

GeoPandas-AI: A Smart Class Bringing LLM as Stateful AI Code Assistant

Gaspard Merten
Data Science and Engineering Lab,
Université libre de Bruxelles
Brussels, Belgium
gaspard.merten@ulb.be

Gilles Dejaegere
Data Science and Engineering Lab,
Université libre de Bruxelles
Brussels, Belgium
gilles.dejaegere@ulb.be

Mahmoud Sakr
Data Science and Engineering Lab,
Université libre de Bruxelles
Brussels, Belgium
mahmoud.sakr@ulb.be

Abstract

Geospatial data analysis plays a crucial role in tackling intricate societal challenges such as urban planning and climate modeling. However, employing tools like GeoPandas, a prominent Python library for geospatial data manipulation, necessitates expertise in complex domain-specific syntax and workflows. GeoPandas-AI addresses this gap by integrating LLMs directly into the GeoPandas workflow, transforming the `GeoDataFrame` class into an intelligent, stateful class for both data analysis and geospatial code development. This paper formalizes the design of such a smart class and provides an open-source implementation of GeoPandas-AI in PyPI package manager. Through its innovative combination of conversational interfaces and stateful exploitation of LLMs for code generation and data analysis, GeoPandas-AI introduces a new paradigm for code-copilots and instantiates it for geospatial development.

CCS Concepts

• **Information systems** → **Geographic information systems**; **Data analytics**; • **Software and its engineering** → **Software development techniques**.

Keywords

GeoPandas, LLM, Stateful Code Copilot

1 Introduction and System Overview

Geospatial data analysis has become central to addressing complex societal challenges, from urban planning to climate modeling. However, leveraging tools like GeoPandas — a widely used Python library for geospatial data manipulation — requires mastering domain-specific syntax and workflows. GeoPandas integrates several Python libraries, acting as a core component in the Python geospatial ecosystem. Its power for building geospatial solutions comes at the cost of a steep learning curve: developers must navigate several packages, interface the data across them, and learn intricate function parameters and syntax to build their data science pipelines. This complexity often forces practitioners to rely on fragmented documentation, coding blogs, or trial-and-error approaches, slowing down software development.

Recent advances in Large Language Models (LLMs) have demonstrated remarkable potential for code generation, enabling natural language queries to be translated into executable scripts. Tools like GitHub Copilot and ChatGPT excel at assisting developers by generating code or explaining logic. Further, many LLM models now natively support data analysis. One could upload their data and obtain summaries and insightful visualizations.

GeoPandas-AI takes advantage of the LLM code generation capabilities by embedding it directly into the GeoPandas workflow. It extends the `GeoDataFrame` into a *smart class*, capable of interacting with an LLM to generate executable code and data analysis tailored to user-defined tasks. GeoPandas-AI is intended to be used by Python programmers as a complement, not a replacement for, the popular geospatial data processing library GeoPandas. It makes GeoPandas `DataFrame` conversational, allowing developers to describe a Python function in natural language or ask data analytics questions about the `GeoDataFrames` and get back the code to perform this task. For example, you can ask GeoPandas-AI to cluster the nearby features in a `GeoDataFrame` and filter out small clusters, and it will produce the code for this and return a `GeoDataFrame` containing the clustering results.

GeoPandas-AI positions itself as an intelligent code co-pilot, yet it distinguishes from existing approaches as illustrated in Table 1. GeoPandas-AI occupies a novel position in the landscape of AI-assisted programming tools. In the top-left cell of the table, we find syntax-level code assistants such as GitHub Copilot. These tools provide support based on syntactic patterns and have access to the overall project code-base. Still, they do not consider runtime information, such as the actual schema or content of a `GeoDataFrame`. However, in geospatial programming, the choice and application of operations are tightly coupled with the structure and semantics of the data. For example, the set of applicable geometric operations varies significantly depending on whether the geometry column contains points or polygons. To compensate for this limitation, users must manually convey detailed knowledge about the `GeoDataFrame`'s schema and content to the assistant. This task becomes increasingly complex as the schema evolves or interacts with other variables, such as during join operations with other `GeoDataFrames`. Further, some tools speak to the needs of data analysts, who lack programming skills but understand the data and the business domain well, represented in the bottom row of Table 1. Contemporary LLM-based chatbots (e.g., ChatGPT) can ingest data files and generate meaningful insights without requiring deep programming expertise. This has enabled such data analysts to perform exploratory data analysis through conversational prompts, often referred to as *vibe coding* (bottom-right cell in the table). Furthermore, zero-code platforms like LIDA analyze the content of data sources and interactively suggest tailored analytical workflows to non-programming users. GeoPandas-AI bridges these two paradigms by supporting software developers while maintaining a tightly integrated understanding of code semantics, enabling developers to work more efficiently.

A typical user interaction consists of initializing a `GeoDataFrameAI` object, then an arbitrary long *conversation* with it,

and ending with code generation. This process is illustrated in Listing 1. In this example, the programmer instantiates a `GeoDataFrameAI` from an existing `GeoDataFrame` and performs a simple conversation to plot the data. In the background, the `GeoDataFrameAI` will communicate with the AI Service to produce the desired code. The programmer will instantiate the `GeoDataFrameAI` and call the first `chat()` function, yielding an initial code and result. The programmer will then inspect the results produced and decide to indicate to the AI Service to add a legend by using the `improve()` function. Once satisfied with the result, the programmer calls `inject()`, which creates a new Python file including the member function `plot_network()` with the last code instance, making it available for reuse and further manual edits. It should also be understood that such a snippet of code is produced incrementally, i.e., after writing each `chat()` or `improve()` operation, the user checks the result before deciding whether further `improve()` operations are necessary. Once satisfied with the results, the user can continue with the `inject()` procedure.

```
1 gdfai = gpdai.GeoDataFrameAI(stib_gdf, "
    GeoDataFrame for the network of public
    transport operator in Brussels.")
2 gdfai.chat("Plot the network")
3 gdfai.improve("add a legend")
4 gdfai.inject("plot_network")
```

Listing 1: Running example for the use of GeoPandas-AI.

The novelty of GeoPandas-AI stems from its design, which integrates conversational interfaces directly encapsulated into the `GeoDataFrame` class. Unlike most AI code generators or assistants that operate externally within the development environment, GeoPandas-AI is executed natively in Python as part of the data structure itself. Further, GeoPandas-AI is a stateful assistant, allowing an incremental refinement of the generated code based on the history of past interactions. To the best of the author’s knowledge, this model for code generation and data analysis has not existed in the literature.

User Type	Syntax-Level Assistance	Semantic Understanding
Software Developer	GitHub Copilot, Cursor, Duet AI, Tabnine	GeoPandas-AI
Data Analyst	Chat-GPT, Gemini, Claude, Mistral AI	LIDA, Tableau Einstein, ThoughtSpot, etc.

Table 1: Landscape of AI code assistants.

Several research papers and studies have attempted to formalize or derive design principles for AI-powered code generators and copilots, especially focusing on human-AI interaction, developer experience, and tool usability. These principles are informed by empirical findings from user studies, field observations, and design evaluations of existing tools such as GitHub Copilot, Amazon CodeWhisperer, and other machine learning-based programming assistants. Table 2 summarizes the key design principles distilled from this body of work.

