



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

Loïc A. Royer<sup>1,\*</sup>

<sup>1</sup>Chan Zuckerberg Biohub, San Francisco, USA.

\*Correspondence: [loic.royer@czi.biohub.org](mailto:loic.royer@czi.biohub.org) 

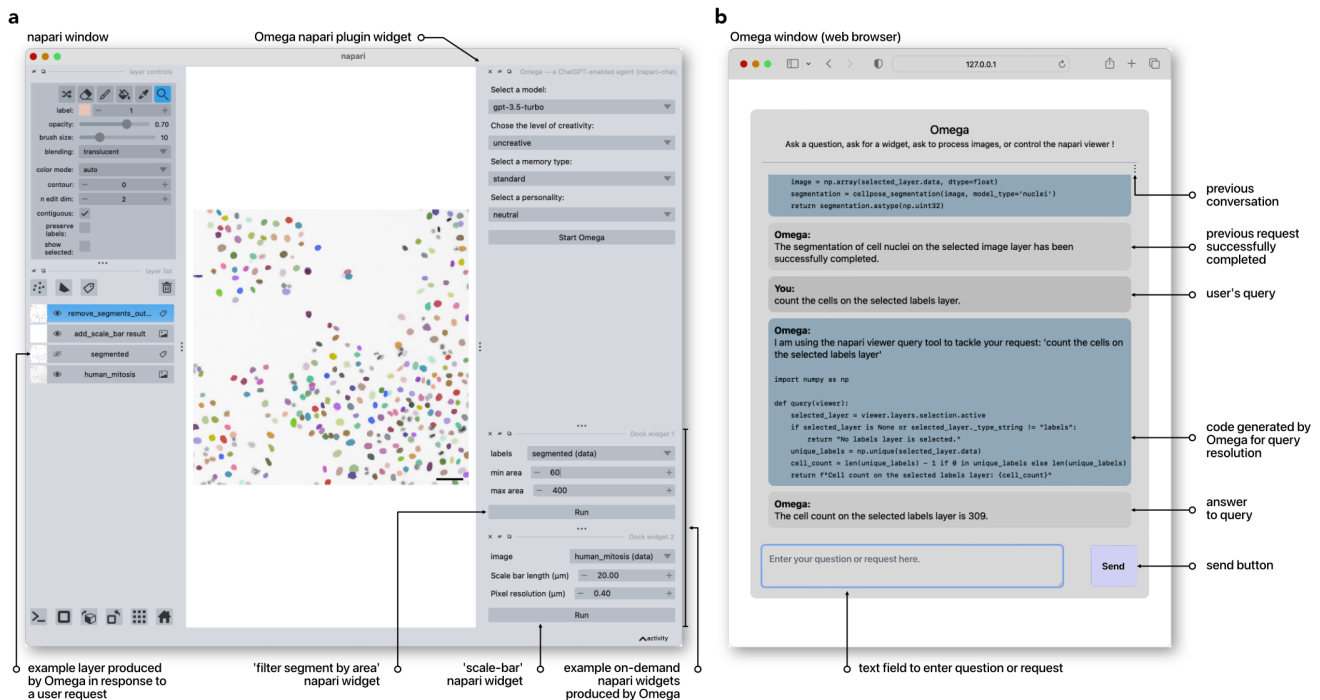


We stand at a pivotal juncture for Artificial Intelligence. Large Language Models (LLMs) such as ChatGPT<sup>1-3</sup> can now engage in insightful conversations, demonstrating an impressive command of human knowledge and logic. These large language models (LLM) are adept conversationalists and possess an increasingly accurate understanding of numerous scientific and engineering disciplines<sup>4</sup>. ChatGPT, a groundbreaking model from OpenAI, can not only code in various programming languages but also explain and rectify code<sup>5</sup>. 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,” 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 color projects a 3D image stack so that the hue is proportional to the depth of voxel of max intensity” (See Fig 1, and Supp. Video [5](#)). On-demand widget generation for user-tailored tasks such as specialized image filtering, transformations, visualizations, and more (see Supp. Table 1 for examples) is one of the most valuable aspects of Omega. To facilitate the reuse, modification, and sharing of generated widgets and code snippets, Omega also includes an AI-augmented code editor that keeps all functional code generated by Omega, allowing users to rerun the code, modify and improve it using AI-augmented tools such as automatic code commenting, cleanup, fixing, modification, and safety checks (Supp. Fig. 2 and Supp. Video [6](#)). Moreover, the editor also lets users on the same local network easily share code (see Supp. Video [7](#)).

Omega is written in Python as a plugin to [napari](#)<sup>6</sup> and leverages the LangChain Python library<sup>7</sup> 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 [8](#)), and other open-source models via [Ollama](#). Omega can converse with the user, like ChatGPT's popular web interface. We utilize the ReAct framework<sup>8</sup> to enable 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 syntax and execution errors encountered (Supp. Video [9](#)).



**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.

Omega’s tools allow it to download files from the web (Supp. Video 10), perform web searches (Supp. Video 11), execute arbitrary Python code (Supp. Video 12), control and query the state as well as contents of the napari viewer (Supp. Video 13), make napari widgets (Supp. Videos 4-5), and query the parameters and documentation of Python functions (Supp. Video 14). Omega also incorporates special-purpose tools that give it access to popular cell and nuclei segmentation algorithms, including: *cellpose*<sup>9</sup> (Supp. Video 15) and *StarDist*<sup>10</sup> (Supp. Video 1 and 4), as well as to our image denoising software *Aydin*<sup>11</sup> (Supp. Video 16). Omega inherits ChatGPT’s Python coding abilities and knowledge (Supp. Video 17). To our surprise, the two best LLMs tested, ChatGPT and Claude, have extensive knowledge of napari’s programming interface (Supp. Video 13) as well as other standard and popular Python libraries such as NumPy<sup>12</sup> (Supp. Video 17), scikit-image<sup>13</sup> (Supp. Video 18), and OpenCV<sup>14</sup> (Supp. Video 19). These models can also utilize hardware

optimization and acceleration libraries such as *numba*<sup>15</sup> for just-in-time compilation (Supp. Video 20) and *CuPy*<sup>16</sup> for GPU acceleration (Supp. Video 21). 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<sup>4,17</sup>. Indeed, we have observed that Omega sometimes uses functions or parameters that do not exist in the installed libraries (see Supp. Video 9) or needs clarification (Supp. Video 12, 13, 17). This is a cause for caution as 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 (note the ambiguity between max and average projection in Supp. Video 20). 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<sup>18</sup>. In the meantime, Omega implements several features that aim to mitigate these problems, including 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. To understand Omega's reliability, we conducted a reproducibility analysis in which we ran five different complex prompts ten times. In 90% of cases, Omega obtained the correct result (see Supp. Table 2). It often suffices to ask Omega to "try again" to resolve most issues.

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](#) & [18](#)). Moreover, Omega can operate in a didactic mode, explaining different approaches and presenting choices to the users before acting. Finally, the multilingual capabilities of ChatGPT and other LLMs mean that Omega is accessible to non-English speakers (Supp. Video [22](#)).

Looking ahead, multimodal foundation models have the growing potential to process and understand diverse data types. Already, state-of-the-art models can incorporate and analyze sound, speech, and videos<sup>19,20</sup>. Omega has access to a Vision tool built upon the vision capabilities of OpenAI's ChatGPT 4, which lets Omega interpret the napari viewer's contents visually (Supp. Video [23](#)). This makes it possible for Omega to decide on the best image processing or analysis approach, for example, when segmenting biological objects (see Supp. Video [24](#)). Eventually, these multimodal models will be able to understand extensive multi-dimensional data directly, circumventing the need to write code to process or analyze complex tabular or graph-based data. This evolution will be a step towards more comprehensive and versatile intelligent agents capable of handling more complex tasks across different data

modalities, expanding the scope of applications and possibilities in image processing and beyond.

The source code and instructions for using Omega are available at [github.com/royerlab/napari-chatgpt](https://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. Thanks to Robert Haase and Ilan Theodoro for bug fixes and Adrian Jacobo for suggestions and discussion. Funding for this work was provided by Chan Zuckerberg Biohub San Francisco (CZB SF). I thank the CZB SF donors Priscilla Chan and Mark Zuckerberg for their generous support.

#### **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 <https://github.com/hwchase17/langchain>.
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. Software 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 Conference 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 Omega’s different components and how they are implemented, from the chat window web server and the code editor to the cascaded LLM approach, different tools, and Python code repair strategies. The most up-to-date version of Omega’s code can be consulted at:

<https://github.com/royerlab/napari-chatgpt>.

The version corresponding to this manuscript is at the branch named [manuscript\\_03\\_2024](#).

**User interface and usage.** 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, start Omega, or open the code editor (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 conversing 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.

**Code editor.** Omega saves all working code in a ‘code editor’ that supports code highlighting and completion. It has AI-augmented tools for automatic code commenting, cleanup, fixing, modification, and safety check (see Supp. Fig. 2). Users can keep all snippets of code and widgets for reuse from one napari session to another, make any manual changes or fixes necessary, and rerun these. Moreover, we implemented a code-sharing feature by which any running instance of Omega’s code editor can find other instances in the local network (via multicast) and share code snippets with others. Notably, the code editor is independent of Omega’s agent and only requires access to any LLM APIs to access its AI-augmented features. If no LLM API keys are provided, the code editor still runs but without any AI features.

**Overall Architecture.** An overview of Omega’s architecture (Supp. Fig. 3) shows the main UI elements at the top: napari itself, the chat window, and the AI-augmented code editor. It also shows the main backend elements: Omega’s ReAct agent, the web server, the editor’s AI tools, and the Python code repair module. The figure also shows Omega’s cascaded architecture, with a main dialog loop driven by a *main LLM* and a series of tools that internally use a delegated *tool LLM*

that leverages online resources, the napari viewer itself, and 3<sup>rd</sup> party libraries such as Cellpose and StarDist.

**Jupyter export.** Omega can save all conversations in runnable Jupyter notebooks. This is done with the standard Jupyter notebook creation and manipulation library ‘nbformat’. In addition, Omega regularly appends screenshots of the viewer to the notebook to document what was visible on the napari viewer during the conversation.

**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 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 in the generated code correspond to functions 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. A high verbosity level can be useful in understanding Omega’s “train of thought” better.

**Chat server.** The chat page is served by [uvicorn](#), an ASGI web server implementation for Python that uses [FastAPI](#) to communicate 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.

**LLMs used in Omega.** Omega leverages [LangChain](#)’s machinery to interface with different LLM APIs. Omega works with OpenAI’s GPT4 models, Anthropic’s Claude models, and all open-source models supported by [Ollama](#). However, OpenAI and Anthropic models (and possibly Google’s new Gemini models, which were not yet tested with Omega) perform best. Unfortunately, open-source models are still not on par with state-of-the-art closed-source models. Omega’s reliance on closed-source LLM models could be seen as an issue. However, this is likely only a temporary situation: the rapid pace of innovation and disruption, especially coming from the open-source flank, bodes well for a future in which Omega would run entirely on open-source models.

**Omega ReAct Agent.** Omega is implemented as [LangChain](#)’s *ConversationalChatAgent*, which uses the ReAct framework<sup>8</sup> 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.**

“Prompt Engineering,” which is the art of designing prompts that nudge LLMs into producing answers relevant to the user's inquiry, was a critical aspect of the building Omega. 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<sup>22</sup>. In the case of Omega's prompt engineering, we made explicit that “You [Omega] are an expert Python programmer with deep expertise in image processing and analysis,” that: “Your [Omega] responses are accurate and informative,” and 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 list of layers in the napari viewer, 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. Therefore, having the system explicitly provide 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, (vi) use special-purpose libraries for cell segmentation and image denoising, and (vii) see the contents of the napari viewer with GPT-4 vision to inform its next steps. Following the ReAct<sup>8</sup> 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.

**Cascaded LLM Architecture.** In Omega, we choose a cascaded LLM approach where, besides the LLM running the 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 blocks of generated code and makes it possible to tailor prompts to each code generation task. For example, when the main LLM running the ReAct agent decides to invoke the ‘widget maker tool’ it calls that tool and forwards the user (or its own) request to the tool. The tool itself is a function that takes the request and inserts it into a long and complex prompt that guides the LLM in generating the Python code for the widget.

**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 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](#)). Recent versions of OpenAI's models have built-in support for ‘function calling’, eliminating this problem.

**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:

**1. 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.

**2. Installing missing packages.** Adding missing import statements is only helpful if the corresponding libraries are installed in the current 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. Omega asks permission to install these missing packages to prevent issues with the Python environment.

**3. 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, Omega carefully constructs 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.

**4. 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. A

specialized prompt explicitly asks the LLM to fix the code using a detailed exception description. This process is repeated recursively until no exceptions occur or the maximum number of repair steps has been reached. Importantly, Omega has a mechanism that forwards task-specific coding instructions used during code generation to be 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.

**1. 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.

**2. 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 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.



**3. 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.

**4. Napari viewer vision tool.** This tool gives Omega a sense of 'vision'. Currently, this is only supported using OpenAI's ChatGPT 4 vision capabilities, but this could be extended to support other and upcoming vision models. Omega can use this tool in two ways: (i) look at the whole viewer or (ii) focus on a single layer. In the first case, the tool takes a snapshot of the whole napari viewer window, sends that picture to the vision model, and obtains a textual description of the image. In the second case, the tool hides all layers except the selected one, resets the viewer to show the layer fully, takes a snapshot of just the viewer's canvas, sends that picture to the vision model, and obtains a textual description of the image. Occasionally, the ChatGPT 4 model refuses to describe the snapshot image. This is documented [here](#) and probably caused by over-zealous security measures preventing the model from describing people. These cases are detected, and the query is repeated until it works – typically less than four times.

