

Research and prototyping study of an LLM-based chatbot for electromagnetic simulations

Albert Piwonski

Department of Theoretical Electrical Engineering, Faculty IV – Electrical Engineering and Computer Science, TU Berlin, Berlin, Germany, and

Mirsad Hadžiefendić

Independent Researcher, Berlin, Germany

Abstract

Purpose – This study aims to address the question of how generative artificial intelligence can be used to reduce the time required to set up electromagnetic simulation models. A chatbot based on a large language model (LLM) is presented, enabling the automated generation of simulation models with various functional enhancements.

Design/methodology/approach – A chatbot-driven workflow based on the LLM Google Gemini-2.0-Flash automatically generates and solves two-dimensional finite element eddy current models using Gmsh and GetDP. Python is used to coordinate and automate interactions between the workflow components. The study considers conductor geometries with circular cross-sections of variable position and number. In addition, users can define custom postprocessing routines and receive a concise summary of model information and simulation results. Each functional enhancement includes the corresponding architectural modifications and illustrative case studies.

Findings – With a defined set of functionalities, the chatbot successfully sets up and solves electromagnetic simulation models. Notably, it automatically infers not only Python code but also the domain-specific language code for GetDP. The case studies conducted revealed open research challenges, particularly with regard to the question of how to ensure that results are both syntactically and semantically valid.

Originality/value – Currently, the application of machine learning methods to solve electromagnetic boundary value problems is an active area of research (see, e.g. physics-informed neural networks or neural operators). However, to the best of the authors' knowledge, little research has examined the potential of artificial-intelligence-assisted generation of simulation models that prioritizes code generation and execution rather than the enhancement of numerical solution schemes. This study leverages a LLM and designs tailored workflows that contextualize it through carefully constructed system prompts.

Keywords Open source software, Finite element modeling, Eddy current problems, Chatbot, Large language models, Prompt engineering, Generative artificial intelligence

Paper type Research paper

1. Introduction

The application of machine learning (ML) methods, a subfield of artificial intelligence (AI), to the solution of electromagnetic boundary value problems (BVPs) is currently a highly



active area of research. Deep neural networks for operators (Kovachki *et al.*, 2023; Lu *et al.*, 2021a, 2021b; Li *et al.*, 2020) and physics-informed neural networks (Raissi and Karniadakis, 2018; Raissi *et al.*, 2019; Karniadakis *et al.*, 2021), in which information about the BVP (and possibly measurement data) is integrated into the loss function of the network, often aim to replace traditional numerical methods such as the finite element (FE) method. For physics-informed neural networks in electromagnetism specifically [see, e.g. Lim and Psaltis (2022); Guo *et al.* (2025); Rezende and Schuhmann (2025)]. Furthermore, deep neural networks have also been applied to reduced order modeling (see, e.g. He *et al.*, 2023).

This work addresses an orthogonal problem: How can AI methods be used to reduce the time required to set up electromagnetic simulation models, rather than solving the numerical models themselves? The focus is thus on the assisted generation of simulation models, whereby the numerical scheme itself remains unaffected. A conceptually related direction has recently emerged in the computational fluid dynamics (CFD) community. In Yue *et al.* (2025), an AI-based multiagent framework called Foam-Agent is introduced that supports users in performing simulations based on the open source CFD software OpenFOAM (OpenFOAM Ltd, 2025). To the best of the authors' knowledge, there are only a few works on this subject in computational electromagnetics [see, e.g. Lupoiu *et al.*, (2025)].

Motivated by this research gap, the present work introduces a chatbot-driven workflow based on the text capabilities of the multimodal large language model (LLM) Google Gemini-2.0-Flash (Google DeepMind, 2025) that facilitates the automated generation of two-dimensional eddy current simulation models using the open source FE tools Gmsh (Geuzaine and Remacle, 2009) and GetDP (Dular *et al.*, 1998). The study focuses on geometries consisting of conductors with circular cross-sections, whose positioning and number can be controlled by the user via natural language prompts. Further, users can define custom postprocessing routines, such as a visualization of the ohmic power loss density for a selected subset of conductors. In addition, the user is automatically provided with a textual summary of the model information and simulation results.

The remainder of this paper is organized as follows: Section 2 introduces the model problem, the main software tools and the basic workflow of the proposed chatbot prototype. Section 3 discusses architectural extensions of the workflow and case studies that demonstrate domain-specific language (DSL) code inference and automatic textual summarization. In addition, we introduce a possible encoding scheme describing what qualifies as an acceptable result from the user's point of view. Section 4 presents an evaluation study quantifying the performance of the chatbot-driven workflow. Different LLM models are benchmarked here on exemplary problems of varying difficulty. Finally, Section 5 summarizes the main findings and outlines future research directions.

2. Overview of the basic AI workflow

This section introduces the two-dimensional eddy current model problem and briefly discusses the open source finite element tools Gmsh and GetDP, focusing on their interfacing capabilities. These tools are subsequently integrated into a chatbot-driven workflow, which is presented here in its basic form.

2.1 Model problem

As an application example, we consider the translationally symmetric eddy current problem shown in Figure 1. Since the electromagnetic fields do not change with respect to the z -direction, the problem can be treated in two spatial dimensions.

We consider $N \in \mathbb{N}$ separated massive conductors with circular cross-sections of radius r_c and electrical conductivity $\sigma > 0$. The position of the i -th conductor is defined

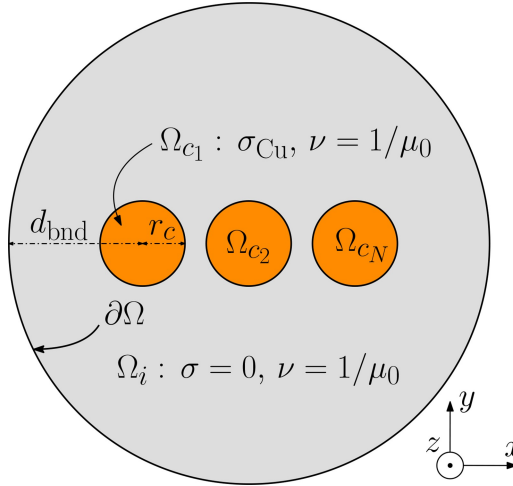


Figure 1. Sketch of the computational domain Ω : insulating domain Ω_i , boundary of the domain $\partial\Omega$ and N separated conductors, each with a radius r_c . The total conducting domain is $\Omega_c = \cup_{i \in N} \Omega_{c_i}$. Each conductor center has at least the minimal distance d_{bnd} to the circular boundary $\partial\Omega$

by its center point $\mathbf{p}_i \in \mathbb{R}^2$, with $i \in N$. We denote the conducting domain as $\Omega_c = \cup_{i \in N} \Omega_{c_i}$, which is surrounded by the insulating domain Ω_i with $\sigma = 0$. The total domain $\Omega = \Omega_c \cup \Omega_i$ has a circular boundary $\partial\Omega$. Since the positions of the conductors are considered variable, it must be ensured that the boundary $\partial\Omega$ fully encloses all of them, including a minimal distance d_{bnd} to reduce the influence of the boundary on the solution. To ensure this, the centroid of all conductors is computed, which defines the center of the circular boundary $\partial\Omega$. The radius of the boundary $\partial\Omega$ is determined by the distance from the centroid to the outermost conductor center \mathbf{p}_i , plus the constant d_{bnd} , see again Figure 1. We model the boundary $\partial\Omega$ as perfectly electrically conductive, i.e. $\sigma \rightarrow \infty$. Furthermore, magnetic materials are excluded from this application example, so the magnetic reluctivity is $\nu = 1/\mu_0$ in the entire domain.

The model is excited by imposed currents, which define the global conditions of the associated eddy current BVP: Each conductor carries a time-harmonic current of amplitude I with frequency f , respectively, angular frequency $\omega = 2\pi f$. For the sake of clarity, all parameters of the model problem are summarized in Table 1.

Table 1. Parameters of the eddy current model problem

Parameter	Description	Numerical value and unit
r_c	Radius of the conductors	5 mm
d_{bnd}	Minimum distance between the center of the outermost conductor and the boundary $\partial\Omega$	$3r_c = 15$ mm
ν	Magnetic reluctivity	$1/\mu_0$, with $4\pi \times 10^{-7}$ Vs/Am
σ	Electrical conductivity	0 in Ω_i , $\sigma_{\text{Cu}} = 58.1$ MS/m in Ω_c
I	Amplitude of the imposed current	1 A
$f = 2\pi/\omega$	Frequency of the imposed current	50 Hz

The physical phenomena are governed by the time-harmonic Maxwell's equations in the magnetoquasistatic limit:

$$\operatorname{div} \mathbf{B} = 0, \quad \operatorname{curl} \mathbf{E} = -j\omega \mathbf{B}, \quad \operatorname{curl} \mathbf{H} = \mathbf{J}, \quad (1)$$

in which \mathbf{B} is the magnetic flux density, \mathbf{E} is the electric field, \mathbf{H} is the magnetic field, \mathbf{J} is the current density and $j = \sqrt{-1}$ is the imaginary unit. As we consider linear materials, the following constitutive laws hold:

$$\mathbf{B} = \nu \mathbf{H}, \quad \mathbf{J} = \sigma \mathbf{E}, \quad \text{in } \Omega_c, \quad \mathbf{J} = \mathbf{0}, \quad \text{in } \Omega_i. \quad (2)$$

The perfectly electrically conductive boundary $\partial\Omega$ leads to vanishing tangential components of the electric field \mathbf{E} :

$$\mathbf{E} \times \mathbf{n}|_{\partial\Omega} = \mathbf{0}, \quad (3)$$

in which \mathbf{n} is the unit external normal vector of the boundary $\partial\Omega$.