GeoPandas-AI brings unique benefits with respect to the first three principles in Table 2. For context awareness, GeoPandas-AI

Principle	Description
Context Awareness [17]	Must understand syntax, variable state, and context.
Low Cognitive Load [4, 17]	Suggestions should be intuitive and require minimal mental effort.
Privacy [8]	Respect code ownership, and sensitive data handling.
Feedback Loops [10, 14]	Enable developers to accept, reject, or refine suggestions.
Seamless Integration [14]	Must fit naturally into the development flows.

Table 2: Key Design Principles for AI-Powered Code Generators and Copilots

goes beyond basic syntax by grounding the LLM in automated metadata (such as column types and coordinate reference systems) and user-provided context. This contextual information is encoded in natural language descriptions (e.g., “This dataset contains OpenStreetMap parks in Berlin with accessibility features”) and structured into domain-specific knowledge, such as the dataset’s intended use (e.g., urban mobility), constraints (e.g., “exclude private parks”), and spatial properties. By providing this dual layer of context, GeoPandas-AI ensures that generated code adheres not only to geospatial best practices but also to the analyst’s specific intent. This rich contextual grounding naturally supports the principle of low cognitive load. As shown in our use case examples in Section 8, users often receive accurate and usable code on the first interaction. Furthermore, the system mitigates common LLM pitfalls such as hallucinated function names or invalid projections by validating outputs against the `DataFrame`’s schema and spatial constraints. For example, if a dataset uses a local coordinate system, GeoPandas-AI ensures distance calculations respect that CRS rather than defaulting to inaccurate latitude-longitude approximations. Regarding privacy, GeoPandas-AI offers a more secure approach than other tools in Table 1. Unlike AI copilots that may expose full source code to the LLM, GeoPandas-AI offers the possibility to the user to share, in addition to its prompt, only the `DataFrame`’s metadata and an eventual data sample. Compared to chatbots and general-purpose automated analytics tools, it gives full control regarding the data and metadata shared with the AI Service. This selective exposure significantly reduces legal and privacy risks associated with AI-assisted coding.

However, two key design principles remain challenging in implementing GeoPandas-AI: feedback loops and seamless integration into development workflows. These lead to the following technical challenges:

- **Supporting feedback loops** requires stateful interactions between the user and the GeoPandas-AI object. This demands a formal memory model to track conversation states and transitions, along with an intuitive API for users to manage and refine their interactions over time.
- **Seamless integration** into Python workflows presents another challenge. Since GeoPandas-AI is implemented as a class, its integration must occur through member functions. The design of these functions must balance minimalism with flexibility, supporting both

script-based and notebook-style Python development environments while maintaining ease of use.

- **Ensuring consistency across runs** is complicated by the stochastic nature of LLMs. Even with the same input, outputs can vary depending on factors like temperature settings. Yet, users expect consistent results when rerunning prompts during a conversation till reaching the code generation. Maintaining reliable behavior across multiple executions is thus a critical concern.

- **Geospatial code generation** poses an additional challenge. While LLMs perform well on general programming tasks, they are less proficient in niche domains such as geospatial analysis. Although fine-tuned models and domain-specific foundation models have emerged in the geospatial community [5, 21], GeoPandas-AI focuses specifically on GeoPandas code generation and its ecosystem. To bridge this gap, we augment the LLM using domain-specific RAG and compiled curated examples tailored to GeoPandas workflows.

Accordingly, the main contributions in this paper are: (1) A design for a smart class mitigating the above challenges. Although out of the scope of this paper, one could consider this design as a template for developing other smart classes, e.g., for raster processing, for audio processing, etc, and (2) an open source implementation of the GeoPandas-AI class available in PyPi¹.

The GeoPandas-AI architecture, which addresses the above challenges, is illustrated in Figure 1. It consists of two main components: the GeoPandas-AI library and an LLM backend, also denoted AI service. The GeoPandas-AI library is the main component. It consists of a new `GeoDataFrameAI` class and a caching system. The class is divided into two parts. The first part, denoted *State and Interface*, is the component that the programmer will use and interact with. It defines the public functions and variables of `GeoDataFrameAI` class. The other part of the class, denoted *Internal and Services*, contains the different components and logic not exposed to the programmer. These are necessary to communicate with the AI Service and create the desired code. The second component of the ecosystem is its LLM backend, also denoted the AI Service. This AI Service can be any system capable of producing code from a prompt. This definition includes RAG-powered LLMs, fine-tuned LLMs, etc. One key responsibility of the AI Service is to deal with discrepancies to minimize the effect of hallucination, inconsistency, wrong answers, etc.

The rest of this paper is organized as follows. `GeoDataFrameAI` is detailed in Sections 3 and 4, with Section 3 formally presenting its exposed interface and its state evolution model while Section 4 presents how the `GeoDataFrameAI` class functions internally: how it communicates with and uses the AI Service to generate the desired code. Section 5 motivates the need for caching and how it is implemented in the GeoPandas-AI library. The AI Service and how it is specialized in generating geospatial Python code are detailed in Section 6. Then, the main use case scenarios are presented in Section 7. Some examples of practical usage of GeoPandas-AI are provided in Section 8. Finally, Section 9 concludes this work and provides avenues of future research.

2 Related Works

LLMs have also proven capable of generating high-quality source code. Further, specialized LLM models optimized for code generation tasks have emerged, such as CodeGemma[15] from Google and Codstral² from Mistral AI. A comprehensive survey by Jiang et al. examined numerous Code LLMs, which are models specifically fine-tuned for translating natural language into source code [7]. Recent studies further benchmark these models across various tasks and programming languages [11, 19].

Although web-based conversational interfaces like ChatGPT.com are widely used by developers [9], they lack the deep integration expected from modern IDEs. This gap has led to the rise of syntax-level assistants embedded within development environments. As summarized in Table 1, such tools provide context-aware code completion and suggestions based on static analysis of the codebase. These tools increasingly adopt agent-like behaviors [20], enabling them to perform complex multistep tasks with minimal human intervention. However, a key limitation remains: these tools operate outside the runtime environment and are thus unaware of the runtime context.

Numerous tools have been specifically designed to support data scientists by leveraging large language models (LLMs) for tasks such as data exploration and visualization. For example, Wang et al. proposed an interactive online visualization platform that employs LLMs to streamline data analysis workflows [16]. These tools span both academic research systems, such as LIDA [3], and commercial solutions like Tableau Einstein and ThoughtSpot (see Table 1).

Geospatial data requires specialized domain knowledge. As such, specialized LLMs have been proposed. For instance, Li et al. proposed Geo-Llama [12], a fine-tuned LLM for the generation of trajectories. In [18], Xue et al. propose an overview of recent efforts and challenges in the exploitation of LLMs for smarter mobility systems. These, however, are not designed to produce code. To solve this problem, researchers have also proposed both dedicated LLMs and integrated tools. GeoCode-GPT [5] is an LLM designed specifically for geospatial code generation. On the tooling side, GeoGPT [21] provides a framework that allows users to interact with various GIS tools through a single natural language interface powered by LLMs. While these two LLM models have been trained/fine-tuned on the specifics of Geospatial, the part related to the GeoPandas ecosystem is limited.

