

How to open a file in Emacs

January 03, 2021 · 74 mins read

A short story about Lisp, technology, and human progress.

Table of contents

Part One: A Lispy Adventure

- The computing experience

- Opening a file

- Searching files

- Trying things out

- Writing a function

Part Two: Computers, and Humans

- The values of Emacs

- Cathedrals, Bazaars, and Fusion Reactors

- From *catching up* to *getting ahead*

- The Why of technology

Part One: A Lispy Adventure



Figure 1: Arun Chanchal

I've recently joined a company that for security reasons doesn't allow their source code on laptops. Development happens strictly on workstations inside their private networks, with some using their text editor's support for remote file editing and others running editors on those machines via SSH.



Adapting to this situation has indirectly led me into a bit of a rabbit hole, forcing me to acknowledge my core values, better understand the relation between progress and human flourishing, and ponder about the question: why technology?

(Chapters are mostly self-contained.)

The computing experience

One of the oldest pieces of software still in use was recently described as

A sort of hybrid between Windows Notepad, a monolithic-kernel operating system, and the International Space Station.

Of course, they were talking about Emacs. And yes, it is kinda true.



I've been using Emacs for a while and had opportunities to use it to work on projects in remote machines. There are a few quirks, but after changing a setting here and there, [Tramp](#) is mostly usable. Tramp makes it possible to transparently treat remote directories, files, commands, and more, *as if they were local*. Opening a remote buffer via `M-x find-file /ssh:user@remote-host:/some/project` and having tools like Magit and Eshell work out of the box feels like magic.

I'm not aware of similar functionality in Vim, but VSCode users recently got [something pretty close](#).

Emacs runs either as a standalone graphical application (GUI) or in a terminal emulator (TUI). In terms of features, GUI Emacs can be seen as a superset of TUI Emacs. Among many

other things, it makes it possible to display:

- images, PDFs, rendered web pages (even YouTube!)
- graphical popups (with code documentation, linting tips, compilation errors, completion candidates, etc.)
- heterogeneous fonts (distinct sizes, families, emphases, etc.)

TUI Emacs has the terminal emulator (and also commonly tmux) limiting and conflicting with the clipboard, keybindings, colors, and more. None of these limitations are present in GUI Emacs.

For these and other reasons I use GUI Emacs and Tramp to work on remote projects instead of TUI Emacs via SSH. Some do the opposite, and that's fine too! Even with the limitations shown above, TUI Emacs is very powerful, and its performance in an SSH session is still superior to Tramp.

Even so, one might think displaying images in Emacs is not a big deal. Just `open image.png` and get an actual image viewer application to render it, right? Things get interesting in a remote Tramp buffer: `M-x find-file image.png` will display the *remote* image right there in your *local* Emacs instance. Or in case you're in a remote *dired* buffer, pressing `RET` on an image file will do the same. It's a non-disruptive workflow that allows one to remain in the comfort of Emacs.

dired is a built-in file manager.

If you're not an Emacs user, chances are the previous sentence doesn't carry much weight. *What's good about staying inside Emacs?*

It is safe to assume that most of your *development*, or more broadly, your *computing environment* is represented below:

- operating system (macOS, GNU/Linux distribution, Windows, etc.)
- window manager (the one provided by your OS, i3, Openbox, XMonad, etc.)
- terminal emulator (iTerm2, xterm, rxvt, alacritty, etc.)
- shell (bash, zsh, fish, etc.)
- terminal multiplexer (tmux, GNU Screen, etc.)
- text editor or IDE (Vim, VSCode, Xcode, IntelliJ IDEA, etc.)
- email, web browsing, multimedia, and communications

Each item above is a discrete computer program. They're extensible in disconnected ways, and to varying degrees. They're *islands*. Most of the time integrating them is an uphill battle, and commonly just plain impractical. Having used a computing environment centered around TUI Vim and tmux for a decade, I know the pain of trying to make the shell, the



terminal, and every command line application running in it have the same look, feel, and behavior of being inside Vim. It is impossible.

The emergence of the web browser as a software platform is in part an answer to this disconnectedness.

In contrast, Emacs unifies and equalizes the computing experience as much as one desires, so that it happens—and is extended—in a single, cohesive environment.

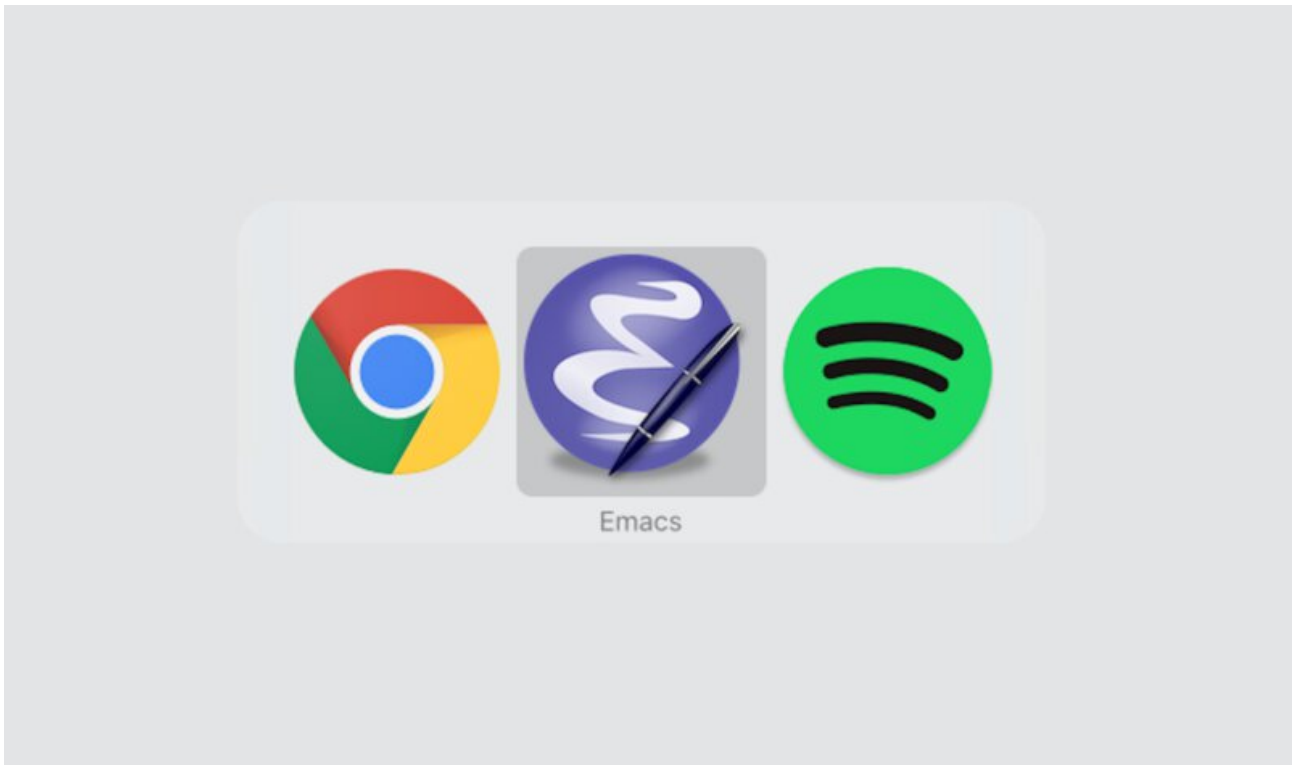


Figure 3: My computing environment.

People commonly [use Emacs](#) to process their email inbox, communicate in chat, navigate the web, write code, and prose. At their core, these activities are one and the same: text-editing. Being able to perform them using the same keybindings and functionality for movement, search, text manipulation, completion, undo-redo, copy-paste, is already a big deal, but it's not all: their integration into a single environment unlocks pleasant and efficient workflows that would be much less convenient elsewhere.

Creating a to-do item for something your partner just asked you (while you were concentrated on a task) that will show up automatically on your agenda tomorrow? Just a few keystrokes and you're back to the task. Converting some text notes you quickly scribbled down into a beautiful PDF via LaTeX, uploading it to S3, and referencing the URL in a reworded existing git commit message? If you're an experienced Emacs user, chances are you can *visualize* effortlessly and efficiently executing these actions without ever leaving Emacs.

This interconnectedness is part of the value of computing environments like the one provided by Emacs. The whole becomes greater than the sum of its parts.



Programmability gives it a further dimension of value: parts can be combined as you see fit, in arbitrary and possibly unforeseen ways.

Now, back to the *displaying an image from a remote host* use case. For that, I can think of some alternatives:

- `scp` the image file between hosts and `open` it locally
- Mount the remote filesystem and `open` it locally
- `ssh` into the remote host from a modern terminal emulator capable of rendering images and use a program like [icat](#)

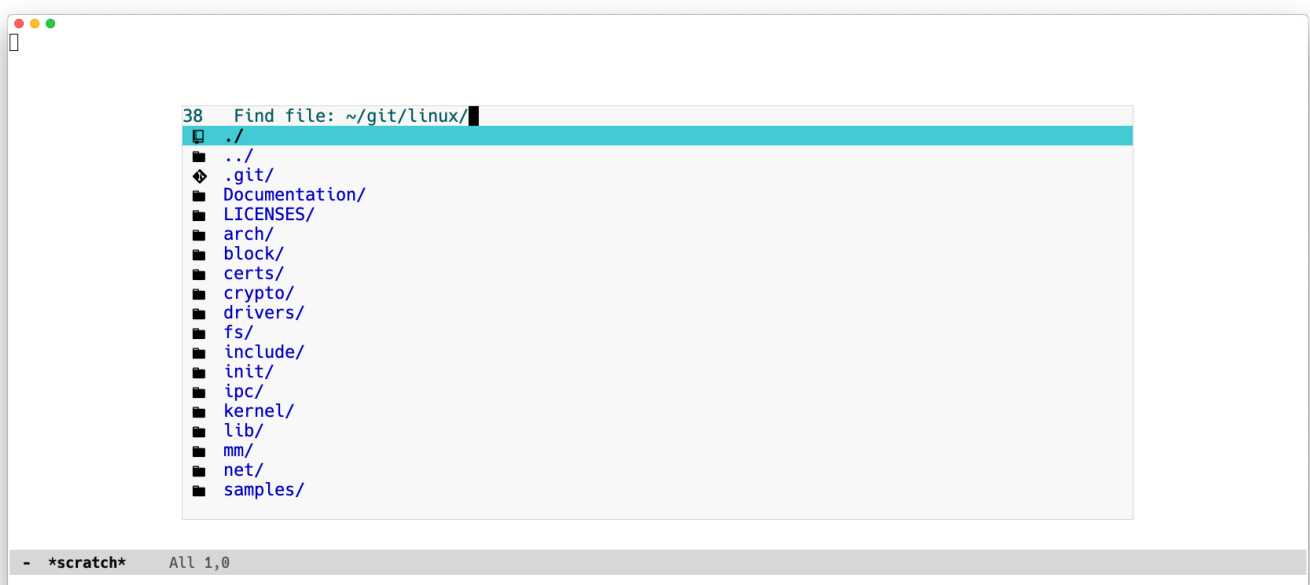
In Emacs, you just press enter.

Opening a file

At work I contribute to a moderately-sized monorepo at 70 thousand files, 8-digit lines of code and hundreds of PRs merged every day. One day I opened a remote buffer at that repository and ran `M-x find-file .`

`find-file` is an interactive function that shows a narrowed list of files in the current directory, prompts the user to filter and scroll through candidates, and for a file to open.

Emacs froze for 5 seconds before showing me the `find-file` prompt. Which isn't great, because when writing software, opening files is actually something one needs to do all the time.



Luckily, Emacs is “the extensible, customizable, self-documenting real-time display editor”, and comes with profiling capabilities: `M-x profiler-start` starts a profile and `M-x profiler-report` displays a call tree showing how much CPU cycles are spent in each function call after starting the profile. Starting a profile and running `M-x find-file` showed that all time was being spent in a function called `ffap-guess-file-name-at-point`, which was being called by `file-name-at-point-functions`, an [abnormal hook](#) run when `find-file` is called.

If you're familiar with Vim you can think of Emacs *hooks* as Vim *autocommands*, only with much better ergonomics.

I checked the documentation for `ffap-guess-file-name-at-point` with `M-x describe-function ffap-guess-file-name-at-point` and it didn't seem to be something essential, so I removed the hook by running `M-x eval-expression`, writing the form below, and pressing `RET`.

```
1 (remove-hook 'file-name-at-point-functions 'ffap-guess-file-name-at-point)
```

This solved the immediate problem of Emacs blocking for 5 seconds every time I ran `find-file`, with no noticeable drawbacks.

As I write this I attempt to reproduce the issue by re-adding `ffap-guess-file-name-at-point` to `file-name-at-point-functions`. I can't reproduce it anymore. The initial issue might have been

- caused by having manually mutated the Emacs environment via ad-hoc code evaluation (drifting from the state defined in configuration)
- caused by settings or packages that aren't in my configuration anymore
- fixed by settings or packages that were recently added to my configuration
- fixed by some recent package upgrade

Or some combination of the above. I have no idea exactly what. Which is to say: maintaining Emacs configurations is complicated.

Searching files

I could now navigate around and open files. The next thing I tried in this remote git repository was searching through project files. The great *projectile* package provides the `projectile-find-file` function for that, but I had previously given up making *projectile* perform well with remote buffers; given how things are currently implemented it seems to be [impractical](#). So I installed the *find-file-in-project* package for use on remote projects exclusively: `M-x package-install find-file-in-project`.

Most Emacs commands are accessible via key combinations, with defaults that can be customized to be anything you want. I'll stick to referencing command names themselves instead of their default keybindings.



Both `projectile-find-file` and `find-file-in-project` (aliased as `ffip`):

- show a narrowed list of all project files in the minibuffer
- prompt the user to filter and scroll through candidates
- open a file when `RET` is pressed on a candidate.



To disable projectile on remote buffers I had the following form in my configuration.

```
1 (defadvice projectile-project-root (around ignore-remote first activate)
2   (unless (file-remote-p default-directory 'no-identification) ad-do-it))
```

Which causes the `projectile-project-root` function to not run its usual implementation on remote buffers, but instead return `nil` unconditionally. `projectile-project-root` is used as a way to either get the project root for a given buffer (remote or not), or as a boolean predicate to test if the buffer is in a project (e.g., a git repository directory). Having it return `nil` on remote buffers effectively disables projectile on remote buffers.

Emacs [advice](#)s are a way of modifying the behavior of existing functions without having to redefine them. They serve a similar purpose as hooks, but are more flexible.

I then wrote a function that falls back to `ffip` when projectile is disabled and bound it to the keybinding I had for `projectile-find-file`, so that I could press the same keybinding whenever I wanted to search for projects files, and not have to think about whether I'm on a remote buffer or not:

```
1 (defun maybe-projectile-find-file ()
2   "Run `projectile-find-file' if in a project buffer, `ffip' otherwise."
3   (interactive)
4   (if (projectile-project-p)
```



```
5      (projectile-find-file)
6      (ffip)))
```

projectile-project-p uses projectile-project-root internally.

And called it:

```
M-x maybe-projectile-find-file
```

Emacs froze for **30 seconds**. After that, it showed the prompt with the narrowed list of files in the project. 30 seconds! What was it doing during the whole time? Let's try out the profiler again.

1. Start a new profile:

```
M-x profiler-start
```

2. Call the function to be profiled:

```
M-x maybe-projectile-find-file (it freezes Emacs again for 30 seconds)
```

3. And display the report:

```
M-x profiler-report
```

Which showed:

1	Function	CPU	samples	%
2	+ ...	21027	98%	
3	+ command-execute	361	1%	

This tells us that 98% of the CPU time was spent in whatever ... is. Pressing `TAB` on a line will expand it by showing its child function calls.

1	Function	CPU	samples	%
2	- ...	21027	98%	
3	+ ivy--insert-minibuffer	13689	64%	
4	+ #<compiled 0x131f715d2b6fa0a8>	3819	17%	
5	Automatic GC	2017	9%	
6	+ shell-command	1424	6%	
7	+ ffip-get-project-root-directory	77	0%	
8	+ run-mode-hooks	1	0%	
9	+ command-execute	361	1%	

Expanding ... shows that Emacs spent 64% of CPU time in `ivy--insert-minibuffer` and 9% of the time—roughly 3 whole seconds!—garbage collecting. I had `garbage-collection-messages` set to `t` so I could already tell that Emacs was GCing a lot; enabling this setting makes a message be displayed in the echo area whenever Emacs garbage collects. I could also see the Emacs process consuming 100% of one CPU core while it was frozen and unresponsive to input.



The `profiler` package implements a [sampling profiler](#). The `elp` package can be used for getting actual wall clock times.

Drilling down on `#<compiled 0x131f715d2b6fa0a8>` shows that cycles there (17% of CPU time) were spent on Emacs waiting for user input, so we can ignore it for now.

