# Python and AI/ML for Weather, Climate and Environmental Applications

Let us enjoy playing with Python and AI/ML!

**Five-Day Schedule Overview**

| Time | Day 1 | Day 2 | Day 3 | Day 4 | Day 5 |
|---|---|---|---|---|---|
| 09:00–10:00 | Opening by ECMWF DG, Start: Coding & Science in the Age of AI | Neural Network Architectures | Diffusion and Graph Networks | MLOps Foundations | Model Emulation, AIFS and AICON |
| 10:00–11:00 | **Lab:** Python Startup: Basics | **Lab:** Feed-forward and Graph NNs | **Lab:** Graph Learning with PyTorch | **Lab:** Containers and Reproducibility | **Lab:** Emulation Case Studies |
| 11:00–12:00 | Python, Jupyter and APIs | Large Language Models | Agents and Coding with LLMs | CI/CD for Machine Learning | AI-based Data Assimilation |
| 12:00–12:45 | **Lab:** Work environments, Python everywhere | **Lab:** Simple Transformer and LLM Use | **Lab:** Agent Frameworks | **Lab:** CI/CD Pipelines | **Lab:** Graph-based Assimilation |
| 12:45–13:30 | | | **Lunch Break** | | |
| 13:30–14:30 | Visualising Fields and Observations | Retrieval-Augmented Generation (RAG) | DAWID System and Feature Detection | Anemoi: AI-based Weather Modelling | AI and Physics |
| 14:30–15:30 | **Lab:** GRIB, NetCDF and Obs Visualisation | **Lab:** RAG Pipeline | **Lab:** DAWID Exploration | **Lab:** Anemoi Training Pipeline | **Lab:** Physics-informed Neural Networks |
| 15:30–16:15 | Introduction to AI and Machine Learning | Multimodal Large Language Models | MLflow: Managing Experiments | **The AI Transformation** | Learning from Observations Only |
| 16:15–17:00 | **Lab:** Torch Tensors and First Neural Net | **Lab:** Radar, SAT and Multimodal Data | **Lab:** MLflow Hands-on | **Lab: How work style could change** | **Lab:** ORIGEN and Open Discussion |
| 17:00–20:00 | | | | Joint Dinner | |

## Why Function Calling Exists

- Pure text generation is insufficient
- Real systems need actions, not prose
- **To make the LLM productive, it needs a link to reality!**
- Let it call functions!

**Key idea**

*The model should decide what to do, the system should decide how to do it.*

**Goal:**

- separate reasoning from execution

```
{
  "tool_call": {
    "name": "<function name>",
    "arguments": { ... }
  }
}
```

Typical problems:

- fragile JSON parsing
- ambiguous intent
- mixed explanation and action

Function calling introduces:

- explicit actions
- typed arguments
- machine-readable decisions

## Tool Contracts and Schemas

A tool is defined by:

- name
- description
- input schema

Schemas specify:

- required arguments
- data types
- allowed structure

**Important:**

- tools are defined by the system
- never invented by the model

**Tool schema**

```
name: get_temperature
arguments:
  location: string
  leadtime: integer
```

The model learns:

- when this tool applies
- how to fill arguments

```
{
  "tool_call": {
    "name": "<function name>",
    "arguments": { ... }
  }
}
```

## Tool Calls in Model Output

Modern LLMs can output:

- normal assistant text
- structured tool calls

A tool call contains:

- tool name
- arguments
- no natural language

This decision is:

- made by the model
- enforced by the API

**LLM output**

```
tool_call:
  name: get_temperature
  arguments:
    location: "Berlin"
    leadtime: 24
```

No parsing of prose required.

```
try:
    tool_call = json.loads(raw_output)["tool_call"]
    print("Parsed tool call:")
    print("Tool name:", tool_call["name"])
    print("Arguments:", tool_call["arguments"])
except Exception as e:
    print("Failed to parse JSON output")
    print("Error:", e)
```

## Streaming and Open-Source Models

In streaming APIs:

- tool calls appear inside the stream
- mixed with text tokens

Open-source models (e.g. LLaMA):

- emit structured patterns
- still require system-side handling

Common fallback:

- detect JSON blocks
- interpret intent manually

**Historical context**

- Older Framework: JSON scanning
- Claude UI: similar hybrid behavior

Native tool calling reduces:

- ambiguity
- parsing complexity

## From JSON Parsing to Native Tool Calling

Old approach:

