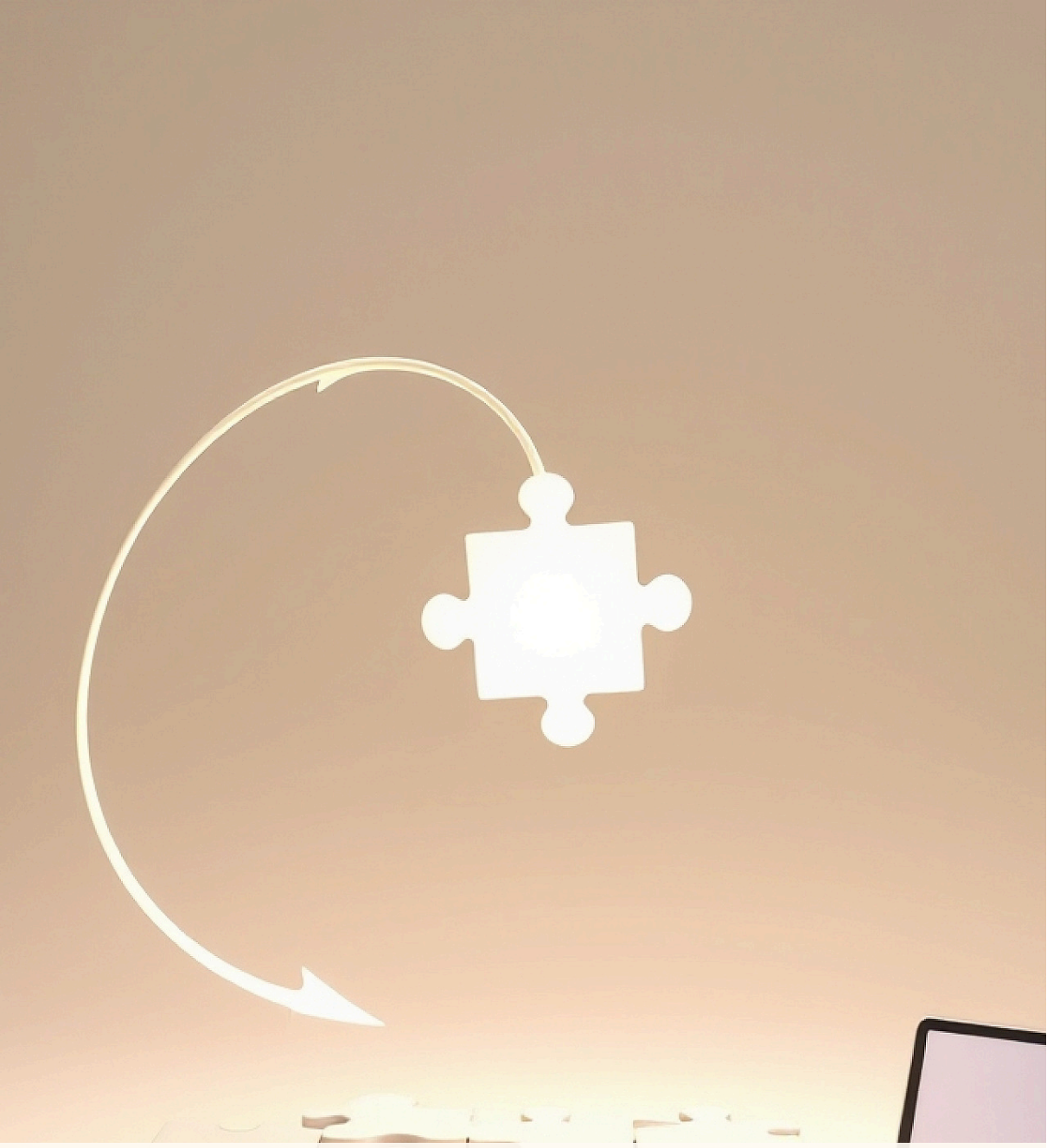
What is an asynchronous operation?



