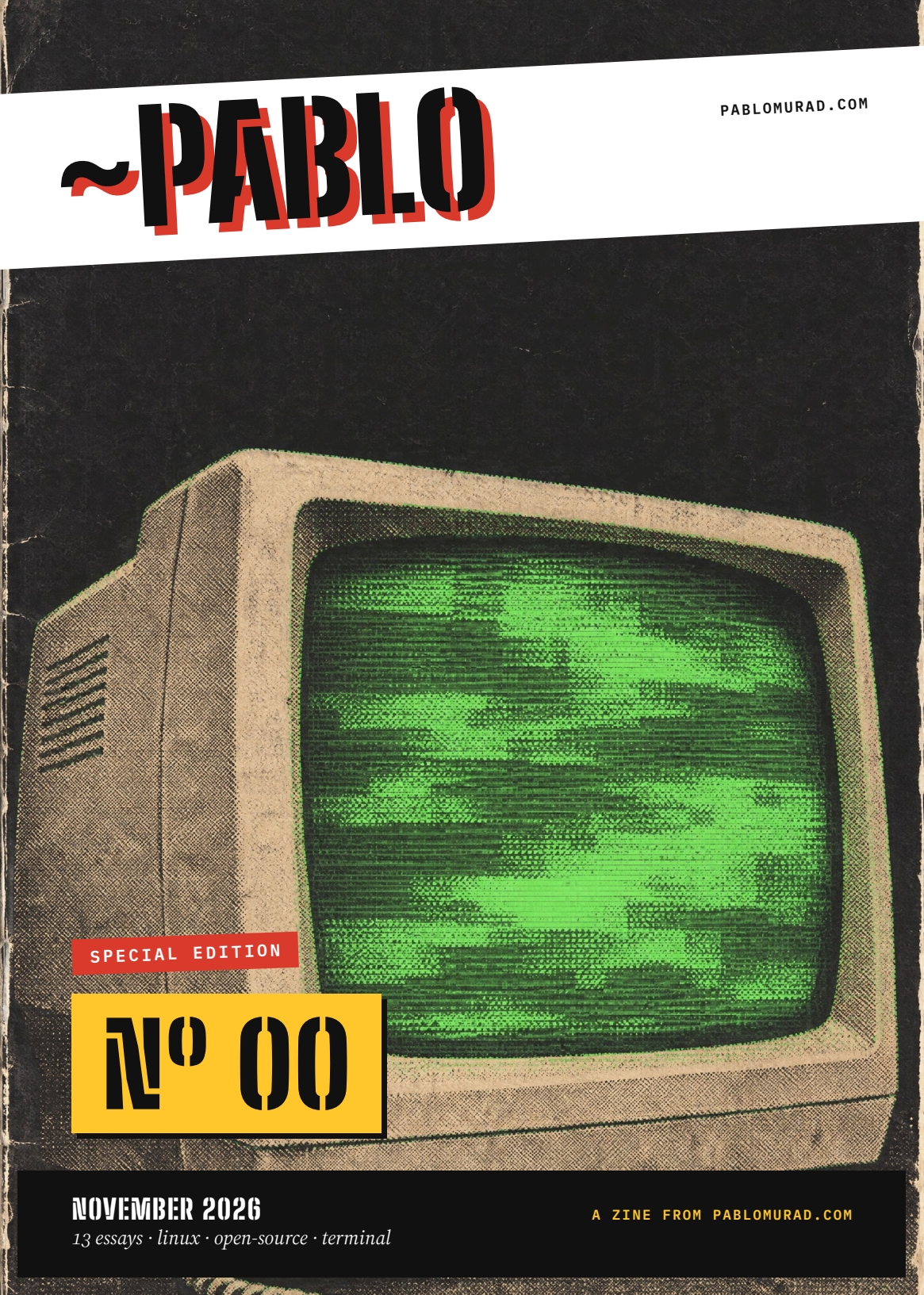
~PABLO

PABLOMURAD.COM



SPECIAL EDITION

Nº 00

NOVEMBER 2026

13 essays · linux · open-source · terminal

A ZINE FROM PABLOMURAD.COM

CONTENTS

Nº 00
SPECIAL EDITION
NOVEMBER 2026

01 Note I: A little about me — 3
PERSONAL · LIFE · REFLECTION

02 Usenet, from 1979 to today — 4
USENET · HISTORY · PROTOCOLS

03 Little things hidden around the internet — 14
TERMINAL · INTERNET · PROTOCOLS

04 Noctem: essays on the arts — 19
PROJECTS · ESSAYS · ART

05 asdf on Arch Linux: one tool for every language version — 24
ARCH-LINUX · LINUX · TUTORIAL

06 Perspective in one SVG path: the floor of pablo.space — 34
SVG · CSS · WEB-DEVELOPMENT

07 Review: Flare puts Bluesky and the fediverse in one timeline — 38
FEDIVERSE · BLUESKY · MASTODON

08 Emmanuel Santiago, the poet of birds — 40
POETRY · BOOKS · BRAZIL

09 Email only the right person can read — 48
EMAIL · PRIVACY · SECURITY

10 The .env file: what it is, where it came from and how to use it well — 58
DOTENV · ENVIRONMENT-VARIABLES · WEB-DEVELOPMENT

11 Dotfiles: the small text files that make a computer mine — 66
DOTFILES · LINUX · TERMINAL

12 Chess over the protocols — 73
CHESSKING · CHESS · FEDIVERSE

13 Note II: motd, an IRC client that gets along with soju — 77
IRC · SOJU · ANDROID

Note I: A little about me

Someone asked me how I manage to produce so much. The short answer was willpower. The real one has to do with attention, crowded rooms, isolation and a head that never stops.

One day someone asked me how I manage to produce so much. I thought about giving the real answer, and just as I was about to start, I stopped. I let it go and said it was willpower. That wasn't a lie, but on its own it wasn't the whole answer either.

The truth is a little **darker** than I'd like.

It's a huge lack of social skills.

I run a pretty great company here in Brazil, one that's known across the country, and still I have no tact at all when it comes to dealing with people.

I have a monstrous attention deficit, and for part of my life it kept me in a depressive state. Yes, that's the part.

So I chose to step away from everything. When I say everything, people think I'm exaggerating, but my trusted circle, what some would call a comfort zone, is tiny. It doesn't include the people I deal with at work, the meetings I run or the events I go to. I won't lie: I hate being in those places.

Crowded places fill me with a dread I can't describe. More than three people at the party? For me, that's packed.

It's hard to explain, but it feels as if I hear everything, every sensory stimulus at once.

What I do have, in place of so many other skills, is the ability to turn a problem into a solution. Nassim Taleb, the author of The Black Swan, has a word for things that get stronger with disorder: antifragile. That's close to what I learned to do with chaos, something past what I could call plain resilience. How I got there is a story for another day.

So, to sum it up.

Take a head that can't focus but can't rest either. One that never stops, with an absurd, almost superhuman need to take in information, plus an enormous hyperfocus on technology and computing. What do you get? Output, and more output.

Most of the posts on this blog were written months ago, some even longer. Others started as notes in Notesnook and were quickly adapted to explain things to someone else. All of it comes from months, sometimes years, of work and observation, and from a deep kind of *isolation*.

That's it. ■

■ ■ ■

02 USENET · HISTORY · PROTOCOLS 15 NOVEMBER 2026

Usenet, from 1979 to today

Usenet has been copying discussions from server to server since 1979. How it started at Duke and UNC, the Big 8 and alt., how an article travels, a real NNTP session, how yEnc, PAR2 and NZB turned it into a file network, and where to read it in 2026.*



Usenet is a worldwide set of discussion groups that no company owns. You write a message, called an article, and post it to a group like `comp.misc`. Your server keeps a copy and passes it to the servers it trades with, they pass it on to theirs, and within minutes the same article sits on servers all over the world. Anyone reading that group, on any of those servers, sees it.

It started in 1979, more than ten years before the web, and it's still running. While writing this post I connected to a free public server and

asked for the newest article in `comp.misc`. It had been posted that same morning, from a newsreader running inside Emacs.

This post covers how Usenet began, how the groups are organized, how an article travels, the protocol your newsreader speaks, and how a network built for short text messages ended up carrying hundreds of terabytes of files a day. At the end, a look at the small news server I run.

Two universities and a phone line

In 1979, Tom Truscott and Jim Ellis, graduate students in computer science at Duke University, had an idea for sharing news and discussion between Unix machines. Steve Bellovin, a graduate student at the University of North Carolina, wrote the first program. The first network was just the computer science departments at Duke and UNC, with Duke's physiology department joining in 1980. The machines called each other over phone lines with UUCP, the Unix-to-Unix copy program, and the modems of the time moved about 300 bits per second.

Ellis and Truscott presented the design at the Usenix conference in Boulder, Colorado, in January 1980. From there, the software went through several generations:

- 1980, A News: a rewrite by Truscott and Steve Daniel, another Duke student.
- 1982, B News: Matt Glickman and Mark Horton made it faster. Rick Adams took over maintenance in 1984.
- 1986, NNTP: a protocol for moving news over TCP/IP networks instead of phone calls. More on it below.
- 1989, C News: Henry Spencer and Geoff Collyer, at the University of Toronto, rewrote it again for speed.
- 1991, INN: Rich Salz released InterNetNews, built around NNTP from the start.

INN is the one that lasted. It's still maintained, version 2.7.4 came out in June 2026, and every server I connected to for this post was running it, mine included.

Groups, hierarchies and the Great Renaming

A newsgroup name reads from general to specific, separated by dots: `comp.lang.c` is about computers, then languages, then C. The first part is the hierarchy, and each hierarchy has its own rules for creating groups.



Before 1987, most groups lived under `net.*`, with moderated ones under `mod.*`. That year, the Great Renaming sorted the main groups into seven hierarchies, the Big Seven:

```
comp: computers
misc: everything that fits nowhere else
news: Usenet itself
rec: recreation, hobbies and the arts
sci: science
soc: society and culture
talk: debate on contested topics
```

Creating a group in those hierarchies took a proposal, a discussion and a vote. Not everyone liked that. Soon after the renaming, John Gilmore and Brian Reid created `alt.*`, a hierarchy outside that process where anyone could create a group. The old joke is that alt stands for "Anarchists, Lunatics and Terrorists". In 1995 `humanities.*` joined the seven, and the official set became the Big 8.

There are also national and regional hierarchies, like `de.*` for German and `pl.*` for Polish groups, and private ones that companies and universities keep to themselves.

Some groups are moderated. Your post goes by email to a moderator, who approves it or not, and only approved posts spread. In unmoderated groups, which are most of them, a post goes out as soon as you send it.

How an article travels

Usenet has two kinds of connections, and it helps to keep them apart. You, the reader, talk only to your own news server. Servers talk to each other, each one trading articles with a few neighbors, called peers.

When a server receives a new article, it stores one copy and offers it to all its peers. They do the same with theirs. That's called flooding, and it's why an article reaches the whole network in minutes without any central server.

Flooding in a network full of loops needs two safety catches. The first is the Message-ID, a unique name for each article, like `<82o6brrd46.fsf@example.com>`. Every server remembers the IDs it has seen and refuses copies of articles it already has. The second is the `Path` header: each server adds its name to the front of it before passing the article on, and a server never sends an article to a peer already listed there.

Here's the header of that `comp.misc` article as the server sent it. I only joined one line the server had folded in two and removed the author's email address:

```
Path: nnrp.usenet.blueworldhosting.com!!spool1.usenet.blueworldhosting.com!
usenet.blueworldhosting.com!diablo1.usenet.blueworldhosting.com!weretis.net!
feeder8.news.weretis.net!reader5.news.weretis.net!news.solani.org!.POSTED!
not-for-mail
From: Richmond <address removed>
Newsgroups: comp.misc
Subject: Re: the Microsoft global outage
Date: Sat, 14 Nov 2026 09:11:53 +0000
Organization: Frantic
Message-ID: <82o6brrd46.fsf@example.com>
References: <6af7cacb$11$74195$882e4bbb@reader.netnews.com>
MIME-Version: 1.0
Content-Type: text/plain
Injection-Info: solani.org; logging-data="1635832"; mail-complaints-
to="abuse@news.solani.org"
User-Agent: Gnus/5.13 (Gnus v5.13)
Cancel-Lock: sha1:Jfrmwh9WRrfAhk/GPoE/RyOAAEw=
sha1:2erpllyTKwSdoe03AnYk97zru/Y=
Xref: nnrp.usenet.blueworldhosting.com comp.misc:41148
```

Read the `Path` from right to left and you have the article's trip. It was posted at `news.solani.org` (that's what `.POSTED` marks), went through three servers at `weretis.net`, and then through four machines at Blue World Hosting before it reached the one I was talking to.

The rest is close to an email, and that's on purpose: the article format is a cousin of the email format. The current rules are in RFC 5536, from 2009, which replaced the older RFC 1036. Every article must have exactly

one each of `From`, `Date`, `Message-ID`, `Newsgroups`, `Path` and `Subject`. A few of the others are worth knowing:

```
References: the Message-IDs of the articles this one replies to, which is
how readers build threads
Newsgroups: can list more than one group (a crosspost), and the article is
still stored only once
Injection-Info: where the article entered the network, and where to send
abuse complaints
Cancel-Lock: lets the author, and only the author, cancel or replace the
article later
Xref: this article's number in each group, on this server only
User-Agent: the newsreader that wrote it, here Gnus, which comes with Emacs
```

About canceling: Usenet always had a control message to ask every server to delete an article. Anyone could forge one, so for years people canceled other people's posts, and many servers stopped obeying cancels at all. `Cancel-Lock` is the later fix, a secret only the original poster can prove they know.

NNTP: the protocol

The first Usenet copied articles in batches between machines over UUCP, and every machine kept a full copy of everything. In February 1986, Brian Kantor and Phil Lapsley published RFC 977, the Network News Transfer Protocol. Its idea was to keep the articles on one server per site and let everyone else read them over the network, instead of keeping duplicate copies on every host. The current version is RFC 3977, from 2006.

NNTP is a text protocol, like SMTP for email: you send a command as a line of text, and the server answers with a three-digit code and some text. The usual port is 119, and 563 for connections with TLS. Here's the real session behind the header above, in order (C is me, S is the server):

```
S: 200 nnrp.usenet.blueworldhosting.com InterNetNews NNRP server INN 2.8.0
(20260912 snapshot) ready (posting ok)
C: CAPABILITIES
S: 101 Capability list:
S: VERSION 2
S: IMPLEMENTATION INN 2.8.0 (20260912 snapshot)
S: AUTHINFO USER
S: COMPRESS DEFLATE
S: HDR
S: LIST ACTIVE ACTIVE.TIMES COUNTS DISTRIB.PATS DISTRIBUTIONS HEADERS
MODERATORS MOTD NEWSGROUPS OVERVIEW.FMT SUBSCRIPTIONS
S: NEWNEWS
S: OVER
S: POST
```

```
S: READER
S: STARTTLS
S: XPAT
S: .
C: MODE READER
S: 200 nnrp.usenet.blueworldhosting.com InterNetNews NNRP server INN 2.8.0
(20260912 snapshot) ready (posting ok)
C: GROUP comp.misc
S: 211 40524 1 41148 comp.misc
C: HEAD 41148
S: 221 41148 <82o6brrd46.fsf@example.com> head
S: (the header shown above)
S: .
C: QUIT
S: 205 Bye!
```

`GROUP comp.misc` selects a group, and the `211` answer means: about 40,524 articles, numbered from 1 to 41148, in `comp.misc`. `HEAD 41148` asks for the header of the newest one. A line with a single dot ends any answer that has more than one line.

The first digit of every answer tells you how it went:

```
1xx: information
2xx: done, it worked
3xx: so far so good, send the rest (for example, the text of a post)
4xx: the command was fine, but it failed
5xx: unknown command, not allowed, or a syntax error
```

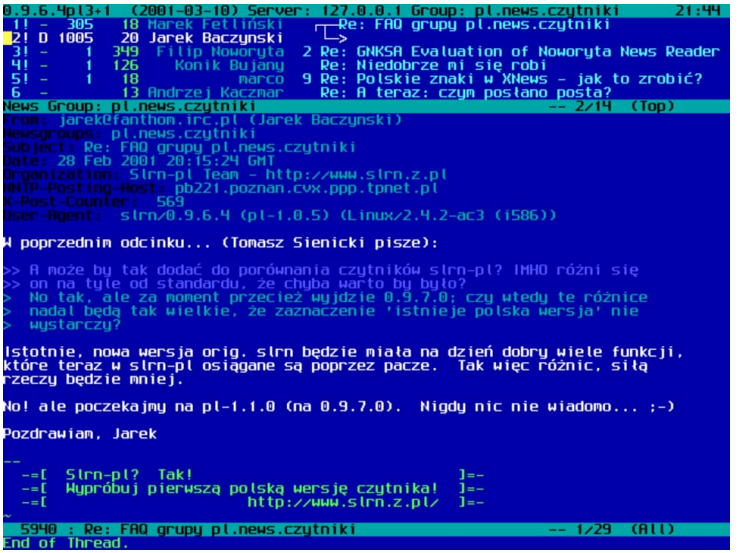
And these are the commands you'd use most:

```
CAPABILITIES: what this server supports
GROUP name: select a group
ARTICLE, HEAD, BODY: get a whole article, only its header, or only its text
OVER: get a summary line of many articles at once (subject, author, date,
size), which is how readers fill their list of threads
POST: send a new article
IHAVE <message-id>: one server offering an article to another
QUIT: say goodbye
```

`IHAVE` is the server-to-server side. Before sending a whole article, a server offers its Message-ID. The other one answers "send it" or "already have it". That one line keeps flooding cheap: an article the other side already has is never sent again.

