Omega – Harnessing the Power of Large Language Models for Bioimage Analysis

Loïc A. Royer[1,*]

[1]Chan Zuckerberg Biohub, San Francisco, USA.
*Correspondence: loic.royer@czbiohub.org

Abstract:
In this study, we present Omega, an advanced conversational agent based on Large Language Models (LLMs) such as OpenAI's ChatGPT, capable of conducting image processing and analysis tasks through a human-friendly natural language interface. The system leverages the LangChain Python library and is integrated as a plugin to napari, allowing users to instruct complex tasks without requiring prior programming knowledge. Omega can also generate instructional strategies, make interface widgets upon request, and rectify its own coding mistakes. Its abilities extend to executing Python code, controlling the napari viewer, querying documentation, and utilizing hardware acceleration libraries. The presented approach, while in its early stages, hints at a future where language-based agents could revolutionize how we interact with and use complex computational systems, making them more accessible to a broader audience.

We stand at a pivotal juncture for Artificial Intelligence. Large Language Models (LLMs) such as ChatGPT[1–3] are now capable of engaging in insightful conversations, demonstrating an impressive command of human knowledge and logic. These large language models (LLM) are not only adept conversationalists but also possess an increasingly accurate understanding of numerous scientific and engineering disciplines[4]. Notably, ChatGPT, a groundbreaking model from OpenAI, can not only code in various programming languages but also explain and rectify provided code[5]. This capability is ushering in an era where users, irrespective of their programming skills, can instruct computers to perform complex tasks that would have previously required coding. Natural language is emerging as the hot new programming language. Could we harness these advancements to make image processing and analysis faster, more accessible, and tailor-made to the user's task?

We introduce Omega, an LLM-based conversational agent capable of performing image processing tasks, analyzing images to gather insights, correcting its own coding mistakes, and conducting follow-up quantifications and analyses. For instance, a user can instruct Omega to “segment cell nuclei in the selected image on the napari viewer,” then “count the number of segmented nuclei,” and finally “return a table that lists the nuclei, their positions, and areas” (see Fig. 1, and Supp Video 1 and 2). Moreover, Omega can provide advice and instructions on various image processing and analysis topics. A user can ask Omega to create a “step-by-step plan to segment nuclei in an image,” and Omega will generate a detailed strategy (see Supp. Video 3). The user can then interactively apply these steps, make changes in response to the outcomes, and ask follow-up questions to complete the task (see also Supp. Video 3). Omega can also create on-demand user interface widgets from user prompts. For example, a user may ask for a “widget that removes segments in a labels layer outside of a range of segment areas” (see Fig 1 and Supp. Video 4) or for a “widget that adds a scale-bar of length 20 µm given a pixel resolution of 0.400 µm.” (See Fig 1, and Supp. Video 5).

Omega is written in Python as a plugin to napari[6] and leverages the LangChain Python library[7] and OpenAI's application programming interface (API). While Omega works best with OpenAI's ChatGPT, it can also leverage other LLMs, such as Anthropic's Claude models (see Supp. Video 6). Omega is a conversational agent that can converse with the user, like ChatGPT's popular web interface. We utilize the ReAct framework[8] to enable

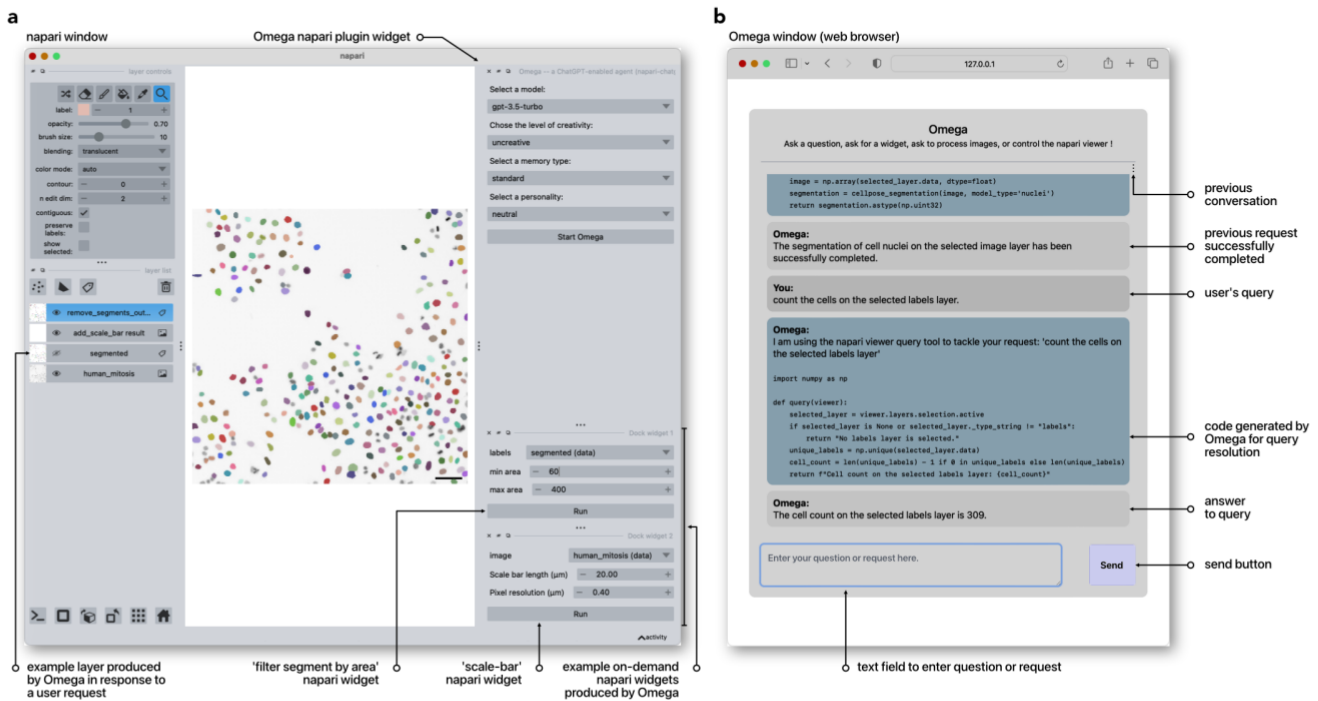


Figure 1. Harnessing the Power of Large Language Models for Bioimage Analysis with Omega. **(a)** Omega is a napari plugin that first appears as a widget that lets users configure and start Omega. Images and other layers (labels, points, shapes) are listed in napari's layer list and accessible to Omega. Omega can add to the layer's list any layer resulting from processing or analysis. Users can ask Omega to make tailor-made widgets that are added to napari. These widgets can input any set of layers and return new layers. **(b)** Upon starting, Omega opens a browser window that displays a chat box page. Users can then begin dialoguing with Omega, asking questions about image processing and analysis, opening images in the napari viewer, asking for a widget, and processing and analyzing images or any other layer supported by napari, such as labels, points, or shapes.

multi-step reasoning and task-specific actions, including access to online sources of information and specialized tools for executing code and interfacing with napari. Omega can also correct its own coding mistakes by receiving feedback on encountered syntax and execution errors (Supp. Video 7).

Omega's tools allow it to download files from the web (Supp. Video 8), perform web searches (Supp. Video 9), execute arbitrary Python code (Supp. Video 10), control the napari viewer (Supp. Video 11), and query the parameters and documentation of Python functions (Supp. Video 12). Additional tools incorporated into Omega include special-purpose tools that give access to two popular cell and nuclei segmentation algorithms: *cellpose*[9] (Supp. Video 13) and *StarDist*[10] (Supp. Video 1 and 4), as well as to our denoising software *Aydin*[11] (Supp. Video 14). Importantly, Omega inherits ChatGPT's Python coding abilities and knowledge (Supp. Video 15). To our surprise, the two LLMs tested, ChatGPT and Claude, have extensive knowledge of napari's programming interface (Supp. Video 11) as well as other standard and popular Python libraries such as NumPy[12] (Supp. Video 15), scikit-image[13] (Supp. Video 16), and OpenCV[14] (Supp. Video 17). These models can also utilize hardware optimization and acceleration libraries such as *numba*[15] for just-in-time compilation (Supp. Video 18) and CuPy[16] for GPU acceleration (Supp. Video 19). Omega can leverage all this knowledge and tools to perform tasks and answer questions.

However, the promise of this approach also requires prudence. It is well known that LLMs sometimes hallucinate facts and occasionally make trivial reasoning mistakes[4,17]. Indeed, we have observed that ChatGPT sometimes uses functions that do not exist in the standard libraries. This is cause for caution because non-expert users might be led astray by an overly confident agent. Moreover, it is incumbent upon the user to explain the task clearly and unambiguously in natural

language. We are still in the early days of this technology, and rapid progress will hopefully reduce the risks and improve the quality of the reasoning and code produced[18]. In the meantime, Omega implements several features that aim to mitigate these problems. Omega implements several introspection routines that check the correctness of generated code by looking for function calls to the wrong library versions, missing import statements, or missing libraries.

Looking ahead, Omega could be extended with the ability to understand and produce speech to facilitate interaction. Moreover, most currently available LLMs are 'blind'. They can't see the content in images and must rely on our description of what images contain when reasoning about how to solve a task. The advent of models capable of ingesting images in addition to text could address this issue [19,20]. Another approach is to combine LLMs with specialized image-centric and generalizable models, such as the zero-shot generalization Segment Anything Model (SAM[21]) or other specialized models to extend Omega's capabilities further.

This work suggests that LLM-based agents could assist many users in image processing, analysis, and visualization. Beyond just completing tasks, Omega offers an interactive platform that can assist in educating users. Users can ask questions about a particular course of action, why a specific function was used, or for an explanation of some of the concepts used or mentioned by Omega (Supp. Video 4 & 16). Moreover, the multilingual capabilities of ChatGPT and other LLMs mean that Omega is accessible to non-English speakers, which could broaden accessibility and dissemination to underserved communities (Supp. Video 20).

The source code and instructions to use Omega are available at github.com/royerlab/napari-chatgpt.

Acknowledgments:

Thanks to: OpenAI for early API access to their ChatGPT 4 models, facilitated by Logan Kilpatrick via Mark Andrew Kittisopikul; Anthropic for early API access to their latest Claude models, facilitated by Josh Batson; and Sandra Schmid for careful proofreading.

Ethics Declaration:

The author declares no conflict of interest.