- prompt for JSON
- parse model output
- recover from errors

Modern approach:

- tools defined explicitly
- model selects tool
- system executes tool

**Key shift:**

- from text parsing to action selection

**Architectural consequence**

- cleaner agent design
- safer execution
- better testability

This enables:

- LangChain tools
- LangGraph nodes
- reliable agents

## From Scripts to Agents: Why AI Agents Exist

### Classical scripts and pipelines

- Deterministic execution
- Fixed control flow
- Explicit inputs and outputs

They work well if:

- tasks are fully specified
- all cases are known in advance

### Limitations

- No interpretation of intent
- Poor handling of ambiguity
- Fragile when requirements change

### Where this moves today

- Natural-language problem descriptions
- Large, evolving code bases
- Underspecified or incomplete tasks

From rigid pipelines to **adaptive systems**

# What Is an AI Agent? — Core Mental Model

## Agent as a closed loop

perception
→ reasoning (LLM)
→ action
→ environment / state

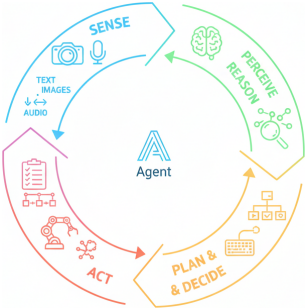
Key ingredients:

- internal state
- access to tools
- explicit control flow

An agent is *active*, not passive.

## LLM vs tool vs agent

- **LLM:** maps text to text
- **Tool:** executes a fixed function
- **Agent:** decides *what to do next*




Decision-making loop with state and actions

## LLM Capabilities and Limits in Practice

### What LLMs are good at

- Understanding natural language
- Generating plausible code and text
- Pattern completion and refactoring
- Explaining existing code

LLMs approximate

$$p(\text{next token} \mid \text{context})$$

from large training corpora.

### Systematic limitations

- No ground truth or verification
- Hallucinated but plausible outputs
- Overconfidence in incorrect answers
- Sensitivity to prompt phrasing

### Sampling matters

$$\text{temperature } T \uparrow \Rightarrow \text{ variance of outputs } \uparrow$$

Low $T$: reproducible but rigid
High $T$: creative but unstable

## Prompting for Code Generation

- Unambiguous task description
- Explicit constraints
- Predictable output format

**Critical constraints**

- Code-only output
- No explanations or markdown
- Explicit library choices

Poor prompts lead to:

- mixed prose and code
- missing imports
- implicit assumptions

**Example: strict system prompt**

System message

```
1 You are an AI coder.
2 Output ONLY valid Python code.
3 No markdown. No explanations.
4 Assume Python 3.10.
```

**User prompt pattern**

User message

```
1 Write a Python function f(x)
2 using numpy that returns a
3 polynomial with |f(x)| < 10
4 for x in [-10, 10].
```

**Key point:** Prompting is part of the program.

## Manual Coding with LLMs (Human in the Loop)

### Typical workflow

- Prompt LLM for code
- Inspect generated output
- Edit or correct manually
- Execute and test

### Why humans stay involved

- Detect logical errors
- Spot missing assumptions
- Judge correctness, not plausibility

LLMs assist, they do not decide.

### Trust boundaries

- Never execute blindly
- Always read generated code
- Treat LLM output as a draft

### Productive collaboration

- LLM: speed, syntax, structure
- Human: intent, validation, responsibility

**Key point:**

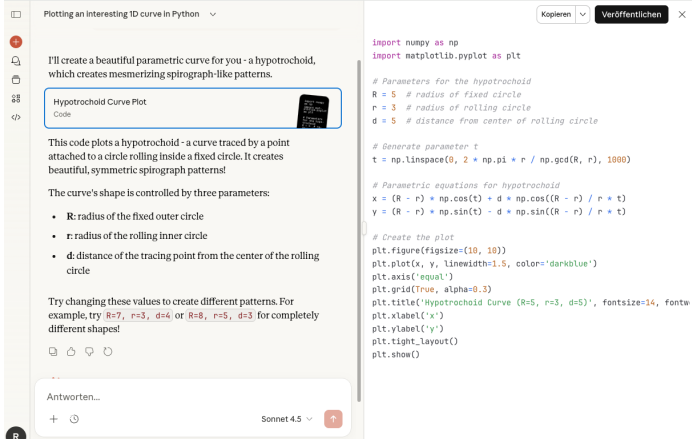
Responsibility stays with the human.

