporting focused reasoning like that of episodic memory. In contrast to related works, our framework does not rely on the use of external services and heavy processing in the cloud. Instead it employs small-sized agents up to 30B parameters that can be operated on platforms with constraints on computational resources, memory, and energy consumption, which makes it suitable for increasing autonomy of remote space systems operating under uncertain conditions.

In summary, our contribution is as follows:

- A multi-agent architecture combining high-level planning with dynamic, on-demand tool generation,
- A mechanism for safe and reusable tool execution with artifact-based state management,
- An approach for maintaining concise, task-relevant memory to support autonomous reasoning in complex, novel environments.

We demonstrate our system with a scenario, where the system is used to assess the collision risk of space debris with a satellite. In general, the field of spacecraft collision avoidance studies the process of minimizing or mitigating the risk of collisions between

spacecraft in Earth orbit and other orbiting objects. These orbiting objects can be other spacecraft, remnants of satellite collisions, or naturally occurring materials, like micrometeoroids and natural satellites. The European Space Agency (ESA) currently tracks around 15,860 (man-made) satellites in Earth orbit, of which about 12,900 are still functional. Furthermore, the overall number of objects in orbit is 54,000 space objects greater than 10 cm, 1.2 million objects between 10 cm and 1 cm, and 130 million objects between 1 mm and 1 cm [12]. While objects of very small size seem noncritical, a potential impact with another object can have catastrophic results, due to the high relative velocity and thus high kinetic energy. For example, the orbital velocity in low Earth orbit (LEO) is around $7.8km/s$. Therefore, two objects perpendicular to each other would collide at roughly $12.2km/s$. Even tiny particles like paint flecks or solidified liquids expelled from spacecraft could permanently damage, or, in the worst case, disintegrate another object. As such, these collisions should be avoided at all cost. However, this currently requires tremendous effort, from building and maintaining ground infrastructure for the tracking of objects, to the computational effort of collision prediction methods, to the organizational and operational effort to coordinate, plan and execute an avoidance maneuver. This is further complicated if one of the objects cannot be remote controlled (e.g., due to being defunct), or there isn't sufficient time to act. While these circumstances already justify the need for increased autonomy, the problem of collision avoidance is predicted to become even worse, with many public companies and national institutions proposing or launching mega-constellations of satellites, like SpaceX's Starlink, Amanzon's Leo (formerly known as Project Kuiper), or China's Guowang satellites. ESA suggests, that, even without additional launches, catastrophic collision numbers will increase, due to the fact that collision events add debris objects faster than debris can naturally re-enter the atmosphere [12].

To deal with this increase in collision events, future spacecraft might be equipped with autonomous on-board capabilities to detect other objects, predict the point and time of closest approach (PCA and TCA) and perform avoidance maneuvers to prevent potential collisions. More specifically, the PCA is defined as the point in each object's orbit, where the magnitude of the relative position vector between the two objects is a minimum, and the TCA is defined as the time, at which the minimum miss distance between the two objects occurs. In the system that is developed as part of this paper, specifically the task of collision prediction is demonstrated, by utilizing dynamic tool generation. For the purposes of this, we refer to the controlled asset (i.e. the spacecraft performing collision avoidance) as the primary object and the object it encounters as the secondary object. Furthermore, encounters can be classified in two different scenarios: 1. short-term and 2. long-term [18]. On the one hand, short-term encounters usually happen between objects with two significantly different orbits, which results in very high velocities at the point of closest approach and lasting a few seconds only. On the other hand, long-term encounters happen between two satellites traveling along near identical orbits. For the purposes of demonstrating our system only the first scenario of encounter is considered. The goal is to combine a tool-agent with a planning agent, in order to dynamically generate Python code, that can solve the task of finding the PCA and TCA, without providing predefined tools.

2 RELATED WORK

Since the introduction of Large Language Models (LLMs), we are witnessing a progressive shift away from passive chatbots with static knowledge towards active agents with interactive capabilities. A significant part of research focuses on *Agents* that leverage predefined tools to extend their reasoning and interaction abilities. Such tools enable them to perform tasks such as query databases, retrieve web information or perform structured calculations. Since such approaches are limited by the tools that are provided to them, the agents are constrained in their flexibility when faced with novel and unexpected tasks or data. For this reason, recent work has begun exploring dynamic tool generation, a concept where tools are no longer required before runtime, but instead dynamically synthesized during the reasoning process.

Among the earliest adopters in the realm of tool-generating systems is *ATLASS* [7]. *ATLASS* is a closed-loop framework designed for problem solving tasks with tool learning and selection. The system is divided into three stages to (i) determine whether tools are required, (ii) generate or retrieve required tools and (iii) execute the respective tools in order to complete the initial task. The Tool Generation process is designed as an iterative loop that involves the installation of dependencies and writing of Python code until the code execution runs without errors. *ATLASS* is designed with the GPT-4.0 model from OpenAI, a large, cloud-based model.

Wölflein et al. [17] present *ToolMaker*, a framework that turns scientific papers and their accompanied code repositories into functional, LLM-compatible tools, facilitating reproducibility of scientific publications. Their system follows an iterative workflow inside a dedicated execution environment to handle repository installation, dependency management and function generation. Wölflein et al. demonstrate how agents can go beyond the generation of simple tools and instead "produce tools for real-world scientific tasks". *ToolMaker* was designed around and evaluated on several large-scale cloud models, namely GPT-4o, GPT-o3-mini and Claude 3.5 Sonnet.