3 Initialization and State Evolution of GeoDataFrameAI Objects

A key challenge in inserting GeoPandas-AI into the geospatial developer’s programming workflow is to make its front-end user-friendly and well-integrated. To address this, we introduce a minimal extension of the `GeoDataFrame` class, `GeoDataFrameAI`. This `GeoDataFrameAI` smart class is the entry point for GeoPandas-AI enabling developers to interact with an AI Service through natural language.

Objects of this `GeoDataFrameAI` class are defined by a state, denoted S , which is defined as a tuple:

$$S = \langle G, M_G, \mathcal{H}, \mathcal{T}, \mathcal{R} \rangle \quad (1)$$

¹<https://pypi.org/project/geopandas-ai/>

²<https://mistral.ai/news/codestral>

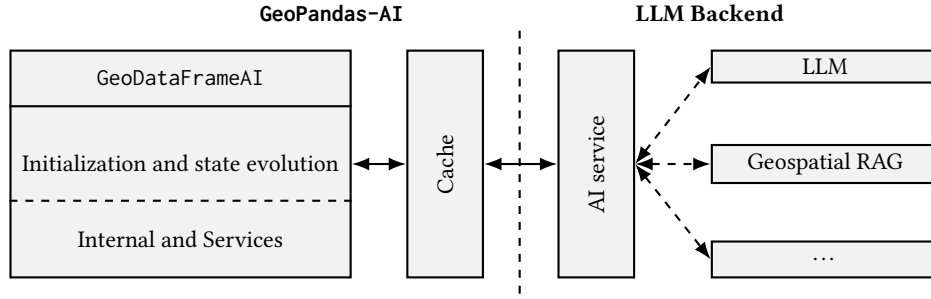


Figure 1: Architecture representation of GeoPandas-AI.

where:

- G : A classical input GeoDataFrame.
- M_G : Metadata derived from G both automatically (e.g., schema, summary statistics, spatial reference), from a user input description of G as well as potentially from other GeoDataFrameAI objects.
- \mathcal{H} : History of prior interactions, i.e., user queries and generated code.
- \mathcal{T} : *Tool State* — A set of permissible Python packages that the model is allowed to invoke in the generated code. Initially $\mathcal{T} = \mathcal{T}^0 = \{\text{contextily, pandas, matplotlib, folium, geopandas}\}$.
- \mathcal{R} : *Return-Type Set* — A set of possible response types. Initially $\mathcal{R} = \mathcal{R}^0 = \{\text{int, float, str, bool, list, dict, geopandas.GeoDataFrame, pandas.DataFrame, folium.Map, matplotlib.Figure}\}$.

In the rest of this text, we will denote $G, M_G, \mathcal{H}, \mathcal{T}, \mathcal{R}$ as variables. These can be split into three categories. G and M_G enable the environment to maintain awareness of the nature of the dataset and its structure, while \mathcal{H} allows the environment to maintain awareness of the user’s workflow history. Finally, \mathcal{T} and \mathcal{R} restrict how the AI Service can operate.

The remainder of this section describes the initialization of GeoDataFrameAI and the state evolution during user interactions, while Section 4 presents how the class functions internally.

3.1 Initialization of a GeoDataFrameAI Object

To create an initial GeoDataFrameAI object, a constructor can be called with a GeoDataFrame G and an optional user-provided natural language description d . This constructor is defined as:

$$\text{GeoDataFrameAI}(G, d) \rightarrow \mathcal{S}^0 = \langle G, M_G^0, \emptyset, \mathcal{T}^0, \mathcal{R}^0 \rangle \quad (2)$$

This is illustrated on the first line of the running example (Listing 1).

At this stage:

$$M_G^0 = (A_G, d, \mathcal{L}^0) = (A_G, d, \emptyset) \quad (3)$$

with A_G and \mathcal{L}^0 representing automated metadata respectively extracted from G , and to be extracted from other GeoDataFrameAI objects (see Section 3.2). The automated metadata \mathcal{A}_G is derived directly from G without user intervention. It includes:

- **Schema**: Column names, data types (e.g., string, numeric), and constraints (e.g., “column ‘population’ is non-negative”).
- **Statistical Summaries**: For numeric columns, this includes measures such as mean, variance, minimum, and maximum values.
- **Spatial Properties**: For the geometry attribute, this captures geometric and coordinate information, such as the Coordinate Reference System (CRS), the geometry type, the spatial extent of the dataset, etc.

Valorizing the LLM’s natural language understanding capabilities, d is kept unprocessed and unstructured — in natural language as given by the user. For instance, a user might provide the description:

“This dataset shows all parks in Brussels, sourced from OpenStreetMap, and includes details like their names, sizes, and accessibility features”. Nevertheless, it distills the input into three broad themes to guide the model behavior:

- **The domain** refers to the field or topic the data belongs to. If the description mentions “parks in Brussels” the AI recognizes this as urban planning or environmental science. This helps it prioritize, for instance, visualization styles relevant to that area.
- **The purpose** is what the dataset is meant to achieve. For instance, if the purpose is real-time monitoring, the AI uses this to align its responses with goals, such as dynamic visualizations or live data updates.
- **The constraints** or rules inferred from your description. If the user notes that data comes from “OpenStreetMap” or “only includes parks larger than 1 acre,” the AI understands these boundaries. This prevents errors, such as suggesting tools incompatible with open-source datasets or ignoring size filters.

While \mathcal{L}^0 is still an empty set at this stage, the variable \mathcal{L} is designed to contain a list of other GeoDataFrameAI objects to be used by the AI Service. This is useful whenever the programmer wants to analyze multiple DataFrames or perform a join operation.

We will denote \mathcal{S}^0 as the initial state.

3.2 State Evolution Trough AI Assisted Development

In order to converse with the GeoDataFrameAI object, we design a set of functions. The possible workflows in a chatting session are illustrated in Figure 2. Once a GeoDataFrameAI object is constructed, two functions allow the invocation of the AI Service:

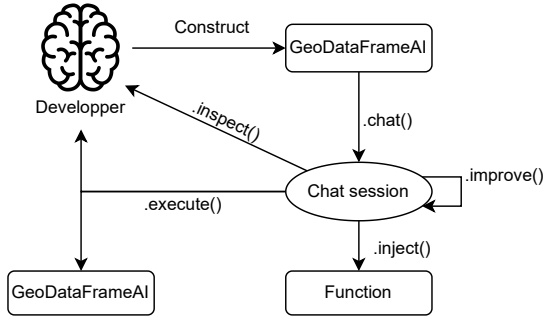


Figure 2: Possible GeoPandas-AI development work flows.

`chat()` and `improve()`. A call to either of these functions results in the `GeoDataFrameAI` interacting with the AI Service and generating some code. This code can then be:

- displayed to the programmer with the `inspect()` function.
- executed with the `execute()` function. The result, which could be a new `GeoDataFrameAI` object, a leaflet map, etc, is then returned to the developer.
- saved into a new Python function with the `inject().()` function.