Equations (1)–(3), together with the global conditions for the conductor currents, define an eddy current BVP that can be solved numerically using the finite element method. A commonly used finite element formulation for this type of problem is the modified $\mathbf{A} - v$ magnetic vector potential formulation, see, e.g. (Dular, 2023). Here, the magnetic vector potential \mathbf{A} satisfying $\mathbf{B} = \operatorname{curl} \mathbf{A}$ is introduced in Ω , while an electric scalar potential v is defined only in Ω_c . The electric field \mathbf{E} can be expressed in terms of the scalar and vector potentials as $\mathbf{E} = -j\omega \mathbf{A} - \operatorname{grad} v$. To ensure the uniqueness of the magnetic vector potential \mathbf{A} , a gauge condition must be imposed. For the translationally symmetric model the current density \mathbf{J} is assumed to be purely z -directed, such that the magnetic flux density \mathbf{B} only has in-plane components, i.e. $\mathbf{B} = \mathbf{e}_x B_x + \mathbf{e}_y B_y$. This is ensured by choosing $\mathbf{A} = \mathbf{e}_z A_z$, which implicitly fulfils the Coulomb gauge $\operatorname{div} \mathbf{A} = \partial A_z / \partial z = 0$, as the component A_z solely depends on the coordinates x and y .

Concisely, the $\mathbf{A} - v$ formulation, expressed in its weak (variational) form, can be stated follows: Seek $\mathbf{A} \in F_A$ and $\operatorname{grad} v \in \mathcal{V}(\Omega_c)$, such that for all test functions $\mathbf{A}' \in F_{A,0}$ and $\operatorname{grad} v' \in \mathcal{V}_0(\Omega_c)$:

$$\int_{\Omega} \nu \operatorname{curl} \mathbf{A} \cdot \operatorname{curl} \mathbf{A}' \, d\Omega + \int_{\Omega_c} j\omega \sigma \mathbf{A} \cdot \mathbf{A}' \, d\Omega + \int_{\Omega_c} \sigma \operatorname{grad} v \cdot \mathbf{A}' \, d\Omega = 0, \quad (4)$$

$$\int_{\Omega_c} j\omega \sigma \mathbf{A} \cdot \operatorname{grad} v' \, d\Omega + \int_{\Omega_c} \sigma \operatorname{grad} v \cdot \operatorname{grad} v' \, d\Omega = \sum_{i \in N} I_i V_i'. \quad (5)$$

The discrete counterpart of the function space $\mathcal{A}(\Omega)$ is spanned by z -directed nodal basis functions that vanish on the boundary, i.e. $A_z|_{\partial\Omega} = 0$. The function space $\mathcal{V}(\Omega_c)$ is represented by N z -directed constants, one associated with each conductor. The term V_i' denotes the voltage that can be associated to the i -th conductor. As the current $I_{i \in N} = I$ is fixed for each conductor, the voltages are resulting from the finite element solution. For additional information, the reader is referred to (Dular, 2023).

2.2 Open source finite element tools Gmsh and GetDP

The computational domain Ω shown in [Figure 1](#) is discretized using the open source finite element mesh generator Gmsh. Here, the coupling with our later chatbot-driven workflow is achieved via its Python ([Python Software Foundation, 2025](#)) application programming interface (API). This interface enables the implementation of a Python function that accepts as input a list of tuples of length 2, each specifying the coordinates of a conductor's center \mathbf{p}_i , with $i \in N$. The number of conductors N is directly determined by the length of the input list. This procedure enables mesh generation independently of the specific number and spatial arrangement of the individual conductors. The minimal and maximal mesh characteristic sizes are defined based on the radius of the boundary $\partial\Omega$ and the radius of the conductors r_c (see [Table 1](#)), to obtain a manageable number of triangles (for fast prototyping) independently of the number and spatial arrangement of the conductors. Prior to meshing, a check is performed to ensure that the conducting domains do not overlap. Furthermore, for the discretized counterparts of the conductive subdomains $\Omega_{c \in N}$ and the boundary $\partial\Omega$, so-called physical groups need to be defined and stored in the resulting mesh file (".msh" file extension), so that common material properties and boundary conditions can be assigned for the subsequent FE solver.

The $\mathbf{A} - \mathbf{v}$ formulation, see [equations \(4\), \(5\)](#), is implemented in the open source finite element solver GetDP. Unlike Gmsh, GetDP cannot be interfaced directly from Python, as no dedicated API is available. Instead, solver files must be written in GetDP's domain-specific language and stored as ASCII files with the ".pro" file extension. These solver files can then be executed via the command-line interface (CLI). After computation, the resulting field plots, e.g. for the magnetic flux density \mathbf{B} (".pos" file extension) can be visualized using the postprocessing facilities provided by Gmsh. It should be emphasized that the CLI commands are executed from Python through the subprocess module.

2.3 Chatbot-driven workflow

Both open source FE tools are now integrated into a chatbot-driven workflow, as illustrated in [Figure 2](#). The chatbot is based on the free tier of Google Gemini-2.0-Flash, a multimodal LLM, but uses only its text generation capabilities. For general literature on LLMs, see ([Alammar and Grootendorst, 2024](#)). The user interface is implemented as an interactive web application using the open source framework Streamlit ([Streamlit, 2025](#)), see [Figure 3](#). Python serves as the coordinating and automation layer, managing the interaction between workflow components.

The program workflow can be summarized as follows: the user's prompt is incorporated into a system prompt containing a task description, rules and examples (see [Appendix 1.1](#)), which is given to the LLM to generate a string of (ideally [\[1\]](#)) syntactically correct Python code. Then, a cleaned version of the LLM's output string serves as the input for a function that executes the dynamically generated Python code. The Gmsh and GetDP code is static and accessed via a predefined Python wrapper function (see [Section 2.2](#)), which takes a list of coordinate tuples as input and runs the finite element simulation as a side effect. It is important to emphasize that the LLM's weights are not updated or re-trained (that is, there is no fine-tuning); rather, the pre-trained model is contextualized at runtime via the system prompt. More precisely, the system prompt supplies task-specific instructions and context regarding the model's input, however, it does not alter the model's parameters. Moreover, this workflow does not employ retrieval-augmented generation (RAG) or any persistent memory mechanism: no external knowledge store is queried during generation and no state is retained beyond the immediate prompt.

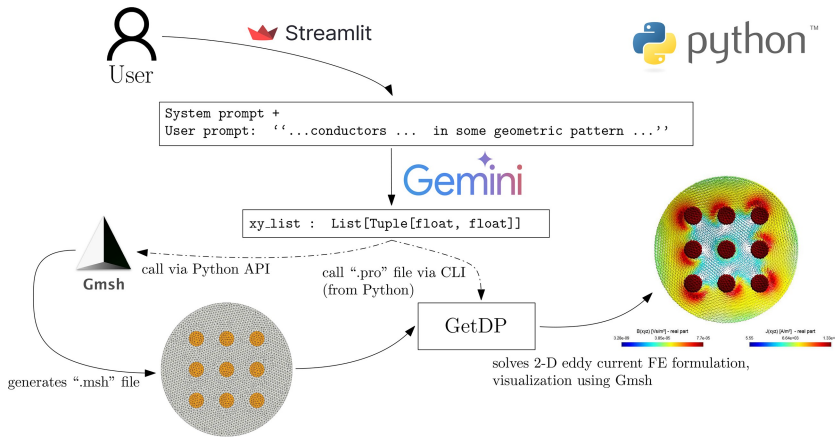


Figure 2. Sketch of the basic AI workflow

Large Language Model for Computational Electromagnetics

- This chatbot invokes an LLM-based workflow where the underlying LLM interfaces with Gemini 2.0 Flash (see, https://ai.google.dev/pricing?hl=en#2_0flash)
- The evaluation of the chatbot is around 89 % and it is carried out in the spirit of the human-driven review by LMarena, i.e., 89 out of 100 answers are rated as passed by a human.

Further used open source software tools

- for generating the 2-D triangulated FE mesh: Gmsh (see, <https://gmsh.info/>)
- for solving the 2-D MQS FE formulation: GetDP (see, <https://getdp.info/>)
- some remarks on the FE formulation: time-harmonic case, $\mathbf{A} - v$ formulation, implicitly Coulomb-gauged, electric boundary conditions, i.e., $(\mathbf{E} \times \mathbf{n}) = \mathbf{0}|_{\partial\Omega}$, total current (global condition) imposed separately into each of the conductors.

Chatbot: LLM for CEM

Let me assist you to set-up and solve a 2-D magnetoquasistatic (eddy current) boundary value problem for you!

Welcome! Which computational domain/geometry should I generate? I will then automatically discretize the domain into a triangular mesh using Gmsh and compute, also automatically, electromagnetic quantities (e.g., the magnetic flux density \mathbf{B} , current density \mathbf{J}) for you using the open source software GetDP.

Run a magnetoquasistatic simulation model with 9 conductors arranged in a square.

Searching for an answer...

Type here your message

Figure 3. User interface of the developed chatbot: Status bar containing additional meta information (left), initial information (top) and text box for user prompts (bottom)

3. Architectural extensions and case studies

This section presents functional enhancements to the basic AI workflow described in Section 2.3. The architectural modifications are detailed, and case studies are conducted and discussed for each autonomy level of the chatbot.

3.1 Inferring python code to generate a list of coordinate tuples

The workflow of Section 2.3 is used without further modifications to generate the exemplary results shown in Figure 4 and 5. Examples of the inferred Python code are provided in Appendix 2.1.