While *ATLASS* and *ToolMaker* use prompt techniques to enable the generation of tools, *DeepAgent* [10] introduced by Li et al. focuses on discovering and invoking existing tools from massive tool sets. Rather than creating new tools, *ToolMaker* uses a retrieval system to match relevant tools and their documentation for a particular task. In order to improve tool-use in subsequent requests, the internal reasoning models maintain a tool memory to store executed tools along with metadata such as success rates and parameter combinations. In their study, Li et al. used the QwQ-32B model [15] as their main reasoning LLM.

Recent work has further introduced a paradigm shift away from purpose-specific tools towards the acquisition of broader skill-sets. *CASCADE* [8] introduces an agentic system that learns to establish executable scientific routines that are autonomously developed and stored for future use. Here, an essential concept is the agents ability to acquire skills through pre-defined "meta-tools" such as web search and code execution. Evaluated on several of OpenAI's

models, Claude and a Qwen 30B model, *CASCADE* significantly outperforms traditional "tool-use" baselines.

The aforementioned frameworks primarily target terrestrial research or labratory environments. Systems like ATLASS and ToolMaker rely on high-resource cloud-based models, a requirement that is misaligned with the constraints of space systems, which operate under severe computational limitations. Instead, our framework is specifically optimized to function with standard mid-sized LLMs (up to 30B parameters) that can be operated on modern edge-processors. In contrast to CASCADE and DeepAgent, which depend on persistent external connectivity for web-based learning or large-scale API retrieval, our architecture is designed to operate independently from external resources. Rather than discovering existing tools through pre-computed indices, as in DeepAgent, or learning and reusing complex external routines, as in CASCADE, our system employs a dual-agent pipeline that dynamically generates Python scripts at runtime. These scripts directly process raw environmental data without relying on predefined task-specific instructions. Thus, our approach can be a building block of autonomous systems operating on spacecraft where energy consumption is a crucial factor as well as in in isolated environments, such as deep space missions where persistent high bandwidth communication with Earth is limited.

3 MOTIVATING EXAMPLE

To illustrate the practical utility of our framework, we employ a scenario where a satellite (primary object) detects a nearby piece of space debris (secondary object) and must independently assess the risk of collision. As mentioned in the introduction (ref. ch. 1), this is referred to as a short-term encounter, and the goal of the system is to find the point of closest approach. Provided with only two location measurements for each object, the system receives the following task:

> *A new piece of space debris has been detected in the vicinity of our satellite orbiting Earth. For the satellite and the debris, two location measurements have been recorded at two different timestamps, stored in the 'satellite_measurements.csv' and 'debris_measurements.csv' files within the 'trajectories' directory.*
> *Determine whether the newly detected space debris poses a collision threat to the satellite.*

For context, the tracked location data are summarized in Table 1. Measurements for both the satellite and the debris include the columns `timestamp`, `x`, `y`, and `z`. The debris data additionally contain a `mass` column, which is irrelevant to the task but introduces minor noise that the agents should ideally ignore.

Our multi-agent setup successfully identifies both the time and point of closest approach (TCA and PCA), yielding a minimum distance of 21.38 units after 11.43 units of time. A visual, step-by-step trace of the agent interactions is provided in Appendix A. Unlike approaches that rely on in-context LLM calculations - which are often unsafe or approximate - our agents outsource all critical computations to Python tools, ensuring precise and reliable results. Notably, the agents autonomously determined that they needed to access and extract measurement data from the provided CSV files, a task that LLMs cannot typically perform on their own. Importantly, these tools were not predefined or externally instructed; the agents themselves identified the necessity of each tool and generated or executed it as required, demonstrating the key advantage of our dynamic, self-directed tool-generation approach.

timestamp	x	y	z	mass
Debris measurements				
0	40	-40	-5	20
10	40	0	15	20
Satellite measurements				
0	0	0	-5	—
10	20	0	5	—

Table 1: Location measurements for debris and satellite objects.

At a high level, the agents proceed as follows (see also Appendix A):

1. The Reasoning Agent initializes a high-level plan consisting of four steps: (1) load and interpret measurements, (2) calculate trajectories, (3) predict closest approach, and (4) assess collision risk.
2. The Reasoning Agent delegates the first execution task to the Tool Agent, requesting the extraction of the measurements of the satellite and debris locations from file.
3. The Tool Agent fulfills this request through the following substeps:

3.1. It generates the tool `inspect_csv_files(file_paths)` to inspect the structure of the input data. The tool determines that the satellite measurements contain the columns `timestamp`, `x`, `y`, and `z`, while the debris measurements additionally include a `mass` column.

3.2. Based on the observed structure, the Tool Agent generates a second tool, `extract_trajectory_data(file_paths)`, which extracts the relevant position measurements. The `mass` column is correctly ignored, as it is not required for the task. The extracted measurements are stored in artifact `42b`.

4. Following the plan, the Reasoning Agent delegates the next task to the Tool Agent, instructing it to compute the trajectories of both objects.
5. To do this, the Tool Agent first generates a new tool called `calculate_trajectory(measurements)`, which computes position and velocity vectors as well as final positions for each object. The tool is then executed using the measurement data from artifact `42b`, and the resulting trajectory parameters are stored in artifact `6f8`.
6. The Reasoning Agent then delegates the computation of TCA and PCA to the Tool Agent.
7. To complete this task, the Tool Agent generates the tool `calculate_closest_approach(trajectory_params)` and executes it using the trajectory parameters stored in artifact `6f8`. The initial execution fails due to a key error; the tool is subsequently repaired and re-executed successfully. The correct TCA and PCA values are stored in artifact `475` and returned to the Reasoning Agent.