www.yzokul.com.tr

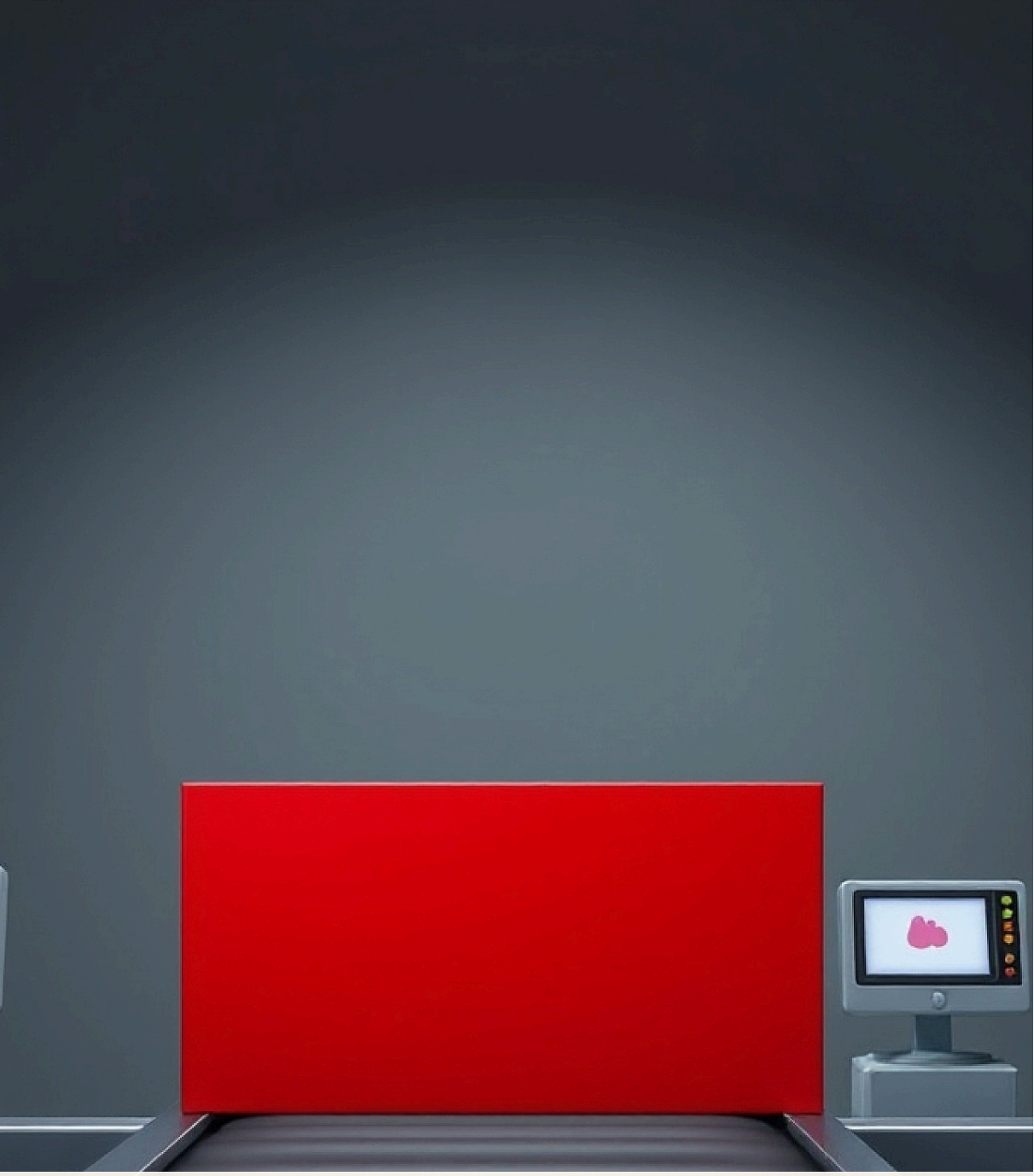
What is an asynchronous operation?