**5. Cell segmentation and image denoising tools.** Even the best LLMs, such as ChatGPT 4 or Anthropic's Claude, do not 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. Therefore, to ensure their availability and facilitate their usage, these libraries are integrated 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.

**6. '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. An optional routine that uses watersheds to split the under-segmented segments is available. Finally, the resulting binary image is labeled, and a label layer is returned and added to napari.

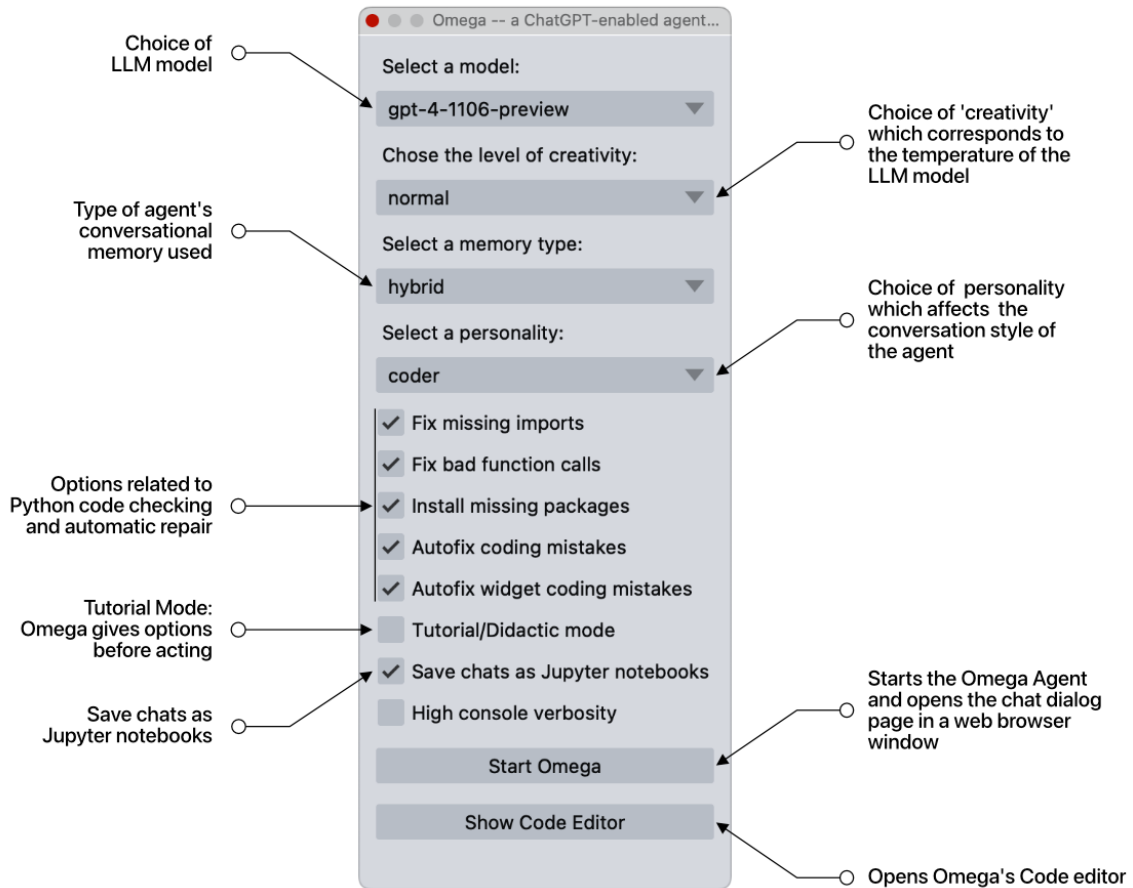
**Didactic posture.** To enhance the interactive learning experience within Omega, we introduced a "Didactic mode". This mode can be activated at Omega's startup (see Supp. Fig. 1). In this mode, Omega adopts a didactic posture, initially presenting the user with a summarization of the task followed by a proposal of multiple strategies to address the request. Users can review these methodologies before initiating any processes, allowing for a conscious selection of the preferred approach. This educational orientation extends further; the system is designed to not only guide but also to educate. It expounds upon the rationales underlying various options, delves into image processing and analytical concepts, and elucidates their practical applications, strengths, and weaknesses. This is achieved by adding instructions to that effect in the agent's system prompt. Such a feature turns Omega into not just a tool for task execution but also a tutor for concept learning.

**API key vault.** The Omega API key vault system is designed to provide a secure method of storing and retrieving API keys required to interface with services such as OpenAI, Anthropic, and others. It ensures that keys are encrypted and accessible only with a proper password, protecting API keys from unauthorized access. The system includes a user-friendly interface that (i) prompts the user, the first time, for an API key

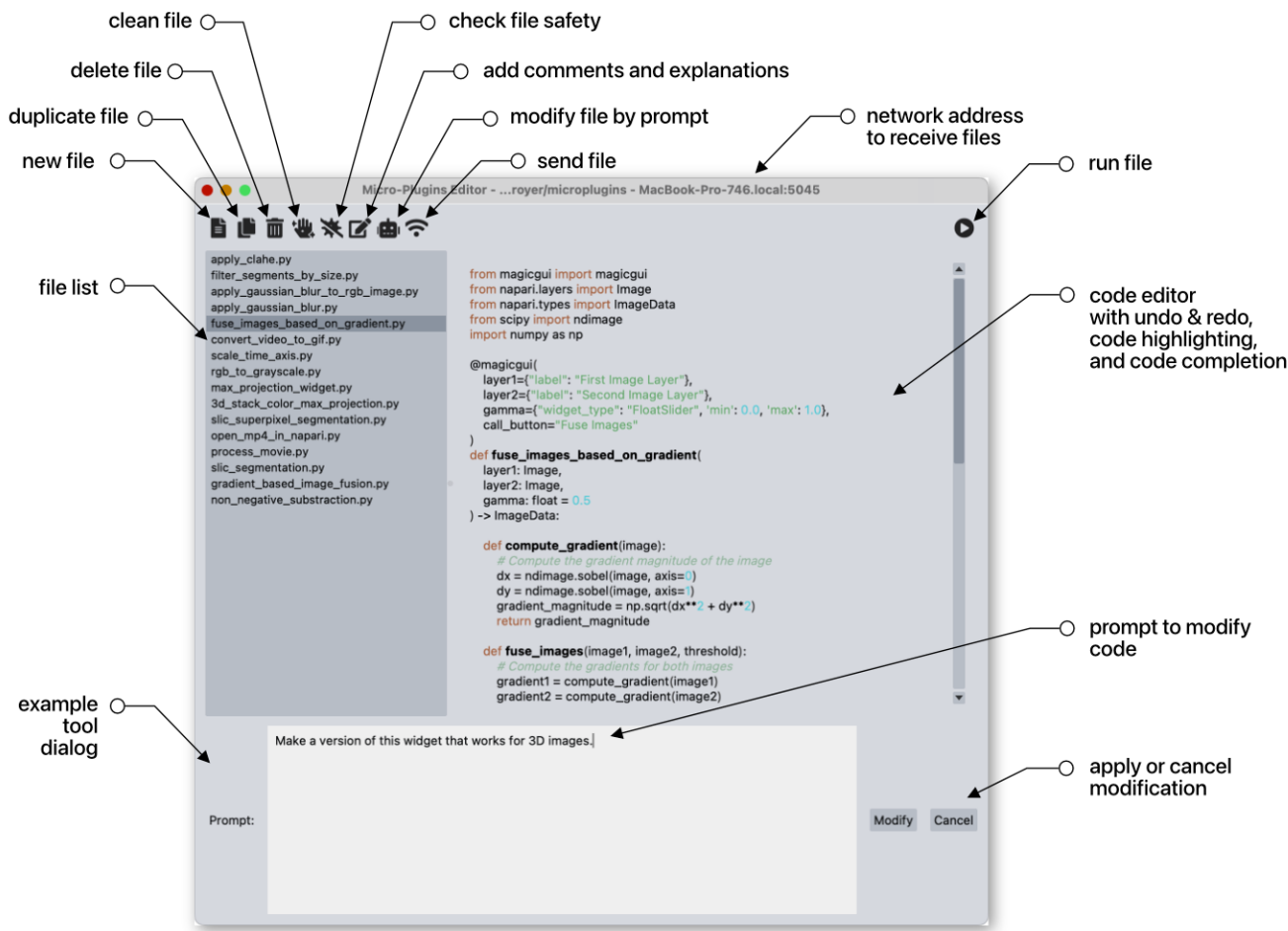
and password to secure it and (ii) subsequently only asks for the password. Some API keys can be set as an environment variable; this is recognized, thus easing the API key management process, but at the risk of having an exposed unencrypted key on the user's system. The Omega API key vault utilizes the Fernet symmetric encryption standard, ensuring that API keys are stored securely on the user's local system and can only be accessed by providing the correct password. The encryption process is bolstered by a password-based key derivation function (PBKDF2HMAC) and a SHA256 hashing algorithm, which provide a secure method of transforming the password into an encryption key.

**Videos.** The videos were recorded with OSX's QuickTime player and were sped up by a factor 2x. Supp. Table 3 lists all prompts for all videos.

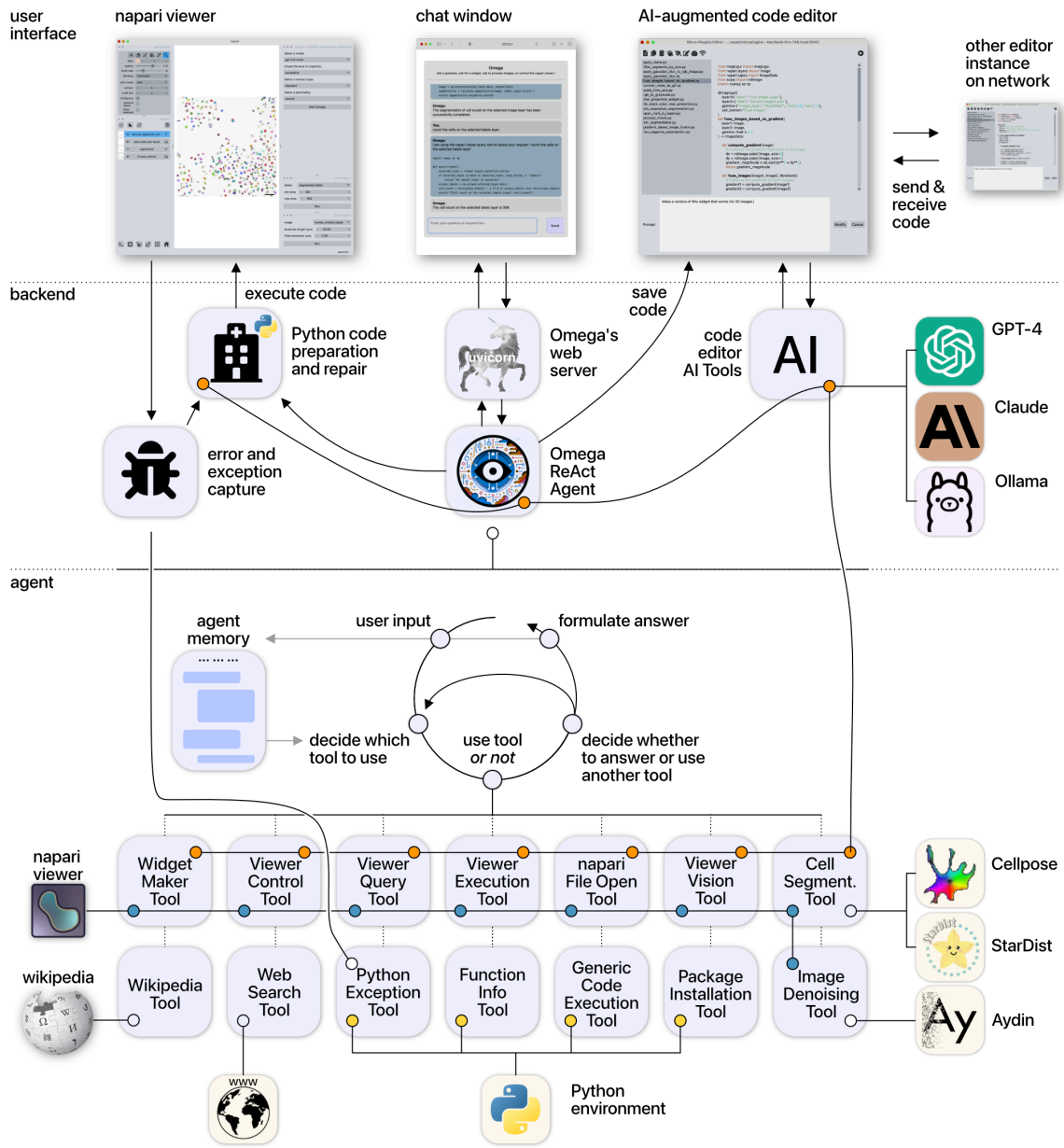
**Supplementary material:**



**Supplementary Figure 1.** Enhanced Code Editor Interface in Omega: Collaborative and Intelligent Code Management for Image Processing. 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, click the "Start Omega" button, and a browser window will open, displaying the agent's chat box. To open the code editor, click on "Show Code Editor". Note: the code does not need Omega to be started or LLM API keys to be minimally functional.