References:

1. OpenAI. GPT-4 Technical Report. Preprint at http://arxiv.org/abs/2303.08774 (2023).
2. Ouyang, L. *et al.* Training language models to follow instructions with human feedback. Preprint at http://arxiv.org/abs/2203.02155 (2022).
3. Sanderson, K. GPT-4 is here: what scientists think. *Nature* **615**, 773–773 (2023).
4. Laskar, M. T. R. *et al.* A Systematic Study and Comprehensive Evaluation of ChatGPT on Benchmark Datasets. Preprint at http://arxiv.org/abs/2305.18486 (2023).
5. Bubeck, S. *et al.* Sparks of Artificial General Intelligence: Early experiments with GPT-4. Preprint at http://arxiv.org/abs/2303.12712 (2023).
6. Sofroniew, N. *et al.* napari: a multi-dimensional image viewer for Python. *Zenodo* (2022).
7. Chase, H. LangChain, 10 2022. *URL Httpsgithub Comhwchase17langchain*.
8. Yao, S. *et al.* ReAct: Synergizing Reasoning and Acting in Language Models. Preprint at http://arxiv.org/abs/2210.03629 (2023).
9. Pachitariu, M. & Stringer, C. Cellpose 2.0: how to train your own model. *Nat. Methods* **19**, 1634–1641 (2022).
10. Weigert, M., Schmidt, U., Haase, R., Sugawara, K. & Myers, G. Star-convex Polyhedra for 3D Object Detection and Segmentation in Microscopy. in *2020 IEEE Winter Conference on Applications of Computer Vision (WACV)* 3655–3662 (IEEE, 2020). doi:10.1109/WACV45572.2020.9093435.
11. Solak, A. C., Loic A. Royer, Abdur-Rahmaan Janhangeer & Kobayashi, H. royerlab/aydin: v0.1.15. (2022) doi:10.5281/ZENODO.5654826.
12. Harris, C. R. *et al.* Array programming with NumPy. *Nature* **585**, 357–362 (2020).
13. Van der Walt, S. *et al.* scikit-image: image processing in Python. *PeerJ* **2**, e453 (2014).
14. Bradski, G. The openCV library. *Dr Dobbs J. Softw. Tools Prof. Program.* **25**, 120–123 (2000).
15. Lam, S. K., Pitrou, A. & Seibert, S. Numba: A llvm-based python jit compiler. in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC* 1–6 (2015).

16. Nishino, R. & Loomis, S. H. C. Cupy: A numpy-compatible library for nvidia gpu calculations. *31st Confernce Neural Inf. Process. Syst.* **151**, (2017).
17. Li, J., Cheng, X., Zhao, W. X., Nie, J.-Y. & Wen, J.-R. HaluEval: A Large-Scale Hallucination Evaluation Benchmark for Large Language Models. Preprint at http://arxiv.org/abs/2305.11747 (2023).
18. Peng, B. *et al.* Check your facts and try again: Improving large language models with external knowledge and automated feedback. *ArXiv Prepr. ArXiv230212813* (2023).
19. Wu, C. *et al.* Visual ChatGPT: Talking, Drawing and Editing with Visual Foundation Models. (2023) doi:10.48550/ARXIV.2303.04671.
20. Royer, L. A. The future of bioimage analysis: a dialog between mind and machine. *Nat. Methods* **20**, 951–952 (2023).
21. Kirillov, A. *et al.* Segment Anything. Preprint at http://arxiv.org/abs/2304.02643 (2023).
22. Kojima, T., Gu, S. S., Reid, M., Matsuo, Y. & Iwasawa, Y. Large Language Models are Zero-Shot Reasoners. Preprint at http://arxiv.org/abs/2205.11916 (2023).

Methods:

In the following, we provide details on how Omega is implemented: from the chat window web server, the cascaded LLM approach to the Python code repair strategies, and more. We can't possibly paragraph the entirety of the code and explain all the tricks that make Omega possible. Therefore, the best and most up-to-date description of how Omega works is simply the code itself, which can be consulted at:
https://github.com/royerlab/napari-chatgpt.

Overall Architecture. Omega is provided as a napari plugin following the latest plugin standard (npe2). The plugin was built by following the instructions described here. Omega's 'widget plugin' provides a simple interface that lets users configure different options and start Omega (see Fig 1a). Once the user starts Omega (see start button on Supp. Fig. 1), the plugin starts a web server at the local address 127.0.0.1 that hosts the Omega Chat page. The plugin then opens a web browser window at that address to display the page. At that point, the user can start dialoguing with Omega. Ideally, this chat window is side-by-side with the napari window so that the user can see both the conversation displayed on the browser and the outcome shown in the viewer. We typically use two LLM instances (via OpenAI or Anthropic APIs): one for dialog and the second for code generation. The LLM instance used for code generation has a temperature setting set to one order of magnitude less than the one for dialog.

Omega configuration. The Omega widget (see Supp. Fig. 1) allows users to set: (i) the LLM model (GPT3.5, GPT4, Claude, etc.); (ii) the model's creativity level (normal, slightly creative, moderately creative, creative) that corresponds to different temperature settings of the LLM model. A temperature near zero (normal creativity) means that the model is nearly deterministic in its answers – which is desirable in factual dialoguing and code generation. When the temperature increases, the LLM models explore more atypical and often creative responses – but they also tend to make more factual, reasoning, and coding mistakes. (iii) The type of agent memory used (infinite, bounded, and hybrid). In the case of infinite memory, the agent remembers every word of the conversation, which in practice only works for LLM models with extensive input text lengths such as GPT4 (32k) or Claude (100k). In contrast, the bounded memory only remembers the last k messages exchanged between the agent and user. The hybrid memory precisely remembers the last k messages and summarizes all previous messages. (iv) The agent's personality (neutral, coder, prof, yoda, mobster) modulates the style and tone of the conversation. The following options are related to code generation and to the different strategies adopted to mitigate and prevent errors: (v) The option "fix missing imports" controls whether to check the generated code, identify missing imports, and prepend them to the code. (vi) the option "fix bad function calls" controls whether to verify if the function calls present in the generated code correspond to functions that exist in the packages installed in the Python environment. (vii) the "Install missing packages" controls whether to list all Python packages required by generated code, compare that list with the list of installed packages, and proceed to install those missing. (viii) The "Autofix coding mistakes" option controls whether Omega will try to fix its own coding mistakes when exceptions occur when interacting with the napari viewer. Similarly, (ix) the "Autofix widget coding

mistakes" controls whether Omega will try to fix its own coding mistakes when exceptions occur while making a new widget. Finally, (x) the last option, "High console verbosity," controls Omega's console verbosity level.

Chat server. The chat page is served by uvicorn – an ASGI web server implementation for Python, and uses FastAPI for communication between the chat box and Python. It leverages Jinja2 as the template engine for generating the served HTML page. The chat box and Python communications are handled via a web socket on the client side and a FastAPI endpoint on the server side. Messages between the agent and user are exchanged as JSON-encoded dictionaries.

Omega ReAct Agent. Omega is implemented as LangChain's *ConversationalChatAgent,* which uses the ReAct framework[8] to decide which tool to use and uses memory to remember the previous messages in the conversation. By default, Omega uses a modified version of LangChain's hybrid conversational memory (see code here).

Prompt engineering for Python code generation. The building of Omega required much effort in "Prompt Engineering," which is the art of designing prompts that nudge LLMs into producing the correct answers expected by users. LLMs are known to require very explicit – if not obvious – instructions. For instance, simply adding to the prompt: "Let's think step by step," improves the quality of results[22]. In the case of Omega, we had to make explicit that: "You are an expert Python programmer with deep expertise in image processing and analysis," that: "Your responses are accurate and informative," that it should "Make sure that the code is correct, complete and functional without any missing code, data, or calculations". LLM prompts used for code generation also contain the current Python version number and the names and versions of all image-processing relevant libraries installed in the environment. Our experiments show that ChatGPT knows about differences in the parameters of a function across different package versions. This means that explicitly providing the information about which specific library versions are installed is critical for correct code generation. We also had to provide detailed instructions so that the code generated could be easily interfaced with Omega's code, thus facilitating interaction with the napari viewer. A simple strategy is to ask the LLM to produce a function with a well-defined signature (input parameters, their types, and return type) and load the code dynamically as a Python module.

Omega's Tools. Omega has at its disposal several tools that give it the ability to: (i) search text and images on the web and Wikipedia, access a Python REPL (Read-Eval-Print Loop) for executing arbitrarily non-napari related code, (ii) gather detailed information about Python functions available in the environment, (iii) get information about the latest exceptions that occurred, (iv) obtain information about the state of the napari viewer and about the layers present, (v) make and add widgets to napari's UI, and (vi) use special-purpose libraries for cell segmentation and image denoising. Following the ReAct[8] approach, the agent maintains a conversation with the user and can use tools to answer questions or perform tasks. This is achieved by listing the available tools and their description in the prompt sent to the LLM. Part of the dialog related to tool usage is internal to the agent and not shared with the user. In Omega, we choose a cascaded LLM approach where, besides the main ReAct agent, the tools invoke LLMs to generate and introspect code and summarize the text. This avoids polluting the main dialog loop trace with long pieces of generated code and makes it possible to tailor prompts to each code generation task.

Custom protocol for tool communication. Most conversational ReAct-based agents use JSON-formatted dictionaries to allow communication between LLMs and the tools. This works well when simple short text strings are exchanged between the LLM and the tool, but this fails for arbitrary code because of all the complexities entailed, such as escaping reserved characters. Imposing such a high competence bar on the LLM by requiring it to produce a very complex JSON string is, therefore, unreasonable. To solve this issue and reduce the complexity of the syntax that the LLM has to adhere to, we use a simplified multi-line key-value format (see code here).

Python code introspection and repair. Careful prompt engineering can be highly effective at ensuring that the generated code is synthetically correct, that function calls refer to existing and available functions, and that the code is interfaceable with the rest of Omega's code. However, despite our best efforts, there are cases in which the generated code is incorrect. Omega implements several mitigation strategies that reduce the probability of error.

Adding missing import statements. The first typical type of code generation error that we noticed is missing import statements. To address this, Omega implements a particular routine using the code generation LLM to introspect the code by listing all missing import statements. This might seem paradoxical: why should the LLM make a mistake during code generation but be able to catch it during verification? The explanation is that code generation is more challenging than code verification because it requires both Python and Application domain knowledge. In contrast, code verification only requires knowledge of Python and its libraries – generally, the more restricted and well-defined the task, the better the outcome.

Installing missing packages. Adding missing import statements is only helpful if the corresponding libraries are installed in the Python environment. Using the same code introspection approach, we ask the code generation LLM instance to list all Python packages required to run the generated code. This list of packages is compared to installed packages, and only missing packages are installed using pip. Edge cases like GPU accelerated libraries like Tensorflow or CuPy are handled with special rules. Future versions of Omega might implement a feature by which the user is asked permission to install packages.

Incorrect function call detection and repair. An additional mitigation approach involves enumerating all function calls occurring in the code using standard Python language introspection features and checking that these functions exist and that the corresponding packages are installed. This detection step is highly reliable because it does not use LLMs. Once an incorrect function call is detected, we carefully construct a specialized prompt that combines all the information, particularly the correct function signature and asks the LLM code generation instance to fix the code accordingly.

Context-aware code repair upon code execution error. Once the above fixes are applied to the generated code, then Omega executes that code. If exceptions are detected during execution, the code and exception(s) are provided to the code generation LLM instance. With a specialized prompt, the LLM is explicitly asked to fix the code, given a detailed error description. This process can be repeated recursively until no exceptions occur or the maximum number of repair steps has been reached. Importantly, Omega has a mechanism so that task-specific coding instructions used during code generation are also available during code repair.

Napari integration and communication. Giving Omega access to the napari viewer is not trivial because of the threading model mismatch between the agent controller and the napari viewer. The agent runs in async mode per LangChain's implementation, while napari's threading model is inherited from the Qt cross-platform application framework. We address this using a thread-safe bidirectional asynchronous communication queue that establishes a bridge between Omega and napari (see the `NapariBridge` class here). The queue passes code as a string and then executes that code using napari's thread-worker functionality. All captured standard output strings are captured and returned to Omega. Exceptions are dealt with according to the error mitigation strategies described above. The following details the tools that provide Omega access to napari.

Napari viewer query tool. This tool lets Omega gather any information about the state of the napari viewer or any of the layers (images, labels, points, etc.) currently loaded. This is achieved by carefully crafting a prompt incorporating the user's question, information on layers that are present in the viewer, and task-specific coding instructions. In this prompt, the code generation LLM instance is asked to write a '`query(viewer: Viewer)`' function that takes the viewer as a parameter and prints out the answer to the user's question.