What is an asynchronous operation? It is a way for a program to start a slow task and, instead of sitting idle, do other useful work while waiting. When the slow task finishes, the program picks it up again. For example, while data is being fetched from the internet, the program can still respond to button clicks. The short answer is: it lets a program do other things while waiting for a slow task.



Picture a program doing tasks one after another, like a single worker at a desk. The worker picks up one task, finishes it, then moves to the next. This is the normal, step-by-step way of working. It is simple and predictable, and it is the starting point for understanding the difference with asynchronous operations.



When the worker waits for a slow task, like fetching data from the internet, everything else stops. The desk is stuck until the result arrives. No other task can be handled during that time. This is called blocking: the wait blocks all further progress. For example, if the program is waiting for a web page to load, it cannot update the screen or respond to clicks.



The long wait is empty time where nothing useful happens. The worker just sits there, unable to do anything else. This shows why waiting is a problem worth solving. For example, if fetching data takes five seconds, those five seconds are wasted if the program does nothing else. Using that time for other work would be much better.



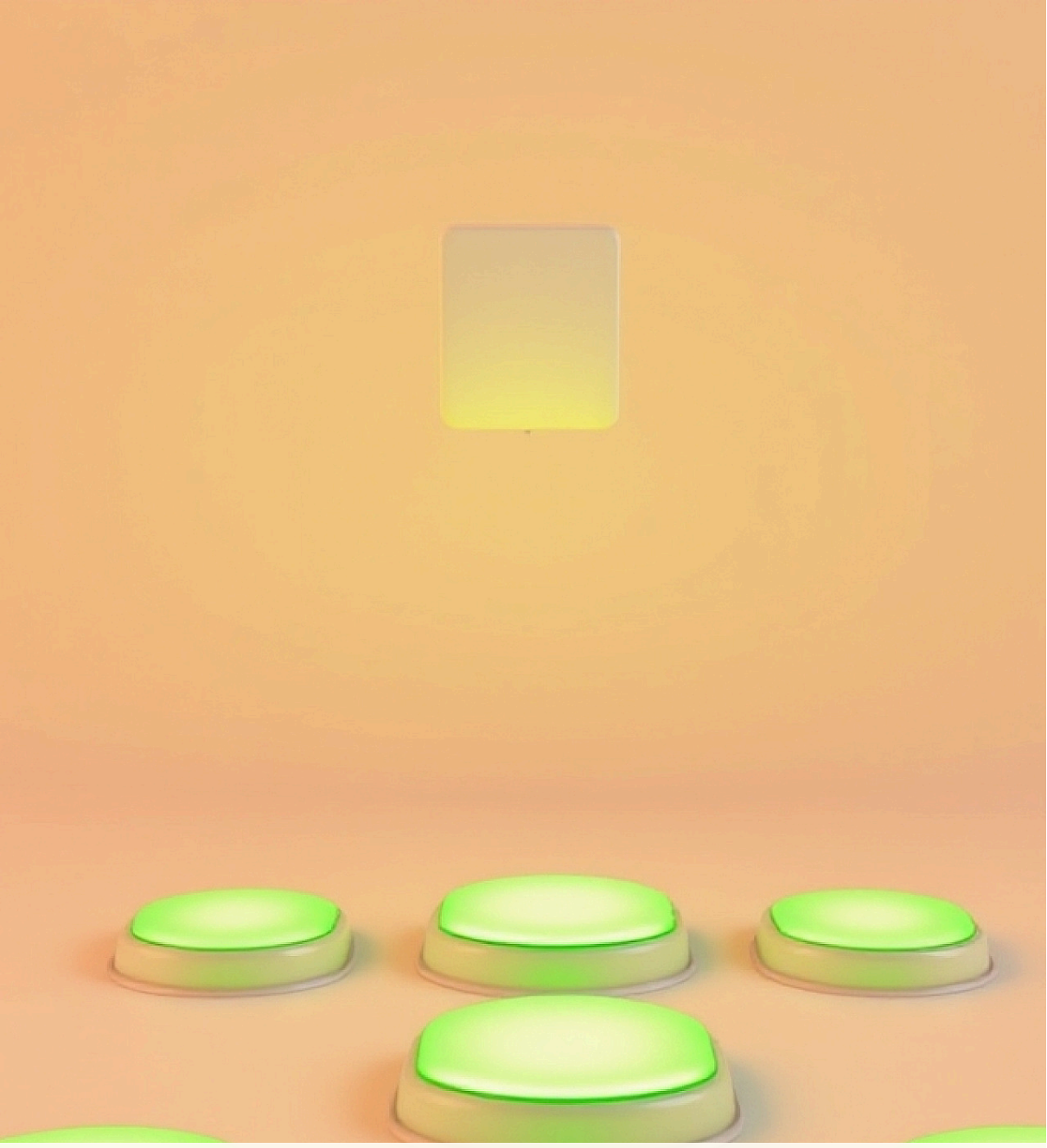
Imagine a program asks the internet for some data, and that request takes a long time. The request goes out, and the answer takes a while to come back. During this time, the program could be doing other things, but in the simple step-by-step way, it just waits. For example, a weather app asks a server for today's forecast, and the reply may take a few seconds.