It can be observed that the workflow successfully infers Python code capable of generating conductor arrangements across multiple levels of complexity. The risk of conductor overlap can be significantly reduced if the user’s prompt includes the specific value of the conductor radius r_c , a parameter that is not known *a priori* to the LLM. The simulation models shown in Figure 5, based on identical user prompts (e) and (f), illustrate the stochastic nature of the underlying LLM. Both generated geometries are correct; however, the Python code for (e) assumes $\alpha = 1$, whereas the code for (f) uses $\alpha = 100$. The resulting simulation model for the user prompt (g) can be interpreted as hallucination of the LLM, that is, an output inconsistent with reality.

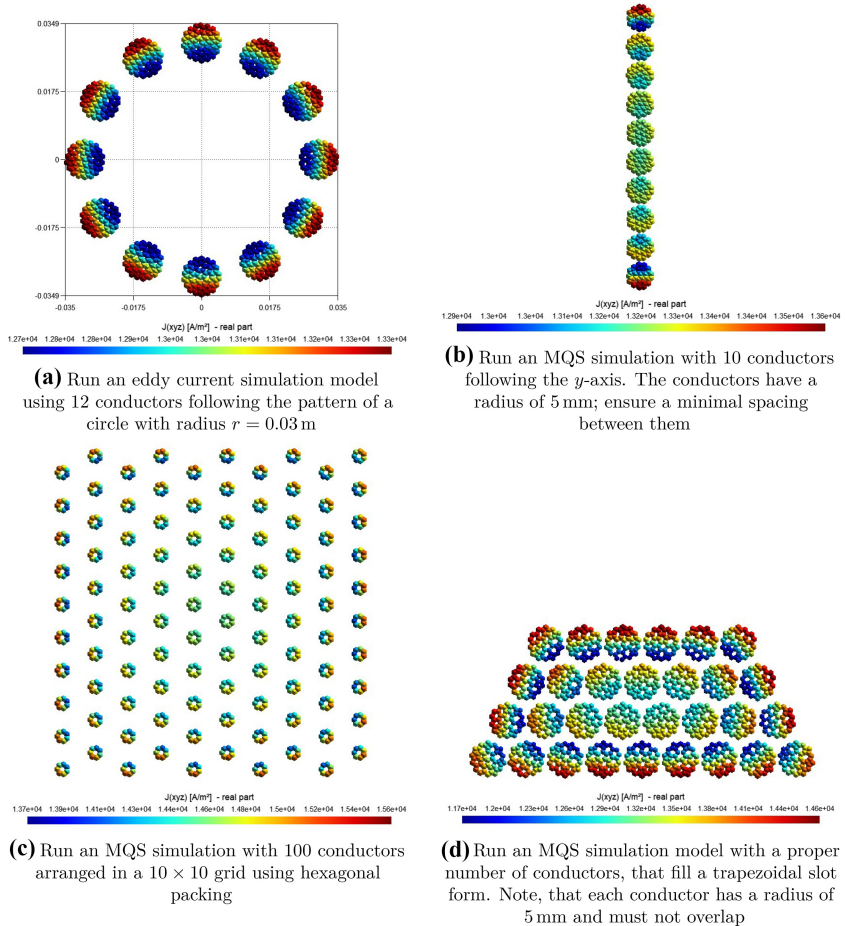


Figure 4. Real parts of the current density J distributions corresponding to the models generated based on user prompts (a)–(d)

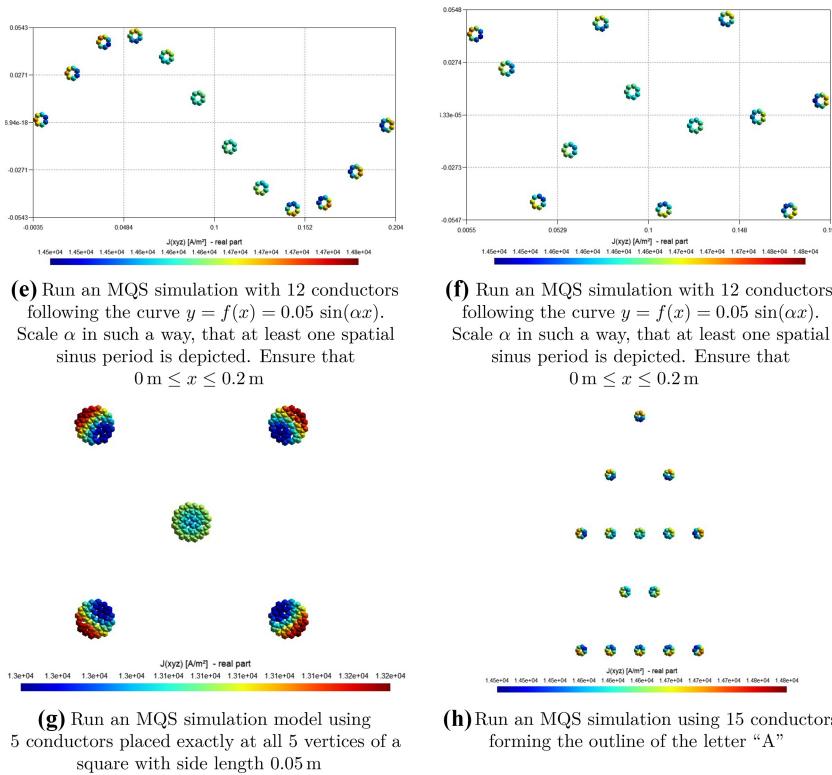


Figure 5. Real parts of the current density \mathbf{J} distributions corresponding to the models generated based on user prompts (e)–(h)

However, such hallucinations were observed only rarely in this use case; in this instance, the effect was triggered by the inconsistent user prompt, as a square obviously has only four vertices. From the standpoint of automated model validation, a notable challenge occurs when the inferred Python code is syntactically valid, yet the resulting model exhibits an incorrect geometric interpretation (see the generated model for the user prompt (h) in Figure 5).

3.2 Inferring domain-specific language code (including meaningful examples in the system prompt)

The workflow described in Section 2.3 is extended to allow the user to define custom postprocessing routines; see Figure 6. As an application example, the visualization of the ohmic power loss density p_Ω is shown, which is defined as a time-averaged quantity as follows:

$$p_\Omega = \frac{\mathbf{E} \cdot \mathbf{J}^*}{2} = \frac{\sigma |\mathbf{E}|^2}{2} = \frac{\sigma}{2} | -j\omega \mathbf{A} - \mathbf{grad} v |^2. \quad (6)$$

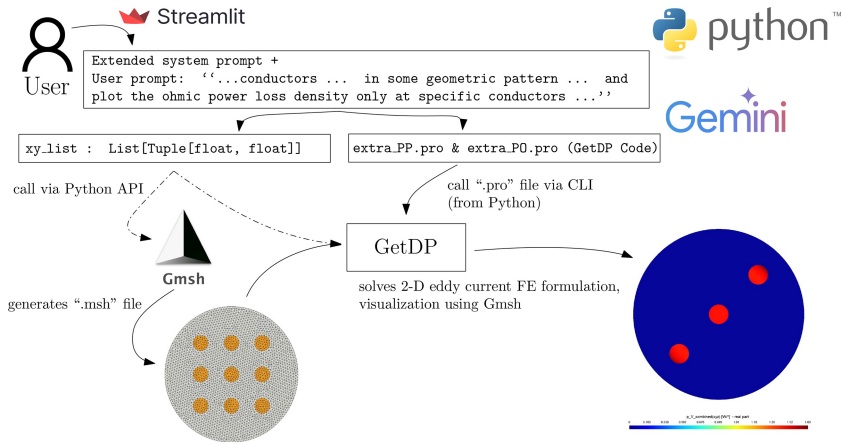


Figure 6. Sketch of the extended AI workflow for dynamically generated postprocessing routines (including meaningful examples in the system prompt)

An extended system prompt is provided to the LLM, that contains GetDP code examples for a postprocessing routine, that computes and plots the ohmic power loss density p_{Ω} only for a selected subset of conductors, see [Appendix 1.2](#). Internally, GetDP uses PostProcessing objects to define quantities based on the primary variables of the finite element formulation – here, the vector potential \mathbf{A} and the gradient of the electric scalar potential $\mathbf{grad} v$. Their visualization and output are controlled via PostOperation objects, which specify formats such as line evaluations, surface plots or data tables. The dynamically generated GetDP code is written into dedicated “.pro” files, which are then included in the main solver file. It should be emphasized that the LLM now infers code in the domain-specific language of GetDP, a language on which, most likely, Google Gemini-2.0-Flash has been trained considerably less such that the LLM’s internal knowledge (cf. ([Huyen, 2025](#), p. 301)) about GetDP is most likely quite low.

The extended workflow leads to the exemplary result shown in [Figure 7](#). Further examples of the LLM outputs are provided in the [Appendix 2.2](#). The results indicate that the basic workflow from the previous subsection can be extended with dynamically generated domain-specific language code. It was observed that including an adequate number of representative code examples significantly reduces syntax errors related to the GetDP language. In their absence, numerous syntax errors occurred, primarily due to missing or superfluous curly brackets.

3.3 Inferring domain-specific language code (excluding meaningful examples in the system prompt)

In [Figure 8](#), the main difference to the previous architecture presented in [Figure 6](#) is that (ideally) syntactically and semantically correct domain-specific language code is generated without having any relevant examples in the system prompt. The goal of this architectural modification is to enable valid responses to user prompts such as:

Using nine conductors that are positioned along a rectangle such that one point is at the center (intersection of the diagonals) and the other eight are distributed along the edges of the rectangle, evaluate the plot of the magnetic energy density in terms of the magnetic vector potential vector field within the frequency domain only for the central conductor and the conductors on the first and third bisectors of the rectangle.