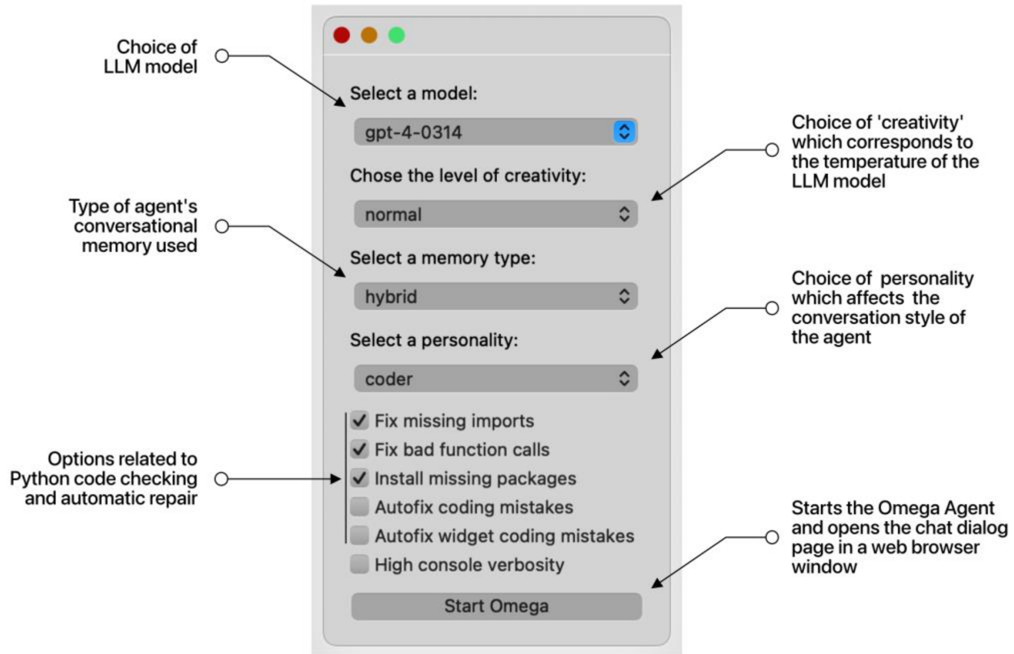
Napari viewer control tool. Similarly, this tool lets Omega control the napari viewer, such as changing the state of the viewer's canvas, adding, and removing layers, etc. In that case, we carefully crafted a prompt incorporating the user's question, information on layers that are present in the viewer, and task-specific coding instructions. In this prompt, the code generation LLM instance is asked to write a script that is then executed.

Widget maker tool. This tool takes the user's plain text description of a widget or instructions on modifying a previously generated widget and adds that widget to napari's user interface. For instance, if the user asks for a 'Gaussian filter with a sigma parameter,' this tool will make the corresponding widget with a single float parameter. This automatic user interface generation is made possible by the MagicGUI library as part of the standard plugin infrastructure of napari. The generated code goes through all the checks and verifications described above.

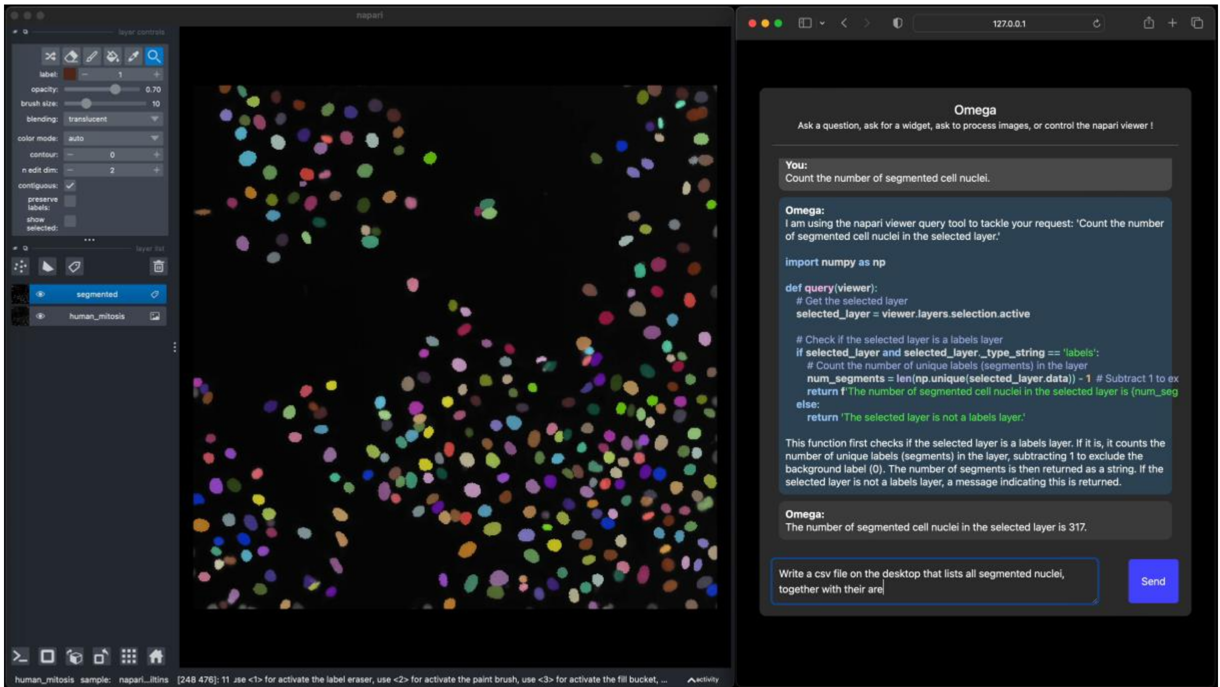
Cell segmentation and image denoising tools. We can't expect LLMs, even the best ones such as ChatGPT 4 or Anthropic's Claude, to know about the latest version of state-of-the-art bioimage analysis libraries such as Cellpose and StarDist for cell and nuclei segmentation and Aydin (aydin.app) for image denoising. To ensure their availability and facilitate their usage, the integration of these libraries is done explicitly through specific interfacing functions that expose a subset of relevant parameters from these libraries. Specially crafted prompts explain in detail how to use these functions, how to choose between the different variants, and how to set the parameters. What is remarkable is that the prompts need to be very explicit and clear – as if one had to carefully explain to a colleague how to use these functions and provide context and examples.

'Classic' cell segmentation. In addition to Cellpose and StarDist, we implemented a straightforward yet configurable threshold-based 'classic' segmentation algorithm using scikit-image functions. This simple algorithm is a reasonably practical baseline for segmenting 3D images of nuclei. Images are first normalized, then using a disk- or ball-based footprint of radius 2; the image is eroded several times. Next, one of the following thresholding functions is applied: otsu, yen, li, minimum, triangle, mean, or isodata. The LLM makes the choice based on user prompt instructions. Next, the closing and opening operators are applied several times to remove potential small segments. At this point, an optional routine is available that uses watersheds to split the under-segmented segments. Finally, the resulting binary image is labeled, and a label layer is returned and added to napari.

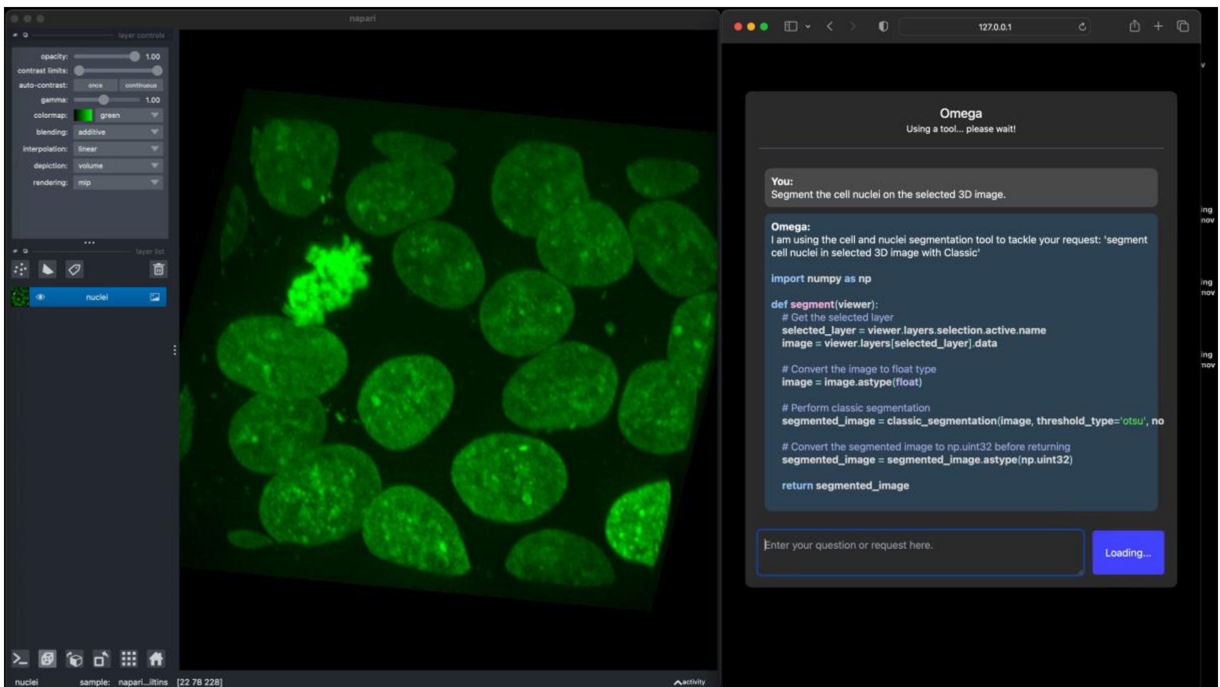
Supplementary material:



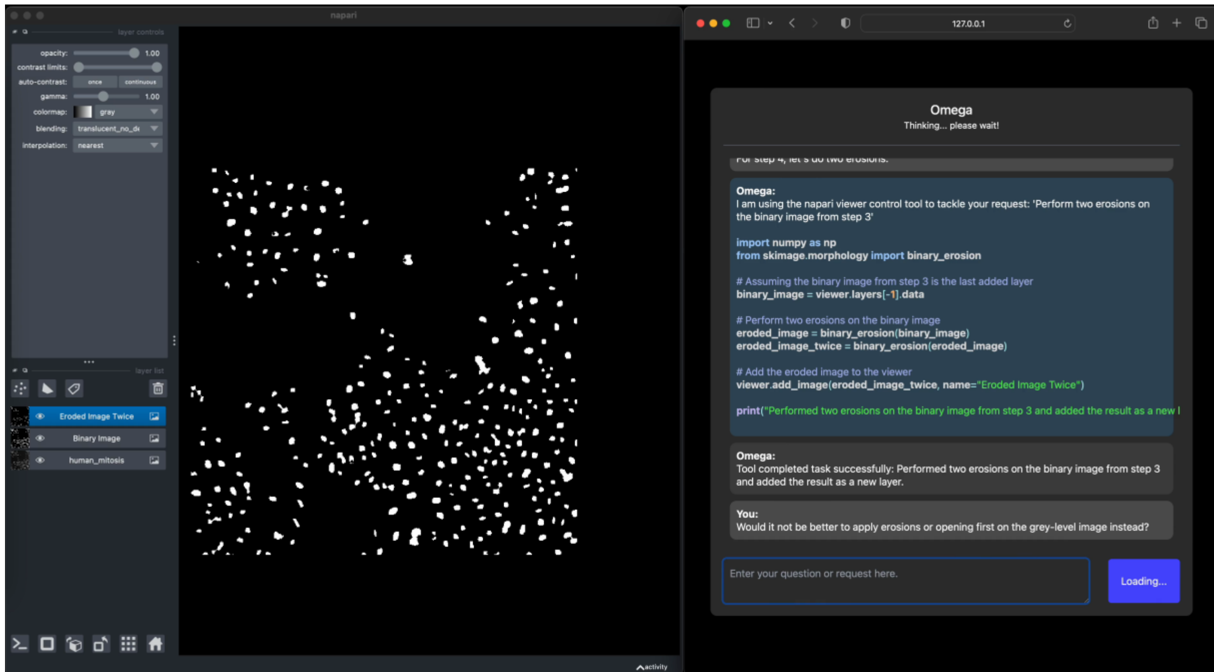
Supplementary Figure 1. Users can use Omega's main widget to select different options, including the LLM model's type and version, the level of creativity (which increases the model's temperature), the type of conversational memory used, and the agent's personality. Other parameters relate to code checking and automatic repair. To begin using Omega, simply click on the "Start Omega" button, and a browser window will open, displaying the agent's chat box.