# Manual Coding with LLMs — Example UI Interaction

## Human in the loop

- Natural-language prompt
- Code suggestion by LLM
- Human inspection
- Manual correction

This is the baseline mode of working with LLMs.




Typical web-based LLM interface used for interactive coding

## Executing Generated Code Safely

### Execution options

- exec inside current process
- Separate process via subprocess
- File-based execution

### Trade-offs

- exec: fast, unsafe
- subprocess: isolated, slower
- Files: traceable, debuggable

### Rule of thumb

Never execute LLM code without isolation or inspection.

### Minimal example

File-based execution

```
1 with open("gen.py","w") as f:
2     f.write(code)
3
4 import subprocess
5 subprocess.run(
6   ["python", "gen.py"],
7   check=True)
```

### Security risks

- File system access
- Network calls
- Infinite loops

Mitigation requires process isolation.

## Error Handling and Feedback Loops

### Why errors are central

- LLM-generated code is often incomplete
- Small syntax or logic errors are common
- First attempt rarely works

Errors provide structured feedback:

- missing imports
- wrong assumptions
- invalid API usage

### Typical feedback loop

**Error capture and retry**

```
try:
    exec(code)
    success = True
except Exception as e:
    error = traceback.format_exc()
    success = False
```

### Key design choices

- feed back full traceback
- limit number of retries
- detect repeating failures

**Key point:** Errors drive self-correction.

## The First Coding Agent: Self-Correcting Loops

### Core idea

- LLM generates code
- Code is executed
- Errors are captured
- LLM is prompted to fix them

This creates an autonomous correction loop.

### Minimal success criteria

- Code executes without error
- Output matches basic expectations

### Minimal agent loop

Self-correcting loop

```
for attempt in range(max_tries):
  code = llm(prompt)
  try:
    exec(code)
    break
  except Exception as e:
    prompt += traceback.format_exc()
```

### Why this is already an agent

- autonomous retries
- internal state (prompt history)
- decision: retry vs stop

No framework required.

## From Prompts to Programs: Abstraction Boundaries

### The core problem

- Prompts mix intent and execution
- Small wording changes alter behavior
- Logic is implicit and fragile

This leads to prompt brittleness:

- hard to debug
- hard to reuse
- hard to test

### Separating responsibilities

- Prompt: what should be done
- Code: how it is executed
- Control flow: when to retry or stop

### Design principle

- Prompts describe intent
- Programs enforce structure

**Key point:** LLMs belong inside programs, not around them.

## Why Agent Frameworks Exist

### Scaling problems of ad-hoc agents

- Prompts grow uncontrollably
- Control logic becomes implicit
- Error handling is duplicated

As systems grow:

- code becomes unstructured
- behavior is hard to reproduce
- debugging is expensive

### What frameworks provide

- Explicit control flow
- Reusable abstractions
- Tool and memory interfaces
- Observability and logging

### What they do *not* provide

- Correct reasoning
- Ground truth
- Guaranteed success

**Key point:** Frameworks add structure, not intelligence.

## Survey of Agent Frameworks (Critical View)

### Why so many frameworks?

- **No standard agent abstraction**
- Rapidly evolving LLM APIs
- Different design philosophies

### Main categories

- Loop-based agents
- Chain-based frameworks
- Graph-based workflows
- Multi-agent orchestration

### Typical examples

- LangChain: chaining + tools
- LangGraph: explicit control flow
- CrewAI: role-based agents
- AutoGPT-style: autonomous loops

### Practical assessment

- Most are experimental
- APIs change quickly
- Production use needs caution

**Rule:** Use frameworks to clarify, not to hide logic.

## LangChain: Motivation and Architecture

### What LangChain targets

- Reusable prompt templates
- Standardized tool access
- Simple memory abstractions

LangChain focuses on:

- composition
- integration
- rapid prototyping

### Core building blocks

- LLM interface
- PromptTemplate
- Chain
- Tool
- Memory

### Limitation

- Linear execution model
- Limited explicit control flow

**Key point:** LangChain glues components, it does not control logic.

# Prompt Templates and Tool Integration (LangChain)

- Separates instructions and variables
- Enforces a fixed structure
- Improves reuse and testing