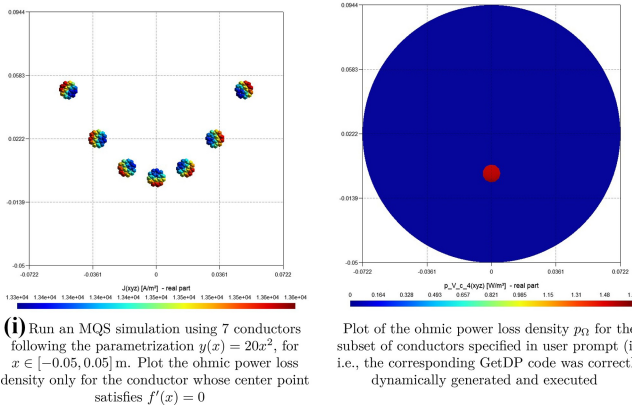


Figure 7. Left: Real part of the current density J corresponding to the simulation model resulting from user prompt (i), right: Customized postprocessing routine, see user prompt (i)

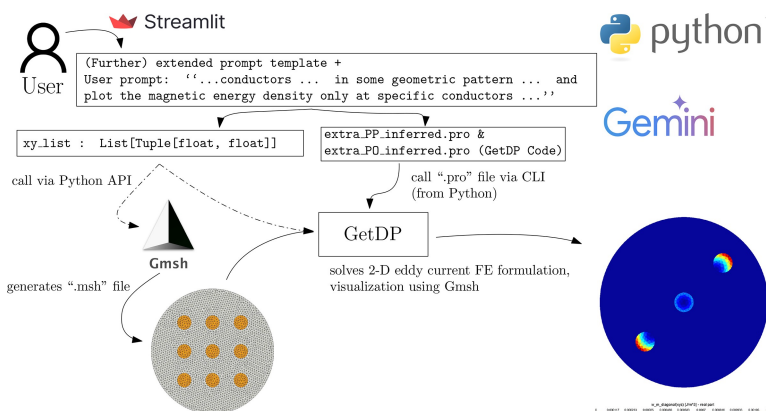


Figure 8. Sketch of the AI workflow for dynamically generated postprocessing routines (excluding meaningful examples in the system prompt)

An exemplary system prompt can be found in the [Appendix 1.3](#).

The distinctive property of this kind of user prompts is that, at the same time:

- these user prompts build upon examples within the system prompt discussed in previous posts such as “only for the central conductor” (cf. Section 3.2) and “along a rectangle” (cf. Section 3.1); and
- these user prompts refer to examples that are not within the system prompt such as “the magnetic energy density.”

Considering the visualization of the magnetic energy density w_m , recall its definition as a time-averaged quantity as follows:

$$w_m = \frac{\mathbf{H} \cdot \mathbf{B}^*}{4} = \frac{\nu}{4} |\mathbf{B}|^2 = \frac{\nu}{4} |\mathbf{curl} \mathbf{A}|^2. \quad (7)$$

A meaningful user prompt provides the LLM-based chatbot with sufficient knowledge regarding (II) such that syntactically and semantically correct DSL code can be inferred. For example, the above-mentioned user prompt has to be extended in such a way such that a contextual answer is received that conforms with the user's expectation. More precisely, the following sentence has to be added to the above-mentioned user prompt:

Mind that the following three constraints regarding the magnetic energy density formula should be satisfied: (1) the correct factor 0.25 is present; (2) the material relation between the magnetic flux density vector field and the magnetic vector field holds to be true w.r.t. the reciprocal permeability, i.e., ν ; (3) the expression is simplified by using norms.

Some observations concerning the inferring of domain-specific language code without using meaningful examples in the system prompt within the chatbot-driven workflow are:

- (1) The lack of the extension for the above-mentioned user prompt may result in:
 - syntactically incorrect code within the context of the DSL;
 - syntactically correct code that is semantically incorrect within the context of the underlying physics (e.g. using a factor of 0.5 vs. 0.25); and
 - syntactically correct code that behaves semantically correct within the context of the underlying physics, though, semantically incorrect within the context of the DSL (e.g. using the magnetic flux density instead of the magnetic vector potential as primary numerical solution quantity).
- (2) Given the system design at hand, there are still many optimization paths regarding the overall prompt engineering to, for example, balance the provision of knowledge by user prompts and system prompts. An interesting approach could be the use of a curated prompt store in a similar manner to a feature store [see, for example, [Huyen, \(2022\)](#), pp. 325ff] in other ML systems.

3.4 Inferring a textual summary of the AI workflow's output

The main extension of the previous section's workflow is the following: a function is added that executes a second LLM API-call. Its input is the output of the function that executes the first LLM API-call; and its output is a textual summary, see [Figure 9](#). Note that there is a system prompt that contains meaningful examples and rules for structuring the output format of the textual summary, see [Appendix 1.4](#). In other words, the dynamically generated syntactically and semantically (ideally) correct code is translated into a textual summary that expresses the physical meaning of the code in a natural language.

A couple of observations and conceptual challenges regarding the textual summary within the chatbot-driven workflow are:

- Specifically, the description regarding the skin- and the proximity-effect is reasonable to a certain degree. Since a well-defined metric for quantifying potential improvements in the quality of a textual summary is currently lacking, it remains unclear whether incorporating the skin depth as an additional physical entity in the code would yield a more nuanced textual summary.

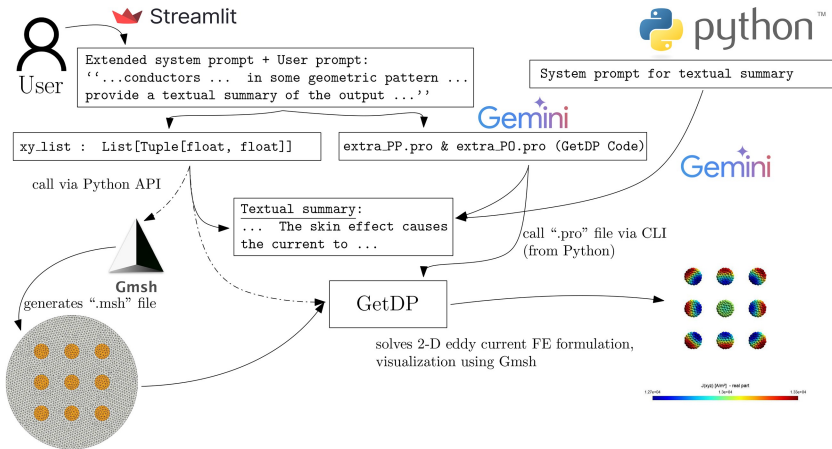
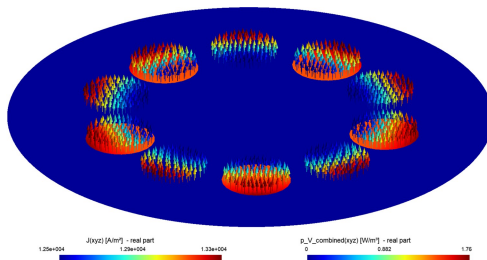


Figure 9. Sketch of the AI workflow for dynamically generated textual summaries

- Generally, the translation of code into natural language seems to work well to a reasonable extent, see Figure 10. Due to the above-mentioned lack of a quality metric for a textual summary, though, it remains unclear whether translating numerical simulation plots (i.e. images) into text or translating numerical



- (j) Run an MQS simulation using 10 conductors that follow the pattern of a circle. Plot the ohmic loss density only at every second conductor. Provide a summary of the output

Figure 10. Above: Real part of the current density \mathbf{J} and custom postprocessing routine for the ohmic power loss density p_{Ω} generated based on user prompt (j). Below: dynamically generated textual summary of the results

Note(s): The plot shows a 2D simulation of 10 conductors arranged in a circle. The ohmic loss density, representing power dissipation due to current flow, is displayed for every second conductor (conductors 1, 3, 5, 7, and 9). The positions of the conductors' center points are determined by a circle of radius 0.02, with coordinates calculated using sine and cosine functions, reflecting their circular arrangement. Due to the alternating current, both skin and proximity effects are present, leading to nonuniform current density distributions within the conductors and influencing the ohmic loss density. The skin effect causes current to concentrate near the surface of each conductor, while the proximity effect modifies the current distribution due to the presence of neighboring conductors

simulation numbers (i.e., tabular data) into text could lead to a more nuanced textual summary.

Notice that the conceptual challenge with regard to a quality metric for a textual summary is linked to the fundamental conceptual challenge of how to enable an automated LLM-workflow evaluation.

3.5 Aspects of semantic and syntactic sources of potential failure in the AI workflow

The workflows presented in Sections 3.1–3.4 introduce different levels of capabilities in the generated code. Here, we introduce a possible encoding scheme that describes what qualifies as an acceptable result from the user’s point of view.

Considering the workflow described in Section 3.1, the dynamically generated Python code must first satisfy the syntax and semantics of the Python programming language. Theoretically, all combinations of the truth values listed in Table 2 could occur. Syntax errors can arise, for instance, if brackets are left unclosed, as in “(a + 1”, or if reserved keywords are used as variable names. Semantic errors can occur, for example, when a variable is used before it is defined, or when operations are incompatible with the types of their inputs, as in “a’ + 3”. While syntactic errors are detected during parsing, both syntax and semantic errors are ultimately revealed when the generated code is executed.