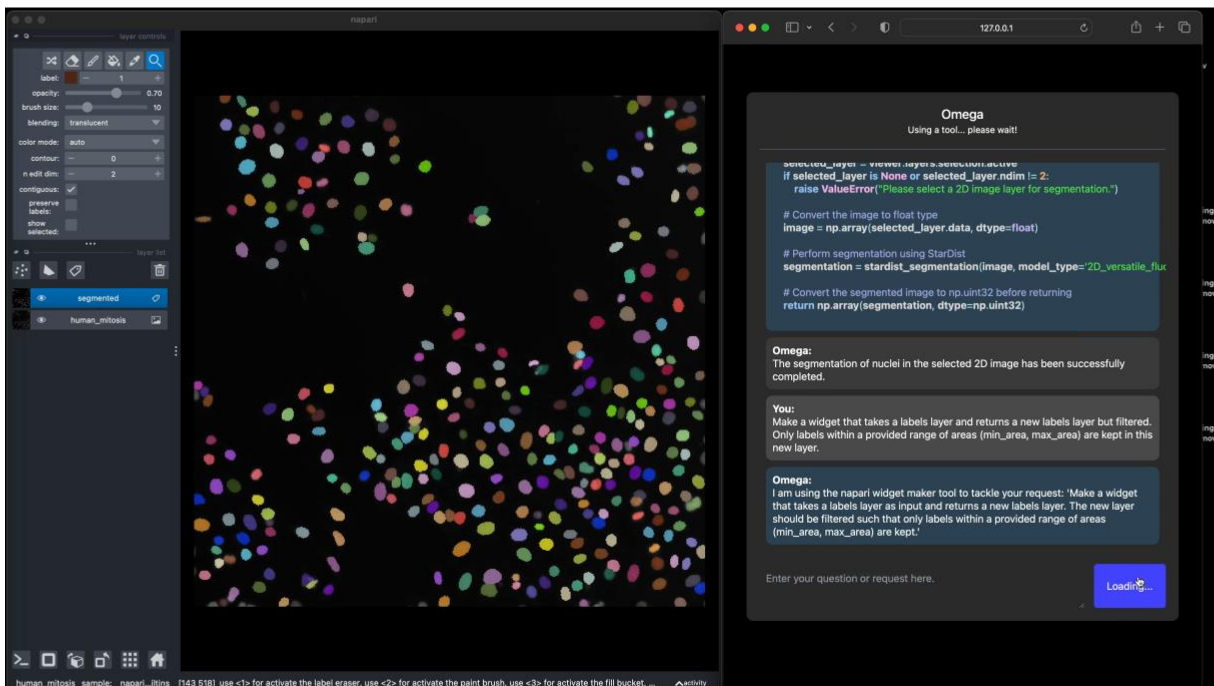
Supplementary Video 1. Omega can segment nuclei with StarDist and perform follow-up analysis. The video showcases Omega's ability to segment cell nuclei in a 2D image using Stardist. Omega successfully segments the nuclei and adds a label layer to the napari viewer. With further instructions, Omega can count the segmented nuclei and create a CSV file on the desktop folder of the machine. This file contains coordinates and areas of all segments, sorted by decreasing area, with one segment per row. Omega also opens the file using the system's default CSV viewer. The video has been sped up by a factor of 2.



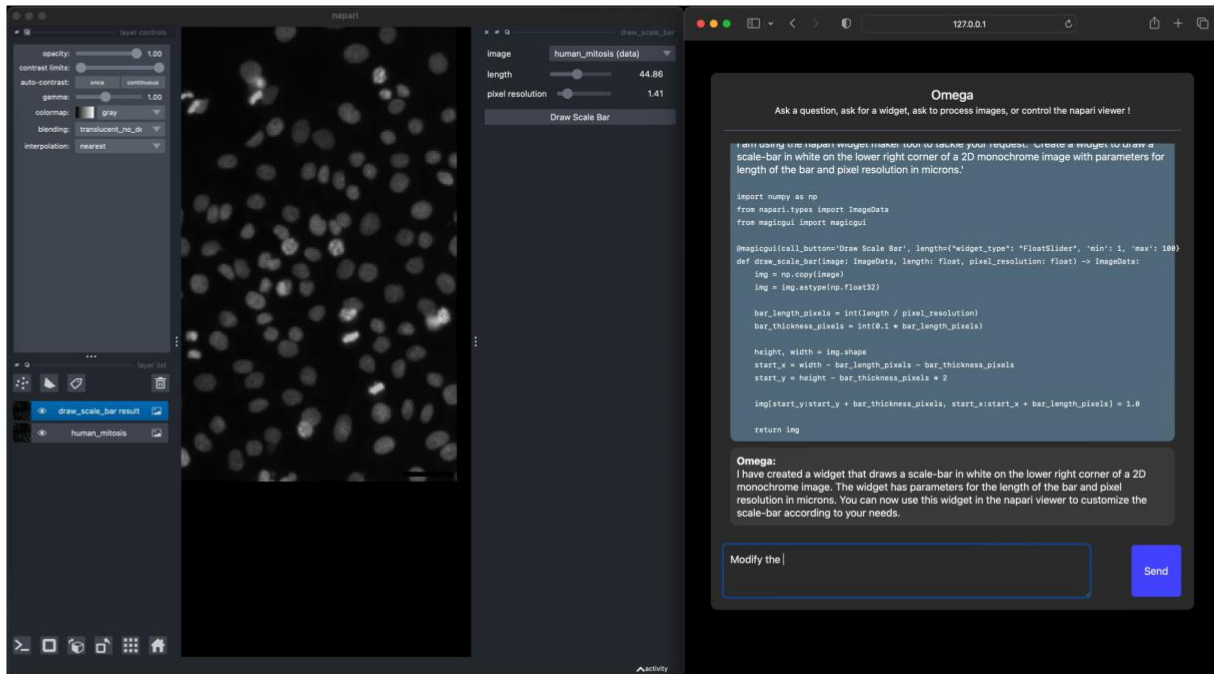
Supplementary Video 2. Omega can segment nuclei in a 3D image. This video shows how Omega segments the nuclei in a 3D image displayed in the napari viewer. Omega uses a specialized tool for cell and nuclei segmentation and employs a 'classic' approach that combines single thresholding, specifically Otsu, with watershed splitting to prevent under-segmentation. After segmentation, Omega adds a labels layer to the viewer, and we inquire about the number of segments detected. The response is 27. The video has been sped up by a factor of 2.



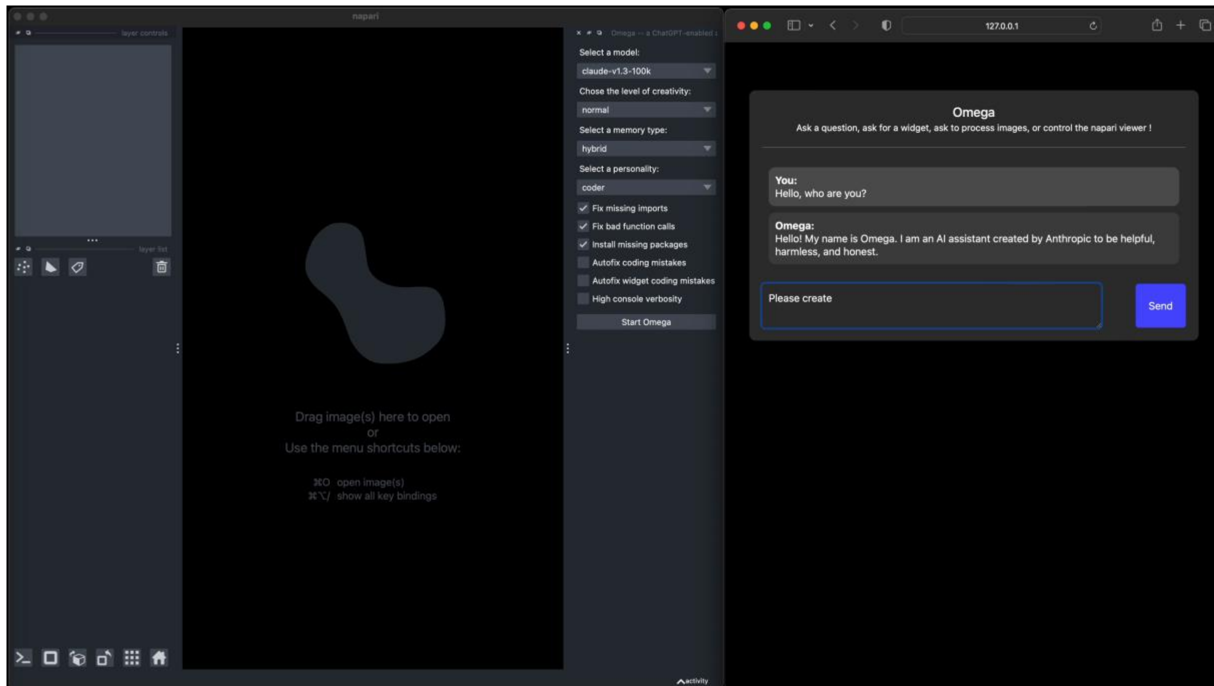
Supplementary Video 3. Omega can devise step-by-step strategies and interactively execute them. In this video, we requested Omega's assistance developing a detailed strategy for segmenting nuclei in a 2D image. We clarified that the nuclei appear brighter than the background. Omega provided us with a 6-step plan. The first step involved loading the image into napari, which was already done. Next, Omega suggested applying a Gaussian filter to smoothen the image and eliminate noise. However, since the image was not noisy, we asked Omega to move on to step 3, which involved thresholding. Using the scikit-image library, Omega utilized the Otsu method to determine the threshold value and change the image to binary form. As a result, a new layer was added to the viewer with the outcome. We then asked Omega to implement step 4, which involved morphological operations to remove minor artifacts and separate touching nuclei. We specifically requested two erosions. However, we were unsure whether applying grey morphology operators to the original would be more sensible. Omega agreed and provided us with an updated plan that swapped the order of thresholding and erosion. We started over and used the new plan, beginning with step 3 and proceeding to steps 4 and 5, resulting in a reasonably good segmentation. The video has been sped up by a factor of 2.