The default behavior is to always execute the generated code. This behavior can, however, be overwritten as shown in Section 4.3. The functions are formally defined hereunder:

`chat()`. The `chat()` function is used to initialize or reinitialize a conversation. The `chat()` method of an `GeoDataFrameAI` object must receive a user query, denoted q^1 , representing a request. An example is provided in the second row of the running example (Listing 1), where q^1 is “Plot the network”. The `chat()` function also accepts some optional parameters. These are a tool set, a return type set as well as a list of other `GeoDataFrameAI`s, respectively denoted \mathcal{T} , \mathcal{R} and \mathcal{L} . By indicating the optional parameters in between square brackets, the `chat()` function is formally defined by:

$$S^i.\text{chat}(q^1, [\mathcal{T}, \mathcal{R}, \mathcal{L}]) \rightarrow S^1 \quad (4)$$

The index i being an integer greater than or equal to zero, as `chat()` can be called on the initial state or any other state. The transformation is applied in multiple steps as follows:

- (1) The state is reset to S^0 .
- (2) If \mathcal{T} is provided then \mathcal{T}^1 is set to \mathcal{T} , otherwise to \mathcal{T}^0 . Similarly for \mathcal{R}^1 and \mathcal{L}^1 .
- (3) M_G^1 is set to (A_G, d, \mathcal{L}^1) .
- (4) $G, M_G^1, \mathcal{T}^1, \mathcal{R}^1$ and q^1 are used by the AI Service to generate the code c^1 . Details about the process used to generate the code can be found in Section 4.2.
- (5) The history of interaction \mathcal{H}^1 is set to $\langle \{q^1, c^1\} \rangle$.

After these successive transformations, the state S^1 is entirely defined by:

$$S^1 = \langle G, M_G^1, \mathcal{H}^1, \mathcal{T}^1, \mathcal{R}^1 \rangle \quad (5)$$

The default behavior of `chat()` is to automatically execute the last generated code, i.e., by calling the `execute()` function.

`execute()`. The `execute()` function is defined as follows:

$$S^i.\text{execute()} \rightarrow o^i \quad (6)$$

with i being a strictly positive integer, which is equal to 1 when the function is called directly after the `chat()` function. The function returns o^i , which is the result of the execution of the code c^i on the `GeoDataFrame` G (using eventually the other `GeoDataFrames` provided in \mathcal{L}^i). In generating c^i , the AI is instructed that the type of o^i must be contained in \mathcal{R}^i . In cases where $|\mathcal{R}^i| > 1$, the exact return type of o^i is determined by the AI Service using the queries present in \mathcal{H}^i . If this type is the `GeoDataFrame` class, then o^i will be a newly generated `GeoDataFrameAI` object in an initial state:

$$o^i = S'^0 = \langle G', M_G^0, \mathcal{H}^0, \mathcal{T}^0, \mathcal{R}^0 \rangle \quad (7)$$

With G' being the `GeoDataFrame` generated by the code c^i . It is important to note that $o^i = S'^0$ is a new object. It is potentially a new `GeoDataFrameAI` based on a modified `DataFrame` (G'), however, the state S^i is not modified by the execution.

`inspect()`. The `inspect()` function allows users to check the generated code history at any time in the conversation. It is thus defined as follows:

$$S^i.\text{inspect()} \rightarrow \mathcal{H}^i \quad (8)$$

`improve()`. As long as the user is not satisfied with the results (either by inspecting or executing the code), they may refine the output proposed by the AI Service. The `improve()` function must receive a user query q^i . This user query should not contain a new request (such as for the `chat()` function) but rather some indication guiding the AI Service into producing better code. An example is provided in the second row of the running example (Listing 1), where q^i (in this case q^2) requires the AI Service to add a legend to its previously generated plot. With this function, the user seeks to guide the model towards adjusting the generated code to their specific intent. Such as `chat()`, `improve()` also accepts a tool set, a return type set, or a list of other `GeoDataFrameAI`s as optional parameters:

$$S^{i-1}.\text{improve}(q^i, [\mathcal{T}, \mathcal{R}, \mathcal{L}]) \rightarrow S^i \quad (9)$$

with i being the strictly positive index corresponding to the iteration of the `improve()` call.

The `improve()` function works similarly to `chat()`, with the difference that the state is not reset to the initial state:

- (1) If \mathcal{T} is provided then \mathcal{T}^i is set to \mathcal{T} , otherwise to \mathcal{T}^{i-1} . Similarly for \mathcal{R}^i and \mathcal{L}^i .
- (2) M_G^i is set to (A_G, d, \mathcal{L}^i) .
- (3) $G, M_G^i, \mathcal{T}^i, \mathcal{R}^i$ and q^i are used by the AI Service to generate the code c^i .
- (4) $\langle \{q^i, c^i\} \rangle$ are append to the history of interactions:

$$\mathcal{H}^i = \mathcal{H}^{i-1} \parallel \{q^i, c^i\} \quad (10)$$

Similar to the `chat()` function, the default behavior of GeoPandas-AI is to automatically invoke `execute()` after calling the `improve()` function.

inject(). Once the programmer is satisfied with the results, they can call the inject() function. This function writes the last generated code as a function in a local Python script, allowing further usage on different datasets. It takes a string parameter representing the desired name for the generated function:

$$S^i.\text{inject}(\text{function_name}) \quad (11)$$

4 Internal and Services

This section presents how the GeoDataFrameAI class operates internally, i.e., how GeoDataFrameAI communicates with the AI Service to generate code. These are a set of private functions in GeoPandas-AI that are not exposed to the user.

4.1 Templating

To interact with an LLM, it is necessary to transform the state S^i and arguments of both chat() and improve() into natural language. This is done using a lightweight templating format inspired by well-established work³. The template language is designed based on the JSON format and adheres to a fixed schema designed for simplicity and clarity. Each template consists of an ordered list of message objects, defined as follows:

```
1 {"messages": [{"role": "system" | "user",
2               "content": string}]}
```

Every object within the messages array contains two fields: role, indicating the speaker (either "system" or "user"), and content, a string representing the message text. The content string may also include specific patterns that indicate template variables to be injected during compilation. A simple example of a template with variables can be seen hereunder:

```
1 {"messages": [
2   {"role": "system",
3    "content": "You are a helpful coding
4      assistant, produce a code answering the
5      user prompt"},
6   {"role": "user",
7    "content": "Please generate a function def '
8      execute(args)' answering: {{prompt}}"}]
```

4.2 Code Generation

GeoPandas-AI relies on an AI Service to generate executable Python code from natural language instructions. This section details the process by which code is synthesized, validated, and optionally refined across the two core interfaces: chat() and improve(). The main steps to generate a code c^i based on the state S^{i-1} and the query q^i are explained next.

1. Resolution of the return type. When multiple return types are permitted (i.e., $|\mathcal{R}^i| > 1$), a dedicated determine_type() function is executed. This function queries the AI Service to define the most appropriate return type. This query generates a prompt using its associated template filled with \mathcal{R}^i and q^i and sends it to the AI

Service. The output is then parsed using a regex to obtain the desired output type (denoted r^i).