You can try all of this by hand. Open a connection to a server that allows reading without an account, type the commands and press Enter after each one:

```
telnet news.blueworldhosting.com 119
```



Newsreaders

Nobody reads Usenet by typing `HEAD` all day. A newsreader does the talking, keeps track of what you've read, builds threads from the `References` headers and lets you hide people and subjects you don't want to see. Usenet's old word for a list of hidden authors is a killfile, and the noise of dropping someone into it is "plonk".

Many of the classic ones are still around:

- tin and slrn run in the terminal. The picture above is slrn in 2001, in a Polish group about newsreaders.
- Gnus comes inside Emacs. The `comp.misc` article above was written with it.
- Thunderbird reads newsgroups next to email, and Pan is a graphical reader that also handles binaries.

If you use Emacs, two lines are enough to read real groups. Gmane, a service that mirrors thousands of mailing lists as newsgroups, has the Org mode list as `gmane.emacs.orgmode`:

```elisp
(setq gnus-select-method '(nntp "news.gmane.io"))
;; then M-x gnus, and U to subscribe to gmane.emacs.orgmode
```

Files on a network made for text

Usenet was built for short text. Articles were 7-bit text, and a binary file, like a picture or a program, had to be turned into printable characters before it could be posted. The usual tools for that, uuencode and later Base64, make a file 33 to 40 percent bigger.

People posted binaries anyway, in groups under `alt.binaries.*`, and I wrote about how that works today in How Usenet serves files. The short version is three inventions, all from the early 2000s.

In 2001, Jürgen Helbing released yEnc. Its bet was that by then nearly every server could carry 8-bit bytes untouched, so there was no need to squeeze files into printable characters. yEnc adds 42 to each byte and escapes only the few values that would break an article (NUL, line feeds, carriage returns and the `=` sign it uses as an escape). The overhead drops to 1 or 2 percent. A file too big for one article is split into parts, and each part is wrapped in lines like these:

```
=ybegin part=1 line=128 size=500000 name=mybinary.dat
=ypart begin=1 end=100000
(the encoded data for those bytes)
=yend size=100000 part=1 pcrc32=abcdef12
```

`size` is the whole file, `begin` and `end` say which bytes this part holds, and `pcrc32` is a checksum of the part, so the downloader can tell whether it arrived intact.

In 2002, Michael Nahas published version 2.0 of Parchive, better known as PAR2. A big file turns into thousands of articles, and some never reach your server or expire before you get to them. PAR2 recovery files carry extra blocks computed from the original, and they can rebuild the missing ones.

The last piece was the NZB, an XML format created by the developers of the Newzbin index. It lists the Message-ID of every article that makes up a file, so a program can ask the server for each one with `ARTICLE <message-id>`, decode the parts, repair what's missing with PAR2 and put the file back together.

The binary side grew enormous. NewsDemon, one of the providers, estimates that a large backbone takes in 400 to 600 terabytes of new articles a day in 2026. Eweka, another provider, advertises more than 6,662 days of retention, which is over 18 years of articles kept online. Most of that is binaries, and the binary groups are mostly read through paid accounts. The text groups are a tiny part of the traffic.

Famous posts and hard years

A lot of internet history was first posted on Usenet. On August 6, 1991, Tim Berners-Lee announced the World Wide Web in `alt.hypertext`. On August 25, 1991, Linus Torvalds asked `comp.os.minix` what people would like to see in Minix, while telling them about the free operating system he was writing, which became Linux.

Then the crowds came. Every September, a new class of university students got accounts and had to learn the local manners. In 1993 and 1994, internet providers and online services started giving Usenet to many new users, AOL among them from March 1994, and the newcomers never stopped arriving. Old users called it the September that never ended, or Eternal September.

On April 12, 1994, two lawyers, Laurence Canter and Martha Siegel, posted an ad for their green card lottery service to at least 5,500 newsgroups, one separate copy per group. It's considered the first big commercial spam on Usenet, and spam never left.

From 2008, internet providers started to drop Usenet. After an agreement with New York's attorney general, several American providers stopped offering it or cut whole hierarchies, and AT&T ended its Usenet service in July 2009. In May 2010, Duke, where it had all started, shut down its own news server, citing low use and rising costs. On February 22, 2024, Google Groups stopped carrying Usenet: no new posts, no new articles, and Google's server stopped peering with the rest of the network. The old archive is still there to search.

Usenet in 2026

The text side is small, but it's alive, and still free. A few places to start:

- Eternal September gives free accounts on a server for the text groups.
- Blue World Hosting let me read `comp.misc` without any account, as in the session above.
- Gmane serves thousands of mailing lists over NNTP at `news.gmane.io`.

Point any newsreader at one of them, subscribe to a handful of groups, and read the way people read in 1990: threads in plain text, no feed choosing what you see, and the same articles copied across servers run by many different people.

A news server of my own

skull.black is a small private news server I put together to learn how NNTP works under the hood. I told how it started in A piece of me in

every corner of the internet. It runs INN 2.7.4, the current release, and this is what a newsreader meets when it connects to `news.skull.black`, before logging in:

```
S: 200 news.skull.black InterNetNews NNRP server INN 2.7.4 ready (no
posting)
C: CAPABILITIES
S: 101 Capability list:
S: VERSION 2
S: IMPLEMENTATION INN 2.7.4
S: AUTHINFO USER
S: COMPRESS DEFLATE
S: HDR
S: LIST ACTIVE ACTIVE.TIMES COUNTS DISTRIB.PATS DISTRIBUTIONS HEADERS
MODERATORS MOTD NEWSGROUPS OVERVIEW.FMT SUBSCRIPTIONS
S: NEWNEWS
S: OVER
S: POST
S: READER
S: XPAT
S: .
C: GROUP skull.general
S: 480 Authentication required for command
C: QUIT
S: 205 Bye!
```

The whole conversation runs over TLS on port 563, from the first byte, so passwords never cross the network in the clear. Until you log in, with `AUTHINFO USER` and `AUTHINFO PASS`, the greeting says `no posting` and any command that touches the groups gets a `480`. The same account works in a newsreader and in the browser at skull.black, which shows the same groups as threads on a web page.

The server has its own hierarchy, `skull.*`, with seven groups:

```
skull.meta: rules, announcements and welcome
skull.general: general chatter
skull.tech: technology and craft
skull.art: art and writing
skull.dev: commits and dev
skull.news: news from RSS feeds
skull.binaries: binaries, stored on S3 with unlimited retention
```

`skull.dev` and `skull.news` are seeded, one with commits and the other with items from RSS feeds. `skull.binaries` keeps its files on S3 storage, with no expiry.

skull.black doesn't trade articles with other servers. Its only feed is itself, so the `skull.*` groups live there and nowhere else, the same idea as the private hierarchies that companies and universities kept on the same software as the public network. If you want to try it, sign up on the site,

then point Thunderbird, Pan, Gnus or any newsreader that speaks TLS at `news.skull.black`, port 563. ■

■ ■ ■

03 TERMINAL · INTERNET · PROTOCOLS 16 NOVEMBER 2026

Little things hidden around the internet

Five days ago I found out Twitch chat works in an IRC client, so I went looking for more little hidden things: Star Wars over telnet, weather by finger and DNS, funny robots.txt files, Terry Pratchett in Mozilla's headers, coffee over SSH and the pages your phone checks on Wi-Fi.

About five days ago I found out that you can read Twitch chat with an ordinary IRC client. That made me curious about what else is out there, quietly answering for anyone who knocks on the right door, so I went looking for more of these little hidden things. This is what I found. Everything here was tested today.

Old protocols that still answer

Twitch chat is IRC. The server is `irc.chat.twitch.tv`, port `6697` with TLS, and each channel is `#` plus the streamer's name. To just read, you don't need an account: a nickname that starts with `justinfan` plus some numbers gets you in. The server's message of the day is a line from the 1970s game Colossal Cave Adventure:

```
:tmi.twitch.tv 001 justinfan48213 :Welcome, GLHF!
:tmi.twitch.tv 372 justinfan48213 :You are in a maze of twisty passages,
all alike.
```

Star Wars in ASCII. Simon Jansen's ASCIImation of A New Hope, put online over telnet, has been playing for decades. The credits end with "So long and thanks for all the fish".

```
telnet towel.blinkenlights.nl
```

A fake 1980s internet. Telehack simulates the ARPANET and Usenet era, with dozens of commands: `advent`, `eliza`, `starwars`, `usenet`, `finger`. When I connected it said “There are 26657 hosts on the network” and “May the command line live forever”.

```
telnet telehack.com
```

NetHack, still going. nethack.alt.org hosts public games of the classic dungeon crawler. When I checked, 12 people were playing and the server had recorded about 8.7 million games.

```
telnet nethack.alt.org
```

Gopher. The protocol that almost became the web, from 1991, is still alive. Floodgap has run a gopher server since 1999, on an IBM Power 520 running AIX, as its welcome message proudly explains. curl speaks gopher:

```
curl gopher://gopher.floodgap.com/
```

Weather by finger. The finger protocol was made to ask who was logged in to a machine. graph.no uses it to send you a weather chart drawn in text:

```
finger oslo@graph.no
```

A quote of the day, by protocol. RFC 865, from 1983, defines a service on port 17 whose only job is to send you a quote and hang up. Some servers still run it:

```
nc djxmmx.net 17
"In Heaven an angel is nobody in particular." George Bernard Shaw
(1856-1950)
```

The time, from NIST. Same idea, RFC 867, port 13. The US time lab still answers:

```
nc time.nist.gov 13
61359 26-11-15 22:22:16 00 0 0 705.5 UTC(NIST) *
```

On Windows, `telnet` is off by default (turn on Telnet Client in “Turn Windows features on or off”), `finger` comes built in, and you can use `telnet host port` wherever this post says `nc`.

Things that live in the terminal

Weather. `curl wttr.in/Sao+Paulo` draws a full forecast with little ASCII clouds.

Cheat sheets. `curl cheat.sh/tar` shows quick examples for almost any command, so you don't have to remember the flags of `tar` ever again.

A dancing parrot. `curl parrot.live`, in color, until you press Ctrl+C.

A chat room over SSH. `ssh ssh.chat` drops you into a public chat room. Your SSH key is your identity, no sign-up.

Coffee over SSH. terminal.shop sells coffee only through the terminal: `ssh terminal.shop` opens a store with menus, an address form and checkout. It launched in 2024, built live on stream by a group of developers, and the first batch sold out within days. According to its site, the coffee is Brazilian.

A free Unix account. SDF has offered public Unix accounts since 1987, and you sign up over SSH with `ssh new@sdf.org`.

DNS tricks

DNS is the internet's phone book, but some servers answer more than phone numbers.

Your IP, from Google. Google's own name server tells you the address your request came from:

```
nslookup -type=TXT o-o.myaddr.l.google.com ns1.google.com
```

Which datacenter you hit. Ask Cloudflare's 1.1.1.1 a question in the CHAOS class, an old DNS class almost nobody uses, and it names the datacenter that answered. Cloudflare names them after airports:

```
nslookup -type=TXT -class=CHAOS id.server 1.1.1.1
"gig11"
```

GIG is Rio de Janeiro's airport code, so my question was answered in Rio.

Weather, again. dns.toys puts small tools inside DNS:

```
nslookup -type=TXT saopaulo.weather dns.toys
"Sao Paulo (BR)"
"19.80C (67.64F)"
"93.50% hu."
```

Notes left for whoever reads the source

Google is a teapot. google.com/teapot answers with HTTP status 418, "I'm a teapot", a joke from an April Fools' RFC published in 1998.

humans.txt. Some sites keep a file for people, next to robots.txt for machines. Google's starts with "Google is built by a large team of engineers, designers, researchers, robots, and others". Netflix's is a whole movie screen drawn in ASCII, "STARRING an all-star cast of talented designers & engineers".

robots.txt with a sense of humor. YouTube's begins: "Created in the distant future (the year 2000) after the robotic uprising of the mid 90's which wiped out all humans." Last.fm's forbids robots from three paths: `/harming/humans`, `/ignoring/human/orders` and `/harm/to/self`, Asimov's three laws in crawler form.

```
curl https://www.youtube.com/robots.txt
curl https://www.last.fm/robots.txt
```

A name in the headers. Mozilla's website sends an extra HTTP header on every page: `X-Clacks-Overhead: GNU Terry Pratchett`. It comes from Pratchett's Discworld, where messages kept a dead person's name moving through the clacks towers so they wouldn't be forgotten. After he died in 2015, people started adding it to their servers.

```
curl -I https://www.mozilla.org | grep -i clacks
x-clacks-overhead: GNU Terry Pratchett
```

GitHub's zen. `curl https://api.github.com/zen` returns a random line of GitHub's design philosophy, like "Encourage flow." And `github.com/username.keys` shows anyone's public SSH keys, which is handy for giving someone access to a server.

Python's inside jokes. Run these in Python:

```
import this: prints The Zen of Python, by Tim Peters
import antigravity: opens xkcd's comic about Python in your browser
from __future__ import braces: SyntaxError: not a chance
```

How your phone knows it's online

When you join a Wi-Fi network, your phone or computer quietly fetches a tiny page to check whether the internet works or a hotel login page is in the way. You can open the same pages:

```
captive.apple.com: Success
connectivitycheck.gstatic.com/generate_204: an empty answer with status 204
(Android, Chrome)
detectportal.firefox.com/success.txt: success
www.msftconnecttest.com/connecttest.txt: Microsoft Connect Test
```

If the answer is anything else, your device assumes there's a login page and shows it to you.

What the internet sees of you

icanhazip.com returns your IP and nothing else. Major Hayden started it in 2009 as an internal tool. By 2021 it was handling over a billion requests a day, then 30 to 35 billion a day after a sudden flood of traffic, more than 90% of it malicious. That June he handed it over to Cloudflare, which runs it today.

Cloudflare's trace. Any site behind Cloudflare answers at `/cdn-cgi/trace` with your IP, country, the datacenter that served you (again by airport code), and your HTTP and TLS versions:

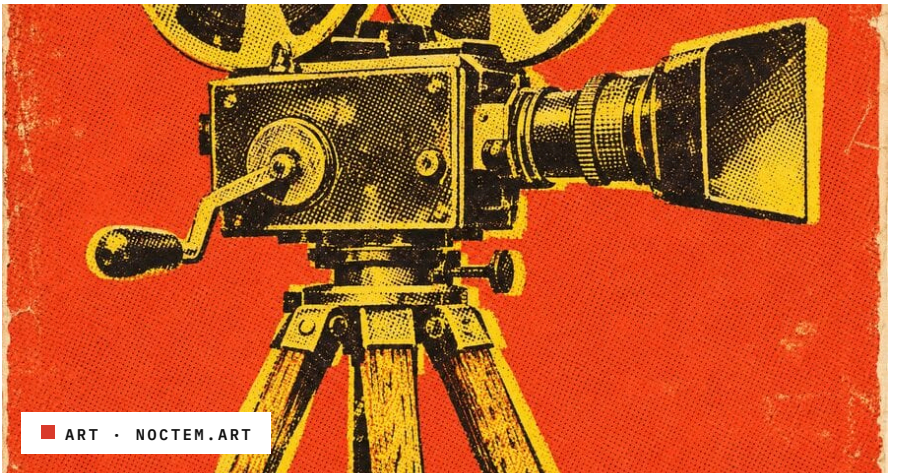
```
curl https://www.cloudflare.com/cdn-cgi/trace
```

Mullvad's check. `curl https://am.i.mullvad.net/connected` tells you whether your traffic is going through Mullvad's VPN, and shows your IP if it isn't.

Feeds nobody advertises

YouTube and Bluesky never mention it, but both publish RSS. Every YouTube channel has a feed at `youtube.com/feeds/videos.xml?channel_id=` followed by the channel ID, and every Bluesky profile has one if you add `/rss` to its address, like `bsky.app/profile/bsky.app/rss`. Put them in a feed reader and you can follow people without opening either app.

If you know more little things like these, send them my way. ■



■ ART · NOCTEM.ART

04. PROJECTS · ESSAYS · ART 16 NOVEMBER 2026

Noctem: essays on the arts

My cousin Henrique Murad built noctem.art so we could write about culture in Portuguese together. Essays on twelve arts, from the old trivium and quadrivium to cinema, games, technology and life, plus a monthly zine to print, fold and staple.



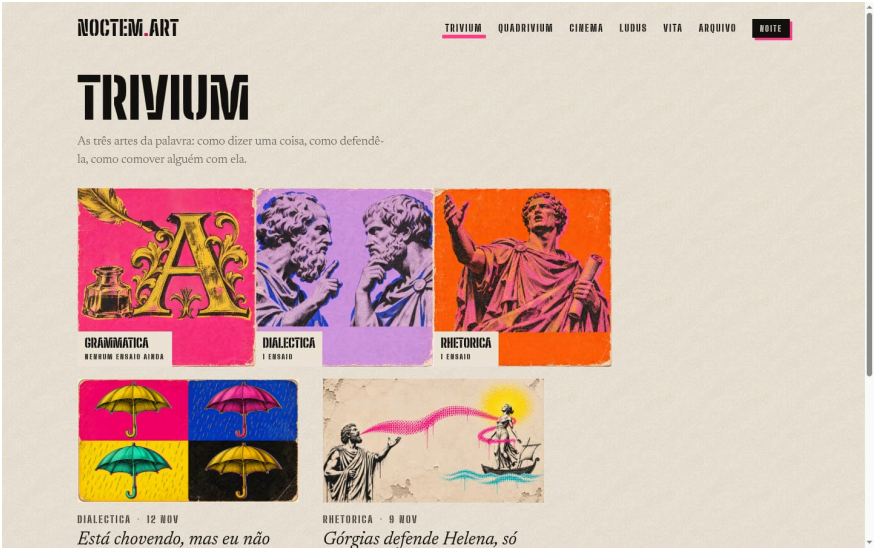
This blog is in English, and it's staying that way. But I've wanted a place to write about culture in Portuguese: analyses of films, of music, of whatever else deserves a long read.