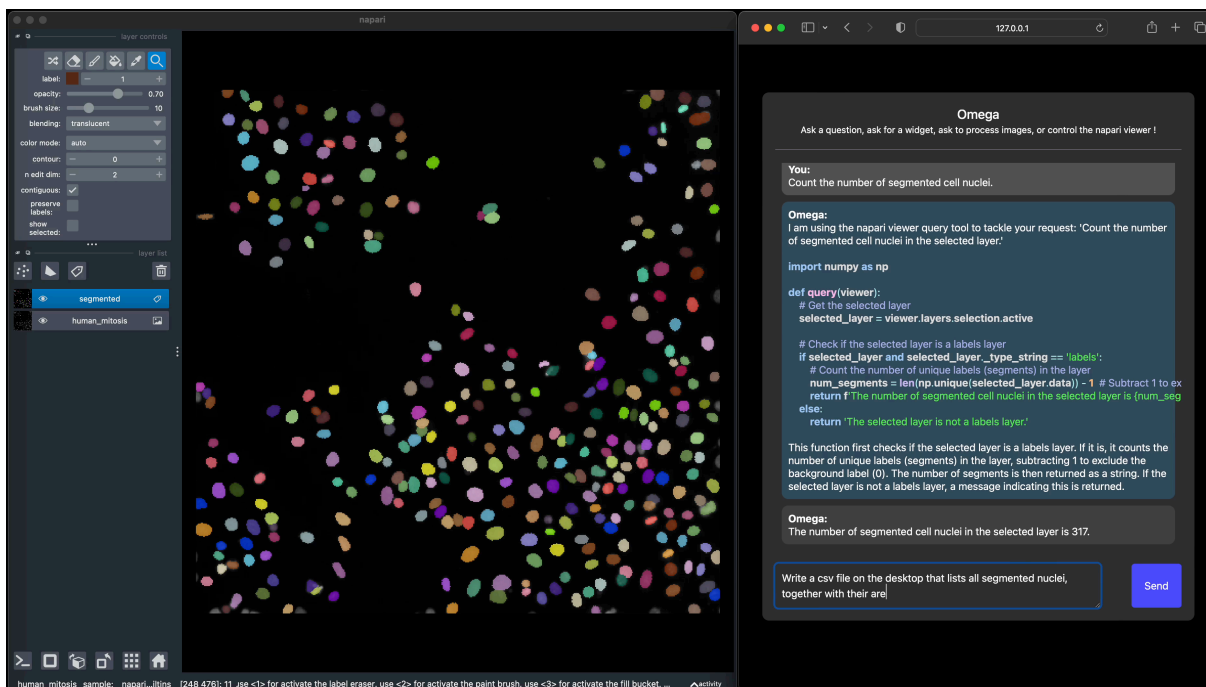


**Supplementary Figure 2.** Omega’s AI-augmented code editor. Omega’s code editor is a key component for managing and refining code for image processing and analysis tasks. The editor is designed to support code highlighting and completion and is enhanced with LLM-augmented tools for automatic code commenting, cleaning, fixing, modification, and safety checks. This allows users to maintain a library of code snippets and widgets, encouraging code reuse and collaboration through a code sharing feature that enables sending code snippets across local network to other code editor instances.

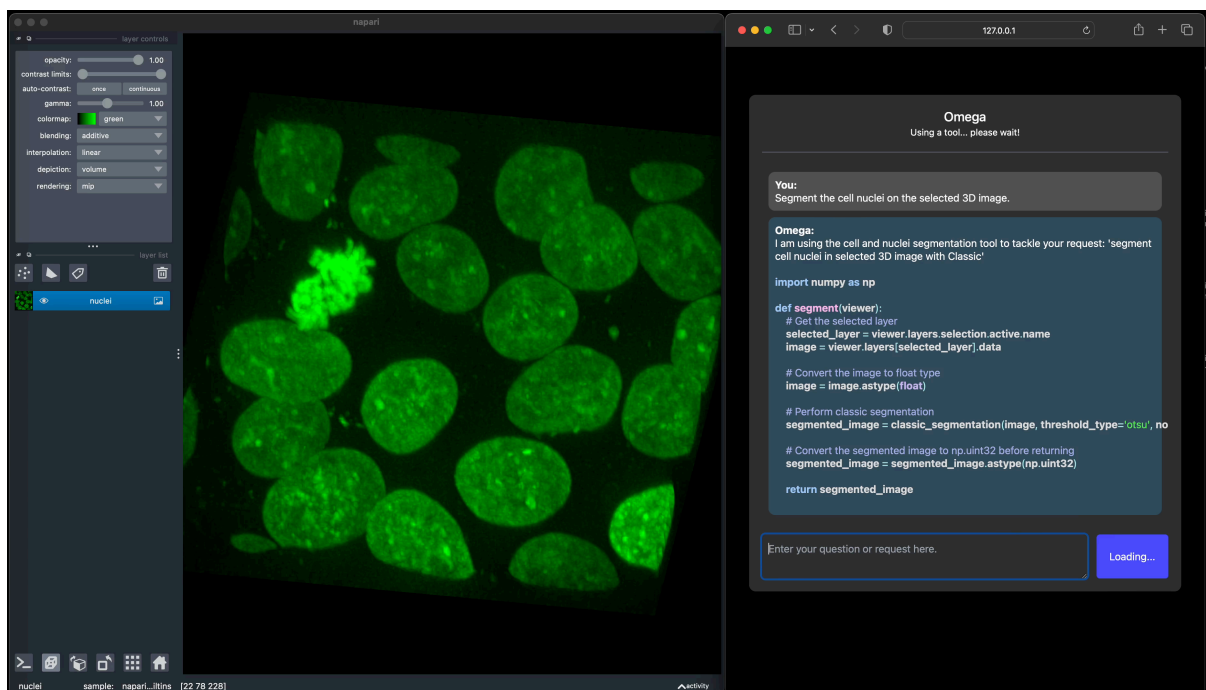


**Supplementary Figure 3. Omega’s System Architecture.** Diagram illustrating the Omega system architecture, encompassing both the user interface and backend components. The user interface is displayed at the top, consisting of the napari viewer, chat window, and AI-enhanced code editor. Key backend elements include the Omega ReAct agent, web server, AI tools within the editor, and the Python code repair module. The architecture employs a cascaded design, with a central dialog loop orchestrated by a primary language model. This model coordinates a suite of specialized tools, each powered by a secondary tool-specific language model that can access online resources, interact with the napari viewer, and integrate functionalities from third-party libraries such as Cellpose and StarDist.

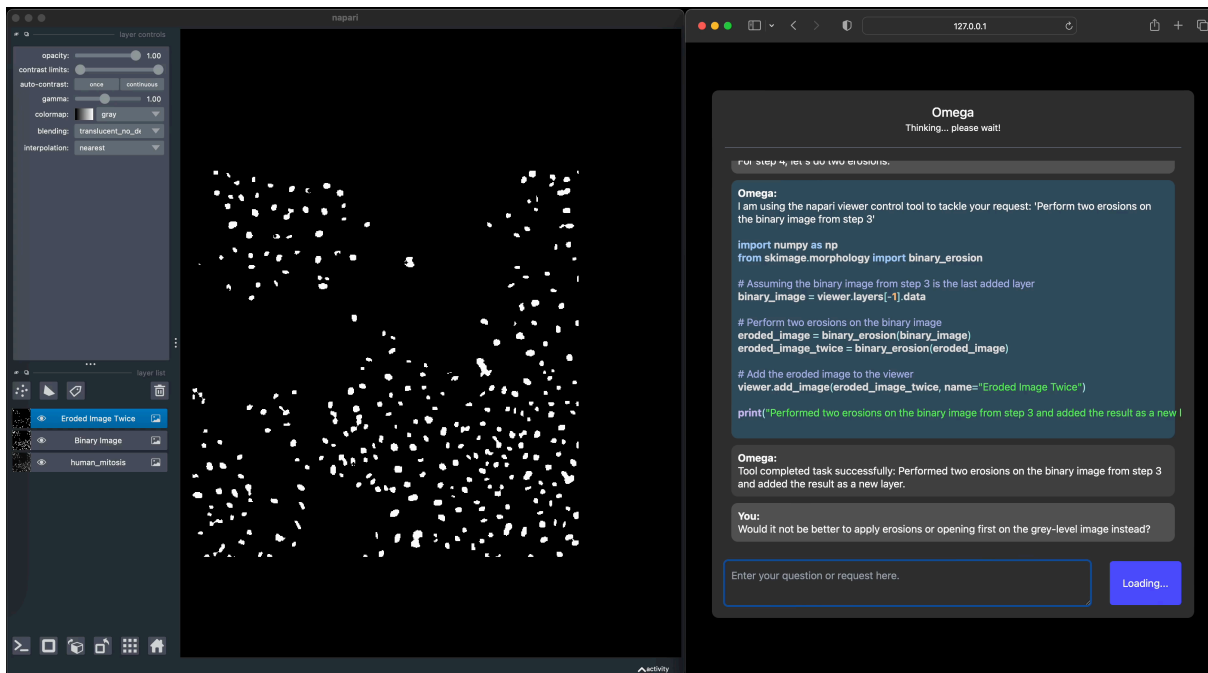




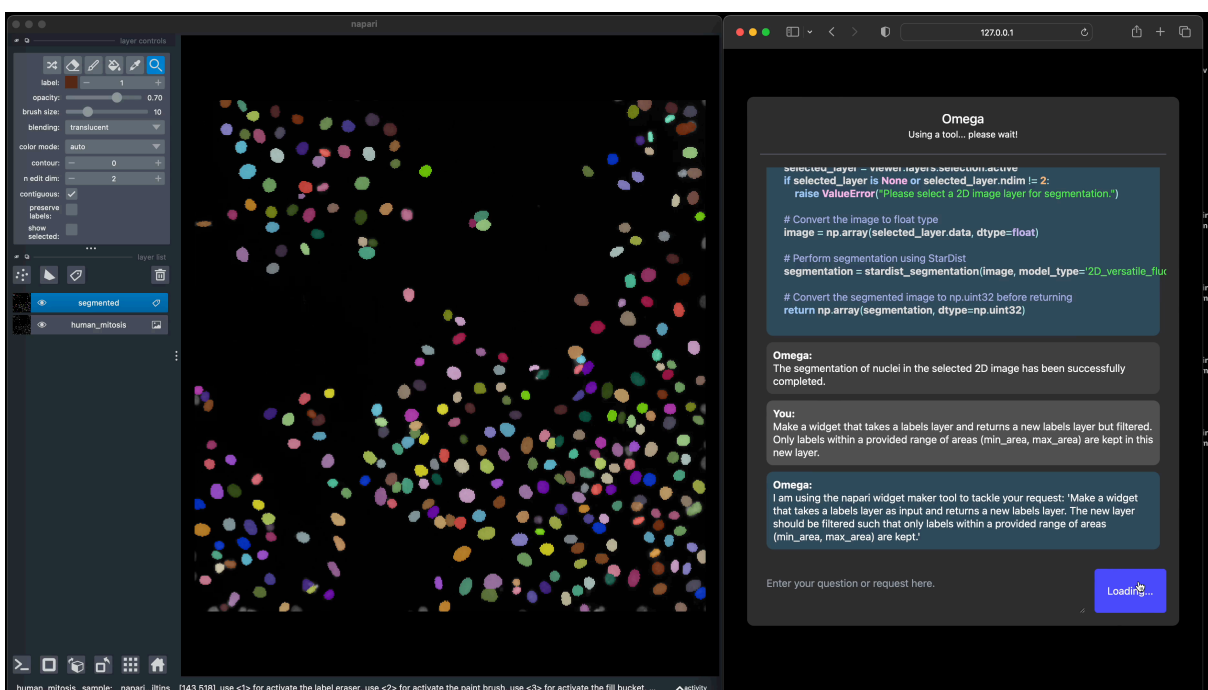
**Supplementary Video 1.** Omega can segment nuclei with StarDist and perform follow-up analysis. The video showcases Omega's ability to segment cell nuclei in a 2D image using [StarDist](#). Omega successfully segments the nuclei and adds a label layer to the napari viewer. With further instructions, 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.