2. Generation of the code-generation prompt. To compile the prompt used to generate c^i , the templates for the chat() and improve() functions are filled with:

- r^i : the return type obtained at step 1;
- \mathcal{M}^i : the metadata about the GeoDataFrames (e.g., column types, CRS);
- \mathcal{T}^i : the currently allowed toolsets or libraries;
- \mathcal{R}^i : the currently expected return type;
- \mathcal{H}^i : the history of interactions (only for improve())

The details about how this information is encoded in natural languages are skipped here but can be found in the git repository of the project⁴. However, it should be mentioned that both templates also indicate to the LLM to generate a function of the form:

$$\text{def execute}(\text{df1}, [\text{df2}, \dots]) \rightarrow r^i \quad (12)$$

With df1 representing the GeoDataFrame G while [df2, ...] represent the other potentially linked GeoDataFrameAIs contained in \mathcal{L}^i (see Section 3.2). This ensures that the generated code conforms to a known signature, which is validated in subsequent steps.

3. Execution and error-driven retries. Once the prompt has been generated by filling in the template, it can be submitted to the LLM. A regular expression is used to extract the code snippet c^i from the generated answer. This also implicitly ensures that the c^i adheres to the requested function signature. This step serves to both constrain and verify the AI Service's output. The code is then executed either against synthetic data derived from the original GeoDataFrame G , or a small excerpt of G (see Section 4.3). This serves two goals:

- (1) Verify that the function runs without exceptions;
- (2) Ensure the result matches the expected return type;

If execution fails due to syntax errors, exceptions, or invalid return types, the system issues a new prompt using a retry template. This prompt includes the original code and the associated traceback, asking the AI Service to revise its response. The conversation history is augmented with each retry attempt, enabling successive refinement. This retry loop is attempted up to five times. If no valid result is obtained, the system raises an error containing the last code produced and halts the process.

4.3 Secure Communication Between GeoDataFrameAI and the AI Service

Since the smart GeoDataFrameAI class must execute code generated by AI services, it introduces several risks. These originate from the nature of such services: many operate in remote, opaque cloud environments, meaning that any data sent to them may be exposed to confidentiality breaches [6]. Even when AI runs locally, the code it produces may be unsafe. One study found 40% of GitHub Copilot-generated code to be vulnerable [13]. Malicious actors can also perform data poisoning, leading to a compromised training process and potentially harmful outputs [2]. Moreover, developers often lack visibility into what the AI is doing, making it difficult to trust and audit the interaction.

³Django <https://docs.djangoproject.com/en/5.2/ref/templates/language/>, Jinja2 <https://jinja.palletsprojects.com/>

⁴<https://github.com/GaspardMerten/geopandas-ai>

These issues impose three requirements for the front-end environment to mitigate these risks.

Data privacy. The content of the `GeoDataFrame` should not be sent directly to the AI service. This is to mitigate the confidentiality of the data from a legal perspective⁵ but also to prevent another type of data leakage.

Source code privacy. In contrast with systems such as GitHub Copilot, the source code in which GeoPandas-AI is involved should not, in any case, be transmitted to the AI service. There are several reasons to keep source code private: (1) it may contain secret keys, such as API credentials⁶; (2) it may include intellectual property; (3) it may embed trade secrets; and (4) it may play a role in security through obfuscation.

Execute code isolation. AI-generated code, whether due to hallucination or intentional manipulation, can pose risks. It may modify critical system files, execute harmful operations, or compromise the host operating system. Additionally, it can be exploited to bypass data protection mechanisms—for example, by issuing unauthorized HTTP requests to exfiltrate sensitive information. These risks highlight the need for strict code execution isolation, where AI-generated code runs in a sandboxed environment without access to the host OS or networks.

To address these three points, GeoPandas-AI implements the following communication framework with the AI Service. The developer can instantiate a `GeoDataFrameAI` from the `GeoDataFrame` G anywhere in their code. At this point, no code or request has been executed. Once the programmer starts a `chat()` interaction, the system will need to generate a unified textual description (UTD) of the `GeoDataFrameAI`. By default, an object of the class `PublicDescriptor` handles the generation of the UTD from the potential user-provided description, the metadata, as well as an excerpt of G . Users handling sensitive schema, which should not be sent to the AI Service, can build their own custom `Descriptor` class from the base abstract class `ADescriptor`. An object of this custom-made class can then be passed to the configuration of GeoPandas-AI which will then be used to generate the UTD. The user can also include, in this descriptor, artificial data generators to simulate the `GeoDataFrame` schema and content. This provides the user with the entire flexibility of determining what information should be sent to the AI Service from the `GeoDataFrame`. Once the UTD is transmitted to the AI Service the latter will generate an answer from which the code will be extracted. Executing it inline may lead to data exposure. Ideally, GeoPandas-AI could serialize the `DataFrames` and pass them to a fully isolated Docker or Firecracker instance, reset after each run. The results should then also be serialized back. This would, however, complicate the installation process and execution of GeoPandas-AI. Instead, two solutions are proposed to mitigate the execution of harmful Python code. The first one is introduced by the abstraction of the AI Service to GeoPandas-AI. This means that the user may use as an AI Service service any trusted LLM, potentially run locally. The second one consists of a `safe_mode` in

which no code is executed automatically. This allows the user to manually inspect generated code before executing it.

These steps allow for (1) ensuring data privacy, (2) preventing code-scope access, and (3) mitigating the execution of malicious code, hence solving issues currently faced by AI services.

5 Caching

GeoPandas-AI relies on Large Language Models (LLMs) for dynamic code generation. While this provides the necessary creativity and problem-solving capabilities, it also inherits several limitations of LLM-based systems:

- **Variability:** LLMs use stochastic sampling techniques, typically governed by a *temperature* parameter, to balance creativity and correctness. This randomness can lead to inconsistent outputs across identical prompts. In the context of GeoPandas-AI, this undermines reproducibility, especially when using chained calls like `chat()` followed by multiple `improve()` steps, which implicitly rely on deterministic intermediate outputs.
- **Latency:** Cloud-based or local LLMs often introduce noticeable response delays, which can accumulate across multiple queries, degrading the interactive experience and lengthening execution time.
- **Cost:** Each LLM query incurs computational and sometimes financial cost. As the metadata \mathcal{M} and history \mathcal{H} grow, so do the token lengths and associated inference time. This increases usage cost and contributes to the environmental footprint of repeated LLM queries.

To address these challenges, we introduce a caching mechanism bringing determinism to GeoPandas-AI and reducing redundant LLM inferences, which, in fine, lowers the latency and cost of GeoPandas-AI.

5.1 Caching Mechanism

Initially, when a user calls `chat()` or `improve()`, it triggers a transition from state S^{i-1} to S^i based on the instructions provided by the user. The main driver of this transition is the generation of c^i by the AI Service.

This generation step is the primary source of variability, latency, and cost. Hence, the proposed caching mechanism targets this part of the transition to improve efficiency.

We define the caching mechanism as the two usual functions `set(key, value)` and `get(key) → value`, respectively updating and retrieving the value (in our case c^i) from a persistent memory unit (e.g. file-system storage). To adapt this caching mechanism to GeoPandas-AI, we must store both the previous state S^{i-1} , as well as the various arguments, leading to the generation of c^i , which eventually allows transitioning to S^i . This is done using the function `buildStateKey`:

$$\text{buildStateKey}(S^{i-1}, q^i, \mathcal{T}^i, \mathcal{R}^i, c^i) \rightarrow k^i \quad (13)$$

The arguments are:

⁵<https://gdpr-info.eu/>

⁶For example, an API key to an online service

- (1) S^{i-1} : the previous state of the GeoDataFrameAI containing the entire history, including the previous sequence of queries and generated codes; $S^{i-1} \supseteq \{q^j, c^j \mid 1 \leq j \leq i\}^7$.
- (2) q^i : the prompt provided by the user to transition from S^{i-1} (or S^0 in the case of a `chat()` call) to S^i .
- (3) \mathcal{R}^i : the desired possible return types for c^i .
- (4) \mathcal{T}^i : the allowed tool set for c^i .