8. With all planned steps completed, the Reasoning Agent finalizes the assessment and terminates the process.

4 METHODOLOGY

We illustrate the setup of our proposed framework in Figure 1. The frameworks' core is composed of the *Reasoning Agent* and the *Tool Agent*. The Reasoning Agent is responsible for the maintenance of the overarching goal, task decomposition and task delegation. The Tool Agent independently executes dedicated, more isolated tasks directly delegated by the Reasoning Agent. It also coordinates the generation and execution of tools, working together closely with the Coding Agent used for the generation of tools (Python scripts).

Once created, tools are stored alongside augmented information describing the tools purpose and usage. Executed tools produce persistent artifacts that can be viewed and invoked directly within tool calls. Finally, our system employs an *Epoisodic Summarizer* to condense the reasoning history of the Reasoning Agent and Tool Agent after logical steps. This approach has proven successful in maintaining focused task execution [10, 11, 16].

Generally, we aim to utilize models as small as possible as practical deployment in space environments is constrained by limited onboard computational resources.

4.1 Components

In this subsection, we introduce the main components of our framework. For this purpose, we focus on outlining their role within the overall architecture.

Reasoning Agent The Reasoning Agent is responsible for planning and coordinating the steps required to reach a defined goal. It maintains a continuous plan to organize sequences of tasks, delegate them to the Tool Agent and store facts and knowledge gathered through the process. As this role requires more general and broader commonsense reasoning abilities, we base the Reasoning Model on the `Gemma 27B` LLM [13].

Tool Agent Within our framework, the Tool Agent operates as an execution-oriented component that fulfills subtasks delegated by the Reasoning Agent. From the perspective of the Reasoning Agent, it is treated as a black box: The Reasoning Agent specifies **what** information is required or what needs to be done, but remains agnostic in **how** that information is obtained or a task is executed. The Tool Agent's novel capability is the autonomous coordination, generation and execution of tools. It uses four specialized commands to initiate any of these actions, which is detailed in Section 4.2. The agent does not generate the raw source code of tools itself, but rather manages the coordination and use around tools. As such, it is assigned with a `Qwen 30B` LLM [14], a model which we found delivers straightforward reasoning steps, ideal for this use case.

Coding Agent The Coding Agent is directly responsible for (i) generating the source code of new tools, (ii) classifying the reason for failed tool executions and (iii) repairing tools that have failed after execution, if the failure reason was assessed to be an issue in the code. Due to its highly focused responsibilities, the Coding Agent is based on a `Qwen-Coder 14B` [9] model by default. In the case of repeated generation of invalid code, the model dynamically increases to its larger `Qwen-Coder 30B` variant.

Tools Tools are isolated Python scripts intended to support specific, encapsulated use cases, ranging from basic operations such as local file access to advanced tasks involving mathematical computation and external data retrieval. Each tool execution yields artifacts containing (i) the raw output, (ii) associated meta-information (e.g., data type), and (iii) a verbalized summary when the output cannot be passed directly to the Tool Agent, such as in cases of large outputs or non-serializable data objects (e.g., database clients or library instances). Tools are invoked by the Tool Agent with a defined set of parameters, which may be specified directly or populated via references to artifacts produced by prior tool executions. This design supports sequential tool execution and the compositional integration of tool outputs.

Episodic Summarizer Finally, an episodic Summarizer LLM continuously condenses the evolving conversational context shared between the Reasoning Agent and the Tool Agent. This strategy has been shown to effectively improve the agents' reasoning ability in several prior studies [10, 11, 16]. During the agents' reasoning process, the lightweight language model LLaMA 8B [1] incrementally organizes the accumulated reasoning history into discrete logical steps. Such steps may include, for instance, tool generation, tool execution, or intermediate planning phases. Each step typically comprises multiple prompts and intermediate reasoning traces, which are consolidated into a single concise summary once the step is completed.

4.2 Workflows

In this subsection, we focus on the core principles of our workflow. Specifically, we detail the process of task planning, tool generation, tool execution and artifact handling.

Task Planning and Delegation Throughout the process, the Reasoning Agent maintains a dynamic plan aimed at fulfilling its overarching goal. The plan contains the overarching goal, the relevant substeps required to complete the overarching goal as well as a dynamic memory used to store information, facts and other data gathered throughout the process. The Reasoning Agent can update the plan at any time during the process, whenever it deems this as necessary. To facilitate the update of the plan and delegation of individual tasks, the Reasoning Agent can use the commands `<UPDATE_PLAN>` and `<DELEGATE>`, which will trigger individual prompt sequences to fulfill this request. As of now, tasks can only be delegated to the Tool Agent, as no other agents are available. However, the principle can be extended to using several additional agents. This would allow the delegation of tasks to most appropriate agents and enable parallel work.

Tool Generation The process of Tool Generation is initiated by the Tool Agent whenever it decides that a new tool is required to fulfill a specific task. The agent will trigger the process by streaming a special `<CREATE_TOOL>` command. The agent will then be asked to provide a tool specification outlining name, purpose, input parameters and expected output of the tool. The specification template is illustrated in (Listing 1). It is forwarded to the Coding Agent, which will generate the source code of the tool based on the specification.