That place exists now. It's called noctem.art, and my cousin, Henrique Murad, built the whole thing so the two of us could write there together.

The name

Noctem is Latin. It's a form of *nox*, night, the one Latin uses after *per* and *in*: *per noctem*, through the night, and *in noctem*, into the night.

The tagline says the rest: "Ensaios sobre as artes, escritos à noite", essays on the arts, written at night. The footer adds "a noite é das artes", the night belongs to the arts. The timestamps agree with both. The first eight essays all went live between 9:50 and 11:15 in the evening, São Paulo time.



Twelve doors

Noctem borrows its map from the medieval school, where an education rested on seven liberal arts.

Three of them, the trivium, are arts of the word: grammar, dialectic and rhetoric. The site describes them as how to say a thing, how to defend it, and how to move someone with it. The other four, the quadrivium, are arts of number. Arithmetic is number by itself, geometry is number in space, music is number in time, and astronomy is number in space and time.

Noctem keeps the Latin names and puts philosophy over all seven. Then it adds four that no medieval master would have put on the list: cinema, games (*ludus*), technology (*techne*) and life (*vita*). The homepage calls the set “Sete artes e mais quatro”, seven arts and four more.

Every essay belongs to exactly one art, and every art has a door: a square illustration in its own color, with a short looping video behind the picture.

What’s on the walls so far

Ten essays are up:

- Cinema: Zé do Caixão, o bicho-papão que o Brasil fez à mão, about José Mojica Marins, who gave Brazil its first horror film and its own boogeyman with no money, no studio and the military regime watching
- Cinema: Ex Drummer, ou como visitar o inferno e voltar para o jantar, on Koen Mortier's 2007 film
- Cinema: O estranho mundo de John Waters
- Musica: The Cult aos quarenta, ainda em chamas, the band's fortieth anniversary show at Vibra São Paulo, seen from up top
- Musica: Marilyn Manson, o monstro que os Estados Unidos encomendaram, his story, ahead of his show at The Town in Interlagos in September 2027
- Rhetorica: Górgias defende Helena, só por diversão
- Dialectica: Está chovendo, mas eu não acredito, on Moore's paradox
- Philosophia: Você se ligaria na máquina?, on Robert Nozick's experience machine
- Geometria: O cubo, a esfera e os olhos que precisaram aprender, on Molyneux's question to John Locke
- Techne: k-punk, o blog que viu o futuro ser cancelado, on Mark Fisher's blog

Five doors are still closed: Grammatica, Arithmetica, Astronomia, Ludus and Vita. I want to open all of them.

How it looks

Henrique gave the site a look of its own. Pages sit on warm, worn paper. Titles come in a big italic serif, and the labels in a heavy stencil face. The covers are stencil and pop art in magenta, yellow and halftone, sprayed on cracked walls and old paper. Between sections, a band of black paint drips down the page, and every essay opens with an oversized drop cap.

There's a day theme and a night theme, and the button in the corner that switches them is labeled "Noite". On a phone, the menu wraps under the logo and the essays read in one column.

The old web is in there too. The site has RSS, Atom and JSON feeds, and the footer carries an 88x31 badge.

The zine

Noctem is going to print. Once a month, the essays leave the screen and become a zine: an A5 PDF that you can read on screen or send to a print shop, and a booklet edition for the printer at home, which you fold and staple yourself.





Issue Nº 01 comes out on December 1. The zine page explains how it works.

What Noctem is for

Noctem is a place for essays about the arts, in Portuguese, meant to be read slowly. Each essay takes one subject and gives it room: the history, the people around it, the context, and an opinion when there is one. You won't find a news feed or star ratings there.

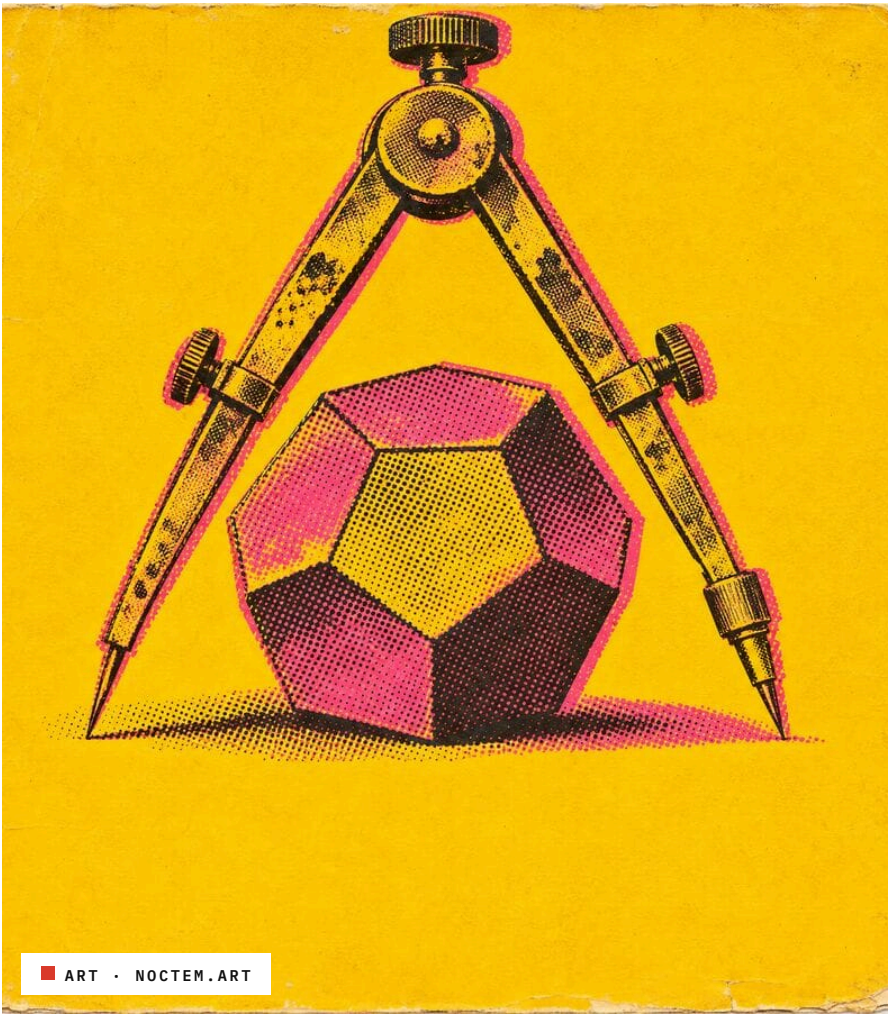
Art gets a wide definition. A Greek sophist, a Belgian punk film, a philosopher's blog and a shock rocker all fit in the same archive, and the twelve doors exist so each of them has a room. When it can, the site stays close to home: a filmmaker from São Paulo, a concert at Vibra São Paulo, a festival in Interlagos.

The reading also goes beyond the screen. Anyone who prefers paper gets the zine.

Who writes there

For now, Henrique and me. My essays are the ones up so far, and his author page is ready for his first.

Everything is in Portuguese. If you read it, noctem.art is open, and the feed is at noctem.art/feed. ■

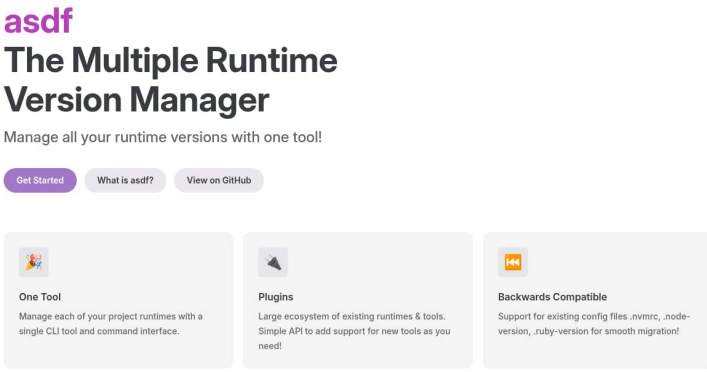


ART · NOCTEM.ART

05 ARCH-LINUX · LINUX · TUTORIAL 16 NOVEMBER 2026

asdf on Arch Linux: one tool for every language version

How I use asdf on Arch Linux to keep Node.js, Python and Go versions per project, away from pacman's rolling updates. Installing from the AUR, shell setup, .tool-versions, real terminal output, and why it works the same on any distro.



Arch is a rolling release. A `pacman -Syu` brings the newest version of everything, Node.js and Python included. The system likes that. A project written for an older Node often doesn't. On my Arch machine, the languages my projects run on stay out of pacman's hands, and the tool that handles them is asdf.

What asdf is

asdf is a version manager. It installs and switches versions of programming languages and tools: Node.js, Python, Ruby, Go, Elixir, Erlang, Java, Terraform, kubectl and hundreds more.

You've probably met tools that do this for one language: nvm for Node, pyenv for Python, rbenv for Ruby. asdf does the same job for all of them with one set of commands. Each language comes in through a plugin, and each project says which versions it wants in a small text file called `.tool-versions`.

Up to version 0.15, asdf was a set of Bash scripts that you loaded into your shell. Version 0.16 rewrote it in Go, so now it's a single binary. The current release is 0.20.2, from September 2026.

Why I use it

- One tool and one set of commands for every language. Install, list, switch and remove work the same way for Node, Python or Go.
- Each project carries its own versions. The `.tool-versions` file goes into git with the code, so whoever clones it (me on another machine, someone else, a CI job) gets the same versions.
- The switch is automatic. You `cd` into a folder and the right version is there. There's no `nvm use` to remember.
- The system is left alone. Everything asdf installs lives in `~/.asdf`, with no `sudo`. The Python that pacman installs, and that Arch's own tools depend on, is never touched.
- Upgrades become a decision. When pacman moves Node to a new major version, my projects don't move with it. I change a version when I choose to.
- It reads the files other managers use. With one config line, asdf respects `.nvmrc`, `.node-version` and `.ruby-version`, so moving from nvm or rbenv doesn't mean editing every project.

Does this work on other distros?

Yes. asdf is a single binary plus a folder of shims on your `PATH`, and nothing in it depends on Arch. The setup, the commands and the `.tool-versions` file in this post are the same on Debian, Ubuntu, Fedora, openSUSE or any other Linux, and on macOS too.

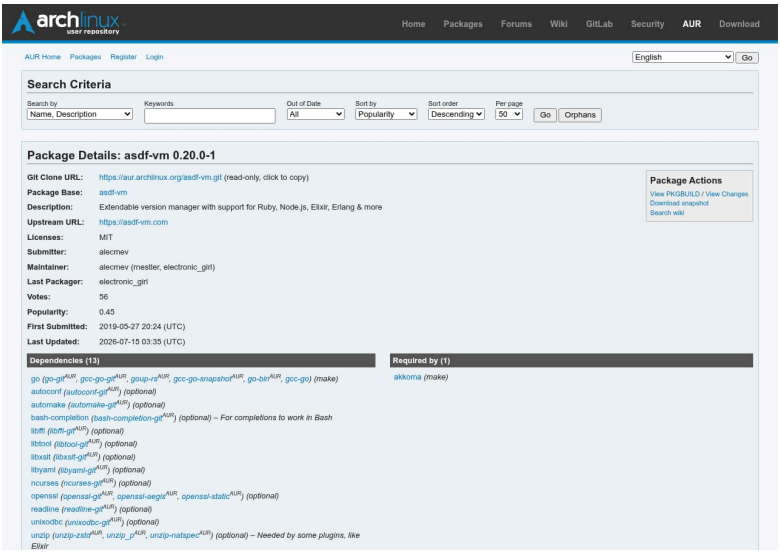
Two things change from distro to distro:

- How you install asdf. The AUR is an Arch thing. On other distros you use the pre-compiled binary, `go install`, Homebrew, or `zypper` on openSUSE. They're all below.
- The names of the packages some plugins need, like the build dependencies for Python. I'll show the Arch names, and each plugin's page lists the others.

The release page has builds for Linux and macOS only. On Windows, the way to use asdf is inside WSL.

Installing on Arch

asdf isn't in the official repositories. It lives in the AUR, as asdf-vm.



With an AUR helper like yay or paru:

```
yay -S asdf-vm
```

Without a helper, the manual way, which needs `git` and `base-devel`:

```
sudo pacman -S --needed git base-devel
git clone https://aur.archlinux.org/asdf-vm.git
cd asdf-vm
makepkg -si
```

The package compiles asdf from source with Go (makepkg pulls `go` in as a build dependency), puts the binary at `/usr/bin/asdf`, and installs completions for Bash, Zsh and Fish.

One warning: search the AUR for “asdf” and you’ll also find `python-asdf`. That one is a Python library for the Advanced Scientific Data Format, used in astronomy. It has nothing to do with this.

OTHER WAYS, FOR ANY DISTRO

The pre-compiled binary works everywhere. Download the archive for your system from the releases page, extract it and put `asdf` somewhere on your `PATH`:

```
mkdir -p ~/.local/bin
curl -LO https://github.com/asdf-vm/asdf/releases/download/v0.20.2/asdf-v0.
20.2-linux-amd64.tar.gz
tar -xzf asdf-v0.20.2-linux-amd64.tar.gz -C ~/.local/bin asdf
type -a asdf
```

If you already have Go, one line does it:

```
go install github.com/asdf-vm/asdf/cmd/asdf@v0.20.2
```

That puts the binary in `~/go/bin`, which then needs to be on your `PATH`.

Telling your shell about it

asdf works through shims: small programs in `~/.asdf/shims` with the names of the tools you install (`node`, `npm`, `python`, `go`). That folder has to come first in your `PATH`, before `/usr/bin`, or your shell will find pacman's versions first.

For Bash, add this to `~/.bash_profile`:

```
export PATH="${ASDF_DATA_DIR:-$HOME/.asdf}/shims:$PATH"
```

For Zsh, the same line goes in `~/.zshrc`.

For Fish, add this to `~/.config/fish/config.fish`:

```
if test -z $ASDF_DATA_DIR
set _asdf_shims "$HOME/.asdf/shims"
else
set _asdf_shims "$ASDF_DATA_DIR/shims"
end
if not contains $_asdf_shims $PATH
set -gx --prepend PATH $_asdf_shims
end
set --erase _asdf_shims
```

On Arch, the AUR package already installed the completions. Bash needs the `bash-completion` package for them to work, and Zsh needs `compinit` enabled in your `.zshrc`. Fish picks them up on its own.

If you want asdf's data somewhere other than `~/.asdf`, export `ASDF_DATA_DIR` before that `PATH` line. Most people never need to.

Open a new terminal and run `asdf version` to check that it's there. When I wrote this, the AUR package was at 0.20.0, two small releases behind 0.20.2. That gap is normal for the AUR, and everything in this post works on both.

Your first language: Node.js

asdf does nothing until you add a plugin. Start with Node:

```
asdf plugin add nodejs
```

The Node plugin downloads the official pre-compiled binaries from nodejs.org, so there's nothing to compile. Its docs list `curl`, `gawk` and `gpg` as dependencies on Debian. On Arch, `curl` and `gawk` are already there on almost any install, and `gpg` comes from the `gnupg` package.

See what versions exist, and which is the newest:

```
asdf list all nodejs
asdf latest nodejs
asdf latest nodejs 22
```

Now install two versions. `latest:22` means "the newest 22.x", resolved the moment you run it:

```
$ asdf install nodejs latest:22
Downloading node-v22.23.5-linux-x64.tar.gz...
-> https://nodejs.org/dist/v22.23.5/node-v22.23.5-linux-x64.tar.gz
Installing node-v22.23.5-linux-x64...
Installed node-v22.23.5-linux-x64 to /home/pablo/.asdf/installs/
nodejs/22.23.5

$ asdf install nodejs latest:24
Installed node-v24.22.3-linux-x64 to /home/pablo/.asdf/installs/
nodejs/24.22.3

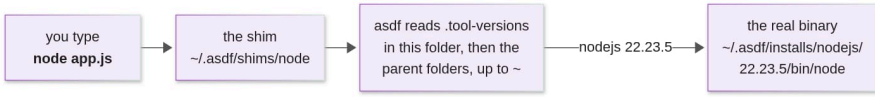
$ asdf list nodejs
22.23.5
24.22.3
```

Choosing a version

Installing a version doesn't activate it. If you try to run `node` now, asdf tells you so:

```
$ node --version
No version is set for command node
Consider adding one of the following versions in your config file at /home/
pablo/.tool-versions
nodejs 24.22.3
nodejs 22.23.5
```

First, set a default for everything under your home folder. The `-u` flag writes to `~/.tool-versions`:




```
$ asdf set -u nodejs 24.22.3
$ cat ~/.tool-versions
nodejs 24.22.3
$ node --version
v24.22.3
```

Then pin an older project to Node 22. Without a flag, `asdf set` writes a `.tool-versions` in the current folder:

```
$ cd ~/code/old-api
$ asdf set nodejs 22.23.5
$ cat .tool-versions
nodejs 22.23.5
$ node --version
v22.23.5
```

Step out of the folder and you're back on 24:

```
$ cd ~ && node --version
v24.22.3
```

`asdf current` shows which version is active and where that choice came from:

```
$ cd ~/code/old-api
$ asdf current
Name Version Source Installed
nodejs 22.23.5 /home/pablo/code/old-api/.tool-versions true
```

How the switch works

When you type `node`, the shell finds `~/.asdf/shims/node` first, because that folder is at the front of your `PATH`. The shim hands the call to asdf. asdf looks for a `.tool-versions` file in the current folder, then in each parent folder up to your home, reads the version it finds, and runs the real binary from `~/.asdf/installs`.