As I get deep in drilling down on `ivy--insert-minibuffer`, names in the “Function” column start getting truncated because the column is too narrow. A quick Google search (via `M-x google-this emacs profiler report width`) shows me how to make it wider:

```
1 (setf (caar profiler-report-cpu-line-format) 80)
2      (caar profiler-report-memory-line-format) 80)
```

Describing those variables with `M-x describe-variable` shows that the default values are `50`.

From the profiler report buffer I run `M-x eval-expression`, paste the form above with `C-y` and press `RET`. I also persist this form to my configuration. Pressing `c` in the profiler report buffer (bound to `profiler-report-render-calltree`) redraws it, now with a wider column, allowing me to see the function names.

Here is the abbreviated expanded relevant portion of the call stack.

1	Function	CPU samples	%
2	- ffip	13586	63%
3	- ffip-find-files	13586	63%
4	- let*	13586	63%
5	- setq	13585	63%
6	- ffip-project-search	13585	63%
7	- let*	13585	63%
8	- mapcar	13531	63%
9	- #<lambda 0xb210342292>	13528	63%
10	- cons	13521	63%
11	- expand-file-name	12936	60%
12	- tramp-file-name-handler	12918	60%
13	- apply	9217	43%
14	- tramp-sh-file-name-handler	9158	42%
15	- apply	9124	42%
16	- tramp-sh-handle-expand-file-name	8952	41%
17	- file-name-as-directory	5812	27%
18	- tramp-file-name-handler	5793	27%
19	+ tramp-find-foreign-file-name-handler	3166	14%
20	+ apply	1237	5%
21	+ tramp-dissect-file-name	527	2%
22	+ #<compiled -0x1589d0aab96d9542>	337	1%
23	tramp-file-name-equal-p	312	1%
24	tramp-tramp-file-p	33	0%
25	+ tramp-replace-environment-variables	6	0%
26	#<compiled 0x1e202496df87>	1	0%
27	+ tramp-connectable-p	1006	4%
28	+ tramp-dissect-file-name	628	2%
29	+ eval	517	2%
30	+ tramp-run-real-handler	339	1%
31	+ tramp-drop-volume-letter	60	0%
32	tramp-make-tramp-file-name	30	0%
33	+ tramp-file-name-for-operation	40	0%
34	+ tramp-find-foreign-file-name-handler	2981	13%
35	+ tramp-dissect-file-name	518	2%
36	tramp-tramp-file-p	34	0%



37	#<compiled 0×1e202496df87>	1	0%
38	+ tramp-replace-environment-variables	1	0%
39	+ replace-regexp-in-string	153	0%
40	+ split-string	15	0%
41	+ ffip-create-shell-command	4	0%
42	cond	1	0%

A couple of things to unpack here. From lines 8-11 it could deduced that `ffip` maps a lambda that calls `expand-file-name` over all completion candidates, which in this case are around 70 thousand file names. Running `M-x find-function ffip-project-search` and narrowing to the relevant region in the function shows [exactly that](#):

`find-function` shows the definition of a given function, in its source file.

`find-file-in-project.el`

```
1 (mapcar (lambda (file)
2         (cons (replace-regexp-in-string "^\\." "" file)
3               (expand-file-name file)))
4         collection)
```

On line 11 of the profiler report we can see that 60% of 30 seconds (18 seconds) was spent on `expand-file-name` calls. By dividing 18 seconds by 70000 we get that `expand-file-name` calls took 250µs on average. 250µs is how long a modern computer takes to [read 1MB sequentially from RAM](#)! Why would my computer need to do that amount of work 70000 times just to display a narrowed list of files?

Trying things out

Let's see if the function documentation for `expand-file-name` provides any clarity.

`M-x describe-function expand-file-name`

```
1 expand-file-name is a function defined in C source code.
2
3 Signature
4 (expand-file-name NAME &optional DEFAULT-DIRECTORY)
5
6 Documentation
7 Convert filename NAME to absolute, and canonicalize it.
8
9 Second arg DEFAULT-DIRECTORY is directory to start with if NAME is relative
10 (does not start with slash or tilde); both the directory name and
11 a directory's file name are accepted. If DEFAULT-DIRECTORY is nil or
12 missing, the current buffer's value of default-directory is used.
13 NAME should be a string that is a valid file name for the underlying
14 filesystem.
```

Ok, so it sounds like `expand-file-name` essentially transforms a file path into an absolute path, based on either the current buffer's directory or optionally, a directory passed in as an additional argument. Let's try evaluating some forms with `M-x eval-expression` both on a local and a remote buffer to get a sense of what it does.



In a local dired buffer at my local home directory:

```
*dired /Users/mpereira @ macbook*
```

```
1 (expand-file-name "foo.txt")
2 ;; ⇒ "/Users/mpereira/foo.txt"
```

In a remote dired buffer at my remote home directory:

```
*dired /home/mpereira @ remote-host*
```

```
1 (expand-file-name "foo.txt")
2 ;; ⇒ "/ssh:mpereira@remote-host:/home/mpereira/foo.txt"
```

The `expand-file-name` call in `ffip-project-search` doesn't specify a `DEFAULT-DIRECTORY` (the optional second parameter to `expand-file-name`) so like in the examples above it defaults to the current buffer's directory, which in the profiled case is a remote path like in the second example above.

With a better understanding of what `expand-file-name` *does*, let's now try to understand how it *performs*. We can benchmark it with `benchmark-run` in local and remote buffers, and compare their runtimes.

```
M-x describe-function benchmark-run
```

```
1 benchmark-run is an autoloaded macro defined in benchmark.el.gz.
2
3 Signature
4 (benchmark-run &optional REPETITIONS &rest FORMS)
5
6 Documentation
7 Time execution of FORMS.
8
9 If REPETITIONS is supplied as a number, run forms that many times,
10 accounting for the overhead of the resulting loop. Otherwise run
11 FORMS once.
12 Return a list of the total elapsed time for execution, the number of
13 garbage collections that ran, and the time taken by garbage collection.
```

Benchmarking it in a local dired buffer at my local home directory

```
*dired /Users/mpereira @ macbook*
```

```
1 (benchmark-run 70000 (expand-file-name "foo.txt"))
2 ;; ⇒ (0.308712 0 0.0)
```

and in a remote dired buffer at my remote home directory

```
*dired /home/mpereira @ remote-host*
```

```
1 (benchmark-run 70000 (expand-file-name "foo.txt"))
2 ;; ⇒ (31.547211 0 0.0)
```

showed that it took 0.3 seconds to run `expand-file-name` 70 thousand times on a local buffer and 30 seconds to do so on a remote buffer: **two orders of magnitude slower**. 30 seconds



more than what we observed in the profiler report (18 seconds), and I'll attribute this discrepancy to unknowns; maybe the `ffip` execution took advantage of byte-compiled code evaluation, or there's some overhead associated with `benchmark-run`, or something else entirely. Nevertheless, this experiment clearly corroborates the profiler report results.

So! Back to `ffip`. Looking again at the previous screenshot, it seems that the list of displayed files doesn't even show absolute file paths. Why is `expand-file-name` being called at all? Maybe calling it isn't too important...



```
71188 Find in linux/:  
init/do_mounts_rd.c  
init/do_mounts.c  
init/do_mounts_initrd.c  
init/Makefile  
init/initramfs.c  
init/Kconfig  
init/init_task.c  
init/main.c  
init/calibrate.c  
init/do_mounts.h  
init/version.c  
init/noinitramfs.c  
crypto/crct10dif_common.c  
crypto/fcrypt.c  
crypto/hash_info.c  
crypto/blake2s_generic.c  
crypto/ecdsa_defs.h  
crypto/hmac.c  
crypto/md4.c
```

Let's remove the `expand-file-name` call by

1. visiting the `ffip-project-search` function in the library file with `M-x find-function ffip-project-search`
2. "raising" `file` in the lambda
3. re-evaluating `ffip-project-search` with `M-x eval-defun`

and see what happens.

`find-file-in-project.el`

```
1 (mapcar (lambda (file)  
2         (cons (replace-regexp-in-string "^\\./" "" file)  
3             (expand-file-name file))))  
4 +      (file))  
5      collection)
```

I run my function again:

`M-x maybe-projectile-find-file`



It's faster. This change alone reduces the time for `ffip` to show the candidate list *from 30 seconds to 8 seconds* with no noticeable drawbacks. Which is better, but still not even close to acceptable.



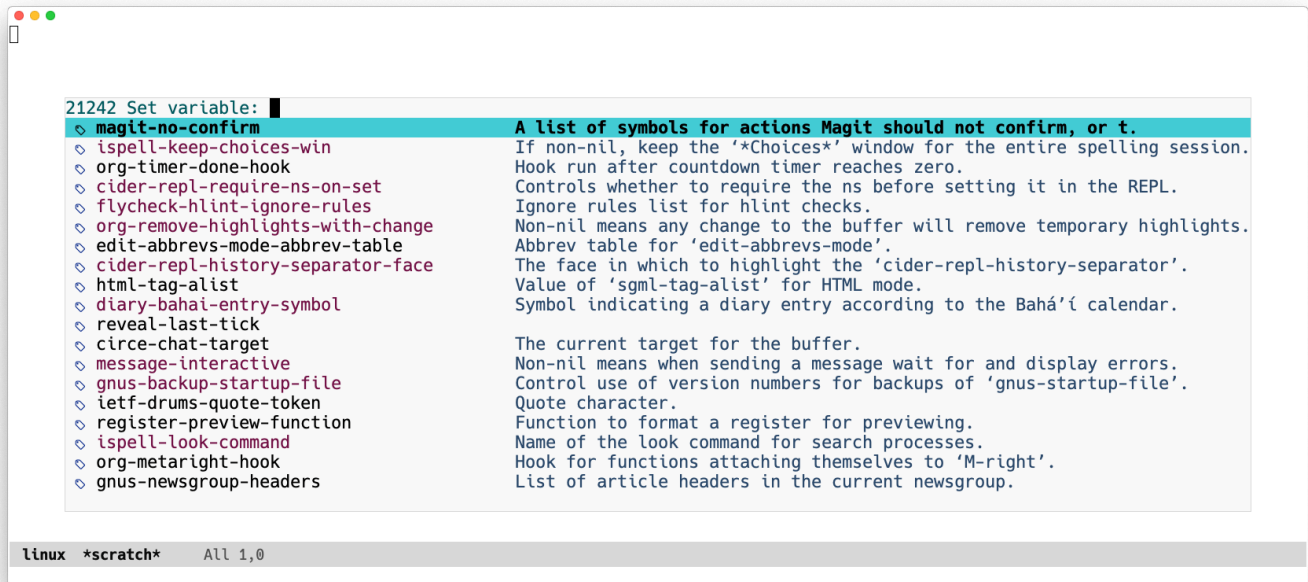
```
71188 Find in linux/:  
init/do_mounts_rd.c  
init/do_mounts.c  
init/do_mounts_initrd.c  
init/Makefile  
init/initramfs.c  
init/Kconfig  
init/init_task.c  
init/main.c  
init/calibrate.c  
init/do_mounts.h  
init/version.c  
init/noinitramfs.c  
crypto/crct10dif_common.c  
crypto/fcrypt.c  
crypto/hash_info.c  
crypto/blake2s_generic.c  
crypto/ecdsa_defs.h  
crypto/hmac.c  
crypto/md4.c
```

linux *scratch* All 1,0

Profiling the changed function shows that now most of the time is spent in sorting candidates with `ivy-prescient-sort-function`, and garbage collection. Automatic sorting of candidates based on selection recency comes from the excellent `ivy` and `ivy-prescient` packages, which I had installed and configured. Disabling `ivy-prescient` with `M-x ivy-prescient-mode` and re-running my function reduces the time further from 8 seconds to 4 seconds.

Another thing I notice is that `ffip` allows `fd` to be used as a backend instead of GNU find. `fd` claims to have better performance, so I install it on the remote host and configure `ffip` to use it. I evaluate the form below like before, but I could also have used the very handy `M-x counsel-set-variable`, which shows a narrowed list of candidates of all variables in Emacs (in my setup there's around 20 thousand) along with a snippet of their docstrings, and on selection allows the variable value to be set. Convenient!





```
1 (setq ffip-use-rust-fd t)
```

Which brings my function's runtime to a little over 2 seconds—a 15x performance improvement overall—achieved via:

1. Manually evaluating a modified function from an installed library file
2. Disabling useful functionality (prescient sorting)
3. Installing a program on the remote host and configuring `ffip` to use it

The last point is not really an issue, but the whole situation is not ideal. Even putting aside all of the above points, I don't want to wait for over 2 seconds every time I search for files in this project.

Let's see if we can do better than that.

Writing a function

So far we've been mostly *configuring* and *introspecting* Emacs. Let's now *extend* it with new functionality that satisfies our needs.

We want a function that:

1. Based on a remote buffer's directory, figures out its remote project root directory
2. Runs `fd` on the remote project root directory
3. Presents the output from `fd` as a narrowed list of candidate files, with it being possible to filter, scroll, and select a candidate from the list
4. Has good performance and is responsive even on large, remote projects



Let's see if there's anything in *find-file-in-project* that we could reuse. I know that `ffip` is figuring out project roots and running shell commands somehow. By checking out its library file with `M-x find-library find-file-in-project` (which opens a buffer with the installed `find-file-in-project.el` package file) I can see that the `shell-command-to-string` function (included with Emacs) is being used for running shell commands, and that there's a function named `ffip-project-root` that sounds a lot like what we need.

I have a keybinding that shows the documentation for the thing under the cursor. I use it to inspect the two functions:

`find-project-root`

```
1 ffip-project-root is an autoloaded function defined in
2 find-file-in-project.el.
3
4 Signature
5 (ffip-project-root)
6
7 Documentation
8 Return project root or default-directory.
```

`shell-command-to-string`

```
1 shell-command-to-string is a compiled function defined in
2 simple.el.gz.
3
4 Signature
5 (shell-command-to-string COMMAND)
6
7 Documentation
8 Execute shell command COMMAND and return its output as a string.
```

Perfect. We should be able to reuse them.

I also know that the `ivy-read` function provided by *ivy* should take care of displaying the narrowed list of files. Looks like we won't need to write a lot of code.

To verify that our code will work on remote buffers we'll need to evaluate forms in the context of one. The `with-current-buffer` macro can be used for that.

`M-x describe-function with-current-buffer`

```
1 with-current-buffer is a macro defined in subr.el.gz.
2
3 Signature
4 (with-current-buffer BUFFER-OR-NAME &rest BODY)
5
6 Documentation
7 Execute the forms in BODY with BUFFER-OR-NAME temporarily current.
8
9 BUFFER-OR-NAME must be a buffer or the name of an existing buffer.
10 The value returned is the value of the last form in BODY. See
11 also with-temp-buffer.
```

For writing our function, instead of evaluating forms ad-hoc with `M-x eval-expression`, we can open a scratch buffer and write and evaluate forms directly from there, which should be



more convenient.

I have a clone of the [Linux](#) git repository on my remote host. Let's assign a remote buffer for the officially funniest file in the Linux kernel, `jiffies.c` —

```
/ssh:mpereira@remote-host:/home/mpereira/linux/kernel/time/jiffies.c
```

—to a variable named `remote-file-buffer` by evaluating the following form with `eval-defun`.

`*scratch*`

```
1 (setq remote-file-buffer
2   (find-file-noselect
3     (concat "ssh:mpereira@remote-host:"
4             "/home/mpereira/linux/kernel/time/jiffies.c")))
5 ;; => #<buffer jiffies.c>
```

Notice that the buffer is just a value, and can be passed around to functions. We'll use it further ahead to emulate evaluating forms as if we had *that* buffer opened, with the `with-current-buffer` macro.

Let's start exploring by writing to the `*scratch*` buffer and continuing to evaluate forms one by one with `eval-defun`.

`*scratch*`

```
1 (shell-command-to-string "hostname")
2 ;; => "macbook"
3
4 default-directory
5 ;; => "/Users/mpereira/.emacs.d/"
6
7 (ffip-project-root)
8 ;; => "/Users/mpereira/.emacs.d/"
```

And now let's evaluate some forms in the context of a remote buffer. Notice that running `hostname` in a shell returns something different.

`*scratch*`

```
1 (with-current-buffer remote-file-buffer
2   (shell-command-to-string "hostname"))
3 ;; => "remote-host"
4
5 (with-current-buffer remote-file-buffer
6   default-directory)
7 ;; => "/ssh:mpereira@remote-host:/home/mpereira/linux/kernel/time/"
8
9 (with-current-buffer remote-file-buffer
10  (ffip-project-root))
11 ;; => "/ssh:mpereira@remote-host:/home/mpereira/linux/"
12
13 (with-current-buffer remote-file-buffer
14  (shell-command-to-string "fd --version"))
15 ;; => "fd 8.1.1"
16
17 (with-current-buffer remote-file-buffer
18  (executable-find "fd" t))
19 ;; => "/usr/bin/fd"
```



`executable-find` requires the second argument to be non-nil to search on remote hosts. Check `M-x describe-function executable-find` for more details.

Emacs is not only running shell commands, but also evaluating forms *as if it were running on the remote host*. That's pretty sweet!