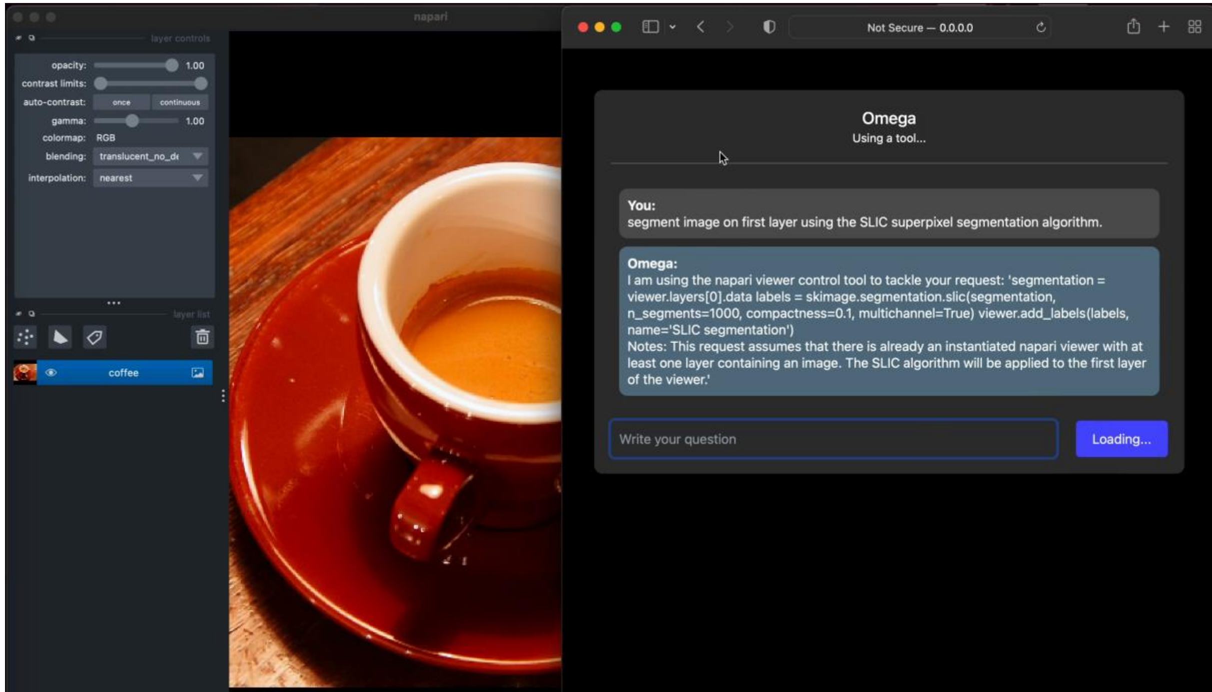
Supplementary Video 4. Omega can make widgets on demand, e.g., to filter segments per area. In this video, we first ask that Omega segment the nuclei in the currently selected 2D image. Then, we tell Omega to make a widget that can filter the segments in a label layer according to their area. Segments whose areas are outside of a given range are removed from the newly created labels layer. We then start using that widget and experiment with the two parameters: *min area* and *max area*. The video has been sped up by a factor of 2.



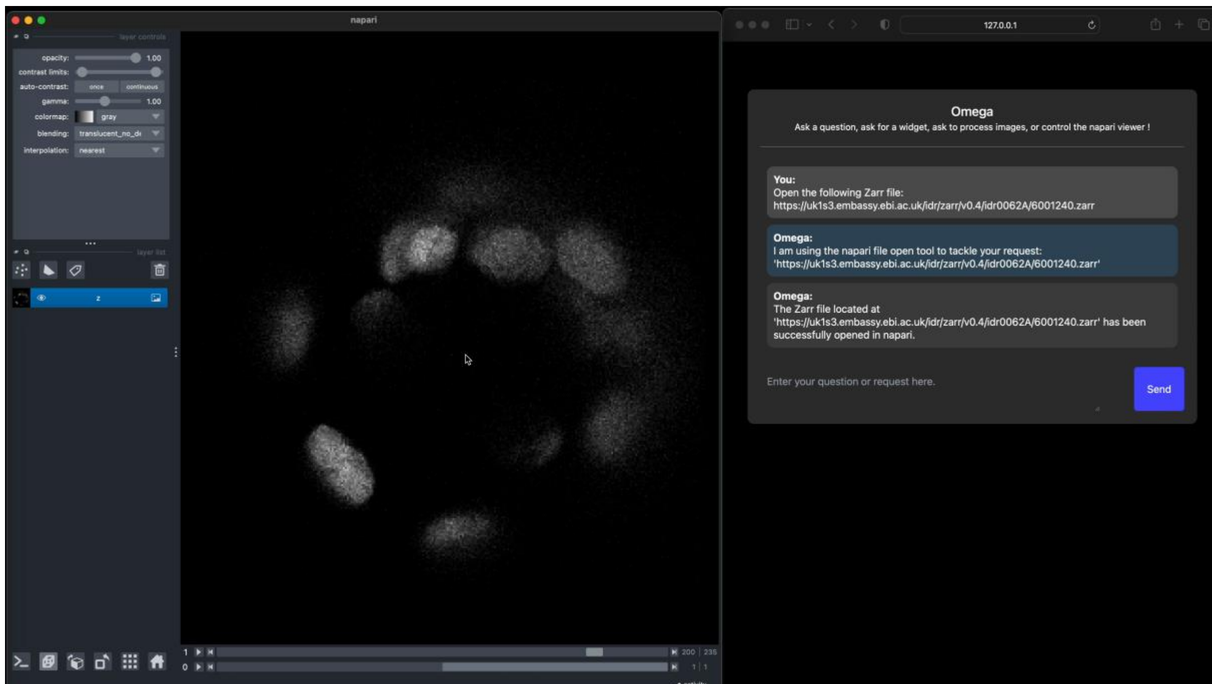
Supplementary Video 5. Omega can make complex widgets such as a widget that draws scale bars. In the video, we requested Omega create a widget for us to draw a scale bar on a 2D single-channel image. We explained that we needed to set the length of the scale bar and the pixel resolution in microns. Although it created a functional widget, it used a default pixel intensity of 1, unsuitable for our image, which had higher values thus making the scale bar very dark. We then requested Omega to make the same widget but with the option to choose the intensity of the scale bar. We achieved our desired outcome this time and successfully drew a satisfactory scale bar. The video has been sped up by a factor of 2.



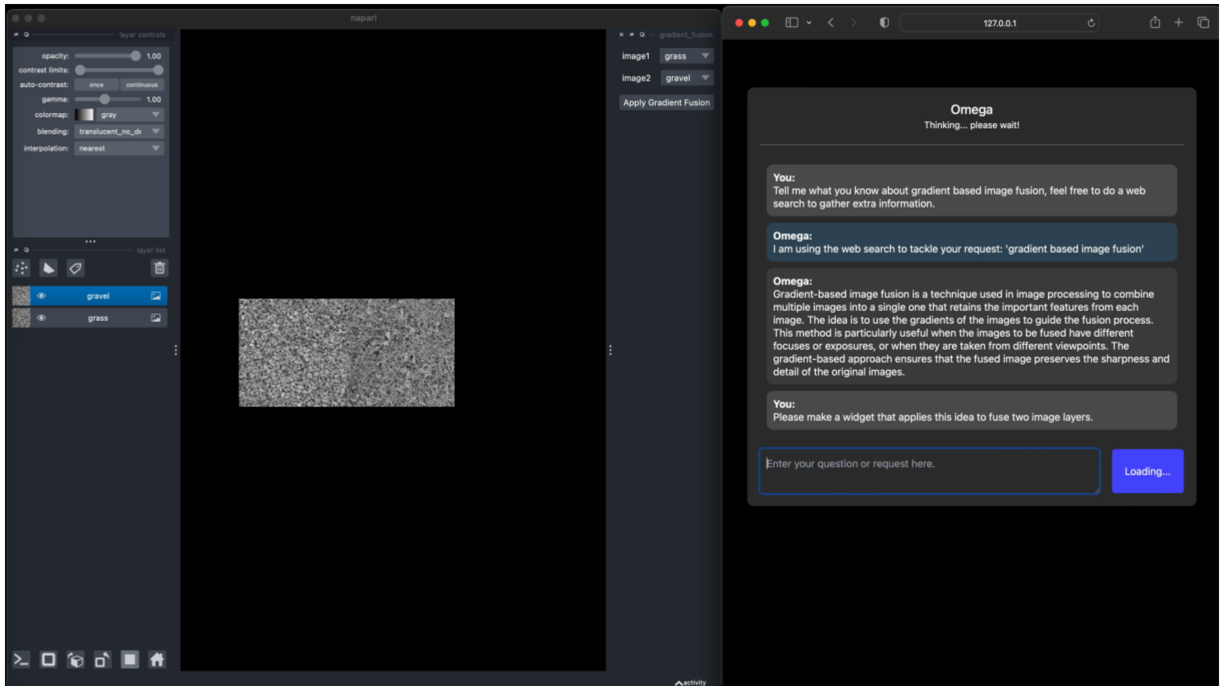
Supplementary Video 6. Omega can also work with other LLM models besides ChatGPT. This short video shows that Omega also works with Anthropic's Claude LLM model. The video has been sped up by a factor of 2.



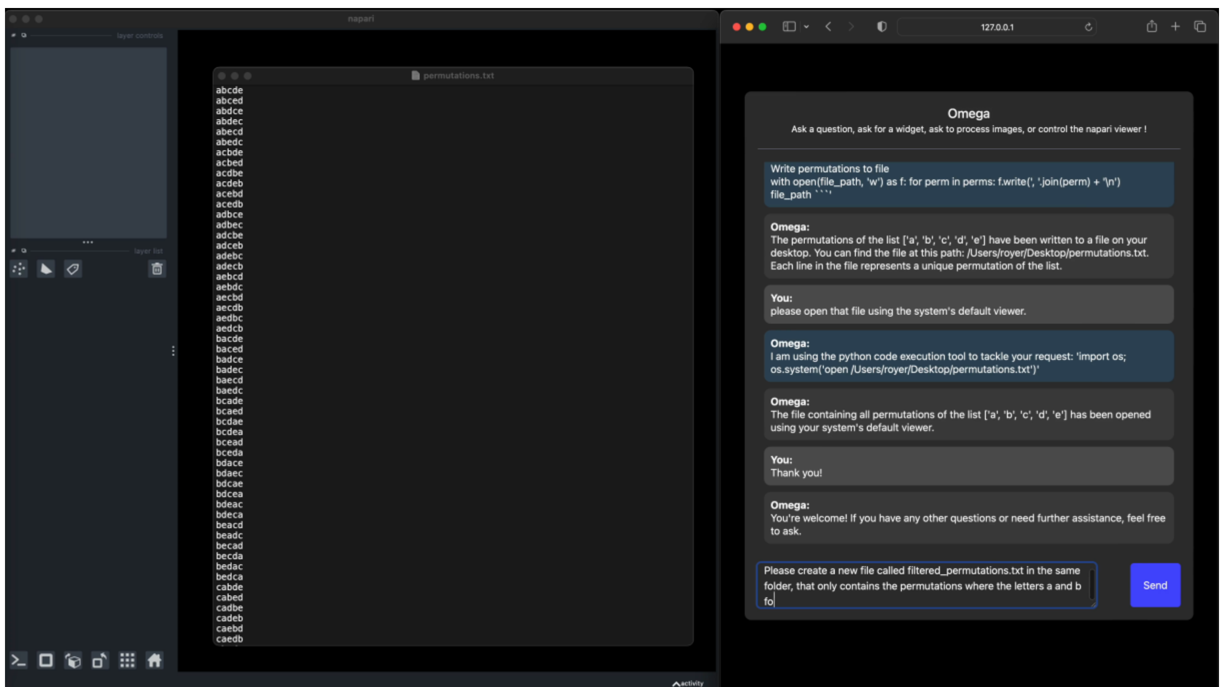
Supplementary Video 7. Omega corrects its own coding mistakes. In the video, Omega applied the SLIC super-pixel segmentation algorithm to a selected image. However, Omega made a mistake using the non-existent 'multichannel' parameter when using the scikit-image SLIC function, resulting in an error. Omega noticed this mistake and corrected it on the second try, successfully adding the segmented image to the viewer. The video has been sped up by a factor of 2.