A few things follow from that:

- Nothing runs when you `cd`. The version is decided at the moment you run the command.
- A `.tool-versions` in a subfolder wins over one in a parent folder, and the one in your home is the fallback.

- An environment variable wins over all files. `ASDF_NODEJS_VERSION=22.23.5 node app.js` runs that one command on Node 22, wherever you are.
- `asdf set python system` hands a tool back to the version pacman installed.

To see the real path behind a shim, use `asdf which`:

```
$ asdf which node
/home/pablo/.asdf/installs/nodejs/22.23.5/bin/node
```

A second language: Python

Python is where asdf asks for more. The Python plugin uses pyenv's python-build, which compiles Python from source on Linux. So first install the build dependencies. On Arch, these are the ones pyenv lists:

```
sudo pacman -S --needed base-devel openssl zlib xz tk zstd
```

Then the usual steps:

```
asdf plugin add python
asdf list all python 3.14
```

And the install itself:

```
$ asdf install python 3.14.10
Downloading Python-3.14.10.tar.xz...
-> https://www.python.org/ftp/python/3.14.10/Python-3.14.10.tar.xz
Installing Python-3.14.10...
Installed Python-3.14.10 to /home/pablo/.asdf/installs/python/3.14.10
```

The build takes a few minutes, depending on your machine. When it's done, `python` follows the same rules as `node`.

Pick an exact version here. At the time of writing, `asdf latest python` answers with a version that ends in `t`, like `3.15.2t`. That `t` is the free-threaded build of Python, which runs without the GIL and isn't what most projects expect. `asdf list all python 3.14` shows both, and the one without the `t` is the regular build.

One file, several tools

A `.tool-versions` file can hold every language a project uses, one per line. Go, for example, has a plugin that downloads the official binaries:

```
$ cd ~/code/new-app
$ asdf plugin add golang
$ asdf set nodejs 24.22.3
$ asdf set golang 1.27.4
$ cat .tool-versions
nodejs 24.22.3
golang 1.27.4
```

Commit that file with the project. On a fresh machine, after adding the plugins, one command installs everything the file lists:

```
$ asdf install
version 1.27.4 of golang is already installed
version 24.22.3 of nodejs is already installed
$ go version
go version go1.27.4 linux/amd64
```

asdf also has an official GitHub Action that reads the same `.tool-versions`, so CI runs on the versions you use at home.

Coming from nvm, pyenv or rbenv

If your projects already have `.nvmrc`, `.node-version` or `.ruby-version` files, asdf can read them. It ignores them by default, so a project with only an `.nvmrc` still gets your home version:

```
$ cd ~/code/legacy
$ cat .nvmrc
22.23.5
$ node --version
v24.22.3
```

Turn the support on with one line in `~/.asdfrc`:

```
legacy_version_file = yes
```

And now the `.nvmrc` is respected:

```
$ node --version
v22.23.5
$ asdf current nodejs
Name Version Source Installed
nodejs 22.23.5 /home/pablo/code/legacy/.nvmrc true
```

Support for these files comes from each plugin. The Node plugin reads `.nvmrc` and `.node-version`, and the Ruby plugin reads `.ruby-version`.

Remove nvm or pyenv afterwards, including the lines they added to your shell config. Two managers fighting over the `PATH` is a classic source of "why is this the wrong version".

Commands worth remembering

- `asdf plugin add <name>`: add a language
- `asdf plugin list`: plugins you have
- `asdf plugin list all`: every plugin in the shortname list
- `asdf plugin update --all`: update every plugin
- `asdf list all <name>`: every version available
- `asdf latest <name> [prefix]`: newest stable version
- `asdf install <name> <version>`: install a version
- `asdf install`: install everything the `.tool-versions` file lists
- `asdf list <name>`: installed versions
- `asdf set <name> <version>`: pin a version in this folder
- `asdf set -u <name> <version>`: set the default in your home
- `asdf current`: what's active here, and why
- `asdf which <command>`: real path of a program
- `asdf where <name>`: install folder of the current version
- `asdf uninstall <name> <version>`: remove a version
- `asdf reshim <name> <version>`: rebuild the shims
- `asdf info`: debug info for bug reports

The shortname list has a little over 800 plugins. For a tool that isn't in it, `asdf plugin add <name> <git-url>` adds the plugin straight from its repository.

Keeping it updated, and removing it

asdf has no self-update command anymore. On Arch, it updates with the rest of your AUR packages:

```
yay -Syu
```

Plugins update separately, and it's worth running this once in a while, because new language versions show up through them:

```
asdf plugin update --all
```

`asdf reshim` rebuilds the shims. Run it when a command you just installed doesn't show up, which can happen after installing global packages with npm or pip.

To remove everything:

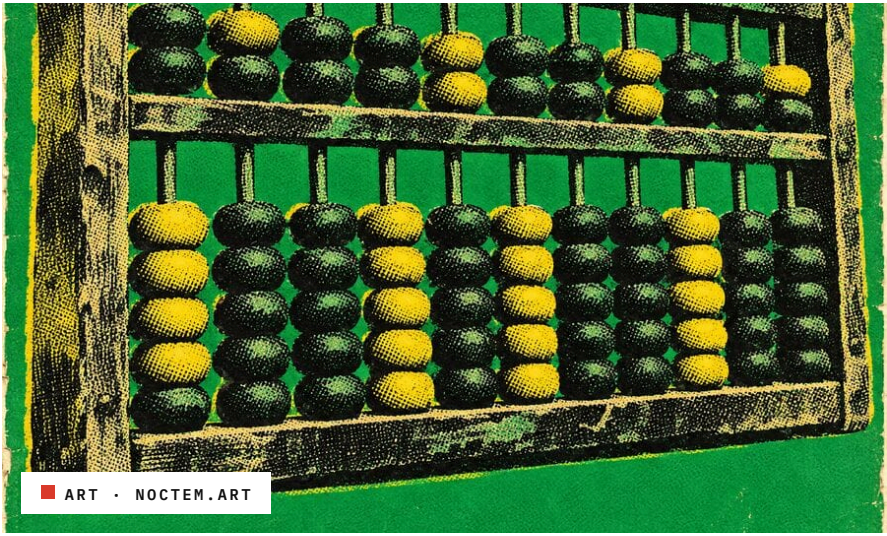
```
yay -Rns asdf-vm
rm -rf ~/.asdf ~/.tool-versions ~/.asdfrc
```

Then delete the shims line from your shell config.

Things to know before you start

- Versions are exact. A `.tool-versions` file never says “22” or “lts”. `latest` and `latest:22` exist only at install and set time, and they turn into a full number the moment you run them. That’s on purpose: the same file gives the same versions on every machine.
- Plugins are community projects, and their quality varies. The ones under `asdf-vm` and `asdf-community` on GitHub are the safest bet. A plugin is a set of scripts that runs on your machine, so look at the repository before adding one from a stranger.
- Languages that compile from source take a while. Node and Go download binaries in seconds, while Python, Ruby and Erlang build on your machine and need their build dependencies installed first.
- Old tutorials are out of date. Anything that tells you to source `asdf.sh` or to run `asdf global`, `asdf local`, `asdf shell` or `asdf update` was written before 0.16. Today it’s the `PATH` line, `asdf set` and your package manager.

There’s also mise, which does the same job, is written in Rust and reads `.tool-versions` files too. If you ever want to try it, your project files come along. ■



ART · NOCTEM.ART

06 SVG · CSS · WEB-DEVELOPMENT 17 NOVEMBER 2026

Perspective in one SVG path: the floor of pablo.space

The checkered floor at the bottom of pablo.space is a single SVG path inside the CSS. What SVG is, why turning a PNG into one is hard, how dividing by distance makes the perspective, and the two masks that fade it into the page.



Scroll to the end of any page on pablo.space and there's a checkered floor, black and cream tiles running off toward a horizon between the two columns. I found the drawing online and converted it to SVG. What's on the site now is one SVG drawing embedded in the CSS, with no external image and no JavaScript.

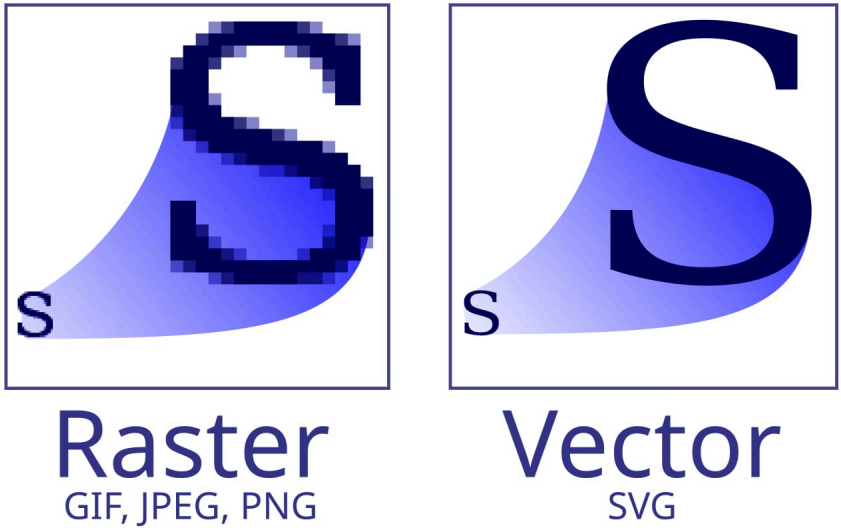
This post is about the two things that make it work: what an SVG actually is, and the small bit of math that makes a flat pattern look like it goes into the distance.

Pixels and shapes

A PNG or a JPG stores pixels. An SVG is a vector file: it describes how to draw the shapes. A circle in SVG is literally a line of text saying where the center is, how big the radius is and what color to fill it with. The browser draws it fresh at whatever size the screen needs, so it never gets blurry.

That's why converting a PNG to SVG is harder than it sounds. The program has to guess which shapes produced those pixels.

Think of a black circle on a white background. You glance at it and see a circle. The program gets a set of black pixels and, around the edge, a ring of grays that make it look smooth. It can recognize the geometry and write a single `<circle>`, or it can follow the outline pixel by pixel and



produce a curve full of tiny wobbles. Both results can look the same at the original size. The difference shows up when you zoom in or try to edit the circle.

How hard it gets depends a lot on the image:

- A simple logo with flat colors usually vectorizes well, especially if the original is large and sharp.
- A signature or an ink drawing: the program can follow the silhouette, but a thin or blurry stroke may come out thicker, broken or crooked.
- A photograph: skin, hair, shadows and reflections have delicate transitions. With few shapes it looks like a poster. With thousands of them the SVG gets heavy and hard to edit.

There's also a trap. It's easy to put a PNG inside a `.svg` file. It opens normally and looks the same, but it's still pixels, and it still gets blocky when you zoom in.

So for a logo you want to edit and use at any size, it's often better to redraw the few shapes that matter than to ask a vectorizer to keep every irregularity of the PNG. A checkered floor is the friendly case: every tile has four straight sides, and the whole thing can be described with a handful of numbers.

Perspective is a division

The rule that makes a floor look like it's moving away is simple: the size of something on screen is its real size divided by its distance. Call the distance `z`. The floor uses that rule twice.

The height of each row on screen:

```
y = horizon + (height - horizon) / z
```

With `z = 1` the line falls on the bottom edge of the drawing. With `z = 7` it climbs almost to the horizon.

The width of a tile:

```
width = 132 / z
```

Up close a tile is 132 units wide. In the back row it's 19.

The rows sit at these distances:

```
7, 5.4, 4.2, 3.3, 2.6, 2.05, 1.62, 1.28, 1
```

Each one is the next divided by about 1.28. Because the steps multiply instead of adding, the rows get thinner and thinner as they approach the horizon, which is exactly what a real floor does when you look across it.

Every tile is a trapezoid

A tile sits between two rows. Its top edge uses the width of the farther row and its bottom edge uses the width of the nearer one, so the sides lean toward the middle on their own. The vanishing point is never drawn. It shows up as a consequence of the numbers.

This is one real tile from the file, the one just right of the center in the back row:

```
M800 16.3 H818.9 L824.4 22.9 H800 Z
```

Read it like instructions for a pen. `M` moves to the top left corner at (800, 16.3). `H` draws a horizontal line to 818.9, so the top edge is 18.9 units wide, which is 132 divided by 7. `L` draws a slanted line down to (824.4, 22.9), the next row. `H` goes back to 800, and that bottom edge is 24.4 wide, 132 divided by 5.4. `Z` closes the shape. Four corners, and the slant comes for free.

The checkerboard comes from skipping. Only the dark tiles exist in the SVG, alternating from row to row. The "light" tiles are just the cream

color of the page showing through. And every dark tile goes into a single `<path>`, one long string of those `M`, `H`, `L`, `Z` commands, which keeps the whole drawing at a few kilobytes.

The CSS places it and fades it

The SVG goes into the CSS as a `url("data:image/svg+xml,...")`, the background of `html::after`. That's a 150px block the browser creates after all the other content on the page, which is why the floor only appears at the very end.

The fade is two masks crossed with `mask-composite: intersect`, so a pixel only shows where both masks let it through:

- a vertical one, going from transparent at the top to opaque at the bottom, so the far end of the floor dissolves into the paper;
- a horizontal one, which fades the sides out starting 250px from the center, so the floor doesn't run over the bases of the two columns.

```css
mask-image:
linear-gradient(to bottom, transparent 4%, #000 70%),
linear-gradient(to right, transparent calc(50% - 430px), #000 calc(50% -
250px), #000 calc(50% + 250px), transparent calc(50% + 430px));
mask-composite: intersect;
```

That's all of it: a few numbers divided by a distance, one path, and two gradients. Go to the bottom of pablo.space and stand on it for a second. ■

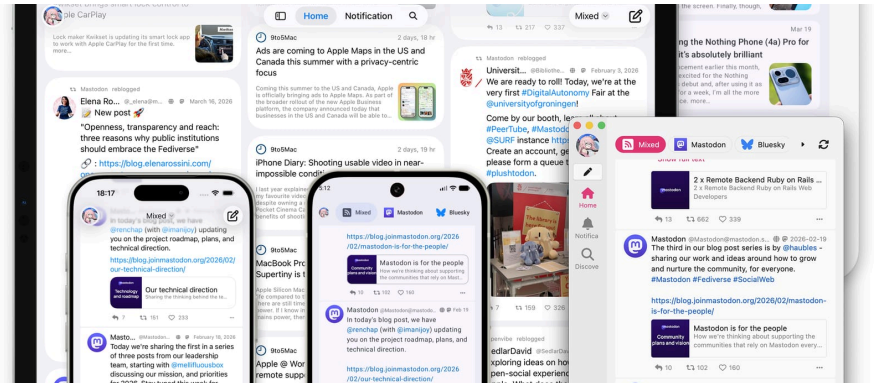


ART · NOCTEM.ART

07 FEDIVERSE · BLUESKY · MASTODON 17 NOVEMBER 2026

Review: Flare puts Bluesky and the fediverse in one timeline

I spent tonight looking for one app for Bluesky and the fediverse, and found Flare. It's free, open source, and mixes Mastodon, Misskey, Bluesky, Nostr and RSS in a single timeline, on phones and on the desktop. The developers did an excellent job.



This post isn't sponsored, and I'm not affiliated with Flare in any way. I don't get anything for writing it.

Tonight I went looking for one app that could handle both Bluesky and the fediverse, so I'd stop jumping between tabs. I found Flare, tried it, and I'm impressed. It's spectacular.

Flare is a social client that brings several networks into the same place: Mastodon, Misskey, Bluesky, X and Nostr, plus RSS feeds. You add your accounts and it can merge everything into a single mixed timeline, or keep each network in its own tab. From there you can post to more than one network at once, and it keeps a local history so you can find something you scrolled past earlier.

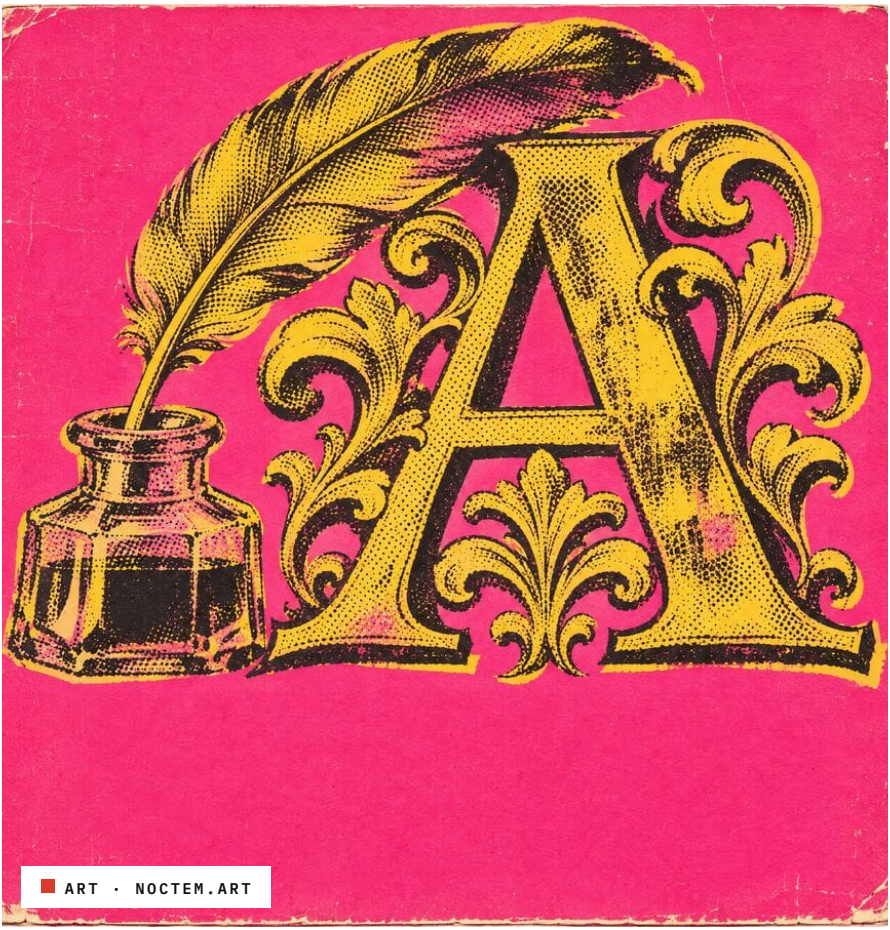
It also works as a full RSS reader, so blogs and news sites can sit in the same timeline as the social posts.