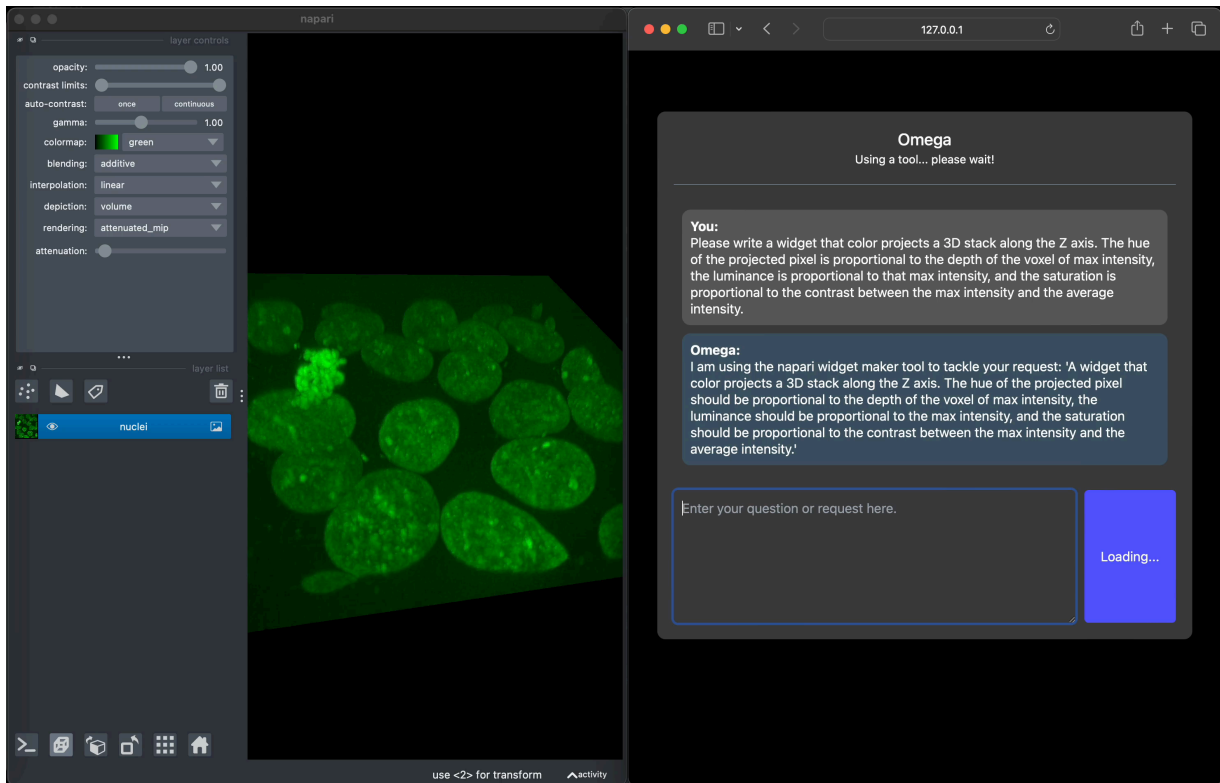
**Supplementary Video 2.** Omega can segment nuclei in a 3D image. This video shows how Omega segments the nuclei in a 3D image displayed in the napari viewer. Omega uses a specialized tool for cell and nuclei segmentation and employs a 'classic' approach that combines single thresholding, specifically [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.



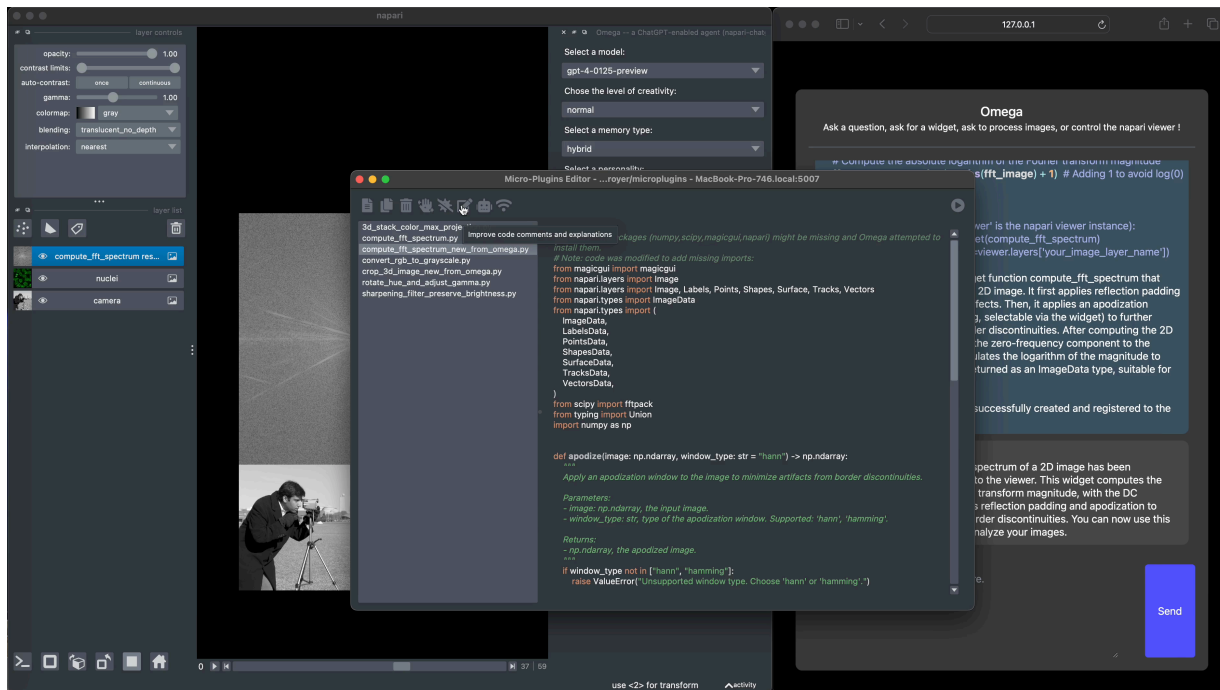
**Supplementary Video 3.** Omega can devise step-by-step strategies and interactively execute them. In this video, we requested Omega's assistance developing a detailed strategy for segmenting nuclei in a 2D image. We clarified that the nuclei appear brighter than the background. Omega provided us with 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.



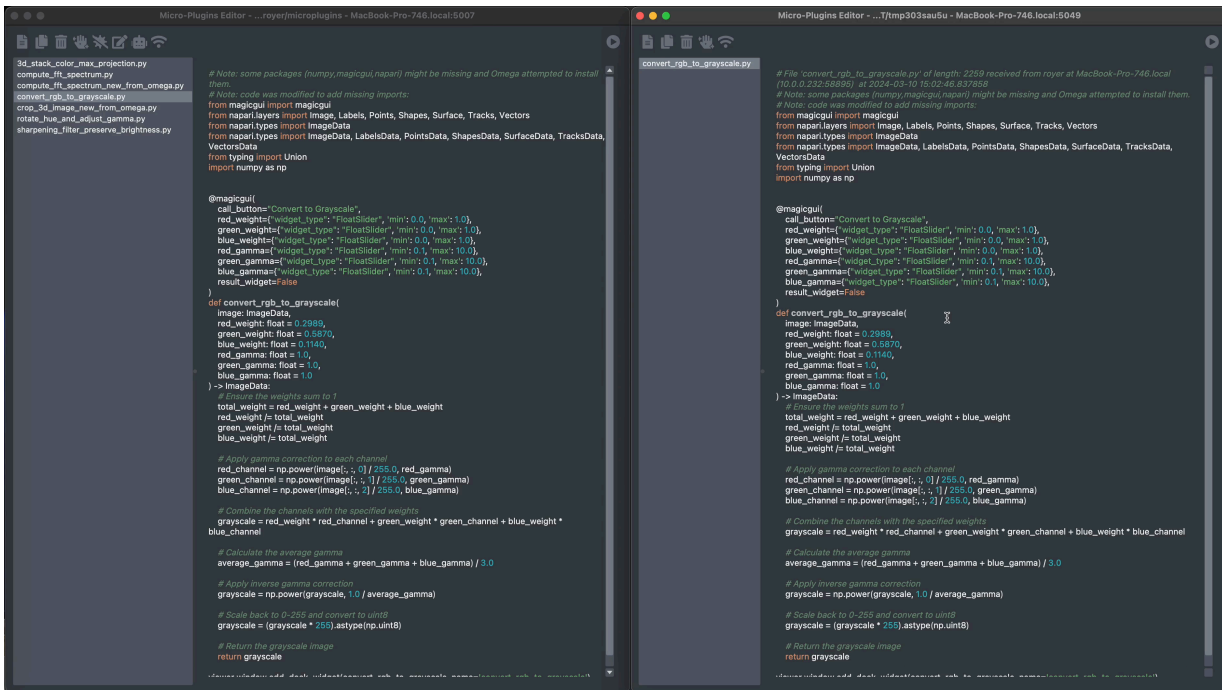
**Supplementary Video 4.** Omega can make widgets on demand, e.g., to filter segments per area. In this video, we first ask that Omega segment the nuclei in the currently selected 2D image. Then, we tell Omega to make a widget that can filter the segments in a label layer according to their area. Segments 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.



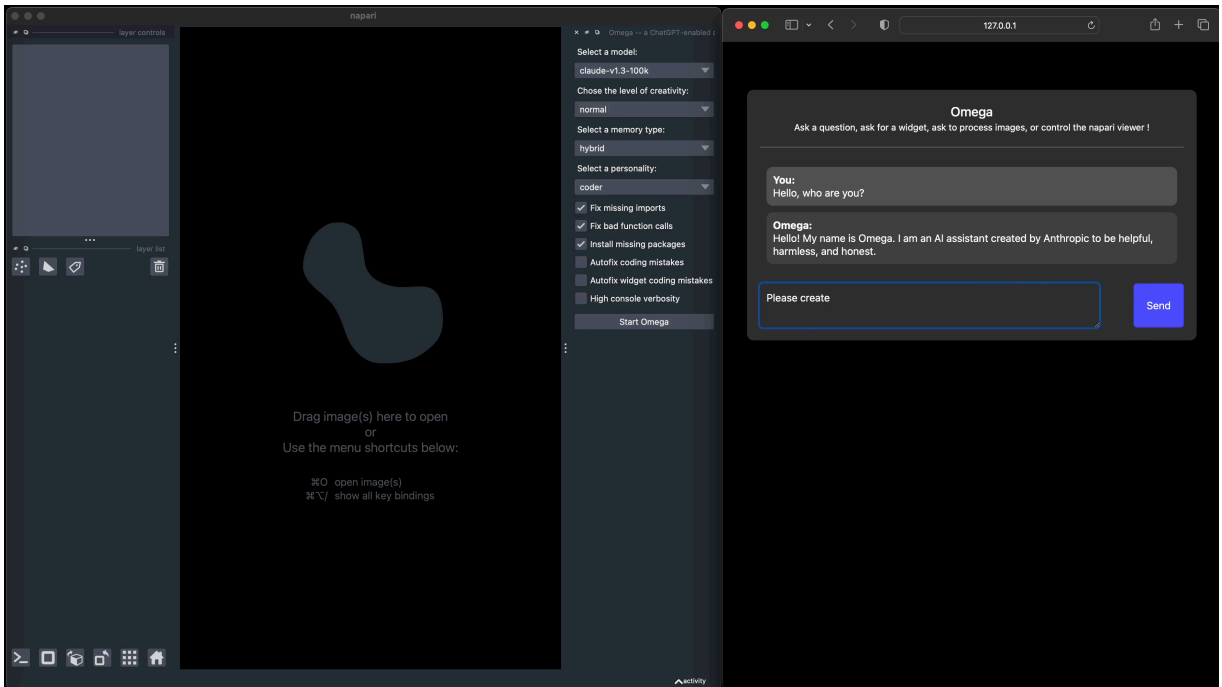
**Supplementary Video 5.** Omega can make complex widgets such as for color max projection. In the video, we requested that Omega create a widget for max color projection of a 3D stack within the napari viewer. The hue variation represents the depth within the stack where the maximum intensity is observed, illuminating the spatial arrangement of the nuclei. Luminance correlates with the maximum intensity of the voxels, highlighting areas of peak fluorescence. Saturation reflects the contrast between the maximum intensity and the average intensity across depth thus suppressing hue variation caused by noise. The video has been sped up by a factor of 2.



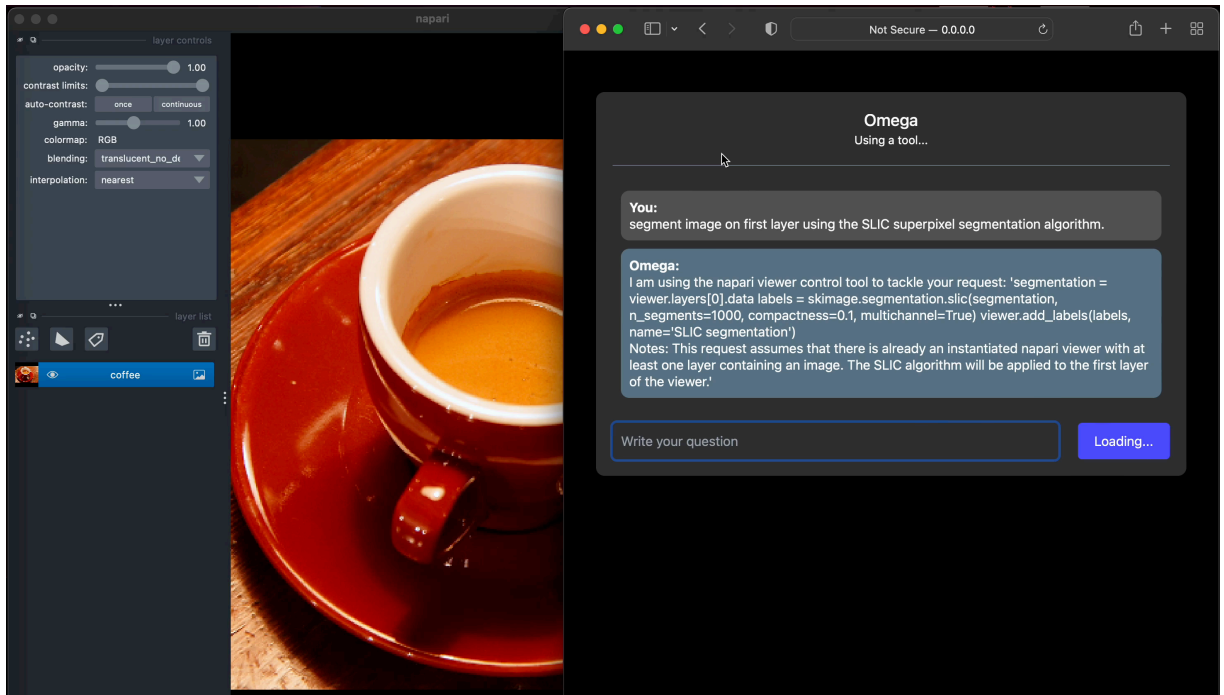
**Supplementary Video 6.** Omega's AI-augmented code editor. In this video, we request a widget that visualizes the Fourier spectrum of a 2D image. Omega makes such a widget, which we test on the camera image. We then can find the source code generated by Omega for that widget in the code editor. We then show the different features of the code editor, such as the code cleanup tool, as well as the AI-powered code safety check tool, code commenting tool, and code modification tool.