More subtle errors occur when the generated Python code is both syntactically and semantically correct, but the resulting geometry does not match the user’s expectations, consider, for example, the resulting model for the user prompt (h) in Figure 5. Consistent with the concepts of syntax and semantics, an additional layer can be introduced to capture geometric information, which gives rise to the tensorial structure depicted in Figure 11. This

Table 2. Boolean values representing the syntax and semantics of the dynamically generated python code

Python syntax	Python semantics
×	×
×	✓
✓	×
✓	✓

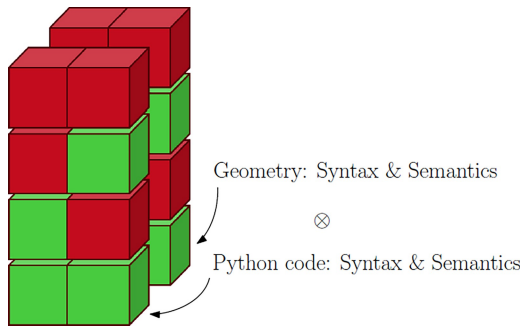


Figure 11. Stack of syntaxes and semantics for workflow as in Section 3.1

structure can be understood as follows: an output is considered acceptable to the user only if it is valid on both the syntax and semantic levels; applied to both the Python code and the corresponding geometry.

Conceptually, for the workflow presented in Sections 3.2 and 3.3, two additional layers are introduced (see Figure 12). Here, also the inferred GetDP code might contain syntax errors, such as improperly matched curly brackets. Semantic errors arise when a field variable is accessed that does not correspond to a primary variable in the FE formulation. Again, these errors are not difficult to detect, as GetDP's parser will produce an error message. Again more subtle, the inferred formula for the magnetic energy density w_m must be physically consistent. In this context, syntactical errors can be interpreted as operations that violate mathematical syntax, such as the addition of scalar and vector fields, consider, e.g. the expression " $v + \mathbf{A}$ ". Semantic errors, on the other hand, refer to operations that are mathematically valid but physically meaningless, for example, the addition of different vector field quantities such as " $\mathbf{E} + \mathbf{H}$ ".

The dynamically generated textual summary (see Section 3.4) introduces yet another layer. In this context, syntactical errors can be interpreted as grammatical or spelling errors in the generated text. Semantic errors, on the other hand, occur when there is a mismatch between the actual generated model and its corresponding textual summary.

4. Evaluation of the AI workflow

The goal of this section is to provide some quantifiable results regarding the presented AI workflow that lay the foundation for more in-depth evaluation analysis in future research.

4.1 Setup

Our proposed evaluation setup is as follows:

- (1) We choose three benchmarking user prompts (see Benchmarking user prompts (basic, intermediate, advanced) for the evaluation of the AI workflow that are representing electrical conductors in different geometrical configurations (recall Section 3.1) and postprocessing routines (recall Section 3.3)). These prompts represent different levels of complexity, that is, basic, intermediate and advanced. The complexity of user prompts is assessed across the three tiers based on the ambiguity potential and the electromagnetic computation demands as judged by a subject matter expert. Mind that the system prompts (cf. Appendix 1) are kept constant for all runs:

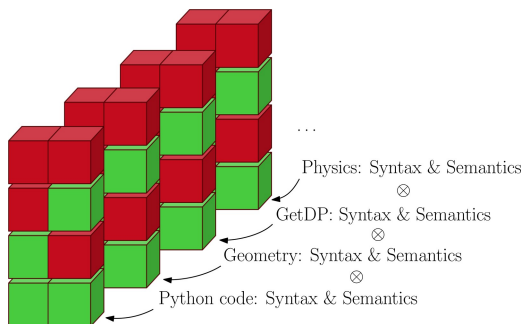


Figure 12. Stack of syntaxes and semantics for workflow as in Section 3.2 and 3.3

- *Basic*: “Run an MQS simulation with ten conductors placed along an outer circle (with a radius of 4 cm) and ten conductors placed in an inner circle. Note that each conductor has a radius of 5 mm. Ensure that the conductors are not overlapping. Plot the ohmic power loss density only for the inner circle’s conductors.”
 - *Intermediate*: “Run an MQS simulation model with a proper number of conductors that fill completely a symmetric trapezoidal slot form. Ensure that, at each vertex of the symmetric trapezoidal slot form, there is a conductor. Avoid that the conductor cross sections are overlapping when considering the radius = 0.005 m. Please enlarge the spacing between the conductors. Plot the ohmic power loss density only for the bottom row of conductors.”
 - *Advanced*: “Run an MQS simulation with a proper number of conductors that are formed like a Milliken-type conductor. Use at least 100 conductors. Arrange this number of conductors in separated segments that are kind of pie-shaped. Typically you need 6 of them. Leave some space in the inner part of the total cross-section (ca. 0.01 m) such that there is no conductor. Leave also some space between the segments such that their separation is clearly visible. Avoid that the conductor cross sections are overlapping when considering the radius = 0.005 m. Please enlarge the spacing between the conductors. Plot the ohmic power loss density only for the conductors of each segment that are along the boundary of the segment.”
- (2) Since the model Gemini-2.0-Flash is expected to be deprecated by publication (cf. [Google DeepMind, 2026a](#)), we select the following five models for the analysis:
- Gemma-3-1b-It
 - Gemma-3-27b-It
 - Gemini-2.5-Flash-Lite
 - Gemini-2.5-Flash
 - Gemini-3.1-Fash-Lite

Gemma models are lightweight, open-weight models built on technology similar to that of Gemini’s full-fledged, closed-source lineup. Both model families are consumed via Google’s corresponding APIs. Note that Gemini-2.5-Flash has higher input and output token prices than Gemini-3.1-Flash-Lite, which in turn exceeds Gemini-2.5-Flash-Lite (cf. [Google DeepMind, 2026b](#)). These prices indicate as a first approximation that, for the Gemini models, Gemini-2.5-Flash is the most capable, followed by Gemini-3.1-Flash-Lite, then Gemini-2.5-Flash-Lite.

- (3) For a given model and a benchmarking user prompt, the number of tries until the first successful *syntactical* AI workflow execution is counted (recall Section 3.5). Two independent attempts are conducted. If more than ten tries are needed, an “x” is marked. The results are shown in [Table 3](#).
- (4) Based on the results from [Table 3](#), it is decided to proceed with a subset of the models to analyze the successful *semantical* AI workflow execution (recall Section 3.5). In [Figure 13](#), there are reference simulation results that show one representative of the class of valid geometrical configurations and valid postprocessing routine executions associated with each benchmarking user prompt. Given these reference simulation results, a subject matter expert evaluates each successful *syntactical* AI workflow execution whether there is (a) a valid geometrical configuration and (b) given (a), whether there is a valid postprocessing routine execution in the sense of that they are

Table 3. For a given model (Gemma-3-1b-it, gemma-3-27b-It, Gemini-2.5-Flash-Lite, Gemini-2.5-Flash and Gemini-3.1-Flash-Lite) and a given benchmarking user prompt (basic, intermediate, advanced), the number of tries until the first successful *syntactical* AI workflow execution is recorded. The light-gray numbers and gray numbers indicate results from the first attempt and the second attempt, respectively. An “x” indicates more than ten tries without success

Model	Basic		Intermediate		Advanced	
Gemma-3-1b-It	x	x	x	x	x	x
Gemma-3-27b-It	1	1	1	2	x	x
Gemini-2.5-Flash-Lite	1	2	3	2	x	9
Gemini-2.5-Flash	1	1	1	1	2	5
Gemini-3.1-Flash-Lite	2	1	1	1	x	x

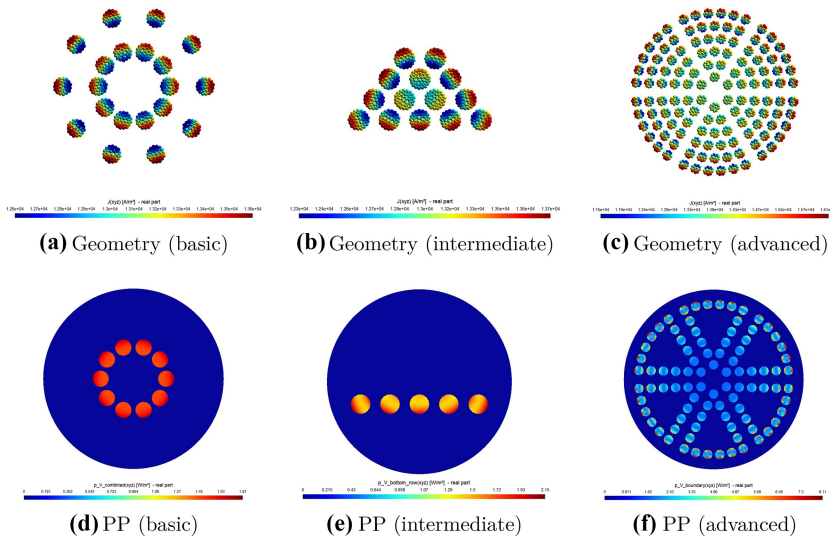


Figure 13. Reference simulations based on benchmarking user prompts (see Benchmarking user prompts (basic, intermediate, advanced) for the evaluation of the AI workflow that are representing electrical conductors in different geometrical configurations (recall Section 3.1) and postprocessing routines (recall Section 3.3)); “PP” denotes postprocessing

members of the class of valid geometrical configurations and valid postprocessing routine executions, respectively.

For example, for one attempt, if there are eight valid geometrical configurations out of ten successful *syntactical* AI workflow executions, then 8 / 10 is recorded. Given the eight valid geometrical configurations, if there are five valid postprocessing routine executions, then 5/8 is recorded. In total two attempts for each pair of models and benchmarking user prompts are conducted. The results are summarized in [Table 4](#).