Cache Consistency. Because each key embeds the full sequence of prior queries and responses, which are contained in S^{i-1} , as well as the user instructions to go from S^{i-1} to S^i , any modification to an earlier prompt (e.g., changing q^{i-2}) yields a different key, ensuring cache consistency. This guarantees semantic consistency and avoids unintended reuse of outdated outputs.

Usage. During execution of `chat()` or `improve()`, GeoPandas-AI computes the corresponding key k^i . If $get(k^i)$ exists, the cached output c^i is returned directly. Otherwise, the AI Service is invoked to produce c^i , and the result is stored via `set(k^i, c^i)` for future reuse.

Limitations. The primary limitation of the current caching mechanism is the absence of an automatic or periodic cache invalidation policy, which could theoretically lead to unbounded persistent storage growth. However, this risk is mitigated by the typically small size of each cache entry c^i , which is expected not to exceed a few dozen lines of code. A practical upper bound can be estimated from the maximum generation capacity of the underlying model. For example, Gemini 2.0 models are limited to 8,192 tokens per output. Given that one token corresponds to approximately four characters, this yields an upper limit of 32,768 characters, or roughly 32 KB in UTF-8 encoding. As such, the storage footprint remains modest in typical use cases. Nevertheless, to manage caching explicitly, we provide two tools:

- (1) An instance method `reset_cache(chat_wise: boolean)` that clears the cache for a specific GeoDataFrameAI instance. The `chat_wise` parameter determines whether to delete only the current chat’s cache or all chats associated with that instance.
- (2) A global function `reset_cache()` that purges all GeoPandas-AI caches across the current environment.

6 LLM Backend

As outlined in the introduction, GeoPandas-AI exploits a default AI Service which has been further specialized for the generation of GeoPandas Python code. The GeoPandas-AI library, however, is backend-agnostic and may exploit any user-provided AI Service with various forms ranging from remote API calls to fully local language models. This flexibility is achieved through the use of the `litellm` Python package⁸, which serves as an abstraction layer over a wide array of LLM providers and configurations. As a result, GeoPandas-AI decouples its functionality from any specific vendor or model and supports diverse deployment scenarios.

While general-purpose LLMs are increasingly proficient at generating code, they often lack the domain expertise required to reason

about geospatial concepts such as projections, topology, or geometry validity. They may also struggle to use specialized tools like GeoPandas, Shapely, or Pyproj effectively. To evaluate the impact of backend design on the system’s performance and geospatial reliability, we experimented with three distinct configurations.

Raw LLM. Our baseline configuration leveraged the Gemini 2.0 Flash model⁹ via Google Cloud’s Vertex AI service. This setup demonstrated that general-purpose LLMs can produce syntactically valid code. However, it frequently failed to capture the user’s intent and often required multiple `improve()` iterations to converge to an acceptable solution. Additionally, geospatial subtleties such as CRS mismatches or geometry simplification were inconsistently addressed.

Fine-Tuned LLM. To enhance geospatial understanding, we fine-tuned the same Gemini model on a subset dataset of Stack Overflow answers[1]. This subset is constructed by filtering answers containing “geopandas” in Python. While the model preserved its ability to generate correct code, it did not significantly outperform the base model in terms of accuracy or concordance with user queries. Moreover, the fine-tuning process introduced a regression: the model lost its ability to handle non-code prompts (e.g., type disambiguation in `determine_type()`) due to overfitting on code-generation patterns. This necessitated maintaining two separate models: one for logic generation, another for other instructions, which increased system complexity.

RAG-enhanced LLM. In a third and more promising approach, we implemented a Retrieval-Augmented Generation (RAG) backend. This system indexed a high-quality corpus of geospatial programming examples sourced from permissively licensed datasets and documentation. At runtime, semantically similar examples were retrieved and embedded alongside the user query in the prompt sent to the LLM. This configuration markedly improved the system’s performance: it reduced the number of `improve()` calls needed, aligned better with domain best practices, and led to more robust, intention-aligned code generation.

7 Usage Scenarios

We envision two main usage scenarios for the use of GeoPandas-AI, namely data exploration and software development. In both scenarios, the user may benefit from a different workflow. GeoPandas-AI was designed to provide a smooth user experience in both cases.

Software development. When developing software, and in particular data pipelines, the programmer’s main concern will be the generation of elementary transformations to be used on input DataFrames. While the user will also need to perform a `chat()` operation for each transformation, their workflow will be characterized by a heavy usage of the `inspect()` and `improve()` operations. This workflow is facilitated by the global `safe_mode` parameter. In this case, the default behavior of GeoPandas-AI will be to print the generated code. The programmer will therefore need to call the `execute()` function explicitly. Once the programmer is satisfied

⁷We only use a subset of S^i in `buildStateKey`, excluding G , as it is already represented through \mathcal{M}_G^0

⁸<https://www.litellm.ai/>

⁹<https://cloud.google.com/vertex-ai/generative-ai/docs/models/gemini/2-0-flash>

with the code generated, they can save it into a local function by using the `inject()` function.

Data exploration. During data exploration, the user can benefit from GeoPandas-AI's ability to plot or query the DataFrames to obtain new insights on the data. These operations are characterized by an intensive usage of the `chat()` and `execute()` functions. In a Jupyter notebook environment, composed of multiple cells, the caching mechanism presented in Section 5 will allow the user to execute or re-execute the cell in a random order, and still receive a deterministic result despite the stochastic nature of the LLM. In order to facilitate the data exploration procedure, the default behavior of GeoPandas-AI is to perform automatically a call to `execute()` after every sequence of `chat()` and `improve()` which are in the same Python expression.

8 Usecase Example

In this section, the usage of GeoPandas-AI is illustrated by adapting a GeoPandas tutorial from HATARILABS¹⁰. This tutorial is about the analysis of flooded areas in the vicinity of the city of Boise. It involves plotting, joining and spatial analysis of multiple GeoPandas DataFrames. The reader can run the examples provided hereunder as they are available on the project's GitHub repository in a Jupyter notebook.