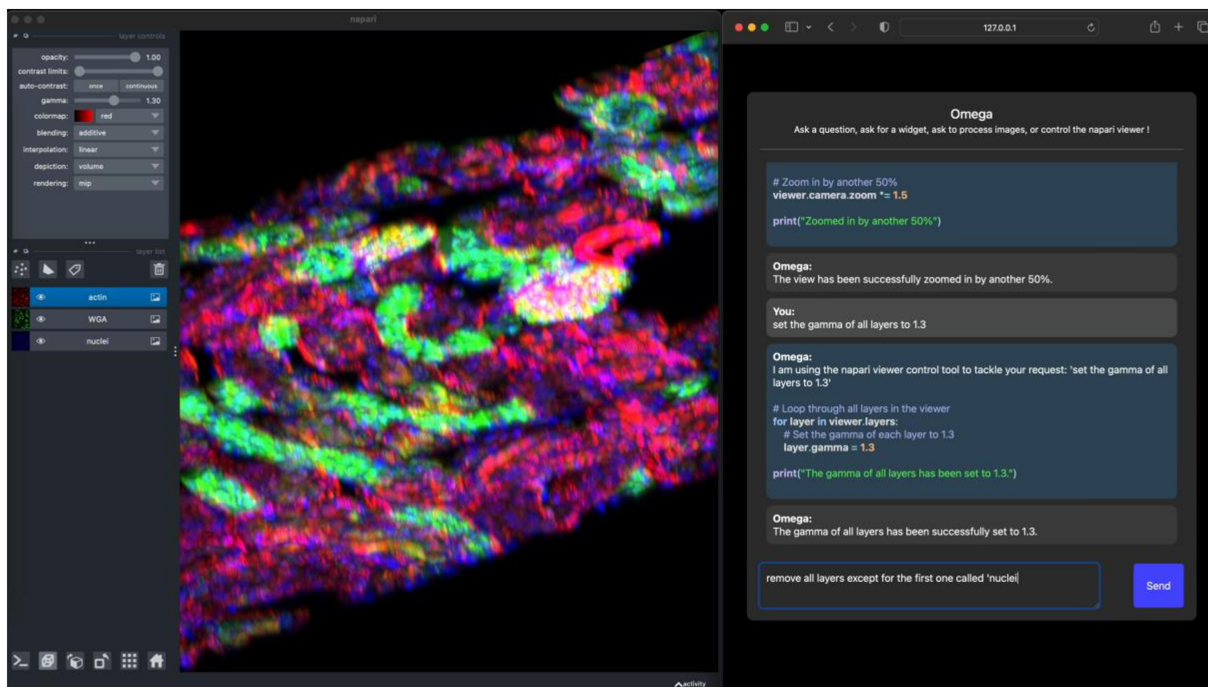
Supplementary Video 8. Omega can search and open image file from the web. In this video, we requested Omega to open a dataset from Blin et al.'s PLOS Biology 2019 in napari. The dataset can be accessed online and streamed using the ZARR image file format and library. Omega was able to fulfill our request successfully letting us explore the dataset. Next, we requested Omega to open a picture of Albert Einstein in napari. Omega then utilized its web image search function to locate a suitable image and loaded it into napari. The video has been sped up by a factor of 2.



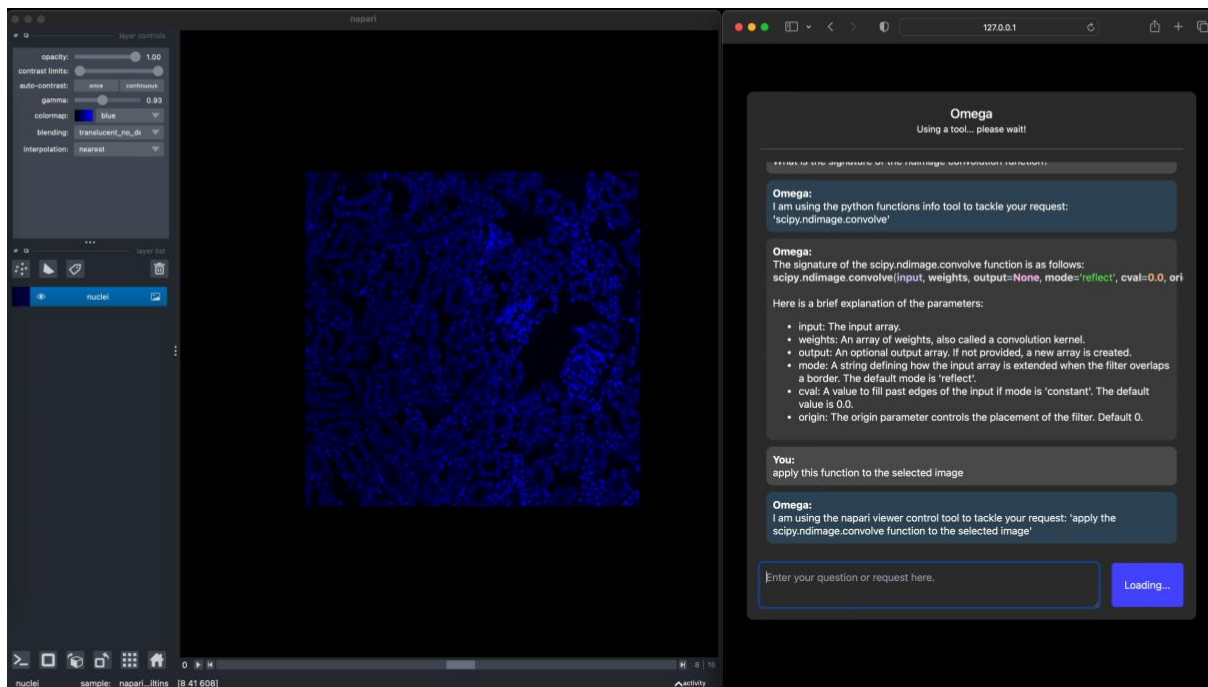
Supplementary Video 9. Omega can teach concepts in image processing. In this video, we ask Omega what it knows about 'gradient-based image fusion.' Omega then proceeds to give an interesting explanation of the general idea behind this approach to image fusion. We then ask Omega to apply these ideas and make a widget that takes two image layers and returns the gradient-based image fusion of these two images. Omega successfully creates a functional widget that we test on two images. The video has been sped up by a factor of 2.



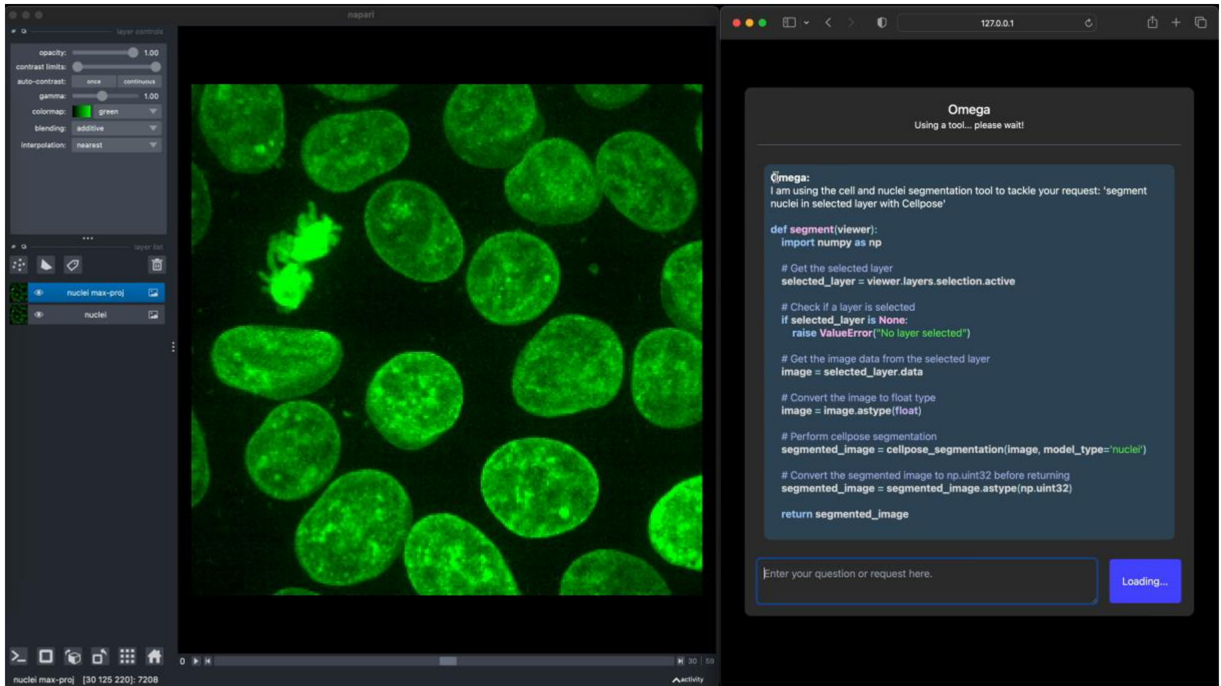
Supplementary Video 10. Omega can do math and write arbitrary Python code. In this video, we test Omega's Python coding skills by asking some basic math questions. For example, we asked for the value of 10^{10}+1 and the number of permutations possible with ten objects. Then, we asked Omega to write all permutations of a list of 5 strings ('a', 'b', 'c', 'd', 'e') to a file on the machine's Desktop folder, with one permutation per row. Omega completed this task and opened the file using the system's default text viewer. Following this, we asked to create a new file containing only permutations where the letters 'a' and 'b' are consecutive, providing some examples. However, we soon realized that our statement could have been clearer as it was ambiguous whether the order of 'a' and 'b' mattered. The video has been sped up by a factor of 2.