Tools are validated based on five metrics:

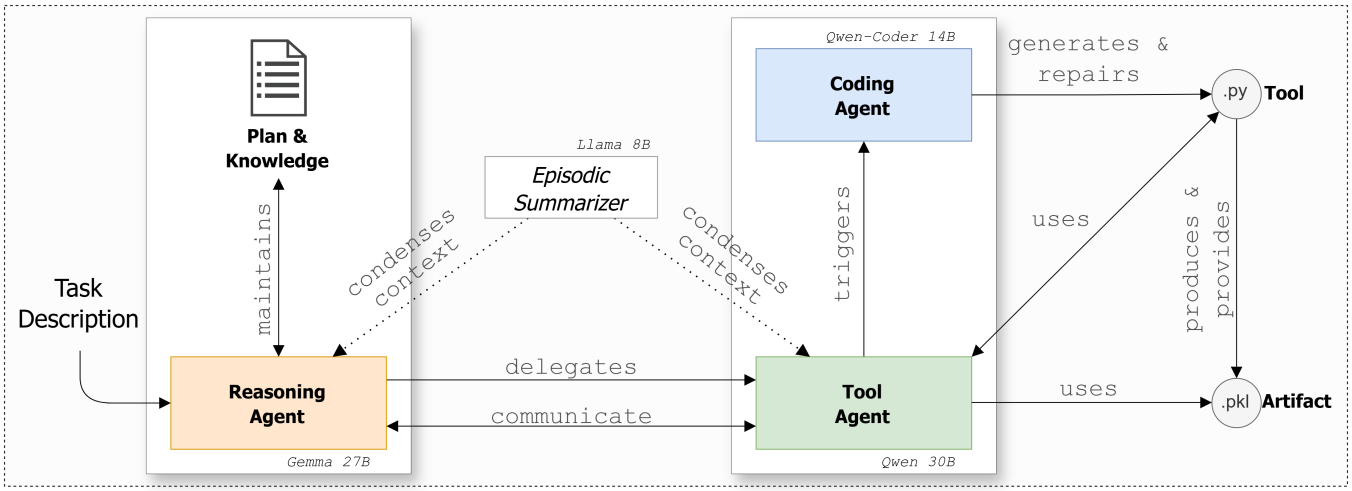

Figure 1: Overview of our framework architecture. The system consists of two main agents: Reasoning Agent and the Tool Agent. A Coding Agent generates tools as directed by the Tool Agent. Executed tools produce persistent artifacts for later use. An episodic summarizer LLM condenses each agent's reasoning history after logical steps.

```
{
     "name" : "<tool name>",
     "description" : "<tool description>",
     "parameters" : {
        "<parameter name>": {
             "type" : "<datatype>",
              "description" : "<param. description>",
              "constraints" : "<constraints>"
        }
    },
     "expectedOutput" : "<expected output>"
}
```

Listing 1: Tool specification format

(1) **Syntax Validation:** Ensures that the generated tool constitutes syntactically valid Python code by attempting compilation without execution.
(2) **Entry-Point Validation:** Verifies the presence of a designated `main()` function, which serves as the required entry point for tool execution.
(3) **Import Validation:** Analyzes all import statements to detect potentially unsafe modules. Additionally, this step ensures that all imported dependencies are available in the execution environment.
(4) **Executability Validation:** Checks whether the tool can be executed without raising runtime errors by running the code in an isolated namespace; for example, tools that trigger exceptions such as division-by-zero or unresolved attribute accesses during execution are rejected.
(5) **Call-Signature Validation:** Validates function calls to imported modules by resolving their call paths and verifying that the provided arguments conform to the corresponding function signatures; for instance, calls supplying an incorrect number of arguments or invalid keyword parameters are flagged.

Once the tool is validated, it is added to a central registry. Here it remains available and the Tool Agent can use it in subsequent steps as many times as needed. Listing 2 (`inspect_csv_files`) and Listing 3 (`calculate_closest_approach`) illustrate two examples of tools that were generated within the process of our Motivating Example (Section 3). Specifically, `inspect_csv_files` allows the agent to inspect the structure of the .csv file containing the location measurements, which is a required step to understand how the data can be processed. The tool `calculate_closest_approach` allows the agent to calculate the trajectories of the two objects based on the previously calculated position and velocity vectors. In our use case, the tools were created by the `Qwen-Coder 14B` LLM of the Coding Agent.

```
import csv
def main(file_paths):
    result = {}
    for file_path in file_paths:
        with open(file_path, mode='r', newline='',
     encoding='utf-8') as csvfile:
            reader = csv.reader(csvfile)
            headers = next(reader)
            result[file_path] = headers
    return result
```

Listing 2: Example of a Generated Tool: inspect_csv_files

```
import numpy as np
def main(trajectory_params):
    # Extract trajectory parameters
    # The keys in trajectory_params are the file paths,
     so we need to identify which is satellite and which
     is debris
    # For this implementation, we'll assume the first two
      keys are satellite and debris respectively
    keys = list(trajectory_params.keys())
```

```
    satellite_position = np.array(trajectory_params[keys
     [0]]['position'])
    satellite_velocity = np.array(trajectory_params[keys
     [0]]['velocity'])

    debris_position = np.array(trajectory_params[keys
     [1]]['position'])
    debris_velocity = np.array(trajectory_params[keys
     [1]]['velocity'])

    # Calculate relative position and velocity
    relative_position = satellite_position -
     debris_position
    relative_velocity = satellite_velocity -
     debris_velocity

    # Calculate the time of closest approach using the
     formula:
    # t = -(r0 . v) / (v . v)
    r0_dot_v = np.dot(relative_position,
     relative_velocity)
    v_dot_v = np.dot(relative_velocity, relative_velocity
     )

    if v_dot_v == 0:
        raise ValueError("Relative velocity is zero,
     closest approach cannot be determined.")

    t_closest_approach = -r0_dot_v / v_dot_v

    # Calculate the position at the time of closest
     approach
    satellite_position_ca = satellite_position +
     satellite_velocity * t_closest_approach
    debris_position_ca = debris_position +
     debris_velocity * t_closest_approach

    # Calculate the distance at the time of closest
     approach
    distance_ca = np.linalg.norm(satellite_position_ca -
     debris_position_ca)

    # Return the result as a dictionary
    return {
        'closest_approach_time': t_closest_approach,
        'minimum_distance': distance_ca
    }
```