GeoDataFrameAIs initialization. First, the user needs to load the three data sources into GeoDataFrameAIs.

```
1 floodedAreas = gai.GeoDataFrameAI(floodedAreas_gdf)
2 highways = gai.GeoDataFrameAI(highways_gdf, description=
  "This dataset contains information about roads.")
3 facilities = gai.GeoDataFrameAI(facilities_gdf)
```

Analytical queries on GeoDataFrameAIs. The user can perform some simple analytical queries. The results produced by GeoPandas-AI are indicated in comments.

```
1 floodedAreas.chat("What are the the bounds of the flooded
  areas.")
2 # [-116.5164, 43.6623, -116.2989, 43.6948]
3 facilities.chat("Find unique amenities.")
4 # ['school', 'hospital', 'fire_station']
```

Join queries and plot on GeoDataFrameAIs. All datasets can be plotted individually with the following queries:

```
1 floodedAreas.chat("Plot the flooded areas.")
2 highways.chat("Plot the roads.").improve("Add a legend.
  Roads with the same name should have the same color.
  ").improve("Improve the legend, it does not fit in
  its box. Make it scrollable.")
3 facilities.chat("Plot the facilities.", return_type=
  Figure)
```

Thanks to the corpus of geospatial programming examples, the AI Service agent tends to plot the different GeoDataFrameAI using specialized libraries such as Folium Maps. For illustration purposes, we specified in line 3 to produce a matplotlib Figure. To produce a clearer map, line 2 needed further refinement using two `improve()` iterations. While it is also possible to obtain similar results using only one call to the `chat()` function by improving the query provided, this example showcases a real experiment, in which the user

first performs an initial request, and then performs the desired refinement based on the intermediary results. The AI Service, however, successfully generated a map of the roads with a scrollable legend, as required.

Often, the user may need to perform operations based on multiple GeoDataFrameAI objects. Two examples are provided hereunder.

```
1 floodedFacilities = facilities.chat(
2   "Add a Flooded column to the facilities based on
   whether they are in the flooded areas",
3   floodedAreas,
4 )
5 floodedFacilities.chat("Export to the Out/floodedSchools.
  gpkg. Keep only the facilities flooded.",
  return_type=None)
```

In this example, the user creates a new GeoDataFrameAI object based on the "facilities" but with an additional column based on the "floodedAreas" GeoDataFrameAI. He later uses this information to filter the GeoDataFrameAI to keep only flooded facilities and to export the result in a GeoPackage file.

```
1 highways.chat(
2   "Plot the highways the schools, and flooded areas.",
3   floodedAreas,
4   facilities,
5   return_type=Map).improve("Save in map.html file")
```

This example illustrates the generation and display of a map containing the data of all three GeoDataFrameAIs. This map is represented in Figure 3. Further, the user could save this map in an ".html" file for later usage.

Inspecting and injecting code. The user can inspect the history of generated code.

```
1 facilities.inspect()
```

Prompt 1: Add a Flooded column ...
Code 1:

```
import geopandas
import geopandas as gpd
```

```
def execute(df_1, df_2) -> GeoDataFrame:
    """Add a Flooded column to the ...

    :param df_1: GeoDataFrame containing facilities.
    :type df_1: geopandas.GeoDataFrame
    :param df_2: GeoDataFrame containing flooded areas.
    :type df_2: geopandas.GeoDataFrame
    :return: GeoDataFrame with a new 'Flooded' ...
    :rtype: geopandas.geodataframe.GeoDataFrame
    """
    df_1['Flooded'] = df_1.intersects(df_2.unary_union)
    return df_1
```

If the user is satisfied with the code and the result, they can materialize the operation in a function as follows:

```
1 facilities.inject("flooded")
```

which will print the following instructions:

Manual injection procedure...

First add, if not already present, the following import statement:

```
import ai
```

Then replace the following code with the function call:

¹⁰<https://hatarilabs.com/ih-en/introduction-to-python-and-geopandas-for-flooded-area-analysis-tutorial>

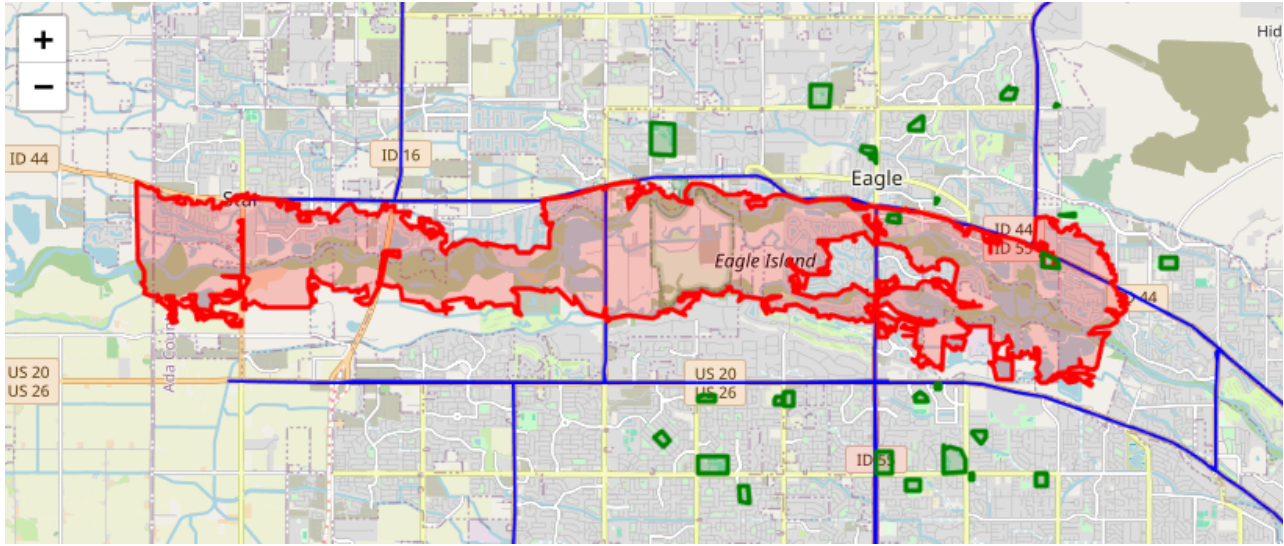


Figure 3: Folium Map of the three GeoDataFrameAIs. Flooded areas, facilities and highways are displayed in red, green and blue, respectively.

```
ai.flooded(gdf1, gdf2, ...)
Make sure to adjust the function call with the correct
parameters.
```

The function `flooded()` is now written locally and can be called again as follows:

```
1 import ai
2
3 ai.flooded(facilities_gdf, floodedAreas_gdf)
```

9 Conclusion

This paper introduces the GeoPandas-AI library, offering an innovative approach for the exploitation of LLM assistants in geospatial software development and data analysis. By extending the GeoDataFrame with a conversational interface, a concept that we call *smart class*, we demonstrated how the desired design principles of AI code copilots of context-awareness, low cognitive load, and privacy can be effectively addressed. Further, we also identified and tackled key technical challenges: supporting feedback loops through a stateful conversation model, achieving seamless integration into Python development environments, ensuring consistency across runs despite the stochastic nature of LLMs, and improving geospatial code generation through domain-specific augmentation techniques such as RAG and curated examples.

We formally defined an extension of the GeoDataFrame class with the support of a minimal set of functions enabling a user-friendly exploitation of an AI Service. As part of our contributions, we provide an open-source implementation of GeoPandas-AI available on PyPI. While our focus has been on GeoPandas, the underlying design principles can serve as a template for developing similar smart classes in other domains, such as raster processing, audio analysis, etc.

Future work includes enhancing the system’s robustness through fine-tuned geospatial code models, expanding support for more

complex geoprocessing tasks, and exploring deeper integration with existing geospatial toolchains and IDEs. Furthermore, an interesting further enhancement of GeoPandas-AI consists in integrating custom or existing artificial data generators, alleviating the burden for users with data sensitivity concerns. Ultimately, GeoPandas-AI represents a step toward more intuitive, collaborative, and domain-aware AI-assisted programming in the geospatial ecosystem.