**PromptTemplate example**

```
from langchain.prompts
    import PromptTemplate

prompt = PromptTemplate(
  input_variables=["x"],
  template=
  "Write Python code
    computing f(x) "
  "such that |f(x)| < 10." )
```

## Tool integration

**Tool definition**

```
from langchain.tools import tool

@tool
def square(x: float) -> float:
    return x * x
```

## Critical limitations

- LLM may select wrong tool
- Arguments may be malformed

**Rule:** Tool calls must be validated outside the LLM.

## Autonomous Coding Agent — Self-Correcting Loop

### Agent task

- Natural-language goal
- LLM generates full script
- Script is executed
- Errors are fed back automatically

This is already a true agent:

- autonomous retries
- internal state (prompt history)
- stop criterion

**Autonomous code agent (excerpt)**

```
def autonomous_code_agent(task):
  for attempt in range(5):
    code = llm.invoke(task).content
    try:
      exec(code)
      return True
    except Exception as e:
      task += traceback.format_exc()
  return False
```

**Key point:** No framework required — this is pure control logic around an LLM.

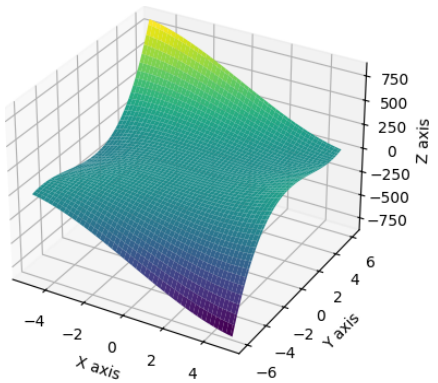
## Autonomous Agent Output — 3D Polynomial Visualization

### Task executed

- Generate 2D polynomial
- Evaluate on grid
- Create 3D visualization
- Save result as image

### Important

- Code was generated
- Code was executed
- Output was validated




3D surface plot generated by an autonomous LLM coding agent

## Memory and Context Management in Agents

### Why agents need memory

- Remember past attempts
- Accumulate errors and feedback
- **Maintain task continuity**

Without memory:

- repeated failures
- no learning across steps
- brittle behavior

### Types of memory

- Prompt memory (growing instruction and error history)
- File-based memory (saved code, plots, logs)
- Explicit state (variables passed between steps)

### Key distinction

- Chat history != agent state
- State must be explicit and inspectable

## Memory Management in LangGraph

- No hidden chat history
- No implicit conversation memory
- No prompt accumulation

Instead:

- All memory lives in a typed state object
- Each node reads and updates this state
- State is passed explicitly between nodes

**Consequence:**

- Deterministic execution
- Fully inspectable memory
- Reproducible agent behavior

**LangGraph state**

```python
class MyState(TypedDict):
    query: str
    fc_datetime: str
    fc_reference_datetime: str
    fc_leadtime: str
    fc_location_of_interest: str
    fc_variable: str
    temperature_data: Any
    output: str
```

In LangGraph, chat history, tool memory, and agent scratchpads are replaced by a single typed state object. Memory is explicit data, not implicit text.

## Control Flow in LangGraph

- No hidden agent loops
- No implicit retries
- No LLM-driven control decisions

Instead:

- Execution follows a directed graph
- Nodes are pure Python functions
- Edges define allowed transitions

**LangGraph control flow**

```
builder.set_entry_point("
    extract_forecast_datetime")
builder.add_edge(
  "extract_forecast_datetime",
  "get_latest_forecast_reference_time
   ")
builder.add_edge(
  "get_latest_forecast_reference_time
   ",
  "calculate_lead_time" )
[...]
builder.set_finish_point("
    plot_temperature")
```

The graph itself defines what happens next. There is no hidden agent controller.

## LLMs as Nodes in LangGraph

- LLMs do *not* control execution
- LLMs do *not* manage memory
- LLMs do *not* decide termination

Instead:

- LLMs are used for local reasoning tasks
- Each call has a well-defined input
- Each call produces a bounded output

**Typical LLM roles:**

- information extraction
- classification
- summarization

LangGraph node

```python
def extr_loc_node(state: MyState) ->
    MyState:
  location = extr_loc(state["query"])
  state["fc_location"] = location
  return state
```

The LLM acts as a pure transformation from input fields to output fields.