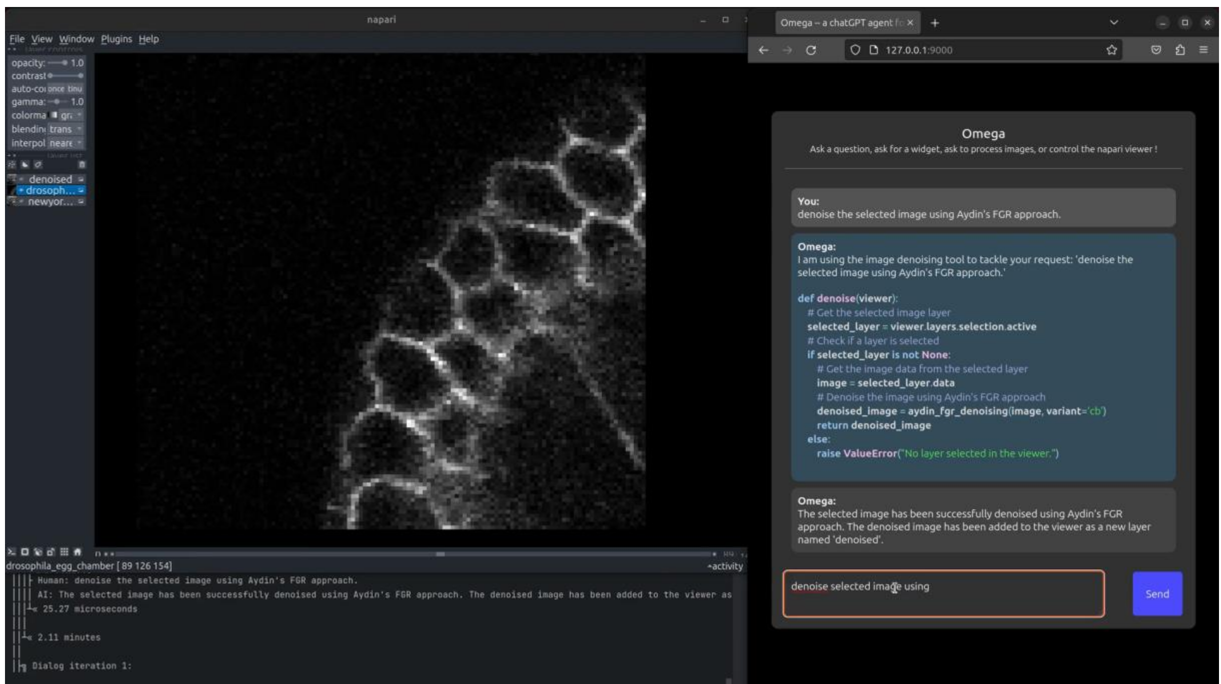
Supplementary Video 11. Omega can control the napari viewer. This video showcases how Omega can manage the napari viewer window. Initially, we requested to change the viewer to 3D rendering mode. Subsequently, we ask it to rotate the orientation of the 3D image by 20 degrees on all axes and zoom in by 50% twice. Then, we request to modify the gamma setting of all layers to a value of 1.5. Finally, we eliminate all layers in the viewer except for the 'nuclei' one. Lastly, we zoom out and switch back to 2D rendering mode. The video has been sped up by a factor of 2.



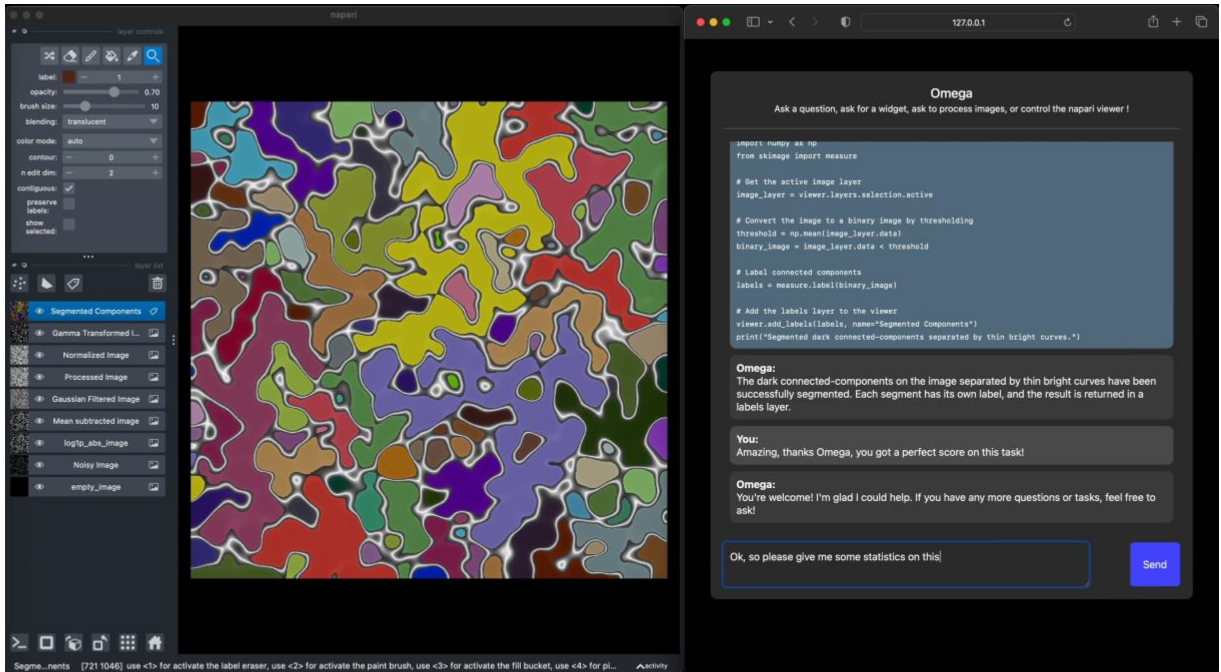
Supplementary Video 12. Omega can determine how to call Python functions. In the video, we requested information from Omega regarding the convolution function in scipy's ndimage package. Omega provided an extensive explanation of the function signature and details about the parameters. However, when we asked to apply the function to a selected image, it generated code for a 2D image instead of a 3D image. After informing Omega that the image was, in fact, 3D, it was able to apply the function successfully with appropriate default parameters. The video has been sped up by a factor of 2.



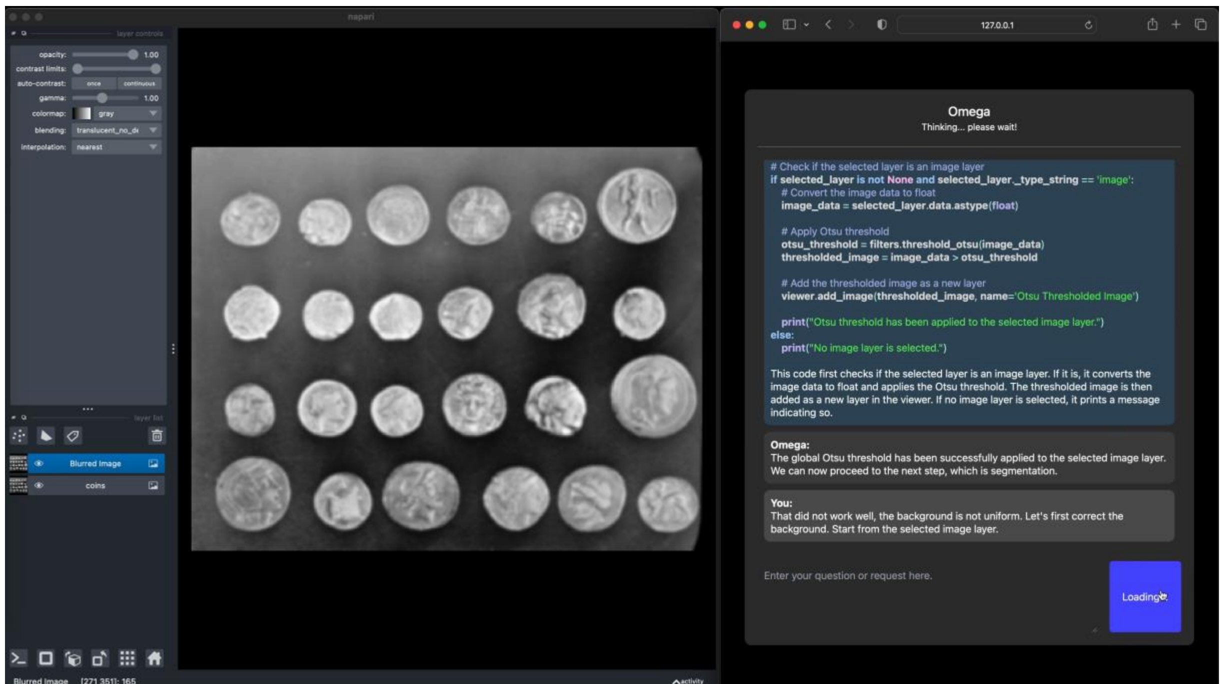
Supplementary Video 13. Omega can use Cellpose to segment cells and nuclei. This brief video showcases how Omega utilizes Cellpose to segment cell nuclei in a 2D image (z-projection). The video has been sped up by a factor of 2.



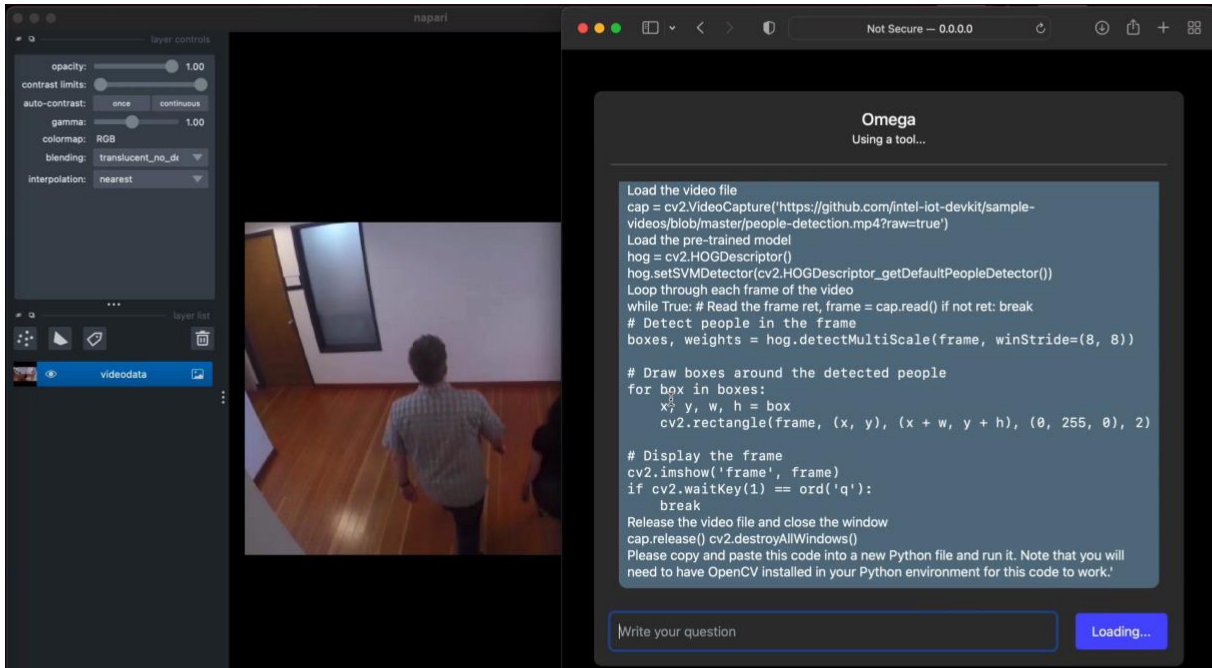
Supplementary Video 14. Omega can use Aydin to denoise images. This video showcases Omega's access to our image-denoising tool Aydin. We first ask Omega to apply Aydin's Noise2Self-FGR (Feature Generation & Regression) approach on a noisy single-channel photograph of the New York skyline (see detailed use case and tutorial here). We see some console output from Aydin running within Omega, and eventually, it displays a denoised version of the image overlayed as a new layer in napari. Next, we ask Omega to apply the same denoising algorithm to a 3D image of Drosophila Egg Chambers (LimSeg Test dataset, Machado et al.), which it does successfully. The video has been sped up by a factor of 2.