Now that we made sure that the executable for `fd` is available on the remote host, let's try running some `fd` commands.

scratch

```
1 (with-current-buffer remote-file-buffer
2   (shell-command-to-string "pwd"))
3 ;; ⇒ "/home/mpereira/linux/kernel/time"
4
5 (with-current-buffer remote-file-buffer
6   (shell-command-to-string "fd --extension c | wc -l"))
7 ;; ⇒ 28
8
9 (with-current-buffer remote-file-buffer
10  (shell-command-to-string "fd . | head"))
11 ;; ⇒ Kconfig
12 ;;   Makefile
13 ;;   alarmtimer.c
14 ;;   clockevents.c
15 ;;   clocksource.c
16 ;;   hrtimer.c
17 ;;   itimer.c
18 ;;   jiffies.c
19 ;;   namespace.c
20 ;;   ntp.c
```

`fd` tells us that there are 28 C files in `/home/mpereira/linux/kernel/time`. Let's see if we can get the project root, which would be `/home/mpereira/linux`.

scratch

```
1 (with-current-buffer remote-file-buffer
2   (ffip-project-root))
3 ;; ⇒ "/ssh:mpereira@remote-host:/home/mpereira/linux/"
```

That seems to work.

Let's now play with `default-directory`. This is a *buffer-local* variable that holds a buffer's working directory. By evaluating forms with a redefined `default-directory` it's possible to emulate being in another directory, which could even be on a remote host. The code block below is an example of that—the second form redefines `default-directory` to be the project root.

scratch

```
1 (with-current-buffer remote-file-buffer
2   (shell-command-to-string "pwd"))
3 ;; ⇒ "/home/mpereira/linux/kernel/time"
4
5 (with-current-buffer remote-file-buffer
6   (let ((default-directory (ffip-project-root)))
```



```
7 (shell-command-to-string "pwd"))
8 ;; => /home/mpereira/linux
```

Nice!

I wonder how much Assembly and C are currently in the project.

scratch

```
1 (with-current-buffer remote-file-buffer
2   (let ((default-directory (ffip-project-root)))
3     (shell-command-to-string "fd --extension asm --extension s --exec-batch cat '{}' | wc -l")))
4 ;; => 373663
5
6 (with-current-buffer remote-file-buffer
7   (let ((default-directory (ffip-project-root)))
8     (shell-command-to-string "fd --extension c --extension h | xargs cat | wc -l")))
9 ;; => 27088162
```

Twenty seven million, eighty eight thousand, one hundred and sixty two lines of C, and almost half a million lines of Assembly. It's fine.

Alright, at this point it feels like we have all the pieces: let's put them together.

scratch

```
1 (defun my-project-find-file (&optional pattern)
2   "Prompt the user to filter, scroll and select a file from a list of all
3   project files matching PATTERN."
4   (interactive)
5   (let* ((default-directory (ffip-project-root))
6          (fd (executable-find "fd" t))
7          (fd-options "--color never")
8          (command (concat fd " " fd-options " " pattern))
9          (candidates (split-string (shell-command-to-string command) "\n" t)))
10    (ivy-read "File: "
11             candidates
12             :action (lambda (candidate)
13                      (find-file candidate)))))
```

This is a bit longer than what we've been playing with, but even folks new to Emacs Lisp should be able to follow it:

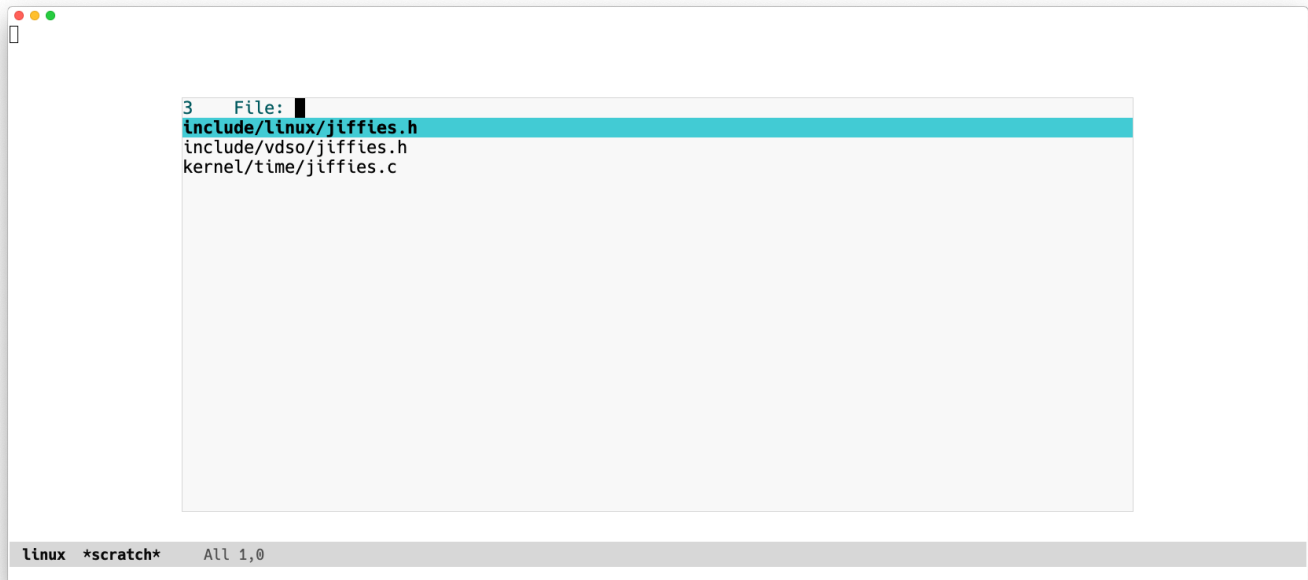
1. Redefine `default-directory` to be the project root directory (line 4)
2. Build, execute, and parse the output of the `fd` command into a list of file names (lines 5-8)
3. Display a file prompt showing a narrowed list of all files in the project (lines 9-12)

Let's see if it works.

scratch

```
1 (with-current-buffer remote-file-buffer
2   (my-project-find-file "jif"))
```



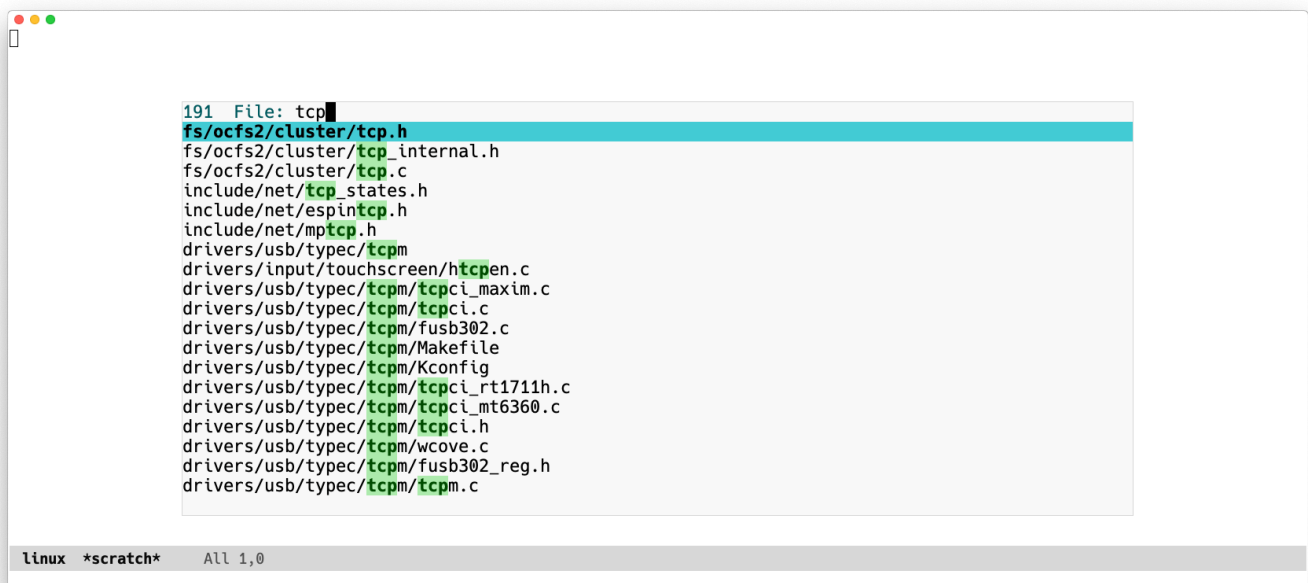


```
3 File:
include/linux/jiffies.h
include/vdso/jiffies.h
kernel/time/jiffies.c
```

linux *scratch* All 1,0

It does!

Since it was declared (interactive) we can also to call it via `M-x my-project-find-file` .



```
191 File: tcp
fs/ocfs2/cluster/tcp.h
fs/ocfs2/cluster/tcp_internal.h
fs/ocfs2/cluster/tcp.c
include/net/tcp_states.h
include/net/espintcp.h
include/net/mptcp.h
drivers/usb/typec/tcpm
drivers/input/touchscreen/htcpn.c
drivers/usb/typec/tcpm/tcpci_maxim.c
drivers/usb/typec/tcpm/tcpci.c
drivers/usb/typec/tcpm/fusb302.c
drivers/usb/typec/tcpm/Makefile
drivers/usb/typec/tcpm/Kconfig
drivers/usb/typec/tcpm/tcpci_rt1711h.c
drivers/usb/typec/tcpm/tcpci_mt6360.c
drivers/usb/typec/tcpm/tcpcl.h
drivers/usb/typec/tcpm/wcove.c
drivers/usb/typec/tcpm/fusb302_reg.h
drivers/usb/typec/tcpm/tcpm.c
```

linux *scratch* All 1,0

Going back to the large remote project and running `my-project-find-file` a few times shows that it now runs in a little over a second—a **30x improvement** compared with what we started with.

This is still not good enough, so I went ahead and evolved the function we were working on to most of the time show something on screen *immediately* and redraw it asynchronously. You can check out the code at fast-project-find-file.el.



As an aside: having the whole text editor block for over a second while I wait for it to show something so simple is unacceptable. Through desensitization and acquiescence, we, users of software have come to expect that it will either not work at all, not work consistently, or exhibit poor or unpredictable performance.

Jonathan Blow addresses this situation somewhat entertainingly in [“Preventing the Collapse of Civilization”](#).

* * *

Did you notice how the function implementation came almost naturally from exploration? The immediate feedback from evaluating forms and modifying a live system—even though old news to Lisp programmers—is incredibly powerful. Combine it with an “extensible, customizable, self-documenting” environment and you have a very satisfying and productive means of creation.

Part Two: Computers, and Humans

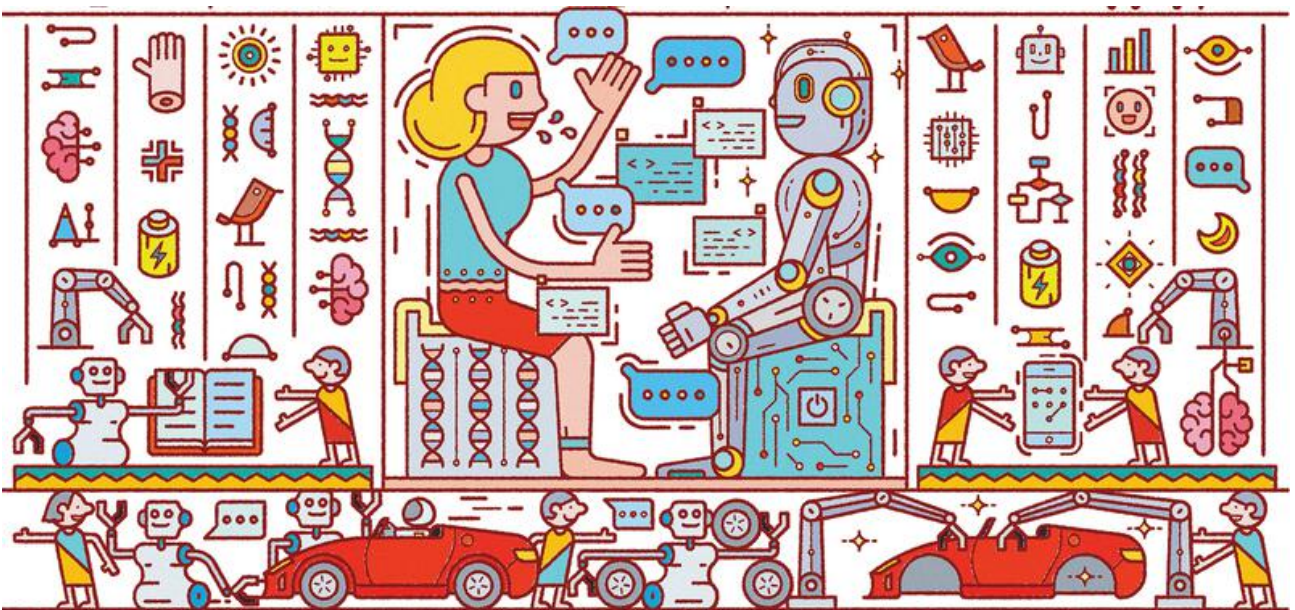


Figure 4: Kuo Cheng Liao

The values of Emacs

In 2018 Bryan Cantrill gave a brilliant talk where he shared his recent experiences with the Rust programming language. More profoundly, he explored a facet of software that is oftentimes overlooked: the *values* of the software we use. To paraphrase him slightly:

Values are defined as expressions of relative importance. Two things that we’re comparing could both be good attributes. The real question is, when you have to make a choice between two of them, what do you choose? That choice that you make, reflects your core values.

He goes ahead to contrast the core values of some programming languages with the core values we demand from systems software, like operating system kernels, file systems, microprocessors, and so on. It is a really good talk and you should [watch it](#).

It is important to think about values because they are *core to the decisions that we make*.

Unlike systems software, the values demanded from text editors or IDEs vary greatly depending on who you ask. These are much more personal tools and make room for a diverse set of desires.

The following listing enumerates values that could be attributed to development tools.

Value	Commentary
Approachability	Ease of getting started with for typical tasks, and contribution friendliness
Doing one thing well	Unix philosophy, fitting into an ecosystem
Editing efficiency	Fewer interactions, mnemonics, composable keystrokes, etc.
Extensibility	The degree to which behavior and appearance can be changed
Freedom	Embraces free software, rejects proprietary software
Integration	Cohesive core and concerted third-party functionality
Introspectability	Capable of being understood and inspected ad-hoc
Keyboard centrism	Focus on keyboard interactions
Maintainability	The degree to which it can be modified without introducing faults
Progressiveness	A measure of eagerness to make progress and leverage modern technology
Stability	Things that worked before continue to work the same way
Text centrism	Text as a universal interface
Velocity	Short and focused release cycles, aligned personpower, leveraging the community effectively

Before we go any further, I'd like to point that out if you care about any of the topics discussed ahead you will likely strongly disagree with something or the other.

That's fine! We probably just have different values.



In my view, Emacs has the following core values:

Emacs

- Extensibility
- Freedom
- Introspectability
- Keyboard centrism
- Stability
- Text centrism

We can feel the clasp of *stability* in the following—rather poetic—exchange in the Emacs development mailing list, which also provides useful historical perspectives.

Emacs is older than the operating systems people use today. (It is almost as old as the first Unix, which barely resembled the Unix of later decades.) It is much older than Linux, the kernel.

The oldest design elements were not designed for the uses we make of them today. And since we wrote those, people have developed other areas of software which don't fit Emacs very well. So there are good reasons to redesign some of them.

However, people actually use Emacs, so a greatly incompatible change in Emacs is as unthinkable as a greatly incompatible change in the New York City subway.

We have to build new lines through the maze of underground pipes and cables.

— [Richard Stallman in “Re: Discoverability \(was: Changes for 28\)” \(2020\)](#)

The following exchange reifies *freedom* and *stability* while demonstrating a disinclination to *progressiveness*. Which is neither good nor bad; it's just what it is.

If Emacs was to become a “modern” app tomorrow, an editor extended in Lisp still only has appeal for a minority of programmers, much like the Lisp language itself. Most programmers looking for easy and modern experiences will likely stick with Atom and Sublime.

Most of the push for a “modern look” comes from the desire for Emacs to play more nicely with proprietary platforms. Rather, the goal of Emacs is to support platforms like GNU/Linux. Platforms that respect your freedom, and also do not push a corporate UI/UX vision of “modernity”.

(Perhaps if we do move forward with modernization, we should think of modernization in the context of something like GNOME rather than MacOS or Windows. Surely Emacs



could be a better citizen of GNOME.)

Given that many of the people complaining about “how Emacs looks” are not submitting patches to fix the problem themselves, resources would be diverted from actual functionality to “modernity”.

By the time we do major code refactoring “modernizing” Emacs on the major proprietary platforms, what is “modern” has now once again changed, and our resources were put towards a project with a poor return on investment.

Basically, I don’t see a “modernizing” project playing out well. We will spend extensive time and energy on a moving target, and even if we succeed, our Lisp-based vision still has limited appeal. Additionally, I don’t think “modernizing” Emacs advances the cause of free software, given that there are other more popular casual libre tools for text editing that individuals can use.