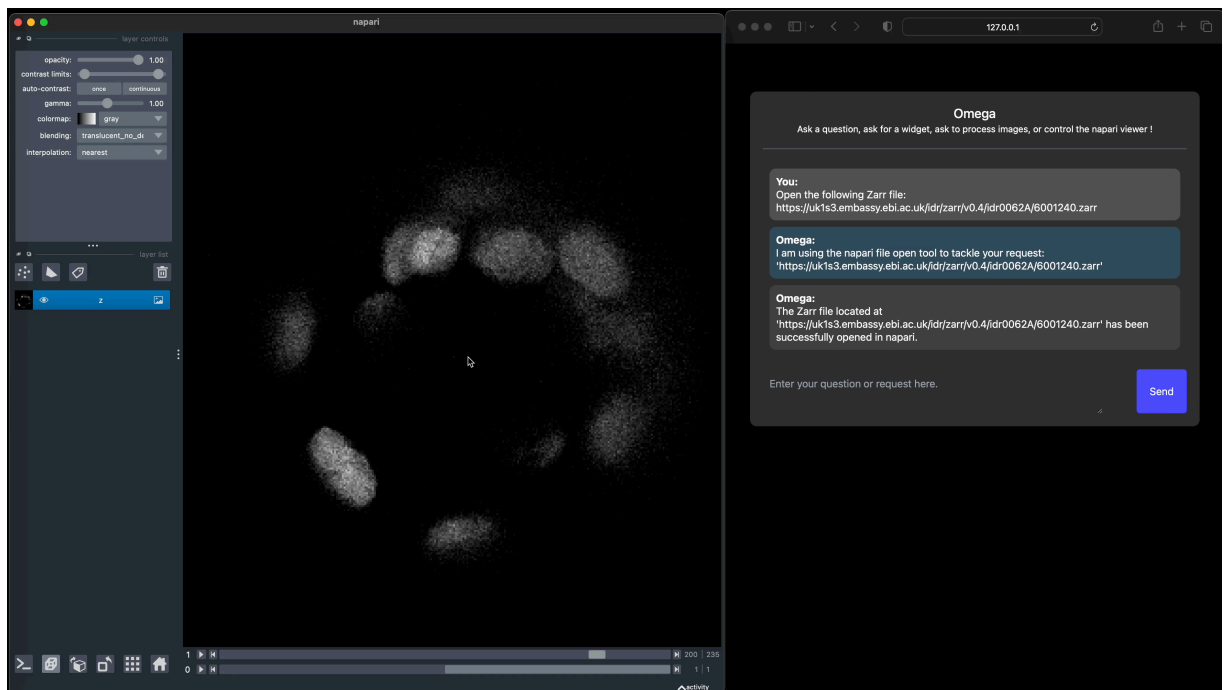
**Supplementary Video 7. Sharing code and widgets across machines.** This video shows how Omega’s code editor can send code across the network to another instance. There is no need to copy and paste the code and send it via email or messaging tool. Simply choose the file, choose the recipient, and it will be sent. All other open instances of the code editor running on machines connected to the same network will be automatically discovered as potential recipients.



**Supplementary Video 8. 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.

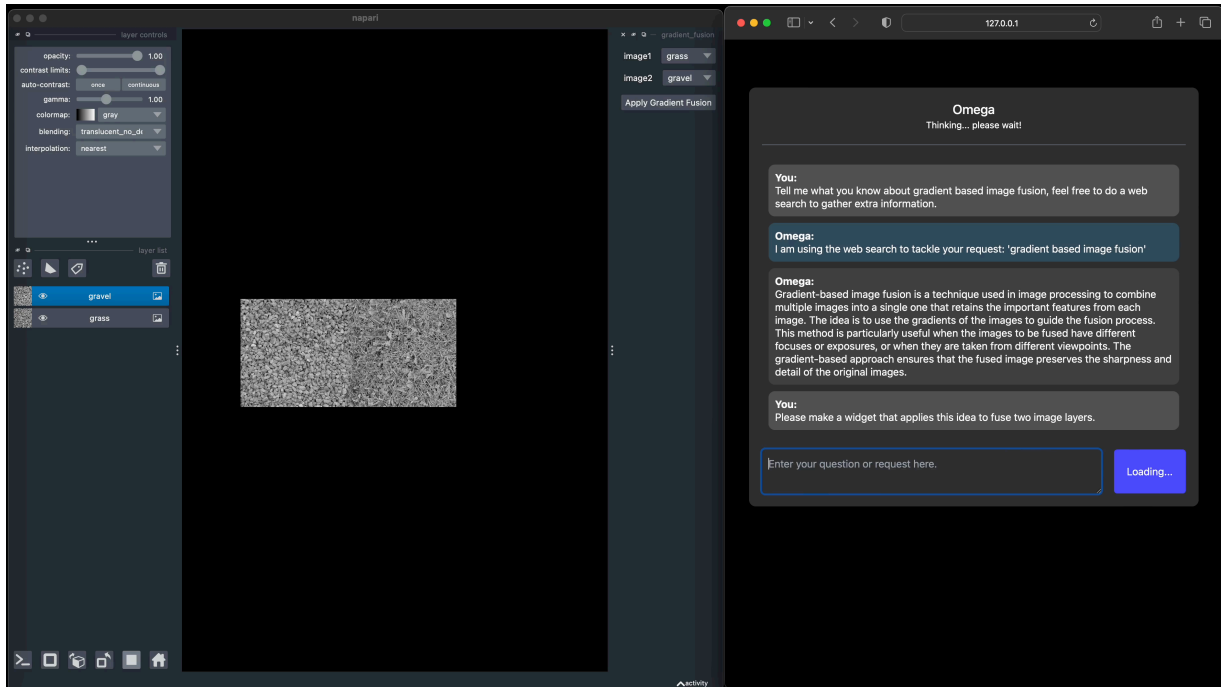


**Supplementary Video 9.** Omega corrects its own coding mistakes. In the video, Omega applied the [SLIC](#) super-pixel segmentation algorithm to a selected image. However, Omega made a mistake using the non-existent 'multichannel' parameter when using the scikit-image SLIC function, resulting in an 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.

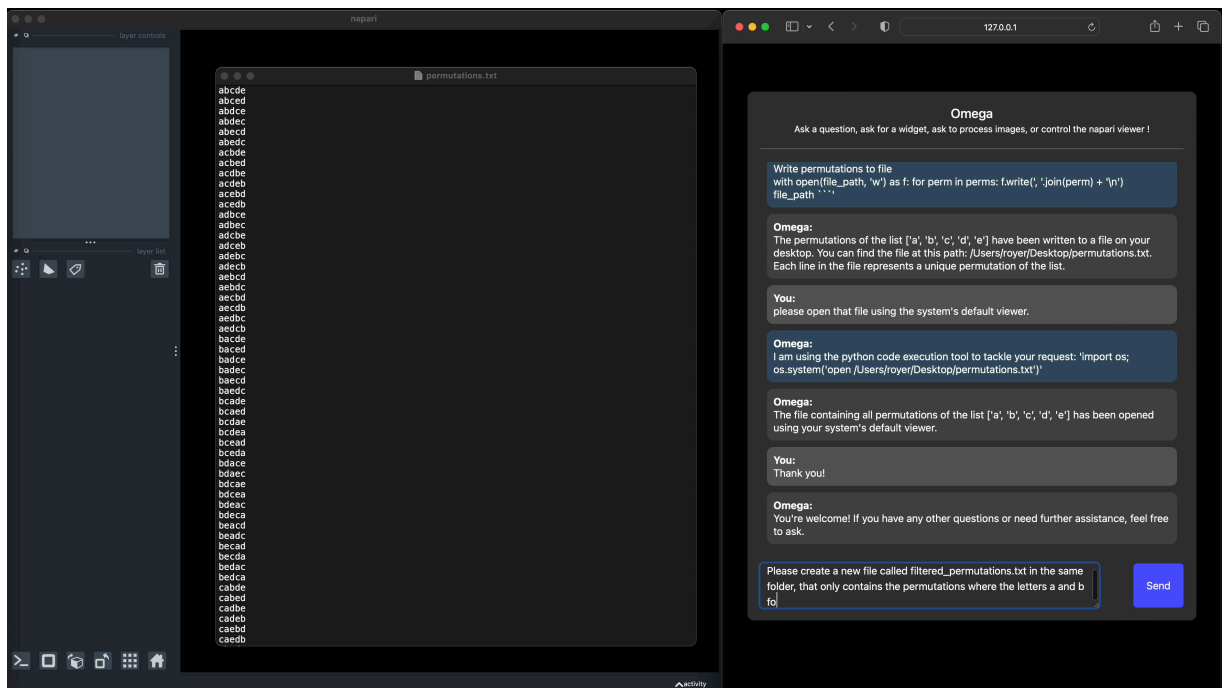


**Supplementary Video 10.** Omega can search and open image file from the web. In this video, we requested Omega to open a dataset from [Blin et al.'s PLOS Biology 2019](#) in napari. The dataset can be accessed online and streamed using the [ZARR image file format and library](#). Omega was able to 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.

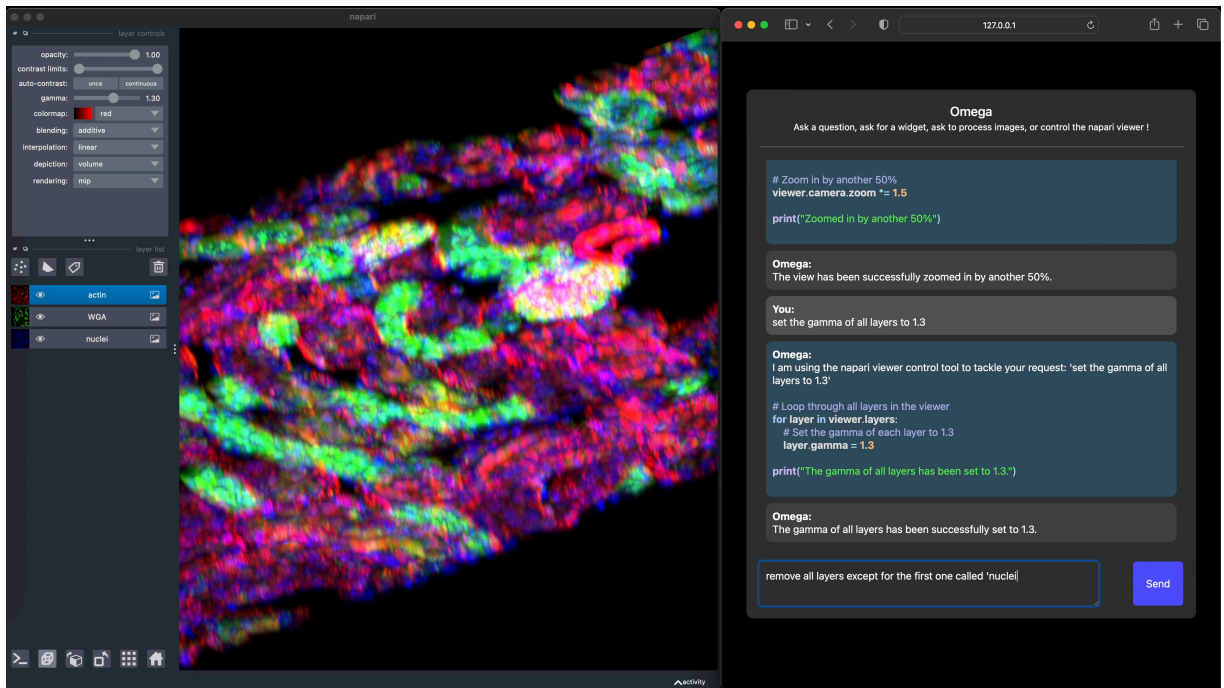




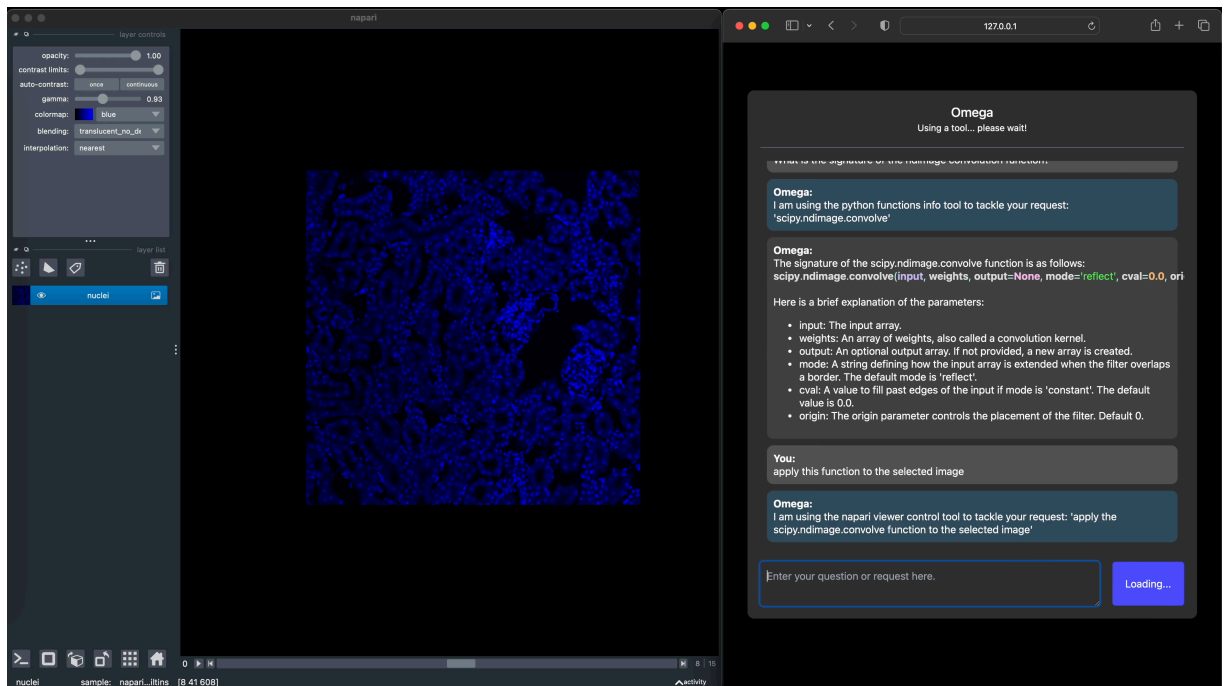
**Supplementary Video 11. Omega can teach concepts in image processing.** In this video, we ask Omega what it knows about 'gradient-based image fusion.' Omega then proceeds to give an interesting explanation of the general idea behind this approach to image fusion. We then ask Omega to apply these ideas 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.