It runs almost everywhere: Android (on Google Play and F-Droid), iOS, Windows, macOS and Linux as an AppImage. There's also a web version in beta, for now only with Mastodon, Misskey and Bluesky. The whole app is built with Kotlin Multiplatform, and the layout adapts well to big screens.

And it's free software. No waitlist, no subscription, and the source code is on GitHub under the AGPL.

The developers did an excellent job. Building one client for so many networks, each with its own rules and quirks, on five platforms, and making it feel this polished, takes a lot of care. Thank you.

If you use both Bluesky and the fediverse, give Flare a try. ■



ART · NOCTEM.ART

08 POETRY · BOOKS · BRAZIL 17 NOVEMBER 2026

Emmanuel Santiago, the poet of birds

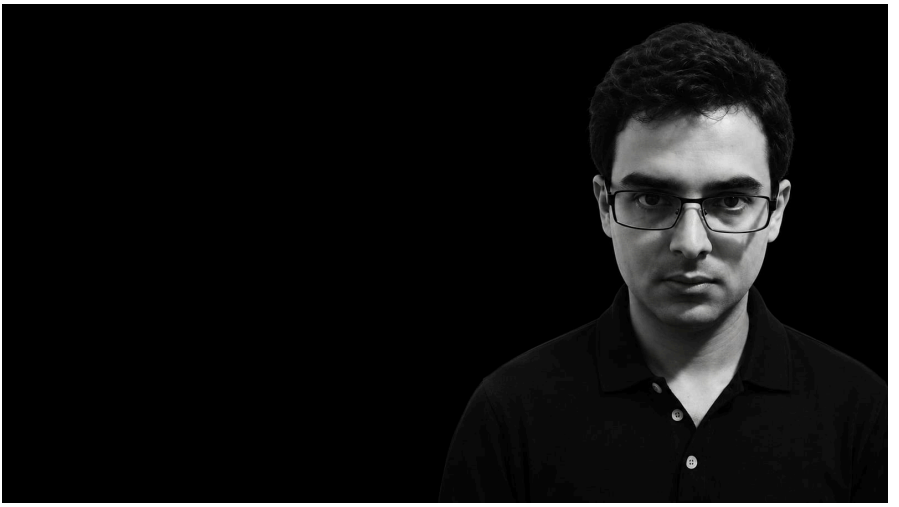
I met Emmanuel Santiago in a cyber café in São Lourenço, in 2002. He became a poet, a literature teacher and one of my best friends. He died in April 2026. This is my memory of him, and a reading of the four books he left, from the peacock to the hummingbirds.



I met Emmanuel when I was still a teenager: an awkward kid, covered in pimples, with a curiosity that never seems to have let go of me. It must have been around four in the afternoon on some Tuesday in 2002. I remember the place well: the old Cyber Café in São Lourenço.

Back then, internet access was still a rare thing. Cyber cafés were everywhere, rooms full of computers where you paid to spend a little time online and, almost always, to drink bad coffee. Between the machines there were tables full of people playing Magic: The Gathering and RPGs, talking for hours about imaginary worlds.

That's where I saw him for the first time. There were three guys. One had long hair and looked like a rocker. Another had a strong presence. And there was a third one, thin, odd, a bit out of place, as if he didn't quite match the other two. That was Emmanuel. Of course, his was the strongest voice, the sharpest criticism and the widest knowledge.



Emmanuel Santiago (1984-2026).

I walked up to them, and the friendship came fast, the way some friendships do when you recognize something in the other person before you even know what to call it.

For the next three years, Emmanuel and I were almost inseparable. I was always amazed at how much he read, how much he knew, and the passion he put into talking about ideas, books, authors, symbols and whole worlds. Believe me: a good part of the hungry reading habit I still carry today was born from that friendship.

Years later, Emmanuel would become a writer and a literature teacher. But before that, he was my friend. And as a friend, he taught me things I still carry with a deep, boundless affection. He taught me theosophy, demonology, occultism, Brazilian literature, classical literature, Russian literature, the Gothic movement, movements from here and from there. And I remember very well how much I loved hearing him explain who the Marquis de Sade was. Man, I learned a lot.

But life doesn't stop. It goes on its way, indifferent to how much we want to stay close to the people we love.

Today I dedicate this text to a great friend who is gone. A friend I talked to the day before his last surgery. A friend who had made it clear, on Facebook, that he wasn't very confident he'd come back.

Who the poet was

Emmanuel Santiago was born in São Lourenço, in the south of Minas Gerais, on April 13, 1984. He graduated in Letters from the Federal University of Ouro Preto in 2006 and did his master's and doctorate at the University of São Paulo. He taught literature to high school students at the Instituto Alpha Lumen, in the Paraíba Valley, published four books of poetry, and scattered poems and translations across magazines, including the Revista Brasileira, the journal of the Brazilian Academy of Letters. In 2018 he spoke from the Academy's own lectern about Olavo Bilac and the Parnassian city, in a lecture series coordinated by Antonio Carlos Secchin.

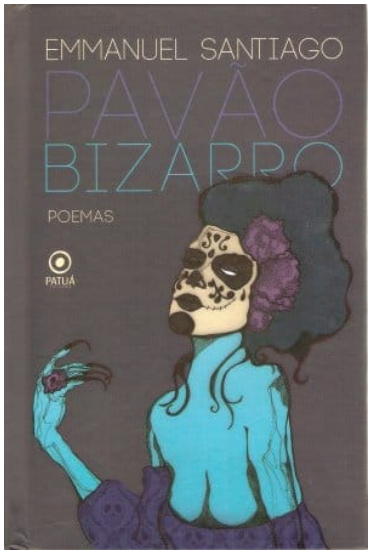
His colon cancer was found in early 2024. Students and friends ran campaigns to help pay for the treatment. He had been in intensive care since March 2026, and he died in Jacareí on April 23, ten days after turning 42. It was World Book Day, the date Shakespeare died too, a poet Emmanuel translated: he turned sonnets from English into twelve-syllable lines, published in the Revista Brasileira.

The critic came first

Before the books of poems came the professional reader. His master's thesis, defended in 2010, is called *A narração dificultosa* (The difficult narration), and it studies "Cara-de-Bronze", one of the novellas in Guimarães Rosa's *Corpo de baile*, a text famous for handing its story to the reader in pieces. To read it, he turned to Hans Robert Jauss's reception aesthetics, the theory that puts the reader at the center of meaning. The work came out as a book in 2014, in the Produção Acadêmica Premiada series.

His doctorate, from 2016, is *A musa de espartilho: o erotismo na poesia parnasiana brasileira* (The corseted muse: eroticism in Brazilian Parnassian poetry). The central idea is beautiful and simple. A fixed form works like a corset, which squeezes the body and, by squeezing it, makes it erotic. With that key he reread Raimundo Correia, Alberto de Oliveira, Luiz Delfino and Francisca Júlia, whose famous "marble coldness" he read as a strategy, against the tradition that saw nothing there but coldness.

That thesis is also the best way into his poetry. Emmanuel wrote sonnets, villanelles, sestinas, ballades, tankas and haiku. He liked rhyme, including assonant rhyme, the kind that matches only the vowels. And in almost everything he published, the form squeezes, and it's inside that squeeze that desire, humor and anger show up.



Pavão bizarro (Patuá, 2014), his first book.

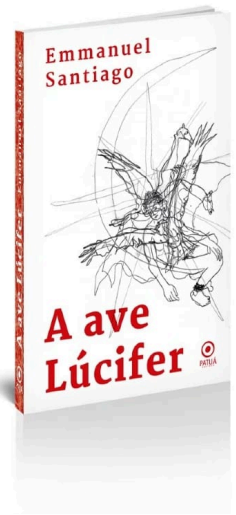
Pavão bizarro (Patuá, 2014)

His debut, whose title means "bizarre peacock", gathers 39 poems written over ten years. The most visible shadow is João Cabral de Melo Neto, with his idea of the poem as a built object, measured, never spilling over. T. S. Eliot is around too. The title sums up the book well: the peacock is the bird that turns its own plumage into a show, and the adjective warns you the show is going to come out crooked.

Two moments show the project. "A fábula de Fabergé" (The fable of Fabergé) brings the czars' jeweled eggs into the book, objects that exist only to show off craft, and it's hard not to see in them a portrait of the poem as a piece of goldsmithing. And the duet that sets João Gilberto and João Cabral side by side joins two masters of economy, one of the almost whispered voice, the other of the dry line. For a first book, it's a clear statement of principles: rigor, concision and an open taste for artifice.

A ave Lúcifer (Patuá, 2020)

Six years later came his most ambitious book, *A ave Lúcifer*, "the Lucifer bird". Here Emmanuel leaves the short poem behind and sets out for long, narrative forms, with an eye on Eliot, on Poe and on Octavio Paz's thoughts about the long poem. The muse has something of Baudelaire,



■ *A ave Lúcifer (Patuá, 2020).*

and the humor became at once more erotic and more bitter. The title is already a little lesson in the demonology he liked to explain: Lucifer means "light-bearer", and it was the Latin name of the morning star before it became the name of the fallen angel.

The fourth part is the most political, and the one that shows his method best. Instead of commenting on the news, Emmanuel dressed the news in an old form. The fire at the National Museum in Rio, in 2018, becomes "Queimemos de Alexandria", which ties the fire in Rio to the lost library of antiquity. Torture shows up in a sestina about Saint Anthony, with an epigraph by Alberto da Cunha Melo taken from the poem he wrote for Vladimir Herzog, the journalist killed under torture by the military dictatorship. The sestina is an obsessive form: the same six words come back at the end of the lines, stanza after stanza, and few forms could fit a torment that repeats itself better. "Havendo paz, eu me rebelo" (If there's peace, I rebel) is a fake sirventes, the satirical genre of the Provençal troubadours, written in the manner of Bertran de Born, the troubadour who sang of war.

In interviews he said poetry doesn't transform reality, but it can denounce barbarism, as long as it admits its own powerlessness. And he said he used allegory so his poems wouldn't stay stuck to the moment. The book's titles show the clash this produces: "Álbum de férias em Abu



A renga do corvo (Bestiário, 2023).

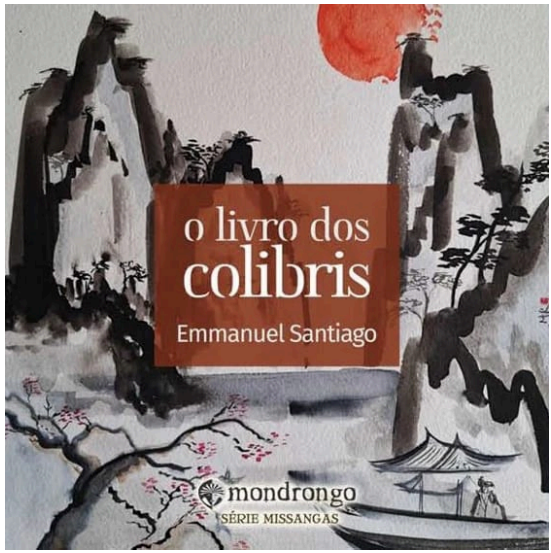
Ghraib" (Vacation album from Abu Ghraib), "Orfeu desce à Cracolândia" (Orpheus goes down to Cracolândia, São Paulo's crack district), "Legenda de Robert Johnson" (Legend of Robert Johnson). The prison in Iraq becomes a travel souvenir, the Greek myth goes down into downtown São Paulo, and the bluesman who supposedly sold his soul to the devil at a crossroads gets a legend, the word that in the Middle Ages named the lives of the saints.

The book closes with "A rola" (The turtledove), a parody of Edgar Allan Poe's "The Raven". The raven stayed perched on his head, and three years later it came back whole.

A renga do corvo (Bestiário, 2023)

"The Raven" had already passed through the hands of Machado de Assis and Fernando Pessoa, among many other translators. Emmanuel did something else with it. In 94 pages, *A renga do corvo* (The raven's renga) turns Poe's poem into a renga, the chain of poems from the Japanese tradition, made here of tankas, five-line stanzas. And he put rhyme where the Japanese don't, in a scheme of his own, aba/bB.

He called the book a platypus himself, an animal that's neither a real renga nor a translation. The comparison is a good one. A traditional renga is written by several poets taking turns, and here a single poet talks



■ *O livro dos colibris (Mondrongo, 2025), a book of haiku.*

with Poe stanza by stanza, as if he were the second player at the table. Poe liked to say, in "The Philosophy of Composition", that he built "The Raven" step by step, like someone solving a math problem. A poet who spent his doctorate studying what form does to a text could hardly find a better partner.

O livro dos colibris (Mondrongo, 2025)

His last book, *O livro dos colibris* (The book of hummingbirds), came out in 2025, in Mondrongo's Série Missangas, while he was already ill. It's haiku. His interest in Japanese forms went way back, to senryū published in magazines and to the renga itself, but here he went all the way down that road. The poet who said he wanted to write a long lyric poem published, last of all, one of the shortest forms there is. And he chose the hummingbird for it, the smallest of birds, which beats its wings so fast it seems to hang still in the air.

An aviary

Four books, four birds: the peacock, the Lucifer bird, the raven and the hummingbirds. Read in order, they trace a path from plumage on display to the smallest flight, from the Cabral-style poem to haiku. What doesn't change is the craft. Emmanuel treated fixed form as a place of freedom,

and he did it with learning, humor and an anger that never turned into a pamphlet.

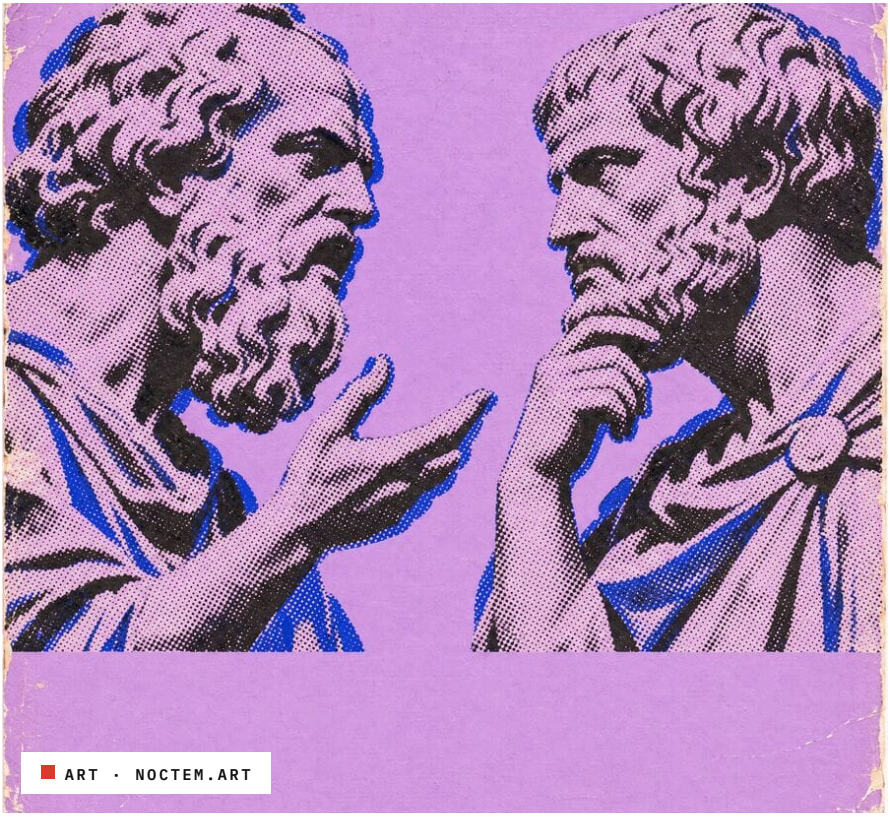
The poet João Filho, from Bahia, called him "one of the greatest contemporary poets". Besides the books, he left an essay in the collective volume *O que restou de 22* (Sétimo Selo, 2022), the texts he published on Substack, the poems scattered across magazines, his Shakespeare translations, and a lot of students who learned literature from him.

Well, Emmanuel. I remember. I remember all of it. And today, especially today, I'm going to remember you.

Emmanuel was a writer. He was a poet. And, somehow, he left this world the way poets tend to: with an absence full of words.

Goodbye, Emmanuel.

I first wrote this in Portuguese, on noctem.art.



ART · NOCTEM.ART

09 EMAIL · PRIVACY · SECURITY 17 NOVEMBER 2026

Email only the right person can read

Email travels like a postcard. GPG seals it so only the recipient can read it. What PGP, OpenPGP and GnuPG are, how keys and signatures work, the common myths, and how to set it up on Windows, Linux and macOS, plus Thunderbird on all three.