— [Ahmed Khanzada in “Re: Why is emacs so square?” \(2020\)](#)

Core values are self-reinforcing. They attract like-minded people, who will then defend them.

I’m an Emacs user, and reading the Emacs mailing lists serves to remind me that my values are very different from the values held by maintainers and core contributors. I don’t value *freedom* or *stability* nearly as strongly and have an inner affinity for *progressiveness* and *velocity*.

One part of valuing progressiveness is constantly re-evaluating: is our current process or technology as good as it could be? What could be improved? How do we measure improvement? How are others solving these problems? Were there any advances in our area that we could leverage?

* * *

Now let’s talk about Vim. I see Vim as intersecting with a few of Emacs’ values, but ultimately diverging radically with its narrow focus on providing really efficient editing capabilities.

Vim

- Doing one thing well
- Editing efficiency
- Keyboard centrism
- Stability
- Text centrism



One might notice that *extensibility* is not in the list. That's intentional. Vim is certainly extensible to a degree, but it just does not compare to Emacs. Vim *has* a “plugin system”, while Emacs is the system. Your code becomes part of it the moment it's evaluated. Since I'm sticking to yes/no indicators for values I'm giving it a *no*.

Stability emanates from communications with the primary maintainer.

Vim development is slow, it's quite stable and still there are plenty of bugs to fix. Adding a new feature always means new bugs, thus hardly any new features are going to be added now. I did add a few for Vim 7.3, and that did introduce quite a few new problems. Even though several people said the patch worked fine.

— [Bram Moolenaar in “Re: Scrolling screen lines, I knew, it's impossible.” \(2011\)](#)

And of course in this famous exchange in a QA session.

How can the community ensure that the Vim project succeeds for the foreseeable future?

Keep me alive.

— [Bram Moolenaar in “10 Questions with Vim's creator” \(2014\)](#)

At the end of 2013, a few folks were trying to get new concurrency primitives merged into Vim. This would empower plugin authors to create entirely new types of functionality and by extension, make Vim better.

This is what one of them had to say about the process:

The author of Neovim (Thiago de Arruda) tried to add support for multi-threaded plugins to Vim and has been stymied.

I'm not sure how to get a patch merged into Vim. Bram Moolenaar is the only person with commit access, and he's not a fan of most changes beyond bug fixes. My co-founder and I tried to add `setTimeout` & `setInterval` to `vimsript`. Even six weeks of full-time effort and bending over backwards wasn't enough. Eventually we were just ignored.

I've contributed to a lot of open source projects, and the Vim community has been the most difficult to work with. I've been writing C for almost two decades, and the Vim codebase is the worst C I've ever seen. The project is definitely showing its age, and I'd love for something new to replace it.



– [Geoff Greer in “Neovim \(HN\)” \(2014\)](#)

While they understood that some of their values were ultimately incompatible with the values of the Vim maintainers—who prioritized *stability*—they still tried to push for a change, because they treasured the *idea of Vim*, embodied by some of its values.

It didn't happen, so a Vim fork came to life: Neovim.

The vision was grand, and is summarized in a statement of its values:

Neovim is a Vim-based text editor engineered for *extensibility* and *usability*, to encourage new applications and contributions.

neovim.io/charter

Some of their concrete plans included

- improving testing, tooling, and CI to simplify maintenance, make aggressive refactorings possible, and greatly reduce contributor friction
- decoupling the core from the UI, making it possible to embed the Vim core into browsers or IDEs (or any computer program really), also making way for more powerful and diverse GUIs
- embedding a Lua runtime and providing concurrency primitives to open the doors for smoother, more efficient, and powerful plugins
- extensive refactoring: bringing C code to modern standards (C99, leveraging new compiler features), replacing platform-specific IO code with libuv, removing support for legacy systems and compilers, including automatic formatting, and fixing static analysis warnings and errors
- creating a scriptable terminal emulator

And they delivered it.

In a very short amount of time they were able to, and I don't use this word lightly, *revolutionize* Vim. The impact can be seen in Vim development, which picked up considerably as Neovim gained ground, with features and processes ending up being reimplemented in Vim.

And they aren't stopping there. Current plans include:

- translating all Vimscript to Lua under the hood, increasing execution performance due to leveraging LuaJIT, a very, very fast runtime
- shipping a built-in LSP client

Neovim builds upon Vim, and the way I see it, holds the following core values:



Neovim

- Approachability
- Editing efficiency
- Extensibility
- Keyboard centrism
- Progressiveness
- Text centrism
- Velocity

As I see it, it also currently has a better story than Emacs on:

- **development process:** [non-mailing-list-driven-development](#), [extensive automated builds](#), a [public roadmap](#), [recurrent funding](#), frequent [stable, automated releases](#)
- **user interface:** all UI-related code for every single platform was removed from the core, replaced by a consistent [API](#) (both TUIs and GUIs use it) that made a diversity of decoupled [display implementations](#) possible (UIs are literally plugins!)
- **out of the box experience:** better defaults, built-in LSP client

[This article](#) provides interesting perspectives on mailing-list-driven-development (and conveniently aligns with my own thinking).

These items are the outcome of massive change that came about through consistent hard work from a few individuals who shared a vision and a set of values. Crucially, it included aggressively improving the *human side* of software: raising money to support development, lowering contribution friction, unblocking contributors, reconciling and combining efforts, documenting processes. In other words, the type of invaluable work non-software engineers do in technology companies. To our detriment, in open source these tasks are often neglected.

Code is the easy part of building software.

It's hard to contest that Neovim's achievement happened **because** of its *approachable* development process focused on *maintainability* and *velocity*, while in contrast, it could be argued that current progress in Emacs happens **despite** its development process.

For example, because Emacs highly values *freedom*, contributing to Emacs core (or to packages in the official repository) requires [assigning copyright](#) to the FSF. To incorporate packages into the main repository, *everyone who committed to the project* needs to have gone through that procedure. Even in the case of a very willing, actual core Emacs maintainer, of an uncontroversially valuable package used by virtually everyone, this process can take [years](#).



It also makes it [impossible](#) for some to contribute to Emacs. Check out this lively discussion about [Emacs copyright assignment](#) on Reddit for more context.

It is also not hard to find [criticism](#) coming from folks who have already and continue to give so much to the community and ecosystem.

It all comes down to core values.

* * *

Let’s now address the 800-pound gorilla in the room: VSCode.

VSCode was released just five years ago, and in this short amount of time it was able to capture [half of the world’s software developers](#).

It provides a powerful, refined, cohesive out of the box experience with great performance.

It has immense leverage by building on top of Electron, NodeJS, and Chromium, projects that receive contributions in the millions of person-hours of work, from both the open source community and heavily invested corporations.

Here’s how I see its values.

VSCode

- Approachability
- Integration
- Maintainability
- Progressiveness
- Velocity

We can now put it all together in this very uncontroversial table.

Value	Emacs	Vim	Neovim	VSCode
Approachability			✓	✓
Doing one thing well		✓		
Editing efficiency		✓	✓	
Extensibility	✓		✓	
Freedom	✓			
Integration				✓
Introspectability	✓			

Value	Emacs	Vim	Neovim	VSCode
Keyboard centrism	✓	✓	✓	
Maintainability				✓
Progressiveness			✓	✓
Stability	✓	✓		
Text centrism	✓	✓	✓	
Velocity			✓	✓

Irrespective of values, VSCode is looking more and more as an acceptable Emacs replacement.

- It is somewhat extensible and very configurable
- It can be mostly driven from a keyboard
- It has a great extension language, TypeScript (which is in my opinion superior to Emacs Lisp in terms of maintainability for non-trivial projects)
- It even has a [libre variant](#)

It also shines in areas where Emacs doesn't: if you're a programmer working on typical contemporary projects, mostly just wanting to get stuff done, things usually... *just work*. You install VSCode, open a source code file, get asked to install the extension for that particular language, and that's it. You get smart completion, static analysis, linting, advanced debugging, refactoring tools, deep integration with git, and on top of that, great performance and a cohesive user experience.

Ironically, LSP (originally developed by Microsoft for VSCode) is one of the main things bringing not only progressiveness and approachability but also integration to Emacs.

This type of experience is the selling point of [Doom](#) and [Spacemacs](#), two initiatives driven by relentless maintainers. These projects bring *approachability* and *integration* to Emacs, and are in my view, along with [LSP](#), [Magit](#), and [Org](#), the biggest reasons drawing people to Emacs nowadays. It is however clear from looking at their issue trackers just how difficult it is to provide this cohesive experience by combining parts from the ecosystem.

Since Emacs is so malleable, it is very easy for packages to interfere with one another, depend on functionality from other packages that get deprecated, changed in incompatible ways, or removed. There's currently no way for a package to depend on a specific version of another package, or for multiple versions of a single package to be loaded at the same time, for example.

With Neovim also shaping up as a worthwhile up-and-comer, this is probably the first time Emacs has actual competition in its own turf.



In “[Emacs is my “favourite Emacs package”](#)” Protesilaos Stavrou talks about the importance of Emacs, the platform. While Emacs packages can be valuable in isolation, combined, they amplify the platform that made them possible. The whole becomes greater than the sum of its parts.

It wouldn't be a stretch to say that Org represents 10% of my cognitive function. Magit really is “Git at the speed of thought”, and I have yet to see a more integrated and rich interactive shell than Eshell.

And yet, even being a very enthusiastic Emacs user, I have a hard time recommending it to folks who mostly just want to get stuff done. Some will argue that those who aren't willing to build their computing environment from scratch shouldn't be using a “power tool” like Emacs anyway. I don't see a fundamental reason for that to be the case, and believe that not having young folks trying out, using, and contributing to Emacs, is not a good thing for Emacs.

This existential threat seems to be acknowledged by maintainers.

One of the gravest problems I see for the future of Emacs development is that we slowly but steadily lose old-timers who know a lot about the Emacs internals and have lots of experience hacking them, whereas the (welcome) newcomers mostly prefer working on application-level code in Lisp. If this tendency continues, we will soon lose the ability to make deep infrastructure changes, i.e. will be unable to add new features that need non-trivial changes on the C level.

— [Eli Zaretskii in Re: \[PATCH\] Add prettify symbols to python-mode \(2015\)](#)

For better or worse, Emacs overfits to the needs and priorities of its maintainers, and contributors who overcome its barriers to entry. Being a decentralized, volunteer-based project, people will commonly scratch their own [itches](#) or work on whatever they find interesting. Which is only fair: they could be doing *literally anything else*, and yet they choose to sacrifice their time and do their best to advance Emacs according to their values. They owe no one anything and deserve gratitude.

* * *

In a 2015 keynote, while laying out an argument for why the Go programming language is open source *at all*, Russ Cox portrayed an active and intentional effort to lower barriers to entry and deliberately improve the human side of Go, so that as many people as possible used and contributed to it.

The core values of Go are incidentally made apparent through the [talk](#):

- Approachability



- Developer productivity
- Large-scale development
- Performance
- Simplicity

Go was created to make Google's developers more productive and give the company a competitive advantage by being able to build products faster and maintain them more easily. Why share it with the world?

Russ argues that the business justification for it is that **it is the only way that Go can succeed.**

A language needs large, broad communities.

A language needs lots of people writing lots of software, so that when you need a particular tool or library, there's a good chance it has already been written, by someone who knows the topic better than you, and who spent more time than you have to make it great.

A language needs lots of people reporting bugs, so that problems are identified and fixed quickly. Because of the much larger user base, the Go compilers are much more robust and spec-compliant than the Plan 9 C compilers they're loosely based on ever were.

A language needs lots of people using it for lots of different purposes, so that the language doesn't overfit to one use case and end up useless when the technology landscape changes.

A language needs lots of people who want to learn it, so that there is a market for people to write books or teach courses, or run conferences like this one.

None of this could have happened if Go had stayed within Google. Go would have suffocated inside Google, or inside any single company or closed environment.

Fundamentally, Go must be open, and Go needs you. Go can't succeed without all of you, without all the people using Go for all different kinds of projects all over the world.

— [Russ Cox in the GopherCon 2015 keynote](#)

The parallel to Emacs isn't direct, but it's clear.

Emacs evolved greatly since its inception in 1976 as a collection of macros for the programmable TECO text editor, which is itself from 1962. Take a look at this example TECO program, from Wikipedia:



```
2 <j 0aua l
3 <0aub
4 qa-qb"q xa k -l ga -1uz '
5 qbua
6 l •-z;>
7 qz;>
```

It was a different world then. Updating the display text in real-time as users typed into the keyboard was a recent innovation.

Since then, Emacs Lisp was created, Emacs forks came and went, a graphical UI was added, lexical scope was implemented, rudimentary networking and concurrency primitives were introduced, and more.

It can be argued that *stability* and incremental evolution are the reasons why Emacs survived and is still thriving. Stability though is necessarily antithetical to progressiveness. The very thing that likely made it succeed is what slows it down.

It doesn't, however, affect *progressiveness* as much as *freedom*. Because of *freedom*, when faced with a question of using technology that is

1. Non-free, but objectively better
2. Free, but objectively worse

[the latter](#) will **always** be picked. Given that most technological progress happens through the mechanisms of capitalism, [“free” alternatives](#) commonly lag behind to a large degree.

Freedom has a price.

It can be seen clearly and succinctly in this exchange:

[...] I think freedom is more important than technical progress. Proprietary software offers plenty of technical “progress”, but since I won't surrender my freedom to use it, as far as I'm concerned it is no progress at all.

If I had valued technical advances over freedom in 1984, instead of developing GNU Emacs and GCC and GDB I would have gone to work for AT&T and improved its nonfree software. What a big head start I could have got!

— [Richard Stallman in “Re: New maintainer” \(2015\)](#)

Agree with him or not, RMS has a point. The ability to inspect and change software running on our computing machines gives us control.

Apple for example seems to be growing increasingly antagonistic to the privacy of its users (and [bullish to its developers](#)). As I write this, it's been discovered that newer versions of macOS include anti-malware functionality that transmits tracking information almost e



time *any program is run*. This information is sent unencrypted via a third-party CDN, so not only this could be seen as a privacy violation but also a dangerous data breach: anyone listening on the network can [roughly](#) know which applications you use, how often you use them, when do you use them, and from where.

There are measures that can still be taken to ameliorate this situation and others, but it's ultimately outside of our control. macOS is a proprietary operating system and can easily prevent users from taking these steps in the future.

I choose to pay the price of compromising my freedom by tolerating invasions of my privacy so that I can have a computer that *mostly just works* and allows me to be productive towards achieving my life goals. We have to pick our battles, and the hills we die on depend strongly on our values.

Becoming aware of these facts is still not enough to make me switch back to [GNU/Linux](#) for my personal computing needs. At least for now...

We are increasingly finding ourselves in a world where we have to choose between extremes: do you want a computing machine that respects your privacy, or a modern, powerful one?

* * *

Back to Emacs.

Maintainers and core contributors likely use it in very different ways than the majority of users and casual contributors and therefore have very different priorities. For example, until it got stolen in 2012, [RMS used a 9-inch netbook](#) because “it could run with free software at the BIOS level” (these days he uses an 11-year-old T400s). The recent [Emacs User Survey 2020](#) might help identify prevailing usage patterns, and hopefully have an impact on the direction of Emacs.