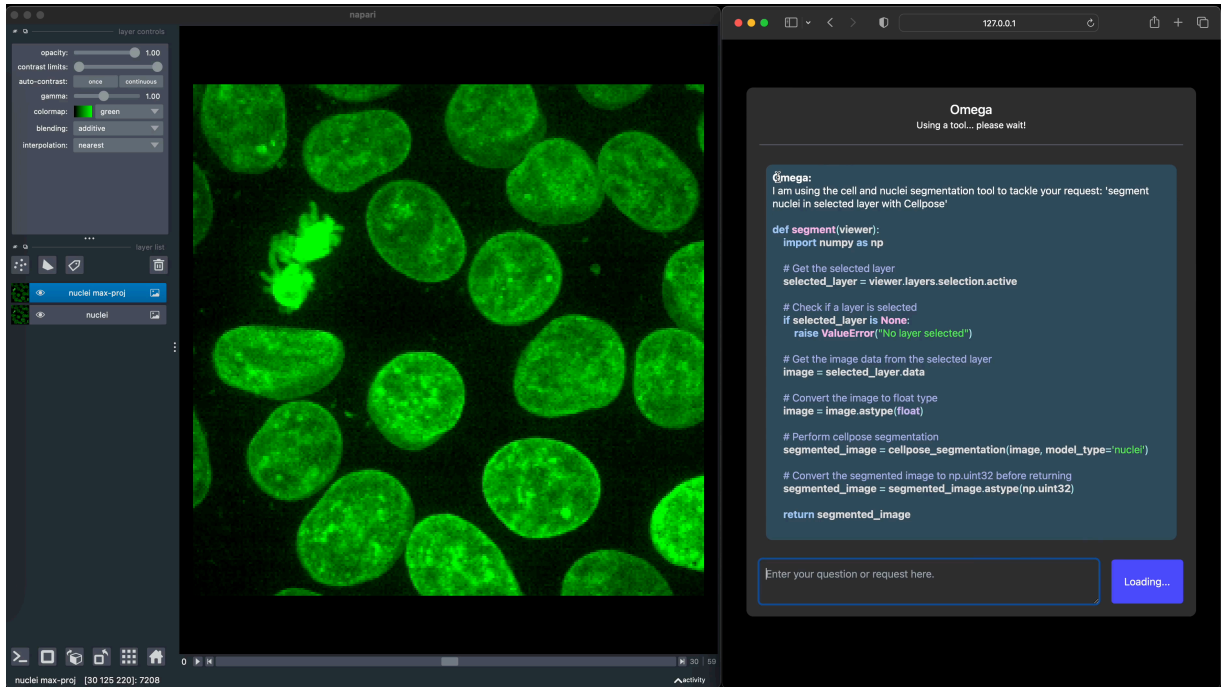
**Supplementary Video 12. Omega can do math and write arbitrary Python code.** In this video, we test Omega's Python coding skills by asking some basic math questions. For example, we asked for the value of  $10^{10}+1$  and the number of permutations possible with ten objects. Then, we asked Omega to 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.



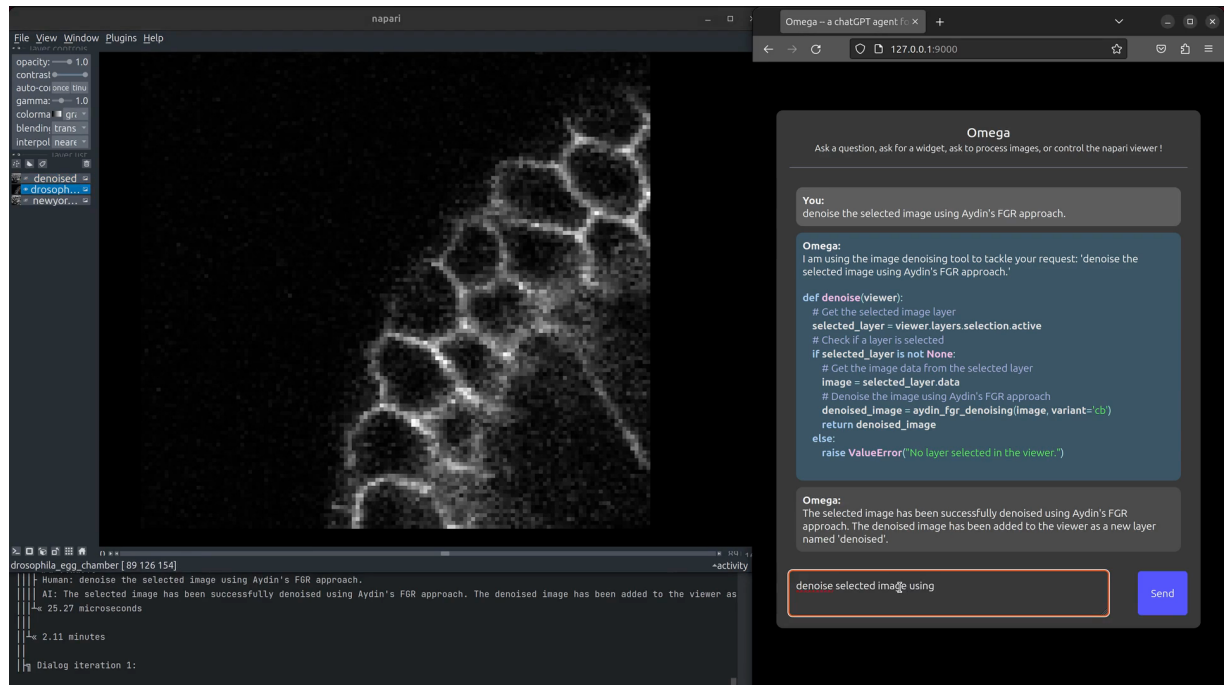
**Supplementary Video 13.** Omega can control the napari viewer. This video showcases how Omega can manage the napari viewer window. Initially, we requested to change the viewer to 3D rendering mode. Subsequently, we ask it to rotate the orientation of the 3D image by 20 degrees on all axes and zoom 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.



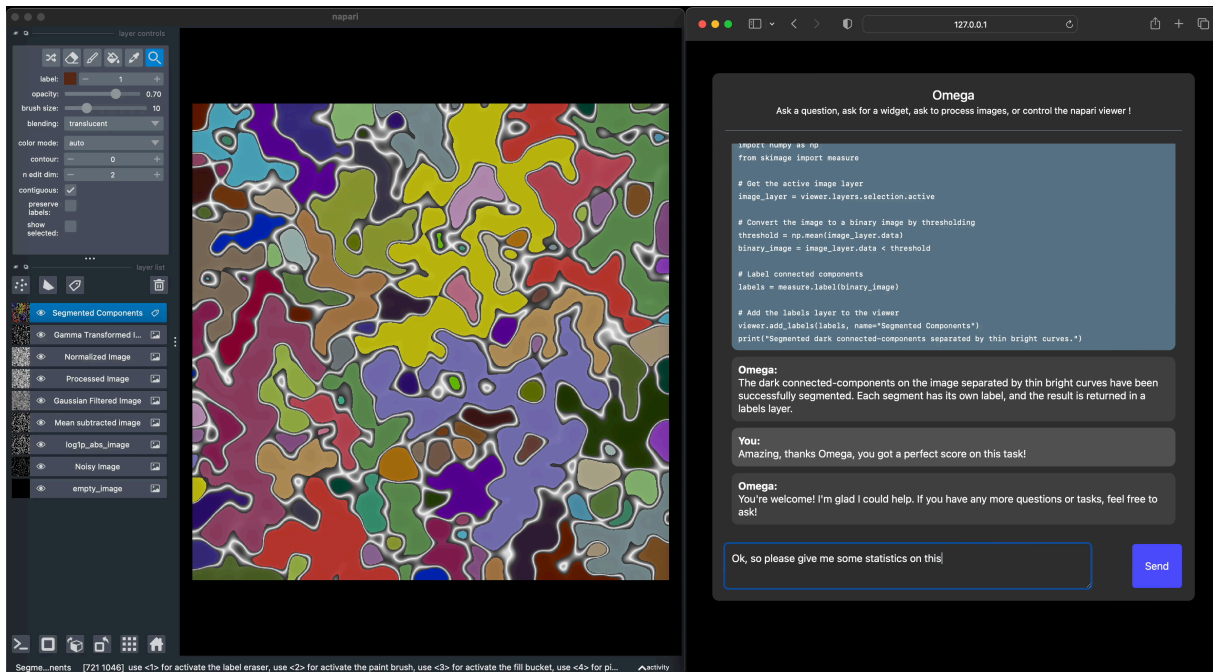
**Supplementary Video 14.** Omega can determine how to call Python functions. In the video, we requested information from Omega regarding the convolution function in [scipy's](https://docs.scipy.org/doc/scipy/reference/ndimage.html) ndimage package. Omega provided an extensive explanation of the function 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.



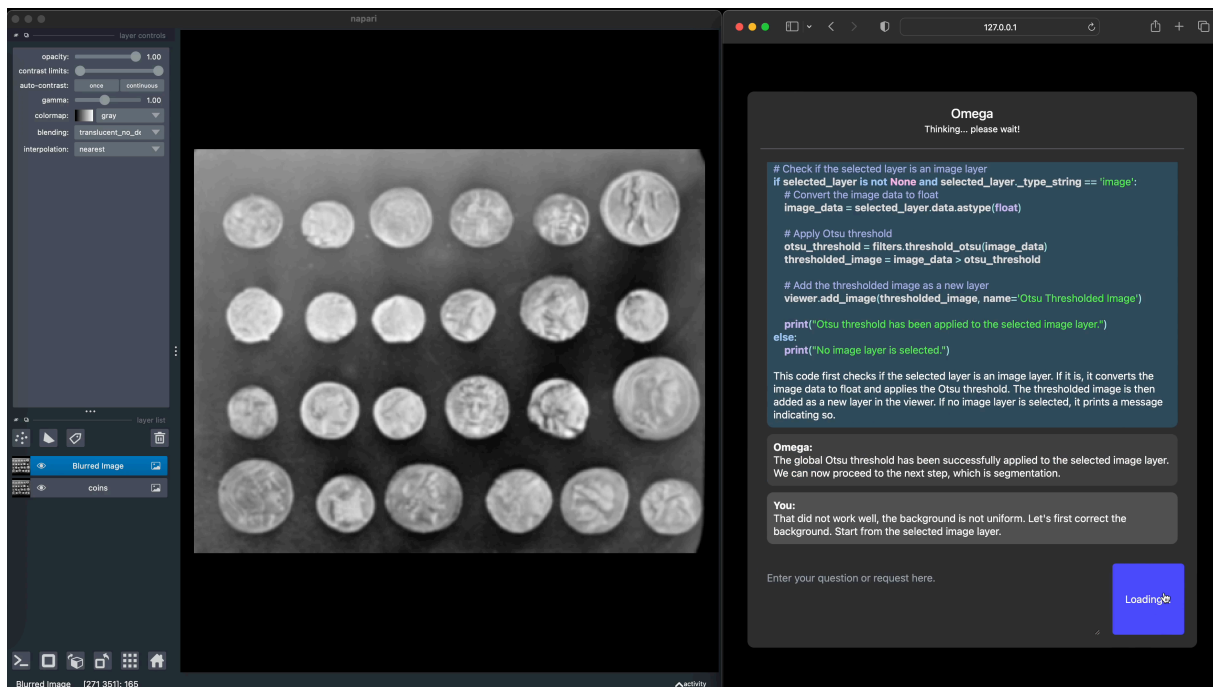
**Supplementary Video 15.** Omega can use Cellpose to segment cells and nuclei. This brief video showcases how Omega utilizes [Cellpose](#) to segment cell nuclei in a 2D image (z-projection). The video has been sped up by a factor of 2.