Listing 3: Example of a Generated Tool: calculate_closest_approach

Tool Execution Similar to tool generation, tool execution is initiated by the Tool Agent via a dedicated <EXECUTE_TOOL> command. This command triggers the tool invocation process, whose detailed process is illustrated in Figure 2. Broadly, it can be decomposed into three logical stages:

(1) **Parameter Specification:** The Tool Agent first identifies the tool to be executed and provides a brief reason for the invocation, which is later used to generate a summary of the output. The agent then supplies the required input parameters, which may either be specified explicitly (e.g., strings or numerical values) or derived from the outputs of previously executed tools by referencing their associated artifacts.

(2) **Tool Execution and Output Handling:** The selected tool is executed with the provided parameters. Each tool is required to expose a single `main()` function as its sole entry point, simplifying and standardizing the execution procedure. If the execution was successful, the output is directly stored as an artifact containing the raw result, data type, handle, size, and - if the raw output is excessively large or cannot be serialized (e.g., complex objects such as database clients) - a concise verbal description generated by a lightweight language model (LLaMA 8B [1]).

(3) **Error Handling and Recovery:** If tool execution fails, the Coding Agent analyzes the failure to determine whether it is likely caused by errors in the tool's source code or by invalid input parameters. Based on this assessment, the agent proposes either a code-level repair or a revision of the input parameters. In cases where the source code can be repaired, the corrected version is executed automatically; otherwise, responsibility for adjusting the input parameters is delegated back to the Tool Agent.

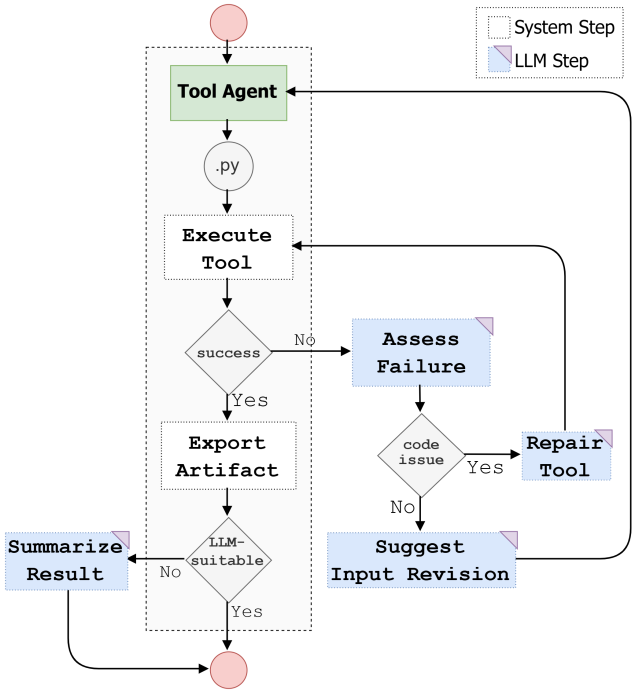

Figure 2: Process of Tool Execution

Artifact Handling By enabling the creation of virtually arbitrary tools, tool outputs may assume unanticipated forms and data types. In general, large language models (LLMs) are limited to processing serializable, text-based inputs; when working with predefined tools, this constraint can typically be enforced by design.

However, once agents are empowered to generate tools dynamically, the resulting output types become inherently unpredictable and are often incompatible with direct consumption by an LLM. For example, a database client or figure object are incompatible with LLMs, as they are not text-based. While larger models may exhibit an implicit awareness of these limitations and could consequently construct tools to output only compatible outputs, smaller language models lack the ability to anticipate whether a tool's output will

```
"artifact_id": "artifact_475",
"summary": "Tool result of tool
    calculate_closest_approach",
"context": "Result stored as artifact (dict)",
"content": {
    "closest_approach_time": 11.4285,
    "minimum_distance": 21.3808
},
"handle": "/artifacts/artifact_475.pkl"
```

Listing 4: Artifact Representation Created from Tool Output (Decimal values rounded to 4 decimal places for readability)

be produced in a consumable format. To address this challenge, our system introduces an explicit artifact abstraction that mediates between tool execution and agent reasoning. Tool outputs are encapsulated into serializable artifacts with a semi-fixed structure, consisting of the raw tool result, its data type and size, a handle for later reference, and a concise verbal description generated by a lightweight language model. Whenever the tool output itself is serializable, it is forwarded directly to the Tool Agent; only in cases where the output cannot be serialized do we fall back to the verbalized description, which provides a high-level semantic summary without exposing an incompatible internal representation. Listing 4 showcases the representation of artifact 475, representing the output of the tool `calculate_closest_approach` (Listing 3). The raw output (`content`) is a small dictionary containing the TCA and PCA of the two objects. As this output format is directly consumable by the Tool Agent, no additional textual description is required here. For LLM-incompatible outputs, the constructed verbal description cannot describe the raw content of the output. Instead, it can describe its metadata and purpose. For example, in case of a database client, the description might state the type of client and how it can be used in other tools.

Overall, our approach departs from the conventional paradigm in which agents directly operate on tool outputs. Instead, we adopt an artifact-centric interaction model in which agents reason over structured representations of results rather than raw outputs themselves. This shift decouples agent reasoning from the concrete data representations produced by tools, enabling robust handling of heterogeneous and non-serializable outputs while preserving composability across sequential tool invocations.