4.2 Observations in terms of quality

Given the results in Table 3, using Gemma-3-1b-It, one is not able to receive a successful *syntactical* AI workflow execution for any of the benchmarking user prompts. The advanced benchmarking user prompt seems to be the most challenging for all five models, though, Gemini-2.5-Flash appears to be the only reliable one that, on average, needs 3.5 tries for getting a successful *syntactical* AI workflow execution for the advanced case. The basic and the intermediate benchmarking user prompt are feasible for most of the models. However, Gemini-2.5-Flash-Lite needs, on average, slightly more tries than the other models. Given these results, we conclude that, in terms of gain of knowledge for the next evaluation step, i.e. the successful *semantical* AI workflow execution analysis, it is sufficient to consider solely the models Gemini-2.5-Flash, Gemini-3.1-Flash-Lite and Gemma-3-27b-It.

Given the results in Table 4, it is clear that Gemma-3-27b-It is not able to properly lead to successful *semantical* AI workflow executions for the intermediate benchmarking user prompt, more precisely, there has not been observed one valid geometrical configurations, that is, a symmetric trapezoidal slot form. In the case of the basic benchmarking user prompt, there has not been observed one valid postprocessing routine execution (given a valid geometric configuration), more precisely, the ohmic power loss density has not been plotted for the inner circle's conductors, cf. Figure 14.

Using Gemini-3.1-Flash-Lite for the basic benchmarking prompt leads, on average, almost surely to a valid geometric configuration and, given those valid geometric configurations, on average, in 80% of cases, it leads to a valid postprocessing routine execution. Using it for the intermediate benchmarking prompt leads, on average, only in 15%

Table 4. For a given model (Gemma-3-27b-It, Gemini-2.5-Flash and Gemini-3.1-Flash-Lite) and a given benchmarking user prompt (basic, intermediate, advanced) and given ten successful *syntactical* AI workflow executions, the relative number of (a) valid geometrical configurations and (b) given (a), there is a valid postprocessing routine execution. The light-gray numbers and gray numbers indicate results from the first attempt and the second attempt, respectively

Model	Basic		Intermediate		Advanced	
Gemma-3-27b-It	(a) 10/10 (b) 0/10	(a) 10/10 (b) 0/10	(a) 0/10 (b) 0	(a) 0/10 (b) 0	/	
Gemini-2.5-Flash	(a) 10/10 (b) 10/10	(a) 10/10 (b) 10/10	(a) 5/10 (b) 5/5	(a) 4/10 (b) 4/4	(a) 8/10 (b) 8/8	(a) 7/10 (b) 7/7
Gemini-3.1-Flash-Lite	(a) 10/10 (b) 7/10	(a) 10/10 (b) 9/10	(a) 2/10 (b) 1/2	(a) 1/10 (b) 1/1	/	

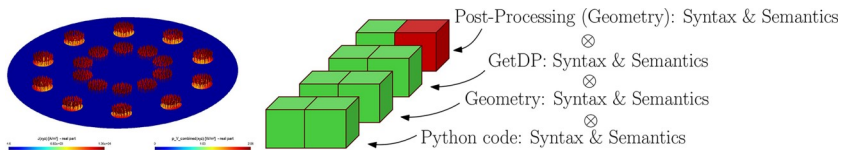


Figure 14. Left: systems result for the basic benchmarking prompt using the LLM Gemma-3-27b-It. The correct conductor arrangement was achieved, however the ohmic power loss density is wrongly plotted for the outer layer of conductors. Right: corresponding interpretation of the stack of syntaxes and semantics, cf. discussion in Section 3.5

of cases to a valid geometric configuration and, given those valid geometric configurations, on average, in 75% of cases, it leads to a valid postprocessing routine execution.

Using Gemini-2.5-Flash for the basic benchmarking prompt leads, on average, almost surely to a valid geometric configuration and, given those valid geometric configurations, on average, almost surely, it leads to a valid postprocessing routine execution as well. Using it for the intermediate benchmarking prompt leads, on average, only in 45% of cases to a valid geometric configuration and, given those valid geometric configurations, on average, almost surely, it leads to a valid postprocessing routine execution. Finally, using Gemini-2.5-Flash for the advanced benchmarking prompt leads, on average, in 75% of cases to a valid geometric configuration and, given those valid geometric configurations, on average, almost surely, it leads to a valid postprocessing routine execution.

Given the above-mentioned observations, it is reasonable to assume that using Gemini-2.5-Flash from Google's AI model families will lead to successful *syntactical* and *semantical* AI workflow executions.

4.3 Observations in terms of costs and time

Costs. The previous Section 4.2 focused primarily on the quality of the AI workflow. Regarding costs, each benchmarking user prompt (plus constant system prompt; cf. Section 4.1) averages approximately 10,000 input and output tokens such that all runs in Tables 3 and 4 total roughly €1.

Time. A single run from the Table 3 and the Table 4 takes just a few seconds. Note the up-front setup investment and ongoing maintenance, but post-setup, time-to-experimentation drops by orders of magnitude. Section 4.2 confirms acceptable results across complexity levels in seconds.

Contrast this with simulation engineers without AI tools: for the basic prompt (see Table 3), an entry-level engineer might need 8 h, mid-level 4 h and master-level 2 h to start experimenting. Subsequently, for the intermediate prompt, these times halve; and for the advanced one, the times (most likely) increase again – even for master-level simulation engineers. Even with AI tools, our proposed AI workflow likely offers lower time-to-experimentation, reliably and cost-efficiently, across more use cases.

5. Conclusion and outlook

5.1 Conclusion

This work has investigated an LLM-based chatbot for electromagnetic simulations as a minimum viable setup within the context of AI-assisted code generations and executions for numerical experiments.

More precisely, we have presented a workflow that is composed by open source FE tools (Gmsh and GetDP) and a LLM (Google Gemini-2.0-Flash) embedded within a common Python-based interface (cf. Section 2). The basic AI workflow presented in Section 3.1 infers Python code to generate a list of coordinate tuples. However, architectural extensions also enabled the inference of domain-specific language code, with and without meaningful examples in the system prompt (see Sections 3.2 and 3.3), and the inference of a textual summary of the AI workflow's output (see Section 3.4). Finally, we have examined the semantic and syntactic sources of potential failure within the AI workflow (see Section 3.5) and quantified these by introducing an evaluation methodology in Section 4.

One key insight is that the provided setup can already enable a useful level of automation of a numerical simulation workflow which can significantly reduce the time required to generate a well-posed numerical simulation model. Mind that this time-to-experimentation is

a critical metric and lowering it facilitates the accelerated exploration of, for instance, various physical scenarios and corresponding numerical simulation model configurations.

Another key insight is that relying merely on the internal knowledge of an LLM (that is, its training data) as a memory mechanism leads to insufficient outcomes of the AI workflow based on human evaluation. For example, physically relevant factors are not taken into account properly. However, by additionally using the LLM's context (via the user prompt and the system prompt) as a further memory mechanism satisfactory outcomes of the AI workflow can be achieved.

A third key insight is that the systematic consideration of both the semantic and syntactic aspects of an AI workflow's essential elements can offer a valuable conceptual guidance for analyzing potential failure modes. The proposed visual representation of a stack of syntaxes and semantics for the workflow's essential elements illustrates the many combinations in which the workflow can fail. Note that this proposed visual representation is highly scalable with respect to the increasing complexity of the underlying setup, thus offering a way to conceptually manage the increased complexity.

A fourth key insight is that the shown AI workflow facilitates a declarative development style where the focus is on describing what the desired outcome is, rather than explicitly specifying how to achieve the outcome step-by-step. In many scenarios, the desired outcome is a correct numerical solution, regardless of the exact method used to achieve it. However, for validating the solution's correctness, domain knowledge is needed – which leads to the last key insight.

The final key insight is that, due to the probabilistic nature of an LLM, it is unclear how to construct a reliable automated evaluation method for the AI workflow. The evaluation of the discussed AI workflow's outcomes relied on human evaluation. Human evaluation defines one end of the spectrum of evaluation methods, formal verification defines the other end of the spectrum. Since most likely no formal guarantees regarding the AI workflow's outcomes can be given, it seems at least conceivable to design semi-automated evaluation methods where human evaluation is potentially reduced to a minimum. An in-depth analysis of these ideas is left for future investigations.

5.2 Outlook

Future research and development efforts can build on the presented results in several directions. One of the most challenging and crucial efforts is the investigation of reliable (semi-) automated evaluation methods for the AI workflow. For evaluating the quality of textual summaries, for instance, one promising approach could be the usage of an additional LLM that is adapted for evaluation purposes. Another promising approach involves the use of embeddings, or vector representations (Weller *et al.*, 2025), along with similarity metrics that enable a textual summary to be quantitatively compared to human-curated or AI-generated benchmarks.

Addressing (semi-)automated evaluation methods is also important for systematically examining the space spanned by system prompts and LLMs to find the optimal configuration for the given AI workflow.

Furthermore, examining (semi-) automated evaluation methods is also critical in the investigation of more complex AI workflows. Notice that, from a control-flow graph point of view, the discussed AI workflow can be mathematically represented as a directed acyclic graph. However, more intricate graphs are conceivable that include both sequential and parallel paths as well as iterations and selections. Moreover, if more tools (i.e. if more callable functions) are available and the planning (i.e. the construction of a control-flow graph) to complete a user-defined task is undertaken by an LLM, then the resulting system is more appropriately characterized as an AI agent rather than a predefined AI workflow. It should be noted that the

research field of AI agents currently lacks a well-defined theoretical foundation (see, e.g. Huyen, 2025, pp. 276). Therefore, there are many open research questions on AI agents in the context of electromagnetic simulations that deserve a thorough investigation.