**Supplementary Video 16.** 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 overlaid 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.

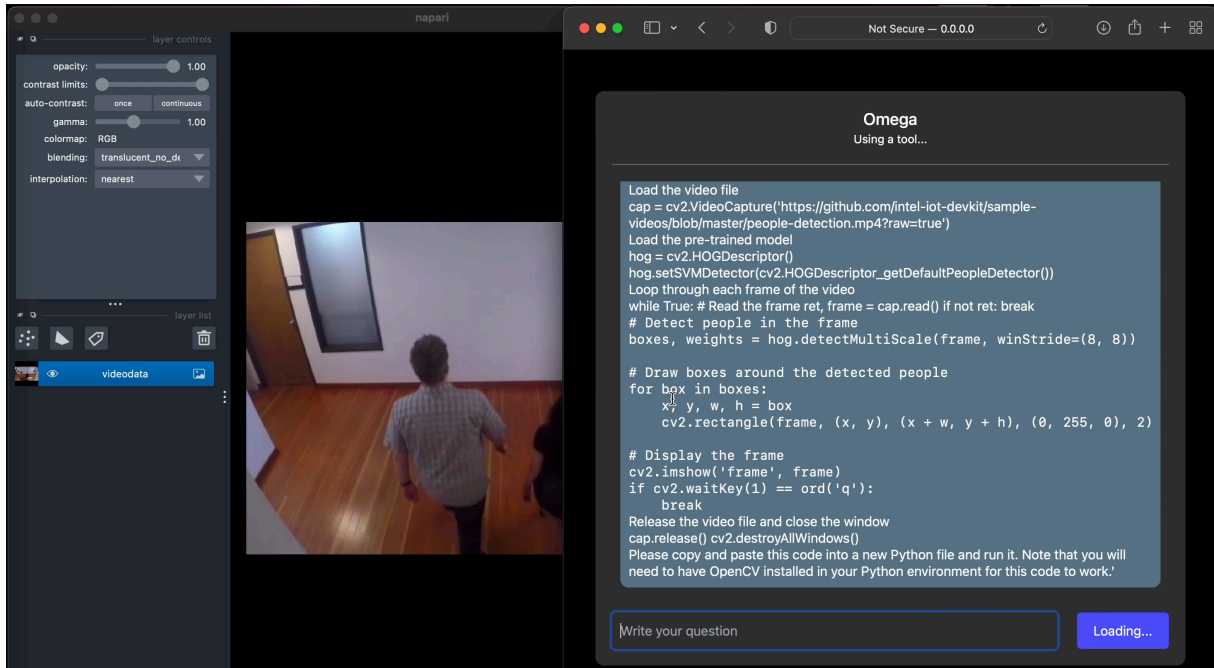


**Supplementary Video 17.** Omega can follow detailed instructions and has extensive knowledge of NumPy. In this video that runs for about 20 minutes, we demonstrate the process of creating a piece of 'Digital Art' by giving Omega detailed step-by-step instructions. We begin by requesting Omega to generate 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.

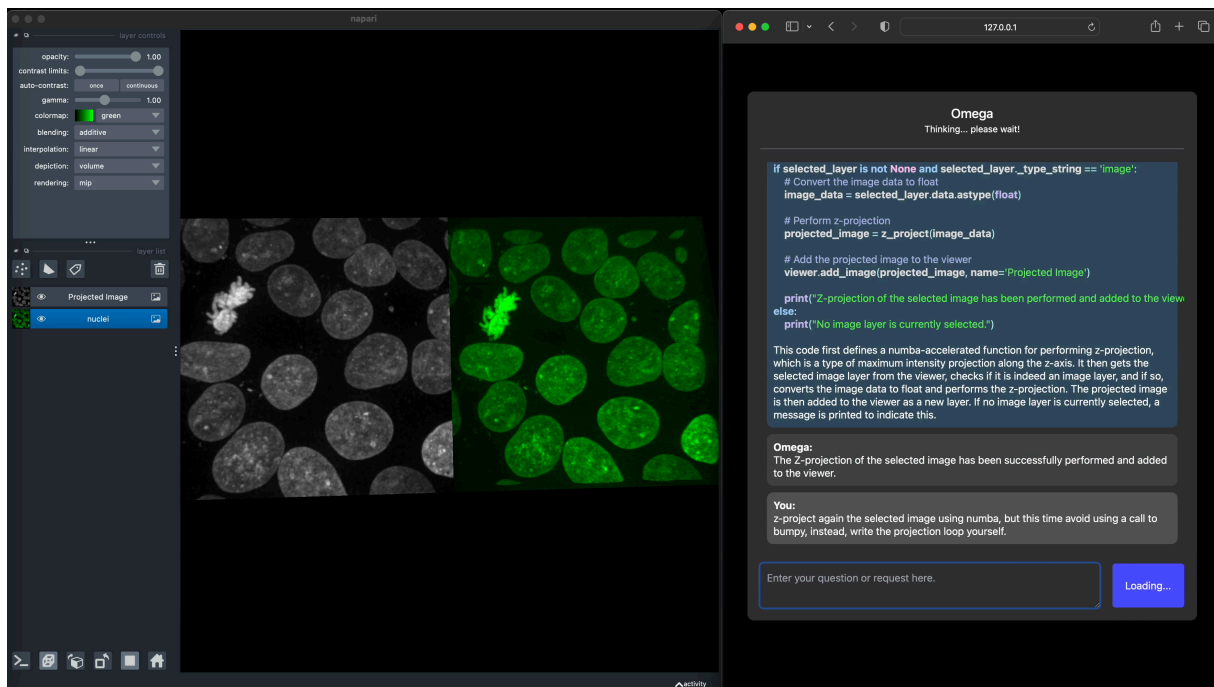


**Supplementary Video 18.** Omega knows how to use the scikit-image library for processing and analyzing images. This video showcases Omega's mastery of the scikit-image library and image processing. We asked Omega to segment an image with bright coins on a dark background, but we realized that the background 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.



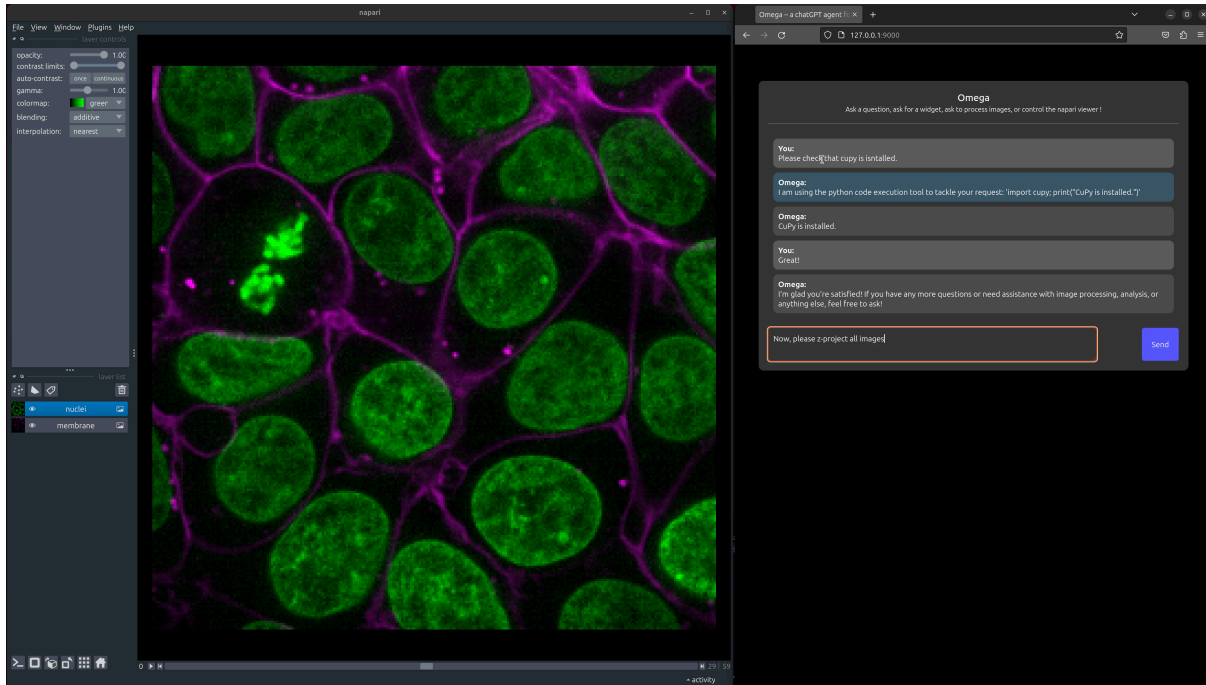


**Supplementary Video 19. Omega knows how to use OpenCV.** In this video, we requested Omega to download an MP4 movie using the provided URL. The movie displays a hallway and people passing by – a commonly used [video](#) for testing person detection algorithms. We then asked Omega to utilize the OpenCV 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.

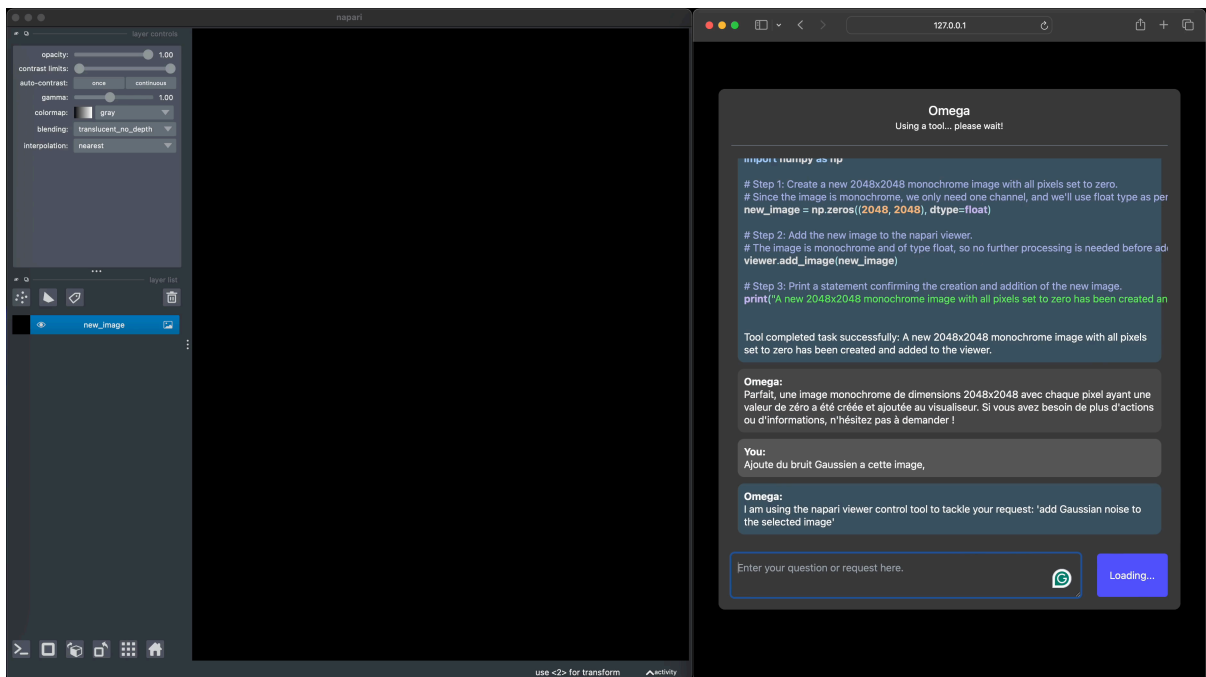


**Supplementary Video 20. Omega knows how to use Numba.** In the video, we asked Omega to perform a z-projection of a 3D image using the [Numba](#) library to speed up the code through just-in-time compilation. Although we did not specify the projection type, Omega used the reasonable choice of max projection 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.