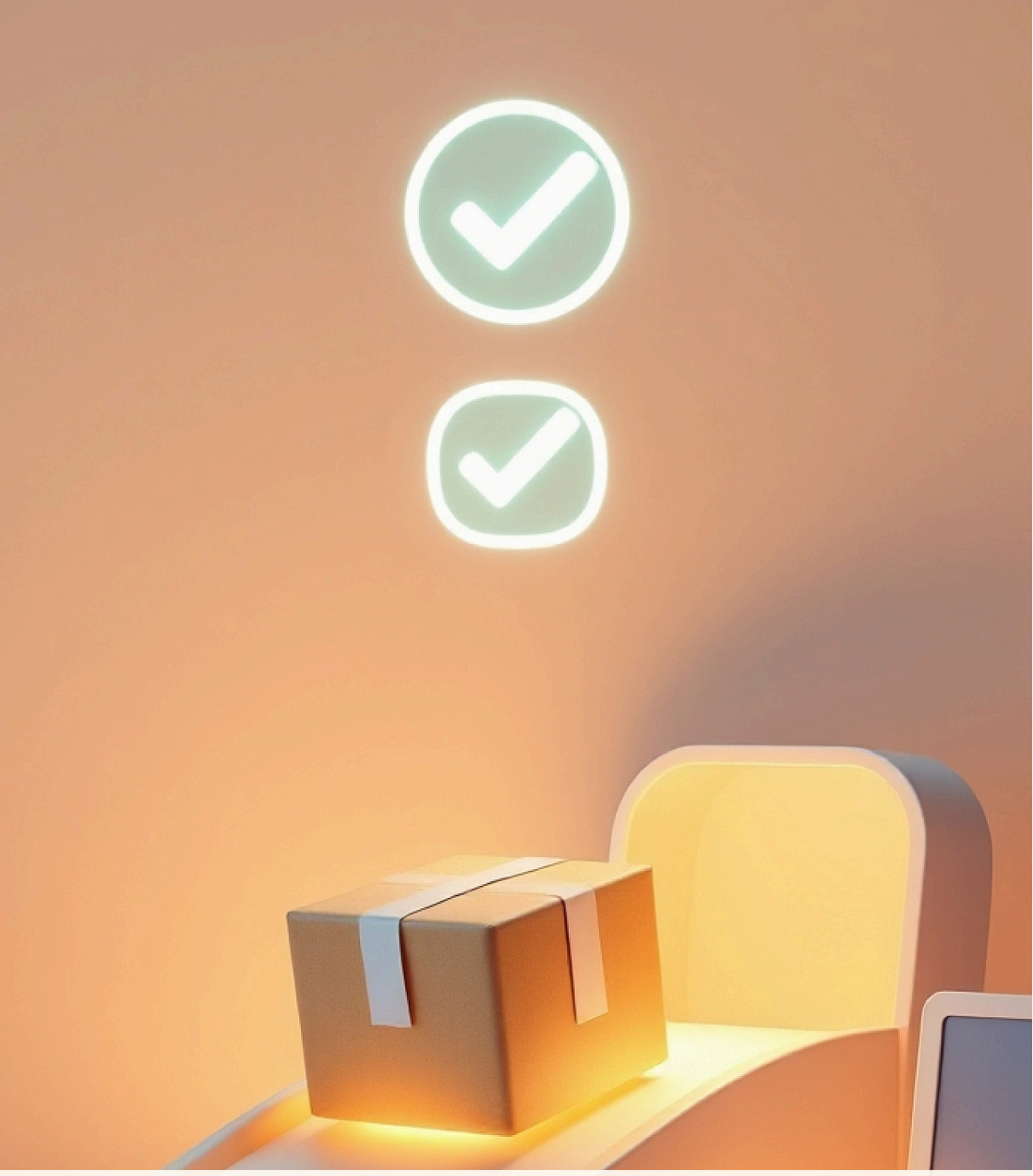
While the data is being fetched, the interface in the example does not respond. The user clicks a button and nothing happens. The buttons seem frozen, which feels broken. For example, if you try to switch tabs or press a refresh button during a long download, the program may ignore your clicks until the download finishes.