Another potential architectural extension to further enhance the developed chatbot's functionalities is the use of retrieval-augmented generation (RAG) techniques (see, e.g. Huyen, 2025, pp. 253–275). These techniques would enable the underlying LLM to access and integrate information that the model was not, or only partially, trained on. In addition, this information is more relevant to a given user prompt than a predefined and user-prompt-agnostic system prompt. In particular, with the FE software tools (Gmsh and GetDP) integrated in the chatbot, it would be beneficial to provide to the LLM only those portions of code repositories or manuals of the FE software tools that are most relevant to a user prompt.

Finally, a valuable opportunity for future work involves expanding the set of open source numerical tools beyond Gmsh and GetDP to include additional frameworks such as openCFS (Schoder and Roppert, 2025) and DeepXDE (Lu *et al.*, 2021a, 2021b). Such an extension would enable a unified platform of diverse numerical simulation software that can be accessed by user prompts in natural language.

Note

- [1.] Mind that the LLM is a probabilistic model where, even though, input and output guardrails are used, there is no complete guarantee that the expected correct structured output is generated and that this output is fully reproducible in a deterministic sense. Thus, even when given the same input, a degree of uncertainty regarding the resulting output remains.

References

- Alammar, J. and Grootendorst, M. (2024), *Hands-On Large Language Models: Language Understanding and Generation*, O'Reilly.
- Dular, J. (2023), "Standard and mixed finite element formulations for systems with Type-II superconductors", PhD thesis. University of Liège, Liège, Belgium.
- Dular, P., Geuzaine, C., Henrotte, F. and Legros, W. (1998), "A general environment for the treatment of discrete problems and its application to the finite element method", *IEEE Transactions on Magnetics*, Vol. 34 No. 5, pp. 3395-3398, doi: [10.1109/20.717799](https://doi.org/10.1109/20.717799).
- Geuzaine, C. and Remacle, J.-F. (2009), "Gmsh: a 3-D finite element mesh generator with built-in pre- and post-processing facilities", *International Journal for Numerical Methods in Engineering*, Vol. 79 No. 11, pp. 1309-1331, doi: [10.1002/nme.2579](https://doi.org/10.1002/nme.2579).
- Google DeepMind (2025), "Gemini 2.0 flash", available at: <https://ai.google.dev/gemini-api/docs> Version 2.0, (accessed 20 October 2025).
- Google DeepMind (2026a), "Gemini deprecations", available at: <https://ai.google.dev/gemini-api/docs/deprecations?hl=en> (accessed 8 March 2026).
- Google DeepMind (2026b), "Gemini developer API pricing", available at: <https://ai.google.dev/gemini-api/docs/pricing?hl=en> (accessed 8 March 2026).
- Guo, Z., Nguyen, B. and Sabariego, R.V. (2025), "Physics-informed neural network for solving 1-D nonlinear time-domain magneto-quasi-static problems", *IEEE Transactions on Magnetics*, Vol. 61 No. 5, pp. 1-9, doi: [10.1109/TMAG.2025.3553236](https://doi.org/10.1109/TMAG.2025.3553236).
- He, X.F., Li, L., Lanteri, S. and Li, K. (2023), "Model order reduction for parameterized electromagnetic problems using matrix decomposition and deep neural networks", *Journal of Computational and Applied Mathematics*, Vol. 431, p. 115271, Issn: 0377-0427, doi: [10.1016/j.cam.2023.115271](https://doi.org/10.1016/j.cam.2023.115271).

- Huyen, C. (2022), *Designing Machine Learning Systems: An Iterative Process for Production-Ready Applications*, O'Reilly.
- Huyen, C. (2025), *AI Engineering: Building Applications with Foundation Models*, O'Reilly.
- Karniadakis, G.E., Kevrekidis, I.G. and Lu, L. (2021), "Physics-informed machine learning". In", *Nature Reviews Physics*, Vol. 3 No. 6, pp. 422-440, doi: [10.1038/s42254-021-00314-5](https://doi.org/10.1038/s42254-021-00314-5).
- Kovachki, N., Li, Z., Liu, B., Azizzadenesheli, K., Bhattacharya, K., Stuart, A. and Anandkumar, A. (2023), "Neural operator: learning maps between function spaces with applications to PDEs", *J. Mach. Learn. Res.*, Vol. 24 No. 1, issn: 1532-4435.
- Li, Z., Kovachki, N., Azizzadenesheli, K., Liu, B., Bhattacharya, K., Stuart, A. and Anandkumar, A. (2020), "Fourier neural operator for parametric partial differential equations", In: CoRR abs/2010.08895, available at: <https://arxiv.org/abs/2010.08895>
- Lim, J. and Psaltis, D. (2022), "MaxwellNet: physics-driven deep neural network training based on maxwell's equations", *APL Photonics*, Vol. 7 No. 1, p. 011301.issn: 2378-0967, doi: [10.1063/5.0071616](https://doi.org/10.1063/5.0071616).
- Lu, L., Meng, X., Mao, Z. and Karniadakis, G.E. (2021a), "DeepXDE: a deep learning library for solving differential equations", *SIAM Review*, Vol. 63 No. 1, pp. 208-228, doi: [10.1137/19M1274067](https://doi.org/10.1137/19M1274067).
- Lu, L., Jin, P. and Pang, G. (2021b), "Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators", *Nature Machine Intelligence*, Vol. 3 No. 3, pp. 218-219, doi: [10.1038/s42256-021-00302-5](https://doi.org/10.1038/s42256-021-00302-5).
- Lupoiu, R., Shao, Y., Dai, T., Mao, C., Edée, K. and Fan, J.A. (2025), "A multi-agentic framework for real-time, autonomous freeform metasurface design", *Science Advances*, Vol. 11 No. 44, p. eadx8006, doi: [10.1126/sciadv.adx8006](https://doi.org/10.1126/sciadv.adx8006).
- OpenFOAM Ltd (2025), "OpenFOAM", available at: www.openfoam.com/ (accessed 21 October 2025).
- Python Software Foundation (2025), "Python 3.11.14 documentation", Version 3.11.14, available at: <https://docs.python.org/3.11/>
- Raissi, M. and Karniadakis, G.E. (2018), "Hidden physics models: Machine learning of nonlinear partial differential equations", *Journal of Computational Physics*, Vol. 357, pp. 125-141.issn: 0021-9991, doi: [10.1016/j.jcp.2017.11.039](https://doi.org/10.1016/j.jcp.2017.11.039).
- Raissi, M., Perdikaris, P. and Karniadakis, G.E. (2019), "Physics-informed neural networks: a deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations", *Journal of Computational Physics*, Vol. 378, pp. 686-707, Issn: 0021-9991, doi: [10.1016/j.jcp.2018.10.045](https://doi.org/10.1016/j.jcp.2018.10.045), available at: www.sciencedirect.com/science/article/pii/S0021999118307125
- Rezende, R.S. and Schuhmann, R. (2025), "An efficient PINNs approach using hard constraints boundary conditions for solving electromagnetic problems", *URSI Radio Science Letters*, Vol. 7, pp. 1-5, doi: [10.46620/25-0032](https://doi.org/10.46620/25-0032).
- Schoder, S. and Roppert, K. (2025), "openCFS: Open source finite element software for coupled field simulation – part acoustics", arXiv: 2207.04443 [math.NA], available at: <https://arxiv.org/abs/2207.04443>
- Streamlit (2025), "Streamlit: Build apps for data science", available at: <https://streamlit.io> Version 1.41.1 (accessed 20 October 2025).
- Weller, O., Boratko, M., Naim, I. and Lee, J. (2025), "On the theoretical limitations of embedding-based retrieval", arXiv: 2508.21038 [cs.IR], available at: <https://arxiv.org/abs/2508.21038>
- Yue, L., Somasekharan, N., Zhang, T., Cao, Y., Chen, Z., Di, S. and Pan, S. (2025), "Foam-agent: towards automated intelligent CFD workflows", arXiv: 2505.04997 [cs.AI], available at: <https://arxiv.org/abs/2505.04997>

Appendix 1. A system prompts

This appendix includes representative code examples from the system prompts of the workflows discussed in Section 3. Each code snippet is part of a Python multiline string that is passed to the LLM. Mind that the full system prompt that is containing a task description, some rules and examples as well as the user input placeholder is only provided for the first system prompt 1.1. For the other system prompts, only some examples are shown and, for the sake of brevity, the rest is omitted.

Note that in the system prompts, curly braces “{” and “}” serve a special function, as they are used to denote placeholders for dynamic content. Therefore, when literal braces are required in the string (i.e. not as placeholders), they must be escaped by doubling them – written as “{{” and “}}”. This ensures that the braces are interpreted as characters rather than as variable placeholders during template rendering. This is particularly necessary for the GetDP syntax code.