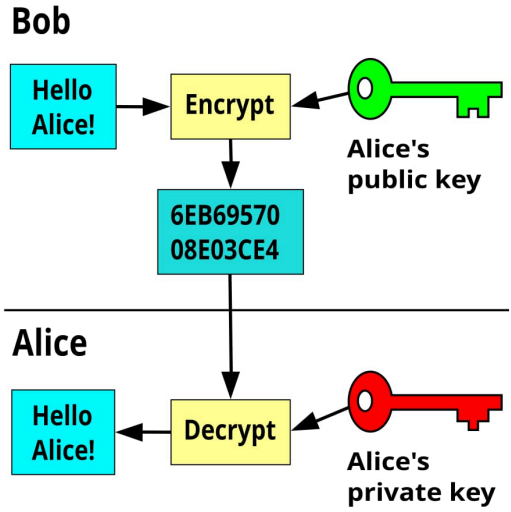
An email travels more like a postcard than a sealed letter. It leaves your computer, passes through your provider's servers, crosses the internet to the recipient's provider and sits in a mailbox there, often for years. The connections along the way are usually encrypted, but at every stop the message is stored as plain text, readable by whoever runs that server, by anyone who breaks into it, and by anyone with a court order.

GPG fixes the part that matters most: the content. With it, a message is locked on your computer and only opens on the recipient's. Everyone in between carries a sealed box.

This guide covers what it is, how it works, the myths around it, and how to set it up on Windows, Linux and macOS.

PGP, OpenPGP, GnuPG: one idea, three names

In 1991 Phil Zimmermann wrote PGP, short for Pretty Good Privacy, and released it with its source code. At the time the US treated strong



cryptography almost like weapons, and Zimmermann spent three years under criminal investigation for "exporting" it. The case was dropped in 1996 without charges.

PGP's message format became an open standard called OpenPGP, so that any program could implement it. In 1999 Werner Koch released GnuPG 1.0, the free implementation that most people use today, usually through the `gpg` command. When someone says "PGP email" or "GPG email", they mean the same thing: messages in the OpenPGP format.

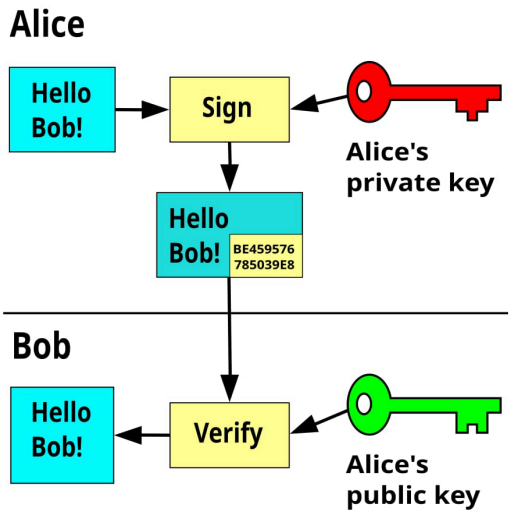
The current release is GnuPG 2.5.24, from September 2026. In 2024 the standard split into two branches, RFC 9580 and LibrePGP, and GnuPG follows LibrePGP. For everyday email with the default key types this doesn't change anything. Keys made with the defaults still work across programs.

How it works

Everything rests on a pair of keys that are made together and are mathematically linked.

The **public key** is for sharing. You can put it on your website, in your email signature, on a key server. With it, anyone can lock a message that only you can open.

The **private key** stays with you, protected by a passphrase. It's the only thing that opens what was locked with your public key.



So if Bob wants to write to Alice, he encrypts with Alice's public key, and only Alice's private key can read the result. Bob himself can't decrypt what he just sent. Under the hood, GPG encrypts the message with a random one-time key and then locks that small key with Alice's public key, which keeps things fast even for big attachments.

The same pair works the other way for **signatures**. Alice signs with her private key, and anyone with her public key can check two things: that the message came from her key, and that not a single character changed on the way.

Encryption keeps a message private. A signature proves who wrote it. Most of the time you'll want both.

Each key also has a **fingerprint** , a 40-character code that identifies it. That's what you compare when you want to be sure a public key really belongs to the person you think it does. Here is a key created for this post, for an imaginary Alice:

```
$ gpg --fingerprint alice@example.org
pub ed25519 2026-11-15 [SC] [expires: 2028-11-14]
4775 63FB D445 7E52 8701 9D86 9A5F 955D 59BC F65E
uid [ultimate] Alice Example <alice@example.org>
sub cv25519 2026-11-15 [E]
```

The `pub` line is the main key, used to sign (S) and certify (C). The `sub` line is a subkey used to encrypt (E). Modern GnuPG creates this pair of elliptic curve keys by default.

What it protects, and what it doesn't

GPG protects the body of the message and the attachments. Some programs, Thunderbird among them, also hide the subject line.

It doesn't hide who wrote to whom, when, from which server, or how big the message was. Mail servers need that information to deliver anything, so it travels in the open. If the fact that you talk to someone is the secret, email isn't the right tool, encrypted or not.

Why bother

Your provider can read your mail. Most do it for spam filtering, some have done it for ads, and any of them can be ordered to hand it over. With GPG, what sits on their disks is a block of text that only opens on your machine.

Breaches happen. Years of stored email in a hacked account are a gift to whoever gets in. Encrypted messages in that same account are useless to them.

Signatures stop impersonation. Faking the "From" line of an email is trivial. Faking a valid signature from someone's key isn't.

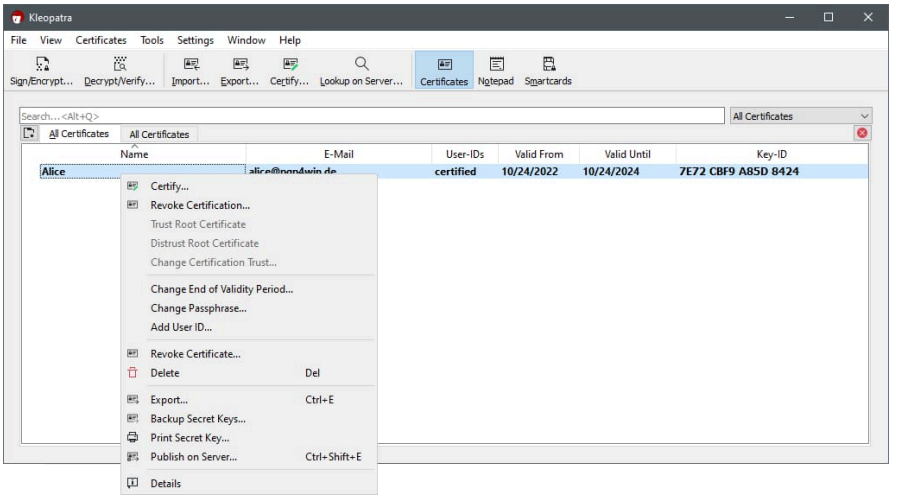
It also goes beyond email. Linux distributions sign their packages with OpenPGP keys, and your package manager checks those signatures every time you install something. Many projects sign their releases the same way. Learning GPG for email teaches you how to check that the software you download is what its authors published.

Myths

"It's only for spies and criminals." You're already using it, indirectly, every time `apt` or `dnf` installs a package. Journalists, lawyers and software developers use it for ordinary, everyday reasons.

"My email already uses HTTPS, so it's encrypted." HTTPS and TLS protect the connection between machines. The message is decrypted at each server along the way and stored readable. GPG protects the message itself, from end to end.

"PGP was broken." In 2018 researchers published EFAIL, a set of attacks that could leak the content of encrypted messages. The flaw was in how some email programs handled HTML and remote content, not



in the encryption. The affected programs were patched. The lesson that stays: read encrypted mail with remote content turned off.

"Encryption hides everything." As above, the sender, recipient, date and usually the subject stay visible unless your program hides the subject.

"You need to be a command line wizard." Every system below has a graphical way to do it. The terminal is optional.

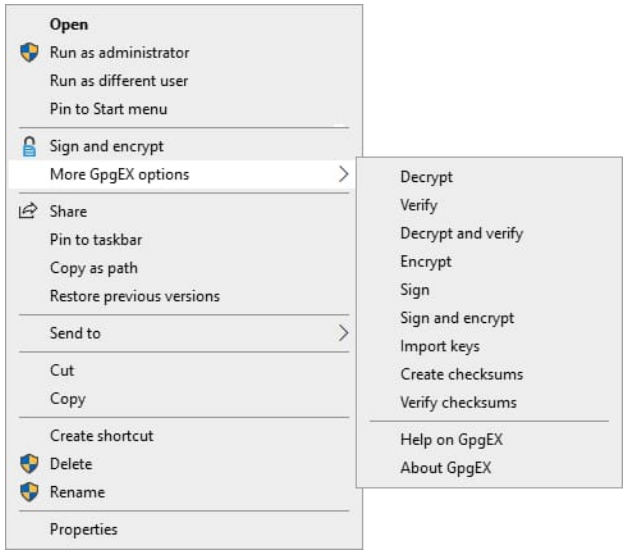
"If a key is on a key server, it belongs to that person." Anyone can create a key with any name and email. keys.openpgp.org at least confirms the email address before publishing it, but the safe habit is to compare fingerprints with the person through another channel.

Windows: Gpg4win and Kleopatra

On Windows, install Gpg4win. The current version, 5.1.1, from September 2026, ships GnuPG 2.5.24 and a few tools on top of it.

The one you'll use the most is **Kleopatra** , the key manager. Open it and create a new OpenPGP key pair from the File menu. It asks for your name and email, then for a passphrase. Pick a long one you won't forget, because there's no way to recover it.

Right-click your key and you'll find what you need next: **Export** saves your public key to send to people, **Backup Secret Keys** saves your private key for safekeeping, and **Publish on Server** sends the public key to a key server. To use someone else's key, save their `.asc` file and import it.



Gpg4win also adds **GpgEX** to Windows Explorer. Right-click any file and you can encrypt, sign, decrypt or verify it without opening another program.

For email, there are two paths. If you use classic Outlook, Gpg4win installs **GpgOL** , which adds encrypt and sign buttons to it. For the new Outlook, the add-in is still marked experimental. The other path, which works the same on every system, is Thunderbird, covered further down.

Linux: it's probably already there

Most Linux distributions come with GnuPG installed, because the package manager depends on it. Check with:

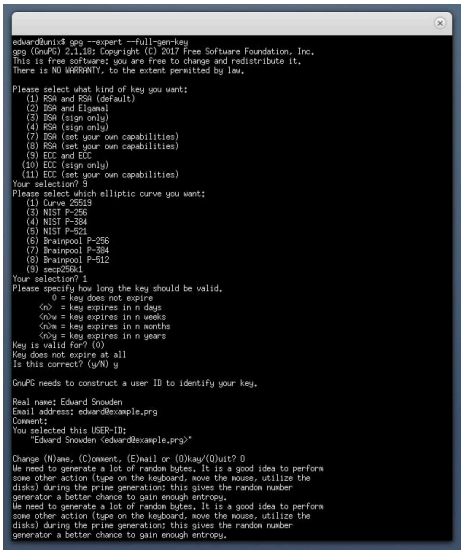
```
gpg --version
```

If it's missing, it's in every repository as `gnupg` (`sudo apt install gnupg`, `sudo dnf install gnupg2`, `sudo pacman -S gnupg`).

Create your key with:

```
gpg --full-generate-key
```

It asks what kind of key you want, how long it should last, your name, your email and a passphrase. The defaults are good choices. It's worth



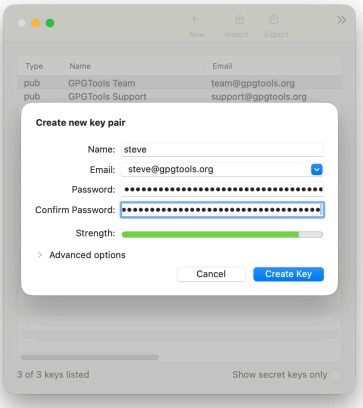
giving the key an expiration date, two years for example, because you can extend it later whenever you want.

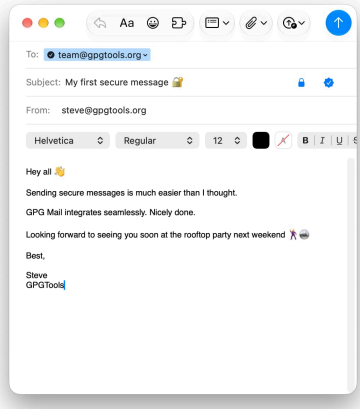
When it finishes, GnuPG also saves a revocation certificate in `~/.gnupg/openpgp-revocs.d/`. That file lets you mark your key as dead if you ever lose it or it gets stolen. Copy it somewhere safe, away from the computer.

These are the commands you'll use most. They work the same on Windows and macOS, in the terminal:

```
gpg --list-keys: list the public keys you have
gpg --list-secret-keys: list your own private keys
gpg --fingerprint alice@example.org: show a key's fingerprint
gpg --armor --export you@example.org > you.asc: export your public key to
share
gpg --import friend.asc: import someone's public key
gpg --armor --encrypt -r friend@example.org file.txt: encrypt a file for
someone
gpg --decrypt file.txt.asc: decrypt something sent to you
gpg --clearsign note.txt: sign a text and keep it readable
gpg --verify note.txt.asc: check a signature
gpg --armor --export-secret-keys you@example.org > backup.asc: back up your
private key
```

`--armor` means the output is text instead of binary, which is what you want for pasting into an email. A message encrypted this way looks like this:






```
-----BEGIN PGP MESSAGE-----

hF4DBaEpbwzwuj0SAQdAJ06jB5TO+6YEg6DGVWcbR2maDOmonN7SkJ6Tr6HA/zUw
cQe65A4W+HPDnRac9lS7U26gfxNUVXKzE/k1TuzK9Cl5Td9U9sYNpZctxSEAIYyS
...
```

And a signed one keeps the text readable, with the signature attached below:

```
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA512

Meet me at the usual place.
-----BEGIN PGP SIGNATURE-----
...
-----END PGP SIGNATURE-----

$ gpg --verify note.txt.asc
gpg: Good signature from "Alice Example <alice@example.org>" [ultimate]
```

If you'd rather click, KDE has Kleopatra (the same program as on Windows) and GNOME has Seahorse, which appears as "Passwords and Keys".

macOS: GPG Suite or Homebrew

On the Mac the friendliest option is GPG Suite. It includes **GPG Keychain**, a free key manager where you create a key with a name, email and password, and import or export keys by dragging files.

The suite also has **GPG Mail** , which adds encryption and signing to Apple Mail. That part is paid: it works with a yearly subscription after a 30-day trial, and at the time of writing the version for the newest macOS is still in beta.

If you prefer the terminal and free tools, install GnuPG with Homebrew and use Thunderbird for email:

```
brew install gnupg pinentry-mac
```

`pinentry-mac` is the small window that asks for your passphrase. After that, all the commands from the Linux section work the same.

Thunderbird, on any system

Thunderbird has OpenPGP built in since version 78, released in 2020. It doesn't need GnuPG installed: it brings its own OpenPGP library, called RNP, and keeps its own keys. If you already have a key from Kleopatra, GPG Keychain or the terminal, you import it.

To set it up, open Account Settings, choose your account and go to **End-To-End Encryption**. Click **Add Key** and either create a new OpenPGP key or import an existing one, then select it for that account.

When you write a message, the Encrypt button in the toolbar locks it for every recipient whose key you have. Turning on encryption also turns on signing, and by default Thunderbird encrypts the subject too and attaches your public key to signed messages, so people can write back encrypted.

One detail that confuses people: when you import someone's key, Thunderbird asks whether you **accept** it. Accepting means "I'm willing to encrypt to this key". Do it after you've compared the fingerprint with the person, not before.

Habits that keep it working

- Back up your private key and the revocation certificate, offline: a USB stick in a drawer, a printed copy, whatever you trust. Without the private key, every message ever encrypted to you is gone for good.
- Use a long passphrase. If someone copies your private key file, the passphrase is all that stands between them and your mail.
- Set an expiration date and extend it when it gets close. It's a safety net in case you lose access to the key and can't revoke it.
- Compare fingerprints in person, by phone or through another app you already trust before you accept someone's key.

- Publish your public key where people can find it. keys.openpgp.org sends you a confirmation email before it shows your address. If you run your own domain, WKD (Web Key Directory) lets email programs find your key automatically from your domain.

My own public key is on keys.openpgp.org, with this fingerprint:

```
7C3A 051F 5EA9 9005 FC13 7882 4E7F 9C51 4F1D AC87
```

Start with one person you write to often. Exchange keys, compare fingerprints, send a signed message, then an encrypted one. After the first few, it stops feeling like a ritual and becomes one more button you press. ■

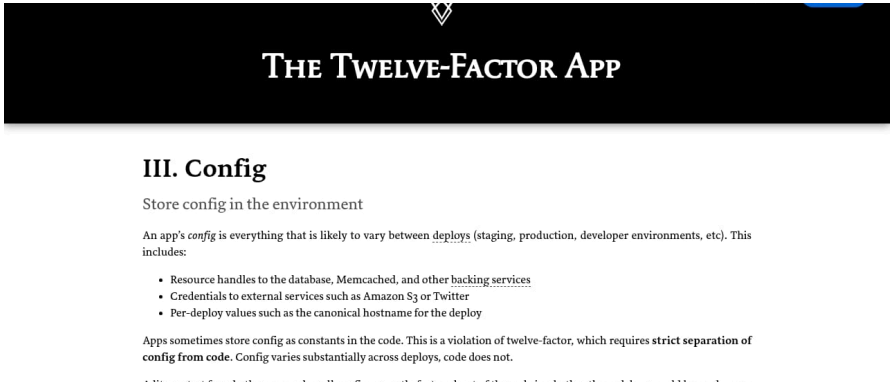


ART · NOCTEM.ART

10 DOTENV · ENVIRONMENT-VARIABLES · WEB-DEVELOPMENT 18 NOVEMBER 2026

The .env file: what it is, where it came from and how to use it well

That little .env file at the root of almost every project: what environment variables are, why config lives there, how Foreman and a 2012 Ruby gem started it all, how to use it in Node, Python, Ruby, PHP and Docker, and the ways it leaks.