5 DISCUSSION

5.1 Advantages of Dynamic Tool Generation

The core contribution of this work is a dynamic tool generation process, autonomously triggered by LLM-based agents. The primary value of this framework is not necessarily to produce code thatis more computationally efficient than human-written code, but to produce code that is adapted to unseen data schemas and scenarios. In the motivating example, the generated tool automatically ignored the irrelevant 'mass' column in the CSV file. A pre-defined human tool might have requird explicit error handling for unexpected columns. Therefore, the distinction lies in rovustness to novelty rather than raw performance. This approach allows agents to overcome two key limitations of tool-based LLMs.

First, the LLMs remain restricted to the tools that were provided to them. Hence, they cannot dynamically adapt them to new or evolving situations. It is infeasible to anticipate every possible scenario in advance and create mechanisms for each one that the LLM can use. On one hand, even minor deviations from nominal use cases - such as adapting a maneuver-planning tool to different propulsion models - would require explicitly predefined tools for each variation. On the other hand, entirely new problem classes may emerge, such as the sudden need to analyze an unprecedented orbital regime or a previously unconsidered debris interaction scenario.

Second, when an LLM faces a situation requiring more advanced (e.g. mathematical) reasoning, it is forced to perform those calculations within its own internal framework, as long as no tool exists for that specific purpose. While modern LLMs have made significant advancements in reasoning, these types of calculations are inherently prone to error. Unlike specialized tools that perform precise computations, LLMs do not actually "calculate" but instead "reason" through them, leading to potential hallucinations or inaccuracies. Figure 3 illustrates this imprecise nature of LLMs when presented with calculations. We prompted the LLMs individually with the task of calculating the TCA and PCA based on the same provided measurements. Their calculations are often approximate, but not precise. This example illustrates one of the major benefits of tool-generating LLMs: it enables them to offload critical computations to external, reliable systems, thus ensuring more accurate results and reducing the cognitive load on the LLM itself.

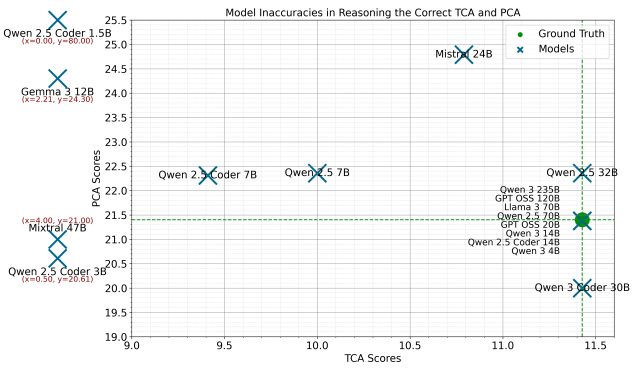

Figure 3: Model Inaccuracies in Reasoning the Correct TCA and PCA. We prompted all models with the same, neutral prompt instructing it to identify the TCA and PCA based on the measurements. The scatter plot displays their calculated TCA and PCA.

5.2 Limitations & Challenges

While the benefits of dynamic tool generation can fundamentally advance the capabilities of agentic systems, our research uncovers new challenges that have to be addressed.

Tool Output Handling. Among the most prominent hurdles of our system is the handling of tool outputs. For conventional approaches, tool outputs are well structured and can be easily formatted into a form that LLMs can interpret - typically serializable, formatted text. In our framework we encountered the challenge

of agents generating tools that produce outputs which are inherently incompatible with the LLMs text-based input requirements. Examples of such cases include tools that output database clients, pandas dataframes or image files. Further, output that is serializable but simply too large to be handled by LLMs efficiently (e.g. large HTML documents) are a highly impractical side-effect of dynamic tool-generation.

In our framework, we tackle this issue by separating the Tool Agent from the immediate consumption of the tool's output. After a tool is executed, we check whether the output is serializable and can be directly used by the agent. If that is not the case, we provide the agent with a handle to the artifact in which the original output is stored. It is pickled, stored to file, and can be reused as input to subsequent tool calls. Crucially, we employ a small LLM to generate a concise description of the output. This description is based on the tool itself, its output and the purpose of the call as provided by the Tool Agent at the time of execution, allowing the agent to understand and reason about the result without needing to process the full raw data. While this process can be cumbersome, it is a first step towards enabling agents to handle incompatible datatypes, an important requirement for explorative situations in which unexpected situations will occur.

Security of Arbitrary Code Execution. One of the more intuitive concerns is the issue of arbitrary code execution. While predefining tools restricts agents capabilities, it also provides a level of security and predictability. With dynamic tool generation, however, agents can, in theory, generate arbitrary code that is potentially harmful, whether done intentionally or by accident. In our implementation, we solely validate whether the tools are executable and scan for unsafe imports based on a minimal heuristic. However, once tool generation is unconstrained, agents could quickly produce code that is unpredictable, introduces subtle errors, or has unintended side effects. Managing these risks is not just a technical problem - it also requires a shift in perspective on autonomy and reliability. In many ways, it's similar to trusting human astronauts: rigorous training and screening reduce risk, but there is never absolute certainty that they will act perfectly. For autonomous agents, pretraining, safety checks, and validation can help, but some uncertainty will always remain. Oversight mechanisms could be added to monitor or constrain agent behavior, yet doing so limits the very flexibility and creativity that make dynamic tool generation valuable. And if the oversight itself becomes AI-driven, the same question remains: how can we ensure that the system enforcing trustworthiness follows its instructions? This layered dilemma underscores that enabling true autonomy in deep-space missions is as much an ethical challenge as it is a technical one.