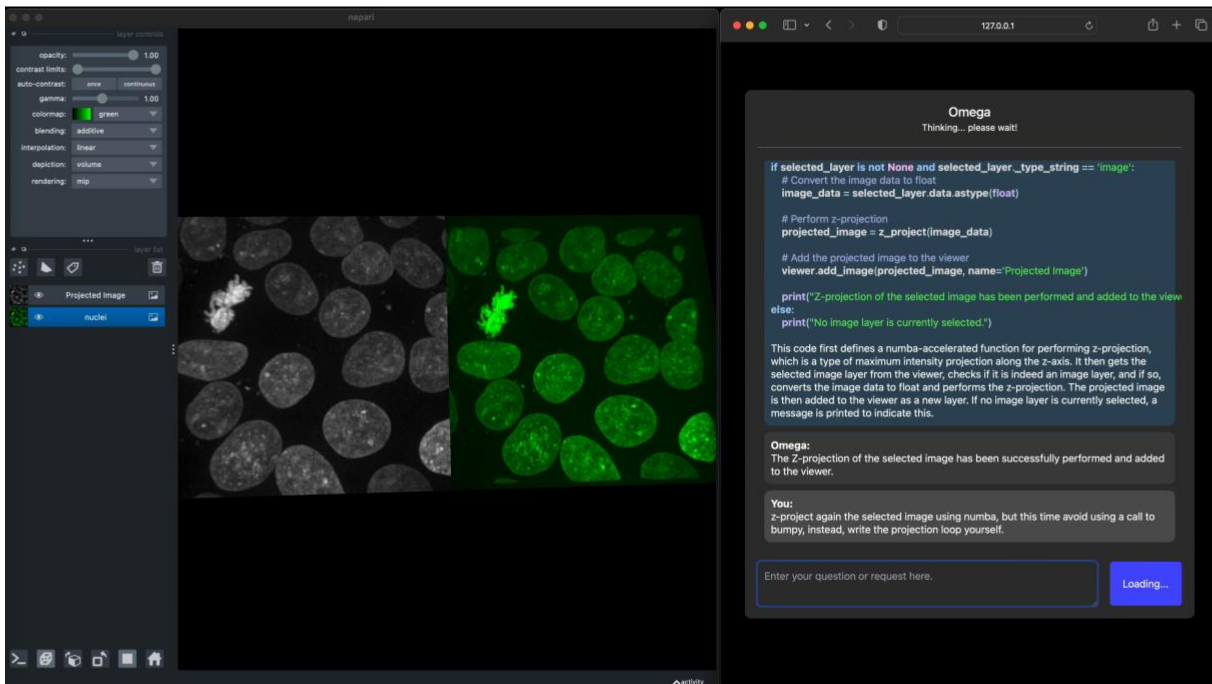
Supplementary Video 15. Omega can follow detailed instructions and has extensive knowledge of NumPy. In this video that runs for about 20 minutes, we demonstrate the process of creating a piece of 'Digital Art' by giving Omega detailed step-by-step instructions. We begin by requesting Omega to generate an empty image and continue by progressively altering it. We add noise and apply functions to the pixel values, threshold, and segment structures. This video highlights Omega's proficiency in NumPy operations and the extensive text conversations that can be utilized for image processing and analysis. The video has been sped up by a factor of 2.



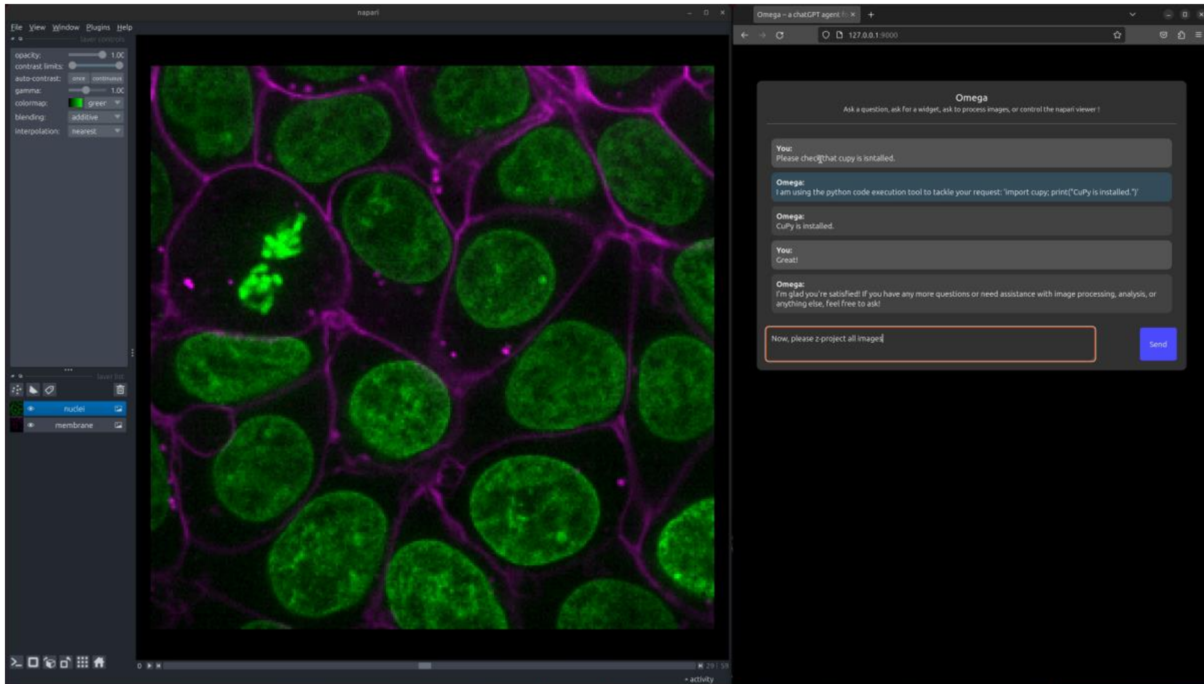
Supplementary Video 16. Omega knows how to use the scikit-image library for processing and analyzing images. This video showcases Omega's mastery of the scikit-image library and image processing. We asked Omega to segment an image with bright coins on a dark background, but we realized that the background was not uniform. To correct the background, we consulted with Omega and learned about different algorithms that could be used. Initially, we attempted to use the rolling-ball algorithm, but we encountered some issues due to Omega's use of a white tophat filter instead of a black tophat filter. We then tried CLAHE (Contrast Limited Adaptive Histogram Equalization), which worked reasonably well, but perhaps we should have used larger tiles. The video has been sped up by a factor of 2.



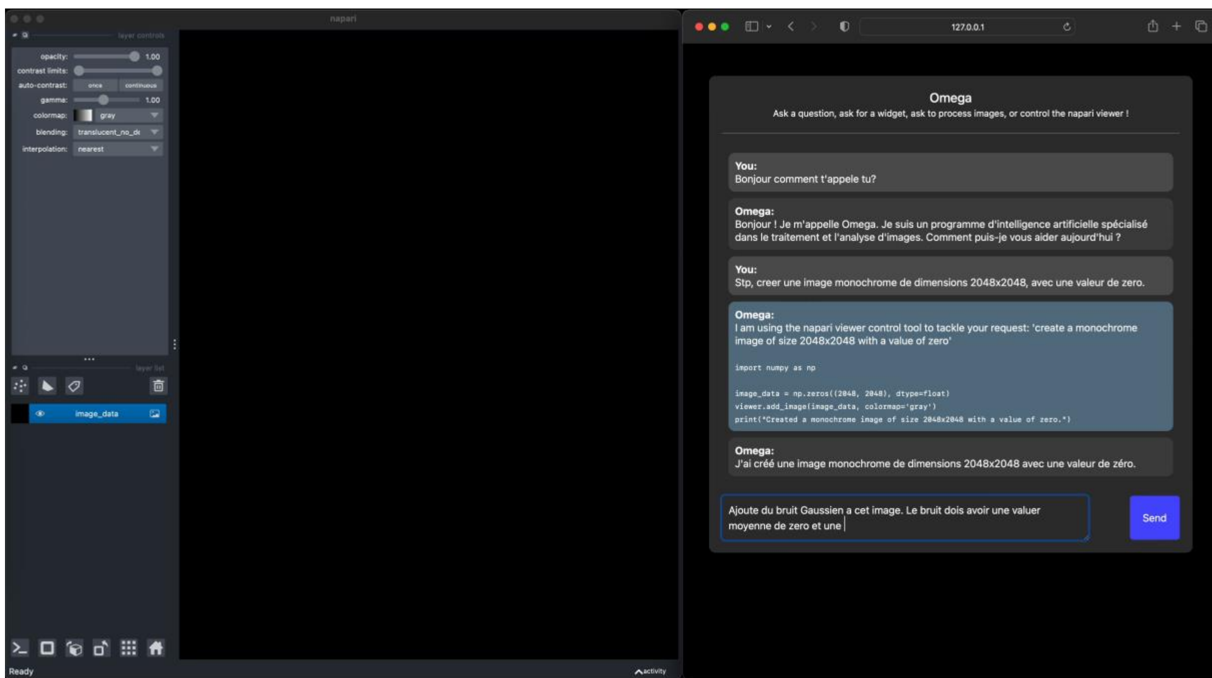
Supplementary Video 17. Omega knows how to use OpenCV. In this video, we requested Omega to download an MP4 movie using the provided URL. The movie displays a hallway and people passing by – a commonly used video for testing person detection algorithms. We then asked Omega to utilize the OpenCV library to detect people in each movie frame and draw a bounding box around each detection. Omega complied with our request and displayed each frame and bounding boxes around each detected person. However, we observed two issues. Firstly, adding each 2D movie frame as individual napari image layers is impractical, resulting in many layers. Secondly, OpenCV's RGB channel ordering is incompatible, causing the napari viewer to display incorrect colors for each frame. The video has been sped up by a factor of 2.



Supplementary Video 18. Omega knows how to use Numba. In the video, we asked Omega to perform a z-projection of a 3D image using the Numba library to speed up the code through just-in-time compilation. Although we did not specify the projection type, Omega used the reasonable choice of max projection and successfully computed it. However, during the process, Omega utilized the NumPy function np.max() in the just-in-time compiled function, defeating our purpose. We then requested Omega to refrain from using NumPy functions and instead write a z-projection loop. Omega completed the task, but this time, it opted for an average projection. We later explicitly asked Omega to perform a max projection. The video has been sped up by a factor of 2.



Supplementary Video 19. Omega knows how to use CuPy. This video presents Omega's proficiency in utilizing the GPU-accelerated CuPy library. Initially, we requested Omega to confirm the installation and functionality of CuPy. Subsequently, we instruct Omega to perform a z-projection of all images displayed in napari. The video has been sped up by a factor of 2.



Supplementary Video 20. Omega can dialog in many different languages. In this video, we speak with Omega in French. This is possible because most LLMs (ChatGPT, Claude, and others) are naturally multilingual. Omega replies to the user in French, but the tools used still operate internally in English, as most of the prompt templates are written in that language. We have tested Omega in several languages, including Spanish, Italian, German, and even Chinese. This feature could hopefully enhance the accessibility of this tool for underserved or non-English-speaking communities. The video has been sped up by a factor of 2.