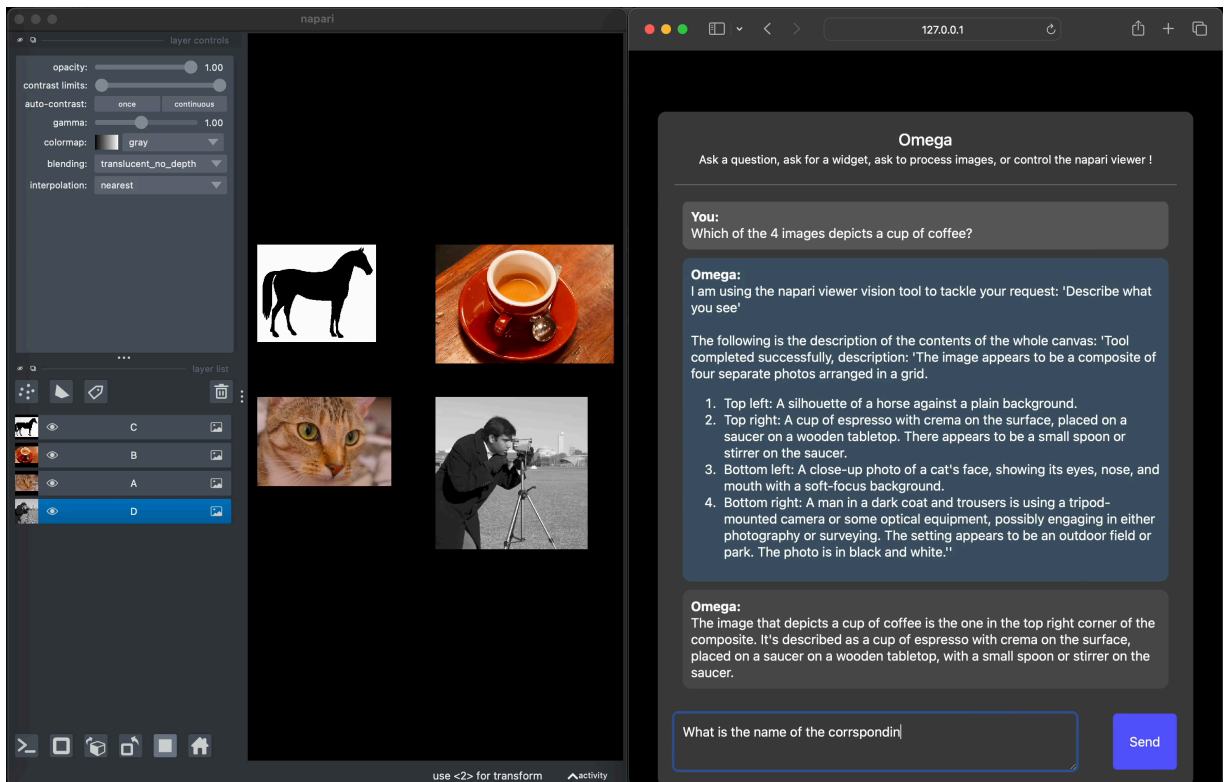




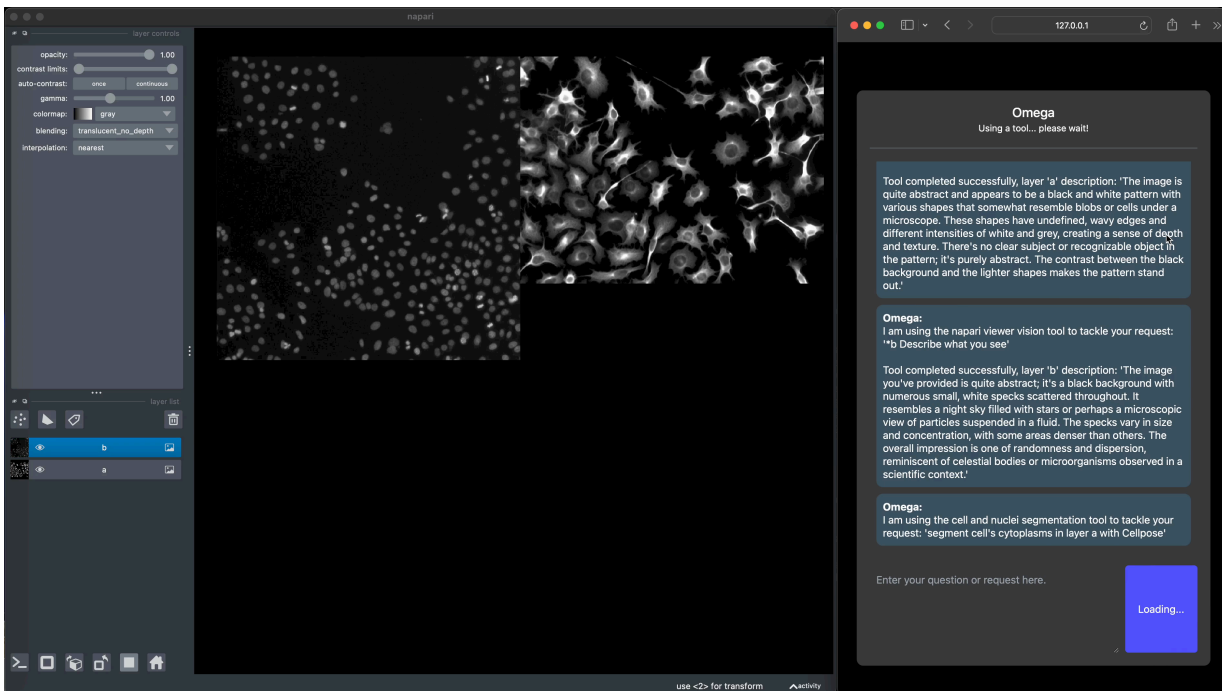
**Supplementary Video 21.** Omega knows how to use CuPy. This video presents Omega's proficiency in utilizing the GPU-accelerated [CuPy](#) library. Initially, we requested Omega to confirm the installation and functionality of CuPy. Subsequently, we instruct Omega to perform a z-projection of all images displayed in napari. The video has been sped up by a factor of 2.



**Supplementary Video 22.** 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 enhances accessibility to non-English speakers. The video has been sped up by a factor of 2.



**Supplementary Video 23. Omega can 'see'.** In this video, we test Omega's ability to see by giving it a visual puzzle to solve. We load 4 images (black & white horse, cup of coffee, cat face, camera test image), change their layer names to 'A', 'B', 'C', 'D', and then ask Omega to find which of the four images depicts a cup of coffee. Omega uses its 'napari viewer vision tool' to see the contents of the napari viewer, correctly describe each image, and identify the one depicting a cup of coffee as the one on the top right corner. We further ask for the name of the layer, and Omega uses again its vision tool to verify that layer 'B' holds that image. The final answer is the correct one: 'B'. The video has been sped up by a factor of 2.



**Supplementary Video 24. Omega decides how to best segment an image using vision.** In this video we present Omega with two 2D microscopy images: one with cytoplasm labeled (b) and another with nuclei labeled (a). We then ask Omega to decide how to best segment the biological structures present in each image by using the vision tool. Omega looks at the image contents, describe them, and correctly decides that Cellpose is best for layer b and StarDist is best for layer a.

Type	Prompts:	Dataset or Image to use:	Notes:
Filtering	Please make me a Gaussian blur widget with sigma parameter	E.g., the 'Human Mitosis' built-in napari sample dataset.	
	Please make me a widget for spatially variable Gaussian blur with a sigma parameter varying over z between two values: sigma_bottom and m sigma_top. The top corresponds to high-z values, and the bottom corresponds to low-z values. Sigma values should range between 0 (no blur) and 10.	3D image	
	Please make a widget that applies Butterworth filtering to a 2D grayscale image with a defined cut-off frequency parameter normalized between 0 (min. freq.) and 1 (max. freq.) and order (between 1 and 10).	2D grayscale image	
	Please make a Sobel edge filter widget that works on 3D grayscale images. The user can choose an isotropic kernel size of 3, 5, or 7. Make sure to optimize the code.	3D image	
	Create a sharpening filter widget for 2D monochrome images and expose relevant parameters. Ensure that the image's total brightness is preserved. Expose a parameter to control the strength of the sharpening.	2D image	
	Make a band-pass filter widget that keeps frequencies in a 2D or 3D grayscale image between two frequencies: f_min, f_max. The frequencies are provided in the normalized range [0f, 1f].		
Transforms	Please make a widget that can crop a 3D image by specifying the number of pixels to remove on all sides (x-, x+, y-, y+, z-, z+).	3D image	
	Please make a widget that can up- and down-scale a 3D image anisotropically along x, y, z. The type of interpolation for each axis can be set independently.	3D image	
	Please make a widget that takes a 2D RGB image and returns one of the following layers: either one for the hue, for the saturation, or luminance (user choice). The resulting layer values should range between 0f and 1f for luminance and saturation but be RGB (fully saturated, maximal luminance) for the hue.	2D RGB image	
	Please write a widget to convert a 2D image from RGB to grayscale. The weights used to project the color pixels to grayscale are configurable. Ensure weights sum to 1. Add a gamma parameter for each component R, G, and B before projection. Apply the inverse geometric average gamma transformation to the resulting gray-scale value.	2D RGB image	
	Please make a widget that computes the max projections of a 3D monochrome image along an arbitrary axis parametrized by Euler angles. Use the Fourier transform and the projection-slice theorem. Pad the input image to have the same dimension along all axes if necessary.	3D image	
	Make a widget that fuses two 2D images using the wavelet transform. Offer the choice of transform. Expose relevant parameters	Two 2D grayscale images.	You will need to install PyWavelets and restart napari for Omega to be able to find it. Omega will attempt to install

... (continued in corresponding [file](#))

**Supplementary Table 1. Example widgets.** Here is a list of example widgets that can be reliably generated using Omega. See the full table in the corresponding [file](#). These prompts can be modified, adjusted, and extended in many ways. If Omega can't make a functional plugin the first time, or if the result is not exactly what is asked for, being more explicit and asking Omega to 'try again' often works. Ideally, these widgets are made once and then can be reused by running the code in Omega's code editor (see Supp. Fig. 2).

Reproducibility test — each prompt is run 10 times, and the success rate is the ratio of successes over tries.

Supplementary Video	Prompts:	Dataset to preload:	Success rate:
<b>Prompt 1</b>	Segment the cell nuclei present in the selected layer. Once done, count the number of segmented nuclei and measure the average segment area.	Load the 'Human Mitosis' built-in napari sample dataset.	80%
<b>Prompt 2</b>	Convert the selected RGB image to grayscale using human-perception-based weights. Then, apply a light Gaussian blur of sigma 1.5 to the resulting grayscale image. Finally, create a new RGB image that visualizes the gradient of this grayscale image: the hue is proportional to the angle of the gradient vector at each pixel, and the luminance is proportional to the magnitude of the gradient.	Load the 'Astronaut' RGB image.	90%
<b>Prompt 3</b>	Please create a widget that takes a 2D image as input and splits it into $k \times k$ tiles, where $k$ is a parameter in the range [4, 256]; snap this parameter to the closest common divisor of both $x$ and $y$ dimensions. For each tile, compute the entropy. The output should be an image of the same dimensions as the input, which is obtained by upscaling the image of tile entropies (one pixel per tile) using a parameterizable interpolation method. The tile size should be a parameter of the widget.	Load the camera image.	80%
<b>Prompt 4</b>	Please write a widget that color projects a 3D stack along the Z axis. The hue of the projected pixel is proportional to the depth of the voxel of max intensity, the luminance is proportional to that max intensity, and the saturation is proportional to the contrast between the max intensity and the average intensity.	Load the 'Cells (3D + 2C)' built-in napari sample dataset. Keep the nuclei channel	100%
<b>Prompt 5</b>	Make a widget to rotate the hue of an RGB image and adjust the gamma for the luminance and the gamma for saturation. The default for the angle and gamma values is 1.0. The range of the gamma values is: [0.01, 10].	Load the 'Astronaut' RGB image.	100%
<b>Average</b>			90%

All prompts were run with default Omega settings and model 'gpt-4-0125-preview'.

**Supplementary Table 2. Reproducibility Analysis.** We conducted a reproducibility analysis in which two complex multi-step prompts and three widget generation prompts were run ten times to assess reproducibility — a total of 50 runs. The results suggest that widget generation is more robust than complex multi-step tasks. We observed a 90% reproducibility rate (ratio of successful attempts versus all ten attempts). Each run was run independently with a blank conversation history. In general, our observation is that if Omega fails to follow instructions, it often suffices to ask Omega to “try again”.

Omega Paper Prompts for Supplementary Videos

Supplementary Video	Prompts:	Dataset to load and notes:
Supp. Video 1	Segment the cell nuclei present in the selected layer.	Load the 'Human Mitosis' built-in napari sample dataset.
	Count the number of segmented nuclei.	
	Write a csv file on the desktop that lists all segmented nuclei, together with their area, and coordinates. Sort the rows by decreasing area. Open the file when done.	
Supp. Video 2	Segment the cell nuclei on the selected 3D image.	Load the 'Cells (3D + 2C)' built-in napari sample dataset. Keep the nuclei channel
	Count the number of segments	
Supp. Video 3	Make a step-by-step plan to segment nuclei in a 2D image. The nuclei are brighter than the background.	Load the 'Human Mitosis' built-in napari sample dataset.
	Let's apply step 3	
	For step 4, let's do two erosions	
	Would it not be better to apply erosions first on the grey-level image instead?	
	Let's do step 3 with just one opening operation.	
	Let's do one step of erosion on the selected image please.	
	Let's go back to the steps, apply 4 and 5.	
Supp. Video 4	Segment the nuclei on the selected 2D image.	Load the 'Human Mitosis' built-in napari sample dataset.
	Make a widget that takes a labels layer and returns a new labels layer but filtered. Only labels within a provided range of areas (min_area, max_area) are kept in this new layer.	
Supp. Video 5	Please write a widget that color projects a 3D stack along the Z axis. The hue of the projected pixel is proportional to the depth of the voxel of max intensity, the luminance is proportional to that max intensity, and the saturation is proportional to the contrast between the max intensity and the average intensity.	Load the 'Cells (3D + 2C)' built-in napari sample dataset. Keep the nuclei channel
Supp. Video 6	Please make a widget that returns the FFT spectrum of a 2D image as the absolute logarithm of the Fourier transform magnitude. Ensure that the DC component is at the center of the image. Use reflection padding and apodization to reduce artifacts due to discontinuities at the image borders.	Load the camera image and load the 'Cells (3D + 2C)' built-in napari sample dataset. Keep the nuclei channel. This way, one can test the original 2D widget and the one modified by the AI tool in the editor.
Supp. Video 7	N/A	
Supp. Video 8	Hello, who are you?	
	Please create a 2D image of dimensions 2000x2000 filled with random Gaussian noise of zero mean and sigma=100	
Supp. Video 9	Segment image on first layer using the SLIC superpixel segmentation algorithm.	Load the 'coffee cup' built-in napari sample image.

... (continued in corresponding [file](#))

**Supplementary Table 3. Prompt Table.** This table (see the full table in the corresponding [file](#)) lists all prompts used for all