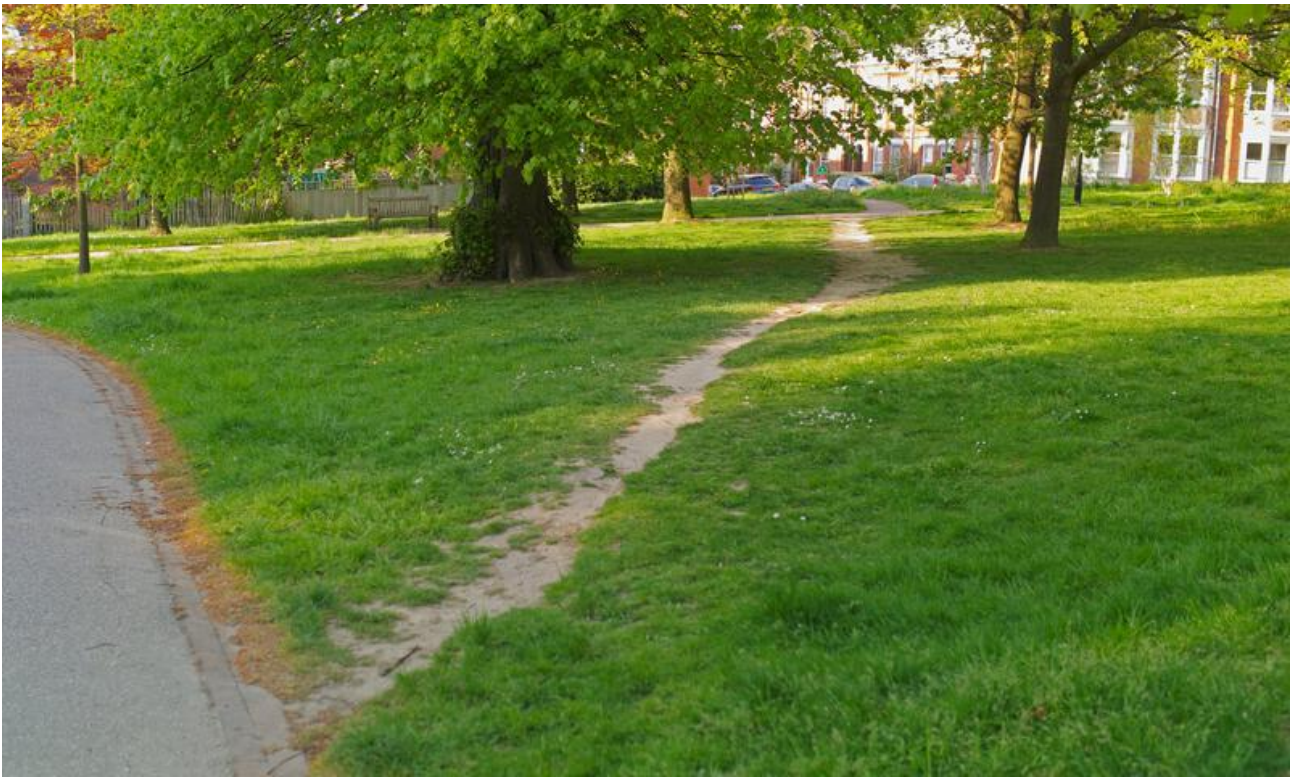


Figure 5: Desire path (Alamy)

Emacs doesn't need a *Neoemacs* as much as Vim needed Neovim. Unlike Vim, Emacs always had a rich ecosystem of active contributors, and maintainers who—to some degree—listen to user feedback. There is still tension, rooted in ideology and values, which throughout Emacs history materialized as forks: Lucid/XEmacs, Guile Emacs, Aquamacs, Mac port, Remacs.

Forking is incredibly difficult to pull off. A successful one requires not only an initial momentum and enthusiasm, but also unrelenting, sustained hard work from a group of individuals, not to mention buy-in from a critical mass of users. As someone said to me, you have to be a “special kind of crazy” to start an Emacs fork.

I hope that Emacs doesn't find itself becoming “perfectly suited for a world that no longer exists”. Still, I understand and appreciate the difficulty of the situation. People have diverging values and are [highly fallible](#). Everything requires so much effort. So is our condition.

Ultimately, building software is a complex and deeply human activity. Everything is contextual and there are rarely easy answers. Most meaningful progress happens through consensus, compromise, luck, and lots of hard work.

In the end, a lot can be understood through the lens of values.

What are yours?

Cathedrals, Bazaars, and Fusion Reactors



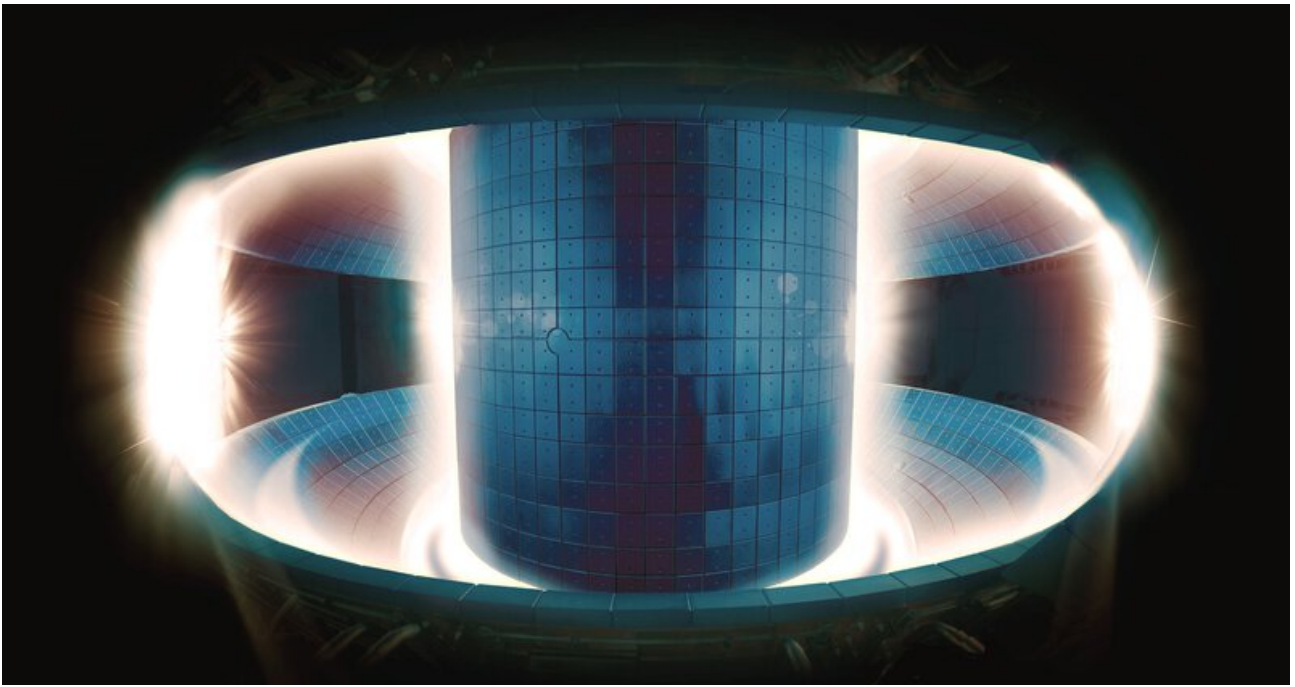


Figure 6: Inside the Korean tokamak KSTAR (NFRI)

With corporations like Microsoft, Oracle, and Google truly reinventing themselves to adapt to an open source world, and typical open source projects moving towards—oftentimes centralized—governance models, the [Cathedral-Bazaar](#) dichotomy feels increasingly less relevant.

It was met with [criticism](#) even back in the 90s.

While being an entertaining piece of history with useful takeaways, its most important achievement was arguably helping create a sense of identity for hacker culture via the revolutionary Open Source movement, and promoting the value of the Internet for software development.

In the Cathedral-Bazaar continuum, contemporary projects like Kubernetes, Chromium, and VSCode are *fusion reactors*.

They have the backing of heavily invested companies with virtually infinite capital, who are able to staff highly competent teams that not only work full-time on these projects, but also have enough personpower to maximally leverage the benefits brought by a gigantic user base.

Like with fusion power, they seem to be able to leverage a high amount of energy to generate even more.

Sometimes, their user base includes *other* organizations with endless resources: by means of its success, VSCode is getting [sizable contributions](#) from Facebook, for example.



In contrast, the vast majority of open source projects depend almost exclusively on decentralized volunteer efforts from people sacrificing time out of their busy schedules and lives to move things forward.

And yet, projects following this style of development can end up becoming backbones of modern computing:

The OpenSSL project has been around since 1998. Since the project is open source, it is an informal group comprised primarily of about a dozen members throughout the world, most of whom have day jobs, and some of whom work on a volunteer basis. Being open source, the OpenSSL project's code has always been public facing. Any person could download it and modify it or implement it in their own software.

[...]

The fascinating, mind-boggling fact here is that you have this critical piece of network infrastructure that really runs a large part of the internet, and there's basically one guy working on it full time.

— [Steve Marquess in “It’s Not A Fun Week To Work at OpenSSL, The Mostly Volunteer Project Responsible for the Heartbleed Bug”, \(2014\)](#)

Heartbleed is a symptom of an ever-existing problem: corporations profit massively by leveraging typically under-resourced open source projects while not giving back proportionally, or (most commonly) at all; either with money or people.

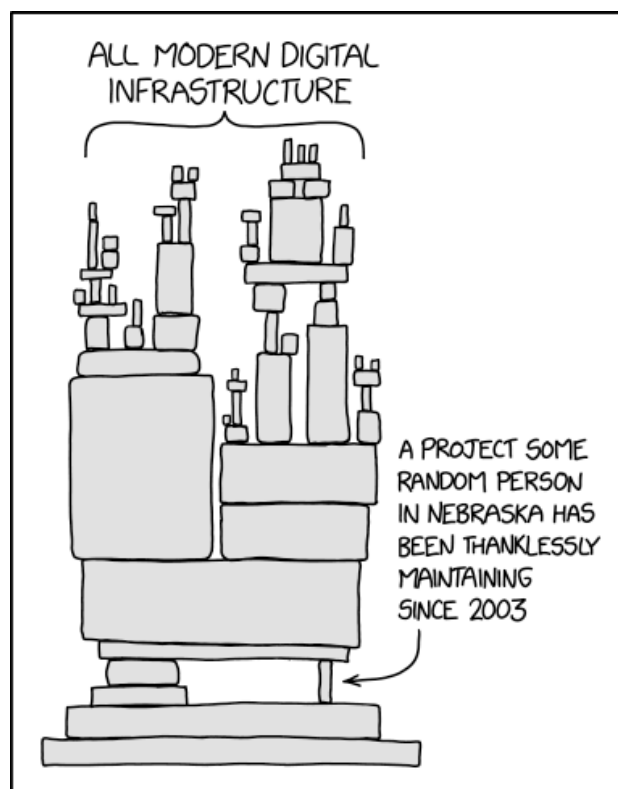


Figure 7: “Load-Bearing Internet People”



After Heartbleed the [Core Infrastructure Initiative](#) was created to support software essential to the "functioning of the Internet and other major information systems".

Less critical software like Emacs also follows this decentralized development style, and similarly, lacks resources.

So if Emacs wants to compete with these tools then it has to have seamless, context aware code completion and refactoring support, and GNU tools has to provide Emacs the necessary information to implement these features.

I agree. But to have that, the only way is to have motivated volunteers step forward and work on these features. Otherwise we will never have them.

Right now, no one is working on that, though everyone is talking. [T]he same as with weather.

— [Eli Zaretskii in “Re: IDE” \(2014\)](#)

One way to incentivize contributions is by funding developers. Some (most?) open source contributors would gladly take income to fund their work.

Long-term my big dream has always been to accumulate enough backing to be able to work full-time on open-source projects, but whether I'll achieve this dream or not is entirely up to you.

— [Bozhidar Batsov in “Patronage Revisited” \(2020\)](#)

While others feel that accepting funding would degrade their [intrinsic motivation](#) to contribute.

I can't speak for all FLOSS developers, but I can speak for myself: I don't want monetary rewarding from users. Mainly for the reason, that I don't want to change the relationship with my users. Currently it is mostly a team attitude, we're working together to solve the problem. And there is also no legal obligation for me to work on something I don't like.

If I accepted contributions I think many users would get a “but I paid for that, so do what I want” attitude. I definitely don't want that. I do FLOSS in my free time to do something that matters, and for my personal fulfillment, not for money.

It will also get harder to do the right thing (in contrary to doing what the users want) since the users can stop the payments.



So, no payments for me, thanks.

— [cjk101010](#) in “Sustainable Emacs development - some thoughts and analysis” (2017)

Which is fair: with compensation comes responsibility, timelines, expectations, and things can get complicated. From the point of view of the project, there doesn't seem to be a conflict: capture funding to enable those who need (or want) it, while still empowering those who don't to contribute on their own terms.

Not everyone is in a position to spend unpaid time on open source, especially consistently. “Free” time [isn't free](#)—it costs *life*. By funding work, a project might get to see contributions from talented folks passionate about it who wouldn't be able to volunteer their time.

And sometimes, the reason why your PR isn't getting immediate attention is that the maintainer is busy literally [fighting a revolution](#).

For Emacs specifically, one problem is that there's no clear way of funding “Emacs”. Sending money to the FSF doesn't guarantee that it will fund Emacs development. *Even if there was a way of “funding Emacs”, the Emacs community, like many things in life, seems to be roughly divided in two sides: those who prioritize [freedom](#), and those who prioritize *progress*. So one would potentially want to fund one side or the other, depending on their values.*

In the excellent “[Working in Public](#): The Making and Maintenance of Open Source Software”, Nadia Eghbal calls attention to a shift in how individuals support open source. Similarly to platforms like Twitch, more and more people are funding creators directly instead of projects, as a way of “incentivizing the ongoing creation of creative work” from developers who produce things that are in their interests.

Maintainers of popular Emacs packages, for example, have their own [separate streams of patronage](#), which receive varying levels of support depending on their popularity and the value added by what they create.

For efforts that involve multiple people, like maintaining and evolving Emacs itself, services like [Open Collective](#) could be of great assistance by helping on three fronts:

1. providing a legal banking entity
2. recurrently collecting funds from individuals and companies
3. distributing funds to contributors

Funds can be transparently collected, and dispersed for specific contract work, infrastructure costs, and even developer salaries. Take for example the [Babel](#) project, which draws in enough recurrent income to finance multiple contractors and a full-time developer earning a San Francisco salary. An Emacs Open Collective could not only be an answer to



“how can I fund Emacs development?” but also a way to financially support developers working on it.

The Clojure community seems to be doing an excellent job at not only [funding](#) efforts that are making the whole Clojure ecosystem better but also at surveying and [responding](#) accordingly to user feedback.

[GitHub Sponsors](#) is another great example of developer empowerment. With it, not only does GitHub equip people to:

- be more productive by providing great code hosting, bug tracking, wiki, code reviewing and merging, project management, continuous integration, documentation, artifact hosting, etc.
- have a broader impact by giving projects more visibility and standardized workflows that are familiar to others already on the platform

It also makes it possible for its [40 million users](#) to frictionlessly fund work on open source, and for a large number of maintainers to be conveniently compensated for their labor. I just started sponsoring someone with literally two clicks!

The power of platforms can't be understated.

Whether you like GitHub or not, it's undeniable that it has, and continues to revolutionize Open Source, simply by providing a significantly better and unified experience for all aspects of building software.

When there are people making over [100k/year](#) on GitHub Sponsors, you better have a great reason to **not** try to [take advantage](#) of it.

In the case of Emacs, the reason is *freedom*.

I wouldn't mind if Emacs development moved to GitHub, but [I don't think it's ever going to happen](#). Maybe for good reason: GitHub is backed by a for-profit corporation and is far from perfect, both in [moral](#) and [technical](#) terms. It might be a great tool today, but being a proprietary platform, its users are at their complete mercy.

I should point out that from my perspective GitHub has been for the most part a force for [good](#).

It would be great if main development at least moved from a mailing-list-driven process to a modern *forge* style of contribution. It [seems that it might](#), but whether or not it will is still [unclear](#).

In the same way that corporations extract value out of open source, open source projects should as much as possible leverage “energy” generated by corporations. In this new open source world, companies have their workforce contributing millions of person-hours to



projects that benefit everyone. LLVM equips people to build programming languages. LSP gives people potent software development capabilities. Rails empowers people to build powerful web applications.

More than 3,000 people have committed man-decades, maybe even man-centuries, of work for free. Buying all that effort at market rates would have been hundreds of millions of dollars. Who would have been able to afford funding that?

That's a monumental achievement of humanity! Thousands, collaborating for a decade, to produce an astoundingly accomplished framework and ecosystem available to anyone at the cost of zero. Take a second to ponder the magnitude of that success. Not just for Rails, of course, but for many other, and larger, open source projects out there with an even longer lineage and success.

— [David Heinemeier Hansson in “The perils of mixing open source and money” \(2013\)](#)

In many [cases](#), the ideology ingrained in Emacs prevents it from leveraging value generated by efforts not totally compatible with the goals of the Free Software movement. Still, there are many non-conflicting opportunities for improvement.

Maybe Emacs doesn't need to be a fusion reactor. I only hope it continues to generate energy for many years to come.

It just needs volunteers to keep the fire going.



From *catching up* to *getting ahead*

I started using [Emacs](#) almost exactly four years ago, after almost a decade of [Vim](#). I made the switch cold turkey. I vividly remember being *extremely* frustrated by unbearable slowness while editing a Clojure file at work. With no sane way of debugging it, just moving the cursor up and down would result in so much lag that I had to step away from the computer to breathe for a while. When I came back I quit Vim (I knew how at that point), opened Emacs, and started building my configuration.

More recently, I've been very put off by the performance and stability (or lack thereof) of building large scale software via Tramp. This has been sufficient to have me looking out again. On a whim, I installed VSCode for the first time and tried its “remote development” capabilities and holy smokes are they good. Getting up and running was trivial and the performance was great. Saving files was snappy and LSP worked out of the box. What a different experience from my carefully-put-together, half-working, slow Emacs setup.

My common denominator for rage-quitting software seems to be consistent: bad performance.

There has recently been more discussion than usual regarding “modernizing” Emacs, by making keybindings more consistent with other applications and using more attractive color schemes and visuals, with the end goal of attracting more users and by extension more contributors.

In my view improving these aspects of user experience wouldn't hurt. The way I see it, though, is that for Emacs to attract more users it needs to be objectively better than the alternatives. And the way to do it is for Emacs to become *even more* like Emacs.