THE TWELVE-FACTOR APP

III. Config

Store config in the environment

An app's *config* is everything that is likely to vary between deploys (staging, production, developer environments, etc). This includes:

- Resource handles to the database, Memcached, and other backing services
- Credentials to external services such as Amazon S3 or Twitter
- Per-deploy values such as the canonical hostname for the deploy

Apps sometimes store config as constants in the code. This is a violation of twelve-factor, which requires **strict separation of config from code**. Config varies substantially across deploys, code does not.

Open almost any modern project and you'll find a small file at the root called `.env`. It's plain text, it doesn't show up in a plain `ls` because of the dot, and it often holds the most sensitive things in the whole project: database passwords, API keys, tokens. Most people learn to use it by copying someone else's, without ever hearing where it came from or why it looks the way it does. So let's start from the ground up.

First, what is an environment variable?

Every program running on your computer is a process, and every process gets a small list of name and value pairs when it starts. That list is its environment. It's inherited: when your shell starts a program, the program gets a copy of the shell's environment, and if that program starts another one, the copy keeps going down the chain.

You can look at yours right now:

```
env
echo $HOME
echo $PATH
```

`HOME` tells programs where your home folder is. `PATH` tells the shell where to look for commands. Nobody wrote those values into the programs. The programs read them from the environment when they need them.

Your own code can do the same, and every language has a way in:

```
# Python
import os
db_url = os.environ.get("DATABASE_URL")

// Node.js
const dbUrl = process.env.DATABASE_URL;

# Ruby
db_url = ENV["DATABASE_URL"]

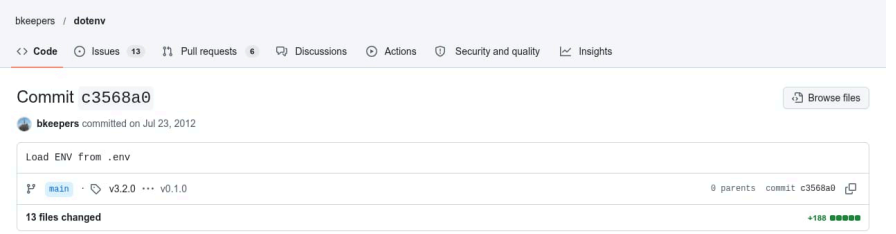
// PHP
$dbUrl = getenv('DATABASE_URL');
```

To set a variable for a single command, write it before the command: `DATABASE_URL=postgres://localhost/myapp python app.py`. To set it for the rest of the terminal session, use `export DATABASE_URL=postgres://localhost/myapp`.

Why configuration lives there

Around 2011, developers at Heroku published The Twelve-Factor App, a short set of rules for building web apps that behave well on servers. The third factor, Config, says that everything that changes between your laptop, a test server and production (database addresses, credentials for outside services, the hostname of each deploy) belongs in environment variables, kept apart from the code.

The test they propose is a good one: could you make your code open source right now without leaking a single credential? If a password is written inside a `config.py`, the answer is no. If the code reads it from `os.environ`, the answer is yes, and the same code runs everywhere, with each machine supplying its own values. The text also explains why environment variables beat config files: they're hard to commit to a repository by accident, and they work the same way in every language and operating system.



The problem the .env file solves

The twelve-factor rule works beautifully on a server, where the platform sets the variables for you. On your own machine it gets old fast. The project needs eight variables, so every new terminal means typing eight `export` lines. Then you open a second project that uses the same names with different values.

The .env file is the shortcut. You write the variables once, in a file next to the code, and a small library reads that file when the app starts and puts each line into the process environment. The code itself doesn't change. It keeps reading `os.environ` or `process.env`, with no idea where the values came from.

Where it came from

Heroku had a tool for running apps on your own machine called Foreman, and it already did this: if a `.env` file existed in the current folder, Foreman read the environment from it before starting the app. The catch was that it only worked for processes started through Foreman. A Rails console or a rake task opened by hand didn't see any of it.

In July 2012 a Ruby gem called dotenv appeared to close that gap, with a first commit titled "Load ENV from .env". Its README explains the problem in those terms: Foreman loads `.env` for what's in your Procfile, but things get difficult when you want a console or a rake task.

The whole parser in that first version was one line of Ruby:

```
self[$1] = $2 if line =~ /\A([\w_]+)=(.*)\z/
```

Read each line, and if it looks like `NAME=value`, keep it. Everything that came later (quotes, comments, multi-line values, variable expansion) was built on top of that one idea.

It spread fast. The dotenv package for Node.js was first published on npm in July 2013, and ports followed for Python, PHP, Go, Rust and pretty

much everything else. Then the runtimes started doing it themselves. Node.js 20.6.0, in 2023, added a `--env-file` flag, and Node.js 21.7.0, in March 2024, added `process.loadEnvFile()`. Bun reads `.env` files on its own, and Docker Compose reads the `.env` in the project folder to fill in the variables of your compose file.

What goes inside

```
# Database
DATABASE_URL=postgres://app:devpassword@localhost:5432/myapp

# Outside services
STRIPE_SECRET_KEY=sk_test_123
MAIL_FROM="My App <hello@example.com>"

# Settings
DEBUG=true
PORT=3000
```

The rules most libraries agree on:

- One variable per line, written as `NAME=value`.
- No spaces around the `=`. `PORT = 3000` breaks in some parsers and in the shell.
- Names in uppercase with underscores. It's a convention, and everybody follows it.
- Lines that start with `#` are comments.
- Quote values that have spaces or special characters. Double quotes usually accept escapes like `\n` and, in some libraries, references to other variables like `${HOME}`. Single quotes keep the text exactly as written, which is what you want for a password with a `$` in it.
- Everything is a string. `PORT=3000` reaches your code as the text "3000", and `DEBUG=false` as the text "false".

That last one bites people. In Python, `if os.environ.get("DEBUG"):` is true even when the file says `DEBUG=false`, because "false" is a non-empty string. Convert the values yourself, check the ones that are required when the app starts, and stop with a clear error if something is missing. Finding out at boot is much better than finding out in the middle of a payment.

There's no official standard

Nobody ever wrote the .env format down as a specification. Each library grew its own rules, and they disagree at the edges: multi-line values, `${VAR}` expansion, `export` at the start of a line, comments after a value, which file wins when two of them define the same name. When Node.js

added native support, the first version didn't handle multi-line values or variable expansion, both of which the npm package already did.

So keep your .env boring. Simple names, simple values, quotes only when needed, no tricks. A boring file reads the same in every tool.

How to use it

NODE.JS

On Node 20.6 or newer, no package is needed:

```
node --env-file=.env app.js
```

Or from inside the code:

```
process.loadEnvFile();
console.log(process.env.PORT);
```

On older versions, or if you want the extra features, use the package:

```
npm install dotenv
```

```
require('dotenv').config();
console.log(process.env.PORT);
```

PYTHON

```
pip install python-dotenv
```

```
import os
from dotenv import load_dotenv

load_dotenv()
port = int(os.getenv("PORT", "8000"))
```

RUBY

Add the gem to the Gemfile:

```
gem 'dotenv', groups: [:development, :test]
```

In Rails it loads by itself. In plain Ruby, put `require 'dotenv/load'` at the top of the app.

PHP

Laravel uses a .env file out of the box, through the phpdotenv library, and reads it with `env('DB_PASSWORD')` inside the config files. Outside Laravel, phpdotenv works on its own with Composer.

THE SHELL

If the file is simple enough for the shell to understand, bash can load it:

```
set -a
source .env
set +a
```

`set -a` marks every variable defined after it for export, so the programs you start from that terminal see them. Keep in mind that this runs the file as shell code, so only do it with a file you wrote yourself.

DOCKER COMPOSE

Here there are two things that look alike and do different jobs. The `.env` in the project folder is read by Compose itself, to fill in `${VARIABLES}` inside `compose.yaml`. The `env_file: .env` option, under a service, is what actually passes the variables into the container. Many "my container can't see the variable" problems come from mixing the two up.

Who wins when a variable exists twice

By default, a variable that already exists in the environment wins over the one in the file. The Ruby gem does this on purpose, on the idea that the deploy environment knows more about the configuration than the app does, and Node's native loader has worked the same way since 20.7.0. On a server this is a gift: the real values are set by the system, and a forgotten .env can't override them.

It's also the most common reason for "I changed the .env and nothing happened". Before anything else, check whether that variable is already exported in your shell with `echo $NAME`.

.env.example and .gitignore

The .env file itself almost never belongs in git. What goes in is a template, usually called `.env.example`, with every name the app needs and fake values or blanks:

```
DATABASE_URL=postgres://user:password@localhost:5432/myapp
STRIPE_SECRET_KEY=
DEBUG=false
PORT=3000
```

Whoever clones the project copies it and fills in their own values:

```
cp .env.example .env
```

And `.gitignore` gets these lines:

```
.env
.env.*
!.env.example
```

The last line is an exception, so the template still gets committed. Many frameworks also use a family of files, like `.env.development`, `.env.production` and `.env.local`, loaded in a fixed order. The `.local` ones are personal and stay out of git.

The front-end trap

Tools like Vite and Next.js read .env files too, with a twist. Variables with a special prefix, `VITE_` in Vite and `NEXT_PUBLIC_` in Next.js, are copied into the JavaScript that goes to the browser. That's how you hand a public API address to the front end. It also means that a secret key with that prefix is public, readable by anyone who opens the developer tools. If it has the prefix, treat it as something you'd print on a billboard.

How .env files leak

The first way is git. Someone commits the .env by mistake, notices, deletes it in the next commit and thinks it's over. The file is still in the history, and anyone who clones the repository can get it back. If a secret ever reached a remote repository, revoke it and create a new one. Cleaning the history comes after that and doesn't replace it.

The second way is the web server. If the project folder is also the folder the server publishes, a request to `https://yoursite.com/.env` may hand over the whole file, and bots make that request all day, on every site they can find. In 2024, researchers at Palo Alto Networks described an extortion campaign that started exactly like this: exposed .env files on at least 110,000 domains, with more than 90,000 unique variables collected, about 7,000 of them for cloud services.

The defense is to keep the .env out of the public folder (most frameworks already publish a `public/` folder one level below the project root, and the .env lives above it) and to tell the web server to refuse dotfiles. In nginx:

```
location ~ /\.(?!well-known) {
deny all;
}
```

The `well-known` exception keeps `/.well-known/`, which certificates and a few other standards need, working.

Environment variables are not a safe

A .env keeps secrets out of your code, and that's the whole job. Once loaded, the values sit in the process memory as plain text. Child processes inherit them. On Linux, a process running as the same user can usually read them from `/proc/<pid>/environ`. Error trackers and debug pages sometimes dump the entire environment into a report. Even the README of the Ruby gem says it was made for development, and that production usually has better ways to handle configuration.

On a server, prefer what the platform gives you: the variables panel of your host, systemd's `EnvironmentFile=` with the file readable only by the service's user, Docker secrets, or a proper secret manager once the project grows. And give each environment its own credentials, so a leaked development key opens nothing that matters.

A short checklist

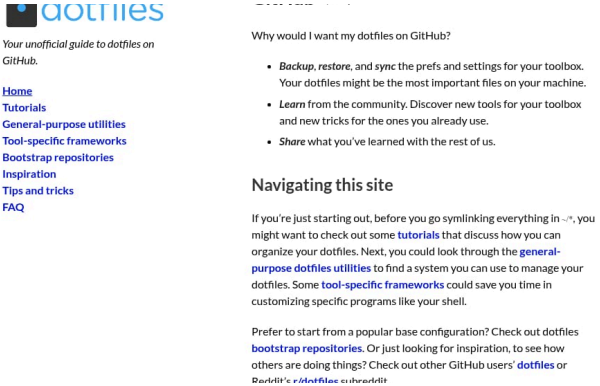
- `.env` in `.gitignore`, `.env.example` committed.
- The .env outside the public folder, and a web server that refuses dotfiles.
- Nothing secret with a `VITE_` or `NEXT_PUBLIC_` prefix.
- Values converted and checked when the app starts.
- A leaked secret gets revoked, not just deleted.
- Different credentials for development and production.

$ cd ~pablo

11 DOTFILES · LINUX · TERMINAL 18 NOVEMBER 2026

Dotfiles: the small text files that make a computer mine

A fresh computer works, but it isn't yours yet. What makes it yours fits in a few dozen text files. What dotfiles are, why they start with a dot, how to build and version your own, how I keep mine with chezmoi, and what they can and can't do.


dotfiles

Your unofficial guide to dotfiles on GitHub.

Home
Tutorials
General-purpose utilities
Tool-specific frameworks
Bootstrap repositories
Inspiration
Tips and tricks
FAQ

Why would I want my dotfiles on GitHub?

- ***Backup***, ***restore***, and ***sync*** the prefs and settings for your toolbox. Your dotfiles might be the most important files on your machine.
- ***Learn*** from the community. Discover new tools for your toolbox and new tricks for the ones you already use.
- ***Share*** what you've learned with the rest of us.

Navigating this site

If you're just starting out, before you go symlinking everything in ~/*, you might want to check out some **tutorials** that discuss how you can organize your dotfiles. Next, you could look through the **general-purpose dotfiles utilities** to find a system you can use to manage your dotfiles. Some **tool-specific frameworks** could save you time in customizing specific programs like your shell.

Prefer to start from a popular base configuration? Check out dotfiles **bootstrap repositories**. Or just looking for inspiration, to see how others are doing things? Check out other GitHub users' **dotfiles** or Reddit's **r/dotfiles** subreddit

A fresh computer works, but it isn't mine yet. The shell has the wrong prompt, git doesn't know my name, the editor opens with someone else's defaults, the terminal font is the wrong size and none of my shortcuts exist. What turns that machine into my machine fits in a few dozen small text files. They're called dotfiles, and they're one of the things I care about most in my whole setup. They live in a repository, and one command puts them on a new computer.

This post is about them: what they are, where the dot came from, how to start your own, how I keep mine, and what they can and can't do.

What dotfiles are

Dotfiles are the configuration files that programs read from your home folder. On Unix-like systems they traditionally start with a dot, which is where the name comes from:

- `~/.bashrc` and `~/.zshrc`, for the shell

- `~/.gitconfig`, for git
- `~/.vimrc`, for Vim
- `~/.ssh/config`, for SSH hosts and shortcuts
- `~/.tmux.conf`, for tmux

Newer programs keep their files in `~/.config/` instead: `~/.config/nvim/`, `~/.config/kitty/kitty.conf`, `~/.config/git/config`. Same idea, tidier place. People still call all of it dotfiles.

They're plain text, meant to be read and edited by a person. That's what makes them special. A setting you clicked in a menu is gone when the machine dies. A line in a text file can be copied, commented, versioned and carried anywhere.

Why they start with a dot

The dot started out as a shortcut.

In the early days of Unix, when the file system became hierarchical, every directory got two special entries: `.` for the directory itself and `..` for its parent. They showed up every time someone typed `ls`, so a check was added to hide them. Rob Pike told the story years later: the right test would have compared the name with `.` and `..`, but the one that went in was shorter and only looked at the first character.

```
if (name[0] == '.') continue;
```

Any file whose name started with a dot vanished from `ls`. Programmers noticed, and started putting their configuration files in everyone's home folder with a dot in front, so they'd stay out of sight. Pike called the hidden file "an unintended consequence" and "certainly a mistake", and pointed out that Plan 9, the system he worked on after Unix, had no dotfiles at all. Configuration lived in normal folders like `$HOME/lib`.

The other half of the name is older still. The `rc` in `.bashrc`, `.vimrc` and `.zshrc` comes from RUNCOM, a program written around 1963 for CTSS, a time-sharing system at MIT. It ran a list of commands stored in a file, and those files came to be called "runcoms". Kernighan and Ritchie described the `rc` in Unix as "a fossil from that usage". So every time you open `.bashrc` you're reading a word from the early sixties.

The home folder got crowded

Since any program could drop a dotfile in your home, many did. Pike counted about a hundred in his own home folder and didn't know what most of them were for.

The modern answer is the XDG Base Directory specification, from freedesktop.org. It defines where each kind of file should go:

- configuration in `~/.config` (or wherever `$XDG_CONFIG_HOME` points)
- data in `~/.local/share`
- state, like history and logs, in `~/.local/state`
- cache in `~/.cache`

Most recent tools follow it, and it helps a lot with dotfiles: everything worth saving sits under `~/.config`, and the cache, which you never want to save, sits somewhere else. Plenty of older programs still ignore it, which is why `~` is still full of dots.

Why they matter so much to me

Every line in a dotfile is a decision someone made once. An alias you got tired of typing, a git setting that stopped a mistake from happening again, a shortcut that matches how your hands move. Over the years those decisions pile up into an environment that fits you, and that environment is worth more than any single machine.

Keeping them in a repository gives you a few things at once:

- **A new machine in minutes.** Clone, run one command, and the computer behaves like yours.
- **The same environment everywhere.** Laptop, desktop, a server you SSH into, a VM you'll throw away tomorrow.
- **History.** Git tells you when a setting changed and why, as long as you write decent commit messages.
- **Courage to experiment.** If a new config breaks everything, `git checkout` brings the old one back.

There's a whole culture around this. People publish their dotfiles on GitHub, and dotfiles.github.io works as an unofficial guide, with tutorials, tools and other people's setups to learn from. GitHub even builds it into Codespaces: point it at your dotfiles repository and every new codespace clones it and runs your `install.sh` (or `bootstrap.sh`, `setup.sh` and a few other names). If there's no script, it links every file that starts with a dot into the home folder.

How I keep mine

My dotfiles live in a private repository managed with chezmoi. It covers two very different systems: Omarchy, the Arch Linux setup with Hyprland, and Windows. On a new Linux machine, restoring everything looks like this:

```
chezmoi init --apply runawaydevil/dotfiles
~/.local/share/chezmoi/meta/restore.sh
hyprctl reload
```

The first line clones the repository and writes every file to its place. The second installs packages, themes, plugins and services from lists kept in the repository, in steps I can also run one at a time. On Windows the same first line works, and a PowerShell script does the equivalent with winget, Scoop and Chocolatey.