Instead of sitting idle, the program can attend to other tasks while the slow operation is in progress. The worker starts other work while the request is out. This is the key idea of asynchronous operations: do not wait doing nothing; do something else. For example, while the weather data is being fetched, the program can let the user type in a city name or scroll the page.



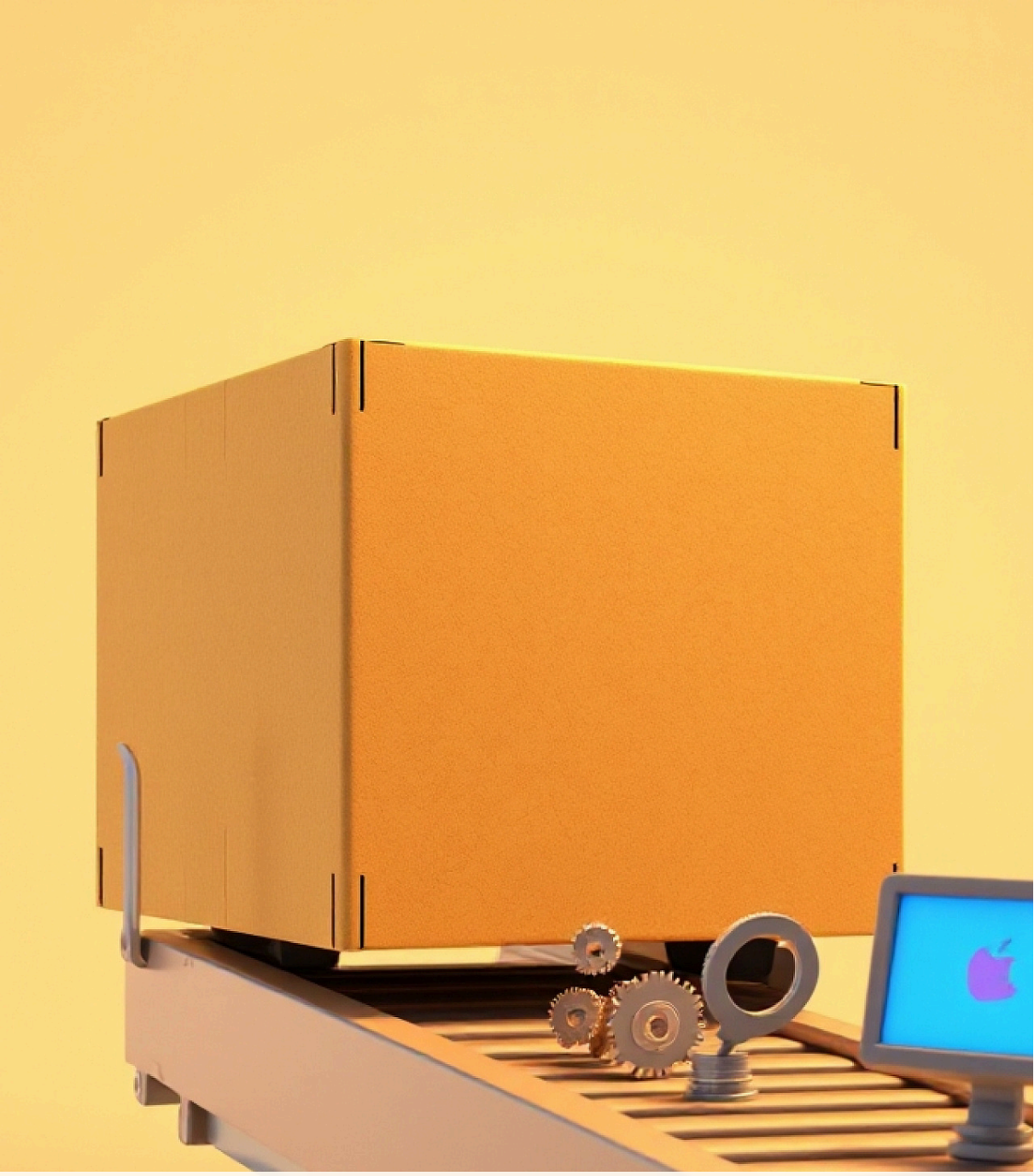
With asynchronous operation, the same example works differently: the buttons still respond while the data is fetched. Clicking works, and the interface stays alive. For example, you can press a menu button, open a settings panel, or type in a search box even though the weather data has not arrived yet. The program handles these small tasks while the big request is still pending.