I see Emacs as being fundamentally two things: a programmable runtime, and a beacon for free software. I'm talking more about the former.

It needs to be a *more* robust, *more* efficient, and *more* integrated platform with a *more* powerful extension language, to empower its users to build their own environment.

Getting LSP integrated *pervasively* in Emacs in a way that it reliably *just works* and performs well out of the box, would go a long way towards making Emacs more attractive not just to new users, but to existing ones too. Imagine an experience similar to VSCode's:

1. Open Emacs for the first time
2. Open a source code file
3. Emacs asks if you want it to configure itself for the programming language of that source file
4. Saying “yes” automatically sets up Emacs to have a modern programming environment for that programming language with smart code completion, navigation, and



refactoring, rich hover information, highlighting, automatic formatting, snippets, etc. Maybe even open a side window with a buffer with a short “getting started” tutorial showing the available keybindings.

Providing good [out of the box support for LSP](#) is one of the current priorities in the Neovim project.

Given enough users, opinionated community-built Emacs “distributions” like Spacemacs, Doom, and Prelude will do the job of making it easier for newcomers to get started with typical contemporary tasks: building software with popular programming languages, writing documents, managing machines, etc.

Building and maintaining these “distributions” also becomes much easier given a more robust, more efficient, and more integrated platform with a more powerful extension language.

Having a [wizard](#) showing up in new Emacs installations might be a great low-hanging fruit way of making Emacs more accessible. Assuming buy-in from core maintainers, the wizard could even directly reference popular Emacs “distributions” like the ones mentioned above, so that new users can kickstart their lives in Emacs.

The way to attract contributors can also be *stated* simply: directly improve the contribution process.

Easier said than done.

[Many](#) have created their Emacs [wishlists](#). This is mine:

1. Improved single-core efficiency
2. Improved display efficiency and rendering engine
3. Leveraging preemptive parallelism
4. Emacs Lisp improvements
5. Enhanced stability
6. Dealing with non-text
7. Improved contribution and development process

Let’s get into it.

1. Improved single-core efficiency

There are two dimensions to this:

- garbage collection efficiency
- code execution efficiency



For the past one and a half years, [Andrea Corallo](#), a compiler engineer, has been [working](#) on adding native compilation capabilities to the Emacs Lisp interpreter. His work is available in a branch in the official Emacs repository. Folks have been trying it out, and according to the reports I'm hearing, the results are staggeringly positive. I am very excited about Andrea's work, which seems to bring enough improvement to the "code execution speed" side of the equation to make it a non-issue for now.

Andrea's work will also allow for more of Emacs to be implemented in Emacs Lisp itself (instead of C), which is what most contributors are used to. This is a great win for maintainability and extensibility: incrementally having more and more of Emacs be implemented in the language with which it's extended.

The garbage collector is still in much need of improvement. Many resort to hacks to ameliorate frequent and sometimes long pauses that seem to be unavoidable while working on large git repositories, fast-scrolling font-locked Eshell buffers, displaying dynamically updating child frames, navigating big Org files, and many other tasks.

Also, try this out: `(setq garbage-collection-messages t)`

2. Improved display efficiency and rendering engine

The display implementation in Emacs core is... less than ideal.

GNU Emacs is an old-school C program emulating a 1980s Symbolics Lisp Machine emulating an old-fashioned Motif-style Xt toolkit emulating a 1970s text terminal emulating a 1960s teletype. Compiling Emacs is a challenge. Adding modern rendering features to the redisplay engine is a miracle.

— [Daniel Colascione in "Buttery Smooth Emacs" \(2016\)](#)

It would be great if Emacs did like Neovim and decoupled the editor runtime from the display engine. This would make it possible for the community to build powerful [GUIs](#) without having to change [Emacs core](#), possibly [using technology not fully sanctioned](#) by core maintainers.

Take a look at the screenshots of these Neovim GUIs:

- [neovide](#)
- [veonim](#)

They're powerful, look great, perform well, and more importantly, are based on industry standard, cross-platform graphics APIs (Vulkan and WebGL respectively) that get lots of personpower contributions from companies and individuals alike.

The [Onivim](#) and [Xi](#) text editors could also be sources of inspiration:



- Separating the core runtime from the user interface
- [Ropes](#) for faster incremental changes and parallelization of text operations
- Game-like drawing pipelines

Check out this talk by Raph Levien: [Xi: an editor for the next 20 years](#).

3. Leveraging preemptive parallelism

Emacs does not support parallel code execution via multi-core processing. Code execution happening on any buffer will freeze the whole program, preventing not only user interaction but other cooperative threads of execution from making progress as well.

Adding parallelism to Emacs in a way that automatically makes existing code run in parallel is about as close to impossible as it can get. What would be more feasible is including new primitives for parallel execution that new code could leverage, to build more powerful extensions to Emacs.

There are stealth efforts currently in the works for doing just that.

There also seems to be advances in the area of immutable data structures that could be leveraged by the Emacs core, as seen in “[Persistence for the Masses: RRB-Vectors in a Systems Language](#)”. Persistent data structures would make building thread-safe parallel code much easier.

4. Enhanced stability

It is very easy to either freeze Emacs or cause it to run very slowly. Multiple times a day I have to hit `C-g` incessantly to bring it back from being frozen. When that fails, I am sometimes able to get it back with `pkill -SIGUSR2 Emacs`. At least once per week I have to `pkill -9 Emacs` because it turned completely unresponsive. I suspect doing more work outside of the main thread might help with this?

There are many hacks to ameliorate issues caused by long lines, but they’re still fundamentally there. Advancements in the “display efficiency and rendering engine” effort would help with this too.

I recently tried a package that displays pretty icons on completion prompts, and noticed that it made scrolling through candidates really slow. Profiling showed that the package was creating thousands of [timers](#), which were somehow causing the issue. There are lots of cases like this, where folks attempt to create something nice, but inevitably have to resort to [hacks](#) to either achieve acceptable performance, or to be able to implement the thing at all. Having a more robust/efficient/integrated core with a more powerful extension language would help here.



Impressive efforts from folks like Lars Ingebrigtsen who routinely comes in and [obliterates 10% of all reported Emacs bugs](#) also have a sizable impact. We users should follow the lead and do a better job not only creating good bug reports but also dipping in our toes and helping out: fixing bugs, writing tests, and documentation.

Yuan Fu recently wrote a [nice guide](#) for contributing to Emacs.

5. Emacs Lisp improvements

Emacs Lisp is a much better language than Vimscript. Unfortunately, that’s not saying much. It’s not a particularly good Lisp and has lots of room for improvement.

For example, if you want to use a map, you have three choices: you can use alists, plists or hash maps. There are no namespaces in Emacs Lisp, so for each of the three data types you get a bunch of functions with weird names. For alists get is `assoc` and set is `add-to-list` , for hash maps get is `gethash` and set is `puthash` , for plists get is `plist-get` and set is `plist-put`. For each of those types it is easy to find basic use cases that are not covered by the standard library, plus it is easy to run into performance pitfalls, so you end up rewriting everything several times to get something working. The experience is the same across the board, when working with files, working with strings, running external processes etc. There are 3rd party libraries for all those things now because using the builtins is so painful.

– [stiff in “Evolution of Emacs Lisp \[pdf\]” \(2018\)](#)

Emacs Lisp APIs evolved incrementally while maintaining backwards compatibility over a long period of time. This is good: code written more than a decade ago still runs. These increments came about via decentralized volunteer efforts, and it shows: there are many inconsistencies and conflicts between and within libraries, which feel like having evolved without an overarching design.

Programmers used to languages that did go through [careful, deliberate design](#) brought some of it to Emacs Lisp:

Package	For working with
a.el	alists, hash tables, and vectors
dash.el	lists
f.el	files
ht.el	hash tables
map.el	alists, hash tables, and arrays
s.el	strings



Package	For working with
seq.el	sequences

It would be great to have more and more of these influencing and being incorporated to the Emacs Lisp standard library and made to be very performant. `map.el` and `seq.el` seem to already be in thanks to Nicolas Petton!

Assuming a multi-core future for Emacs, it will also be critical to have good ergonomics for writing concurrent code. It should be easy to do the right thing (writing thread-safe code), and hard to do the wrong thing. I believe Clojure can also be a [source of inspiration](#).

How easy it is to just say these things! Easy to do the right thing, hard to do the wrong thing!

Other than that, a great module system, possibly one that allows different versions of libraries to coexist, would also be a great addition. Andrea Corallo seems to be trying out some [new ideas](#) in this space ([discussion](#)).

6. Dealing with non-text

It is currently possible to browse the web in Emacs in an embedded fully-featured WebKit widget. We need to go further—I want to have the same experience of the likes of [Nyxt](#) and [vimperator](#), integrated to Emacs:

- switch between tabs with fuzzy completion (ivy, helm, etc.)
- navigate via link hinting
- Isearch web pages
- easily copy text content from web pages, paste it elsewhere
- create macros to repeat actions on web pages

I can kinda do some of these things right now with the existing WebKit widget along with some [clever hacks](#). There are other more [adventurous hacks](#) which work *around* Emacs to create a full graphical interface. It would be great if this type of functionality was deeply integrated to Emacs. It's a difficult thing to do because of the existing display engine implementation and Emacs Lisp limitations.

I believe doing like Neovim (and others) and separating the [core](#) from [display](#) would help here. But, it would likely bring its [own problems](#). There are unfortunately no silver bullets.

Less importantly but still desirable: email. Even though I write most of my email messages in Emacs, I read them mostly outside of it. I prefer to exchange plain text email, but sometimes I receive HTML email. When I do, I'd prefer to visualize it as the author intended. This is currently technically possible, but suffers from the same challenges as web browsing.

7. Improved contribution and development process



Contributing to Emacs core and packages in the official repository requires assigning copyright to the FSF. Employed software developers need to get paperwork signed by their employers' legal departments, a process that takes many days. Copyright assignment is likely not going away—can it be made more convenient?

It would be great to move to a forge style of contribution. It is honestly incredible to me how people keep track of patches flying around in email threads. Unless something like sourcehut or Patchwork is being used there's no automated CI making sure individual patches and overall contributions are in a good state. Hopefully the Emacs [GitLab instance](#) starts being more actively used and becomes the official way to contribute.

Copyright assignment and mailing-list driven development are definitely off-putting to folks who just want to contribute, and aren't looking forward to having to sign paperwork or learn a special way to contribute to every project they work with. The GitHub generation of open source developers are used to a standardized, powerful and convenient platform—anything other than that just feels not worth it.

Emacs will likely always have a niche of users, but it could grow to not have developers. Having large parts of the core implementation be in C makes it not very approachable to anyone outside the handful of contributors who do feel confident to change it.

It probably makes sense to continuously look for functionality implemented in the C core that could be replaced with focused libraries, like Neovim did by replacing almost all of their hacky, platform-specific code with [libuv](#).

Also, would it make sense to start an Emacs [Open Collective](#) to fund work on Emacs?

Last “small” thing

Improve Tramp performance to match the experience of using terminal Emacs via SSH, or VSCode's [Remote Development](#).

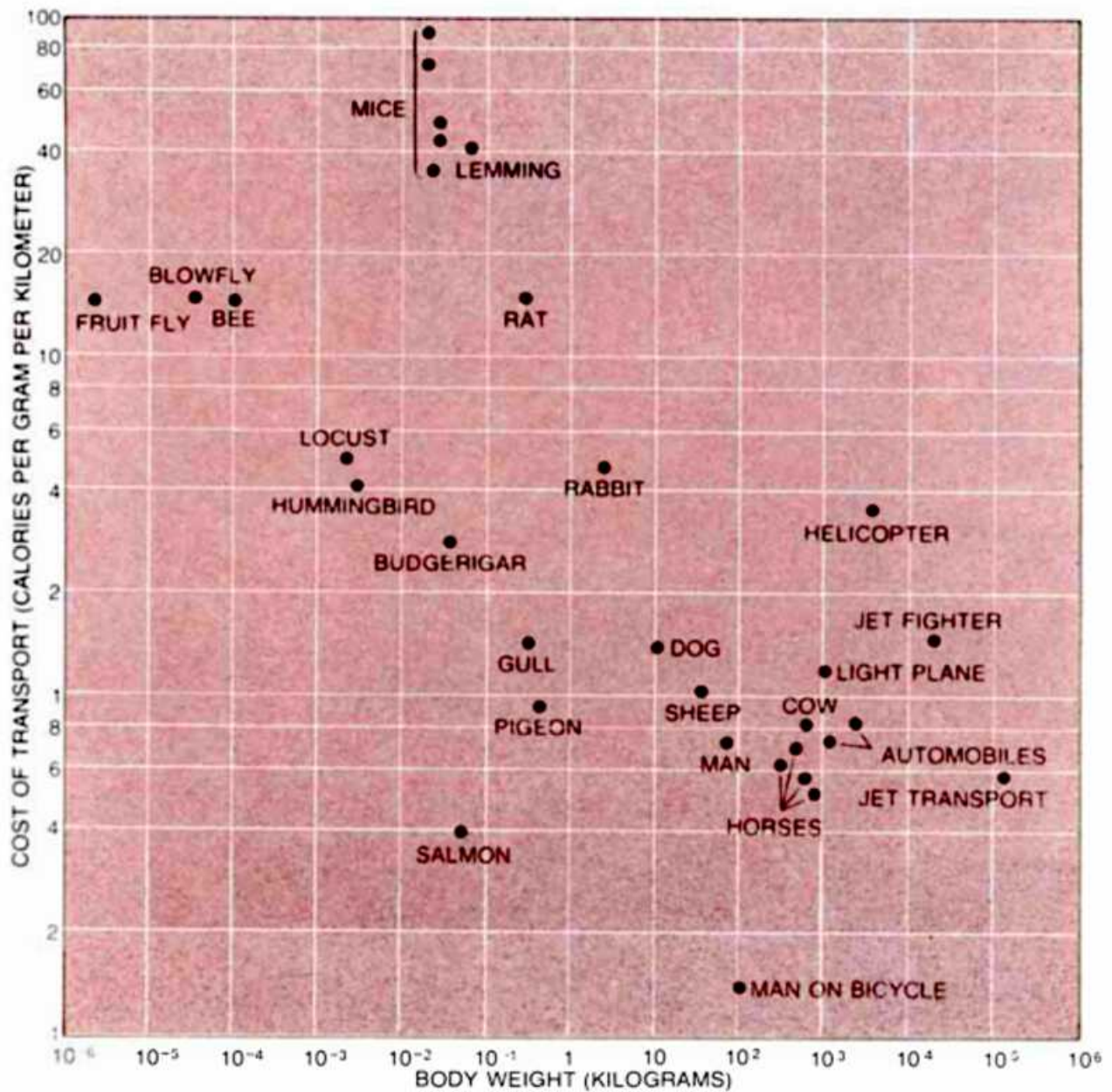
* * *

Talking is easy. Accomplishing any of these would require lots of work. It may not seem like it but text editors are a [hard problem](#). And *people*, an even harder one.

I wonder if Emacs will stick around long enough and grow the necessary functionality for us to someday run `M-x neuralink-mode` and evaluate Lisp in the brain?

The Why of technology





I think one of the things that really separates us from the high primates is that we're tool builders. I read a study that measured the efficiency of locomotion for various species on the planet. The condor used the least energy to move a kilometer. Humans came in with a rather unimpressive showing about a third of the way down the list. It was not too proud a showing for the crown of creation. So, that didn't look so good.

But then, somebody at Scientific American had the insight to test the efficiency of locomotion for a man on a bicycle. And, a man on a bicycle, a human on a bicycle, blew the condor away, completely off the top of the charts.

And that's what a computer is to me. What a computer is to me is it's the most remarkable tool that we've ever come up with.

It's the equivalent of a bicycle for our minds.

— [Steve Jobs \(1980\)](#)



* * *

No one knows when or how we, the human species, started talking to each other. It is likely a natural progression from gesturing, but we can only speculate about it.

Language allowed us to break out of our brains and reveal the inner workings of our consciousness to others.

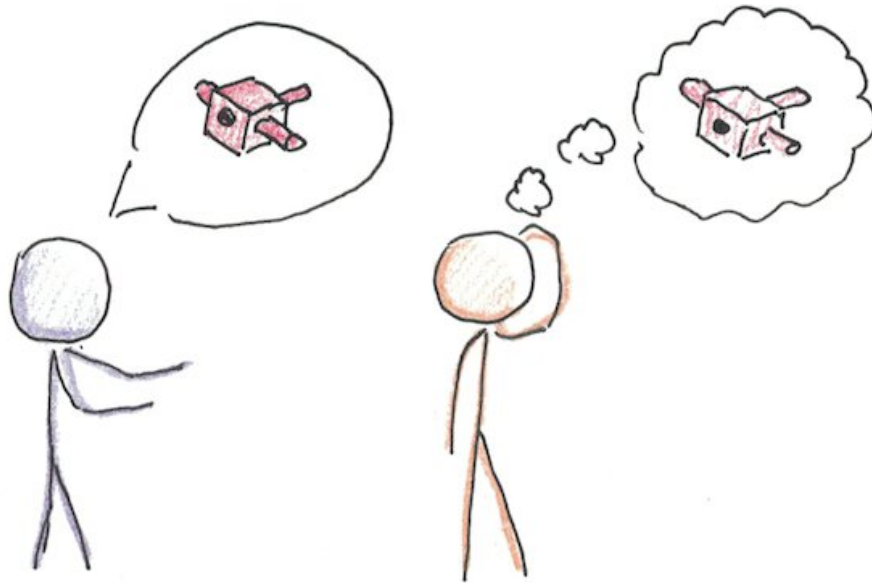


Figure 10: Scott H. Young

Language is the vessel that carried us from the stone age through the agricultural revolution, the development of written language, the scientific and industrial revolutions, and now, the digital age.