Acknowledgments

The research leading to the results presented in this paper has received funding from the European Union’s funded projects MoBiSpaces under Grant No. 101070279 and EMERALDS under Grant No. 101093051.

This work was inspired by the open-source GitHub repository PANDAS-AI¹¹. While the repository lacks a formal design and theoretical foundation, it motivated the idea of extending the DataFrame with a chat interface. In this work, we have systematically developed and formalized this concept.

References

- [1] Loubna Ben Allal Anton Lozhkov, Raymond Li. 2024. StarCoder 2 and The Stack v2: The Next Generation. *arXiv:2402.19173* [cs.SE]
- [2] Domenico Cotroneo, Cristina Improta, Pietro Liguori, and Roberto Natella. 2024. Vulnerabilities in AI Code Generators: Exploring Targeted Data Poisoning Attacks. In *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension* (Lisbon, Portugal) (ICPC ’24). Association for Computing Machinery, New York, NY, USA, 280–292. doi:10.1145/3643916.3644416
- [3] Victor Dibia. 2023. LIDA: A tool for automatic generation of grammar-agnostic visualizations and infographics using large language models. *arXiv preprint arXiv:2303.02927* (2023).
- [4] Ebtesam Al Haque, Chris Brown, Thomas D. LaToza, and Brittany Johnson. 2025. Towards Decoding Developer Cognition in the Age of AI Assistants. *arXiv:2501.02684* [cs.HC] <https://arxiv.org/abs/2501.02684>

¹¹<https://github.com/sinaptik-ai/pandas-ai>

- [5] Shuyang Hou, Zhangxiao Shen, Anqi Zhao, Jianyuan Liang, Zhipeng Gui, Xuefeng Guan, Rui Li, and Huayi Wu. 2024. GeoCode-GPT: A Large Language Model for Geospatial Code Generation Tasks. *arXiv preprint arXiv:2410.17031* (2024).
- [6] Jessica Ji, Jenny Jun, Maggie Wu, and Rebecca Gelles. 2024. Cybersecurity Risks of AI-Generated Code. <https://doi.org/10.51593/2023CA010> Accessed June 4, 2025.
- [7] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A Survey on Large Language Models for Code Generation. *arXiv:2406.00515* [cs.CL] <https://arxiv.org/abs/2406.00515>
- [8] Georgia M. Kapitsaki. 2024. Generative AI for Code Generation: Software Reuse Implications. In *Reuse and Software Quality*, Achilleas Achilleos, Lidia Fuentes, and George Angelos Papadopoulos (Eds.). Springer Nature Switzerland, Cham, 37–47.
- [9] Ranim Khojah, Mazen Mohamad, Philipp Leitner, and Francisco Gomes de Oliveira Neto. 2024. Beyond code generation: An observational study of chatgpt usage in software engineering practice. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1819–1840.
- [10] Kim Tuyen Le and Artur Andrzejak. 2024. Rethinking AI code generation: a one-shot correction approach based on user feedback. *Automated Software Engineering* 31, 2 (2024), 60. doi:10.1007/s10515-024-00451-y
- [11] Jia Li, Ge Li, Xuanming Zhang, Yihong Dong, and Zhi Jin. 2024. EvoCodeBench: An Evolving Code Generation Benchmark Aligned with Real-World Code Repositories. *arXiv:2404.00599* [cs.CL] <https://arxiv.org/abs/2404.00599>
- [12] Siyu Li, Toan Tran, Haowen Lin, John Krumm, Cyrus Shahabi, Lingyi Zhao, Khurram Shafique, and Li Xiong. 2024. Geo-Llama: Leveraging LLMs for Human Mobility Trajectory Generation with Spatiotemporal Constraints. *arXiv preprint arXiv:2408.13918* (2024).
- [13] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2025. Asleep at the Keyboard? Assessing the Security of GitHub Copilot’s Code Contributions. *Commun. ACM* 68, 2 (Jan. 2025), 96–105. doi:10.1145/3610721
- [14] Qiusi Sun, Zhirui Chen, Fangzhi Xu, Kanzhi Cheng, Chang Ma, Zhangyue Yin, Jianing Wang, Chengcheng Han, Renyu Zhu, Shuai Yuan, Qipeng Guo, Xipeng Qiu, Pengcheng Yin, Xiaoli Li, Fei Yuan, Lingpeng Kong, Xiang Li, and Zhiyong Wu. 2025. A Survey of Neural Code Intelligence: Paradigms, Advances and Beyond. *arXiv:2403.14734* [cs.SE] <https://arxiv.org/abs/2403.14734>
- [15] CodeGemma Team, Heri Zhao, Jeffrey Hui, Joshua Howland, Nam Nguyen, Siqu Zuo, Andrea Hu, Christopher A. Choquette-Choo, Jingyue Shen, Joe Kelley, Kshitij Bansal, Luke Vilnis, Mateo Wirth, Paul Michel, Peter Choy, Pratik Joshi, Ravin Kumar, Sarmad Hashmi, Shubham Agrawal, Zhitao Gong, Jane Fine, Tris Warkentin, Ale Jakse Hartman, Bin Ni, Kathy Korevec, Kelly Schaefer, and Scott Huffman. 2024. CodeGemma: Open Code Models Based on Gemma. *arXiv:2406.11409* [cs.CL] <https://arxiv.org/abs/2406.11409>
- [16] Chenglong Wang, Bongshin Lee, Steven Drucker, Dan Marshall, and Jianfeng Gao. 2024. Data Formulator 2: Iteratively Creating Rich Visualizations with AI. (2024).
- [17] Ruotong Wang, Ruijia Cheng, Denae Ford, and Thomas Zimmermann. 2024. Investigating and Designing for Trust in AI-powered Code Generation Tools. In *Proceedings of the 2024 ACM Conference on Fairness, Accountability, and Transparency* (Rio de Janeiro, Brazil) (FAccT ’24). Association for Computing Machinery, New York, NY, USA, 1475–1493. doi:10.1145/3630106.3658984
- [18] Hao Xue, Ming Jin, Shirui Pan, and Flora Salim. 2025. Transforming Urban Dynamics: Harnessing Large Language Models for Smarter Mobility. *IEEE Intelligent Systems* 40, 2 (2025), 5–7.
- [19] Weixiang Yan, Haitian Liu, Yunkun Wang, Yunzhe Li, Qian Chen, Wen Wang, Tingyu Lin, Weishan Zhao, Li Zhu, Hari Sundaram, and Shuiguang Deng. 2024. CodeScope: An Execution-based Multilingual Multitask Multidimensional Benchmark for Evaluating LLMs on Code Understanding and Generation. *arXiv:2311.08588* [cs.CL] <https://arxiv.org/abs/2311.08588>
- [20] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*.
- [21] Yifan Zhang, Cheng Wei, Zhengting He, and Wenhao Yu. 2024. GeoGPT: An assistant for understanding and processing geospatial tasks. *International Journal of Applied Earth Observation and Geoinformation* 131 (2024), 103976.