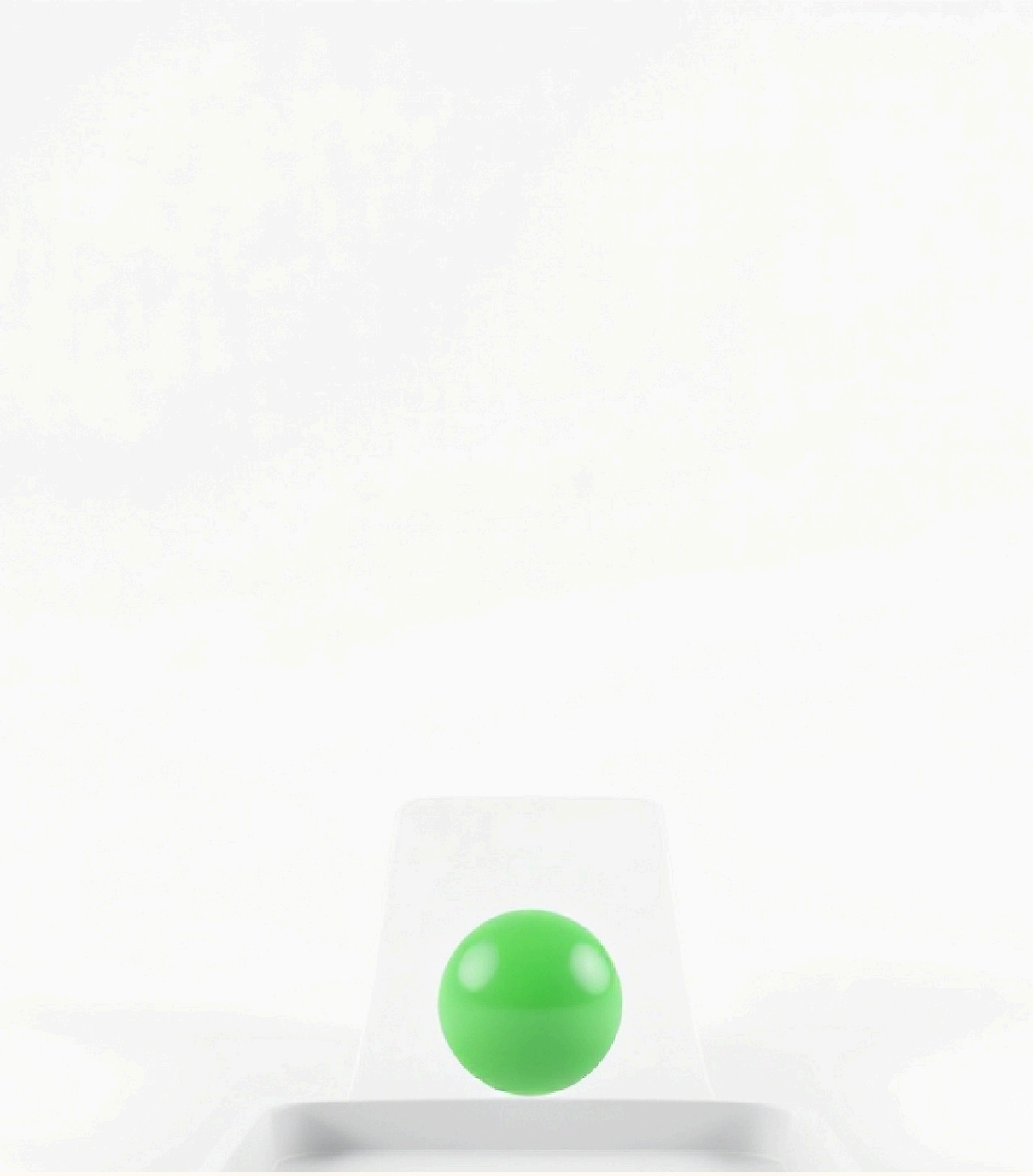
Finally, the data arrives. The program picks up the paused operation where it left off. The result is used, and the relevant operation continues. For example, when the weather data arrives, the app displays the forecast on the screen. The work that was waiting now moves forward, and the user sees the result.