Writing allowed us to *offload* memories to the physical world—outside of our brains. Through our collective and external memories, each generation has a head start on the previous one. Little by little, standing on the shoulders of taller and taller giants, we accumulate knowledge about ourselves and everything around us.

We've been for long using tools to help us think: notebooks help us calculate formulas, reason geometrically and preserve our ideas. With computers, our *thinking* is now occurring outside of our brains.

Computers are extensions of our minds in that they allow us to store, process, and retrieve information from them. With the advent of the internet we now have immediate access to not only almost all of the information ever produced by humankind but also to reproducible *thinking* encoded into these machines: algorithms.



Our brain is still a much more impressive device than any of today's computers. Computers learn [mostly](#) by finding patterns in massive quantities of examples given by us. Teaching a young kid about cars—how to recognize one, what they are, what their purpose is, and how they're related to other things—requires little supervision. Noam Chomsky talks about it in [this interview](#).

Each of these processes—storing, processing and retrieving information—have concrete effects on the physical world: if I'm in Munich, saying “show route to Hamburg” to my phone will immediately show me the distance, ETAs and paths for different types of transport to reach my destination. Not only do I now suddenly know how to navigate across the country to reach another city, I'm also able to follow through the exact path via GPS—a sixth sense giving me perfect geolocation!

These *things* that we created—computers, and the internet—are literally [rewiring our brains](#), right now, shaping how we think, and engage in social relationships, changing not only our individual selves but the societies we live in.

They started as mechanical machines that filled entire laboratories, turned into beige boxes in our homes and places of work, and are now sleek slabs of plastic, metal and glass in everyone's pockets. Step by step they get closer to our bodies, their interfaces more intuitive and natural.

The way we communicate with them is changing: before, we could only interact with them by speaking their language. We have now taught them ours. The torch of progress blazes on: it's a matter of *time* until they're connected directly with our brains—which is equally terrifying and awe-inspiring.

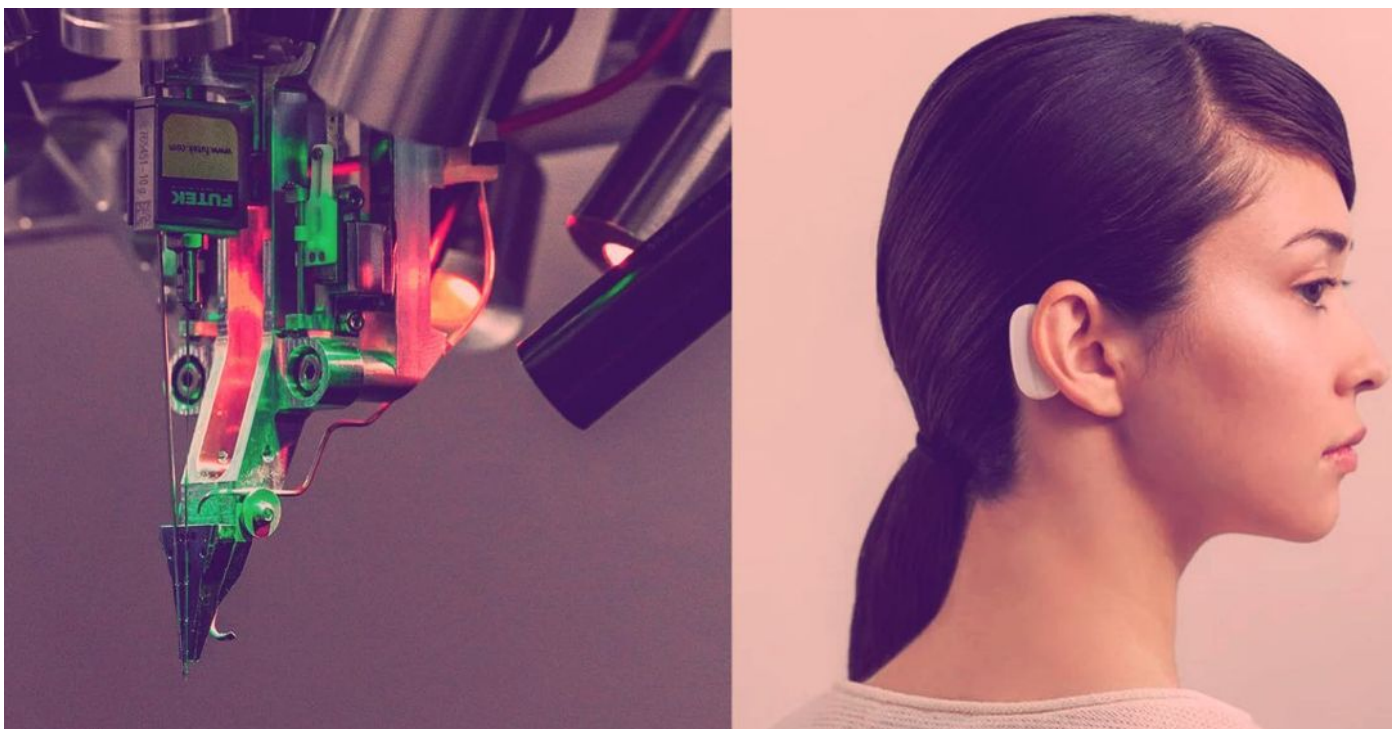
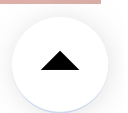


Figure 11: [Neuralink](#)



Brain-computer interfaces present a monumental scientific and engineering challenge, and brain-to-brain, a whole other category of difficulty.

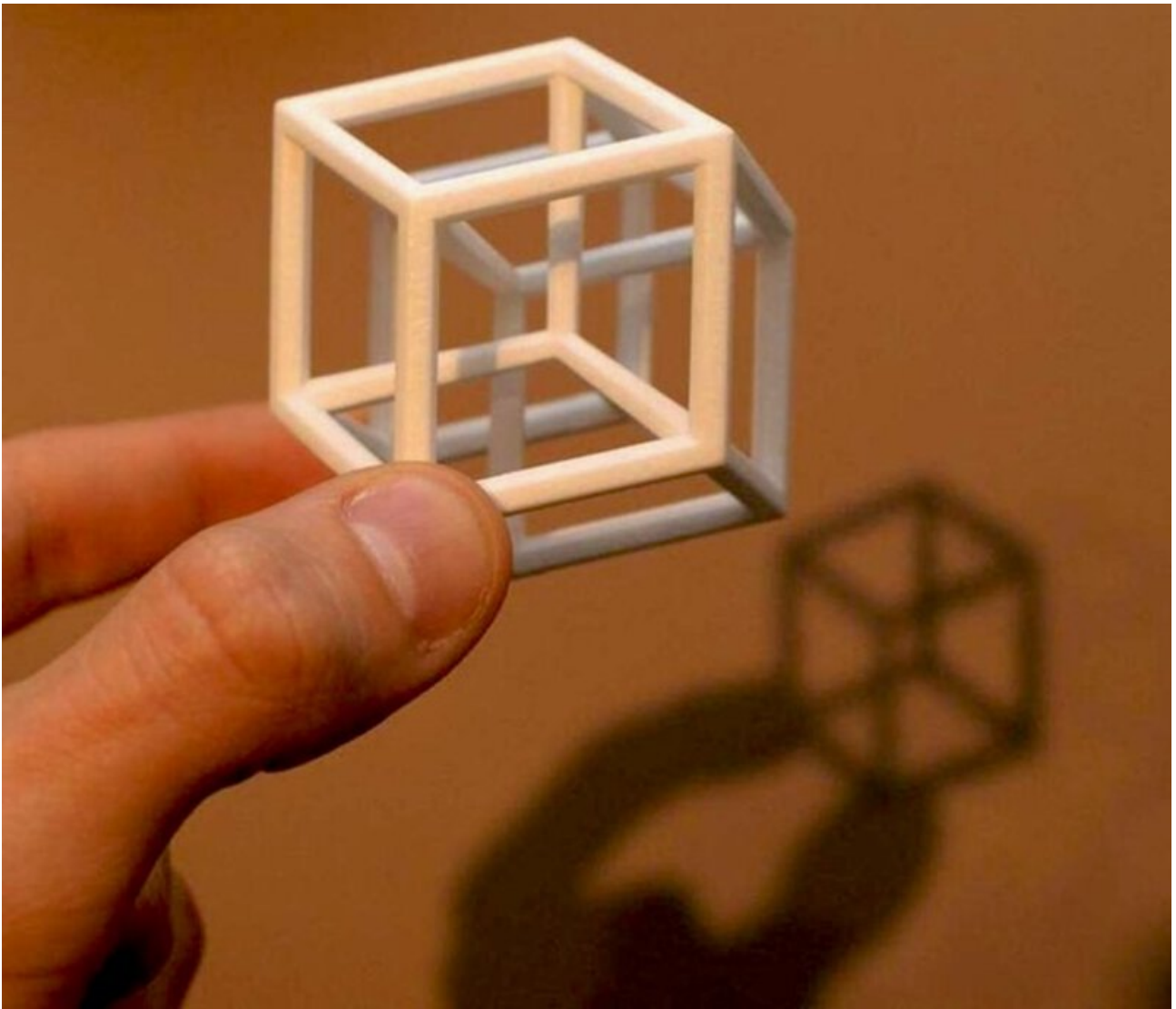
First, we have no idea how information is encoded in the brain. That needs to be understood. Second, even assuming we're able to take a perfect snapshot of a piece of information in someone's brain—for example, how a particular movie scene makes them feel—we still need to be able to encode it in a way that includes the full context of their subjective experiences. Maybe the scene evokes unique memories of their childhood or is somehow entangled with the smell of a particular cinema's leather seats. Third, we need to figure out how to safely write this perfect snapshot into someone else's brain in a way that can be perceived identically.

Which is to say, it's a difficult problem. But a worthwhile one: imagine having the capability to suddenly become aware of answers for questions you just thought about. To expertly control [truly integrated](#) prosthetics giving you superhuman abilities. To give movement to the paralyzed, sound to the deaf, and sight to the blind.

What would be the impacts on society if we were able to communicate an order of magnitude more effectively? What if *everyone* was equipped with the same undisputed basic knowledge of history and science?

There are internal thoughts that we can attempt to describe with a thousand words, but ultimately fail to capture in a way that's precise, much less comprehensible by someone else. Words and sentences are an incomplete representation of our internal thoughts. In the same way that 3D objects cast 2D shadows ([and 4D, 3D](#)) communicating through language doesn't carry all of our cultural and developmental context—transmitting all of that along with every phrase would be impractical. Language is in this sense, lossily compressed thought.





Inert strings of words of ink and paper take a life of their own inside our heads. It's why the exact same information can be interpreted completely differently by different people.

Before language, fire and cooking technology allowed us to reallocate energy usage from the digestive system to the brain by outsourcing digestion to outside of our bodies, making macronutrients more efficiently absorbable. Almost all of a cooked meal is metabolized by the body, whereas raw foods yield less than half of their nutrients.

Cooking is an extension of our digestive system, and enabled us to develop large, calorie-hungry brains. It also gave us time to think: our primate cousins spend half of their days chewing raw food to consume enough calories to stay alive.

Brains can be seen as *survival machines*, locked inside dark skulls, constantly building a model of the outside world by predicting and learning through senses and memory. The biological human brain evolved to have the necessary sophistication to not only expertly navigate and understand the brute physical reality but also to construct *social* reality. Democracy, religion, money: all made up by us, for us.

We remember the past so that we can predict the future, and by doing so, we thrive.



We create technology, which functions as a non-biological extra layer to our brains and bodies, augmenting, complementing, and sometimes replacing our natural capabilities.

The wheel... is an extension of the foot.
The book... is an extension of the eye...
Clothing, an extension of the skin...
Electric circuitry, an extension of the central nervous system.

— [Understanding Media: The Extensions of Man \(1964\)](#)

Relatively speaking, we are done evolving *biologically*. Further adaptations and enhancements to our bodies and minds will come through technology.

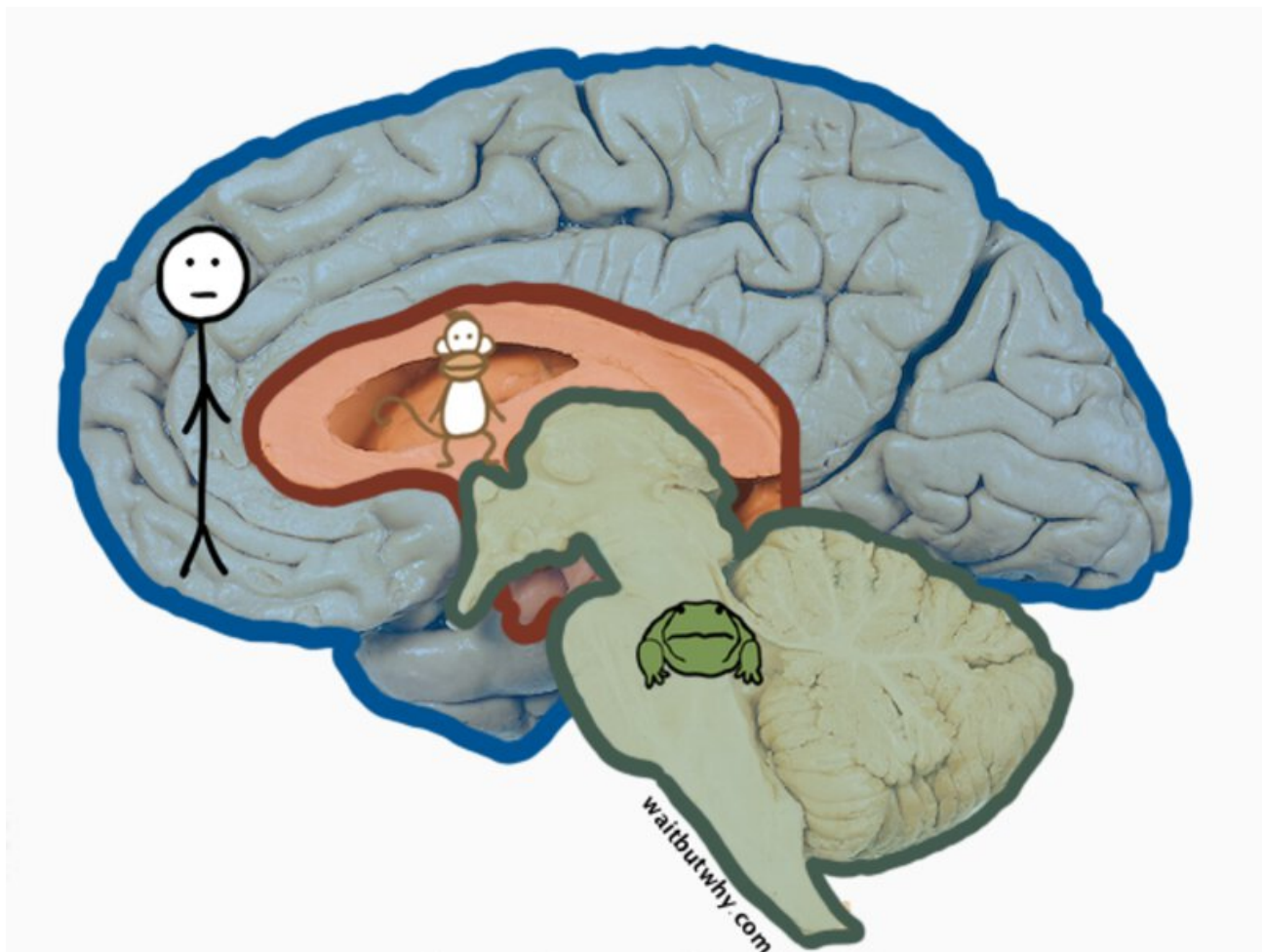


Figure 13: Check out “[Neuralink and the Brain’s Magical Future](#)” for a very entertaining primer on the brain.

To be human is to have the ability to change the world around us. The shift from hunting and gathering to farming allowed us to spend less energy to acquire food while giving us a predictable calorie supply.

The resulting food surplus made it possible for populations to settle down and grow quickly while supporting people not being directly involved in the production of food—before agriculture that was everyone’s job. For one, it allowed some to specialize and focus on



developing better farming tools and more resistant crops, starting a vicious cycle of improvement and consumption that continues until today.