Fidelity of Generated Tools and Physics Models In the motivating example, the generated tools utilized a linear approximation for trajectory propagation to demonstrate the framework's code generation capabilities. This approximation is valid only for extremely short horizons (seconds), where gravitational curvature is negligible, For operational collision avoidance, higher-fidelity models such as the Clohessy-Wiltshire-Hill (CWH) equations for close-proximity relative motion are required. This highlights a critical dependency: the frameworks output quality is bounded by the agents knowledge of such domain-specific physics. Future iterations will require to integrate domain-informed constraints or domain-specific libraries into the agents context to ensure the generated tools adhere to orbital mechanics standards suitable for flight.

5.3 Implications for Space Missions

Artificial Intelligence (AI) is already being adopted into the space domain. NASA uses AI to enable robots like the *Perseverance* [5] rover to make real-time navigation decisions without Earth's interventions. *CIMON* [6], an interactive assistant for astronauts, was deployed in the Columbus module of the ISS during the Horizons mission [3] in 2018.

NASA explicitly states that autonomous operations are a key capability for deep-space missions, where live communication to and interventions from Earth is impossible [2, 4]. As a result, future space exploration will need to do more than simply follow predefined procedures - they must be able to reason, adapt, and respond to unexpected situations. Current AI systems onboard rovers and robotic assistants, while capable of sophisticated navigation and decision-making, still rely on fixed tools and pre-programmed responses. In contrast, a system based on large language models that can dynamically generate tools opens the door to true autonomous discovery: it could analyze sensor data, formulate hypotheses, and create new computational or operational tools on the fly, without human intervention. For example, when encountering unexpected sensor readings, the system could develop tools for data-filtering or clustering, aiming to identify anomalies in the system readings. This kind of adaptability is especially critical for deep-space missions to outer planets and beyond, where severe communication delays make real-time guidance from Earth impossible. By enabling onboard reasoning, flexible problem-solving, and autonomous tool creation, LLM-based systems could dramatically extend the scientific and operational reach of future missions, turning spacecrafts into explorers capable of handling challenges that humans on Earth cannot anticipate.

We envision the Multi-Agent System operating as an autonomy layer on top of the vehicle's standard systems. The Reasoning Agent does not directly access hardware. Instead, it accesses a shared backboard where subsystems (e.g. power, thermal, navigation, etc.) publish telemetry, In an operational scenario the agent acts proactively: it continuously monitors specific telemetry artifacts for threshold violations and upon detection, it triggers planning workflows.

6 CONCLUSION

We presented a framework enabling autonomous multi-agent systems to dynamically generate and execute Python tools at runtime, addressing deep space communication constraints. Our approach connects high-level reasoning with environment interaction by creating task-specific tools on demand. Central to our approach is separating reasoning from computation. Agents delegate numerical operations to tools, ensuring precise trajectory estimation while the LLM focuses on orchestration. In our space debris example, agents autonomously generated tools for trajectory analysis and closest-approach estimation without predefined instructions. Future work will address challenges in tool output handling, security and integration with spacecraft autonomy concepts. As missions

move further from Earth, dynamic tool generation offers a path to handle unfamiliar situations without predefined functionality.

REFERENCES

[1] AI@Meta. 2024. Llama 3 Model Card. (2024). https://github.com/meta-llama/llama3/blob/main/MODEL_CARD.md

[2] Molly Anderson and Julia Badger. 2024. Gateway Autonomy for Enabling Deep Space Exploration. In *2024 IEEE Aerospace Conference*. 1–7. https://doi.org/10.1109/AERO58975.2024.10521067

[3] Marius Bach and Dieter Sabath. 2018. Horizons Mission-Challenges and Highlights. (2018).

[4] Abigail Bowman. 2025. *Intelligent and Adaptive Systems.* https://www.nasa.gov/ames/core-area-of-expertise-intelligent-and-adaptive-systems/?utm_source=chatgpt.com

[5] Abigail Bowman. 2026. *Artificial Intelligence - Safely using advanced AI tools to support missions and research projects across the agency.* https://www.nasa.gov/artificial-intelligence/

[6] German Aerospace Center (DLR). 2018. *CIMON - the intelligent astronaut assistant.* https://www.dlr.de/en/latest/news/2018/1/20180302_cimon-the-intelligent-astronaut-assistant_26307

[7] Mohd Ariful Haque, Justin Williams, Sunzida Siddique, Md. Hujaifa Islam, Hasmot Ali, Kishor Datta Gupta, and Roy George. 2025. Advanced Tool Learning and Selection System (ATLASS): A Closed-Loop Framework Using LLM. In *2025 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, 64–73. https://doi.org/10.1109/SOSE67019.2025.00012

[8] Xu Huang, Junwu Chen, Yuxing Fei, Zhuohan Li, Philippe Schwaller, and Gerbrand Ceder. 2025. CASCADE: Cumulative Agentic Skill Creation through Autonomous Development and Evolution. arXiv:2512.23880 [cs.AI] https://arxiv.org/abs/2512.23880

[9] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. 2024. Qwen2. 5-Coder Technical Report. *arXiv preprint arXiv:2409.12186* (2024).

[10] Xiaoxi Li, Wenxiang Jiao, Jiarui Jin, Guanting Dong, Jiajie Jin, Yinuo Wang, Hao Wang, Yutao Zhu, Ji-Rong Wen, Yuan Lu, and Zhicheng Dou. [n.d.]. DeepAgent: A General Reasoning Agent with Scalable Toolsets. https://doi.org/10.48550/arXiv.2510.21618

[11] Yuan Li, Yixuan Zhang, and Lichao Sun. [n.d.]. MetaAgents: Simulating Interactions of Human Behaviors for LLM-based Task-oriented Coordination via Collaborative Generative Agents. http://arxiv.org/pdf/2310.06500v1

[12] ESA Space Debris Office. 2025. *ESA's Annual Space Environment Report.* Technical Report. European Space Agency (ESA).