Asynchronous operations do not always make the computation faster. The slow task still takes the same time to complete. Only the waiting is used better. For example, fetching data from the internet still takes five seconds, but during those five seconds the program can do other things. The total time for the slow task is unchanged; the benefit is that other work gets done.



The main benefit is clear: waiting time is used efficiently for other work. This is the real point of asynchronous operations. For example, while a large file downloads, you can continue writing an email or browsing another page. The waiting time is not wasted; it is filled with useful tasks that do not depend on the download.



Tasks may finish in a different order than they started. A task started later can complete earlier, and that is normal here. For example, you might start a quick request for a small image and a slow request for a large video. The small image may arrive first even though it was requested second. The program must handle results as they come, not in the order the tasks began.



A common mistake is thinking asynchronous means faster computation. It does not. The picture shows the same slow task taking the same time, with other work fitted around it. For example, if a calculation takes ten seconds, making it asynchronous will not shorten those ten seconds. It only allows other tasks to run during the wait. Do not expect asynchronous code to speed up the slow operation itself.



Another common mistake is assuming tasks finish in the order they started. The picture shows completion order differing from start order. For example, if you start three downloads at the same time, the smallest file may finish first, then the medium one, and the largest last, even if you started them in a different order. Always handle results as they arrive, not in the order you began.



A third mistake is thinking every task should be asynchronous. Only operations with real waiting, like fetching data from the internet, benefit from it. For example, a simple addition or a quick calculation does not need to be asynchronous; making it so adds complexity without benefit. Use asynchronous operations when there is a genuine wait that can be filled with other work.



Remember these points: an asynchronous operation lets the program do other tasks while waiting, keeps things like buttons working, uses waiting time well, is not always faster, and may finish out of order. For example, a weather app that fetches data asynchronously stays responsive, lets you click around, and shows the forecast when it arrives. Keep these ideas in mind whenever you meet asynchronous operations.