The transition from active foraging to a more sedentary lifestyle resulted in worse health for the general population. The average farmer worked harder than the average forager and got a worse diet in return. Our teeth, bones and joints became more fragile, and we became afflicted by novel diseases coming from newly domesticated animals, carriers of pathogens that incubated in our new densely populated cities.

Owning land suddenly became really important. Agriculture and the concept of private property reinforced each other and grew together, allowing us to create value and secure the fruits of our labor. It also created the circumstances for slavery to arise, and wars to be waged.

The groups of people growing the first crops could not have anticipated all of the collateral effects of their breakthrough. They just wanted more food.

If the past has taught us anything is that we have to be mindful of the consequences of our progress. In an increasingly connected world, change is often [nonlinear](#) and unpredictable. Cars didn't just replace horses—they forever changed the entire outlook of every city. Did Tim Berners-Lee anticipate his invention adding to forces pulling whole countries apart?

Our progress will continue to bring us previously unimaginable challenges. Against an unknowable future, it doesn't hurt to keep improving our capabilities to adapt and, more difficultly, to cooperate—especially at scale.





Figure 14: “Humanity” by Pawel Kuczynski

Computers are getting pretty good at driving cars—even in the most difficult situations—and can already instantly diagnose some diseases better than human doctors. Technology has a way to [reveal](#) the potential of our environment, and ourselves. We have to be careful not to look at what surrounds us as mere raw materials to be consumed for the purposes we conceive—sooner or later we’ll start calling humans *resources* too...

It serves us well to leverage technology to give us time. Time to create and enjoy art, follow the trail of our curiosities and passions, be fully present with loved ones, or even just appreciate the freedom to idle and ponder about the inconsequential—the stuff that seems to make us, *us*.

We are born with incomplete brains that get imbued with language and the accumulated collective knowledge of our previous generation. Knowledge, [roughly](#) defined as a justified *true belief*, doesn’t fit the bill of much of the waves of man-made information hitting the

shores of our eyes and ears these days. Acquiring it requires many things, and passively consuming content curated by profit-maximizing algorithms is not one of them.

We attempt to transfer our gathered knowledge to machines and we specify the rules for their learning, and by doing that we're inherently encoding our own biases and limitations in algorithms that will be making life-altering choices. Should your out-of-control self-driving car [automatically swerve](#) to avoid running over kids on the street, and by doing so put your own life at considerable risk? Should you be able to opt-out of this behavior with a checkbox?

We need to be careful about what we teach machines, and prevent them from making the same mistakes we do, because they will do it orders of magnitude more efficiently and at scale. Human judgment is both fallible and (still) indispensable, especially when the stakes are higher.

Our learning machines already exhibit emergent behaviors that go beyond human understanding and could be interpreted as *creativity*, like AlphaGo's [37th move](#) on the second match against Lee Sedol in 2016.

To be human is to have the ability to change oneself. Through open source and hardware hacking, people with type 1 diabetes—who need to continuously measure and manipulate their glucose and insulin levels to *stay alive*—took it upon themselves to build an [artificial pancreas](#) and hook it up to their own bodies. The technology they created not only removed an enormous cognitive burden from their lives but also decreased the likelihood of physical complications and increased their lifespans.

What *wouldn't* you give to free up a large part of your brain processing power and at the same time considerably improve *all* of your health indicators?

Empowered by knowledge and technology, they didn't have to wait for the world around them to change—they went ahead and *changed it themselves*. And in the process, they changed their lives.

Arunachalam Muruganantham, who grew up in poverty and dropped out of school at 14 to support his single mother, also didn't wait for the world around him to change. Going against conservative rural India—who treats sex education as taboo—he provided underprivileged women with [affordable sanitary pads](#) by creating a set of pad-producing machines. Poor menstrual hygiene cause women to miss school, risk infections, and die from cervical cancer. The industry around his invention provides women with income, gives them dignity, and saves their lives.

Technology and the tools we create drastically accelerate our progress. Matt Taylor compellingly puts it in perspective in "[Humanity 2.0](#)". In it he presents the chart below,



which seems to show life's steady and even progress from unicellular organisms to us, human beings, building [artificial suns](#), taking pictures of [black holes](#), and unlocking the [mysteries of life itself](#).

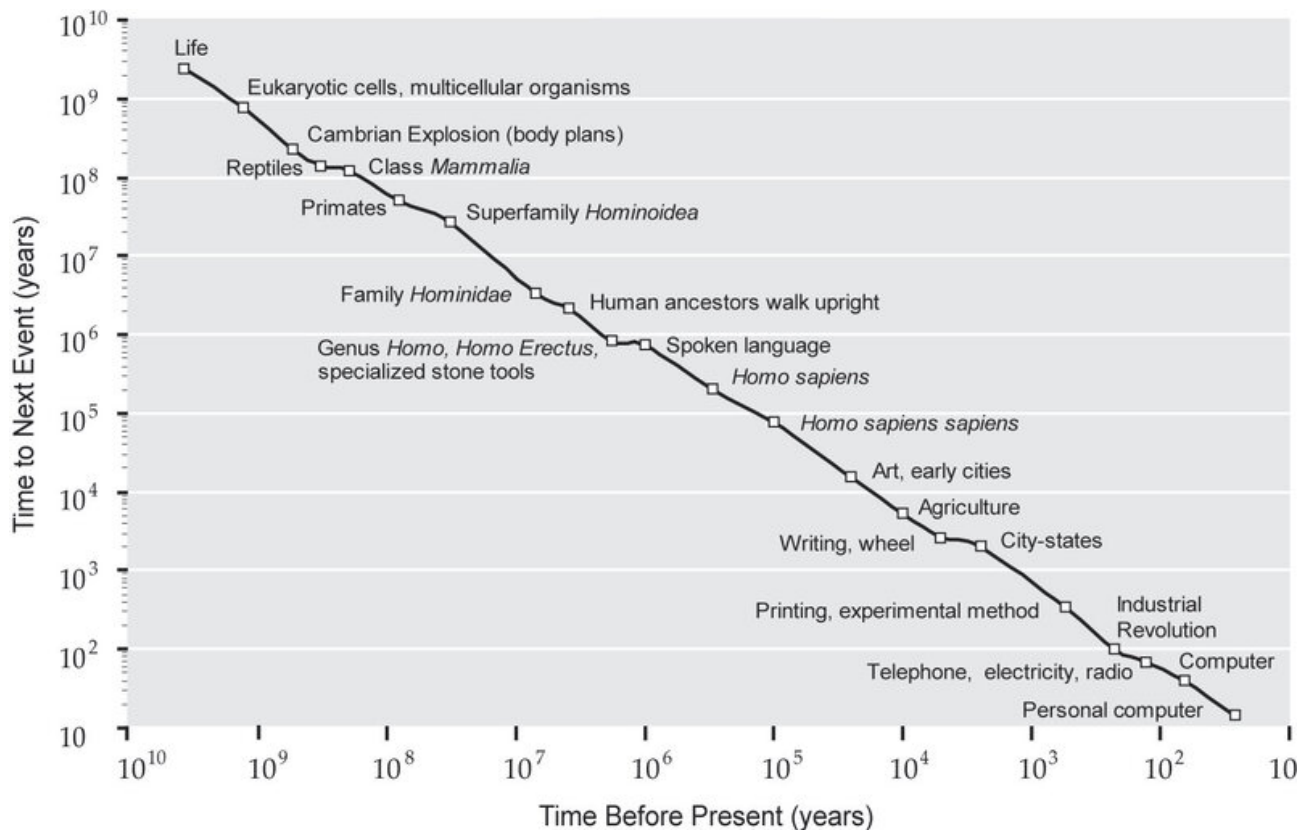


Figure 15: From *The Singularity Is Near* by Ray Kurzweil

The scale of this chart could be initially misleading: the visual distance between the birth of life and the first eukaryotic cells—2 billion years—is represented identically as the distance between the industrial revolution and the personal computer—200 years.

The chart below tells the same story on a scale more easily digestible by us.



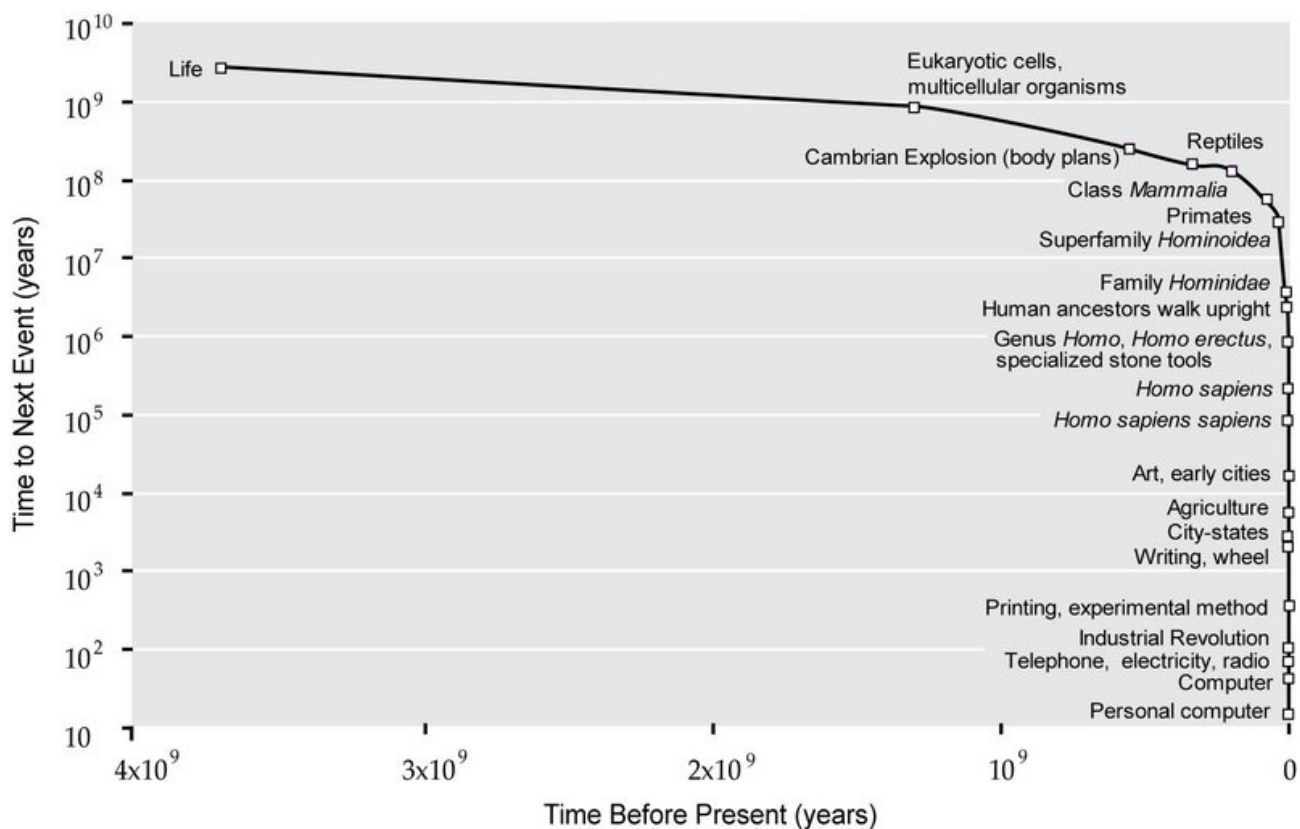
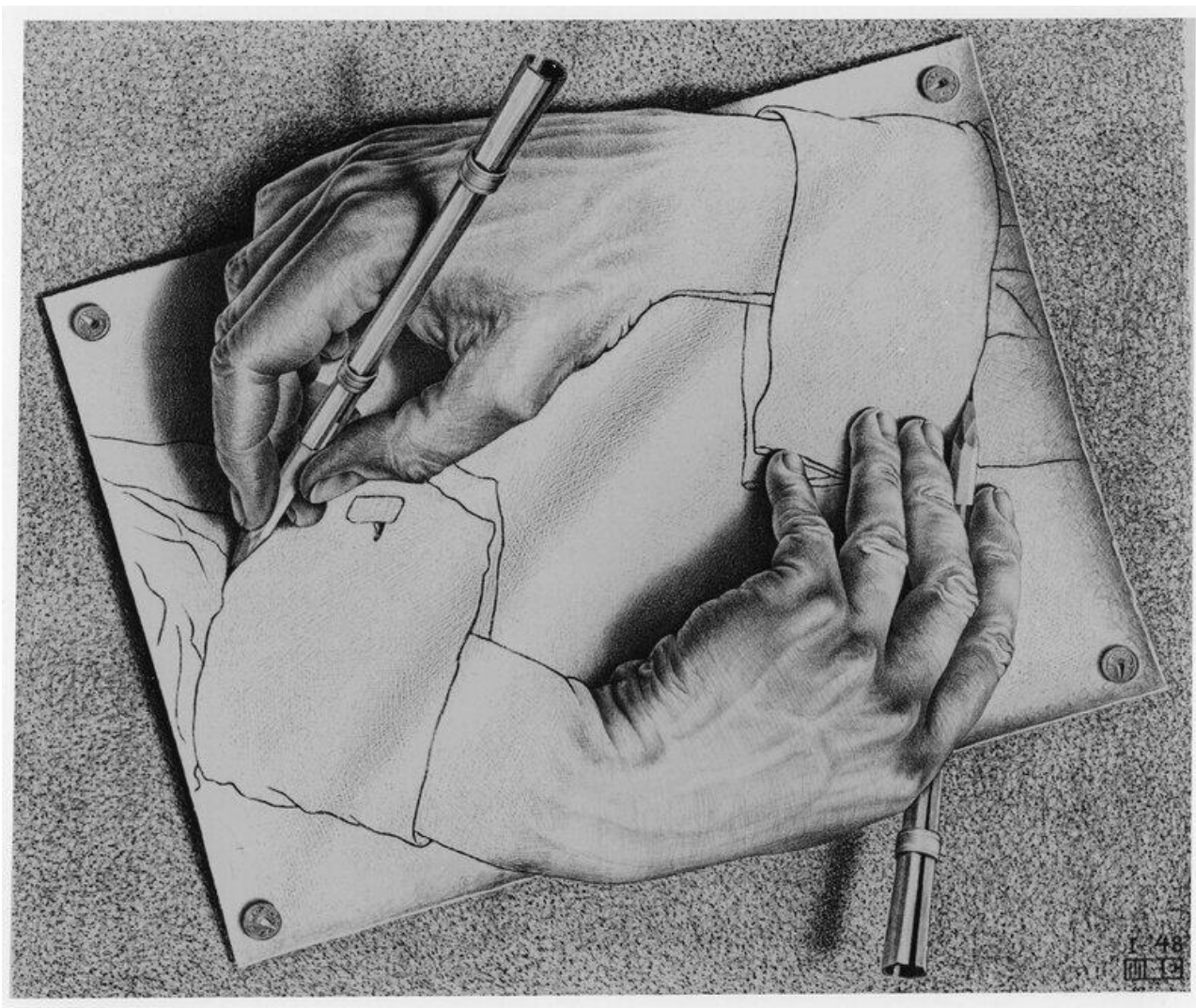


Figure 16: From *The Singularity Is Near* by Ray Kurzweil

The mostly horizontal line depicts the slow process of biological evolution, which eventually —out of only randomness and constraints—brought our neomammalian brains into existence, kick-starting the journey of fire and language towards modern civilization. It's been a long ride, and in the relative time scale of biology our evolution through technology is happening fast and only seems to be accelerating.

Each technological advance builds upon the last, creating a positive feedback loop of progress.





We shape our tools, and then our tools shape us.

— Marshall McLuhan

We are inevitably shaped by what we create, and fundamentally driven by our deeply human essence: the anticipation of discovery, the satisfaction of attainment, and the joy in relationships we cultivate along the way.

The technology we create will survive us, and its impact will be unevenly felt—the fruits of progress aren't unconditionally good. [We've come a long way](#) towards improving our lives, and we can still go so much further. There are so many [problems to solve](#).

In these unprecedented times of tremendous individual potential, it's good to keep our values in check and constantly revisit the question: are we building the right things?

History is a metaphorical pathway, and just like physical ones, it's built purposefully by us, based on the topography and constraints of the environment. Unlike physical ones, it sometimes takes us by surprise.



The future is still not determined, and “the best way to predict it is to invent it”. Informed by our knowledge and empowered by our technology, it is up to us to lay the bricks.

* * *

Now, back to work. What was I doing again? Oh yeah, let's open that file.

Thanks to Evan Lezar, Fabrício Nascimento, Helder Ribeiro, John Wiegley and Protesilaos Stavrou for reading and giving suggestions for drafts of parts of this.

Special thanks to Bozhidar Batsov, Hunter McClelland and Thorsten Ball for insightful, attentive and precise feedback.

Murilo Pereira 



I'm a software engineer working on self-driving technology at Argo AI. Previously I worked on stateful distributed systems and container orchestration at Mesosphere.

Reply on Twitter [@mpereira](https://twitter.com/mpereira)

Follow me on GitHub [@mpereira](https://github.com/mpereira)

© Murilo Pereira 2019-2021