# LangGraph Weather Forecast Assistant

```
[3]: dawid.ai("Please give me the temperature forecast for Monday 4pm.")
```

*extract_forecast_datetime: 2026-01-01 16*

*get_latest_forecast_reference_time: 2025-12-31 00*

*calculate_forecast_lead_time: 40.0*

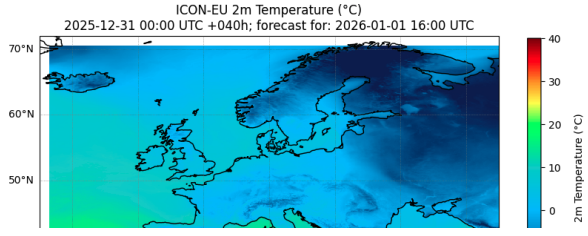
*extract_location: Location: Not specified*

*extract_variable: Temperature*

*Downloading: https://opendata.dwd.de/weather/nwp/icon-eu/grib/00/t_2m/icon-eu_europe_regular-lat-lon_single-level_2025123100_040_T_2M.grib2.bz2*

*Plot saved as temperature_forecast.png*

I've created and displayed the temperature forecast for Monday at 4 PM. If you have a specific location in mind, feel free to let me know for more detailed information!




End-to-end agent pipeline: natural language → structured state → tools → visualization

## Failure Handling and Robustness

- LLM extraction errors
- Missing or delayed data
- Tool execution failures
- Invalid intermediate state

**LangGraph strategy**
Each node handles local failure
State records partial results
Graph execution remains controlled

### Key principle

- Failures are data, not crashes

**Failure-aware node**

```python
def plot_temperature_node(state: MyState)
    -> MyState:
  if state.get("temperature_data") is None:
    state["output"] = "No data available"
    return state

  plot_t2m_EU(state["temperature_data"],
    save_plot=True)
  state["output"] = "Plot created"
  return state
```

Failures do not break the agent. They update the state and allow the graph to terminate safely.

## Why This Scales: From Prototype to System

### Problems with classic agent loops

- hidden prompt growth
- non-reproducible behavior
- difficult debugging
- unclear failure causes

### LangGraph advantages

- explicit state evolution
- deterministic control flow
- inspectable intermediate results
- testable nodes

### Engineering outcome

- agents become systems
- not demos or experiments

### What becomes possible

- unit tests per node
- regression tests on state
- logging and metrics
- CI/CD integration

### Key insight

*Agents scale when they obey the same rules as software.*

## When Multi-Agent Systems Make Sense

### One agent is sufficient when

- tasks are sequential
- state is compact
- logic is well-defined
- tools dominate execution

### Multiple agents are useful when

- responsibilities are clearly separable
- different reasoning styles are needed
- tasks can proceed independently
- software components can be done independently

### Key warning

- multi-agent systems are not better by default

### Typical multi-agent roles

- planner / coordinator
- **domain expert**
- **tool executor**
- verifier or critic

### Design principle

*Add agents only when you can explain their responsibility.*

## CrewAI: Role-Based Agent Collaboration

### What CrewAI provides

- explicit agent roles
- task delegation
- simple coordination logic
- readable high-level structure

### Typical use cases

- document analysis
- research workflows
- report generation
- exploratory automation

### Strength

- fast prototyping of multi-agent ideas

### Limitations

- implicit memory handling
- limited state visibility
- weak failure control
- hard to test systematically

### Key takeaway

*CrewAI is useful for coordination demos, not for operating critical systems.*

## CrewAI Demo: Internal Achievements Report

### Scenario

- 3–4 months of AI-related achievements
- Research and software development
- Target: internal newsletter or ministry

### Agent roles

- Planner: report structure
- Scientific writer: technical accuracy
- Impact translator: non-expert framing
- Editor: clarity and consistency

### Notebook flow

- raw bullet-point inputs
- sequential task execution
- agent-to-agent refinement
- final Markdown output

### What this demonstrates

- realistic knowledge work
- role separation
- human-facing automation

## AI Agent Landscape — January 2026

### What has stabilized

- **LLMs as reasoning engines**
- Tool calling as standard interface
- Explicit state and control flow
- Strong separation of roles

### What is fading

- prompt-only agents
- hidden scratchpads
- uncontrolled self-loops
- purely conversational systems

### Dominant design patterns

- Graph-based agents (LangGraph)
- Tool-driven execution
- **Typed, inspectable state**
- Human-in-the-loop checkpoints

### Reality check

- Agents are software systems
- Not autonomous intelligence
- Require engineering discipline