[13] Gemma Team. 2025. Gemma 3. (2025). https://goo.gle/Gemma3Report

[14] Qwen Team. 2025. Qwen3 Technical Report. arXiv:2505.09388 [cs.CL] https://arxiv.org/abs/2505.09388

[15] Qwen Team. 2025. QwQ-32B: Embracing the Power of Reinforcement Learning. https://qwenlm.github.io/blog/qwq-32b/

[16] Qingyue Wang, Yanhe Fu, Yanan Cao, Shuai Wang, Zhiliang Tian, and Liang Ding. 2025. Recursively summarizing enables long-term dialogue memory in large language models. *Neurocomputing* 639 (2025), 130193. https://doi.org/10.1016/j.neucom.2025.130193

[17] Georg Wölflein, Dyke Ferber, Daniel Truhn, Ognjen Arandjelovic, and Jakob Nikolas Kather. 2025. LLM Agents Making Agent Tools. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (Eds.). Association for Computational Linguistics, Stroudsburg, PA, USA, 26092–26130. https://doi.org/10.18653/v1/2025.acl-long.1266

[18] Andrea Zollo, Cristina Parigini, Roberto Armellin, Juan Felix San Juan Diaz, Annarita Trombetta, and Ralph Kahle. 2026. A polynomial-based Monte Carlo approach for estimating long-term collision probabilities. *Acta Astronautica* 242 (2026), 178–192.

A MOTIVATING EXAMPLE WALKTHROUGH

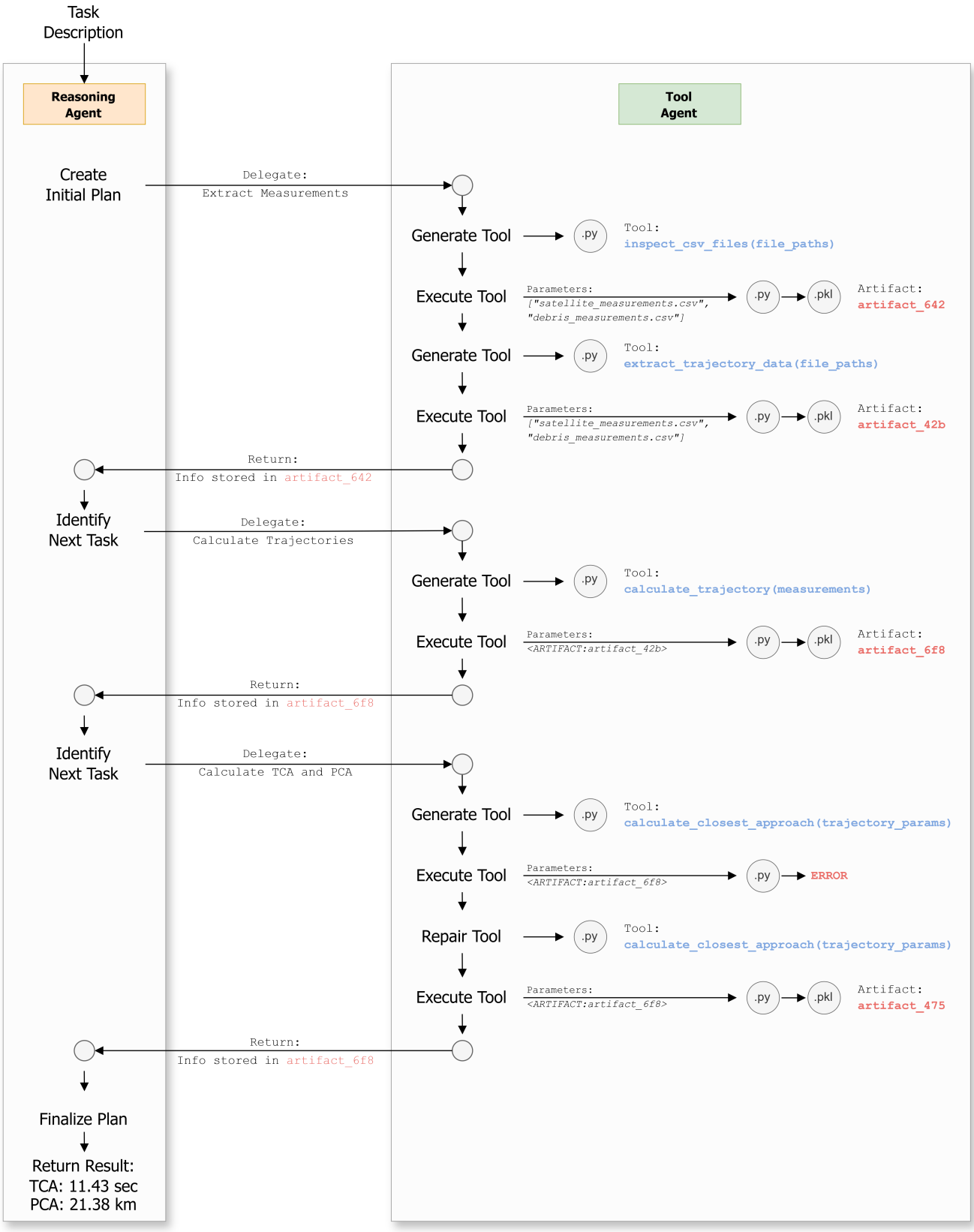

Figure 4: Walkthrough of Collision Assessment