1.1 System prompt for inferring python code to generate a list of coordinate tuples

```

1 """
2 Task:
3 You are an expert Python programmer. Your task is to generate Python code that
4 creates a list of 2D points based on user specifications.
5 Write clean Python code that defines these points as a list of two-element float tuples.
6
7 Rules:
8 - Use only the provided libraries and functions: numpy, matplotlib.pyplot, random. Do
  not use any other
9 libraries or functions that are not explicitly provided.
10 - Do not include any explanations or apologies in your responses.
11 - Ensure the function is syntactically correct and uses valid Python methods.
12 - Call the generated function afterwards.
13 - Don't make up any result, i.e., if you don't know how to implement a specific feature,
14 say you don't know.
15 - Interpret the variable xy_list as type List[Tuple[float, float]]
16 - Do not produce markdown.
17 - Produce only a valid python string-expression.
18
19 Example 1: "Run a minimal magnetoquasistatic simulation with some initial points"
20     from typing import List, Tuple
21     from simulation_functions.test_gmsh_api_given_list import
22         run_simulation_from_terminal
23     xy_list : List[Tuple[float, float]] = [(0.0, 0.0), (0.02, 0.0), (0.0, 0.02)]
24     run_simulation_from_terminal(xy_list)
25
26 Example 2: "Run an initial mqs simulation using only one conductor"
27     from typing import List, Tuple
28     from simulation_functions.test_gmsh_api_given_list import
29         run_simulation_from_terminal
30     xy_list : List[Tuple[float, float]] = [(0.0, 0.0)]
31     run_simulation_from_terminal(xy_list)
32
33 (...)
34
35 The user input is:
36 {user_input}
37 """

```

1.2 System prompt for inferring domain-specific language code (including meaningful examples in the system prompt)

```
1 Example 6: "Evaluate the plot of the power loss density for the first conductor of three
   conductors along the x-axis"
2
3 with open("simulation_functions/round_2D_conductors_extra_PP.pro", "w") as file:
4   text_1 = "PostProcessing {{
5     {{ Name MagDyn_b; NameOfFormulation MagDyn_a;
6       PostQuantity {{
7         {{ Name OhmicLossDensity_conductor_1;
8           Value {{ Local {{ [sigma[]/2 * Norm[ (- Dt[{{a}}] - {{grad_phi}})]^2 };
9             In Region[{{Omega_c_1}}]; Jacobian Vol; }} }} }}
10        }} // end of PostQuantity block
11      }} // end of specific PostProcessing MagDyn_b
12    }} // end of PostProcessing block
13  "
14  file.write(text_1)
15
16 with open("simulation_functions/round_2D_conductors_extra_PO.pro", "w") as file:
17   text_1 = "PostOperation {{
18     {{ Name MagDyn_b; NameOfPostProcessing MagDyn_b;
19       Operation {{
20         Print[ OhmicLossDensity_conductor_1, OnElementsOf Omega,
21           File "Results/p_V_conductor_selected.pos",
22           Name "p_V_c_1(xyz) [W/m^3] ", Format Gmsh
23         ];
24       }} // end of Operation block
25     }} // end of specific PostOperation MagDyn_b
26   }} // end of PostOperation block
27  "
28  file.write(text_1)
29
30 from typing import List, Tuple
31 from simulation_functions.test_gmsh_api_given_list import run_simulation_from_terminal
32 xy_list : List[Tuple[float, float]] = [(0.02, 0.0), (0.0, 0.0), (-0.02, 0.0)]
33 run_simulation_from_terminal(xy_list)
```

1.3 System prompt for inferring domain-specific language code (excluding meaningful examples in the system prompt)

```

1 Example 12: "Evaluate the plot of the magnetic vector field of three conductors along
  the x-axis"
2
3 with open("simulation_functions/round_2D_conductors_extra_PP_inference.pro", "w") as
  file:
4   text_1 = "PostProcessing {{
5     {{ Name MagDyn_c; NameOfFormulation MagDyn_a;
6     PostQuantity {{
7       {{ Name h_vector_field;
8       Value {{ Term {{ [nu[] * {{d a}}}};
9       In Omega; Jacobian Vol; }} }} }}
10    }} // end of PostQuantity block
11  }} // end of specific PostProcessing MagDyn_b
12 }} // end of PostProcessing block
13 "
14   file.write(text_1)
15
16 with open("simulation_functions/round_2D_conductors_extra_P0_inference.pro", "w") as
  file:
17   text_1 = "PostOperation {{
18     {{ Name MagDyn_c; NameOfPostProcessing MagDyn_c;
19     Operation {{
20       Print[ h_vector_field, OnElementsOf Omega,
21       File "Results/h_vector_field.pos",
22       Name "H(xyz) [A/m]", Format Gmsh
23     ]};
24     }} // end of Operation block
25   }} // end of specific PostOperation MagDyn_c
26 }} // end of PostOperation block
27 "
28   file.write(text_1)
29
30 from typing import List, Tuple
31 from simulation_functions.test_gmsh_api_given_list import run_simulation_from_terminal
32 xy_list : List[Tuple[float, float]] = [(0.02, 0.0), (0.0, 0.0), (-0.02, 0.0)]
33 run_simulation_from_terminal(xy_list)

```

1.4 System prompt for inferring a textual summary of the LLM's output

```

1 Example 4:
2 "Run a basic mqs simulation using three conductors where one is on the y-axis and the
   other two are on the x-axis"
3
4 Plot Execution Result:
5 import os
6 current_dir = os.getcwd()
7 print(current_dir)
8 from typing import List, Tuple
9 from simulation_functions.test_gmsh_api_given_list import run_simulation_from_terminal
10 xy_list : List[Tuple[float, float]] = [(0.02, 0.0), (0.0, 0.02), (-0.02, 0.0)]
11 run_simulation_from_terminal(xy_list)
12
13

```

The output of Example 4 shows a 2D-plot of three electric conductors whose center points are at (0.02, 0.0), (0.0, 0.02), and (-0.02, 0.0) and all radii are 5 mm. Thus, two conductors lie on the x-axis with respect to a Cartesian coordinate frame and one lies on the y-axis. Current density vector fields are observed only within the conductors, in the surrounding non-conducting computational domain, there current density vector fields are zero. All three current density vector fields possess solely a non-zero z-component and are oriented in positive z-direction. Since there are more than one conductor, in addition to the skin effect, the so-call proximity physical effect can be observed. This leads to non-uniformly spatially distributed current density vector fields. The given spatial configuration of the conductors and the spatial direction of the currents results in shift of the current density distribution towards the outer regions of cross-sections of the outer conductors. Due the given spatial configuration of the conductors and the spatial direction of the currents results, the magnetic flux densities associated with each individual conductor are exhibiting different directions within the computational domain between both conductors, and are exhibiting same directions within the computational domain outside of the in-between area of both conductors. Hence, the total magnetix flux density's magnitude is lower within the computational domain between both conductors, and higher elsewhere. Finally, a shift of the current density distribution towards the outer regions of cross-sections of the conductors can be observed.

Appendix 2. LLM outputs

This appendix includes representative code examples generated by the LLM. Note that the LLM's output is originally a multiline string. Here, we show its cleaned version, that is, the version that is actually executed by Python (2.1) and GetDP (2.2 and 2.3).

2.1 LLM outputs for inferred python code to generate a list of coordinate tuples

The Python code below generates the conductor arrangement shown in [Figure 4](#) (as a consequence of user prompt (c)).

```

1 import numpy as np
2 from typing import List, Tuple
3 import matplotlib.pyplot as plt
4
5 def generate_hexagonal_grid_points(num_x: int, num_y: int, spacing: float) -> List[Tuple
  [float, float]]:
6     """Generates a list of 2D points arranged in a hexagonal grid.
7
8     Args:
9         num_x: The number of points in the x-direction.
10        num_y: The number of points in the y-direction.
11        spacing: The spacing between the points.
12
13    Returns:
14        A list of tuples, where each tuple is a 2D point (x, y).
15    """
16    points: List[Tuple[float, float]] = []
17    for i in range(num_x):
18        for j in range(num_y):
19            x = i * spacing
20            y = j * spacing
21            if i % 2 == 1:
22                y += spacing / 2
23            points.append((x, y))
24    return points
25
26
27 from simulation_functions.test_gmsh_api_given_list import run_simulation_from_terminal
28 num_x = 10
29 num_y = 10
30 spacing = 0.02
31 xy_list: List[Tuple[float, float]] = generate_hexagonal_grid_points(num_x, num_y,
    spacing)

```

2.2 LLM outputs for inferred domain-specific language code (including meaningful examples in the system prompt)

The GetDP code below generates the custom postprocessing routine for the ohmic power loss density p_{Ω} shown in [Figure 7](#) (as a consequence of user prompt (i)).

```

1 PostProcessing{
2     { Name MagDyn_b ; NameOfFormulation MagDyn_a ;
3       PostQuantity {
4         { Name OhmicLossDensity_conductor_4 ; Value { Local { [ sigma[]/2 * Norm
5           [ (- Dt[{a}] - {grad_phi}) ]^2 ] ; In Region[{Omega_c_4}] ; Jacobian
6             Vol ; } } }
7     }
8 }
9 PostOperation{
10    { Name MagDyn_b ; NameOfPostProcessing MagDyn_b ;
11      Operation {
12        Print[ OhmicLossDensity_conductor_4          , OnElementsOf Omega , File "
13          Results/p_V_conductor_selected.pos" , Name "p_V_c_4(xyz) [W/m^3] "
14            , Format Gmsh ] ;
15    }
16 }

```

2.3 LLM outputs for inferred domain-specific language code (excluding meaningful examples in the system prompt)

The GetDP code below generates the custom postprocessing routine for the magnetic energy density w_m shown on the right hand side of Figure 8 (as a consequence of the user prompt mentioned in Section 3.3).

662

```

1 PostProcessing{
2   {
3     Name MagDyn_c;
4     NameOfFormulation MagDyn_a;
5     PostQuantity {
6       {
7         Name MagneticEnergyDensity_Diagonal;
8         Value {
9           Local {
10            [
11              0.25 * nu[] * Norm[{d a}]^2
12            ];
13            In Region[{Omega_c_1, Omega_c_5, Omega_c_9}];
14            Jacobian Vol;
15          }
16        }
17      }
18    }
19  }
20 }
21
22 PostOperation{
23   {
24     Name MagDyn_c;
25     NameOfPostProcessing MagDyn_c;
26     Operation {
27       Print[
28         MagneticEnergyDensity_Diagonal,
29         OnElementsOf Omega,
30         File "Results/magnetic_energy_density_diagonal.pos",
31         Name "w_m_diagonal(xyz) [J/m^3]",
32         Format Gmsh
33       ];
34     }
35   }
36 }

```

Corresponding author

Albert Piwonski can be contacted at: a.piwonski@tu-berlin.de