A file called `.chezmoiignore` decides what goes where. Hyprland, the terminals (Alacritty, foot, Ghostty and kitty), Neovim, Zed, mise and the small scripts in `~/.local/bin` only exist on Linux. The Windows Terminal settings and the PowerShell profiles only exist on Windows. The git config is shared, as a template with the few lines that change per system:

```
{{ if eq .chezmoi.os "windows" }}
# lines only Windows gets
{{ end }}
```

Saving changes is a snapshot script and a normal git commit. Even the wallpapers are in there, a folder of old seascapes by Aivazovsky, Hokusai, Turner and Winslow Homer, so a new machine looks like home from the first boot.

What stays out matters as much. Themes and plugins I download from elsewhere go in as a list of URLs, and binaries that come from packages are installed by the restore script. The repository stores my decisions and leaves out whatever can be downloaded again.

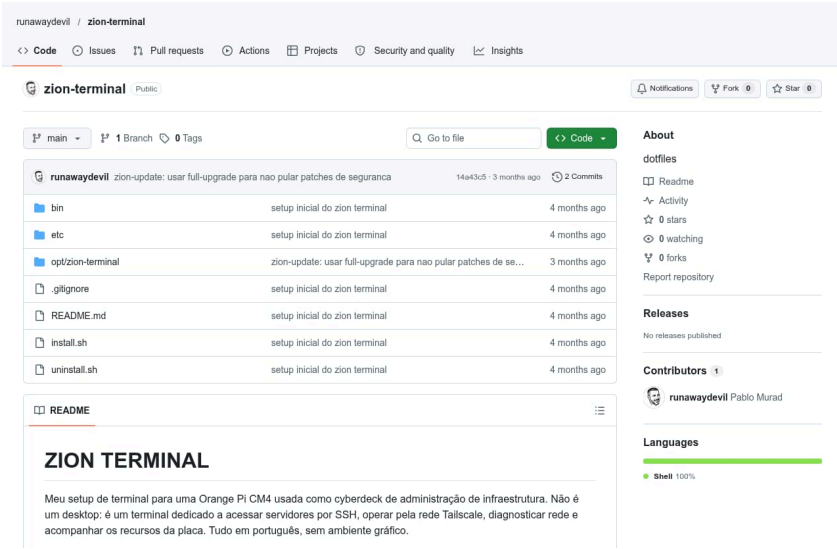
I have a second, smaller example that's public: Zion Terminal, the terminal setup for an Orange Pi CM4 I use as a cyberdeck. It started as dotfiles and grew into a small tool, with an `install.sh` that can run twice without breaking anything and an `uninstall.sh` with a `--dry-run` flag.

How to start yours

STEP 1: FIND WHAT YOU'VE ACTUALLY CHANGED

Don't start by copying someone's famous repository. Start with your own machine and list the files you've edited by hand. For most people the first batch is short:

- the shell: `~/.bashrc` or `~/.zshrc`
- git: `~/.gitconfig`
- the editor: `~/.vimrc`, `~/.config/nvim/` or your editor's settings file
- the terminal: something in `~/.config/`



Three or four files are a perfectly good first commit.

STEP 2: PICK A WAY TO MANAGE THEM

There are four common approaches, from the simplest to the most capable.

A folder and symbolic links. Move the real file into a folder that's a git repository, and leave a link in its old place:

```
mkdir ~/dotfiles
cd ~/dotfiles
git init
mv ~/.bashrc ~/dotfiles/bashrc
ln -s ~/dotfiles/bashrc ~/.bashrc
```

The program still finds `~/.bashrc`, which now points into the repository. It works anywhere and needs nothing installed, but on a new machine you have to recreate every link yourself, so sooner or later you write a script for it.

GNU Stow. Stow describes itself as "a symlink farm manager", and it makes the links for you. You arrange the repository in one folder per program, mirroring where the files live in your home:

```
~/dotfiles/bash/.bashrc
~/dotfiles/git/.gitconfig
~/dotfiles/nvim/.config/nvim/init.lua
```

Then, from inside `~/dotfiles`:

```
stow bash git nvim
```

Stow links each file into your home folder in the right place. `stow -D nvim` removes the links again. It's small, old and dependable, and it's a good choice if you only use one system.

A bare git repository. This trick became popular through a Hacker News comment in 2016. Your home folder itself becomes the working tree, with the repository's internals tucked into a hidden folder, and an alias keeps it from interfering with your other projects:

```
git init --bare $HOME/.cfg
alias config='/usr/bin/git --git-dir=$HOME/.cfg/ --work-tree=$HOME'
config config --local status.showUntrackedFiles no
config add ~/.bashrc
config commit -m "Add bashrc"
```

No links and no extra tools. The catch is that you have to remember to use `config` instead of `git`, and a careless `config add .` would try to add your whole home folder.

chezmoi. A dedicated tool, and the one I use. It keeps a copy of your files in its own repository and writes them out when you run `chezmoi apply`. It handles templates for differences between machines, can read secrets from password managers, can encrypt whole files with age or gpg and can run scripts, which covers installing packages too. The basic loop:

```
chezmoi init
chezmoi add ~/.bashrc
chezmoi edit ~/.bashrc
chezmoi apply
chezmoi cd
git add -A
git commit -m "Add bashrc"
git push
```

On a new machine, `chezmoi init --apply your-github-user` does the rest.

If you use one system and a handful of files, Stow is enough. If you move between Linux, macOS and Windows, or have files that must differ per machine, chezmoi pays for itself quickly.

STEP 3: WRITE THE RESTORE SCRIPT

Whatever you pick, put an `install.sh` at the root of the repository that gets a clean machine to your environment: install the packages you need, link or apply the files, set the shell. Make it safe to run more than once. If

running it twice breaks something, you'll be afraid to run it at all, and a script you're afraid to run doesn't help.

STEP 4: KEEP THE HABIT

Changed a setting? Commit it the same day. Dotfiles rot the same way notes do: the improvement that never made it into the repository is lost on the next machine.

Limitations

Secrets. This is the big one. SSH keys, API tokens, `.netrc`, cloud credentials and shell history do not go in the repository, private or not. A private repository is still a copy of your secrets on someone else's server, and one wrong click makes it public. Keep secrets in a password manager or encrypted, and let the tool pull them in at apply time. Before your first push, read `git log -p` and look for anything that looks like a key.

Machines are different. Paths, package managers, screen sizes and installed tools change from one computer to the next, and a config that's perfect on the laptop can error out on a server. Templates and per-system ignore lists fix most of it, at the cost of some complexity.

Not everything is a text file. Some programs keep settings in a database, the Windows registry or macOS property lists, and some rewrite their own config file every time you change something in the interface, so every commit comes with noise you didn't write.

Other people's dotfiles. Copying a famous setup gives you hundreds of lines you don't understand, and some popular repositories install themselves with `curl ... | bash`, which means running a stranger's script on your machine without reading it. Read first, take what you understand, leave the rest.

The rabbit hole. Tweaking dotfiles is fun, and it can eat whole evenings that were supposed to go to the work the setup exists for.

And what they can do

The other side is that almost anything you can write down fits in a dotfiles repository. Mine has shell and editor configs, and also systemd user services, small scripts, lists of packages to install, the window manager layout and the wallpapers. Anything that's text, or that can be fetched again from a list, can come along.

It can also grow well past configuration. Zion Terminal turned into a little command-line tool. Codespaces turns your repository into a setup that follows you into the cloud. Omarchy itself is, in my reading, a very

well-organized set of dotfiles and install scripts that turns a plain Arch install into a complete desktop. Nix with home-manager goes further still and describes your entire user environment, programs included, in files.

The limit is your patience and your discipline with secrets. Everything else can be versioned. ■

■ ■ ■

12 CHESSKING · CHESS · FEDIVERSE 18 NOVEMBER 2026

Chess over the protocols

My first programs were tiny chess games, and I never really stopped making them. The next one plays over the protocols: a bot on both the fediverse and Bluesky that keeps the board, so people on either network can play each other. It's called chessk.ing, and it isn't open yet.



My first experiments with programming were tiny chess games. That was more than twenty years ago: a board on the screen, pieces that moved when you clicked them, and rules I kept getting wrong. Castling, en passant, the stalemate I forgot existed. Then came chess engines, code that looks at a position and tries to pick a move. Most of mine were bad. I loved every one of them.

I never really stopped. Whenever there's a free weekend and an idea, a chessboard shows up somewhere. Last year alone I think I built three of these experiments. They don't have to be useful. Writing a move generator is a bit like doing a crossword you invented yourself.

So when I started thinking about the next one, I already knew I didn't want to make just another chess game. There are millions of those, and some of them are wonderful. I wanted to answer a different question.

What if you could play over the protocols?

I spend a lot of my time on the fediverse and on Bluesky. Both are open networks, built on protocols instead of on one company's servers: ActivityPub on one side, the AT Protocol on the other. Anyone can write software that speaks them.

Chess fits that kind of place perfectly. A move is a short piece of text, like `e4`, `Nf3` or `O-O`. You could play a whole game in replies, the way people used to play by mail, one move a day, in between other conversations.

That left me with a few questions I kept turning over:

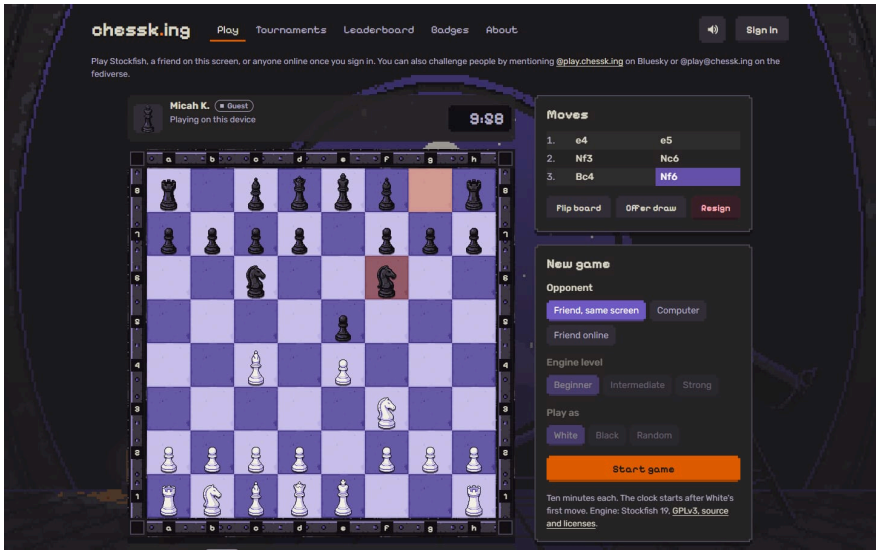
- How would someone on Mastodon or GoToSocial play against someone on Bluesky, when the two networks don't talk to each other?
- Who keeps the board? Who checks that a move is legal, and who says whose turn it is?
- What does a game look like in a timeline, without turning your followers' feeds into a chess broadcast?

A bot in the middle

Where I landed is a bot that lives on both networks at once. On the fediverse it's an ActivityPub account, and on Bluesky it's a regular account too. Underneath, it's the same server holding the same games.

You mention the bot and challenge someone, and they accept by replying. From then on every move is a reply in the thread. The bot answers with the new position as an image and tells the other player it's their turn. When the two players are on different networks, the bot carries each move across, so one of you can be on the fediverse and the other on Bluesky, and the game doesn't care.

The bot writes the moves down, checks them, keeps the clock for correspondence games and knows when the game is over. You just play. On Bluesky it answers to `@play.chessk.ing`, and on the fediverse to `@play@chessk.ing`.



A project that kept getting parked

None of this is new in my head. The idea has been around for a long time, and it kept getting parked, one weekend project losing to the next one, or to the work that pays the bills.

This time it moved, and much of that is thanks to the artist Gabriela. With her on board, chessk.ing stopped being a technical experiment with a board in it and started to have a face and a world of its own.

Something of my own

I could have stopped at the bot. But I wanted something that felt like mine, not a generic board with a generic font.

So it grew into a small world. There's a website where you can play live games with a clock, or play against Stockfish at a few levels. There are tournaments, and a monthly leaderboard that also keeps score of the friendly rivalry between the fediverse and Bluesky. There are badges for the silly and the serious things you do on the board, and stars you earn by playing, which you can spend on the look of your board. The themes are pixel art: an old observatory with a telescope and a moon in the window, a galaxy, a Van Gogh room. Every piece set was drawn for it.

Signing in uses the accounts you already have, on Bluesky or on your own fediverse server. If you want, your finished games can also be copied



into your own Bluesky repository, as records you own, and not only as rows in my database.

Privacy was part of the design from the start. Public games are public, discreet games stay out of feeds, and private games are encrypted on the server. You can also keep yourself off the leaderboard entirely.

It's called chessk.ing

That's the name, and it isn't open yet.

We've already run some test games, and so far everything has been promising. The site stays in "coming soon" mode while I play the first games myself and fix whatever breaks. If everything goes well, maybe it opens in February 2027. I honestly don't know yet, and I'd rather open it late than open it broken.

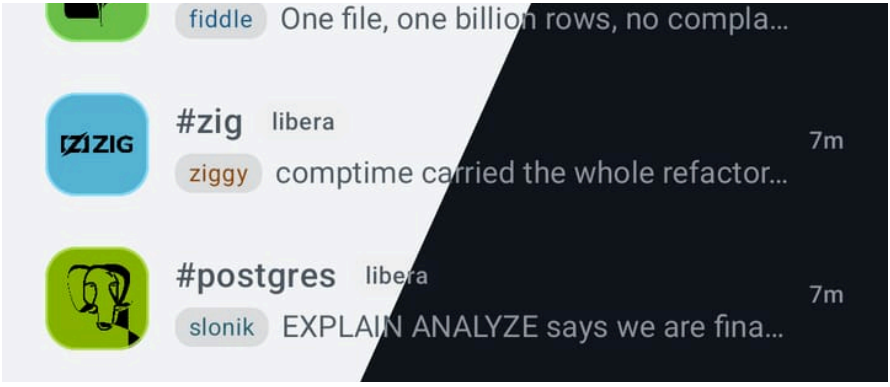
If you stop by chessk.ing today, you'll find a knight waiting next to a chess clock, under the moon. If you mention the bot, it'll tell you the same thing: soon.

I'll write again when the board is open. Until then, I'll keep doing what I've done for twenty years: moving pieces around and seeing what happens. ■

13 IRC · SOJU · ANDROID 18 NOVEMBER 2026

Note II: motd, an IRC client that gets along with soju

I was looking for Android IRC clients that work well with soju and came across motd, a native, Google-free client on F-Droid with bouncer history, synced read markers and UnifiedPush. I'm liking it so far. Not affiliated, not sponsored.

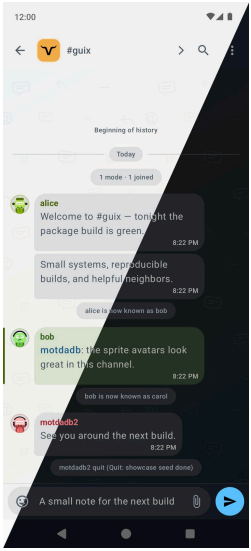


I was looking for IRC clients for Android that work well with soju, the bouncer that stays connected to IRC for you and keeps the history while your phone is asleep. Along the way I came across motd, on F-Droid, and I'm liking it.

It's a native Android IRC client with a modern chat interface. You can connect straight to an IRC network, but the part that caught my eye is how it handles soju: it pulls the history from the bouncer, keeps read markers in sync between devices and lets you manage soju's networks from inside the app. ZNC playback works too.

A few other things worth knowing:

- Local history with full-text search, across all chats or inside one, with a jump to the message.
- Replies, reactions, typing indicators, nick completion and slash commands, when the server supports them.
- Inline images, link previews and file uploads.



- TLS, SASL PLAIN and EXTERNAL, and client certificates.
- Optional routing per network through SOCKS5 or Tor with Orbot.
- No Google inside. Push notifications go through UnifiedPush instead of Firebase, which needs a compatible server or bouncer and a UnifiedPush distributor on the phone. A persistent connection is also an option.

It's free software under the GPL v3 or later, the source is on GitHub and the developer ships updates often. It needs Android 8.0 or newer and a 64-bit ARM phone.

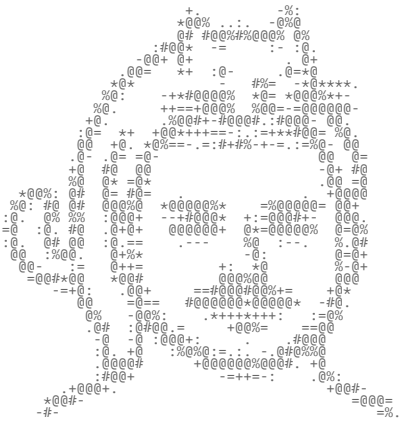
For the record: I'm not affiliated with the project and nobody sponsored this note. I just found a client that plays well with soju and wanted to share it. ■



~pablo

A zine made of what I wrote on pablomurad.com.

00



COLOPHON

ISSUE	N° 00, special edition, November 2026
PUBLISHED	18 November 2026, on pablomurad.com
TYPESET	18 November 2026, A5
ESSAYS	13, each one also on the website
PLATES	Pictures between the essays are courtesy of noctem.art.
AT HOME	Print pablo-00-booklet.pdf double-sided on A4, flip on the short edge, fold the stack in half and staple the spine.

pablomurad.com/zine/00

~PABLO

Nº 00

Essays from pablomurad.com, set on paper.

